Nondeterministic polynomial time

Lecture 22

Thursday, November 26, 2020

LaTeXed: October 22, 2020 15:15

22.1
Review

22.1.1
Review: Polynomial reductions

Polynomial-time Reduction

Definition 22.1.

$X \leq_P Y$: **polynomial time reduction** from a decision problem X to a decision problem Y is an algorithm $\mathcal{A}$ such that:

1. Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
2. $\mathcal{A}$ runs in time polynomial in $|I_X|$. ($|I_Y| =$ size of I_Y).
3. Answer to I_X YES $\iff$ answer to I_Y is YES.

Polynomial-time Reduction

Definition 22.1.

$X \leq_P Y$: **polynomial time reduction** from a decision problem X to a decision problem Y is an algorithm $\mathcal{A}$ such that:

1. Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
2. $\mathcal{A}$ runs in time polynomial in $|I_X|$. ($|I_Y|$ = size of I_Y).
3. Answer to I_X YES $\iff$ answer to I_Y is YES.

Proposition 22.2.

If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.

Polynomial-time Reduction

Definition 22.1.

$X \leq_P Y$: **polynomial time reduction** from a decision problem X to a decision problem Y is an algorithm $\mathcal{A}$ such that:

1. Given an instance I_X of X, $\mathcal{A}$ produces an instance I_Y of Y.
2. $\mathcal{A}$ runs in time polynomial in $|I_X|$. ($|I_Y|$ = size of I_Y).
3. Answer to I_X YES $\iff$ answer to I_Y is YES.

Proposition 22.2.

If $X \leq_P Y$ then a polynomial time algorithm for Y implies a polynomial time algorithm for X.

This is a **Karp reduction**.

Composing polynomials...

A quick reminder

1. f and g monotone increasing. Assume that:
   1. $f(n) \leq a * n^b$ (i.e., $f(n) = O(n^b)$)
   2. $g(n) \leq c * n^d$ (i.e., $g(n) = O(n^d)$)

   a, b, c, d: constants.
2. $g\left(f(n)\right) \leq g\left(a * n^b\right) \leq c * \left(a * n^b\right)^d \leq c \cdot a^d * n^{bd}$
3. $\implies g(f(n)) = O\left(n^{bd}\right)$ is a polynomial.
4. **Conclusion:** Composition of two polynomials, is a polynomial.

Composing polynomials...

A quick reminder

1. f and g monotone increasing. Assume that:
   1. $f(n) \leq a * n^b$ (i.e., $f(n) = O(n^b)$)
   2. $g(n) \leq c * n^d$ (i.e., $g(n) = O(n^d)$)

   a, b, c, d: constants.
2. $g\Big(f(n)\Big) \leq g\left(a * n^b\right) \leq c * \left(a * n^b\right)^d \leq c \cdot a^d * n^{bd}$
3. $\implies g(f(n)) = O\left(n^{bd}\right)$ is a polynomial.
4. **Conclusion:** Composition of two polynomials, is a polynomial.

Composing polynomials...

A quick reminder

1. f and g monotone increasing. Assume that:
    1. $f(n) \le a * n^b$ (i.e., $f(n) = O(n^b)$)
    2. $g(n) \le c * n^d$ (i.e., $g(n) = O(n^d)$)

    a, b, c, d: constants.
2. $g\left(f(n)\right) \le g\left(a * n^b\right) \le c * \left(a * n^b\right)^d \le c \cdot a^d * n^{bd}$
3. $\implies g(f(n)) = O\left(n^{bd}\right)$ is a polynomial.
4. **Conclusion:** Composition of two polynomials, is a polynomial.

Composing polynomials...

A quick reminder

1. f and g monotone increasing. Assume that:
    1. $f(n) \leq a * n^b$ (i.e., $f(n) = O(n^b)$)
    2. $g(n) \leq c * n^d$ (i.e., $g(n) = O(n^d)$)

   a, b, c, d: constants.
2. $g\Big(f(n)\Big) \leq g\left(a * n^b\right) \leq c * \left(a * n^b\right)^d \leq c \cdot a^d * n^{bd}$
3. $\implies g(f(n)) = O\left(n^{bd}\right)$ is a polynomial.
4. **Conclusion:** Composition of two polynomials, is a polynomial.

Composing polynomials...

A quick reminder

1. f and g monotone increasing. Assume that:
   1. $f(n) \leq a * n^b$ (i.e., $f(n) = O(n^b)$)
   2. $g(n) \leq c * n^d$ (i.e., $g(n) = O(n^d)$)

   a, b, c, d: constants.
2. $g\Big(f(n)\Big) \leq g\big(a * n^b\big) \leq c * \big(a * n^b\big)^d \leq c \cdot a^d * n^{bd}$
3. $\implies g(f(n)) = O\Big(n^{bd}\Big)$ is a polynomial.
4. **Conclusion:** Composition of two polynomials, is a polynomial.

Composing polynomials...

A quick reminder

1. f and g monotone increasing. Assume that:
   1. $f(n) \leq a * n^b$ (i.e., $f(n) = O(n^b)$)
   2. $g(n) \leq c * n^d$ (i.e., $g(n) = O(n^d)$)

   a, b, c, d: constants.
2. $g\Big(f(n)\Big) \leq g\big(a * n^b\big) \leq c * \big(a * n^b\big)^d \leq c \cdot a^d * n^{bd}$
3. $\implies g(f(n)) = O\Big(n^{bd}\Big)$ is a polynomial.
4. **Conclusion:** Composition of two polynomials, is a polynomial.

Composing polynomials...

A quick reminder

1. f and g monotone increasing. Assume that:
   1. $f(n) \leq a * n^b$ (i.e., $f(n) = O(n^b)$)
   2. $g(n) \leq c * n^d$ (i.e., $g(n) = O(n^d)$)

   a, b, c, d: constants.
2. $g\Big(f(n)\Big) \leq g\left(a * n^b\right) \leq c * \left(a * n^b\right)^d \leq c \cdot a^d * n^{bd}$
3. $\implies g(f(n)) = O\Big(n^{bd}\Big)$ is a polynomial.
4. **Conclusion:** Composition of two polynomials, is a polynomial.

Transitivity of Reductions

Proposition 22.3.

$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.

1. Note: $X \leq_P Y$ does not imply that $Y \leq_P X$ and hence it is very important to know the FROM and TO in a reduction.
2. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y
3. ...show that an algorithm for Y implies an algorithm for X.

Transitivity of Reductions

Proposition 22.3.

$X \leq_P Y$ and $Y \leq_P Z$ implies that $X \leq_P Z$.

1. Note: $X \leq_P Y$ does not imply that $Y \leq_P X$ and hence it is very important to know the FROM and TO in a reduction.
2. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y
3. ...show that an algorithm for Y implies an algorithm for X.

Transitivity of Reductions

Proposition 22.3.

$X \leq_P Y$ *and* $Y \leq_P Z$ *implies that* $X \leq_P Z$.

1. Note: $X \leq_P Y$ does not imply that $Y \leq_P X$ and hence it is very important to know the FROM and TO in a reduction.
2. To prove $X \leq_P Y$ you need to show a reduction FROM X TO Y
3. ...show that an algorithm for Y implies an algorithm for X.

Polynomial time reduction...

Proving Correctness of Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ that:

1. Transforms an instance I_X of X into an instance I_Y of Y.
2. Satisfies the property that answer to I_X is YES iff I_Y is YES.
   1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
   2. typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_X is NO if answer to I_Y is NO).
3. Runs in **polynomial** time.

Polynomial time reduction...

Proving Correctness of Reductions

To prove that $X \leq_P Y$ you need to give an algorithm $\mathcal{A}$ that:

1. Transforms an instance I_X of X into an instance I_Y of Y.
2. Satisfies the property that answer to I_X is YES iff I_Y is YES.
   1. typical easy direction to prove: answer to I_Y is YES if answer to I_X is YES
   2. typical difficult direction to prove: answer to I_X is YES if answer to I_Y is YES (equivalently answer to I_X is NO if answer to I_Y is NO).
3. Runs in **polynomial** time.

THE END

...

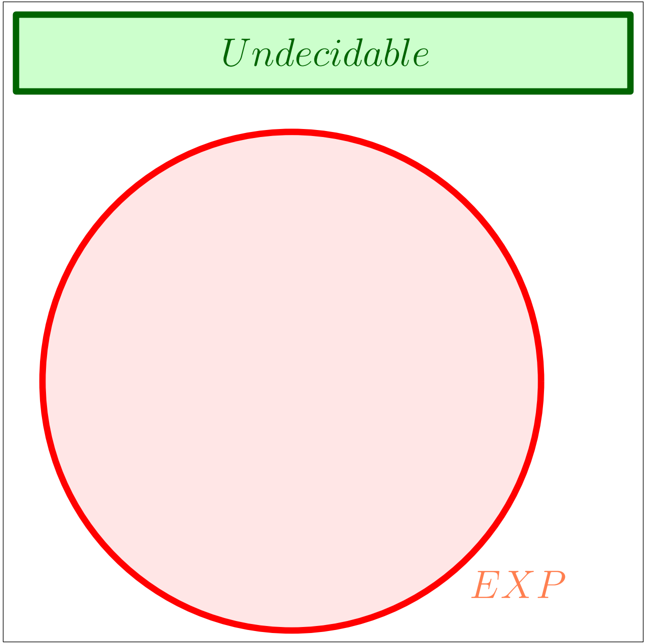
(for now)

22.1.2
A quick pre-review of complexity classes

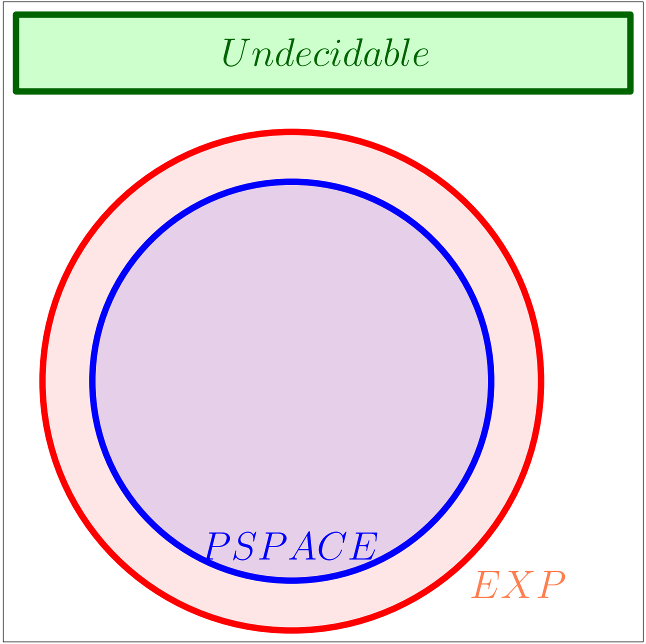
In the beginning...

In the beginning...

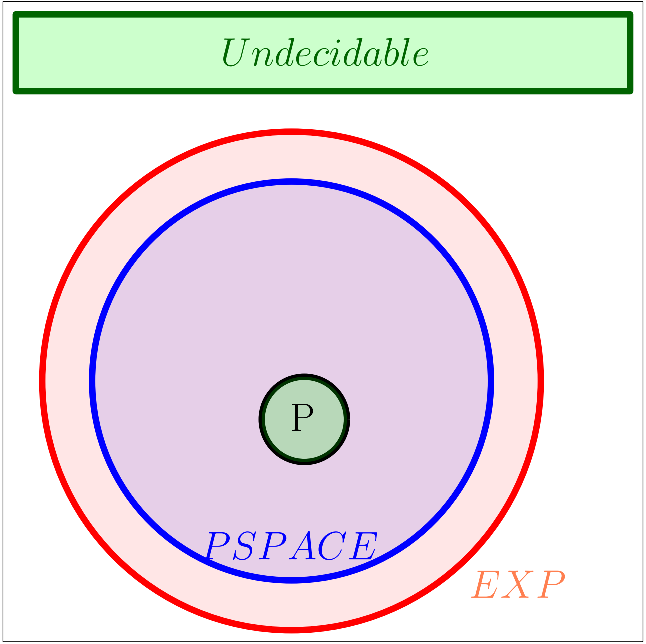
In the beginning...



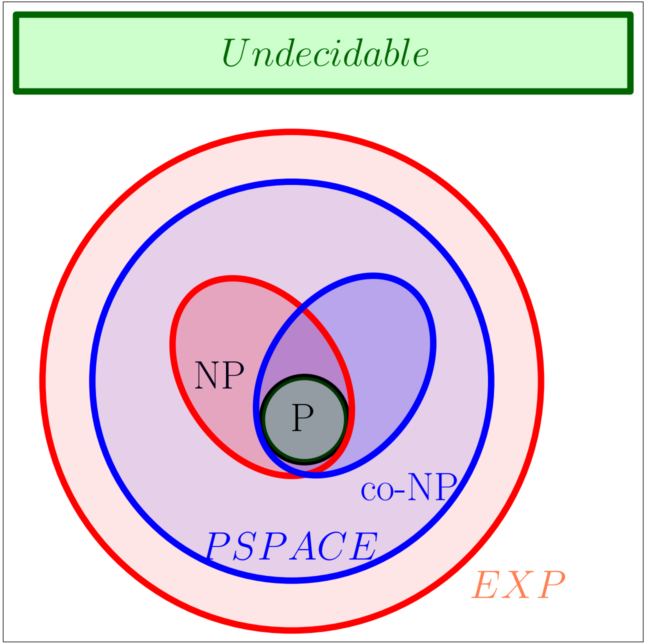
In the beginning...



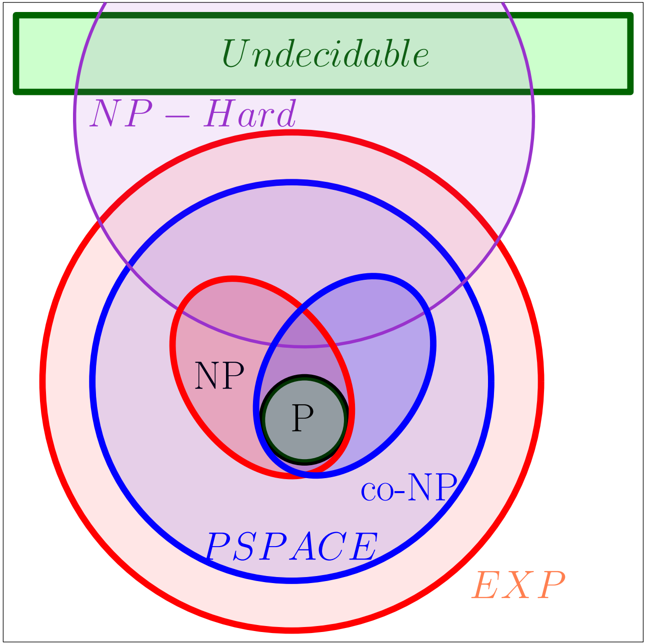
In the beginning...



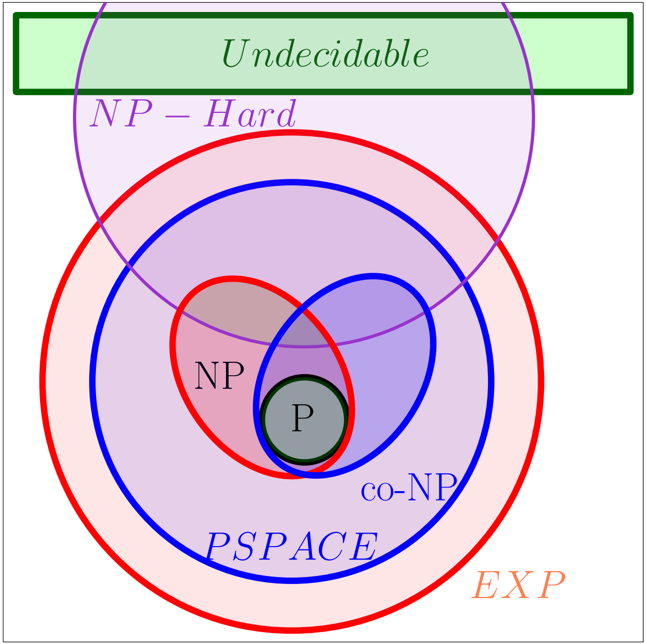
In the beginning...



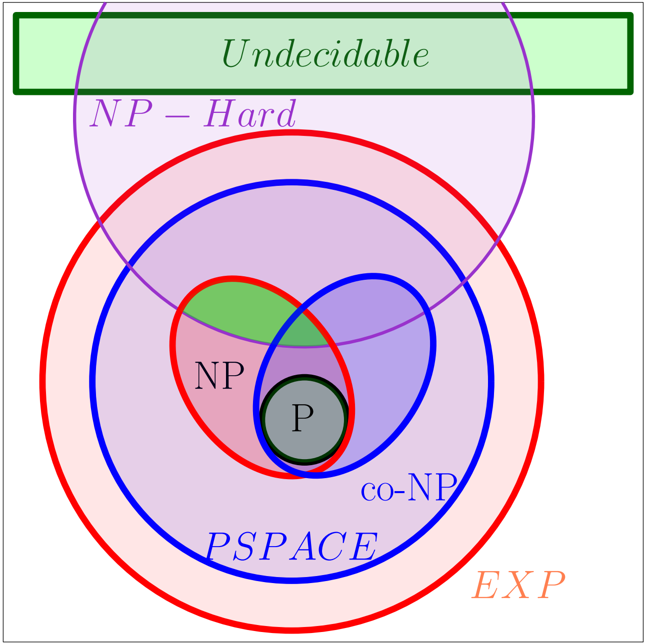
In the beginning...



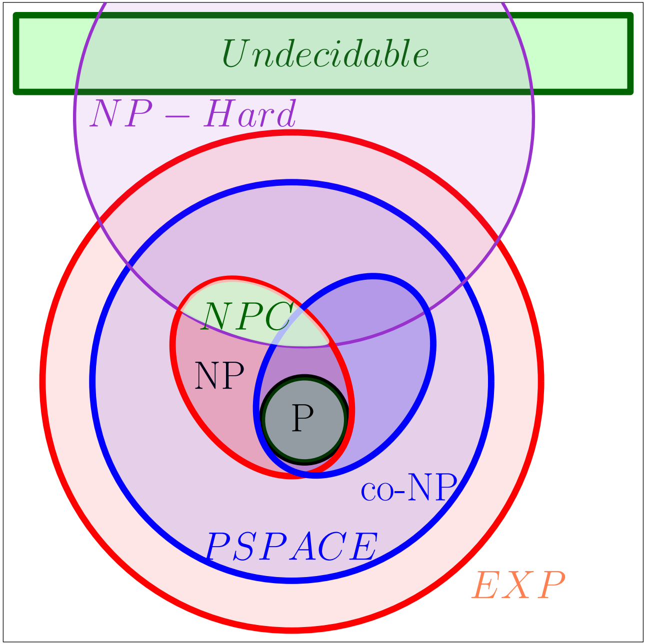
In the beginning...



In the beginning...



In the beginning...



THE END

...

(for now)

22.1.3
Polynomial equivalent problems: What do we know so far

What do we know so far

1. **Independent Set** $\leq_P$ **Clique**
   Clique $\leq_P$ **Independent Set**.
   $\implies$ Clique $\cong_P$ Independent Set.
2. Vertex Cover $\leq_P$ Independent Set
   Independent Set $\leq_P$ Vertex Cover.
   $\implies$ Independent Set $\cong_P$ Vertex Cover.
3. 3SAT $\leq_P$ SAT
   SAT $\leq_P$ 3SAT.
   $\implies$ 3SAT $\cong_P$ SAT.
4. Clique $\cong_P$ Independent Set $\cong_P$ Vertex Cover
   3SAT $\cong_P$ SAT.

What do we know so far

1. **Independent Set** $\le_P$ **Clique**
   Clique $\le_P$ **Independent Set**.
   $\implies$ **Clique** $\approxeq_P$ **Independent Set**.
2. **Vertex Cover** $\le_P$ **Independent Set**
   Independent Set $\le_P$ **Vertex Cover**.
   $\implies$ **Independent Set** $\approxeq_P$ **Vertex Cover**.
3. **3SAT** $\le_P$ **SAT**
   SAT $\le_P$ **3SAT**.
   $\implies$ **3SAT** $\approxeq_P$ **SAT**.
4. **Clique** $\approxeq_P$ **Independent Set** $\approxeq_P$ **Vertex Cover**
   3SAT $\approxeq_P$ **SAT**.

What do we know so far

1. **Independent Set** $\leq_P$ **Clique**
   Clique $\leq_P$ **Independent Set**.
   $\implies$ **Clique** $\cong_P$ **Independent Set**.
2. **Vertex Cover** $\leq_P$ **Independent Set**
   Independent Set $\leq_P$ **Vertex Cover**.
   $\implies$ **Independent Set** $\cong_P$ **Vertex Cover**.
3. **3SAT** $\leq_P$ **SAT**
   SAT $\leq_P$ **3SAT**.
   $\implies$ **3SAT** $\cong_P$ **SAT**.
4. **Clique** $\cong_P$ **Independent Set** $\cong_P$ **Vertex Cover**
   3SAT $\cong_P$ **SAT**.

What do we know so far

1. **Independent Set** $\leq_P$ **Clique**
   Clique $\leq_P$ **Independent Set**.
   $\implies$ **Clique** $\approxeq_P$ **Independent Set**.
2. **Vertex Cover** $\leq_P$ **Independent Set**
   Independent Set $\leq_P$ **Vertex Cover**.
   $\implies$ **Independent Set** $\approxeq_P$ **Vertex Cover**.
3. 3SAT $\leq_P$ SAT
   SAT $\leq_P$ 3SAT.
   $\implies$ 3SAT $\approxeq_P$ SAT.
4. Clique $\approxeq_P$ Independent Set $\approxeq_P$ Vertex Cover
   3SAT $\approxeq_P$ SAT.

What do we know so far

1. **Independent Set** $\leq_P$ **Clique**
   Clique $\leq_P$ **Independent Set**.
   $\implies$ **Clique** $\cong_P$ **Independent Set**.
2. **Vertex Cover** $\leq_P$ **Independent Set**
   Independent Set $\leq_P$ **Vertex Cover**.
   $\implies$ **Independent Set** $\cong_P$ **Vertex Cover**.
3. **3SAT** $\leq_P$ **SAT**
   SAT $\leq_P$ **3SAT**.
   $\implies$ **3SAT** $\cong_P$ **SAT**.
4. **Clique** $\cong_P$ **Independent Set** $\cong_P$ **Vertex Cover**
   3SAT $\cong_P$ **SAT**.

What do we know so far

1. **Independent Set** $\leq_P$ **Clique**
   Clique $\leq_P$ **Independent Set**.
   $\implies$ **Clique** $\cong_P$ **Independent Set**.
2. **Vertex Cover** $\leq_P$ **Independent Set**
   Independent Set $\leq_P$ **Vertex Cover**.
   $\implies$ **Independent Set** $\cong_P$ **Vertex Cover**.
3. **3SAT** $\leq_P$ **SAT**
   SAT $\leq_P$ **3SAT**.
   $\implies$ **3SAT** $\cong_P$ **SAT**.
4. **Clique** $\cong_P$ **Independent Set** $\cong_P$ **Vertex Cover**
   3SAT $\cong_P$ **SAT**.

What do we know so far

1. **Independent Set** $\leq_P$ **Clique**
   Clique $\leq_P$ **Independent Set**.
   $\implies$ **Clique** $\cong_P$ **Independent Set**.
2. **Vertex Cover** $\leq_P$ **Independent Set**
   Independent Set $\leq_P$ **Vertex Cover**.
   $\implies$ **Independent Set** $\cong_P$ **Vertex Cover**.
3. **3SAT** $\leq_P$ **SAT**
   SAT $\leq_P$ **3SAT**.
   $\implies$ **3SAT** $\cong_P$ **SAT**.
4. **Clique** $\cong_P$ **Independent Set** $\cong_P$ **Vertex Cover**
   3SAT $\cong_P$ **SAT**.

THE END

...

(for now)

22.2
NP: Nondeterministic polynomial time

22.2.1
Introduction

P and **NP** and Turing Machines

1. **P**: set of decision problems that have polynomial time algorithms.
2. **NP**: set of decision problems that have polynomial time non-deterministic algorithms.

- Many natural problems we would like to solve are in ***NP***.
- Every problem in ***NP*** has an exponential time algorithm
- $\boldsymbol{P} \subseteq \boldsymbol{NP}$
- Some problems in ***NP*** are in ***P*** (example, shortest path problem)

Big Question: Does every problem in ***NP*** have an efficient algorithm? Same as asking whether $\boldsymbol{P} = \boldsymbol{NP}$.

Problems with no known polynomial time algorithms

Problems

1. **Independent Set**
2. **Vertex Cover**
3. **Set Cover**
4. **SAT**
5. **3SAT**

There are of course undecidable problems (no algorithm at all!) but many problems that we want to solve are of similar flavor to the above.

Question: What is common to above problems?

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X of X there is a proof/certificate/solution that is of length poly($|I_X|$) such that given a proof one can efficiently check that I_X is indeed a YES instance.

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph G and k: a subset S of vertices.
3. **Homework**

Efficient Checkability

Above problems share the following feature:

Checkability

For any YES instance I_X of X there is a proof/certificate/solution that is of length poly($|I_X|$) such that given a proof one can efficiently check that I_X is indeed a YES instance.

Examples:

1. **SAT** formula φ: proof is a satisfying assignment.
2. **Independent Set** in graph G and k: a subset S of vertices.
3. **Homework**

Sudoku

			2	5				
	3	6		4		8		
	4					1	6	
2								
7	6						1	9
								3
	1	5					7	
		9		8		2	4	
				3	7			

Given $n \times n$ sudoku puzzle, does it have a solution?

Solution to the Sudoku example...

1	8	7	**2**	**5**	6	9	3	4
9	**3**	**6**	7	**4**	1	**8**	5	2
5	**4**	2	8	9	3	**1**	**6**	7
2	9	1	3	7	4	6	8	5
7	**6**	3	5	2	8	4	**1**	**9**
8	5	4	6	1	9	7	2	**3**
4	**1**	**5**	9	6	2	3	**7**	8
3	7	**9**	1	**8**	5	**2**	**4**	6
6	2	8	4	**3**	**7**	5	9	1

THE END

...

(for now)

22.2.2
Certifiers/Verifiers

Certifiers

Definition 22.1.

An algorithm $C(\cdot, \cdot)$ is a **certifier** for problem X if the following two conditions hold:

- For every $s \in X$ there is some string t such that $C(s, t) =$ "yes"
- If $s \notin X$, $C(s, t) =$ "no" for every t.

The string t is called a certificate or proof for s.

Efficient (polynomial time) Certifiers

Definition 22.2 (Efficient Certifier.).

A certifier C is an **efficient certifier** for problem X if there is a polynomial $p(\cdot)$ such that the following conditions hold:

- For every $s \in X$ there is some string t such that $C(s, t) =$ "yes" **and** $|t| \leq p(|s|)$ (proof is polynomially short)..
- If $s \notin X$, $C(s, t) =$ "no" for every t.
- $C(\cdot, \cdot)$ runs in polynomial time in the size of s.

Since $|t| = |s|^{O(1)}$, and certifier runs in polynomial time in $|s| + |t|$, it follows that certifier runs in polynomial time in the size of s.

Proposition 22.3.

If $s \in X$, then there exists a certificate t of polynomial length in s, such that $C(s, t)$ returns YES, and runs in polynomial time in $|s|$.

Efficient (polynomial time) Certifiers

Definition 22.2 (Efficient Certifier.).

A certifier C is an **efficient certifier** for problem X if there is a polynomial $p(\cdot)$ such that the following conditions hold:

- For every $s \in X$ there is some string t such that $C(s, t) =$ "yes" **and** $|t| \leq p(|s|)$ (proof is polynomially short)..
- If $s \notin X$, $C(s, t) =$ "no" for every t.
- $C(\cdot, \cdot)$ runs in polynomial time in the size of s.

Since $|t| = |s|^{O(1)}$, and certifier runs in polynomial time in $|s| + |t|$, it follows that certifier runs in polynomial time in the size of s.

Proposition 22.3.

If $s \in X$, then there exists a certificate t of polynomial length in s, such that $C(s, t)$ returns YES, and runs in polynomial time in $|s|$.

Example: Independent Set

1. Problem: Does $G = (V, E)$ have an independent set of size $\geq k$?
   1. Certificate: Set $S \subseteq V$.
   2. Certifier: Check $|S| \geq k$ and no pair of vertices in S is connected by an edge.

THE END

...

(for now)

22.2.3
Examples to problems with efficient certifiers

Example: Vertex Cover

1. Problem: Does G have a vertex cover of size $\leq k$?
   1. Certificate: $S \subseteq V$.
   2. Certifier: Check $|S| \leq k$ and that for every edge at least one endpoint is in S.

Example: **SAT**

1. Problem: Does formula φ have a satisfying truth assignment?
   1. Certificate: Assignment $\boldsymbol{a}$ of $\mathbf{0/1}$ values to each variable.
   2. Certifier: Check each clause under $\boldsymbol{a}$ and say "yes" if all clauses are true.

Example: Composites

Problem: Composite

Instance: A number s.
Question: Is the number s a composite?

1. Problem: **Composite**.
    1. Certificate: A factor $t \leq s$ such that $t \neq 1$ and $t \neq s$.
    2. Certifier: Check that t divides s.

Example: NFA Universality

Problem: NFA Universality

> **Instance:** Description of a NFA M.
> **Question:** Is $L(M) = \Sigma^*$, that is, does M accept all strings?

1. Problem: **NFA Universality**.
   1. Certificate: A DFA M' equivalent to M
   2. Certifier: Check that $L(M') = \Sigma^*$

Certifier is efficient but certificate is not necessarily short! We do not know if the problem is in NP.

Example: NFA Universality

Problem: NFA Universality

> **Instance:** Description of a NFA M.
> **Question:** Is $L(M) = \Sigma^*$, that is, does M accept all strings?

1. Problem: **NFA Universality**.
   1. Certificate: A DFA M' equivalent to M
   2. Certifier: Check that $L(M') = \Sigma^*$

Certifier is efficient but certificate is not necessarily short! We do not know if the problem is in NP.

Example: A String Problem

Problem: PCP

Instance: Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
Question: Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
   1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
   2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

PCP = Posts Correspondence Problem and it is undecidable!
Implies no finite bound on length of certificate!

Example: A String Problem

Problem: PCP

> **Instance:** Two sets of binary strings $\alpha_1, \ldots, \alpha_n$ and $\beta_1, \ldots, \beta_n$
> **Question:** Are there indices $i_1, i_2, \ldots, i_k$ such that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

1. Problem: **PCP**
    1. Certificate: A sequence of indices $i_1, i_2, \ldots, i_k$
    2. Certifier: Check that $\alpha_{i_1}\alpha_{i_2}\ldots\alpha_{i_k} = \beta_{i_1}\beta_{i_2}\ldots\beta_{i_k}$

PCP = Posts Correspondence Problem and it is undecidable!
Implies no finite bound on length of certificate!

THE END

...

(for now)

22.2.4
NP: Definition

Nondeterministic Polynomial Time

Definition 22.4.

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Nondeterministic Polynomial Time

Definition 22.4.

Nondeterministic Polynomial Time (denoted by **NP**) is the class of all problems that have efficient certifiers.

Example 22.5.

Independent Set, **Vertex Cover**, **Set Cover**, **SAT**, **3SAT**, and **Composite** are all examples of problems in **NP**.

Why is it called...

Nondeterministic Polynomial Time

A certifier is an algorithm $C(I, c)$ with two inputs:

1. I: instance.
2. c: proof/certificate that the instance is indeed a YES instance of the given problem.

One can think about C as an algorithm for the original problem, if:

1. Given I, the algorithm guesses (non-deterministically, and who knows how) a certificate c.
2. The algorithm now verifies the certificate c for the instance I.

NP can be equivalently described using Turing machines.

Asymmetry in Definition of NP

Note that only YES instances have a short proof/certificate. NO instances need not have a short certificate.

Example 22.6.

SAT formula φ. No easy way to prove that φ is NOT satisfiable!

More on this and **co-NP** later on.

THE END

...

(for now)

22.2.5
Intractability

P versus NP

Proposition 22.7.

$\mathbf{P} \subseteq \mathbf{NP}$.

For a problem in **P** no need for a certificate!

Proof.

Consider problem $X \in \mathbf{P}$ with algorithm A. Need to demonstrate that X has an efficient certifier:

1. Certifier C on input s, t, runs $A(s)$ and returns the answer.
2. C runs in polynomial time.
3. If $s \in X$, then for every t, $C(s, t) =$ "yes".
4. If $s \notin X$, then for every t, $C(s, t) =$ "no". □

P versus **NP**

Proposition 22.7.

P $\subseteq$ **NP**.

For a problem in **P** no need for a certificate!

Proof.

Consider problem $X \in$ **P** with algorithm A. Need to demonstrate that X has an efficient certifier:

1. Certifier C on input s, t, runs $A(s)$ and returns the answer.
2. C runs in polynomial time.
3. If $s \in X$, then for every t, $C(s, t) =$ "yes".
4. If $s \notin X$, then for every t, $C(s, t) =$ "no". □

Exponential Time

Definition 22.8.

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e., $O(2^{\mathrm{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O(2^{n^3})$, …

Exponential Time

Definition 22.8.

Exponential Time (denoted **EXP**) is the collection of all problems that have an algorithm which on input s runs in exponential time, i.e., $O(2^{\text{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O(2^{n^3})$, ...

NP versus EXP

Proposition 22.9.

NP $\subseteq$ **EXP**.

Proof.

Let $X \in$ **NP** with certifier C. Need to design an exponential time algorithm for X.

1. For every t, with $|t| \leq p(|s|)$ run $C(s, t)$; answer "yes" if any one of these calls returns "yes".
2. The above algorithm correctly solves X (exercise).
3. Algorithm runs in $O(q(|s| + |p(s)|)2^{p(|s|)})$, where q is the running time of C. □

Examples

1. **SAT**: try all possible truth assignment to variables.
2. **Independent Set**: try all possible subsets of vertices.
3. **Vertex Cover**: try all possible subsets of vertices.

Is **NP** efficiently solvable?

We know $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$.

Is **NP** efficiently solvable?

We know **P** $\subseteq$ **NP** $\subseteq$ **EXP**.

Big Question

Is there are problem in **NP** that does not belong to **P**? Is **P** $=$ **NP**?

If $\mathbf{P} = \mathbf{NP}$ . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce . . .
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If $\mathbf{P} = \mathbf{NP}$ . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce . . .
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If P = NP . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce . . .
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If $\mathbf{P} = \mathbf{NP}$ . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce . . .
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If $\mathbf{P} = \mathbf{NP}$ . . .

Or: If pigs could fly then life would be sweet.

1. Many important optimization problems can be solved efficiently.
2. The RSA cryptosystem can be broken.
3. No security on the web.
4. No e-commerce . . .
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

P versus NP

Status

Relationship between **P** and **NP** remains one of the most important open problems in mathematics/computer science.

Consensus: Most people feel/believe $\boldsymbol{P \neq NP}$.

Resolving **P** versus **NP** is a Clay Millennium Prize Problem. You can win a million dollars in addition to a Turing award and major fame!

Review question: If **P** $=$ **NP** this implies that...

(A) **Vertex Cover** can be solved in polynomial time.

(B) **P** $=$ **EXP**.

(C) **EXP** $\subseteq$ **P**.

(D) All of the above.

THE END

...

(for now)