# Computational complexity

lecture 8

# Berman's theorem (⋆)

Theorem (Berman 1978)
If **P**≠**NP**, then no language over a single-letter alphabet is **NP**-hard.

In consequence there are difficult (and even undecidable) languages that are not **NP**-hard.

# Berman's theorem (∗)

Theorem (Berman 1978)

If **P**≠**NP**, then no language over a single-letter alphabet is **NP**-hard.

Proof

Let $L$ be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P**≠**NP**.

# Berman's theorem (∗)

<u>Theorem</u> (Berman 1978)

If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard.

<u>Proof</u>

Let $L$ be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P≠NP**.

By assumption there is a reduction $g$ from SAT to $L$.

The algorithm is as follows:

- We are given a formula $\phi$
- We will keep a list of formulas $\psi_1,\ldots,\psi_k$ such that: $\phi$ is satisfiable iff some of $\psi_1,\ldots,\psi_k$ is satisfiable. Initially the list contains $\phi$.

# Berman's theorem (∗)

Theorem (Berman 1978)

If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard.

Proof

Let $L$ be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P≠NP**.
By assumption there is a reduction $g$ from SAT to $L$.

The algorithm is as follows:

- We are given a formula $\phi$
- We will keep a list of formulas $\psi_1,\dots,\psi_k$ such that: $\phi$ is satisfiable iff some of $\psi_1,\dots,\psi_k$ is satisfiable. Initially the list contains $\phi$.
- We alternatingly repeat two kinds of steps:

1) Replace every $\psi_i$ by two formulas: $\psi_i[true/x]$ and $\psi_i[false/x]$, obtained by substituting true/false for one of variables.
(clearly $\psi_i$ is satisfiable iff some of $\psi_i[true/x]$, $\psi_i[false/x]$ is satisfiable)

# Berman's theorem (∗)

Theorem (Berman 1978)

If **P≠NP**, then no language over a single-letter alphabet is **NP**-hard.

Proof

Let $L$ be an **NP**-hard language over a single-letter alphabet. We will give a polynomial-time algorithm for SAT, contradicting **P≠NP**.

By assumption there is a reduction $g$ from SAT to $L$.

The algorithm is as follows:

- We are given a formula $\phi$
- We will keep a list of formulas $\psi_1,\ldots,\psi_k$ such that: $\phi$ is satisfiable iff some of $\psi_1,\ldots,\psi_k$ is satisfiable. Initially the list contains $\phi$.
- We alternatingly repeat two kinds of steps:

1) Replace every $\psi_i$ by two formulas: $\psi_i[true/x]$ and $\psi_i[false/x]$, obtained by substituting true/false for one of variables.
(clearly $\psi_i$ is satisfiable iff some of $\psi_i[true/x]$, $\psi_i[false/x]$ is satisfiable)

2) For every pair $\psi_i,\psi_j$ such that $g(\psi_i)=g(\psi_j)$, remove $\psi_i$ from the list, leave only $\psi_j$ (notice that $\psi_i$ is satisfiable iff some of $\psi_j$ is satisfiable)

# Berman's theorem (∗)

We alternatingly repeat two kinds of steps:

1) Replace every $\psi_i$ by two formulas: $\psi_i[true/x]$ and $\psi_i[false/x]$, obtained by substituting true/false for one of variables.
(clearly $\psi_i$ is satisfiable iff some of $\psi_i[true/x]$, $\psi_i[false/x]$ is satisfiable)

2) For every pair $\psi_i, \psi_j$ such that $g(\psi_i)=g(\psi_j)$, remove $\psi_i$ from the list, leave only $\psi_j$ (notice that $\psi_i$ is satisfiable iff some of $\psi_j$ is satisfiable)

The algorithm is correct. Why does it work in polynomial time?

- Recall that $g$ is a polynomial-time reduction to a single-letter language. Thus $|g(\psi_i)|<p(|\psi_i|)$ for some polynomial $p$.
  Since there is only one single-letter word of every length, there are only $p(|\psi_i|)\leq p(|\phi|)$ possibilities for $g(\psi_i)$.
- In effect, the list has length $\leq p(|\phi|)$ after every execution of step 2, and $\leq 2 \cdot p(|\phi|)$ after every execution of step 1.
- Moreover, every step can be performed in polynomial time.

This finishes the proof.

# Relativisation

Many proofs in the complexity theory uses Turing machines as “black-boxes” – the proofs are of the form:

- assume that there is a machine $M$ working in time ... recognizing ...
- Out of it, we create $M'$, which executes $M$ many times in a loop...
- ... then it negates the results, executes itself on every machine ...
- at the end we obtain a machine $M'''''$, about which we know that it cannot exist, thus $M$ could not exist.

Such proofs <u>relativize</u>, i.e., they work also when every machine in the world has access to some fixed oracle (that is, it can ask whether a word belongs to a language $L$, and immediately obtain an answer)

# Relativisation

Many proofs in the complexity theory uses Turing machines as "black-boxes" – the proofs are of the form:

- assume that there is a machine $M$ working in time ... recognizing ...
- Out of it, we create $M'$, which executes $M$ many times in a loop...
- ...

Such proofs relativize, i.e., they work also when every machine in the world has access to some fixed oracle.

Examples of relativizing proofs: Turing theorem about undecidability, hierarchy theorems, gap theorems, Ladner's theorem, Immerman-Szelepcseny theorem, Savitch theorem, ...

On the other hand, proofs based on circuits do not relativize (it is not at all clear what is an oracle for a circuit)

The next theorem shows that using relativizing arguments we cannot solve the **P** vs. **NP** problem.

# Baker-Gill-Solovay theorem

Theorem (Baker-Gill-Solovay, 1975)

There exist languages $A$ and $B$ such that $\mathbf{P}^{A}=\mathbf{NP}^{A}$ and $\mathbf{P}^{B}\neq\mathbf{NP}^{B}$

# Baker-Gill-Solovay theorem

Theorem (Baker-Gill-Solovay, 1975)

There exist languages $A$ and $B$ such that $\mathbf{P}^{A}=\mathbf{NP}^{A}$ and $\mathbf{P}^{B}\neq\mathbf{NP}^{B}$

Proof

As $A$ we can take QBF – we have:

$$\mathbf{NP}^{\mathrm{QBF}}\subseteq\mathbf{NPSPACE}=\mathbf{PSPACE}=\mathbf{P}^{\mathrm{QBF}}$$

Steps from the left:

- instead of asking the QBF oracle about a word, a machine can itself compute the answer (questions are of polynomial length, and QBF can be solved in polynomial space)
- Savitch theorem
- **PSPACE**-completeness of the QBF problem

# Baker-Gill-Solovay theorem

Theorem (Baker-Gill-Solovay, 1975)

There exist languages $A$ and $B$ such that $\mathbf{P}^A=\mathbf{NP}^A$ and $\mathbf{P}^B\neq\mathbf{NP}^B$

Proof

As $A$ we can take QBF – we have:

$$\mathbf{NP}^{\mathrm{QBF}}\subseteq\mathbf{NPSPACE}=\mathbf{PSPACE}=\mathbf{P}^{\mathrm{QBF}}$$

Steps from the left:

- instead of asking the QBF oracle about a word, a machine can itself compute the answer (questions are of polynomial length, and QBF can be solved in polynomial space)
- Savitch theorem
- **PSPACE**-completeness of the QBF problem

Does $A$=SAT work as well? – $\mathbf{NP}^{\mathrm{SAT}}\subseteq\mathbf{NP}\subseteq\mathbf{P}^{\mathrm{SAT}}$

# Baker-Gill-Solovay theorem

Theorem (Baker-Gill-Solovay, 1975)

There exist languages $A$ and $B$ such that $\mathbf{P}^A=\mathbf{NP}^A$ and $\mathbf{P}^B\neq\mathbf{NP}^B$

Proof

As $A$ we can take QBF – we have:

$$\mathbf{NP}^{\mathrm{QBF}}\subseteq\mathbf{NPSPACE}=\mathbf{PSPACE}=\mathbf{P}^{\mathrm{QBF}}$$

Steps from the left:

- instead of asking the QBF oracle about a word, a machine can itself compute the answer (questions are of polynomial length, and QBF can be solved in polynomial space)
- Savitch theorem
- **PSPACE**-completeness of the QBF problem

Does $A$=SAT work as well? – $\cancel{\mathbf{NP}^{\mathrm{SAT}}\subseteq\mathbf{NP}\subseteq\mathbf{P}^{\mathrm{SAT}}}$

NO – an **NP** algorithm for SAT doesn't give the inclusion $\mathbf{NP}^{\mathrm{SAT}}\subseteq\mathbf{NP}$ (maybe the external algorithm „prefers" to obtain that a formula is not satisfiable, and it will incorrectly compute its satisfiability)

It is important that QBF can be solved in deterministic **PSPACE**

# Baker-Gill-Solovay theorem (∗)

<u>Theorem</u> (Baker-Gill-Solovay, 1975)

There exist languages $A$ and $B$ such that $\mathbf{P}^A = \mathbf{NP}^A$ and $\mathbf{P}^B \neq \mathbf{NP}^B$

<u>Proof</u>

We now construct an oracle $B$, and we consider the language

$L = \{1^n$ : some word $w$ of length $n$ belongs to $B\}$

- Clearly $L \in \mathbf{NP}^B$ – nondeterministic machine can guess some $w \in B$
- A deterministic machine recognizing $L$ has a problem: it can only ask the oracle for consecutive words, but it has not enough time to check all of them. We only need to choose $B$ so that indeed it is impossible to do anything better.

# Baker-Gill-Solovay theorem (∗)

<u>Theorem</u> (Baker-Gill-Solovay, 1975)

There exist languages $A$ and $B$ such that $\mathbf{P}^A=\mathbf{NP}^A$ and $\mathbf{P}^B\neq\mathbf{NP}^B$

<u>Proof</u>

$L=\{1^n :$ some word $w$ of length $n$ belongs to $B\}$

We now choose $B$:

- Fix a list $M_1,M_2,M_3,\ldots$ of all Turing machines with oracle working in polynomial time
  - ➔ an oracle is not a part of the definition of the machine,
  - ➔ for every $M_i$ there should exist a polynomial $p_i$ such that for every oracle the machine $M_i$ works in time $p_i(n)$
  - ➔ if some $M$ with oracle $C$ recognizes a language $L$ in polynomial time, then some $M_i$ with oracle $C$ also recognizes $L$
  - ➔ such a list $M_1,M_2,M_3,\ldots$ is created as in the proof of Ladner's theo.
  - ➔ this time, we do not use the fact that the list is computable (conversely to the proof of the Ladner's theorem)
- We construct $B$ gradually, cheating consecutive machines

# Baker-Gill-Solovay theorem (∗)

$L=\{1^n$ : some word $w$ of length $n$ belongs to $B\}$

We create $B=\bigcup_{i\in\mathbb{N}}B_i$ and a sequence $n_i$ such that:

- $M_i^{B_i}$ incorrectly recognizes the word $1^{n_i}$
- $M_i^B$ agrees with $M_i^{B_i}$ on the word $1^{n_i}$

We start with $B_0=\emptyset$; then for consecutive $i$:

- we take $n_i$ so large that for all $j<i$, machine $M_j$ for on the word $1^{n_j}$ produces only queries shorter than $n_i$ (thanks to this the machines that were cheated earlier remain cheated), and such that $M_i$ on the word $1^{n_i}$ works in less than $2^{n_i}$ steps

# Baker-Gill-Solovay theorem (∗)

$L=\{1^n$ : some word $w$ of length $n$ belongs to $B\}$

We create $B=\bigcup_{i\in\mathbb{N}} B_i$ and a sequence $n_i$ such that:

- $M_i^{B_i}$ incorrectly recognizes the word $1^{n_i}$
- $M_i^B$ agrees with $M_i^{B_i}$ on the word $1^{n_i}$

We start with $B_0=\emptyset$; then for consecutive $i$:

- we take $n_i$ so large that for all $j<i$, machine $M_j$ for on the word $1^{n_j}$ produces only queries shorter than $n_i$ (thanks to this the machines that were cheated earlier remain cheated), and such that $M_i$ on the word $1^{n_i}$ works in less than $2^{n_i}$ steps
- run $M_i^{B_{i-1}}$ on the word $1^{n_i}$
- if it accepts, take $B_i=B_{i-1}$ – then $1^{n_i}\notin L$, we have cheated $M_i$
- if it rejects, find a word $w$ of length $n_i$ about which $M_i$ haven't asked (it exists, since $M_i$ has made $<2^{n_i}$ step) and define $B_i=B_{i-1}\cup\{w\}$ Then $1^{n_i}\in L$, and we have cheated $M_i$

# Baker-Gill-Solovay theorem (∗)

$L=\{1^n$ : some word $w$ of length $n$ belongs to $B\}$

We create $B=\bigcup_{i\in\mathbb{N}}B_i$ and a sequence $n_i$ such that:

- $M_i^{B_i}$ incorrectly recognizes the word $1^{n_i}$
- $M_i^{B}$ agrees with $M_i^{B_i}$ on the word $1^{n_i}$

The language $B$ is computable, but in this theorem this is meaningless

We start with $B_0=\emptyset$; then for consecutive $i$:

- we take $n_i$ so large that for all $j<i$, machine $M_j$ for on the word $1^{n_j}$ produces only queries shorter than $n_i$ (thanks to this the machines that were cheated earlier remain cheated), and such that $M_i$ on the word $1^{n_i}$ works in less than $2^{n_i}$ steps
- run $M_i^{B_{i-1}}$ on the word $1^{n_i}$
- if it accepts, take $B_i=B_{i-1}$ – then $1^{n_i}\notin L$, we have cheated $M_i$
- if it rejects, find a word $w$ of length $n_i$ about which $M_i$ haven't asked (it exists, since $M_i$ has made $<2^{n_i}$ step) and define $B_i=B_{i-1}\cup\{w\}$ Then $1^{n_i}\in L$, and we have cheated $M_i$

# Search problems

The **NP** class was defined for decision problems („yes/no”),

e.g., does there exist a valuation satisfying a formula,
does there exist a Hamiltonian cycle, ...

We can also consider search problems,

e.g., find a valuation satisfying a formula,
find a Hamiltonian cycle, ...

- Of course search problems are not easier than decision problems. Thus if **P**≠**NP**, then search problems cannot be solved in polynomial time as well.
- And what if **P**=**NP**? Maybe it is possible to decide quickly whether there is a Hamiltonian cycle, but it is impossible to quickly find it?

# Search problems

The **NP** class was defined for decision problems („yes/no”),

e.g., does there exist a valuation satisfying a formula,
does there exist a Hamiltonian cycle, ...

We can also consider search problems,

e.g., find a valuation satisfying a formula,
find a Hamiltonian cycle, ...

- Of course search problems are not easier than decision problems. Thus if **P**≠**NP**, then search problems cannot be solved in polynomial time as well.
- And what if **P**=**NP**? ~~Maybe it is possible to decide quickly whether there is a Hamiltonian cycle, but it is impossible to quickly find it?~~
- Then it possible to solve also search problems in polynomial time.

# Search problems

Theorem

If **P**=**NP**, then for every language $L \in$ **NP** there is a polynomial algorithm that reads $v \in L$ and finds a witness for $v$.

We refer here to the definition of **NP** using witnesses:

**NP** contains languages of the form $\{v : \exists w.\ v\$w \in R\}$, where $R$ is a relation recognizable in polynomial time and such that $v\$w \in R$ implies $|w| \leq p(|v|)$ for some polynomial $p$.

# Search problems

<u>Theorem</u>

If **P**=**NP**, then for every language $L\in$**NP** there is a polynomial algorithm that reads $v\in L$ and finds a witness for $v$.

We refer here to the definition of **NP** using witnesses:

**NP** contains languages of the form $\{v : \exists w.\ v\$w\in R\}$, where $R$ is a relation recognizable in polynomial time and such that $v\$w\in R$ implies $|w|\leq p(|v|)$ for some polynomial $p$.

<u>Proof</u>

Consider first the SAT problem – we assume that there is a polynomial-time algorithm $A$ for SAT, we want to find a valuation:

- Using $A$ we check whether the formula is satisfiable
- If yes, we set $x_1=1$ and we check whether it is still satisfiable
- Yes ⇒ keep $x_1=1$ and continue for a smaller formula
- No ⇒ set $x_1=0$ and continue for a smaller formula
- In this way we eliminate consecutive variables, and we obtain a whole valuation

# Search problems

Theorem

If **P**=**NP**, then for every language $L \in$ **NP** there is a polynomial algorithm that reads $v \in L$ and finds a witness for $v$.

We refer here to the definition of **NP** using witnesses:
**NP** contains languages of the form $\{v : \exists w.\ v\$w \in R\}$, where $R$ is a relation recognizable in polynomial time and such that $v\$w \in R$ implies $|w| \leq p(|v|)$ for some polynomial $p$.

Proof

- For SAT we already know, consider now an arbitrary problem from **NP**
- It is enough to see that the reduction from the Cook-Levin theorem (**NP**-hardness of SAT) is actually a Levin reduction (i.e., it allows to recover witnesses)

# Polynomial hierarchy

The following problem is in **NP**:

INDSET = {$(G,k)$ : in graph $G$ there is an independent set of size $\geq k$}

Consider now a slightly more difficult problem:

EXACT-INDSET = {$(G,k)$ : the largest independent set in $G$ is of size $k$}

We see no reason for this problem to be in **NP**...

What would be a witness?

# Polynomial hierarchy

EXACT-INDSET = {$(G,k)$ : the largest independent set in $G$ is of size $k$}

A similar problem:

MIN−DNF = { ϕ : ϕ is a formula in the DNF form, not equivalent to any smaller formula in the DNF form}

$= \{ \phi : \forall \psi, |\psi| < |\phi| \Rightarrow \exists$ valuation $s$ such that $\phi(s) \neq \psi(s)\}$

In order to describe these problems, it is not enough to use one „exists" quantifier (as in **NP**), neither one „for all" quantifier (as in **coNP**). We have here a combination of two quantifiers.

# Polynomial hierarchy

EXACT-INDSET = {$(G,k)$ : the largest independent set in $G$ is of size $k$}

A similar problem:

MIN−DNF = { ϕ : ϕ is a formula in the DNF form, not equivalent to any smaller formula in the DNF form}

= { ϕ : ∀ ψ, |ψ| < |ϕ| ⇒ ∃ valuation $s$ such that ϕ(s)≠ψ(s)}

In order to describe these problems, it is not enough to use one „exists” quantifier (as in **NP**), neither one „for all” quantifier (as in **coNP**). We have here a combination of two quantifiers.

Class $\boldsymbol{\Sigma}_2^p$ contains languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \exists\, u \in \{0,1\}^{q(|x|)}\ \forall\, v \in \{0,1\}^{q(|x|)}\ M(x,u,v)=1$$

The language EXACT-INDSET is of this form:

$\exists S\ \forall S'$ . $S$ is an independent set of size $k$ and

$S'$ is not an independent set of size $>k$

# Polynomial hierarchy

Class $\mathbf{\Sigma}_2^p$ contains languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \exists\, u \in \{0,1\}^{q(|x|)}\ \forall\, v \in \{0,1\}^{q(|x|)}\ M(x,u,v)=1$$

The language EXACT-INDSET is of this form

Class $\mathbf{\Pi}_2^p$ contains complements of languages from $\mathbf{\Sigma}_2^p$; it is easy to see that it contains languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \forall\, u \in \{0,1\}^{q(|x|)}\ \exists\, v \in \{0,1\}^{q(|x|)}\ M(x,u,v)=1$$

The language EXACT-INDSET is of this form as well:

$\forall S'\ \exists S$ . $S$ is an independent set of size $k$ and

$S'$ is not an independent set of size $>k$

Also the language MIN−DNF is of this form:

$\forall\ \psi\ \exists s\ .\ |\psi| < |\phi| \Rightarrow \phi(s) \neq \psi(s)$

However, it is believed that MIN-DNF does not belong to $\mathbf{\Sigma}_2^p$

# Polynomial hierarchy

Class $\Sigma_2^p$ contains languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \exists\, u \in \{0,1\}^{q(|x|)}\ \forall\, v \in \{0,1\}^{q(|x|)}\ M(x,u,v)=1$$

<u>Fact</u>

Class $\Sigma_2^p$ contains precisely languages recognizable by nondetermi-nistic Turing machines with an oracle for SAT (or with an oracle for an arbitrary language in **NP**).

For this reason, the class is sometimes denoted $\mathbf{NP}^{\mathbf{NP}}$

Obviously $\Sigma_2^p$ contains all languages from **NP** and from **coNP**

# Polynomial hierarchy

Class $\mathbf{\Sigma}_k^p$ contains languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \exists u_1 \in \{0,1\}^{q(|x|)} \ \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Qu_k \in \{0,1\}^{q(|x|)} \,.\, M(x,u_1,\ldots,u_k)=1$$

Class $\mathbf{\Pi}_k^p$ contains complements of languages from $\mathbf{\Sigma}_k^p$, i.e., languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \forall u_1 \in \{0,1\}^{q(|x|)} \ \exists u_2 \in \{0,1\}^{q(|x|)} \ldots Qu_k \in \{0,1\}^{q(|x|)} \,.\, M(x,u_1,\ldots,u_k)=1$$

We also define **PH**$=\cup_k \mathbf{\Sigma}_k^p$

# Polynomial hierarchy

Class $\mathbf{\Sigma}^p_k$ contains languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \exists u_1 \in \{0,1\}^{q(|x|)} \; \forall u_2 \in \{0,1\}^{q(|x|)} \ldots Q u_k \in \{0,1\}^{q(|x|)} \,.\, M(x,u_1,\ldots,u_k)=1$$

Class $\mathbf{\Pi}^p_k$ contains complements of languages from $\mathbf{\Sigma}^p_k$, i.e., languages $L$ for which there is a machine $M$ working in polynomial time, and a polynomial $q$ such that:

$$x \in L \Leftrightarrow \forall u_1 \in \{0,1\}^{q(|x|)} \; \exists u_2 \in \{0,1\}^{q(|x|)} \ldots Q u_k \in \{0,1\}^{q(|x|)} \,.\, M(x,u_1,\ldots,u_k)=1$$

We also define **PH**$=\cup_k \mathbf{\Sigma}^p_k$

How are these classes related?

Fact 1: $\mathbf{\Sigma}^p_k \subseteq \mathbf{\Sigma}^p_{k+1}$, $\mathbf{\Sigma}^p_k \subseteq \mathbf{\Pi}^p_{k+1}$, $\mathbf{\Pi}^p_k \subseteq \mathbf{\Sigma}^p_{k+1}$, $\mathbf{\Pi}^p_k \subseteq \mathbf{\Pi}^p_{k+1}$

# Polynomial hierarchy

Fact 1: $\mathbf{\Sigma}^p_k \subseteq \mathbf{\Sigma}^p_{k+1}$, $\mathbf{\Sigma}^p_k \subseteq \mathbf{\Pi}^p_{k+1}$, $\mathbf{\Pi}^p_k \subseteq \mathbf{\Sigma}^p_{k+1}$, $\mathbf{\Pi}^p_k \subseteq \mathbf{\Pi}^p_{k+1}$

Proof

For $L \in \mathbf{\Sigma}^p_k$ we have a machine $M$ working in polynomial time, and a polynomial bound $q$ on the length of $u_1,\ldots,u_k$, such that:

$$x \in L \Leftrightarrow \exists u_1\ \forall u_2 \ldots Q u_k\ M(x,u_1,\ldots,u_k)=1$$

Consider $M'$ such that $M'(x,u_0,u_1,\ldots,u_k)=M(x,u_1,\ldots,u_k)$. Then

$$x \in L \Leftrightarrow \forall u_0\ \exists u_1\ \forall u_2 \ldots Q u_k\ M'(x,u_0,u_1,\ldots,u_k)=1$$

So $L \in \mathbf{\Pi}^p_{k+1}$

Consider $M''$ such that $M''(x,u_1,\ldots,u_k,u_{k+1})=M(x,u_1,\ldots,u_k)$. Then

$$x \in L \Leftrightarrow \exists u_1\ \forall u_2 \ldots Q u_k\ \overline{Q} u_{k+1}\ M''(x,u_1,\ldots,u_k,u_{k+1})=1$$

So $L \in \mathbf{\Sigma}^p_{k+1}$

Similarly we proceed for $L \in \mathbf{\Pi}^p_k$

# Polynomial hierarchy

Fact 1: $\Sigma^p_k \subseteq \Sigma^p_{k+1}$, $\Sigma^p_k \subseteq \Pi^p_{k+1}$, $\Pi^p_k \subseteq \Sigma^p_{k+1}$, $\Pi^p_k \subseteq \Pi^p_{k+1}$

$$\begin{tikzcd}
 & \mathbf{NP}=\Sigma^p_1 \arrow[r, phantom, "\subseteq"] \arrow[dr, phantom, "\subseteq"] & \Sigma^p_2 \arrow[r, phantom, "\subseteq"] \arrow[dr, phantom, "\subseteq"] & \Sigma^p_3 \arrow[r, phantom, "\subseteq"] \arrow[dr, phantom, "\subseteq"] & \cdots \\
\mathbf{P} \arrow[ur, phantom, "\subseteq" sloped] \arrow[dr, phantom, "\subseteq" sloped] & & & & \\
 & \mathbf{coNP}=\Pi^p_1 \arrow[r, phantom, "\subseteq"] \arrow[ur, phantom, "\subseteq"] & \Pi^p_2 \arrow[r, phantom, "\subseteq"] \arrow[ur, phantom, "\subseteq"] & \Pi^p_3 \arrow[r, phantom, "\subseteq"] \arrow[ur, phantom, "\subseteq"] & \cdots
\end{tikzcd}$$

Are these inclusions strict? And how are $\Sigma^p_k$ and $\Pi^p_k$ related?

# Polynomial hierarchy

Fact 1: $\Sigma^p_k \subseteq \Sigma^p_{k+1}$, $\Sigma^p_k \subseteq \Pi^p_{k+1}$, $\Pi^p_k \subseteq \Sigma^p_{k+1}$, $\Pi^p_k \subseteq \Pi^p_{k+1}$

$$\begin{array}{ccccccccc} & & \mathbf{NP}=\Sigma^p_1 & \subseteq & \Sigma^p_2 & \subseteq & \Sigma^p_3 & \subseteq & \cdots \\ \mathbf{P} & \subseteq & & & & & & & \\ & & \mathbf{coNP}=\Pi^p_1 & \subseteq & \Pi^p_2 & \subseteq & \Pi^p_3 & \subseteq & \cdots \end{array}$$

Are these inclusions strict? And how are $\Sigma^p_k$ and $\Pi^p_k$ related?

We don't know (it is believed that all these classes are different).

But there are only two possibilities:

- either all the classes are different, or
- they are different to some point, and then they start to be equal

Fact 2:

If $\Sigma^p_k = \Pi^p_k$, then $\Sigma^p_k = \Sigma^p_{k+1} = \ldots = \Pi^p_k = \Pi^p_{k+1} = \ldots = \mathbf{PH}$.

If $\mathbf{P} = \mathbf{NP}$, then $\mathbf{P} = \Sigma^p_1 = \Sigma^p_2 = \ldots = \Pi^p_1 = \Pi^p_2 = \ldots = \mathbf{PH}$.

# Polynomial hierarchy

Fact 2:

If $\Sigma^p_k=\Pi^p_k$, then $\Sigma^p_k=\Sigma^p_{k+1}=\ldots=\Pi^p_k=\Pi^p_{k+1}=\ldots=$ **PH**.

If **P**=**NP**, then **P**$=\Sigma^p_1=\Sigma^p_2=\ldots=\Pi^p_1=\Pi^p_2=\ldots=$ **PH**.

Proof (first part, the second part is analogous):

Consider a language $L$ in **PH**. It is in some $\Sigma^p_n$, where $n>k$. There is a machine $M$ working in polynomial time, and a polynomial bound on the length of $u_1,\ldots,u_n$, such that (suppose that $n,k$ even):

$$x\in L \Leftrightarrow \exists u_1 \forall u_2 \ldots \exists u_{n-k-1} \forall u_{n-k} \exists u_{n-k+1} \forall u_{n-k+2} \ldots \exists u_{n-1} \forall u_n\ M(x,u_1,\ldots,u_n)=1$$

Consider now the language

$$L'=\{(x,u_1,\ldots,u_k) : \exists u_{n-k+1} \forall u_{n-k+2} \ldots \exists u_{n-1} \forall u_n\ M(x,u_1,\ldots,u_n)=1\}$$

We have $L'\in\Sigma^p_k=\Pi^p_k$, so $L'$ is of the form (for some $M'$):

$$L'=\{(x,u_1,\ldots,u_k) : \forall u_{n-k+1} \exists u_{n-k+2} \ldots \forall u_{n-1} \exists u_n\ M'(x,u_1,\ldots,u_n)=1\}$$

This means that

$$x\in L \Leftrightarrow \exists u_1 \forall u_2 \ldots \exists u_{n-k-1} \forall u_{n-k} \forall u_{n-k+1} \exists u_{n-k+2} \ldots \forall u_{n-1} \exists u_n\ M'(x,u_1,\ldots,u_n)=1$$

We can merge $u_{n-k}$ and $u_{n-k+1}$ to a single word (longer twice),

so $L\in\Sigma^p_{n-1}$

# Polynomial hierarchy

$$\begin{array}{ccccccccc}
 & \rotatebox{45}{$\subseteq$} & \mathbf{NP}=\Sigma_1^p & \subseteq & \Sigma_2^p & \subseteq & \Sigma_3^p & \subseteq & \cdots \\
\mathbf{P} & & & \rotatebox{-45}{$\subseteq$}\ \rotatebox{45}{$\subseteq$} & & \rotatebox{-45}{$\subseteq$}\ \rotatebox{45}{$\subseteq$} & & \rotatebox{-45}{$\subseteq$}\ \rotatebox{45}{$\subseteq$} & \\
 & \rotatebox{-45}{$\subseteq$} & \mathbf{coNP}=\Pi_1^p & \subseteq & \Pi_2^p & \subseteq & \Pi_3^p & \subseteq & \cdots
\end{array}$$

There are only two possibilities:

- either all the classes are different, or
- they are different to some point, and then they start to be equal

Complete language in $\Sigma_k^p$?

Input: a sentence of the following form (with $k$ blocks of quantifiers)

$$\exists x_{11},\ldots,x_{1n} \forall x_{21},\ldots,x_{2n} \exists x_{21},\ldots,x_{2n} \ldots Q x_{k1},\ldots,x_{kn}\ \phi(x_{11},\ldots,x_{kn})$$

Question: is the sentence true?

# Polynomial hierarchy

$$\begin{array}{cccccccc}
 & & \mathbf{NP}=\Sigma_1^p & \subseteq & \Sigma_2^p & \subseteq & \Sigma_3^p & \subseteq & \cdots \\
 & \rotatebox{45}{$\subseteq$} & & & & & & & \\
\mathbf{P} & & & \rotatebox{-45}{$\subseteq$}\;\rotatebox{45}{$\subseteq$} & & \rotatebox{-45}{$\subseteq$}\;\rotatebox{45}{$\subseteq$} & & \rotatebox{-45}{$\subseteq$}\;\rotatebox{45}{$\subseteq$} & \\
 & \rotatebox{-45}{$\subseteq$} & & & & & & & \\
 & & \mathbf{coNP}=\Pi_1^p & \subseteq & \Pi_2^p & \subseteq & \Pi_3^p & \subseteq & \cdots
\end{array}$$

There are only two possibilities:

- either all the classes are different, or
- they are different to some point, and then they start to be equal

Complete language in $\Sigma_k^p$?

Input: a sentence of the following form (with $k$ blocks of quantifiers)

$$\exists x_{11},\ldots,x_{1n}\forall x_{21},\ldots,x_{2n}\exists x_{21},\ldots,x_{2n}\ldots Qx_{k1},\ldots,x_{kn}\ \phi(x_{11},\ldots,x_{kn})$$

Question: is the sentence true? (similarly for $\Pi_k^p$)

Complete language in **PH**?

Fact 3:

If there exists a **PH**-complete language, then $\mathbf{PH}=\Sigma_k^p$ for some $k$

Proof – The **PH**-complete language belongs to some $\Sigma_k^p$, and $\Sigma_k^p$ is closed under reductions in polynomial time.

# Polynomial hierarchy

Fact 1: $\Sigma_k^p \subseteq \Sigma_{k+1}^p$, $\Sigma_k^p \subseteq \Pi_{k+1}^p$, $\Pi_k^p \subseteq \Sigma_{k+1}^p$, $\Pi_k^p \subseteq \Pi_{k+1}^p$

Fact 2:

If $\Sigma_k^p = \Pi_k^p$, then $\Sigma_k^p = \Sigma_{k+1}^p = \ldots = \Pi_k^p = \Pi_{k+1}^p = \ldots =$ **PH**.

If **P**=**NP**, then **P**$=\Sigma_1^p = \Sigma_2^p = \ldots = \Pi_1^p = \Pi_2^p = \ldots =$ **PH**.

Fact 3:

If there exists a **PH**-complete language, then **PH**$=\Sigma_k^p$ for some $k$

Fact 4: **PH**$\subseteq$**PSPACE**

Proof: The $\Sigma_k^p$-complete language mentioned above is a special case of QBF, which belongs to **PSPACE**.

# Polynomial hierarchy

Fact 1: $\Sigma^p_k \subseteq \Sigma^p_{k+1}$, $\Sigma^p_k \subseteq \Pi^p_{k+1}$, $\Pi^p_k \subseteq \Sigma^p_{k+1}$, $\Pi^p_k \subseteq \Pi^p_{k+1}$

Fact 2:

If $\Sigma^p_k = \Pi^p_k$, then $\Sigma^p_k = \Sigma^p_{k+1} = \ldots = \Pi^p_k = \Pi^p_{k+1} = \ldots =$ **PH**.

If **P**=**NP**, then **P**$= \Sigma^p_1 = \Sigma^p_2 = \ldots = \Pi^p_1 = \Pi^p_2 = \ldots =$ **PH**.

Fact 3:

If there exists a **PH**-complete language, then **PH**$= \Sigma^p_k$ for some $k$

Fact 4: **PH**$\subseteq$**PSPACE**

Fact 5: If the classes $\Sigma^p_k$ are all different, then **PH**$\neq$**PSPACE**

Proof: Follows from Fact 3 – in **PSPACE** there is a complete language.

# Alternating machines

- Alternating Turing machines (ATM) generalize nondeterministic ones (NTM)
- Both NTM and ATM are not a realistic model of computation (we cannot build such machines). But NTM help us to observe a very natural phenomenon: a difference between finding a solution and verifying a solution.
- ATMs have a similar role for some languages, for which there are no short witnesses, i.e., which cannot be characterized using nondeterminism.

# Alternating machines

Definition of ATM:

- a configuration can have multiple successors (as in NTM)
- additionally states of the machine (and in effect its configurations) are divided to existential and universal ones

# Alternating machines

Definition of ATM:

- a configuration can have multiple successors (as in NTM)
- additionally states of the machine (and in effect its configurations) are divided to existential and universal ones

The set of wining configurations is defined as the smallest set s.t.:

- accepting configurations are winning
- every existential configuration, whose some successor is winning, is also winning
- every universal configuration, whose all successors are winning, is also winning

We accept a word $w$, if the initial configuration for this word is winning.

$M$ works in time $T(n)$ / in space $S(n)$, if every computation fits in this time / space.

# Alternating machines

Definition of ATM:

- a configuration can have multiple successors (as in NTM)
- additionally states of the machine (and in effect its configurations) are divided to existential and universal ones

The set of wining configurations is defined as the smallest set s.t.:

- accepting configurations are winning
- every existential configuration, whose some successor is winning, is also winning
- every universal configuration, whose all successors are winning, is also winning

We accept a word $w$, if the initial configuration for this word is winning.

$M$ works in time $T(n)$ / in space $S(n)$, if every computation fits in this time / space.

Observation:
NTM is a special case of an ATM – only existential states

# Alternating machines

Equivalently: acceptance can be defined using a game:

- we consider the configuration graph (edges = possible transitions)
- players ∃ and ∀ alternatingly move a pawn (common to both player) around the graph
- in existential states player ∃ decides, in universal states player ∀ decides (player ∃ wants to accept, player ∀ wants to reject)
- we accept a word, if player ∃ has a winning strategy – he can reach an accepting configuration regardless moves of player ∀