Computational Complexity; slides 1, HT 2023
Introduction, motivation, background, Turing machines, deterministic complexity classes

Paul W. Goldberg (Dept. of CS, Oxford)

HT 2023

Administrative notes

www.cs.ox.ac.uk/people/paul.goldberg/CC/index.html

- Slides, exercise sheets, often updated

www.cs.ox.ac.uk/teaching/courses/2022-2023/complexity/

- General info

Problem sheets:	classes in weeks 3 – 8
	Available on web page by Monday of previous week
	check hand-in deadlines

Aims

(from the web page)

- Introduce the most important complexity classes
- Give you tools to classify problems into appropriate complexity classes
- Enable you to reduce one problem to another

Aims

(from the web page)

- Introduce the most important complexity classes
- Give you tools to classify problems into appropriate complexity classes
- Enable you to reduce one problem to another

Above terminology to be made precise

We will see there are major gaps in our understanding of computation!

here, mostly focus on time/space requirements; there is also "communication complexity", "query complexity", ...

note usage of word "complexity"

Background, other courses

algorithms, big-O notation, “problem”, TM, propositional logic.
See e.g. chapter 7.1 in Sipser’s textbook

- Models of Computation (part A)
  - Introduce Turing machines as a universal computing device
  - Classification of problems into decidable/undecidable
  - further classification of undecidable problems
- Intro to Formal Proof (prelims)
  - SAT, CNF, etc
- Algorithms (part A)
  - address “intractability” studied here
- Design and Analysis of Algorithms (prelims)
  - design of efficient algorithms
  - asymptotic runtime analysis

Polynomial-time computation, the class **P**

problems solvable in time $O(n)$, $O(n \log n)$, $O(n^{10})$, ...

Given a novel problem, usual Q1: is it in **P**?

Why do we like this concept?

- nice closure/composition properties
  composition of 2 poly-time algorithms is poly-time
- **P** works surprisingly well as a model of "efficiently computable", "fast algorithm"
  If a problem is solvable in time $O(n^{100})$, usually it has a genuinely efficient algorithm
- We can ignore details of model of computation, representation of input; "clean" analysis
- poly-time algorithms highlight structure of a problem they solve (quote on next slide)

"poly-time" not just about computational efficiency

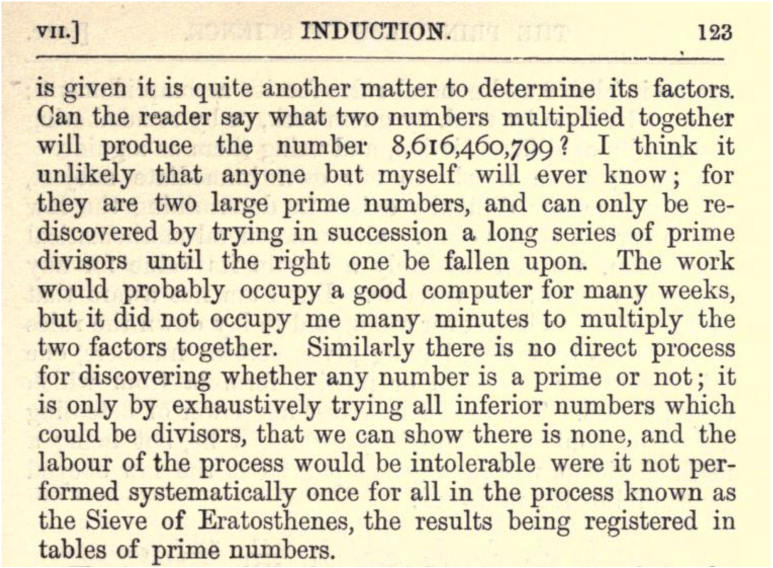
Poly-time algorithm tells you about the structure of a problem.
Contrast with "brute-force" algorithm

A relevant quote (context: looking for "equilibrium prices" in markets)

What do we learn by proving that an equilibrium computation problem is "difficult" in a complexity-theoretic sense? First, assuming widely believed mathematical conjectures, it implies that there will never be a fast, general-purpose computational method for solving the problem. Second, it rules out many structural results, such as convexity or duality, that are often used to understand and justify economic models.

Tim Roughgarden: Computing equilibria: a computational complexity perspective *Economic Theory* (2010)

Some problems don't seem to have efficient algorithms



VII.] INDUCTION. 123

is given it is quite another matter to determine its factors. Can the reader say what two numbers multiplied together will produce the number 8,616,460,799? I think it unlikely that anyone but myself will ever know; for they are two large prime numbers, and can only be rediscovered by trying in succession a long series of prime divisors until the right one be fallen upon. The work would probably occupy a good computer for many weeks, but it did not occupy me many minutes to multiply the two factors together. Similarly there is no direct process for discovering whether any number is a prime or not; it is only by exhaustively trying all inferior numbers which could be divisors, that we can show there is none, and the labour of the process would be intolerable were it not performed systematically once for all in the process known as the Sieve of Eratosthenes, the results being registered in tables of prime numbers.

Road map (roughly)

1. **[2 lectures]** introduction, Turing machines, (un)decidability, reductions

move swiftly from qualitative to quantitative considerations:

2. **[1 lecture] Deterministic Complexity Classes**. DTIME[t]. Linear Speed-up Theorem. PTime. Polynomial reducibility.
3. **[3 lectures] NP, co-NP, (co-)NP-completeness**. Non-deterministic Turing machines. NTIME[t]. Polynomial time verification. NP-completeness. Cook-Levin Theorem.
4. **[3 lectures] Space complexity and hierarchy theorems**. DSPACE[s]. Linear Space Compression Theorem. PSPACE, NPSPACE. PSPACE = NPSPACE. PSPACE-completeness. Quantified Boolean Formula problem is PSPACE-complete. L, NL and NL-completeness. NL = coNL. Hierarchy theorems.

Road map

5. **[2 lectures] Randomized Complexity**. The classes BPP, RP, ZPP. Interactive proof systems: IP = PSPACE.
6. **Advanced topics.** Randomised complexity, Circuit complexity, total search

Reading List

Primary:

- S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press
- M. Sipser, *Introduction to the Theory of Computation*, 2005

Further:

- C.H. Papadimitriou, *Computational Complexity*, 1994.
- I. Wegener, *Complexity Theory*, Springer, 2005.
- O. Goldreich, *Complexity Theory*, CUP, 2008.
- M.R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1979.
- T.H. Cormen, S. Clifford, C.E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, 2001.

Classifying problems according to membership of **P**

SHORTEST PATH

Given a weighted graph and two vertices s, t, find a *shortest path* between s and t.

Can be solved efficiently (for instance with Dijkstra's algorithm)

LONGEST PATH

Given a weighted graph and two vertices s, t, find a *longest simple (cycle-free) path* between s and t.

LENGTH CONSTRAINED DISJOINT PATHS

Given a graph, two vertices s, t and $c, k \in \mathbb{N}$, find k *disjoint paths* between s and t of length $\leq c$.

No efficient solution known (and conjectured not to exist)

Problem: solving polynomial equations over integers

DIOPHANTINE EQUATIONS

Given a system of *Diophantine equations*, check whether it has an integer solution.

Example:

$$\begin{aligned} xyz - y^3 + z^2 &= 2 \\ y - 3z &= 5 \end{aligned}$$

Undecidable — no algorithmic solution!

https://en.wikipedia.org/wiki/Diophantine_equation
https://en.wikipedia.org/wiki/Hilbert's_tenth_problem

A graph problem: CLIQUE

A *clique* in a graph G is a complete subgraph of G.

MAX CLIQUE

Input: Graph G

Find: largest clique $C \subseteq G$

That's an search problem. Corresponding decision problem would specify a number k and ask for a "k-clique".

`https://en.wikipedia.org/wiki/Clique_problem`

...another notoriously hard problem; although has an obvious brute-force (exponential time) algorithm.

A graph problem: CLIQUE

Can search for a k-clique of an n-vertex graph in time $O(n^k.k^2)$, poly(n) if k is constant.

Follow-up question:
Can we search for a k-clique in time $f(k).p(n)$, where $p(\cdot)$ is a polynomial?
Unlikely — CLIQUE is "fixed parameter intractable".

General observations

- Why are some problems so much harder to solve than other – seemingly very similar – problems?
- Are they really harder to solve?

  Or have we just not found the right method to do so?

General observations

- Why are some problems so much harder to solve than other – seemingly very similar – problems?
- Are they really harder to solve?

  Or have we just not found the right method to do so?

Computational Complexity: classify problems according to the amount of resources (runtime, space, communication, etc) needed

Relies on various mathematical conjectures of which the most famous is the "P$\neq$NP" belief. Others include the "exponential time hypothesis", used to prove that k-CLIQUE cannot be solved in time $n^{o(k)}$.

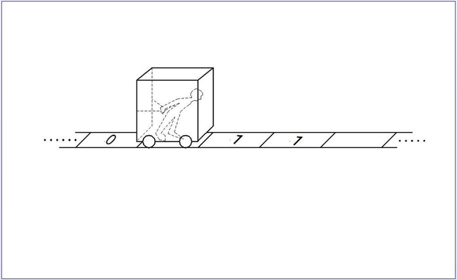
Lower bounds on runtime requirements are hard to show! Needs details of model of computation. More progress is often possible for lower bounds on *query complexity* and *communication complexity* of various problems.

Turing machines

Alan Turing considered qn. of “What is computation?” in 1936.

He argued, that *any* computation can be done using the following steps (writing on a sheet of paper):

- Concentrate on one part of the problem (one symbol on the paper)
- Depending on what you read there
    - Change into a new state (remember a finite amount of information)
    - Modify this part of the problem
    - Move to another part of the input
- Repeat until finished



Next: detailed definition, notation, basic results

Key points

Why we care about TMs:

- precise notion of "runtime", "memory usage"
- well-defined operations on algorithms (when represented as TMs) — (operations such as pass output of Alg 1 to Alg 2, etc)
- variants of TM (e.g. NTM) define important classes of problems

Sometimes we'll use pseudocode but with understanding that there's an equivalent TM

Deterministic Turing Machines

Definition. (one of many variants, all "equivalent")
A (deterministic) *k*-tape *Turing machine* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is *input alphabet* – a finite alphabet of *symbols*
- $\Gamma \supseteq \Sigma \cup \{\Box\}$ is *working tape alphabet* (finite)
- δ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is a set of final states (each of which either accepts or rejects)

Deterministic Turing Machines

Definition. (one of many variants, all "equivalent")
A (deterministic) k-tape *Turing machine* is a 6-tuple
$(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is *input alphabet* – a finite alphabet of *symbols*
- $\Gamma \supseteq \Sigma \cup \{\square\}$ is *working tape alphabet* (finite)
- δ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is a set of final states (each of which either accepts or rejects)

Tape. Infinite tape, bounded to the left.

Each cell contains one symbol from Γ ($\square$: special "blank" symbol)

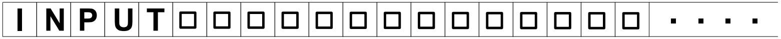
Deterministic Turing Machines

Definition. (one of many variants, all “equivalent”)
A (deterministic) *k*-tape *Turing machine* is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$ where

- Q is a finite set of *states*
- Σ is *input alphabet* – a finite alphabet of *symbols*
- $\Gamma \supseteq \Sigma \cup \{\Box\}$ is *working tape alphabet* (finite)
- δ is the *transition function*
- $q_0 \in Q$ is the *initial state*
- $F \subseteq Q$ is a set of final states (each of which either accepts or rejects)

Tape. Infinite tape, bounded to the left.

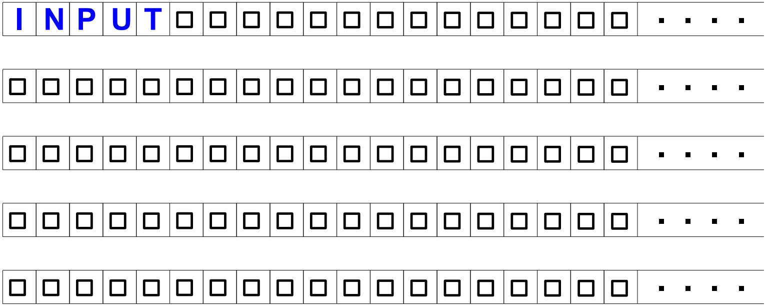
Each cell contains one symbol from Γ ($\Box$: special “blank” symbol)



Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, 1\}^k$

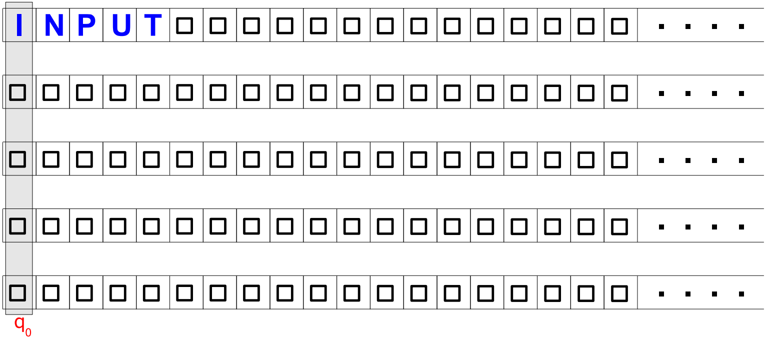
(-1: “left” 0: “stay put” 1: “right”)



Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{-1, 0, 1\}^k$

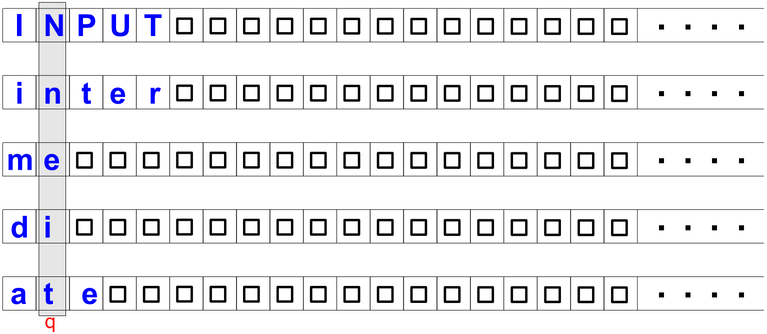
(-1: "left" 0: "stay put" 1: "right")



Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, 1\}^k$

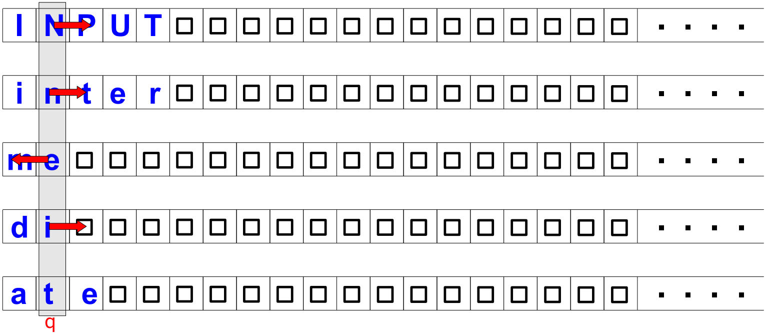
(-1: "left" 0: "stay put" 1: "right")



Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times \{-1, 0, 1\}^k$

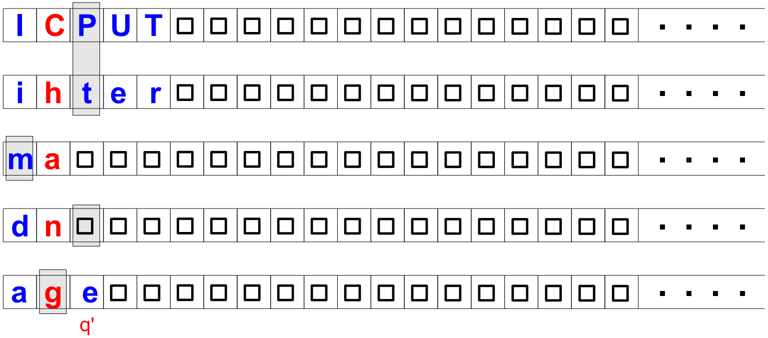
(-1: "left" 0: "stay put" 1: "right")



Deterministic TM (multiple tape version)

Transition function: $\delta : (Q \setminus F) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{-1, 0, 1\}^k$

(-1: "left" 0: "stay put" 1: "right")



Turing Machine operation

1. At each step of operation the machine is in one state $q \in Q$
2. Initially:
   - Machine is in state $q_0 \in Q$
   - the input is contained on tape 1
   - all other tape symbols are $\square$
3. The machine is reading one symbol on each tape: $s_1 \dots s_k$
4. To execute one step, the machine looks up

$$\delta(q, s_1, \dots, s_k) := \big(q', (s'_1, \dots, s'_k), (m_1, \dots, m_k)\big)$$

5. The machine:
   - changes to state q'
   - replaces each s_i by s'_i
   - moves the heads on the individual tapes according to m_i ($1 =$ move right, $-1 =$ move left, $0 =$ stay)
   - Execution stops when a final state is reached.
   - In this case, the content of the last tape k contains the output.

more general points

(I assume you've seen examples of TMs already)

- TM: general-purpose notion of "algorithm", "computational procedure"
- equivalence of alternative defs of TM assure us of above; simulations have "polynomial overhead"
- Algorithm pseudocode is readable, usually we use it to describe algorithms, tacit assumption: can be converted to TM
- TMs $\rightsquigarrow$ precise notion of runtime/space. Used in various theorems in this course.

Configurations (definition, notation)

For $M := (Q, \Sigma, \Gamma, \delta, q_0, F)$, what's going on is described by

- the current state
- the contents of all tapes
- the position of all its heads

$\big(q,\ (w_1, \dots, w_k),\ (p_1, \dots, p_k)\big)$ where $q \in Q, w_i \in \Gamma^*,\ p_i \in \mathbb{N}$

where Γ^* denotes words over alphabet Γ

Start configuration on input w: $\big(q_0, (w, \varepsilon, \dots, \varepsilon), (0, \dots, 0)\big)$

where ε denotes empty word

Stop (or, halt) configuration:
Configuration $\big(q, (w_1, \dots, w_k), (p_1, \dots, p_k)\big)$ such that $q \in F$.

Computation

Notation:

- $C \vdash_M C'$ if M can change from configuration C to C' in one step.
- $C \vdash_M^* C'$ if M can change from configuration C to C' in arbitrarily many steps.

Computation

Notation:

- $C \vdash_M C'$ if M can change from configuration C to C' in one step.
- $C \vdash_M^* C'$ if M can change from configuration C to C' in arbitrarily many steps.

The *computation* of a TM M on input $w \in \Sigma^*$ is either

- an infinite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \dots$ of configurations, or
- a finite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \cdots \vdash_M C_n$.

  In the latter case we say that M *halts* on input w.

  Notation: $T_M(w) := n$ number of steps upon input w.

C_n: stop configuration C_0: start config of M on input w.

Computation

Notation:

- $C \vdash_M C'$ if M can change from configuration C to C' in one step.
- $C \vdash_M^* C'$ if M can change from configuration C to C' in arbitrarily many steps.

The *computation* of a TM M on input $w \in \Sigma^*$ is either

- an infinite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \dots$ of configurations, or
- a finite sequence $C_0 \vdash_M C_1 \vdash_M C_2 \cdots \vdash_M C_n$.

  In the latter case we say that M *halts* on input w.

  Notation: $T_M(w) := n$ number of steps upon input w.

C_n: stop configuration C_0: start config of M on input w.

A TM *halts on input* w (and generates output o) if the computation of M on w terminates in configuration

$$(q, (w_1, \dots, w_{k-1}, o), (p_1, \dots, p_k)) \quad \text{with} \quad q \in F.$$

Run-time of a TM

Let M be a Turing machine with alphabet Σ
$f : \Sigma^* \to \Sigma^*$
$g : \mathbb{N} \to \mathbb{N}$

M computes f in time $g(n)$ if for every $w \in \Sigma^*$ M halts on input w after at most $g(|w|)$ steps with $f(w)$ on its output (last) tape.

(i.e. $T_M(w) \leq g(|w|)$)

Example (TM as transducer)

The following 2-tape Turing machine

$$M := \left(\{q_0, q_1, q_f\}, \{a, b\}, \{a, b, \Box\}, \delta, q_0, \{q_f\}\right)$$

where

$$\delta := \left\{ \begin{array}{l} \left(q_0, \binom{a}{-}, \binom{a}{-}, \binom{1}{0}, q_0\right) \\ \left(q_0, \binom{b}{-}, \binom{b}{-}, \binom{1}{0}, q_0\right) \\ \left(q_0, \binom{\Box}{-}, \binom{\Box}{-}, \binom{-1}{0}, q_1\right) \\ \left(q_1, \binom{a}{-}, \binom{\Box}{a}, \binom{-1}{1}, q_1\right) \\ \left(q_1, \binom{b}{-}, \binom{\Box}{b}, \binom{-1}{1}, q_1\right) \\ \left(q_1, \binom{\Box}{-}, \binom{\Box}{-}, \binom{0}{0}, q_f\right) \end{array} \right\}$$

computes the *reverse*-function $\textit{reverse}(a_1 \ldots a_n) := a_n \ldots a_1$ in

time $g(n) = 2n + 2 = \mathcal{O}(n)$.

For various alternative definitions of TM, including changes to alphabet, runtimes needed are polynomially related.

Decision problems as languages; Turing acceptors

TM M solves a decision problem if the language accepted by M (M as a *language acceptor*) is the yes-instances of the decision problem.

For decision problem D, $\mathcal{L}(D)$ denotes the *yes-instances* of D (needs an agreed-on encoding).

Search problems can generally be reduced to decision problems...

Example: Travelling Salesperson Problem (TSP)

Given pairwise distances between cities, we ask for the shortest tour, or the length of the shortest tour

Decision version: given the pairwise distances and a number k, is there a tour of length at most k?

Recall: decidable languages

Definition/notation

The language $\mathcal{L}(M) \subseteq \Sigma^*$ *accepted* by a Turing acceptor $M := \left(Q, \Sigma, \Gamma, \delta, q_0, F\right)$ is defined as

$$\{w \in \Sigma^* : M \text{ accepts } w\}.$$

(Note that we do not require M to halt on rejected inputs.)

A language $\mathcal{L} \subseteq \Sigma^*$ is *recursively enumerable*, if there is an acceptor M such that $\mathcal{L} = \mathcal{L}(M)$.

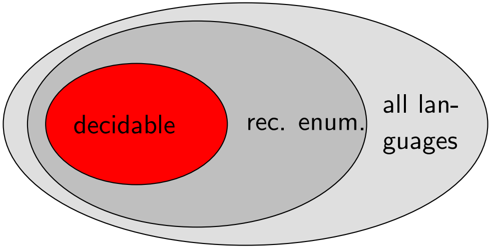
A language $\mathcal{L} \subseteq \Sigma^*$ is *decidable* (or, "recursive") if there is an acceptor M such that for all $w \in \Sigma^*$:

$$\begin{array}{lcl} w \in \mathcal{L} & \Longrightarrow & M \text{ halts on input } w \text{ in an accepting state} \\ w \notin \mathcal{L} & \Longrightarrow & M \text{ halts on input } w \text{ in a rejecting state} \end{array}$$

Decidable and Enumerable Languages

Recall:

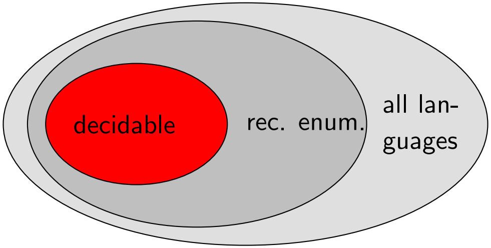
1. If a language $\mathcal{L}$ is *decidable* then it is *recursively enumerable*
2. If $\mathcal{L}$ and $\Sigma^* \setminus \mathcal{L}$ are *recursively enumerable* then $\mathcal{L}$ is decidable.



Decidable and Enumerable Languages

Recall:

1. If a language $\mathcal{L}$ is *decidable* then it is *recursively enumerable*
2. If $\mathcal{L}$ and $\Sigma^* \setminus \mathcal{L}$ are *recursively enumerable* then $\mathcal{L}$ is decidable.



Note: recursively enumerable a.k.a. *semi-decidable*, *partially decidable*

Problems as languages

Main points:

- decision problems viewed as language recognition problems
  We can use "decision problem" and "language" interchangeably
- We're allowed to be vague about encoding of problems (e.g. CLIQUE, TSP) — we will see that details of encoding don't affect the problem classifications of interest. Details of alphabet also unimportant (but *unary* alphabet is too big a restriction!). ("standard encoding", should be sensible.)

Undecidable Languages

Aim of this section

- Recursion theory — a brief reminder
- 2 techniques: *diagonalisation* and *reductions* — variants appear in complexity-theory classification of problems

A counting argument (sketch):

- The number of Turing machines is infinite but *countable*
- The number of different languages is infinite but *uncountable*; diagonalisation
- Therefore, there are "more" languages than Turing machines

It follows that there are languages that are not decidable.
Indeed some aren't even semi-decidable.

The Halting Problem

previous argument shows that there are undecidable languages.

Can we find a concrete example?

Halting problem (HALT)

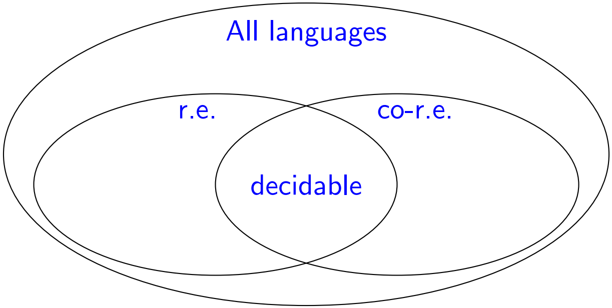
Input: A Turing machine M and an input string w
Question: Does M halt on w?

Again, undecidability of HALT is proved by diagonalisation:
consider effective listing of TMs, new TM that differs from all in listing
details in e.g. Sipser Chapter 4.2

Classification of Languages

Definition. A language $\mathcal{L} \subseteq \Sigma^*$ is *co-recursively enumerable*, or *co-r.e.*, if $\Sigma^* \setminus \mathcal{L}$ is recursively enumerable.

Example: $\mathcal{L}(\overline{\mathsf{HALT}})$ is co-r.e (but not r.e.).



Looking ahead, relationship between NP and co-NP is more complicated...

Reductions

A major tool in analysing and classifying problems is the idea of "reducing one problem to another"

As you expect — or have already seen — use undecidability of HALT to prove undecidability of variants, e.g. TM acceptance problem.

- Informally, a problem $\mathcal{A}$ is *reducible* to a problem $\mathcal{B}$ if we can use methods to solve $\mathcal{B}$ in order to solve $\mathcal{A}$.
- We want to capture the idea, that $\mathcal{A}$ is "no harder" than $\mathcal{B}$.

  (as we can use $\mathcal{B}$ to solve $\mathcal{A}$.)

Turing Reductions

Informally, problem $\mathcal{A}$ is *Turing reducible* to $\mathcal{B}$ if we can solve $\mathcal{A}$ using a program solving $\mathcal{B}$ as sub-program.

We write $\mathcal{A} \leq_T \mathcal{B}$.

Example: $\overline{\mathsf{HALT}}$ is Turing reducible to HALT.

take a Turing acceptor accepting HALT as sub-program and reverse its output

Turing Reductions

Informally, problem $\mathcal{A}$ is *Turing reducible* to $\mathcal{B}$ if we can solve $\mathcal{A}$ using a program solving $\mathcal{B}$ as sub-program.

We write $\mathcal{A} \leq_T \mathcal{B}$.

Example: $\overline{\text{HALT}}$ is Turing reducible to HALT.

take a Turing acceptor accepting HALT as sub-program
and reverse its output

Turing reductions are free/unrestricted; sometimes too much so for our purposes.

$\rightsquigarrow$ **Many-One Reductions** (Sipser: "mapping reduction") *are more informative*: $\mathcal{A} \leq_T \mathcal{B}$ relates (un)decidability of problems; use $\mathcal{A} \leq_m \mathcal{B}$ (next slide) to find out if a problem (or its complement) is recursively enumerable.

Many-One Reductions

Definition. A language $\mathcal{A}$ is *many-one reducible* to a language $\mathcal{B}$ if there exists a computable function f such that for all $w \in \Sigma^*$:

$$x \in \mathcal{A} \iff f(x) \in \mathcal{B}.$$

We write $\mathcal{A} \leq_m \mathcal{B}$.

Observation 1. If $\mathcal{A} \leq_m \mathcal{B}$ and $\mathcal{B}$ is decidable, then so is $\mathcal{A}$.

Proof. A many-one reduction is a Turing reduction, so it inherits that functionality

Observation 2. If $\mathcal{A} \leq_m \mathcal{B}$ and $\mathcal{B}$ is recursively enumerable, then so is $\mathcal{A}$.

Many-one reductions can classify problems into:
decidable/r.e./co-r.e/neither.

Properties of Many-One Reductions

1. $\leq_m$ is *reflexive* and *transitive*
   (if $\mathcal{A} \leq_m \mathcal{B}$ and $\mathcal{B} \leq_m \mathcal{C}$ then $\mathcal{A} \leq_m \mathcal{C}$, by composition of functions.)
2. If $\mathcal{A}$ is decidable and $\mathcal{B}$ is *any* language apart from $\emptyset$ and Σ^*, then $\mathcal{A} \leq_m \mathcal{B}$.

   As $\mathcal{B} \neq \emptyset$ and $\mathcal{B} \neq \Sigma^*$ there are $w_a \in \mathcal{B}$ and $w_r \notin \mathcal{B}$.

   For $w \in \Sigma^*$, define $f(w) := \begin{cases} w_a & \text{if } w \in \mathcal{A} \\ w_r & \text{if } w \notin \mathcal{A} \end{cases}$

Hence, many-one reductions are too crude to distinguish between decidable problems. later: "smarter" reductions

Examples of Many-One Reductions

We will show the following chain of reductions:

HALT $\leq_m$ ε-HALT $\leq_m$ EQUIVALENCE

ε-HALT: Does M halt on the empty input?

EQUIVALENCE: $\mathcal{L}(M) = \mathcal{L}(M')$?

Hence, all these problems are undecidable.

HALT $\leq_m$ ε-HALT

Proof.

Define function f such that $w \in \text{HALT} \iff f(w) \in \varepsilon\text{-HALT}$

For $w := \langle M, v \rangle$ compute the following Turing machine M_w :

1. Write v onto the input tape.
2. Simulate M.

Clearly, M_w accepts the empty word if, and only if, M accepts v.

Let M_r be a TM that does not halt on the empty input.

Define $f(w) := \begin{cases} M_w & \text{if } w = \langle M, v \rangle \\ M_r & \text{if } w \text{ is not of the correct input form }^1 \end{cases}$

[1] i.e. doesn't encode a TM with word

ε-HALT $\leq_m$ EQUIVALENCE

Proof.

Define f such that $w \in \varepsilon\text{-HALT} \iff f(w) \in \text{EQUIVALENCE}$

Let M_a be a Turing machine that accepts all inputs.

For a TM M compute the following Turing machine M^* :

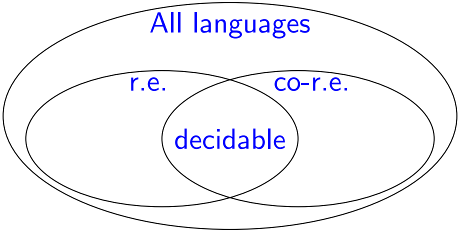
1. Run M on the empty input
2. If M halts, accept.

M^* is equivalent to M_a if, and only if, M halts on the empty input.

Define

$$f(w) := \begin{cases} (\langle M^* \rangle, \langle M_a \rangle) & \text{if } w = \langle M \rangle \\ (w, \langle M_a \rangle) & \text{if } w \text{ is not of the correct input form} \end{cases}$$

Decidable and Enumerable Languages



Recursion Theory:

Study the border between decidable and undecidable languages
Study the fine structure of undecidable languages.

The work of Turing, Church, Post, ... pre-dated modern computational machinery.

Complexity Theory:

Look at the fine structure of decidable languages.

Decision vs. optimisation vs. search

Mostly, problems of interest are *decision problems*, equivalent to language recognition problems.

But, recall e.g. TSP: naturally thought of as optimisation problem, or for a specific tour length k, as a search problem

But they are *efficiently* reducible to a decision problem ("does there exist a tour of length k"), and similarly for other optimisation problems.

Hence our focus on decision problems, although note that problems like FACTORING are naturally thought-of as search problems.

Measuring Complexity

Our general interest: detailed classification 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 Turing machine)
- **space** – the amount of additional memory needed (cells on the Turing tapes)

Next: basic complexity classes, polynomial-time reductions

Measuring Complexity

Definition.

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

1. M is *T-time bounded* if it halts on every input $w \in \Sigma^*$ after $\leq T(|w|)$ steps.
2. 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 $T, S : \mathbb{N} \to \mathbb{N}$ be monotone increasing 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 increasing 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:
  - **P** (or PTIME) $:= \bigcup_{d \in \mathbb{N}} \text{DTIME}(n^d)$ polynomial time
  - EXP $:= \bigcup_{d \in \mathbb{N}} \text{DTIME}(2^{n^d})$ exponential time
  - 2-EXP $:= \bigcup_{d \in \mathbb{N}} \text{DTIME}(2^{2^{n^d}})$ double exp time
- Space classes:
  - LOGSPACE $:= \bigcup_{d \in \mathbb{N}} \text{DSPACE}(d \log n)$
  - PSPACE $:= \bigcup_{d \in \mathbb{N}} \text{DSPACE}(n^d)$
  - EXPSPACE $:= \bigcup_{d \in \mathbb{N}} \text{DSPACE}(2^{n^d})$

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), DSPACE(S);
No for the others

Indeed, usually don’t need to refer explicitly to “Turing machine”.
But watch out for nondeterminism (details later)

Time Complexity Classes

Important Time Complexity Classes:

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

Not quite so important:

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

Note: these are all classes of decision problems, i.e. languages.

Observation:

$$\mathbf{P} \subseteq \mathrm{EXP} \subseteq 2\text{-}\mathrm{EXP} \subseteq \cdots \subseteq i\text{-}\mathrm{EXP} \subseteq \ldots$$

Time Complexity Classes

Important Time Complexity Classes:

- $\mathbf{P} := \bigcup_{d \in \mathbb{N}} \mathsf{DTIME}(n^d)$ polynomial time
- $\mathsf{EXP} := \bigcup_{d \in \mathbb{N}} \mathsf{DTIME}(2^{n^d})$ exponential time

Not quite so important:

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

Note: these are all classes of decision problems, i.e. languages.

Observation:

$$\mathbf{P} \subseteq \mathsf{EXP} \subseteq \mathsf{2\text{-}EXP} \subseteq \cdots \subseteq i\text{-}\mathsf{EXP} \subseteq \ldots$$

Alternative definition/notation:

$$\mathbf{P} := \mathsf{DTIME}(n^{O(1)})$$

Linear Speed-Up

Theorem. (Linear Speed-Up Theorem)

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

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

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

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

$$M^* := (Q', \Sigma, \Gamma', q_0', \delta', F').$$

Linear Speed-Up

Proof idea. Let $\Gamma' := \Sigma \cup \Gamma^s$ where $s := 6c$. To construct M^*:

Step 1: Compress M's input.

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

Step 2: Simulate M's computation, s steps at once.

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

(see Papadimitriou Thm 2.2, page 32)

A Hierarchy of Complexity Classes?

Questions we will study:

- 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?
- How do we show that some problem is in one of these classes but not in another?
- Are there any other interesting models of computation?
    - Non-deterministic computation
    - Randomised algorithms

Next: robustness of **P**
polynomial-time reductions

Robustness of the definition of **P**

If **P** 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 of **P**

If **P** 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 $O(t)$ steps.
3. We can simulate t steps of a RAM-machine with a 3-tape TM in $O(t^3)$ steps. Vice-versa in $O(t)$ steps.

Robustness of the definition of **P**

If **P** 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 $O(t)$ steps.
3. We can simulate t steps of a RAM-machine with a 3-tape TM in $O(t^3)$ steps. Vice-versa in $O(t)$ steps.

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

Different Encodings

Observation.

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

Observation.

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: (for problems on numbers, graphs) **P** is the same for all these encoding (unlike linear time)

Robustness of the definition of **P**

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.

(but doesn't apply to quantum or randomised algorithms)

I also pointed out that "in **P**" corresponds well to existence of a practical algorithm; problem is "tractable"

Proving a problem is in **P**

Good news: proofs of "in **P**" are often cleaner than detailed runtime analysis;
"in **P**" less specific than, e.g. "in DTIME(n^2)"; some technical details are avoided

- The most direct way to show that a problem is in **P** 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.

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

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 negated variable $\neg X_i$

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

3-CNF example:
$(X_1 \vee X_2 \vee \neg X_5) \;\wedge\; (\neg X_2 \vee \neg X_4) \;\wedge\; (X_2 \vee X_3 \vee X_4) \;\wedge X_6$

Conjunctive Normal Form

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 negated variable $\neg X_i$

k-CNF: CNF φ with at most k literals per clause.

3-CNF example:

$(X_1 \vee X_2 \vee \neg X_5) \;\wedge\; (\neg X_2 \vee \neg X_4) \;\wedge\; (X_2 \vee X_3 \vee X_4) \;\wedge X_6$

common CNF notation:

$\varphi := \{\ \{X_1, X_2, \neg X_5\}, \quad \{\neg X_2, \neg X_4\}, \quad \{X_2, X_3, X_4\}, \quad \{X_6\}\ \}$

Satisfiability

Definition. A formula φ is satisfiable if there is a satisfying assignment (a.k.a. model) 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:

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

The Satisfiability Problem

Some important problems concerning propositional formulae:

SAT

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

k-**SAT**

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

Let us also note CIRCUIT SAT: given a *circuit* with n inputs, one output, can we set input values to get output=TRUE?

2-SAT is in **P**

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

```
Let $\varphi$ be the input formula
Repeat
    If $\varphi$ contains clauses $\{X\}$ and $\{\neg X\}$, halt and output "no";
    If $\varphi$ contains clauses $\{X\}$ and $\{\neg X, Y\}$, add clause $\{Y\}$;
    If $\varphi$ contains clauses $\{X, Y\}$ $\{\neg X, Z\}$, add clause $\{Y, Z\}$;
    Any clause $\{X, X\}$ simplifies to $\{X\}$
Output "yes".
```

2-SAT is in **P**

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

```
Let $\varphi$ be the input formula
Repeat
    If $\varphi$ contains clauses $\{X\}$ and $\{\neg X\}$, halt and output "no";
    If $\varphi$ contains clauses $\{X\}$ and $\{\neg X, Y\}$, add clause $\{Y\}$;
    If $\varphi$ contains clauses $\{X, Y\}$ $\{\neg X, Z\}$, add clause $\{Y, Z\}$;
    Any clause $\{X, X\}$ simplifies to $\{X\}$
Output "yes".
```

Poly-time:

- there are $O(n^2)$ iterations.
- Each "if" test searches for $O(n^2)$ items in φ
- Each search is linear in length of φ

above analysis is crude but does the job.

Polynomial-Time Reductions

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

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 $w \in \Sigma^*$

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

Polynomial-Time Reductions

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

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 $w \in \Sigma^*$

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

Lemma. If $\mathcal{L}_1 \leq_p \mathcal{L}_2$ and $\mathcal{L}_2 \in \mathbf{P}$ then $\mathcal{L}_1 \in \mathbf{P}$.

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

Generally, members of **P** can be poly-time reduced to each other.

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 3-SAT

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

Proof. Introduce $X_{v,c}$ to represent "in a solution, v gets colour c".

clauses impose constraints, e.g. $X_{vc} \Rightarrow \neg X_{vc'}$ (or rather, $\neg X_{vc} \vee \neg X_{vc'}$)

$X_{vc} \Rightarrow \neg X_{v'c}$ for (v, v') any edge

$X_{v1} \vee X_{v2} \vee \ldots \vee X_{vk}$ for each v

can replace e.g. $X_{v1} \vee X_{v2} \vee X_{v3} \vee X_{v4}$ with $X_{v1} \vee X_{v2} \vee X_{new}$ and $\neg X_{new} \vee X_{v3} \vee X_{v4}$

We also have k-SAT$\leq_p$3-SAT, and CIRCUIT-SAT$\leq_p$3-SAT.

Reducible to 2-SAT ??