CSE 373: P vs NP

Michael Lee

Monday, Mar 5, 2018

Previously:

- We spent a lot of time learning how to solve problems

Overview

Previously:

- We spent a lot of time learning how to solve problems
- We spent a lot of time analyzing algorithms

Overview

Today:

- Take a step back and look at the bigger picture

Overview

Today:

- Take a step back and look at the bigger picture
- Discuss an important open question in computer science: does $\mathrm{P} = \mathrm{NP}$?

What is “efficiency”?

But first:

What does it mean for a problem to be “efficient”?

What is “efficiency”?

But first:

What does it mean for a problem to be “efficient”?

What do we even mean by “problem”, anyways?

What is a “decision problem”?

Decision problem

A **decision problem** is any arbitrary yes-or-no question on an infinite set of inputs.

What is a "decision problem"?

Decision problem

A **decision problem** is any arbitrary yes-or-no question on an infinite set of inputs.

Which of these are decision problems?

- `IS-PRIME`: "Is X prime? (Where X is some input)"
- `FIND-PRIME`: "What is the *n*-th prime number?"
- `SORT`: "Sort this list of numbers."
- `IS-SORTED`: "Is this list of numbers sorted?"

What is a "decision problem"?

Decision problem

A **decision problem** is any arbitrary yes-or-no question on an infinite set of inputs.

Which of these are decision problems?

- `IS-PRIME`: "Is X prime? (Where X is some input)"
  Yes, it's a yes-or-no question.
- `FIND-PRIME`: "What is the *n*-th prime number?"
  No. The answer is a number, not a boolean.
- `SORT`: "Sort this list of numbers."
  No; not a question.
- `IS-SORTED`: "Is this list of numbers sorted?"
  Yes, it's a yes-or-no question.

What is a “decision problem”?

Question: Why only talk about decision problems?

What is a “decision problem”?

Question: Why only talk about decision problems?

Answer: It simplifies things. Also, most problems can be turned into a decision problem with some tweaking, so not a big deal.

What is a "decision problem"?

Question: Why only talk about decision problems?

Answer: It simplifies things. Also, most problems can be turned into a decision problem with some tweaking, so not a big deal.

Example:

`SHORTEST-PATH`: "What is the shortest path between two given nodes?"

What is a "decision problem"?

Question: Why only talk about decision problems?

Answer: It simplifies things. Also, most problems can be turned into a decision problem with some tweaking, so not a big deal.

Example:

SHORTEST-PATH: "What is the shortest path between two given nodes?"

...can be turned into:

PATH: "Does there exist a path between two given nodes that consists of k edges?"

$k=1$ $k=2$ $k=3$

What is a “solvable” problem?

Solvable

A decision problem is **solvable** if there exists some algorithm that given any input, or *instance*, can correctly *decide* “yes” or “no”.

What is a "solvable" problem?

Solvable

A decision problem is **solvable** if there exists some algorithm that given any input, or *instance*, can correctly *decide* "yes" or "no".

Example: `IS-PRIME` is solvable. Here's an algorithm:

```
boolean isPrimeSolver(n):
    for (int i = 2; i < n; i++)
        if (X % i == 0):
            return false
    return true
```

What is a "solvable" problem?

Question: Are there problems that are unsolvable – problems that are impossible to solve?

What is a “solvable” problem?

Question: Are there problems that are unsolvable – problems that are impossible to solve?

Surprisingly, yes.

What is a “solvable” problem?

Question: Are there problems that are unsolvable – problems that are impossible to solve?

Surprisingly, yes.

We won’t go into that today; look up the “halting problem” if you’re curious.

Definitions

Questions:

- What do we even mean by “problem”, anyways?

Definitions

Questions:

- What do we even mean by “problem”, anyways?
- What does it mean for a problem to be “efficient”?

What is an “efficient algorithm”?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

What is an "efficient algorithm"?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are "efficient"?

- $\mathcal{O}\left(n^2\right)$:

What is an "efficient algorithm"?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are "efficient"?

- $\mathcal{O}\left(n^2\right)$: Yes, it's a polynomial

What is an "efficient algorithm"?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are "efficient"?

- $\mathcal{O}\left(n^2\right)$: Yes, it's a polynomial
- $\mathcal{O}\left(2^n\right)$:

What is an “efficient algorithm”?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are “efficient”?

- $\mathcal{O}\left(n^2\right)$: Yes, it’s a polynomial
- $\mathcal{O}\left(2^n\right)$: No, it’s an exponential

What is an “efficient algorithm”?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are “efficient”?

- $\mathcal{O}\left(n^2\right)$: Yes, it's a polynomial
- $\mathcal{O}\left(2^n\right)$: No, it's an exponential
- $\mathcal{O}\left(n \log(n)\right)$:

What is an "efficient algorithm"?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are "efficient"?

- $\mathcal{O}\left(n^2\right)$: Yes, it's a polynomial
- $\mathcal{O}\left(2^n\right)$: No, it's an exponential
- $\mathcal{O}\left(n\log(n)\right)$: Yes, $n\log(n) \in \mathcal{O}\left(n^2\right)$, which is a polynomial

What is an "efficient algorithm"?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are "efficient"?

- $\mathcal{O}\left(n^2\right)$: Yes, it's a polynomial
- $\mathcal{O}\left(2^n\right)$: No, it's an exponential
- $\mathcal{O}\left(n\log(n)\right)$: Yes, $n\log(n) \in \mathcal{O}\left(n^2\right)$, which is a polynomial
- $\mathcal{O}\left(n^{10000000}\right)$:

What is an "efficient algorithm"?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are "efficient"?

- $\mathcal{O}\left(n^2\right)$: Yes, it's a polynomial
- $\mathcal{O}\left(2^n\right)$: No, it's an exponential
- $\mathcal{O}\left(n \log(n)\right)$: Yes, $n \log(n) \in \mathcal{O}\left(n^2\right)$, which is a polynomial
- $\mathcal{O}\left(n^{10000000}\right)$: Technically yes...

What is an “efficient algorithm”?

Efficient algorithm

An algorithm is **efficient** if the worst-case bound is a **polynomial**.

Examples: which of these runtime bounds are “efficient”?

- $\mathcal{O}\left(n^2\right)$: Yes, it’s a polynomial
- $\mathcal{O}\left(2^n\right)$: No, it’s an exponential
- $\mathcal{O}\left(n \log(n)\right)$: Yes, $n \log(n) \in \mathcal{O}\left(n^2\right)$, which is a polynomial
- $\mathcal{O}\left(n^{10000000}\right)$: Technically yes...
- $\mathcal{O}\left(30000000000000n^3\right)$: Technically yes...

What is an "efficient algorithm"?

Question: Are $n^{10000000}$ and $30000000000000n^3$ *actually* efficient in practice?

What is an “efficient algorithm”?

Question: Are $n^{10000000}$ and $30000000000000n^3$ *actually* efficient in practice?

No, but...

What is an "efficient algorithm"?

Question: Are $n^{10000000}$ and $3000000000000000n^3$ *actually* efficient in practice?

No, but...

- Once we find a polynomial algorithm to a problem, we've historically been able to improve it to something reasonable

What is an "efficient algorithm"?

Question: Are $n^{10000000}$ and $30000000000000n^3$ *actually* efficient in practice?

No, but...

- Once we find a polynomial algorithm to a problem, we've historically been able to improve it to something reasonable
- Finding a polynomial runtime is a *VERY* low bar. If we can't even get that...

Examples of problems

Pretty much all problems we've studied have efficient solutions!

Examples of problems

Pretty much all problems we've studied have efficient solutions!

We've studied two main types of algorithms: sorting algorithms and graph algorithms, and every one we've looked at so far could run in polynomial time.

(e.g "How do I sort this list", "What is the shortest path", "What is the MST"...)

Examples of problems

Great: do all solvable problems have efficient solutions?

Examples of problems

Great: do all solvable problems have efficient solutions?

Haha, no.

Examples of problems

Great: do all solvable problems have efficient solutions?

Haha, no.

Well, ok – do all *practical* problems we actually care about have efficient solutions?

Examples of problems

Great: do all solvable problems have efficient solutions?

Haha, no.

Well, ok – do all *practical* problems we actually care about have efficient solutions?

lol

PATH vs LONGEST-PATH

PATH

inputs

Given a graph and two vertices, does there exist some path between those two vertices that visits exactly k edges?

another input

PATH vs LONGEST-PATH

PATH

Given a graph and two vertices, does there exist some path between those two vertices that visits exactly k edges?

- To solve, run BFS and see if we visit the dest in k edges.
- We can solve this efficiently!

PATH vs LONGEST-PATH

PATH

Given a graph and two vertices, does there exist some path between those two vertices that visits exactly k edges?

- To solve, run BFS and see if we visit the dest in k edges.
- We can solve this efficiently!

What if we tweak the problem a little?

PATH vs LONGEST-PATH

PATH

Given a graph and two vertices, does there exist some path between those two vertices that visits exactly k edges?

- To solve, run BFS and see if we visit the dest in k edges.
- We can solve this efficiently!

What if we tweak the problem a little?

LONGEST-PATH

Given a graph, does there exist a path between **any** two vertices that visits exactly k edges?

PATH vs LONGEST-PATH

PATH

Given a graph and two vertices, does there exist some path between those two vertices that visits exactly k edges?

- To solve, run BFS and see if we visit the dest in k edges.
- We can solve this efficiently!

What if we tweak the problem a little?

LONGEST-PATH

Given a graph, does there exist a path between **any** two vertices that visits exactly k edges?

There is no known efficient solution to this problem.

To solve, use brute force.

2-COLOR vs 3-COLOR

2-COLOR

Given a graph, is it possible to assign each node one of two colors such that no two adjacent nodes share the same color?

- To solve, run BFS or DFS, alternate colors...
- We can solve this efficiently!

2-COLOR vs 3-COLOR

2-COLOR

Given a graph, is it possible to assign each node one of two colors such that no two adjacent nodes share the same color?

- To solve, run BFS or DFS, alternate colors...
- We can solve this efficiently!

What if we tweak the problem a little?

2-COLOR vs 3-COLOR

2-COLOR

Given a graph, is it possible to assign each node one of two colors such that no two adjacent nodes share the same color?

- To solve, run BFS or DFS, alternate colors...
- We can solve this efficiently!

What if we tweak the problem a little?

3-COLOR

Given a graph, is it possible to assign each node one of **three** colors such that no two adjacent nodes share the same color?"

2-COLOR vs 3-COLOR

2-COLOR

Given a graph, is it possible to assign each node one of two colors such that no two adjacent nodes share the same color?

- To solve, run BFS or DFS, alternate colors...
- We can solve this efficiently!

What if we tweak the problem a little?

3-COLOR

Given a graph, is it possible to assign each node one of **three** colors such that no two adjacent nodes share the same color?"

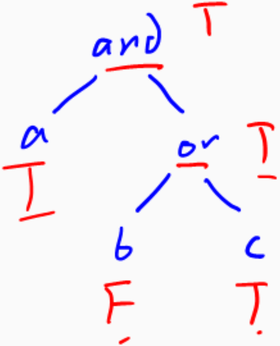
There is no known efficient solution to this problem.

To solve, use brute force: try all $\mathcal{O}\left(3^{|V|}\right)$ combinations.

CIRCUIT-VALUE

T F T

Given a boolean expression such as "a && (b || c)" and the truth values for every variable, is the final expression T?



CIRCUIT-VALUE vs CIRCUIT-SAT

CIRCUIT-VALUE

Given a boolean expression such as "`a && (b || c)`" and the truth values for every variable, is the final expression T?

- To solve, convert into an abstract syntax tree and evaluate.
- We can solve this efficiently!

CIRCUIT-VALUE vs CIRCUIT-SAT

CIRCUIT-VALUE

Given a boolean expression such as "a && (b || c)" and the truth values for every variable, is the final expression T?

T

- To solve, convert into an abstract syntax tree and evaluate.
- We can solve this efficiently!

CIRCUIT-SAT

Given a boolean expression such as "a && (b || c)" and the truth values for **some** of the variables, is there a way to set the remaining variables so that the output is T?

CIRCUIT-VALUE vs CIRCUIT-SAT

CIRCUIT-VALUE

Given a boolean expression such as "`a && (b || c)`" and the truth values for every variable, is the final expression T?

- To solve, convert into an abstract syntax tree and evaluate.
- We can solve this efficiently!

CIRCUIT-SAT

Given a boolean expression such as "`a && (b || c)`" and the truth values for **some** of the variables, is there a way to set the remaining variables so that the output is T?

There is no known efficient solution to this problem.

To solve, use brute force: try every combination of variables.

Complexity classes

Observation: Some problems have polynomial solutions, some have worse.

Can we formalize this?

Complexity classes

Observation: Some problems have polynomial solutions, some have worse.

Can we formalize this?

Complexity class

A **complexity class** is a set of problems limited by some resource constraint (time, space, etc)

Complexity class: P and EXP

The complexity class P

P is the set of all decision problems where there exists an algorithm that can solve all inputs in worst-case polynomial time.

Complexity class: P and EXP

The complexity class P

P is the set of all decision problems where there exists an algorithm that can solve all inputs in worst-case polynomial time.

Examples: `IS-PRIME`, `IS-SORTED`, `PATH`, `2-COLOR`, `CIRCUIT-VALUE`, ...

Complexity class: P and EXP

The complexity class P

P is the set of all decision problems where there exists an algorithm that can solve all inputs in worst-case polynomial time.

Examples: `IS-PRIME`, `IS-SORTED`, `PATH`, `2-COLOR`, `CIRCUIT-VALUE`, ...

The complexity class EXP

EXP is the set of all decision problems where there exists an algorithm that can solve all inputs in worst-case exponential time.

Examples: `LONGEST-PATH`, `3-COLOR`, `CIRCUIT-SAT`...

Is P a subset of EXP?

Question: Suppose we have some random decision problem in P. Is that problem also in EXP?

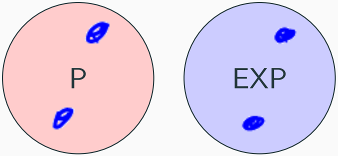
E.g. is `2-COLOR` in EXP?

Is P a subset of EXP?

There are three reasonable possibilities:

Answer 1: The sets are disjoint

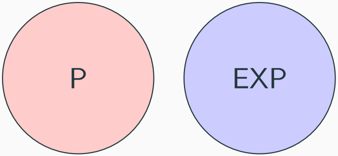
E.g. if a problem is in P, it's not in EXP.



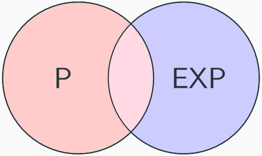
Is P a subset of EXP?

There are three reasonable possibilities:

Answer 1: The sets are disjoint
E.g. if a problem is in P, it's not in EXP.



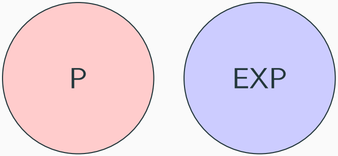
Answer 2: The sets overlap
E.g. some, but not all problems in P are in EXP



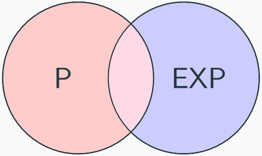
Is P a subset of EXP?

There are three reasonable possibilities:

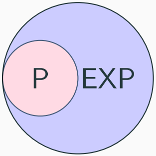
Answer 1: The sets are disjoint
E.g. if a problem is in P, it's not in EXP.



Answer 2: The sets overlap
E.g. some, but not all problems in P are in EXP

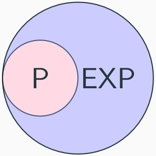


Answer 3: P is a subset of EXP
All problems in P are also in EXP



Is P a subset of EXP?

It turns out it's answer 3: P is a subset of EXP.



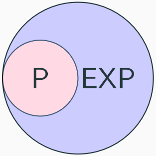
Answer 3: P is a subset of EXP

All problems in P are also in EXP

Is P a subset of EXP?

It turns out it's answer 3: P is a subset of EXP.

Answer 3: P is a subset of EXP
All problems in P are also in EXP

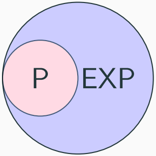


Reason: EXP is the set of decision problems where there exists an algorithm that solves the problem in *worst-case exponential time*.

Is P a subset of EXP?

It turns out it's answer 3: P is a subset of EXP.

Answer 3: P is a subset of EXP
All problems in P are also in EXP



Reason: EXP is the set of decision problems where there exists an algorithm that solves the problem in *worst-case exponential time*.

So, if we can find a polynomial-time algorithm to a problem, we can definitely find an exponential one!

Is P a subset of EXP?

Example: We previously showed there exists an $\mathcal{O}(n)$ algorithm to check if a number n is prime:

```
boolean isPrimeSolver(n):
    for (int i = 2; i < n; i++):
        if (X % i == 0):
            return false
    return true
```

So IS-PRIME $\in$ P.

Is P a subset of EXP?

Example: We previously showed there exists an $\mathcal{O}(n)$ algorithm to check if a number n is prime:

```
boolean isPrimeSolver(n):
    for (int i = 2; i < n; i++):
        if (X % i == 0):
            return false
    return true
```

So IS-PRIME $\in$ P.

How do we show that IS-PRIME is in EXP?

Is P a subset of EXP?

Example: We previously showed there exists an $\mathcal{O}(n)$ algorithm to check if a number n is prime:

```
boolean isPrimeSolver(n):
    for (int i = 2; i < n; i++):
        if (X % i == 0):
            return false
    return true
```

So IS-PRIME $\in$ P.

How do we show that IS-PRIME is in EXP?

```
boolean isPrimeSolver2(n):
    for (int i = 0; i < Math.pow(2, n); i++):
        print("lol")

    return isPrimeSolver(n)
```

This runs in exponential time and correctly solves all inputs.
So IS-PRIME is also in EXP.

Recap

To recap:

- What is a decision problem?

Recap

To recap:

- What is a decision problem?
    - What does it mean to "solve" a decision problem?

Recap

To recap:

- What is a decision problem?
    - What does it mean to "solve" a decision problem?
    - What does it mean for an algorithm to be "efficient"?

Recap

To recap:

- What is a decision problem?
  - What does it mean to “solve” a decision problem?
  - What does it mean for an algorithm to be “efficient”?
- What is a complexity class?

Recap

To recap:

- What is a decision problem?
  - What does it mean to “solve” a decision problem?
  - What does it mean for an algorithm to be “efficient”?
- What is a complexity class?
  - P

Recap

To recap:

- What is a decision problem?
  - What does it mean to “solve” a decision problem?
  - What does it mean for an algorithm to be “efficient”?
- What is a complexity class?
  - P
  - EXP

Recap

To recap:

- What is a decision problem?
  - What does it mean to "solve" a decision problem?
  - What does it mean for an algorithm to be "efficient"?
- What is a complexity class?
  - P
  - EXP
  - P is a subset of EXP

Recap

To recap:

- What is a decision problem?
  - What does it mean to "solve" a decision problem?
  - What does it mean for an algorithm to be "efficient"?
- What is a complexity class?
  - P
  - EXP
  - P is a subset of EXP
- **Unfortunately, some problems we care about are in EXP**

A glimmer of hope...

Observation: Some problems in EXP have an interesting property:

A glimmer of hope...

Observation: Some problems in EXP have an interesting property:

- They may take either polynomial or exponential time to *solve*, but either way...

A glimmer of hope...

Observation: Some problems in EXP have an interesting property:

- They may take either polynomial or exponential time to *solve*, but either way...
- *Checking or verifying* if a solution is correct always takes polynomial time!

A glimmer of hope...

Observation: Some problems in EXP have an interesting property:

- They may take either polynomial or exponential time to *solve*, but either way...
- *Checking or verifying* if a solution is correct always takes polynomial time!

Big idea: NP is the set of decision problems that can be verified in polynomial time.

A glimmer of hope...

Observation: Some problems in EXP have an interesting property:

- They may take either polynomial or exponential time to *solve*, but either way...
- *Checking or verifying* if a solution is correct always takes polynomial time!

Big idea: NP is the set of decision problems that can be verified in polynomial time.

If we can *verify* answers efficiently, can we *find* answers efficiently?

Solving vs verifying

Reminder: a solver is an algorithm that accepts an *instance* of a decision-problem and returns true or false.

Solving vs verifying

Reminder: a solver is an algorithm that accepts an *instance* of a decision-problem and returns true or false.

Another kind of algorithm – a *verifier*

Reminder: a solver is an algorithm that accepts an *instance* of a decision-problem and returns true or false.

Another kind of algorithm – a *verifier*

Verifier

A **verifier** accepts as input:

Solving vs verifying

Reminder: a solver is an algorithm that accepts an *instance* of a decision-problem and returns true or false.

Another kind of algorithm – a *verifier*

Verifier

A **verifier** accepts as input:

1. Some instance of the decision problem

Solving vs verifying

Reminder: a solver is an algorithm that accepts an *instance* of a decision-problem and returns true or false.

Another kind of algorithm – a *verifier*

Verifier

A **verifier** accepts as input:

1. Some instance of the decision problem
2. Some sort of "proof" or *certificate* of why the solver made whatever decision it made on that instance.

The complexity class NP

Suppose that we have some decision problem X where...

- There exists some solver for X

The complexity class NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says "yes" for some instance of X

The complexity class NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says “yes” for some instance of X
- Whenever the solver says “yes”, it also returns some sort of “proof” or *certificate* of why they said “yes”.

The complexity class NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says "yes" for some instance of X
- Whenever the solver says "yes", it also returns some sort of "proof" or *certificate* of why they said "yes".

If there exists a verifier that...

The complexity class NP

The complexity class NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says "yes" for some instance of X
- Whenever the solver says "yes", it also returns some sort of "proof" or *certificate* of why they said "yes".

If there exists a verifier that...

- When given the instance and the certificate, always agrees the correct answer was "yes"

The complexity class NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says "yes" for some instance of X
- Whenever the solver says "yes", it also returns some sort of "proof" or *certificate* of why they said "yes".

If there exists a verifier that...

- When given the instance and the certificate, always agrees the correct answer was "yes"
- Always runs in polynomial time

The complexity class NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says "yes" for some instance of X
- Whenever the solver says "yes", it also returns some sort of "proof" or *certificate* of why they said "yes".

If there exists a verifier that...

- When given the instance and the certificate, always agrees the correct answer was "yes"
- Always runs in polynomial time

...then X is in NP.

The complexity class co-NP

Important note: The verifier only needs to exist when the solver says “yes”.

If the solver says “no”, we don’t care.

The complexity class co-NP

Important note: The verifier only needs to exist when the solver says "yes".

If the solver says "no", we don't care.

A related complexity class: co-NP. Almost identical to NP, except for "NO" instances.

The complexity class co-NP

Suppose that we have some decision problem X where...

- There exists some solver for X

The complexity class co-NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says **"no"** for some instance of X

The complexity class co-NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says **"no"** for some instance of X
- Whenever the solver says "no", it also returns some sort of "proof" or *certificate* of why they said **"no"**.

The complexity class co-NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says **"no"** for some instance of X
- Whenever the solver says "no", it also returns some sort of "proof" or *certificate* of why they said **"no"**.

If there exists a verifier that...

The complexity class co-NP

The complexity class co-NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says **"no"** for some instance of X
- Whenever the solver says "no", it also returns some sort of "proof" or *certificate* of why they said **"no"**.

If there exists a verifier that...

- When given the instance and the certificate, always agrees the correct answer was **"no"**

The complexity class co-NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says **"no"** for some instance of X
- Whenever the solver says "no", it also returns some sort of "proof" or *certificate* of why they said **"no"**.

If there exists a verifier that...

- When given the instance and the certificate, always agrees the correct answer was **"no"**
- Always runs in polynomial time

The complexity class co-NP

The complexity class co-NP

Suppose that we have some decision problem X where...

- There exists some solver for X
- That solver says **"no"** for some instance of X
- Whenever the solver says "no", it also returns some sort of "proof" or *certificate* of why they said **"no"**.

If there exists a verifier that...

- When given the instance and the certificate, always agrees the correct answer was **"no"**
- Always runs in polynomial time

...then X is in **co-NP**.

Example: showing 3-COLOR is in NP

I claim that `3-COLOR` is in NP. How do we show this?

Example: showing 3-COLOR is in NP

I claim that `3-COLOR` is in NP. How do we show this?

Step 1: Assume the preconditions are met.

Example: showing 3-COLOR is in NP

I claim that `3-COLOR` is in NP. How do we show this?

Step 1: Assume the preconditions are met.

Suppose we have a magical solver for 3-COLOR, and it says "yes" for some graph G.

Example: showing 3-COLOR is in NP

I claim that `3-COLOR` is in NP. How do we show this?

Step 1: Assume the preconditions are met.

Suppose we have a magical solver for 3-COLOR, and it says "yes" for some graph G.

Step 2: Show that we can build a polynomial-time verifier, given G and some certificate.

Example: showing 3-COLOR is in NP

I claim that `3-COLOR` is in NP. How do we show this?

Step 1: Assume the preconditions are met.

Suppose we have a magical solver for 3-COLOR, and it says "yes" for some graph G.

Step 2: Show that we can build a polynomial-time verifier, given G and some certificate.

Three things we must do:

1. How do we modify the solver so it returns a convincing certificate?

Example: showing 3-COLOR is in NP

I claim that `3-COLOR` is in NP. How do we show this?

Step 1: Assume the preconditions are met.

Suppose we have a magical solver for 3-COLOR, and it says "yes" for some graph G.

Step 2: Show that we can build a polynomial-time verifier, given G and some certificate.

Three things we must do:

1. How do we modify the solver so it returns a convincing certificate?
2. How do we check the certificate, whatever it is?

Example: showing 3-COLOR is in NP

I claim that `3-COLOR` is in NP. How do we show this?

Step 1: Assume the preconditions are met.

Suppose we have a magical solver for 3-COLOR, and it says "yes" for some graph G.

Step 2: Show that we can build a polynomial-time verifier, given G and some certificate.

Three things we must do:

1. How do we modify the solver so it returns a convincing certificate?
2. How do we check the certificate, whatever it is?
3. Does our verifier actually run in polynomial time?

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

A map of vertices to colors! E.g. $\{v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}, v_4 = \text{green}, \ldots\}$.

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

A map of vertices to colors! E.g.
$\{v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}, v_4 = \text{green}, \ldots\}$.

Part 2b: How do we double-check this certificate?

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

A map of vertices to colors! E.g.
$\{v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}, v_4 = \text{green}, \ldots\}$.

Part 2b: How do we double-check this certificate?

Loop through all vertices, make sure neighbors have diff colors!

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

A map of vertices to colors! E.g.
$\{v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}, v_4 = \text{green}, \ldots\}$.

Part 2b: How do we double-check this certificate?

Loop through all vertices, make sure neighbors have diff colors!

```
boolean verify3Color(G, colorMap):
    for (v : G.vertices):
        for (w : v.neighbors):
            if (colorMap.get(v) == colorMap.get(w)):
                return false
    return true
```

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

A map of vertices to colors! E.g.
$\{v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}, v_4 = \text{green}, \ldots\}$.

Part 2b: How do we double-check this certificate?

Loop through all vertices, make sure neighbors have diff colors!

```
boolean verify3Color(G, colorMap):
    for (v : G.vertices):
        for (w : v.neighbors):
            if (colorMap.get(v) == colorMap.get(w)):
                return false
    return true
```

Part 2c: Does this verifier run in polynomial time?

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

A map of vertices to colors! E.g.
$\{v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}, v_4 = \text{green}, \ldots\}$.

Part 2b: How do we double-check this certificate?

Loop through all vertices, make sure neighbors have diff colors!

```
boolean verify3Color(G, colorMap):
    for (v : G.vertices):
        for (w : v.neighbors):
            if (colorMap.get(v) == colorMap.get(w)):
                return false
    return true
```

Part 2c: Does this verifier run in polynomial time?

Yes! It runs in $\mathcal{O}\left(|V| + |E|\right)$ time!

Example: showing 3-COLOR is in NP

Part 2a: What would be a convincing certificate?

A map of vertices to colors! E.g.
$\{v_1 = \text{red}, v_2 = \text{blue}, v_3 = \text{red}, v_4 = \text{green}, \ldots\}$.

Part 2b: How do we double-check this certificate?

Loop through all vertices, make sure neighbors have diff colors!

```
boolean verify3Color(G, colorMap):
    for (v : G.vertices):
        for (w : v.neighbors):
            if (colorMap.get(v) == colorMap.get(w)):
                return false
    return true
```

2-COLOR $\in$ P
$\in$ NP

Part 2c: Does this verifier run in polynomial time?

Yes! It runs in $\mathcal{O}(|V| + |E|)$ time!

Is 2-color also in NP?

So, 3-COLOR $\in$ NP.

Example: showing CIRCUIT-SAT is in NP

Question: is `CIRCUIT-SAT` in NP?

Example: showing CIRCUIT-SAT is in NP

Question: is `CIRCUIT-SAT` in NP?

`CIRCUIT-SAT`

Given a boolean expression such as "`a && (b || c)`" and the truth values for **some** of the variables, is there a way to set the remaining variables so that the output is T?

Example: showing CIRCUIT-SAT is in NP

Question: is `CIRCUIT-SAT` in NP?

`CIRCUIT-SAT`

Given a boolean expression such as "`a && (b || c)`" and the truth values for **some** of the variables, is there a way to set the remaining variables so that the output is T?

As before, assume you have a magical solver, and it said "yes" for some boolean expression B.

Example: showing CIRCUIT-SAT is in NP

Question: is `CIRCUIT-SAT` in NP?

`CIRCUIT-SAT`

Given a boolean expression such as "`a && (b || c)`" and the truth values for **some** of the variables, is there a way to set the remaining variables so that the output is T?

As before, assume you have a magical solver, and it said "yes" for some boolean expression B.

Three questions to answer:

1. How do we modify the solver so it returns a convincing certificate?
2. How do we check the certificate, whatever it is?
3. Does our verifier actually run in polynomial time?