NP Completeness

Lecture 3

September 3, 2015

3.1: Definition of NP

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Independent Set $\leq_P$ **Clique**

Recap . . .

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Independent Set $\leq_P$ **Clique**

Recap . . .

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Independent Set $\leq_P$ **Clique**$\leq_P$**Independent Set**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Independent Set $\approx_P$ **Clique**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Independent Set $\approx_P$ **Clique**
Independent Set $\leq_P$ **Vertex Cover**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Independent Set $\approx_P$ **Clique**

Independent Set $\leq_P$ **Vertex Cover** $\leq_P$ **Independent Set**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Independent Set $\approx_P$ **Clique**
Independent Set $\approx_P$ **Vertex Cover**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Vertex Cover $\approx_P$ **Independent Set** $\approx_P$**Clique**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Vertex Cover $\approx_P$ **Independent Set** $\approx_P$**Clique**
3SAT $\leq_P$**SAT**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Vertex Cover $\approx_P$ **Independent Set** $\approx_P$**Clique**
3SAT $\leq_P$**SAT**$\leq_P$**3SAT**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Vertex Cover $\approx_P$ **Independent Set** $\approx_P$**Clique**
3SAT $\approx_P$**SAT**

Recap ...

Problems

1. **Clique**
2. **Independent Set**
3. **Vertex Cover**

1. **Set Cover**
2. **SAT**
3. **3SAT**

Relationship

Vertex Cover $\approx_P$ **Independent Set** $\approx_P$**Clique**
3SAT $\approx_P$**SAT**
3SAT$\leq_P$**Independent Set**

3.1.1: Preliminaries

3.1.1.1: Problems and Algorithms

Problems and Algorithms: Formal Approach

Decision Problems

1. Problem Instance: Binary string s, with size $|s|$
2. Problem: Set X of strings s.t. answer is "yes": members of X are **YES instances** of X.
   Strings not in X are **NO instances** of X.

Definition

1. **alg**: algorithm for problem X if $\textbf{alg}(s) = \text{"yes"} \iff s \in X$.
2. **alg** have polynomial running time $\exists p(\cdot)$ polynomial s.t. $\forall s$, $\textbf{alg}(s)$ terminates in at most $O\left(p(|s|)\right)$ steps.

Problems and Algorithms: Formal Approach

Decision Problems

1. Problem Instance: Binary string s, with size $|s|$
2. Problem: Set X of strings s.t. answer is "yes": members of X are **YES instances** of X.
   Strings not in X are **NO instances** of X.

Definition

1. **alg**: algorithm for problem X if $\mathbf{alg}(s) =$ "yes" $\iff s \in X$.
2. **alg** have polynomial running time $\exists p(\cdot)$ polynomial s.t. $\forall s$, $\mathbf{alg}(s)$ terminates in at most $O\Big(p(|s|)\Big)$ steps.

Polynomial Time

Definition

Polynomial time (denoted by **P**): class of all (decision) problems that have an algorithm that solves it in polynomial time.

Polynomial Time

Definition

Polynomial time (denoted by **P**): class of all (decision) problems that have an algorithm that solves it in polynomial time.

Example

Problems in **P** include

1. Is there a shortest path from s to t of length $\leq k$ in **G**?
2. Is there a flow of value $\geq k$ in network **G**?
3. Is there an assignment to variables to satisfy given linear constraints?

Efficiency Hypothesis

Efficiency hypothesis.

A problem X has an efficient algorithm
$\iff$ $X \in$ **P**, *that is X has a polynomial time algorithm*.

1. Justifications:
   1. Robustness of definition to variations in machines.
   2. A sound theoretical definition.
   3. Most known polynomial time algorithms for "natural" problems have small polynomial running times.

Efficiency Hypothesis

Efficiency hypothesis.

A problem X has an efficient algorithm
$\iff$ $X \in \mathbf{P}$, *that is X has a polynomial time algorithm*.

1. Justifications:
   1. Robustness of definition to variations in machines.
   2. A sound theoretical definition.
   3. Most known polynomial time algorithms for "natural" problems have small polynomial running times.

Efficiency Hypothesis

Efficiency hypothesis.

A problem X has an efficient algorithm
$\iff$ $X \in \mathbf{P}$*, that is X has a polynomial time algorithm*.

1. Justifications:
   1. Robustness of definition to variations in machines.
   2. A sound theoretical definition.
   3. Most known polynomial time algorithms for "natural" problems have small polynomial running times.

Efficiency Hypothesis

Efficiency hypothesis.

A problem $\boldsymbol{X}$ has an efficient algorithm
$\iff$ $\boldsymbol{X} \in \mathbf{P}$, *that is $\boldsymbol{X}$ has a polynomial time algorithm*.

1. Justifications:
   1. Robustness of definition to variations in machines.
   2. A sound theoretical definition.
   3. Most known polynomial time algorithms for "natural" problems have small polynomial running times.

Problems that are hard...

...with no known polynomial time algorithms

Problems

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

1. undecidable problems are way harder (no algorithm at all!)
2. ...but many problems want to solve: similar to above.
3. Question: What is common to above problems?

Problems that are hard...

...with no known polynomial time algorithms

Problems

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

1. undecidable problems are way harder (no algorithm at all!)
2. ...but many problems want to solve: similar to above.
3. Question: What is common to above problems?

Problems that are hard...

...with no known polynomial time algorithms

Problems

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

1. undecidable problems are way harder (no algorithm at all!)
2. ...but many problems want to solve: similar to above.
3. Question: What is common to above problems?

Problems that are hard...

...with no known polynomial time algorithms

Problems

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

1. undecidable problems are way harder (no algorithm at all!)
2. ...but many problems want to solve: similar to above.
3. Question: What is common to above problems?

Efficient Checkability

1. Above problems have the property:

Checkability

For any YES instance I_X of X:

(A) there is a proof (or certificate) C.

(B) Length of certificate $|C| \leq \mathrm{poly}(|I_X|)$.

(C) Given C, I_x: efficiently check that I_X is YES instance.

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

Efficient Checkability

1. Above problems have the property:

Checkability

For any YES instance I_X of X:

(A) there is a proof (or certificate) C.

(B) Length of certificate $|C| \leq \mathbf{poly}(|I_X|)$.

(C) Given C, I_x: efficiently check that I_X is YES instance.

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

Efficient Checkability

1. Above problems have the property:

Checkability

For any YES instance I_X of X:

(A) there is a proof (or certificate) C.

(B) Length of certificate $|C| \leq \mathrm{poly}(|I_X|)$.

(C) Given C, I_x: efficiently check that I_X is YES instance.

2. Examples:
   1. **SAT** formula φ: proof is a satisfying assignment.
   2. **Independent Set** in graph $\mathbf{G}$ and k:
      Certificate: a subset S of vertices.

Efficient Checkability

1. Above problems have the property:

Checkability

For any YES instance I_X of X:

(A) there is a proof (or certificate) C.

(B) Length of certificate $|C| \leq \mathrm{poly}(|I_X|)$.

(C) Given C, I_x: efficiently check that I_X is YES instance.

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

Efficient Checkability

1. Above problems have the property:

Checkability

For any YES instance I_X of X:

(A) there is a proof (or certificate) C.

(B) Length of certificate $|C| \leq \mathbf{poly}(|I_X|)$.

(C) Given C, I_x: efficiently check that I_X is YES instance.

2. Examples:
   1. **SAT** formula φ: proof is a satisfying assignment.
   2. **Independent Set** in graph **G** and k:

      Certificate: a subset S of vertices.

Efficient Checkability

1. Above problems have the property:

Checkability

For any YES instance I_X of X:
  (A) there is a proof (or certificate) C.
  (B) Length of certificate $|C| \leq \mathbf{poly}(|I_X|)$.
  (C) Given C, I_x: efficiently check that I_X is YES instance.

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

3.1.2: Certifiers/Verifiers

Certifiers

Definition

Algorithm $C(\cdot,\cdot)$ is **certifier** for problem X: $\forall s \in X$ there $\exists t$ such that $C(s,t) =$ "YES", and conversely, if for some s and t, $C(s,t) =$ "yes" then $s \in X$.

t is the certificate or proof for s.

Certifiers

Definition

Algorithm $C(\cdot, \cdot)$ is **certifier** for problem X: $\forall s \in X$ there $\exists t$ such that $C(s, t) =$ "YES", and conversely, if for some s and t, $C(s, t) =$ "yes" then $s \in X$.

t is the certificate or proof for s.

Certifiers

Definition

Algorithm $C(\cdot,\cdot)$ is **certifier** for problem X: $\forall s \in X$ there $\exists t$ such that $C(s,t) = $"YES", and conversely, if for some s and t, $C(s,t) = $"yes" then $s \in X$.

t is the certificate or proof for s.

Definition (Efficient Certifier.)

Certifier C is **efficient certifier** for X if there is a polynomial $p(\cdot)$ s.t. for every string s:

- $\star$ $s \in X$ if and only if
- $\star$ there is a string t:
    1. $|t| \leq p(|s|)$,
    2. $C(s,t) = $"yes",
    3. and C runs in polynomial time.

Example: Independent Set

1. Problem: Does $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ 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.

3.1.3: Examples

Example: Vertex Cover

1. Problem: Does $\mathbf{G}$ 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 a of $0/1$ values to each variable.
   2. Certifier: Check each clause under a and say "yes" if all clauses are true.

Example:Composites

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.

3.2: $\boldsymbol{NP}$

3.2.1: Definition

Nondeterministic Polynomial Time

Definition

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

Nondeterministic Polynomial Time

Definition

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

Example

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

Why is it called...

Nondeterministic Polynomial Time

1. Certifier is 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.
2. C "algorithm" for original problem, if:
    1. Given I, the algorithm guess (non-deterministically, and who knows how) the certificate c.
    2. Algorithm verifies certificate c for the instance I.
3. Usually **NP** is described using Turing machines (gag).

Why is it called...

Nondeterministic Polynomial Time

1. Certifier is 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.
2. C "algorithm" for original problem, if:
   1. Given I, the algorithm guess (non-deterministically, and who knows how) the certificate c.
   2. Algorithm verifies certificate c for the instance I.
3. Usually **NP** is described using Turing machines (gag).

Why is it called...

Nondeterministic Polynomial Time

1. Certifier is 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.
2. C "algorithm" for original problem, if:
    1. Given I, the algorithm guess (non-deterministically, and who knows how) the certificate c.
    2. Algorithm verifies certificate c for the instance I.
3. Usually **NP** is described using Turing machines (gag).

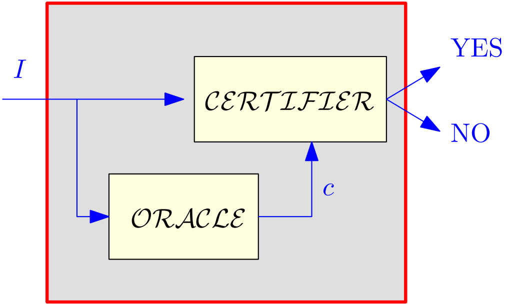
Why is it called...

Nondeterministic Polynomial Time

1. Certifier is 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.
2. C "algorithm" for original problem, if:
    1. Given I, the algorithm guess (non-deterministically, and who knows how) the certificate c.
    2. Algorithm verifies certificate c for the instance I.
3. Usually **NP** is described using Turing machines (gag).

Certifiers as algorithms...

...with a little help from an oracle friend.



1. Oracle: Guesses certificate c for given instance I.
2. Certifier: Polynomial time, given I and c, verify that indeed c proves that I is a YES instance.

Asymmetry in Definition of NP

1. Only YES instances have a short proof/certificate. NO instances need not have a short certificate.

2. For example...

Example

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

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

Asymmetry in Definition of NP

1. Only YES instances have a short proof/certificate. NO instances need not have a short certificate.
2. For example...

Example

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

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

Asymmetry in Definition of NP

1. Only YES instances have a short proof/certificate. NO instances need not have a short certificate.
2. For example...

Example

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

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

3.2.2: Intractability

P versus **NP**

Proposition

P $\subseteq$ **NP**.

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

Proof.

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

1. Certifier C (input s, t):
   runs **alg**(s) and returns its 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". $\square$

P versus NP

Proposition

P $\subseteq$ **NP**.

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

Proof.

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

1. Certifier C (input s, t):
   runs **alg**(s) and returns its answer.
2. C runs in polynomial time.
3. If $s \in X$, then for every t, $C(s,t) = \text{"YES"}$.
4. If $s \notin X$, then for every t, $C(s,t) = \text{"NO"}$. $\square$

P versus NP

Proposition

P $\subseteq$ **NP**.

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

Proof.

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

1. Certifier C (input s, t):
   runs **alg**(s) and returns its 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". □

$\mathbf{P}$ versus $\mathbf{NP}$

Proposition

P $\subseteq$ **NP**.

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

Proof.

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

1. Certifier C (input s, t):
   runs **alg**(s) and returns its 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".

$\mathbf{P}$ versus $\mathbf{NP}$

Proposition

P $\subseteq$ **NP**.

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

Proof.

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

1. Certifier C (input s, t):
   runs **alg**(s) and returns its 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

P $\subseteq$ **NP**.

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

Proof.

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

1. Certifier C (input s, t):
   runs **alg**(s) and returns its 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". $\square$

Exponential Time

Definition

Exponential Time (denoted **EXP**) set of all problems with algorithm that runs in exponential time.
For input s: Running time is $O(2^{\mathrm{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O\left(2^{n^3}\right)$, $\ldots$

Exponential Time

Definition

Exponential Time (denoted **EXP**) set of all problems with algorithm that runs in exponential time.
For input s: Running time is $O(2^{\mathrm{poly}(|s|)})$.

Example: $O(2^n)$, $O(2^{n \log n})$, $O\left(2^{n^3}\right)$, $\ldots$

NP versus EXP

Proposition

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 $|t| \le p(|s|)$ 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. $\square$

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

Is NP efficiently solvable?

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

Big Question

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

If $\mathrm{P} = \mathrm{NP} \ldots$

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 $\ldots$
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

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

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 $\mathrm{P} = \mathrm{NP}\ldots$

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 $\mathrm{P} = \mathrm{NP} \ldots$

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 $\ldots$
5. Creativity can be automated! Proofs for mathematical statement can be found by computers automatically (if short ones exist).

If $\mathrm{P} = \mathrm{NP}\ldots$

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 **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!

3.3: NP Completeness

Certifiers

Definition

An algorithm $C(\cdot,\cdot)$ is a **certifier** for problem X if for every $s \in X$ there is some string t such that $C(s,t) =$ "yes", and conversely, if for some s and t, $C(s,t) =$ "yes" then $s \in X$.
The string t is called a certificate or proof for s.

Definition (Efficient Certifier.)

A certifier C is an **efficient certifier** for problem X if there is a polynomial $p(\cdot)$ such that for every string s, we have that

- ⋆ $s \in X$ if and only if
- ⋆ there is a string t:
  1. $|t| \leq p(|s|)$,
  2. $C(s,t) =$ "yes",
  3. and C runs in polynomial time.

NP-Complete Problems

Definition

A problem X is said to be **NP-Complete** if

1. $X \in$ **NP**, and
2. (Hardness) For any $Y \in$ **NP**, **Y** $\leq_P$ **X**.

Solving **NP-Complete** Problems

Proposition

Suppose X is **NP-Complete**. *Then X can be solved in polynomial time if and only if* **P** $=$ **NP**.

Proof.

$\Rightarrow$ Suppose X can be solved in polynomial time

1. Let $Y \in$ **NP**. We know **Y** $\leq_P$ **X**.
2. We showed that if **Y** $\leq_P$ **X** and X can be solved in polynomial time, then Y can be solved in polynomial time.
3. Thus, every problem $Y \in$ **NP** is such that $Y \in P$; $NP \subseteq P$.
4. Since **P** $\subseteq$ **NP**, we have **P** $=$ **NP**.

$\Leftarrow$ Since **P** $=$ **NP**, and $X \in$ **NP**, we have a polynomial time algorithm for X. $\square$

NP-Hard Problems

1. Formal definition:

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in$ **NP**, we have that **Y** $\leq_P$ **X**.

2. An **NP-Hard** problem need not be in **NP**!
3. Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

NP-Hard Problems

1. Formal definition:

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in$ **NP**, we have that **Y** $\leq_P$ **X**.

2. An **NP-Hard** problem need not be in **NP**!
3. Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

NP-Hard Problems

1. Formal definition:

Definition

A problem X is said to be **NP-Hard** if

1. (Hardness) For any $Y \in$ **NP**, we have that **Y** $\leq_P$ **X**.

2. An **NP-Hard** problem need not be in **NP**!
3. Example: Halting problem is **NP-Hard** (why?) but not **NP-Complete**.

Consequences of proving **NP-Complete**ness

1. If X is **NP-Complete**
   1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
   2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

   X is unlikely to be efficiently solvable.
2. At the very least, many smart people before you have failed to find an efficient algorithm for X.
3. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving **NP-Complete**ness

1. If X is **NP-Complete**
   1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
   2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

   X is unlikely to be efficiently solvable.
2. At the very least, many smart people before you have failed to find an efficient algorithm for X.
3. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving **NP-Complete**ness

1. If X is **NP-Complete**
   1. Since we believe **P** $\neq$ **NP**,
   2. and solving X implies **P** $=$ **NP**.

   X is unlikely to be efficiently solvable.
2. At the very least, many smart people before you have failed to find an efficient algorithm for X.
3. (This is proof by mob opinion — take with a grain of salt.)

Consequences of proving **NP-Complete**ness

1. If X is **NP-Complete**
   1. Since we believe $\mathbf{P} \neq \mathbf{NP}$,
   2. and solving X implies $\mathbf{P} = \mathbf{NP}$.

   X is unlikely to be efficiently solvable.
2. At the very least, many smart people before you have failed to find an efficient algorithm for X.
3. (This is proof by mob opinion — take with a grain of salt.)

Notes

Notes

Notes

Notes