P vs NP

Billy Hardy

West Virginia University

April 29, 2015

Outline

1. What if **P** $=$ **NP** ?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity

Outline

1. What if **P** $=$ **NP** ?
    - The Great Collapse
    - The Power of Nondeterminism
    - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard

Outline

1. What if **P** $=$ **NP** ?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

Outline

1. What if $\mathbf{P} = \mathbf{NP}$?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

Outline

1. What if $\mathbf{P} = \mathbf{NP}$?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

Difficulty

The Meaning of **P** vs **NP**

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to **find solutions**

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to **find solutions** than it is to **check solutions**.

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to **find solutions** than it is to **check solutions**.

Intuition leads one to believe that it is.

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to **find solutions** than it is to **check solutions**.

Intuition leads one to believe that it is.

Big Consequences of **P** = **NP**

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to **find solutions** than it is to **check solutions**.

Intuition leads one to believe that it is.

Big Consequences of $\mathbf{P} = \mathbf{NP}$

We will see that $\mathbf{P} = \mathbf{NP}$ leads to a great many complexity classes to also be equal to **P**

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to **find solutions** than it is to **check solutions**.

Intuition leads one to believe that it is.

Big Consequences of **P** = **NP**

We will see that **P** = **NP** leads to a great many complexity classes to also be equal to **P**

One such example is **P** = **coNP** ,

Difficulty

The Meaning of **P** vs **NP**

The biggest consequence of the relationship between **P** and **NP** is whether it is harder to **find solutions** than it is to **check solutions**.

Intuition leads one to believe that it is.

Big Consequences of $\mathbf{P} = \mathbf{NP}$

We will see that $\mathbf{P} = \mathbf{NP}$ leads to a great many complexity classes to also be equal to **P**

One such example is $\mathbf{P} = \mathbf{coNP}$, since one can easily switch the outputs “yes” and “no” of polynomial-time algorithms.

More Complexity Classes

NP and **coNP**

More Complexity Classes

NP and coNP

NP and **coNP** can be thought of as **P** problems which ask for existence (or lack there of).

More Complexity Classes

NP and **coNP**

NP and **coNP** can be thought of as **P** problems which ask for existence (or lack there of).

This is because the definition of NP is $\exists w : B(x, w)$ where w is the witness and B is in **P** , and **coNP** is $\forall w : B(x, w)$.

More Complexity Classes

NP and **coNP**

NP and **coNP** can be thought of as **P** problems which ask for existence (or lack there of).

This is because the definition of NP is $\exists w : B(x, w)$ where w is the witness and B is in **P** , and **coNP** is $\forall w : B(x, w)$.

Extending the Idea

More Complexity Classes

NP and **coNP**

NP and **coNP** can be thought of as **P** problems which ask for existence (or lack there of).

This is because the definition of NP is $\exists w : B(x, w)$ where w is the witness and B is in **P** , and **coNP** is $\forall w : B(x, w)$.

Extending the Idea

One can extend this idea by adding more and more quantifiers.

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties *A* of the form

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

where B is in **P** , and where $|y|$ and $|z|$ are polynomial in $|x|$.

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

where B is in **P** , and where $|y|$ and $|z|$ are polynomial in $|x|$.

Smallest Boolean Circuit

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

where B is in **P** , and where $|y|$ and $|z|$ are polynomial in $|x|$.

Smallest Boolean Circuit

Input: A Boolean circuit C that computes a function f_C of its input.

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

where B is in **P** , and where $|y|$ and $|z|$ are polynomial in $|x|$.

Smallest Boolean Circuit

Input: A Boolean circuit C that computes a function f_C of its input.
Query: Is C the smallest circuit that computes f_C?

The Class Π_2**P**

Π_2**P**

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

where B is in **P** , and where $|y|$ and $|z|$ are polynomial in $|x|$.

Smallest Boolean Circuit

Input: A Boolean circuit C that computes a function f_C of its input.
Query: Is C the smallest circuit that computes f_C?

Logically: $\forall C' < C : \exists x : f_{C'}(x) \neq f_C(x)$

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

where B is in **P** , and where $|y|$ and $|z|$ are polynomial in $|x|$.

Smallest Boolean Circuit

Input: A Boolean circuit C that computes a function f_C of its input.
Query: Is C the smallest circuit that computes f_C?

Logically: $\forall C' < C : \exists x : f_{C'}(x) \neq f_C(x)$

Observation

The Class $\Pi_2\mathbf{P}$

$\Pi_2\mathbf{P}$

The class of properties A of the form

$$A(x) = \forall y : \exists z : B(x, y, z)$$

where B is in **P** , and where $|y|$ and $|z|$ are polynomial in $|x|$.

Smallest Boolean Circuit

Input: A Boolean circuit C that computes a function f_C of its input.
Query: Is C the smallest circuit that computes f_C?

Logically: $\forall C' < C : \exists x : f_{C'}(x) \neq f_C(x)$

Observation

Obviously Smallest Boolean Circuit is in $\Pi_2\mathbf{P}$.

Further Classes

Σ_k**P**

Further Classes

Σ_k**P**

Σ_k**P** is the class of properties *A* of the form

Further Classes

$\Sigma_k \mathbf{P}$

$\Sigma_k \mathbf{P}$ is the class of properties A of the form

$$A(x) = \exists y_1 : \forall y_2 : \exists y_3 : \cdots : Q y_k : B(x, y_1, \ldots, y_k),$$

Further Classes

$\Sigma_k \mathbf{P}$

$\Sigma_k \mathbf{P}$ is the class of properties A of the form

$$A(x) = \exists y_1 : \forall y_2 : \exists y_3 : \cdots : Q y_k : B(x, y_1, \ldots, y_k),$$

where B is in **P** , $|y_i| = poly(|x|)$ for all i, and $Q = \exists$ if k is odd, otherwise $\forall$.

Further Classes

Σ_k**P**

Σ_k**P** is the class of properties A of the form

$$A(x) = \exists y_1 : \forall y_2 : \exists y_3 : \cdots : Qy_k : B(x, y_1, \ldots, y_k),$$

where B is in **P** , $|y_i| = poly(|x|)$ for all i, and $Q = \exists$ if k is odd, otherwise $\forall$.

Π_k**P**

Further Classes

Σ_k**P**

Σ_k**P** is the class of properties A of the form

$$A(x) = \exists y_1 : \forall y_2 : \exists y_3 : \cdots : Qy_k : B(x, y_1, \ldots, y_k),$$

where B is in **P** , $|y_i| = poly(|x|)$ for all i, and $Q = \exists$ if k is odd, otherwise $\forall$.

Π_k**P**

Π_k**P** is the class of properties A of the form

Further Classes

Σ_k**P**

Σ_k**P** is the class of properties A of the form

$$A(x) = \exists y_1 : \forall y_2 : \exists y_3 : \cdots : Qy_k : B(x, y_1, \ldots, y_k),$$

where B is in **P** , $|y_i| = poly(|x|)$ for all i, and $Q = \exists$ if k is odd, otherwise $\forall$.

Π_k**P**

Π_k**P** is the class of properties A of the form

$$A(x) = \forall y_1 : \exists y_2 : \forall y_3 : \cdots : Qy_k : B(x, y_1, \ldots, y_k),$$

Further Classes

$\Sigma_k\mathbf{P}$

$\Sigma_k\mathbf{P}$ is the class of properties A of the form

$$A(x) = \exists y_1 : \forall y_2 : \exists y_3 : \cdots : Qy_k : B(x, y_1, \ldots, y_k),$$

where B is in **P** , $|y_i| = poly(|x|)$ for all i, and $Q = \exists$ if k is odd, otherwise $\forall$.

$\Pi_k\mathbf{P}$

$\Pi_k\mathbf{P}$ is the class of properties A of the form

$$A(x) = \forall y_1 : \exists y_2 : \forall y_3 : \cdots : Qy_k : B(x, y_1, \ldots, y_k),$$

where B is in **P** , $|y_i| = poly(|x|)$ for all i, and $Q = \forall$ if k is odd, otherwise $\exists$.

Further Classes

Understanding These Classes

Further Classes

Understanding These Classes

These classes correspond to two-player games that last for k moves.

Further Classes

Understanding These Classes

These classes correspond to two-player games that last for k moves.

For instance, consider a Chess game where white claims they can mate in k moves.

Further Classes

Understanding These Classes

These classes correspond to two-player games that last for k moves.

For instance, consider a Chess game where white claims they can mate in k moves.

This means there exists a move for white, such that for all of black's replies, there exists a move for white, ... until white has won.

Further Classes

Understanding These Classes

These classes correspond to two-player games that last for k moves.

For instance, consider a Chess game where white claims they can mate in k moves.

This means there exists a move for white, such that for all of black's replies, there exists a move for white, ... until white has won.

Given the initial position and the sequence of moves, it is easy to check whether white has mated black.

Relationships of the Classes

Subsets

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Nondeterminism

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Nondeterminism

As before, each $\exists$ can be thought of as a layer of nondeterminism that asks whether there is a witness that makes the statement inside that quantifier true.

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Nondeterminism

As before, each $\exists$ can be thought of as a layer of nondeterminism that asks whether there is a witness that makes the statement inside that quantifier true.

So we can say,

$$\Sigma_k \mathbf{P} = \mathbf{N}\Pi_{k-1}\mathbf{P} .$$

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Nondeterminism

As before, each $\exists$ can be thought of as a layer of nondeterminism that asks whether there is a witness that makes the statement inside that quantifier true.

So we can say,

$$\Sigma_k\mathbf{P} = \mathbf{N}\Pi_{k-1}\mathbf{P} .$$

And since $\Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}$,

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Nondeterminism

As before, each $\exists$ can be thought of as a layer of nondeterminism that asks whether there is a witness that makes the statement inside that quantifier true.

So we can say,

$$\Sigma_k \mathbf{P} = \mathbf{N}\Pi_{k-1}\mathbf{P} .$$

And since $\Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}$, we have

$$\Sigma_1\mathbf{P} = \mathbf{NP} \textit{ and } \Pi_1\mathbf{P} = \mathbf{coNP}$$

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Nondeterminism

As before, each $\exists$ can be thought of as a layer of nondeterminism that asks whether there is a witness that makes the statement inside that quantifier true.

So we can say,

$$\Sigma_k\mathbf{P} = \mathbf{N}\Pi_{k-1}\mathbf{P} .$$

And since $\Sigma_0\mathbf{P} = \Pi_0\mathbf{P} = \mathbf{P}$, we have

$$\Sigma_1\mathbf{P} = \mathbf{NP} \textit{ and } \Pi_1\mathbf{P} = \mathbf{coNP}$$

Or, more generally,

$$\Pi_k\mathbf{P} = \mathbf{co}\Sigma_k\mathbf{P} .$$

Relationships of the Classes

Subsets

One can easily add quantifiers with dummy variables inside or outside each of the problems in the classes, so

$$\Sigma_k \subseteq \Sigma_{k+1}, \Sigma_k \subseteq \Pi_{k+1}, \Pi_k \subseteq \Sigma_{k+1}, \Pi_k \subseteq \Pi_{k+1}$$

Nondeterminism

As before, each $\exists$ can be thought of as a layer of nondeterminism that asks whether there is a witness that makes the statement inside that quantifier true.

So we can say,

$$\Sigma_k \mathbf{P} = \mathbf{N}\Pi_{k-1}\mathbf{P} .$$

And since $\Sigma_0 \mathbf{P} = \Pi_0 \mathbf{P} = \mathbf{P}$, we have

$$\Sigma_1 \mathbf{P} = \mathbf{NP} \textit{ and } \Pi_1 \mathbf{P} = \mathbf{coNP}$$

Or, more generally,

$$\Pi_k \mathbf{P} = \mathbf{co}\Sigma_k \mathbf{P} .$$

since the negation of $\forall$ is $\exists$, and vice versa.

Polynomial Hierarchy

Polynomial Hierarchy

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Taking their union over all k gives the class

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Taking their union over all k gives the class

$$\mathbf{PH} = \bigcup_{k=0}^{\infty} \Sigma_k \mathbf{P} = \bigcup_{k=0}^{\infty} \Pi_k \mathbf{P} \,,$$

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Taking their union over all k gives the class

$$\mathbf{PH} = \bigcup_{k=0}^{\infty} \Sigma_k \mathbf{P} = \bigcup_{k=0}^{\infty} \Pi_k \mathbf{P} ,$$

which consists of problems that can be phrased with any constant number of quatifiers.

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Taking their union over all k gives the class

$$\mathbf{PH} = \bigcup_{k=0}^{\infty} \Sigma_k \mathbf{P} = \bigcup_{k=0}^{\infty} \Pi_k \mathbf{P} ,$$

which consists of problems that can be phrased with any constant number of quatifiers.

Classes are Distinct

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Taking their union over all k gives the class

$$\mathbf{PH} = \bigcup_{k=0}^{\infty} \Sigma_k\mathbf{P} = \bigcup_{k=0}^{\infty} \Pi_k\mathbf{P},$$

which consists of problems that can be phrased with any constant number of quatifiers.

Classes are Distinct

Analogous to the belief that $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{NP} \neq \mathbf{coNP}$,

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Taking their union over all k gives the class

$$\mathbf{PH} = \bigcup_{k=0}^{\infty} \Sigma_k \mathbf{P} = \bigcup_{k=0}^{\infty} \Pi_k \mathbf{P},$$

which consists of problems that can be phrased with any constant number of quatifiers.

Classes are Distinct

Analogous to the belief that $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{NP} \neq \mathbf{coNP}$, it is believed that the classes Σ_k and Π_k are all distinct.

Polynomial Hierarchy

Polynomial Hierarchy

These complexity classes are known, collectively, as the *polynomial hierarchy*.

Taking their union over all k gives the class

$$\mathbf{PH} = \bigcup_{k=0}^{\infty} \Sigma_k \mathbf{P} = \bigcup_{k=0}^{\infty} \Pi_k \mathbf{P} ,$$

which consists of problems that can be phrased with any constant number of quatifiers.

Classes are Distinct

Analogous to the belief that $\mathbf{P} \neq \mathbf{NP}$ and $\mathbf{NP} \neq \mathbf{coNP}$, it is believed that the classes Σ_k and Π_k are all distinct.

In other words, whenever one adds a quantifier, or a layer of nondeterminism, a fundamentally deeper kind of problem is obtained.

The Great Collapse

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

The Great Collapse

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

If $\mathbf{P} = \mathbf{NP}$, then

The Great Collapse

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

If $\mathbf{P} = \mathbf{NP}$, then

$$\textit{if } B(x,y) \in \mathbf{P}\ ,\ \textit{then } A(x) = \exists y : B(x,y) \in \mathbf{P}\ ,$$

by definition.

The Great Collapse

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

If $\mathbf{P} = \mathbf{NP}$, then

$$\textit{if } B(x,y) \in \mathbf{P} \,,\ \textit{then } A(x) = \exists y : B(x,y) \in \mathbf{P} \,,$$

by definition.

Since $\mathbf{P} = \mathbf{coNP}$ as well, we also absorb $\exists$ and $\forall$.

The Great Collapse

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

If $\mathbf{P} = \mathbf{NP}$, then

$$\textit{if } B(x,y) \in \mathbf{P}\ , \ \textit{then } A(x) = \exists y : B(x,y) \in \mathbf{P}\ ,$$

by definition.

Since $\mathbf{P} = \mathbf{coNP}$ as well, we also absorb $\exists$ and $\forall$.

By continually absorbing quantifiers, we get $\mathbf{PH} = \mathbf{P}$

The Great Collapse

Claim

The Great Collapse

Claim

If **NP** $=$ **coNP**, then **PH** $=$ **NP**.

The Great Collapse

Claim

If **NP** = **coNP**, then **PH** = **NP**.

Proof

The Great Collapse

Claim

If **NP** $=$ **coNP** , then **PH** $=$ **NP** .

Proof

1. Let $A(x) = \exists y : B(x, y)$ be in **NP** $= \Sigma_1\mathbf{P}$

The Great Collapse

Claim

If $\mathbf{NP} = \mathbf{coNP}$, then $\mathbf{PH} = \mathbf{NP}$.

Proof

1. Let $A(x) = \exists y : B(x, y)$ be in $\mathbf{NP} = \Sigma_1\mathbf{P}$
2. $C(x) = \forall z : A(x)$ is in $\Pi_2\mathbf{P}$

The Great Collapse

Claim

If **NP** $=$ **coNP**, then **PH** $=$ **NP**.

Proof

1. Let $A(x) = \exists y : B(x, y)$ be in **NP** $= \Sigma_1$**P**
2. $C(x) = \forall z : A(x)$ is in Π_2**P**
3. $A(x)$ is also in **coNP**, so $A(x) = \forall y : B(x, y)$

The Great Collapse

Claim

If **NP** $=$ **coNP** , then **PH** $=$ **NP** .

Proof

1. Let $A(x) = \exists y : B(x, y)$ be in **NP** $= \Sigma_1\mathbf{P}$
2. $C(x) = \forall z : A(x)$ is in $\Pi_2\mathbf{P}$
3. $A(x)$ is also in **coNP** , so $A(x) = \forall y : B(x, y)$
4. $C(x) = \forall z : \forall y : B(x, y)$

The Great Collapse

Claim

If **NP** = **coNP** , then **PH** = **NP** .

Proof

1. Let $A(x) = \exists y : B(x, y)$ be in **NP** $= \Sigma_1$**P**
2. $C(x) = \forall z : A(x)$ is in Π_2**P**
3. $A(x)$ is also in **coNP** , so $A(x) = \forall y : B(x, y)$
4. $C(x) = \forall z : \forall y : B(x, y)$
5. $C(x) = \forall (z, y) : B(x, (z, y)) = A(x)$ So **NP** $= \Pi_2$**P**

The Great Collapse

Claim

If **NP** $=$ **coNP** , then **PH** $=$ **NP** .

Proof

1. Let $A(x) = \exists y : B(x, y)$ be in **NP** $= \Sigma_1\mathbf{P}$
2. $C(x) = \forall z : A(x)$ is in $\Pi_2\mathbf{P}$
3. $A(x)$ is also in **coNP** , so $A(x) = \forall y : B(x, y)$
4. $C(x) = \forall z : \forall y : B(x, y)$
5. $C(x) = \forall(z, y) : B(x, (z, y)) = A(x)$ So **NP** $= \Pi_2\mathbf{P}$
6. The inductive proof follows from here.

Outline

1. What if $\mathbf{P} = \mathbf{NP}$?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

P vs. NP

Not about Polynomial Time

P vs. **NP**

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

P vs. **NP**

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

In fact, it is really about how powerful nondeterminism is in general.

P vs. **NP**

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

In fact, it is really about how powerful nondeterminism is in general.

As in, whether finding solutions is inherently harder than checking them.

P vs. NP

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

In fact, it is really about how powerful nondeterminism is in general.

As in, whether finding solutions is inherently harder than checking them.

EXP

P vs. NP

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

In fact, it is really about how powerful nondeterminism is in general.

As in, whether finding solutions is inherently harder than checking them.

EXP

Recall that **EXP** is the class of problems that one can solve in an exponential amount of time,

P vs. **NP**

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

In fact, it is really about how powerful nondeterminism is in general.

As in, whether finding solutions is inherently harder than checking them.

EXP

Recall that **EXP** is the class of problems that one can solve in an exponential amount of time, where "exponential" is defined as

$$\mathbf{EXP} = \bigcup_k \mathbf{TIME}(2^{n^k}) = \mathbf{TIME}(2^{poly(n)}).$$

P vs. NP

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

In fact, it is really about how powerful nondeterminism is in general.

As in, whether finding solutions is inherently harder than checking them.

EXP

Recall that **EXP** is the class of problems that one can solve in an exponential amount of time, where “exponential” is defined as

$$\mathbf{EXP} = \bigcup_k \mathbf{TIME}(2^{n^k}) = \mathbf{TIME}(2^{poly(n)}).$$

NEXP

P vs. NP

Not about Polynomial Time

The common misconception is that **P** vs. **NP** is about polynomial time.

In fact, it is really about how powerful nondeterminism is in general.

As in, whether finding solutions is inherently harder than checking them.

EXP

Recall that **EXP** is the class of problems that one can solve in an exponential amount of time, where "exponential" is defined as

$$\mathbf{EXP} = \bigcup_k \mathbf{TIME}(2^{n^k}) = \mathbf{TIME}(2^{poly(n)}).$$

NEXP

Also recall that $\mathbf{NEXP} = \mathbf{NTIME}(2^{poly(n)})$ is the class of problems that one can check a solution in an exponential amount of time.

P vs. NP

The Relationship between **EXP** and **NEXP**

P vs. **NP**

The Relationship between **EXP** and **NEXP**

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

P vs. NP

The Relationship between **EXP** and **NEXP**

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

Furthermore, the extension can be made to **EXPEXP** $\neq$ **NEXPEXP**, and so on.

P vs. **NP**

The Relationship between **EXP** and **NEXP**

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

Furthermore, the extension can be made to **EXPEXP** $\neq$ **NEXPEXP**, and so on.

Claim

P vs. NP

The Relationship between EXP and NEXP

In analogy to $\mathbf{P} \neq \mathbf{NP}$, one can check whether $\mathbf{EXP} \neq \mathbf{NEXP}$.

Furthermore, the extension can be made to $\mathbf{EXPEXP} \neq \mathbf{NEXPEXP}$, and so on.

Claim

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

P vs. NP

The Relationship between **EXP** and **NEXP**

In analogy to $\mathbf{P} \neq \mathbf{NP}$, one can check whether $\mathbf{EXP} \neq \mathbf{NEXP}$.

Furthermore, the extension can be made to $\mathbf{EXPEXP} \neq \mathbf{NEXPEXP}$, and so on.

Claim

If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$,

P vs. NP

The Relationship between EXP and NEXP

In analogy to $\mathbf{P} \neq \mathbf{NP}$, one can check whether $\mathbf{EXP} \neq \mathbf{NEXP}$.

Furthermore, the extension can be made to $\mathbf{EXPEXP} \neq \mathbf{NEXPEXP}$, and so on.

Claim

If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$, $\mathbf{EXPEXP} = \mathbf{NEXPEXP}$,

P vs. **NP**

The Relationship between **EXP** and **NEXP**

In analogy to $\mathbf{P} \neq \mathbf{NP}$, one can check whether $\mathbf{EXP} \neq \mathbf{NEXP}$.

Furthermore, the extension can be made to $\mathbf{EXPEXP} \neq \mathbf{NEXPEXP}$, and so on.

Claim

If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$, $\mathbf{EXPEXP} = \mathbf{NEXPEXP}$, and so on.

P vs. **NP**

The Relationship between **EXP** and **NEXP**

In analogy to $\mathbf{P} \neq \mathbf{NP}$, one can check whether $\mathbf{EXP} \neq \mathbf{NEXP}$.

Furthermore, the extension can be made to $\mathbf{EXPEXP} \neq \mathbf{NEXPEXP}$, and so on.

Claim

If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$, $\mathbf{EXPEXP} = \mathbf{NEXPEXP}$, and so on.

Proof

P vs. NP

The Relationship between EXP and NEXP

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

Furthermore, the extension can be made to **EXPEXP** $\neq$ **NEXPEXP**, and so on.

Claim

If **P** $=$ **NP** then **EXP** $=$ **NEXP**, **EXPEXP** $=$ **NEXPEXP**, and so on.

Proof

- Let problem A be in **NEXP**, so witnesses can be checked in time $t(n) = 2^{O(n^c)}$, for some constant c.

P vs. NP

The Relationship between EXP and NEXP

In analogy to $\mathbf{P} \neq \mathbf{NP}$, one can check whether $\mathbf{EXP} \neq \mathbf{NEXP}$.

Furthermore, the extension can be made to $\mathbf{EXPEXP} \neq \mathbf{NEXPEXP}$, and so on.

Claim

If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$, $\mathbf{EXPEXP} = \mathbf{NEXPEXP}$, and so on.

Proof

- Let problem A be in **NEXP**, so witnesses can be checked in time $t(n) = 2^{O(n^c)}$, for some constant c.
- Now pad the input, making it $t(n)$ bits long, by adding $t(n) - n$ zeros.

P vs. NP

The Relationship between EXP and NEXP

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

Furthermore, the extension can be made to **EXPEXP** $\neq$ **NEXPEXP**, and so on.

Claim

If **P** $=$ **NP** then **EXP** $=$ **NEXP**, **EXPEXP** $=$ **NEXPEXP**, and so on.

Proof

- Let problem A be in **NEXP**, so witnesses can be checked in time $t(n) = 2^{O(n^c)}$, for some constant c.
- Now pad the input, making it $t(n)$ bits long, by adding $t(n) - n$ zeros.
- This takes $O(t(n))$ time, since $t(n)$ is *time constructible*

P vs. NP

The Relationship between **EXP** and **NEXP**

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

Furthermore, the extension can be made to **EXPEXP** $\neq$ **NEXPEXP**, and so on.

Claim

If **P** $=$ **NP** then **EXP** $=$ **NEXP**, **EXPEXP** $=$ **NEXPEXP**, and so on.

Proof

- Let problem A be in **NEXP**, so witnesses can be checked in time $t(n) = 2^{O(n^c)}$, for some constant c.
- Now pad the input, making it $t(n)$ bits long, by adding $t(n) - n$ zeros.
- This takes $O(t(n))$ time, since $t(n)$ is *time constructible*
- This new problem is in **NP** .

P vs. NP

The Relationship between **EXP** and **NEXP**

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

Furthermore, the extension can be made to **EXPEXP** $\neq$ **NEXPEXP**, and so on.

Claim

If **P** $=$ **NP** then **EXP** $=$ **NEXP**, **EXPEXP** $=$ **NEXPEXP**, and so on.

Proof

- Let problem A be in **NEXP**, so witnesses can be checked in time $t(n) = 2^{O(n^c)}$, for some constant c.
- Now pad the input, making it $t(n)$ bits long, by adding $t(n) - n$ zeros.
- This takes $O(t(n))$ time, since $t(n)$ is *time constructible*
- This new problem is in **NP** .
- So we can solve it deterministically in time $poly(t(n)) = 2^{O(n^c)}$, since **P** $=$ **NP** .

P vs. NP

The Relationship between **EXP** and **NEXP**

In analogy to **P** $\neq$ **NP** , one can check whether **EXP** $\neq$ **NEXP**.

Furthermore, the extension can be made to **EXPEXP** $\neq$ **NEXPEXP**, and so on.

Claim

If **P** $=$ **NP** then **EXP** $=$ **NEXP**, **EXPEXP** $=$ **NEXPEXP**, and so on.

Proof

- Let problem A be in **NEXP**, so witnesses can be checked in time $t(n) = 2^{O(n^c)}$, for some constant c.
- Now pad the input, making it $t(n)$ bits long, by adding $t(n) - n$ zeros.
- This takes $O(t(n))$ time, since $t(n)$ is *time constructible*
- This new problem is in **NP** .
- So we can solve it deterministically in time $poly(t(n)) = 2^{O(n^c)}$, since **P** $=$ **NP** .
- So A is in **EXP**.

P vs. NP

The Relationship between EXP and NEXP

In analogy to $\mathbf{P} \neq \mathbf{NP}$, one can check whether $\mathbf{EXP} \neq \mathbf{NEXP}$.

Furthermore, the extension can be made to $\mathbf{EXPEXP} \neq \mathbf{NEXPEXP}$, and so on.

Claim

If $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$, $\mathbf{EXPEXP} = \mathbf{NEXPEXP}$, and so on.

Proof

- Let problem A be in **NEXP**, so witnesses can be checked in time $t(n) = 2^{O(n^c)}$, for some constant c.
- Now pad the input, making it $t(n)$ bits long, by adding $t(n) - n$ zeros.
- This takes $O(t(n))$ time, since $t(n)$ is *time constructible*
- This new problem is in **NP** .
- So we can solve it deterministically in time $poly(t(n)) = 2^{O(n^c)}$, since $\mathbf{P} = \mathbf{NP}$.
- So A is in **EXP**.
- The inductive proof follows similarly.

Time-Constructible

Time-Constructible

Time-Constructible

Time-Constructible

A function f is called *time-constructible* if there exists a positive integer n_0 and Turing machine M which, given a string 1^n consisting of n ones, stops after exactly $f(n)$ steps for all $n \geq n_0$.

P vs. NP

More Generally

P vs. NP

More Generally

We can say if **P** $=$ **NP** , then for any time-constructible function $t(n) \geq n$,

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{TIME}(poly(t(n)))$$

P vs. **NP**

More Generally

We can say if **P** $=$ **NP** , then for any time-constructible function $t(n) \geq n$,

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{TIME}(poly(t(n)))$$

Or for a class of superpolynomial functions such that $t(n)^c \in C$ for any $t(n) \in C$ and any constant c, then

$$\mathbf{NTIME}(C) = \mathbf{TIME}(C)$$

P vs. NP

More Generally

We can say if $\mathbf{P} = \mathbf{NP}$, then for any time-constructible function $t(n) \geq n$,

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{TIME}(poly(t(n)))$$

Or for a class of superpolynomial functions such that $t(n)^c \in C$ for any $t(n) \in C$ and any constant c, then

$$\mathbf{NTIME}(C) = \mathbf{TIME}(C)$$

The Collapse

P vs. NP

More Generally

We can say if **P** $=$ **NP** , then for any time-constructible function $t(n) \geq n$,

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{TIME}(poly(t(n)))$$

Or for a class of superpolynomial functions such that $t(n)^c \in C$ for any $t(n) \in C$ and any constant c, then

$$\mathbf{NTIME}(C) = \mathbf{TIME}(C)$$

The Collapse

This applies not only to exponentials $2^{poly(n)}$, double exponential $2^{2^{poly(n)}}$ and so on.

P vs. NP

More Generally

We can say if **P** = **NP** , then for any time-constructible function $t(n) \geq n$,

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{TIME}(poly(t(n)))$$

Or for a class of superpolynomial functions such that $t(n)^c \in C$ for any $t(n) \in C$ and any constant c, then

$$\mathbf{NTIME}(C) = \mathbf{TIME}(C)$$

The Collapse

This applies not only to exponentials $2^{poly(n)}$, double exponential $2^{2^{poly(n)}}$ and so on.

So we have **EXP** = **NEXP**, **EXPEXP** = **NEXPEXP**, and so on up the hierarchy.

P vs. NP

More Generally

We can say if **P** $=$ **NP** , then for any time-constructible function $t(n) \geq n$,

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{TIME}(poly(t(n)))$$

Or for a class of superpolynomial functions such that $t(n)^c \in C$ for any $t(n) \in C$ and any constant c, then

$$\mathbf{NTIME}(C) = \mathbf{TIME}(C)$$

The Collapse

This applies not only to exponentials $2^{poly(n)}$, double exponential $2^{2^{poly(n)}}$ and so on.

So we have **EXP** $=$ **NEXP**, **EXPEXP** $=$ **NEXPEXP**, and so on up the hierarchy.

It is easy to show that any equality in the hierarchy propagates up,

P vs. NP

More Generally

We can say if **P** $=$ **NP** , then for any time-constructible function $t(n) \geq n$,

$$\mathbf{NTIME}(t(n)) \subseteq \mathbf{TIME}(poly(t(n)))$$

Or for a class of superpolynomial functions such that $t(n)^c \in C$ for any $t(n) \in C$ and any constant c, then

$$\mathbf{NTIME}(C) = \mathbf{TIME}(C)$$

The Collapse

This applies not only to exponentials $2^{poly(n)}$, double exponential $2^{2^{poly(n)}}$ and so on.

So we have **EXP** $=$ **NEXP**, **EXPEXP** $=$ **NEXPEXP**, and so on up the hierarchy.

It is easy to show that any equality in the hierarchy propagates up, and inequality propagates down.

Outline

1. What if $\mathbf{P} = \mathbf{NP}$?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

Proof Finding vs. Checking

PROOF CHECKING

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P
Query: Is P a valid proof of S?

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P
Query: Is P a valid proof of S?

Short Proof

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P
Query: Is P a valid proof of S?

Short Proof

Input: A statement S and an integer n given in unary

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P
Query: Is P a valid proof of S?

Short Proof

Input: A statement S and an integer n given in unary
Query: Does S have a proof of length n or less?

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P
Query: Is P a valid proof of S?

Short Proof

Input: A statement S and an integer n given in unary
Query: Does S have a proof of length n or less?

Observation

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P
Query: Is P a valid proof of S?

Short Proof

Input: A statement S and an integer n given in unary
Query: Does S have a proof of length n or less?

Observation

Obviously Proof Checking is in **P** , which implies Short Proof is in **NP** .

Proof Finding vs. Checking

Proof Checking

Input: A statement S and a proof P
Query: Is P a valid proof of S?

Short Proof

Input: A statement S and an integer n given in unary
Query: Does S have a proof of length n or less?

Observation

Obviously Proof Checking is in **P** , which implies Short Proof is in **NP** . Furthermore, since S can be a SAT formula, Short Proof is **NP** -complete.

Elegant Theory

Exhaustive Search

Elegant Theory

Exhaustive Search

If there are k letters in the alphabet for proofs, there are k^n possible proofs.

Elegant Theory

Exhaustive Search

If there are k letters in the alphabet for proofs, there are k^n possible proofs.

So we can solve Short Proof in polynomial time precisely if we can do better than an exhaustive search.

Elegant Theory

Exhaustive Search

If there are k letters in the alphabet for proofs, there are k^n possible proofs.

So we can solve Short Proof in polynomial time precisely if we can do better than an exhaustive search.

Not Just Computer Science

ELEGANT THEORY

Exhaustive Search

If there are k letters in the alphabet for proofs, there are k^n possible proofs.

So we can solve SHORT PROOF in polynomial time precisely if we can do better than an exhaustive search.

Not Just Computer Science

The consequences of $\mathbf{P} = \mathbf{NP}$ reach beyond Computer Science, as we can tweak SHORT PROOF a bit.

ELEGANT THEORY

Exhaustive Search

If there are k letters in the alphabet for proofs, there are k^n possible proofs.

So we can solve SHORT PROOF in polynomial time precisely if we can do better than an exhaustive search.

Not Just Computer Science

The consequences of **P** $=$ **NP** reach beyond Computer Science, as we can tweak SHORT PROOF a bit.

ELEGANT THEORY

ELEGANT THEORY

Exhaustive Search

If there are k letters in the alphabet for proofs, there are k^n possible proofs.

So we can solve SHORT PROOF in polynomial time precisely if we can do better than an exhaustive search.

Not Just Computer Science

The consequences of $\mathbf{P} = \mathbf{NP}$ reach beyond Computer Science, as we can tweak SHORT PROOF a bit.

ELEGANT THEORY

Input: A set E of experimental data and an integer n given in unary

Elegant Theory

Exhaustive Search

If there are k letters in the alphabet for proofs, there are k^n possible proofs.

So we can solve Short Proof in polynomial time precisely if we can do better than an exhaustive search.

Not Just Computer Science

The consequences of **P** = **NP** reach beyond Computer Science, as we can tweak Short Proof a bit.

Elegant Theory

Input: A set E of experimental data and an integer n given in unary
Query: Is there a theory T of length n or less that explains E?

Outline

1. What if **P** = **NP** ?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

Proving **P** $\neq$ **NP**

Strategy

Proving **P** $\neq$ **NP**

Strategy

The direct strategy is to prove that for some problem $A \in$ **NP** , $A \notin$ **P** .

Proving **P** $\neq$ **NP**

Strategy

The direct strategy is to prove that for some problem $A \in$ **NP** , $A \notin$ **P** .

So one must prove that **every possible** polynomial time algorithm that could solve A, fails.

Proving **P** $\neq$ **NP**

Strategy

The direct strategy is to prove that for some problem $A \in$ **NP** , $A \notin$ **P** .

So one must prove that **every possible** polynomial time algorithm that could solve A, fails.

Which is not easy.

Proving **P** $\neq$ **NP**

Strategy

The direct strategy is to prove that for some problem $A \in$ **NP** , $A \notin$ **P** .

So one must prove that **every possible** polynomial time algorithm that could solve A, fails.

Which is not easy.

Complexity Classes

Proving $\mathbf{P} \neq \mathbf{NP}$

Strategy

The direct strategy is to prove that for some problem $A \in \mathbf{NP}$, $A \notin \mathbf{P}$.

So one must prove that **every possible** polynomial time algorithm that could solve A, fails.

Which is not easy.

Complexity Classes

This leads us to realize that proving upper bounds on classes is easy compared to proving lower bounds.

Proving **P** $\neq$ **NP**

Strategy

The direct strategy is to prove that for some problem $A \in$ **NP** , $A \notin$ **P** .

So one must prove that **every possible** polynomial time algorithm that could solve A, fails.

Which is not easy.

Complexity Classes

This leads us to realize that proving upper bounds on classes is easy compared to proving lower bounds.

(One just has to find one such algorithm that solves A in polynomial time to increase the upper bound on **P**)

Easy Lower Bounds

Sorting a List

Easy Lower Bounds

Sorting a List

As shown in Chapter 3, sorting a list of numbers has to be done in at least $O(n \cdot log(n))$ time,

Easy Lower Bounds

Sorting a List

As shown in Chapter 3, sorting a list of numbers has to be done in at least $O(n \cdot log(n))$ time, when comparisons between list members are made.

Easy Lower Bounds

Sorting a List

As shown in Chapter 3, sorting a list of numbers has to be done in at least $O(n \cdot log(n))$ time, when comparisons between list members are made.

Better Than $O(n \cdot log(n))$

Easy Lower Bounds

Sorting a List

As shown in Chapter 3, sorting a list of numbers has to be done in at least $O(n \cdot log(n))$ time, when comparisons between list members are made.

Better Than $O(n \cdot log(n))$

Radix sort achieves $O(mn)$ time, where m is the number of bits used to represent the elements.

Outline

1. What if **P** $=$ **NP** ?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

Proving Inequality of Classes

Technique

Proving Inequality of Classes

Technique

While proving a particular problem can not be solved in polynomial time appears daunting.

Proving Inequality of Classes

Technique

While proving a particular problem can not be solved in polynomial time appears daunting.

Our actual stategy for proving problems are outside of **P** is to construct artificial problems that any polynomial time algorithm gets incorrect in at least one case.

Proving Inequality of Classes

Technique

While proving a particular problem can not be solved in polynomial time appears daunting.

Our actual stategy for proving problems are outside of **P** is to construct artificial problems that any polynomial time algorithm gets incorrect in at least one case.

Thus, for the class **C** which these problems belong to, we can conclude that

Proving Inequality of Classes

Technique

While proving a particular problem can not be solved in polynomial time appears daunting.

Our actual stategy for proving problems are outside of **P** is to construct artificial problems that any polynomial time algorithm gets incorrect in at least one case.

Thus, for the class **C** which these problems belong to, we can conclude that **P** $\neq$ **C**.

Proving Inequality of Classes

Technique

While proving a particular problem can not be solved in polynomial time appears daunting.

Our actual stategy for proving problems are outside of **P** is to construct artificial problems that any polynomial time algorithm gets incorrect in at least one case.

Thus, for the class **C** which these problems belong to, we can conclude that $\mathbf{P} \neq \mathbf{C}$.

Our Goal

Proving Inequality of Classes

Technique

While proving a particular problem can not be solved in polynomial time appears daunting.

Our actual stategy for proving problems are outside of **P** is to construct artificial problems that any polynomial time algorithm gets incorrect in at least one case.

Thus, for the class **C** which these problems belong to, we can conclude that $\mathbf{P} \neq \mathbf{C}$.

Our Goal

We will use the above technique to prove $\mathbf{P} \neq \mathbf{EXPTIME}$.

Proving Inequality of Classes

Technique

While proving a particular problem can not be solved in polynomial time appears daunting.

Our actual stategy for proving problems are outside of **P** is to construct artificial problems that any polynomial time algorithm gets incorrect in at least one case.

Thus, for the class **C** which these problems belong to, we can conclude that $\mathbf{P} \neq \mathbf{C}$.

Our Goal

We will use the above technique to prove $\mathbf{P} \neq \mathbf{EXPTIME}$.

Or, more generally, that $\mathbf{TIME}(g(n)) \subset \mathbf{TIME}(f(n))$, for $g(n) \in o(f(n))$.

Outline

1. What if $\mathbf{P} = \mathbf{NP}$?
   - The Great Collapse
   - The Power of Nondeterminism
   - The Demise of Creativity
2. Upper Bounds are Easy and Lower Bounds, Hard
3. Diagonalization and Time Hierarchy
   - Time Hierarchy Theorem

Problem Construction

General Problem

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT,

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Input: A program Π and an input x

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Input: A program Π and an input x
Output: If Π halts within $f(|x|)$ steps when given x as input, return its output $\Pi(x)$.

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Input: A program Π and an input x
Output: If Π halts within $f(|x|)$ steps when given x as input, return its output $\Pi(x)$.
Otherwise, return "don't know."

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Input: A program Π and an input x
Output: If Π halts within $f(|x|)$ steps when given x as input, return its output $\Pi(x)$.
Otherwise, return "don't know."

PREDICT's Behavior

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Input: A program Π and an input x
Output: If Π halts within $f(|x|)$ steps when given x as input, return its output $\Pi(x)$.
Otherwise, return "don't know."

PREDICT's Behavior

- Since f is fixed, different values of f we get different versions of PREDICT.

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Input: A program Π and an input x
Output: If Π halts within $f(|x|)$ steps when given x as input, return its output $\Pi(x)$.
Otherwise, return "don't know."

PREDICT's Behavior

- Since f is fixed, different values of f we get different versions of PREDICT.
- PREDICT captures Π's behavior for *precisely* $f(|x|)$ steps or less.

Problem Construction

General Problem

For a fixed function $f(n)$, we create the problem PREDICT, which will take in as input a problem Π and Π's input x.

PREDICT(Π,x)

Input: A program Π and an input x
Output: If Π halts within $f(|x|)$ steps when given x as input, return its output $\Pi(x)$. Otherwise, return "don't know."

PREDICT's Behavior

- Since f is fixed, different values of f we get different versions of PREDICT.
- PREDICT captures Π's behavior for *precisely* $f(|x|)$ steps or less. Not some constant times $f(|x|)$.

Diagonalization

CATCH22(Π)

Diagonalization

CATCH22(Π)

Input: A program Π

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input, return the *negation* of its output $\overline{\Pi(\Pi)}$.

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input,
return the *negation* of its output $\overline{\Pi(\Pi)}$.
Otherwise, return "don't know."

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input,
return the *negation* of its output $\overline{\Pi(\Pi)}$.
Otherwise, return "don't know."

Claim

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input, return the *negation* of its output $\overline{\Pi(\Pi)}$. Otherwise, return "don't know."

Claim

CATCH22 can not be solved in $f(n)$ steps or less.

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input, return the *negation* of its output $\overline{\Pi(\Pi)}$.
Otherwise, return "don't know."

Claim

CATCH22 can not be solved in $f(n)$ steps or less.

Proof

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input, return the *negation* of its output $\overline{\Pi(\Pi)}$.
Otherwise, return "don't know."

Claim

CATCH22 can not be solved in $f(n)$ steps or less.

Proof

- Assume the contrary. So $\exists \Pi_{22}$ which runs on inputs x in $f(|x|)$ steps or less.

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input, return the *negation* of its output $\overline{\Pi(\Pi)}$.
Otherwise, return "don't know."

Claim

CATCH22 can not be solved in $f(n)$ steps or less.

Proof

- Assume the contrary. So $\exists \Pi_{22}$ which runs on inputs x in $f(|x|)$ steps or less.
- So $\Pi_{22}(\Pi_{22})$ runs within $f(|\Pi_{22}|)$ steps.

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input, return the *negation* of its output $\overline{\Pi(\Pi)}$.
Otherwise, return "don't know."

Claim

CATCH22 can not be solved in $f(n)$ steps or less.

Proof

- Assume the contrary. So $\exists \Pi_{22}$ which runs on inputs x in $f(|x|)$ steps or less.
- So $\Pi_{22}(\Pi_{22})$ runs within $f(|\Pi_{22}|)$ steps.
- So $\Pi_{22}(\Pi_{22}) = \overline{\Pi_{22}(\Pi_{22})}$

Diagonalization

CATCH22(Π)

Input: A program Π
Output: If Π halts within $f(|\Pi|)$ steps when given its own source code as input,
return the *negation* of its output $\overline{\Pi(\Pi)}$.
Otherwise, return “don’t know.”

Claim

CATCH22 can not be solved in $f(n)$ steps or less.

Proof

- Assume the contrary. So $\exists \Pi_{22}$ which runs on inputs x in $f(|x|)$ steps or less.
- So $\Pi_{22}(\Pi_{22})$ runs within $f(|\Pi_{22}|)$ steps.
- So $\Pi_{22}(\Pi_{22}) = \overline{\Pi_{22}(\Pi_{22})}$
- So there can not exist any program Π_{22}.

Diagonalization

PREDICT

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

This means PREDICT can not be solved in less than $f(n)$ steps, as well.

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

This means PREDICT can not be solved in less than $f(n)$ steps, as well.

CATCH22's Running Time

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

This means PREDICT can not be solved in less than $f(n)$ steps, as well.

CATCH22's Running Time

- We can solve CATCH22 by running an interpreter on Π's source code for $f(|\Pi|)$ steps and seeing what happens.

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

This means PREDICT can not be solved in less than $f(n)$ steps, as well.

CATCH22's Running Time

- We can solve CATCH22 by running an interpreter on Π's source code for $f(|\Pi|)$ steps and seeing what happens.
- In order to ensure only at most $f(|\Pi|)$ steps occur, the interpreter takes $s(t)$ steps to run t steps of Π.

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

This means PREDICT can not be solved in less than $f(n)$ steps, as well.

CATCH22's Running Time

- We can solve CATCH22 by running an interpreter on Π's source code for $f(|\Pi|)$ steps and seeing what happens.
- In order to ensure only at most $f(|\Pi|)$ steps occur, the interpreter takes $s(t)$ steps to run t steps of Π.
- By the previous proof, $s(t) > t$.

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

This means PREDICT can not be solved in less than $f(n)$ steps, as well.

CATCH22's Running Time

- We can solve CATCH22 by running an interpreter on Π's source code for $f(|\Pi|)$ steps and seeing what happens.
- In order to ensure only at most $f(|\Pi|)$ steps occur, the interpreter takes $s(t)$ steps to run t steps of Π.
- By the previous proof, $s(t) > t$.
- Assuming a random access machine, $s(t) = O(t)$.

Diagonalization

PREDICT

Since CATCH22 is just a special case of PREDICT and takes one extra step to negate the result.

This means PREDICT can not be solved in less than $f(n)$ steps, as well.

CATCH22's Running Time

- We can solve CATCH22 by running an interpreter on Π's source code for $f(|\Pi|)$ steps and seeing what happens.
- In order to ensure only at most $f(|\Pi|)$ steps occur, the interpreter takes $s(t)$ steps to run t steps of Π.
- By the previous proof, $s(t) > t$.
- Assuming a random access machine, $s(t) = O(t)$.
- So CATCH22 can be solved in $s(f(n)) + O(f(n)) = O(s(f(n)))$ time.

Time Hierarchy Thereom

Time Hierarchy Thereom

Time Hierarchy Thereom

Time Hierarchy Thereom

Assume an interpreter can simulate t steps of an arbitrary program Π that runs in at most $f(n)$ steps, while keeping track of the number of steps computed thus far in $s(t)$ steps.

Time Hierarchy Thereom

Time Hierarchy Thereom

Assume an interpreter can simulate t steps of an arbitrary program Π that runs in at most $f(n)$ steps, while keeping track of the number of steps computed thus far in $s(t)$ steps. Then if $g(n) = o(f(n))$,

$$\mathbf{TIME}(g(n)) \subset \mathbf{TIME}(s(f(n)))$$

Time Hierarchy Thereom

Time Hierarchy Thereom

Assume an interpreter can simulate t steps of an arbitrary program Π that runs in at most $f(n)$ steps, while keeping track of the number of steps computed thus far in $s(t)$ steps. Then if $g(n) = o(f(n))$,

$$\mathbf{TIME}(g(n)) \subset \mathbf{TIME}(s(f(n)))$$

Proof

Time Hierarchy Thereom

Time Hierarchy Thereom

Assume an interpreter can simulate t steps of an arbitrary program Π that runs in at most $f(n)$ steps, while keeping track of the number of steps computed thus far in $s(t)$ steps. Then if $g(n) = o(f(n))$,

$$\mathbf{TIME}(g(n)) \subset \mathbf{TIME}(s(f(n)))$$

Proof

Since CATCH22 can not be solved exactly $f(n)$ steps, it can not be solved in $O(g(n))$ steps for any $g(n) = o(f(n))$.

Time Hierarchy Thereom

Time Hierarchy Thereom

Assume an interpreter can simulate t steps of an arbitrary program Π that runs in at most $f(n)$ steps, while keeping track of the number of steps computed thus far in $s(t)$ steps. Then if $g(n) = o(f(n))$,

$$\textbf{TIME}(g(n)) \subset \textbf{TIME}(s(f(n)))$$

Proof

Since CATCH22 can not be solved exactly $f(n)$ steps, it can not be solved in $O(g(n))$ steps for any $g(n) = o(f(n))$.

This proves the Time Hierarchy Thereom.

Time Hierarchy Thereom

Time Hierarchy Thereom

Assume an interpreter can simulate t steps of an arbitrary program Π that runs in at most $f(n)$ steps, while keeping track of the number of steps computed thus far in $s(t)$ steps. Then if $g(n) = o(f(n))$,

$$\mathbf{TIME}(g(n)) \subset \mathbf{TIME}(s(f(n)))$$

Proof

Since CATCH22 can not be solved exactly $f(n)$ steps, it can not be solved in $O(g(n))$ steps for any $g(n) = o(f(n))$.

This proves the Time Hierarchy Thereom.

More Time Does Mean More Computation

Time Hierarchy Thereom

Time Hierarchy Thereom

Assume an interpreter can simulate t steps of an arbitrary program Π that runs in at most $f(n)$ steps, while keeping track of the number of steps computed thus far in $s(t)$ steps. Then if $g(n) = o(f(n))$,

$$\mathbf{TIME}(g(n)) \subset \mathbf{TIME}(s(f(n)))$$

Proof

Since CATCH22 can not be solved exactly $f(n)$ steps, it can not be solved in $O(g(n))$ steps for any $g(n) = o(f(n))$.

This proves the Time Hierarchy Thereom.

More Time Does Mean More Computation

The Time Hierarchy Thereom proves:

$$\mathbf{P} \subset \mathbf{EXP} \subset \mathbf{EXPEXP} \subset \cdots$$