TECHNISCHE UNIVERSITÄT DRESDEN

International Center for Computational Logic

COMPLEXITY THEORY

Lecture 14: P vs. NP: Ladner's Theorem

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 3 Dec 2024

Review

Review: Hierarchies and Gaps

Hierarchy theorems tell us that more time/space leads to more power:

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSpace} \subseteq \mathsf{ExpTime} \subseteq \mathsf{NExpTime} \subseteq \mathsf{ExpSpace}$$

Gap theorems tell us that, for non-constructible functions as time/space bounds, arbitrary (constructible or not) boosts in resources may not lead to more power

Any natural problems in the hierarchy?

To show that complexity classes are different

- we have defined concrete diagonalisation languages that can show the difference (i.e., our argument was constructive),
- but these diagonalisation languages are rather artificial (i.e., not natural).

Are there, e.g., any natural ExpTime problems that are not in P?

Any natural problems in the hierarchy?

To show that complexity classes are different

- we have defined concrete diagonalisation languages that can show the difference (i.e., our argument was constructive),
- but these diagonalisation languages are rather artificial (i.e., not natural).

Are there, e.g., any natural ExpTime problems that are not in P?

Yes, many:

Theorem 14.1: If **L** is ExpTime-hard, then $\mathbf{L} \notin$ P.

Any natural problems in the hierarchy?

To show that complexity classes are different

- we have defined concrete diagonalisation languages that can show the difference (i.e., our argument was constructive),
- but these diagonalisation languages are rather artificial (i.e., not natural).

Are there, e.g., any natural ExpTime problems that are not in P?

Yes, many:

Theorem 14.1: If **L** is ExpTime-hard, then **L** $\notin$ P.

Proof: We have shown that there is a language **D** $\in$ ExpTime $\setminus$ P. If **L** is ExpTime-hard, then there is a polynomial many-one reduction **D** $\leq_p$ **L**. Therefore, if **L** were in P, then so would **D** – contradiction. □

Similar results hold for other classes we separated: A problem that is hard for the larger class cannot be included in the smaller.

Ladner's Theorem

P vs. NP revisited

We have seen that a great variety of difficult problems in NP turn out to be NP-complete.

A natural question to ask is whether this apparent dichotomy is a law of nature:

Hypothesis: Every problem in NP is either in P or NP-complete.

P vs. NP revisited

We have seen that a great variety of difficult problems in NP turn out to be NP-complete.

A natural question to ask is whether this apparent dichotomy is a law of nature:

Hypothesis: Every problem in NP is either in P or NP-complete.

In 1975, Richard E. Ladner showed that this is wrong, unless P = NP

(in the latter case, uninterestingly, the non-trivial problems in P would turn out to be exactly the set of NP-complete problems)

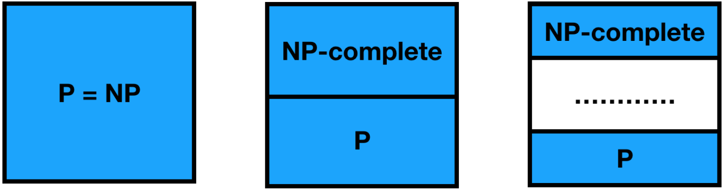
Theorem 14.2 (Ladner, 1975): If P $\neq$ NP, then there are problems in NP that are neither in P nor NP-complete.

Such problems are called NP-intermediate.

Illustration

Theorem 14.2 (Ladner, 1975): If P $\neq$ NP, then there are problems in NP that are neither in P nor NP-complete.

In other words, given the following illustrations of the possible relationships between P and NP:



a) b) c)

Ladner tells us that the middle cannot be correct.

Proving the Theorem

Theorem 14.2 (Ladner, 1975): If P $\neq$ NP, then there are problems in NP that are neither in P nor NP-complete.

Proof idea: We will directly define an NP-intermediate language by defining an NTM $\mathcal{K}$ that recognises it.

Proving the Theorem

Theorem 14.2 (Ladner, 1975): If P $\neq$ NP, then there are problems in NP that are neither in P nor NP-complete.

Proof idea: We will directly define an NP-intermediate language by defining an NTM $\mathcal{K}$ that recognises it.

We want to construct $\mathbf{L}(\mathcal{K})$ to be:

(1) different from all problems in P

(2) different from all problems that **SAT** can be reduced to

Observation: This is similar to two concurrent diagonalisation arguments

Proving the Theorem

Theorem 14.2 (Ladner, 1975): If P $\neq$ NP, then there are problems in NP that are neither in P nor NP-complete.

Proof idea: We will directly define an NP-intermediate language by defining an NTM $\mathcal{K}$ that recognises it.

We want to construct $\mathbf{L}(\mathcal{K})$ to be:
(1) different from all problems in P
(2) different from all problems that **SAT** can be reduced to

Observation: This is similar to two concurrent diagonalisation arguments

Moreover, the sets we diagonalise against are effectively enumerable:

- There is an effective enumeration $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \ldots$ of all polynomially time-bounded DTMs, each together with a suitable bounding function
  For example, enumerate all pairs of TMs and polynomials, and make the enumeration consist of the TMs obtained by artificially restricting the run of a TM with a suitable countdown.
- There is an effective enumeration $\mathcal{R}_0, \mathcal{R}_1, \mathcal{R}_2, \ldots$ of all polynomial many-one reductions, each together with a suitable bounding function
  This is similar to enumerating polytime TMs; we can restrict to one input alphabet that we also use for **SAT**

The problem with diagonalisation

How can we do two diagonalisations at once?

The problem with diagonalisation

How can we do two diagonalisations at once? — Simply interleave the enumerations:

- In each even "step" $2i$, show that the ith polytime TM $\mathcal{M}_i$ is not equivalent to $\mathcal{K}$: there is w such that $\mathcal{M}_i(w) \neq \mathcal{K}(w)$
- In each odd "step" $2i + 1$, show that the ith reduction $\mathcal{R}_i$ does not reduce **Sat** to $\mathcal{K}$: there is w such that $\mathcal{K}(\mathcal{R}_i(w)) \neq \textbf{Sat}(w)$

The problem with diagonalisation

How can we do two diagonalisations at once? — Simply interleave the enumerations:

- In each even "step" $2i$, show that the ith polytime TM $\mathcal{M}_i$ is not equivalent to $\mathcal{K}$: there is w such that $\mathcal{M}_i(w) \neq \mathcal{K}(w)$
- In each odd "step" $2i + 1$, show that the ith reduction $\mathcal{R}_i$ does not reduce **Sat** to $\mathcal{K}$: there is w such that $\mathcal{K}(\mathcal{R}_i(w)) \neq$ **Sat**(w)

Nevertheless, there is a problem: How can we flip the output of **Sat**?

- $\mathcal{K}$ is required to run in NP
- Computing the actual result of **Sat** is NP-hard
- To show $\mathcal{K}(\mathcal{R}_i(w)) \neq$ **Sat**(w), one might have to show $w \notin$ **Sat**, which is presumably not in NP

$\leadsto$ the required computation seems too hard!

Solution: Lazy diagonalisation

Idea: Do not attempt to show too much on small inputs, but wait patiently until inputs are large enough to show the required differences

Main ingredients:

- A very slow growing but polynomially computable function f
- A problem in NP that is NP-hard: **Sat**
- A problem in NP that is not NP-hard:

Solution: Lazy diagonalisation

Idea: Do not attempt to show too much on small inputs, but wait patiently until inputs are large enough to show the required differences

Main ingredients:

- A very slow growing but polynomially computable function f
- A problem in NP that is NP-hard: **Sat**
- A problem in NP that is not NP-hard: $\emptyset$

Solution: Lazy diagonalisation

Idea: Do not attempt to show too much on small inputs, but wait patiently until inputs are large enough to show the required differences

Main ingredients:

- A very slow growing but polynomially computable function f
- A problem in NP that is NP-hard: **Sat**
- A problem in NP that is not NP-hard: $\emptyset$

We will define a TM $\mathcal{K}$ that does the following on input w:

(1) Compute the value $f(|w|)$

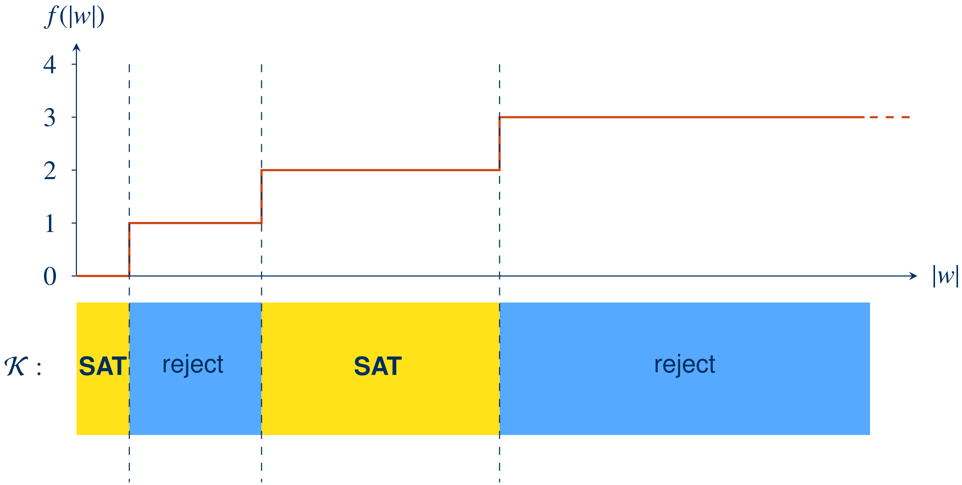
(2) If $f(|w|)$ is even: return whether $w \in$ **Sat**

(3) If $f(|w|)$ is odd: return whether $w \in \emptyset$, i.e., reject

Intuition: the NP-intermediate language $\mathbf{L}(\mathcal{K})$ is **Sat** with "holes punched out of it" (namely for all inputs where f is odd)

Illustration of $\mathcal{K}$'s behaviour

We can sketch the behaviour of $\mathcal{K}$ as follows:



What is f?

Reminder: $\mathcal{K}(w)$ is **Sat**(w) if $f(|w|)$ is even, and false if $f(|w|)$ is odd.

The key to the proof is the definition of f – this is where the diagonalisation happens.

What is f?

Reminder: $\mathcal{K}(w)$ is **Sat**(w) if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

The key to the proof is the definition of f – this is where the diagonalisation happens.

Intuition: Keep the current value of f until progress has been made in diagonalisation

- Keep an even value $f(|w|) = 2i$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{M}_i(v) \neq \mathcal{K}(v)$
- Keep an odd value $f(|w|) = 2i + 1$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{K}(\mathcal{R}_i(v)) \neq$ **Sat**(v)

What is f?

Reminder: $\mathcal{K}(w)$ is **Sat**(w) if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

The key to the proof is the definition of f – this is where the diagonalisation happens.

Intuition: Keep the current value of f until progress has been made in diagonalisation

- Keep an even value $f(|w|) = 2i$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{M}_i(v) \neq \mathcal{K}(v)$
- Keep an odd value $f(|w|) = 2i + 1$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{K}(\mathcal{R}_i(v)) \neq$ **Sat**(v)

If we can do this in NP, it will be enough already:

What is f?

Reminder: $\mathcal{K}(w)$ is **Sat**(w) if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

The key to the proof is the definition of f – this is where the diagonalisation happens.

Intuition: Keep the current value of f until progress has been made in diagonalisation

- Keep an even value $f(|w|) = 2i$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{M}_i(v) \neq \mathcal{K}(v)$
- Keep an odd value $f(|w|) = 2i + 1$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{K}(\mathcal{R}_i(v)) \neq$ **Sat**(v)

If we can do this in NP, it will be enough already:

- If $\mathcal{K}$ were equivalent to any $\mathcal{M}_i$, then f would eventually become an even constant, and $\mathcal{K}$ would solve **Sat** on all but finitely many instances
  $\leadsto$ $\mathcal{K}$ would be NP-hard, and equivalent to a polytime TM $\leadsto$ P = NP

What is f?

Reminder: $\mathcal{K}(w)$ is **Sat**(w) if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

The key to the proof is the definition of f – this is where the diagonalisation happens.

Intuition: Keep the current value of f until progress has been made in diagonalisation

- Keep an even value $f(|w|) = 2i$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{M}_i(v) \neq \mathcal{K}(v)$
- Keep an odd value $f(|w|) = 2i + 1$ until you can show in polynomial time (in $|w|$) that there is v such that $\mathcal{K}(\mathcal{R}_i(v)) \neq$ **Sat**(v)

If we can do this in NP, it will be enough already:

- If $\mathcal{K}$ were equivalent to any $\mathcal{M}_i$, then f would eventually become an even constant, and $\mathcal{K}$ would solve **Sat** on all but finitely many instances
  $\leadsto$ $\mathcal{K}$ would be NP-hard, and equivalent to a polytime TM $\leadsto$ P = NP
- If $\mathcal{K}$ would allow **Sat** to be reduced to it by some reduction $\mathcal{R}_i$, then f would eventually become an odd constant, and $\mathbf{L}(\mathcal{K})$ would be a finite language
  $\leadsto$ $\mathcal{K}$ would be in P, and **Sat** would reduce to it $\leadsto$ P = NP

In either case, this contradicts our assumption that P $\neq$ NP

What is f?

We consider some fixed deterministic TM $\mathcal{S}$ with $\mathbf{L}(\mathcal{S}) = $ **Sat**, and an enumeration $v_0, v_1, \ldots$ of all words ordered by length, and lexicographic for words of equal length.

Reminder: $\mathcal{K}(w)$ is $\mathcal{S}(w)$ if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

What is f?

We consider some fixed deterministic TM $\mathcal{S}$ with $\mathbf{L}(\mathcal{S}) = \textsc{Sat}$, and an enumeration $v_0, v_1, \ldots$ of all words ordered by length, and lexicographic for words of equal length.

Reminder: $\mathcal{K}(w)$ is $\mathcal{S}(w)$ if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

Definition: The value of f on input w with $|w| = n$ is defined recursively

What is f?

We consider some fixed deterministic TM $\mathcal{S}$ with $\mathbf{L}(\mathcal{S}) = \textsc{Sat}$, and an enumeration $v_0, v_1, \ldots$ of all words ordered by length, and lexicographic for words of equal length.

Reminder: $\mathcal{K}(w)$ is $\mathcal{S}(w)$ if $f(|w|)$ is even, and false if $f(|w|)$ is odd.

Definition: The value of f on input w with $|w| = n$ is defined recursively

(1) Perform the computations of $f(0), f(1), f(2), \ldots$ in order until n computing steps have been performed in total. Store the largest value $f(\ell) = k$ that could be computed in this time (set $k = 0$ if no value was computed).

What is f?

We consider some fixed deterministic TM $\mathcal{S}$ with $\mathbf{L}(\mathcal{S}) = \textsc{Sat}$, and an enumeration $v_0, v_1, \ldots$ of all words ordered by length, and lexicographic for words of equal length.

Reminder: $\mathcal{K}(w)$ is $\mathcal{S}(w)$ if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

Definition: The value of f on input w with $|w| = n$ is defined recursively

(1) Perform the computations of $f(0), f(1), f(2), \ldots$ in order until n computing steps have been performed in total. Store the largest value $f(\ell) = k$ that could be computed in this time (set $k = 0$ if no value was computed).

(2) Determine if $f(n)$ should remain k or increase to $k + 1$:

What is f?

We consider some fixed deterministic TM $\mathcal{S}$ with $\mathbf{L}(\mathcal{S}) = \textsc{Sat}$, and an enumeration $v_0, v_1, \ldots$ of all words ordered by length, and lexicographic for words of equal length.

Reminder: $\mathcal{K}(w)$ is $\mathcal{S}(w)$ if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

Definition: The value of f on input w with $|w| = n$ is defined recursively

(1) Perform the computations of $f(0), f(1), f(2), \ldots$ in order until n computing steps have been performed in total. Store the largest value $f(\ell) = k$ that could be computed in this time (set $k = 0$ if no value was computed).

(2) Determine if $f(n)$ should remain k or increase to $k + 1$:

- (2.a) If $k = 2i$ is even: Iterate over all words v, simulate $\mathcal{M}_i(v)$, $\mathcal{S}(v)$, and (recursively) compute $f(|v|)$. Terminate this effort after n steps. If a word is found such that $\mathcal{K}(v) \neq \mathcal{M}_i(v)$, then return $k + 1$; else return k

What is f?

We consider some fixed deterministic TM $\mathcal{S}$ with $\mathbf{L}(\mathcal{S}) = \textsc{Sat}$, and an enumeration $v_0, v_1, \ldots$ of all words ordered by length, and lexicographic for words of equal length.

Reminder: $\mathcal{K}(w)$ is $\mathcal{S}(w)$ if $f(|w|)$ is even, and *false* if $f(|w|)$ is odd.

Definition: The value of f on input w with $|w| = n$ is defined recursively

(1) Perform the computations of $f(0), f(1), f(2), \ldots$ in order until n computing steps have been performed in total. Store the largest value $f(\ell) = k$ that could be computed in this time (set $k = 0$ if no value was computed).

(2) Determine if $f(n)$ should remain k or increase to $k + 1$:

- (2.a) If $k = 2i$ is even: Iterate over all words v, simulate $\mathcal{M}_i(v)$, $\mathcal{S}(v)$, and (recursively) compute $f(|v|)$. Terminate this effort after n steps. If a word is found such that $\mathcal{K}(v) \neq \mathcal{M}_i(v)$, then return $k + 1$; else return k
- (2.b) If $k = 2i + 1$ is odd: Iterate over all words v, simulate $\mathcal{R}_i(v)$ (this produces a word), $\mathcal{S}(v)$, $\mathcal{S}(\mathcal{R}_i(v))$, and (recursively) compute $f(|\mathcal{R}_i(v)|)$. Terminate this effort after n steps. If a word is found such that $\mathcal{K}(\mathcal{R}_i(v)) \neq \mathcal{S}(v)$, then return $k + 1$; else return k.

Is f well-defined?

Our definition of f computes values for f recursively. Is this ok?

- Yes, the computation that needs to be done for each $f(n)$ is fully defined
- All the simulated TMs are known or computable
- Since computation is time-limited to the input value n, there is no danger of endless recursion
- For example, $f(0) = 0$: nothing will be achieved in 0 steps

Is f well-defined?

Our definition of f computes values for f recursively. Is this ok?

- Yes, the computation that needs to be done for each $f(n)$ is fully defined
- All the simulated TMs are known or computable
- Since computation is time-limited to the input value n, there is no danger of endless recursion
- For example, $f(0) = 0$: nothing will be achieved in 0 steps

Indeed, f grows very slowly!

- A large input n might be needed to find the next counterexample word v needed in diagonalisation
- Even if such v was found in n steps (making progress from n to $n + 1$), it will be only much later that $f(n)$ can be computed in step (1) and f will even start to look for a way of getting to $n + 2$.
- In fact, already the requirement to recompute all previous values of f before considering an increase ensures that $f \in O(\log \log n)$.

Concluding the Proof

Theorem 14.2 (Ladner, 1975): If P $\neq$ NP, then there are problems in NP that are neither in P nor NP-complete.

Proof: Let $\mathcal{K}$ be defined as before.

Concluding the Proof

Theorem 14.2 (Ladner, 1975): If P ≠ NP, then there are problems in NP that are neither in P nor NP-complete.

Proof: Let $\mathcal{K}$ be defined as before.

$\mathcal{K}$ runs in nondeterministic polynomial time:

- The computation of f is in polynomial deterministic time (since it is artificially bounded to a short time)
- The computation of **Sat** for the cases where $f(|w|)$ is even is possible in NP

Concluding the Proof

Theorem 14.2 (Ladner, 1975): If P ≠ NP, then there are problems in NP that are neither in P nor NP-complete.

Proof: Let $\mathcal{K}$ be defined as before.

$\mathcal{K}$ runs in nondeterministic polynomial time:

- The computation of f is in polynomial deterministic time (since it is artificially bounded to a short time)
- The computation of **Sat** for the cases where $f(|w|)$ is even is possible in NP

L($\mathcal{K}$) is not in P: As argued before: if it were in P, it would be equivalent to some polytime TM $\mathcal{M}_i$, and f would eventually be constant at $2i$, making $\mathcal{K}$ equivalent to **Sat** (up to finite variations), which contradicts P ≠ NP.

Concluding the Proof

Theorem 14.2 (Ladner, 1975): If P ≠ NP, then there are problems in NP that are neither in P nor NP-complete.

Proof: Let $\mathcal{K}$ be defined as before.

$\mathcal{K}$ runs in nondeterministic polynomial time:

- The computation of f is in polynomial deterministic time (since it is artificially bounded to a short time)
- The computation of **Sat** for the cases where $f(|w|)$ is even is possible in NP

L$(\mathcal{K})$ is not in P: As argued before: if it were in P, it would be equivalent to some polytime TM $\mathcal{M}_i$, and f would eventually be constant at $2i$, making $\mathcal{K}$ equivalent to **Sat** (up to finite variations), which contradicts P ≠ NP.

L$(\mathcal{K})$ is not NP-hard: As argued before: if it were NP-hard, there would be a polynomial many-one reduction $\mathcal{R}_i$ from **Sat**, and f would eventually be constant at $2i + 1$, making $\mathcal{K}$ equivalent to ∅ (up to finite variations), which contradicts P ≠ NP. □

Discussion: Proof of Ladner's Theorem

Note 1: It is interesting to meditate on the following facts:

- We have defined a rather "busy" computation of f that checks that diagonalisation (over two different sets) must happen
- This definition of computation is essential to prove the result
- Nevertheless, diagonalisation remained "internal": from the outside, $\mathcal{K}$ is just a TM that sometimes solves **Sat** (for a long range of inputs), and at other times just rejects every input (again for very long ranges of inputs)

Discussion: Proof of Ladner's Theorem

Note 1: It is interesting to meditate on the following facts:

- We have defined a rather "busy" computation of f that checks that diagonalisation (over two different sets) must happen
- This definition of computation is essential to prove the result
- Nevertheless, diagonalisation remained "internal": from the outside, $\mathcal{K}$ is just a TM that sometimes solves **Sat** (for a long range of inputs), and at other times just rejects every input (again for very long ranges of inputs)

Note 2: The constructed language is very artificial

- It is very "non-uniform" in terms of how hard it is, alternating between long stretches of NP-hardness and long stretches of triviality

Discussion: Proof of Ladner's Theorem

Note 1: It is interesting to meditate on the following facts:

- We have defined a rather "busy" computation of f that checks that diagonalisation (over two different sets) must happen
- This definition of computation is essential to prove the result
- Nevertheless, diagonalisation remained "internal": from the outside, $\mathcal{K}$ is just a TM that sometimes solves **Sat** (for a long range of inputs), and at other times just rejects every input (again for very long ranges of inputs)

Note 2: The constructed language is very artificial

- It is very "non-uniform" in terms of how hard it is, alternating between long stretches of NP-hardness and long stretches of triviality

Note 3: Are there any natural problems that are known to be NP-intermediate?

Discussion: Proof of Ladner's Theorem

Note 1: It is interesting to meditate on the following facts:

- We have defined a rather "busy" computation of f that checks that diagonalisation (over two different sets) must happen
- This definition of computation is essential to prove the result
- Nevertheless, diagonalisation remained "internal": from the outside, $\mathcal{K}$ is just a TM that sometimes solves **Sat** (for a long range of inputs), and at other times just rejects every input (again for very long ranges of inputs)

Note 2: The constructed language is very artificial

- It is very "non-uniform" in terms of how hard it is, alternating between long stretches of NP-hardness and long stretches of triviality

Note 3: Are there any natural problems that are known to be NP-intermediate?

- No: finding one would prove P $\neq$ NP
- Candidate problems (link) include, e.g., **Graph Isomorphism** and **Factoring**
  Beware: the latter is not about deciding if a number is prime, but about checking something specific about its factors, e.g., whether the largest factor contains at least one 7 when written in decimal

Summary and Outlook

Ladner's theorem tells us that, in the intuitive case that P $\neq$ NP, there must (counterintuitively?) be many problems in NP that are neither polynomially solvable nor NP-complete

The proof is based on a technique of lazy diagonalisation

What's next?

- Generalising Ladner's Theorem
- Computing with oracles (reprise)
- The limits of diagonalisation, proved by diagonalisation