W13: NP-Completeness

CS3230 AY21/22 Sem 2

Table of Contents

Click on the link to jump to the relevant sections!

Complexity Classes

Last Week: Reductions

$A \leq_p B$ is a **polynomial time reduction** between decision problems A and B, when it transforms `input_A` of problem A to `input_B` of problem B such that:

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_B` is YES-instance → `input_A` is also a YES-instance
- The transformation takes polynomial time in the size of `input_A`

Last Week: Reductions

```
def solve_A(input_A):
1.  input_B = reduction_from_A_to_B(input_A)
2.  output_B = solve_B(input_B) # magically given
3.
4.  output_A = transform_output_B_to_A(output_B)
5.  return output_A
```

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is “easily solvable”, then so is A
- If A is “hard”, then so is B

IMPORTANT

Please get the direction of reduction right!

To show a problem is “hard”, you need to reduce **FROM** a “hard” problem

Doing it in the wrong direction means you are proving the wrong thing → likely to get 0 marks for the question if in an exam

A note on polynomial time

In the context of the NP-Completeness chapter:

- We usually discuss polynomial time **with respect to encoding of input**
- Examples on encoding of input:
  - If input is a number *n*, then the encoding is *O(logn)* bits
  - Array of *n* elements, with max value *M*: *O(nlogM)* bits

- The reason has to do more with Theory of Computation, related to Turing Machines. You aren't required to know this, but keep in mind!

This week: Complexity Class

We will learn about complexity classes:

- P
- NP
- NP-Hard
- NP-Complete

Reductions allows us to compare hardness, allowing us to group problems into the necessary classes

Complexity Class

- To build up our understanding on the hardness of these problems, I will use this “number line” (based on MIT notes)
- Left-to-right: increasing hardness

Complexity Class: P

- P: The **set** of **decision** problems that can be solved in polynomial time

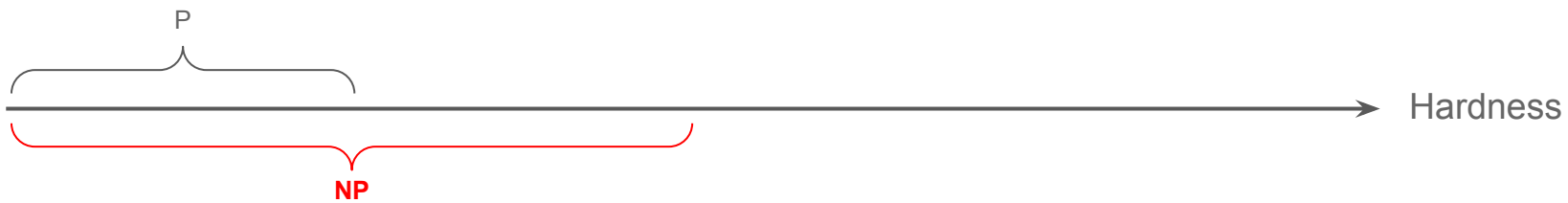
Complexity Class: P

- P: The **set** of **decision** problems that can be solved in polynomial time

- Examples:
  - Shortest Path
  - Minimum Spanning Tree

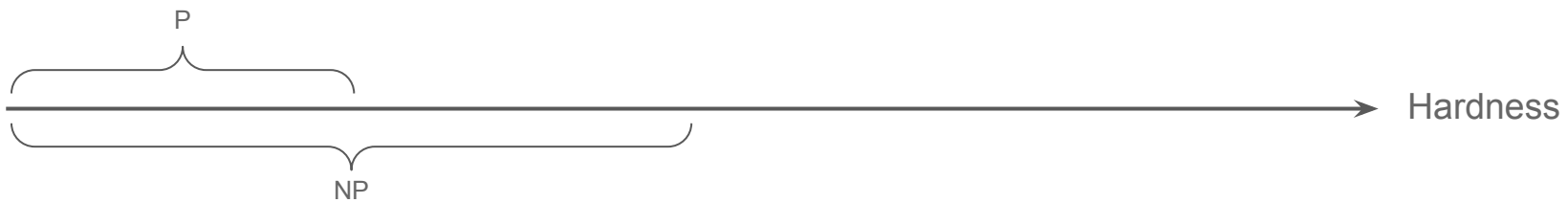
Complexity Class: NP

- NP: The **set** of **decision** problems whose YES-instance can be **verified** in polynomial time



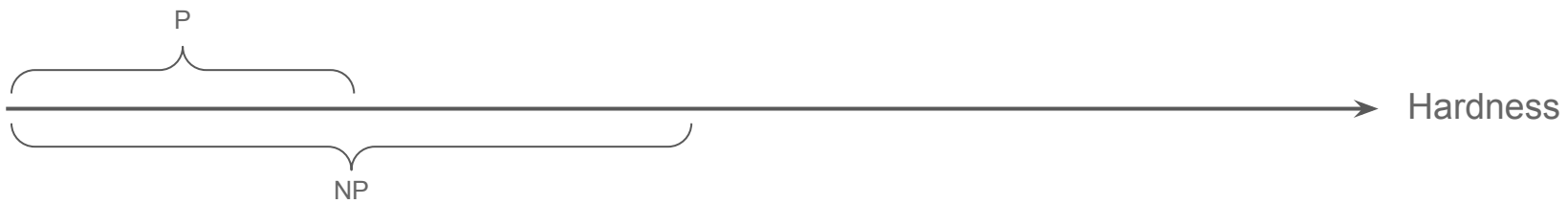
Complexity Class: NP

- NP: The **set** of **decision** problems whose YES-instance can be **verified** in polynomial time
- NP: “Non-deterministic polynomial” (not “non-polynomial”)



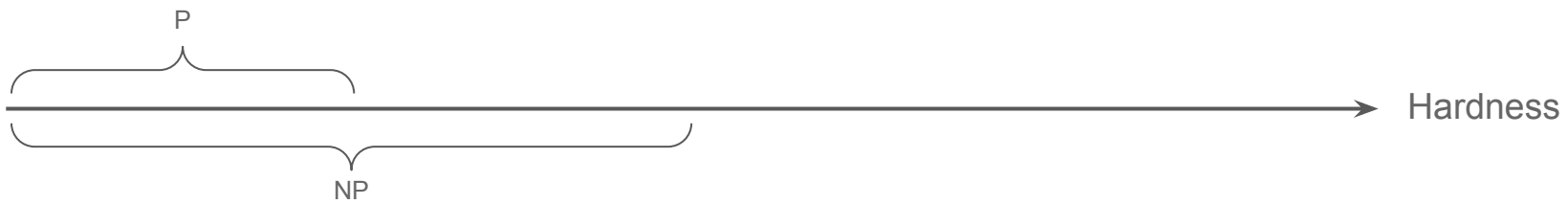
Complexity Class: NP

- NP: The **set** of **decision** problems whose YES-instance can be **verified** in polynomial time
- NP: “Non-deterministic polynomial” (not “non-polynomial”)
- Someone will give you a proposed solution, and your algo efficiently checks that proposed solution (known as **certificate**) is correct



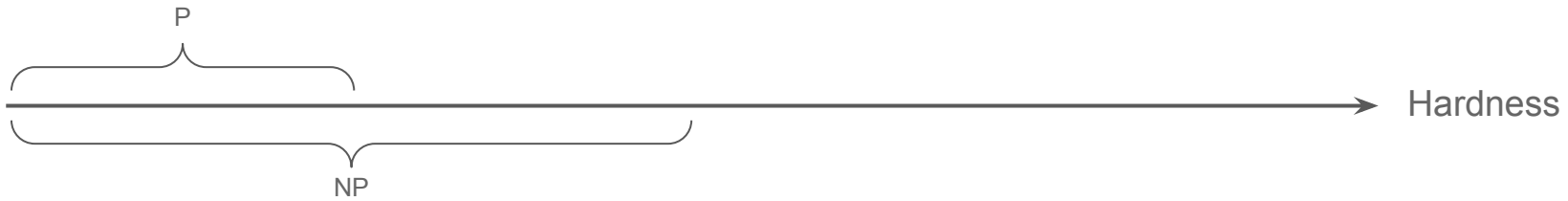
Complexity Class: NP

- NP: The **set** of **decision** problems whose YES-instance can be **verified** in polynomial time
- NP: “Non-deterministic polynomial” (not “non-polynomial”)
- Someone will give you a proposed solution, and your algo efficiently checks that proposed solution (known as **certificate**) is correct
- Intuition: Easier to **check** answer key and thinking if it’s correct, rather than coming up with the correct answer yourself



Complexity Class: NP

- NP: The **set** of **decision** problems whose YES-instance can be **verified** in polynomial time
- NP: “Non-deterministic polynomial” (not “non-polynomial”)
- Someone will give you a proposed solution, and your algo efficiently checks that proposed solution (known as **certificate**) is correct
- Intuition: Easier to **check** answer key and thinking if it’s correct, rather than coming up with the correct answer yourself
- Examples:
  - All problems in P (will explain soon)
  - 3-SAT, Vertex Cover, Independent Set



Example: Partition is in NP

Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?

Example: Partition is in NP

Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?

Certificate (Proposed solution):

Example: Partition is in NP

Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?

Certificate (Proposed solution):

Polytime Verifier Algorithm:

1. Sum up both partitions and check that they are equal

Example: Partition is in NP

Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?

Certificate (Proposed solution):

Polytime Verifier Algorithm:

1. Sum up both partitions and check that they are equal
2. Check that the elements in the proposed solution also matches the input

Example: Partition is in NP

Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?

Certificate (Proposed solution):

Polytime Verifier Algorithm:

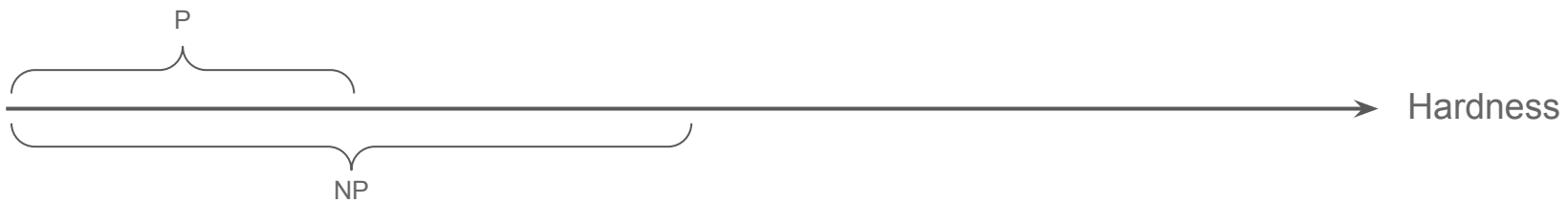
1. Sum up both partitions and check that they are equal
2. Check that the elements in the proposed solution also matches the input

Showing that a problem is in NP:

1. Come up with a polynomially-sized certificate format (what does your proposed solution look like?)
2. Design a polytime algorithm that verifies the certificate (how do you confirm that it's not bluffing you?)

Complexity Class: P ⊆ NP

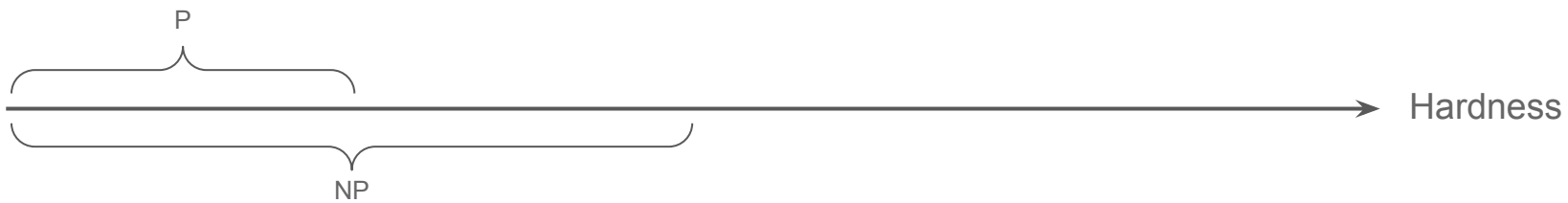
- P: The **set** of **decision** problems that can be solved in polynomial time
- NP: The **set** of **decision** problems that can be **verified** in polynomial time



Complexity Class: P ⊆ NP

- P: The **set** of **decision** problems that can be solved in polynomial time
- NP: The **set** of **decision** problems that can be **verified** in polynomial time

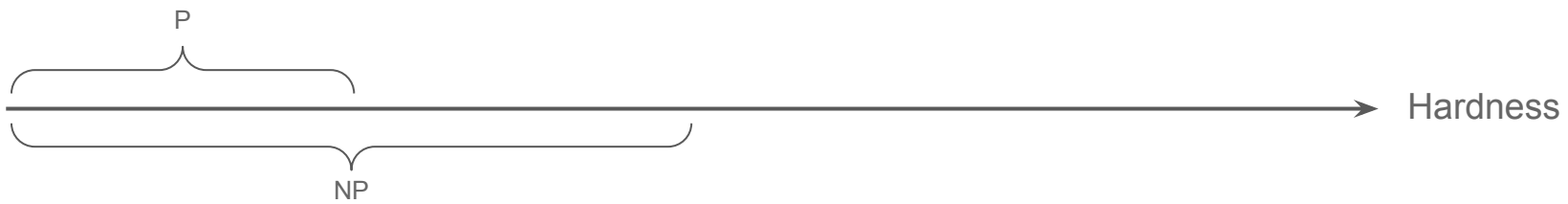
- If problem is in P, and you are given a ‘proposed solution’. How to check?



Complexity Class: P ⊆ NP

- P: The **set** of **decision** problems that can be solved in polynomial time
- NP: The **set** of **decision** problems that can be **verified** in polynomial time

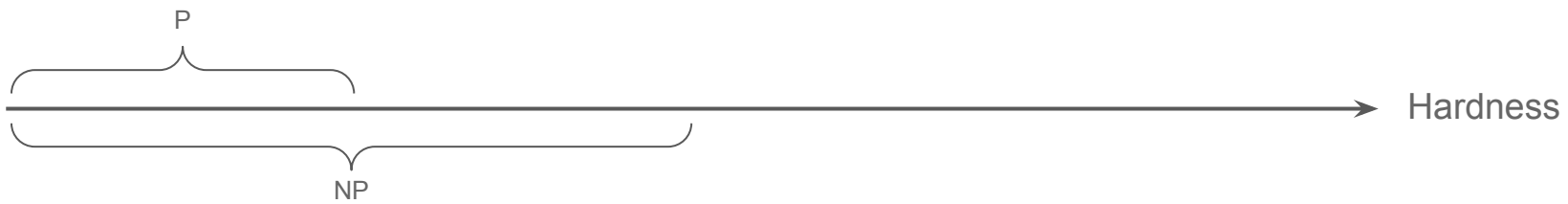
- If problem is in P, and you are given a ‘proposed solution’. How to check?
  - Just run the polynomial time solution to the problem, and return that YES or NO answer



Complexity Class: P ⊆ NP

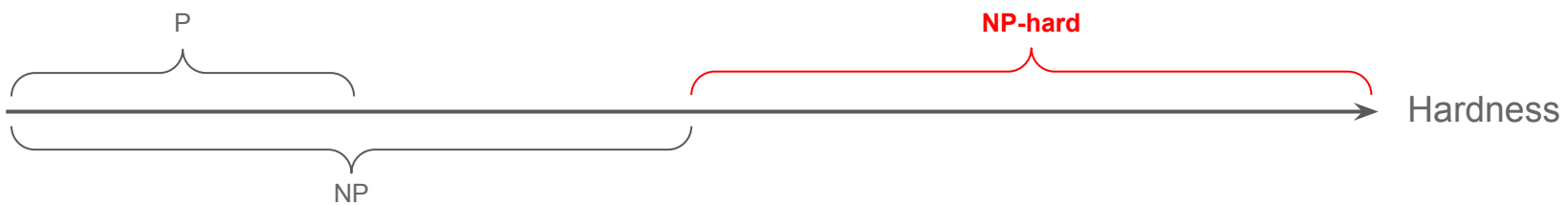
- P: The **set** of **decision** problems that can be solved in polynomial time
- NP: The **set** of **decision** problems that can be **verified** in polynomial time

- If problem is in P, and you are given a ‘proposed solution’. How to check?
  - Just run the polynomial time solution to the problem, and return that YES or NO answer
  - Thus every problem in P is also in NP



Complexity Class: NP-hard

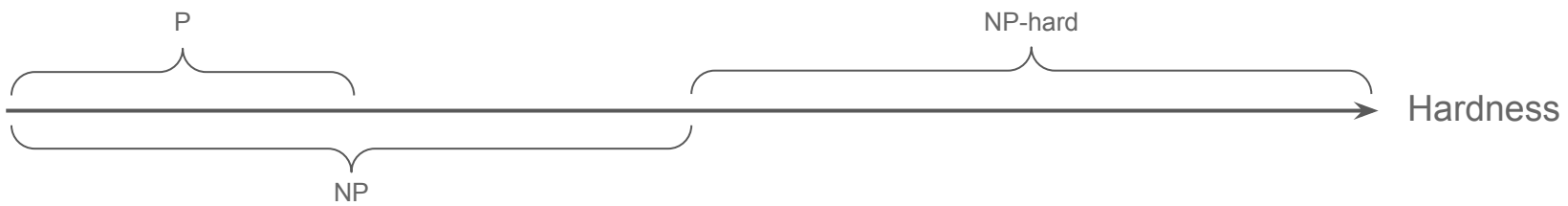
NP-hard: The set of decision problems that are “at least as hard” as the **hardest problem in NP**



Complexity Class: NP-hard

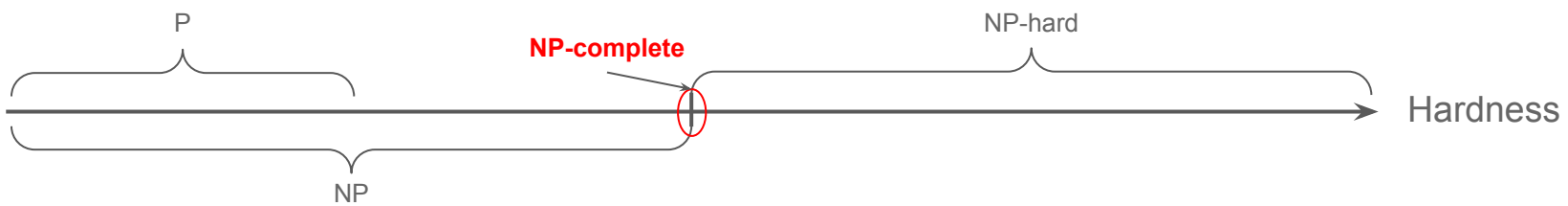
NP-hard: The set of decision problems that are “at least as hard” as the **hardest problem in NP**

How to know if a problem X is NP-hard? Stay tuned!



Complexity Class: NP-complete

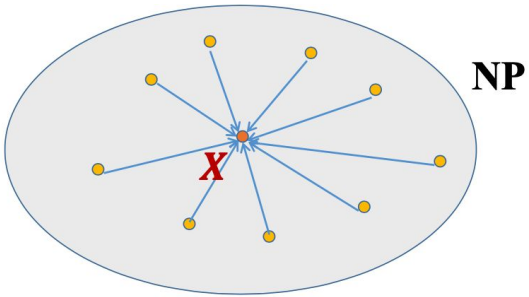
NP-complete: The set of decision problems that are 1) NP-hard and 2) in NP

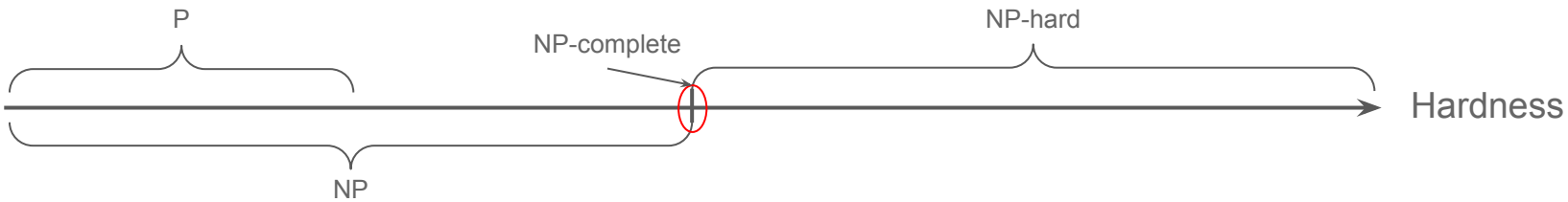


Complexity Class: NP-complete

NP-complete: The set of decision problems that are 1) NP-hard and 2) in NP

Alternative: Problem X in NP is NP-complete if for all A in NP, $A \leq_p X$



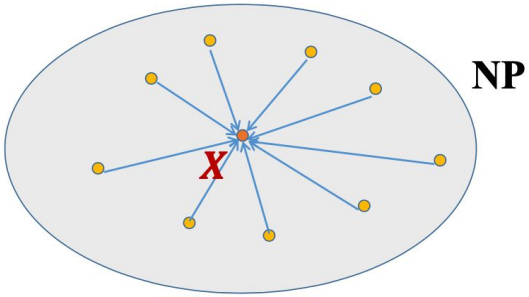


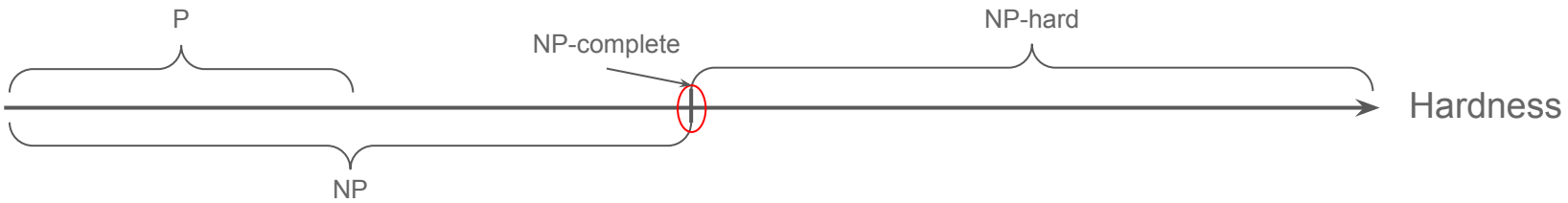
Complexity Class: NP-complete

NP-complete: The set of decision problems that are 1) NP-hard and 2) in NP

Alternative: Problem X in NP is NP-complete if for all A in NP, $A \leq_p X$

Intuition: they are the “hardest” problems in NP



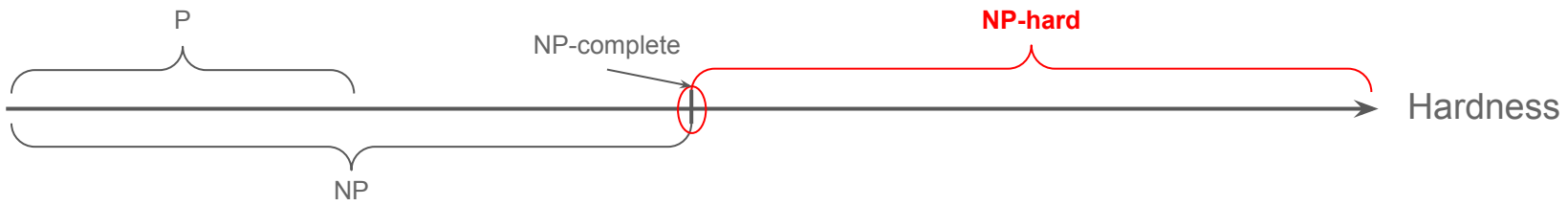


Complexity Class: NP-hard

NP-hard: The set of decision problems that are “at least as hard” as the **hardest problem in NP**

How to know if a problem X is NP-hard?

1. Take any problem A that is **known to be NP-complete**
2. Show that $A \leq_p X$



Complexity Class: NP-hard

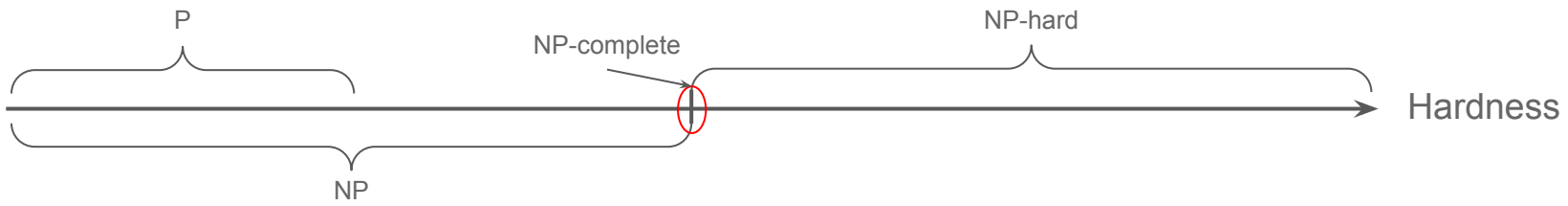
IMPORTANT

Please get the direction of reduction right!

To show a problem is “hard”, you need to reduce **FROM** a “hard” problem

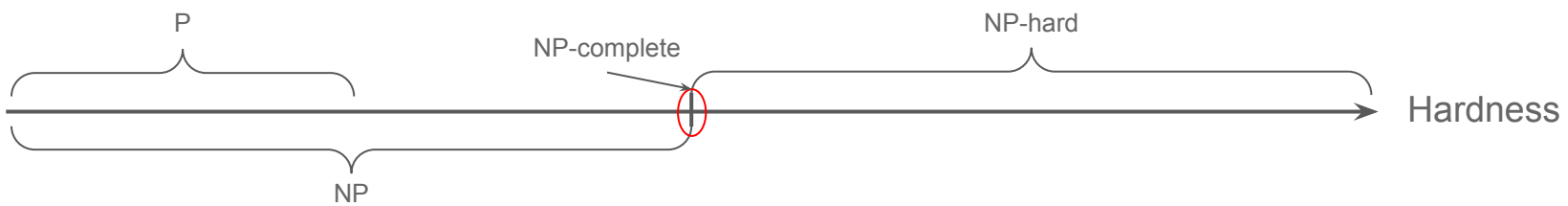
How to know if a problem X is NP-hard?

1. Take any problem A that is **known to be NP-complete**
2. Show that $A \leq_p X$



Complexity Class: P vs NP

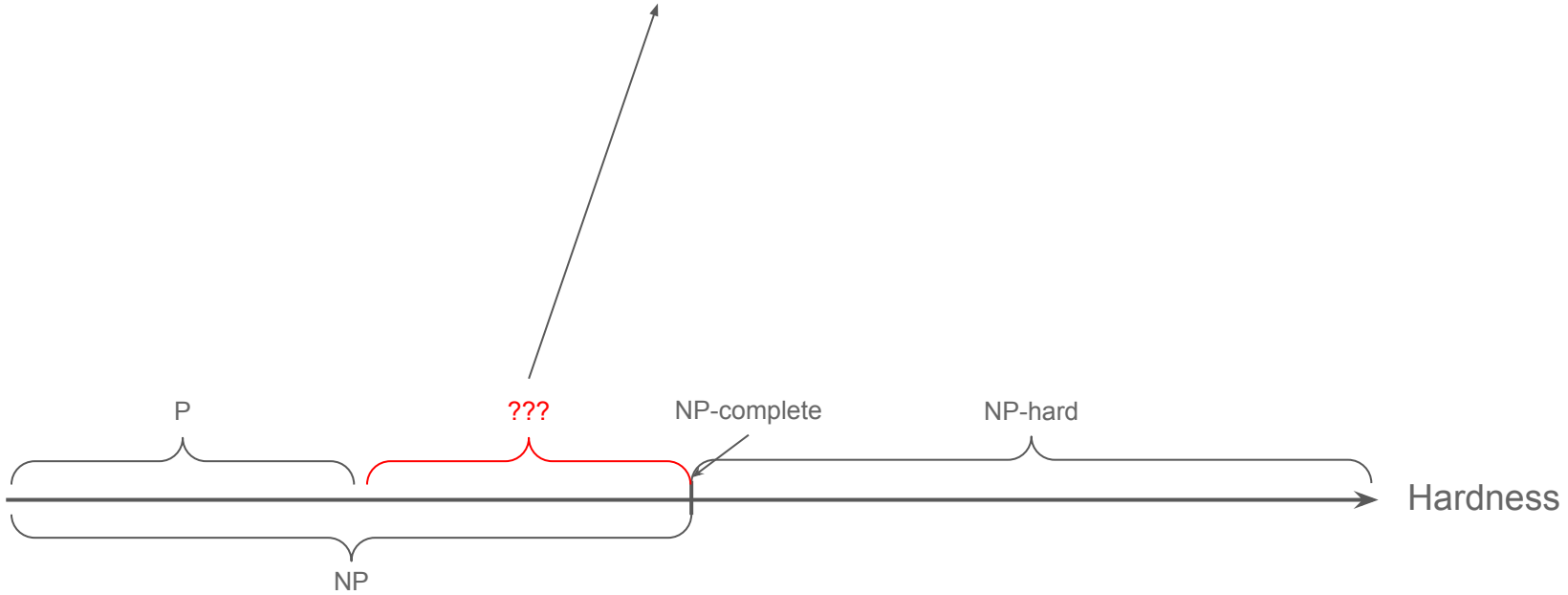
The biggest unanswered question: is P = NP?



Complexity Class: P vs NP

The biggest unanswered question: is P = NP?

Alternative: is there a problem not in P, but in NP?



NP-complete

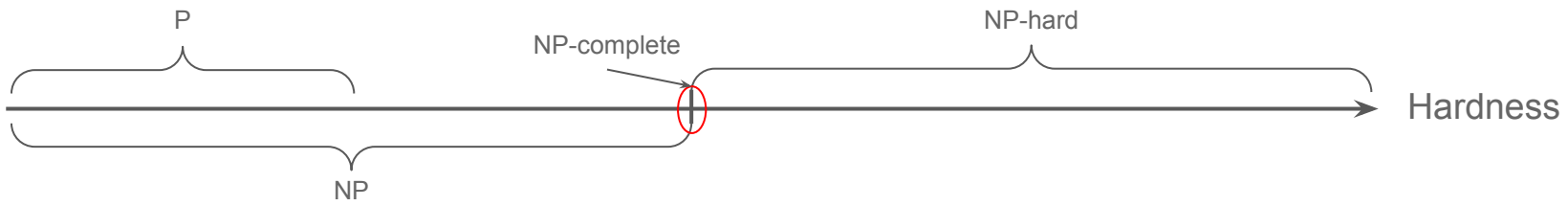
There are many problems known to be NP-complete!

Graphs and hypergraphs [edit]

Graphs occur frequently in everyday applications. Examples include biological or social networks, which contain hundreds, thousands and even billions of nodes in some cases (e.g. Facebook or LinkedIn).

- 1-planarity[1]
- 3-dimensional matching[2][3]
- Bipartite dimension[4]
- Capacitated minimum spanning tree[5]
- Route inspection problem (also called **Chinese postman problem**) for mixed graphs (having both directed and undirected edges). The program is solvable in polynomial time if the graph has all undirected or all directed edges. Variants include the rural postman problem.[6]
- Clique problem[2][7]
- Complete coloring, a.k.a. achromatic number[8]
- Domatic number[9]
- Dominating set, a.k.a. domination number[10]

  NP-complete special cases include the edge dominating set problem, i.e., the dominating set problem in line graphs. NP-complete variants include the connected dominating set problem and the maximum leaf spanning tree problem.[11]



NP-complete

There are many problems known to be NP-complete!

- If we know polynomial solution to just **one** of them, we will get efficient solution for **ALL** of them
  (Because they are reducible to each other)

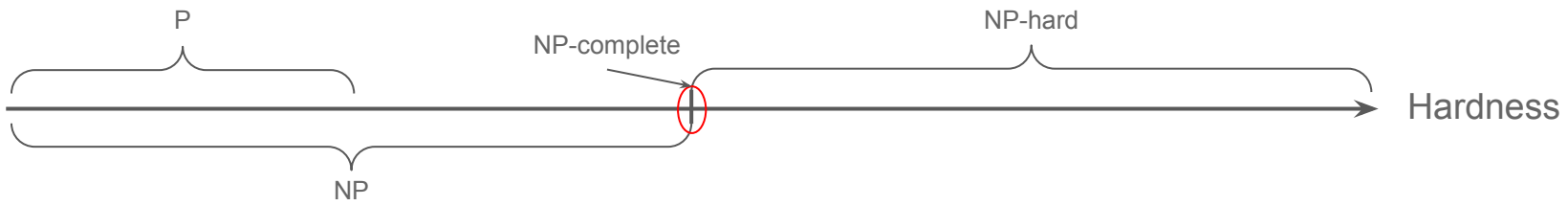
- But right now, we don't have any efficient solution

Graphs and hypergraphs [edit]

Graphs occur frequently in everyday applications. Examples include biological or social networks, which contain hundreds, thousands and even billions of nodes in some cases (e.g. Facebook or LinkedIn).

- 1-planarity[1]
- 3-dimensional matching[2][3]
- Bipartite dimension[4]
- Capacitated minimum spanning tree[5]
- Route inspection problem (also called **Chinese postman problem**) for mixed graphs (having both directed and undirected edges). The program is solvable in polynomial time if the graph has all undirected or all directed edges. Variants include the rural postman problem.[6]
- Clique problem[2][7]
- Complete coloring, a.k.a. achromatic number[8]
- Domatic number[9]
- Dominating set, a.k.a. domination number[10]

  NP-complete special cases include the edge dominating set problem, i.e., the dominating set problem in line graphs. NP-complete variants include the connected dominating set problem and the maximum leaf spanning tree problem.[11]



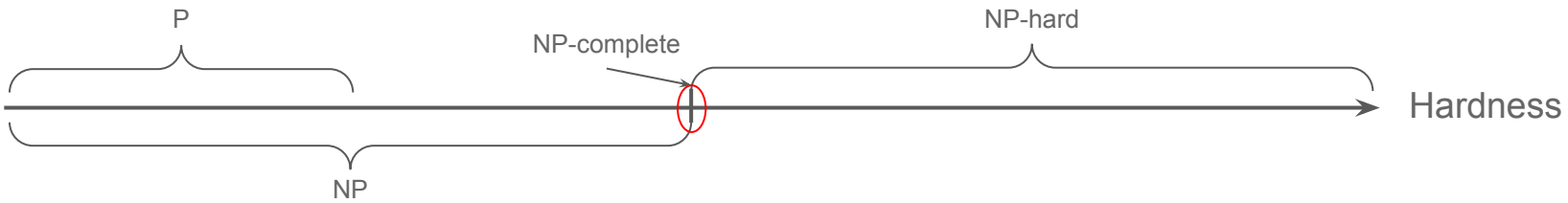
Question 1: When is P = NP?

Question 1

NUS National University of Singapore

Which of the following imply P=NP?

1. There is a problem in P that is also in NP-COMPLETE.
2. There is a problem in P that is also in NP.
3. There is a problem in NP that is also in NP-HARD.

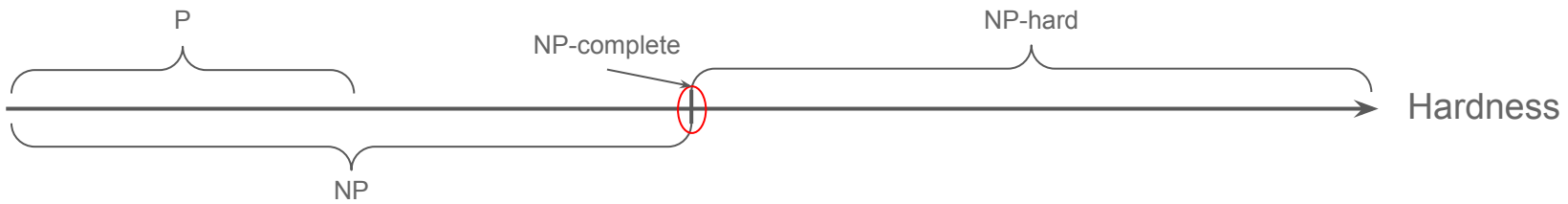


Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

Question 1: Solution

Option 1: There is a problem in P that is also NP-complete (True)



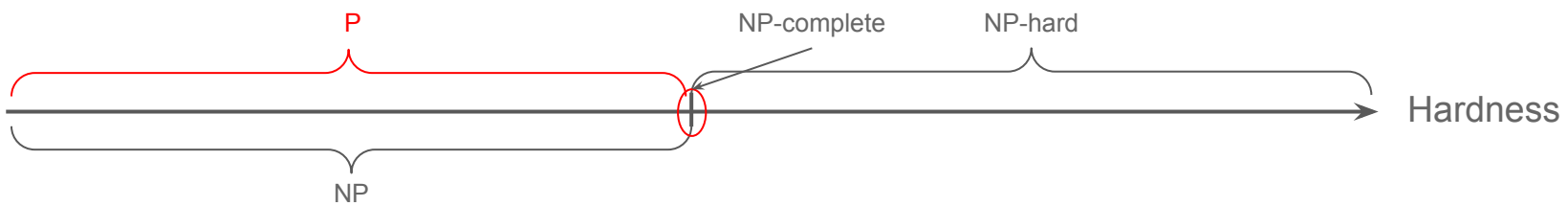
Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

Question 1: Solution

Option 1: There is a problem in P that is also NP-complete (True)

Intuition: the set of P now reaches NP-complete



Last Week: Reductions

```
def solve_A(input_A):
1.  input_B = reduction_from_A_to_B(input_A)
2.  output_B = solve_B(input_B) # magically given
3.
4.  output_A = transform_output_B_to_A(output_B)
5.  return output_A
```

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is “easily solvable”, then so is A
- If A is “hard”, then so is B

Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

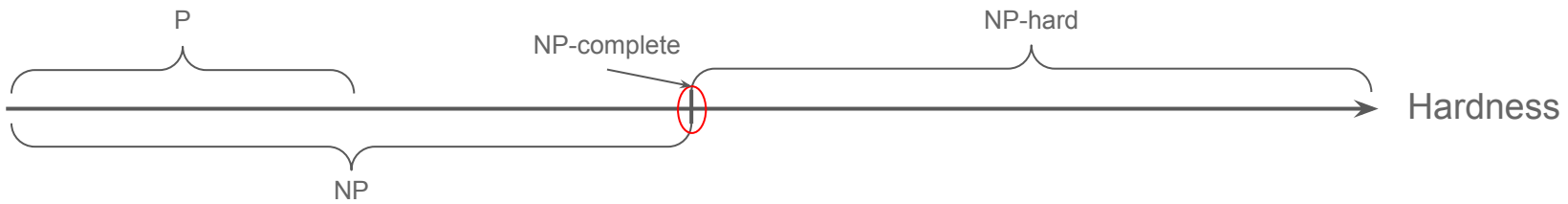
Question 1: Solution

Option 1: There is a problem in P that is also NP-complete (True)

Intuition: the set of P now reaches NP-complete

Formally:

- (NP-complete problems = some NP-hard problems) solvable in polytime



Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

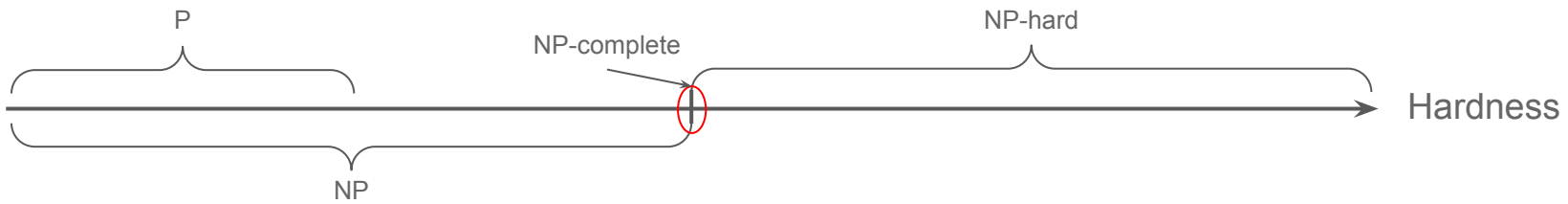
Question 1: Solution

Option 1: There is a problem in P that is also NP-complete (True)

Intuition: the set of P now reaches NP-complete

Formally:

- (NP-complete problems = some NP-hard problems) solvable in polytime
- *All NP $\leq_p$ NP-hard problems*, by definition



Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

Question 1: Solution

Option 1: There is a problem in P that is also NP-complete

Intuition: the set of P now reaches NP-complete

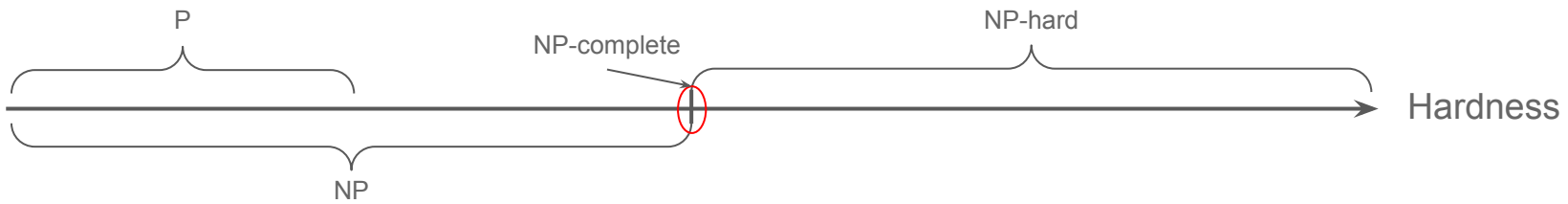
Notation for poly-time reduction from A to B:

$A \leq_p B$

- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A
- If A is "hard", then so is B

Formally:

- (NP-complete problems = some NP-hard problems) solvable in polytime
- *All NP* $\leq_p$ *NP-hard problems*, by definition
- NP-hard problems solvable in polytime → All NP problems solvable in polytime



Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

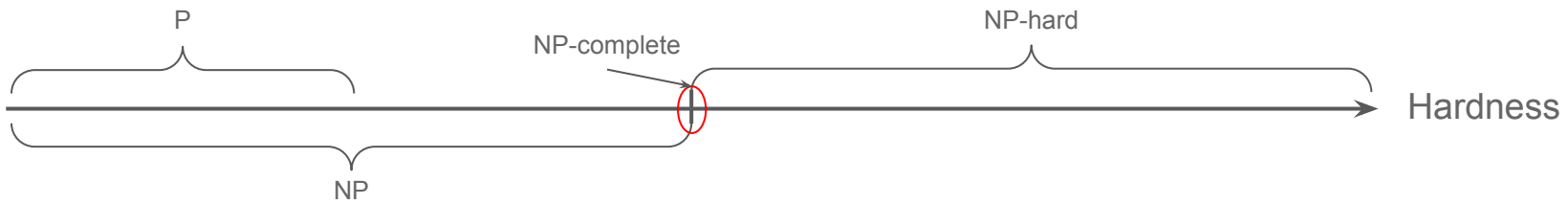
Question 1: Solution

Option 1: There is a problem in P that is also NP-complete (True)

Intuition: the set of P now reaches NP-complete

Formally:

- (NP-complete problems = some NP-hard problems) solvable in polytime
- *All NP* $\leq_p$ *NP-hard problems*, by definition
- NP-hard problems solvable in polytime $\rightarrow$ All NP problems solvable in polytime $\rightarrow$ P = NP, by definition of P



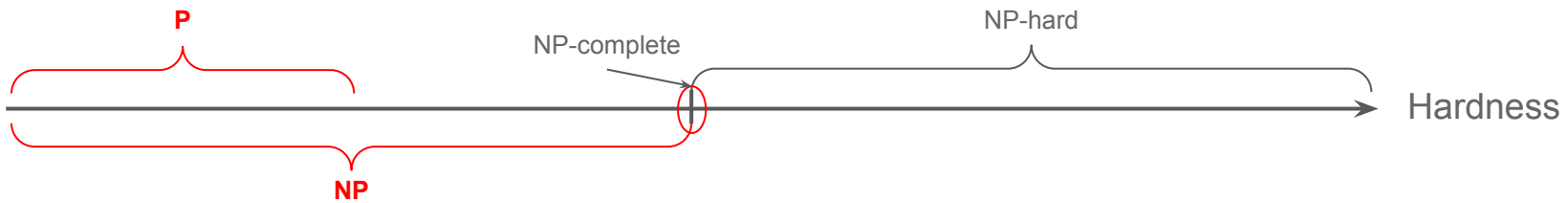
Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

Question 1: Solution

Option 2: There is a problem in P that is also in NP (False)

We have earlier argued that P ⊆ NP, so this does not imply anything about P = NP



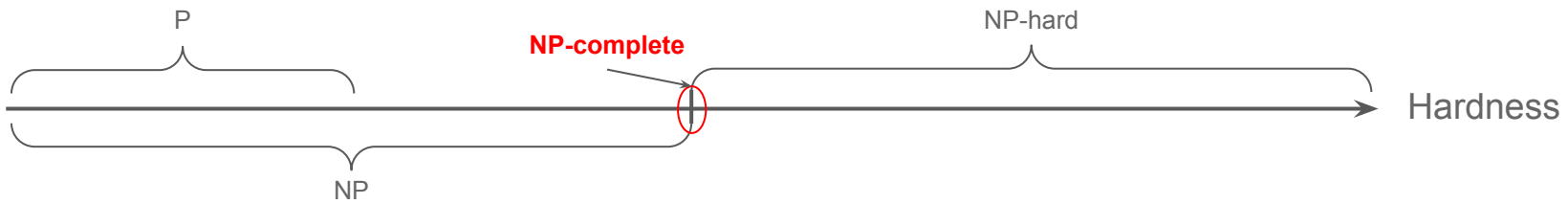
Which of the following imply P = NP?

1. There is a problem in P that is also NP-complete
2. There is a problem in P that is also in NP
3. There is a problem in NP that is also NP-hard

Question 1: Solution

Option 3: There is a problem in NP that is also NP-hard (False)

NP-complete problems are such problems. Also does not imply P = NP

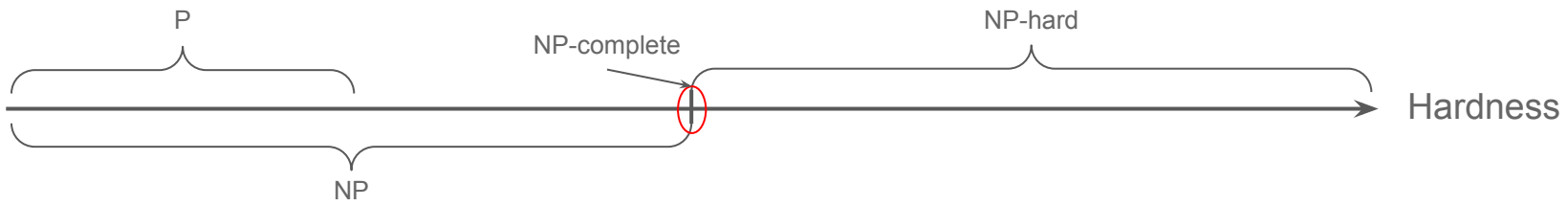


Question 2: What if there is no polytime for problems in NP?

Q2:

L is an NP problem. One day, Mr Oh proves that it is impossible to find an algorithm to solve L in polynomial time. After this big event, which of the following classes of problems are now **known** to be **not solvable** in polynomial time?

1. All NP-HARD problems.
2. All NP problems
3. All NP-COMPLETE problems.



Last Week: Reductions

```
def solve_A(input_A):
1.  input_B = reduction_from_A_to_B(input_A)
2.  output_B = solve_B(input_B) # magically given
3.
4.  output_A = transform_output_B_to_A(output_B)
5.  return output_A
```

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is “easily solvable”, then so is A
- If A is “hard”, then so is B

Suppose L in NP is proven to not be solvable in polytime, which can't be solved in polytime now?

1. All NP-hard problems
2. All NP problems
3. All NP-complete problems

Notation for poly-time reduction from A to B:

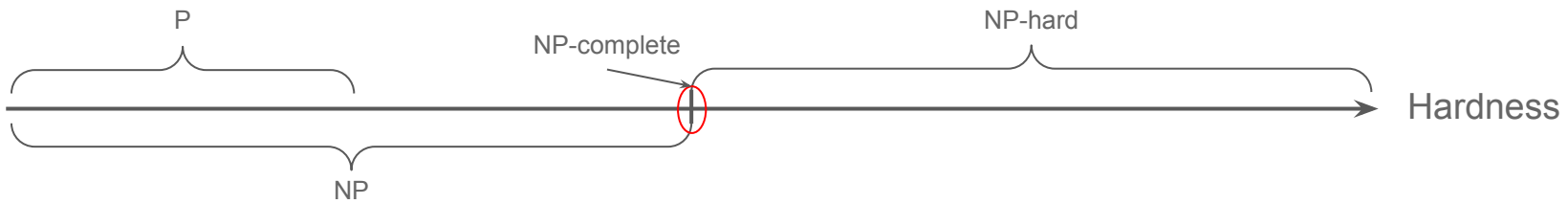
$A \leq_p B$

- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A
- If A is "hard", then so is B

Question 2: Solution

Option 1: All NP-hard Problems (True)

- L is now "hard"



Suppose L in NP is proven to not be solvable in polytime, which can't be solved in polytime now?

1. All NP-hard problems
2. All NP problems
3. All NP-complete problems

Notation for poly-time reduction from A to B:

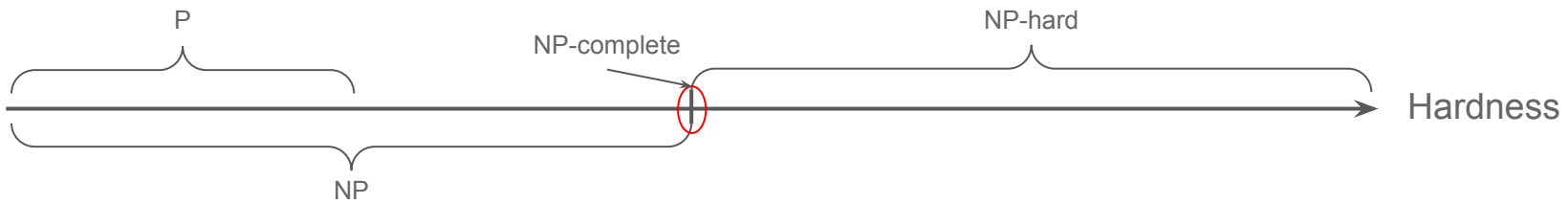
$A \leq_p B$

- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A
- If A is "hard", then so is B

Question 2: Solution

Option 1: All NP-hard Problems (True)

- L is now "hard"
- *$L \leq_p$ NP-hard problems*
- NP-hard problems are also "hard"



Suppose L in NP is proven to not be solvable in polytime, which can't be solved in polytime now?

1. All NP-hard problems
2. All NP problems
3. All NP-complete problems

Notation for poly-time reduction from A to B:

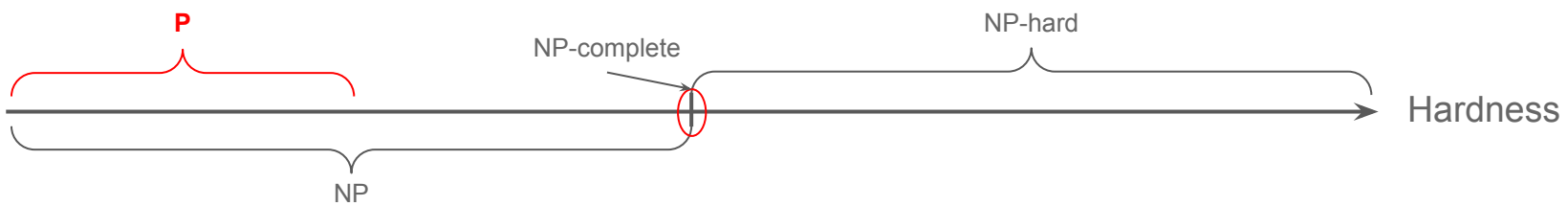
$A \leq_p B$

- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A
- If A is "hard", then so is B

Question 2: Solution

Option 2: All NP problems (False)

- P ⊆ NP, and P can be solved in polytime (by definition)



Suppose L in NP is proven to not be solvable in polytime, which can't be solved in polytime now?

1. All NP-hard problems
2. All NP problems
3. All NP-complete problems

Notation for poly-time reduction from A to B:

$A \leq_p B$

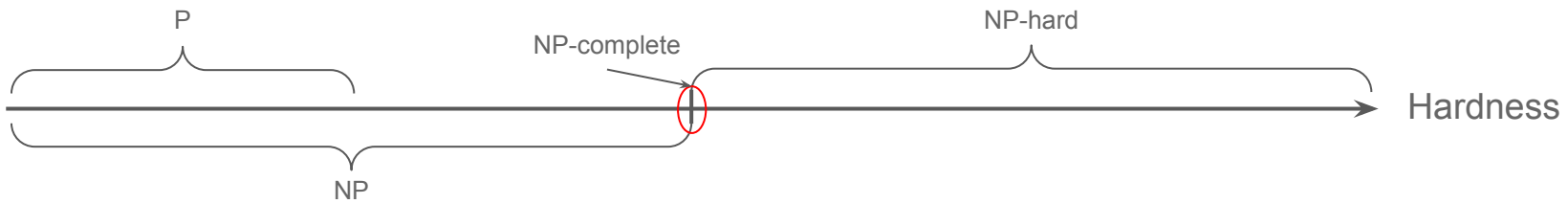
- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A
- If A is "hard", then so is B

Question 2: Solution

Option 3: All NP-complete Problems (True)

- L is now "hard"
- *$L \leq_p$ NP-complete problems*
- NP-complete problems are also "hard"

(Same as option 1)

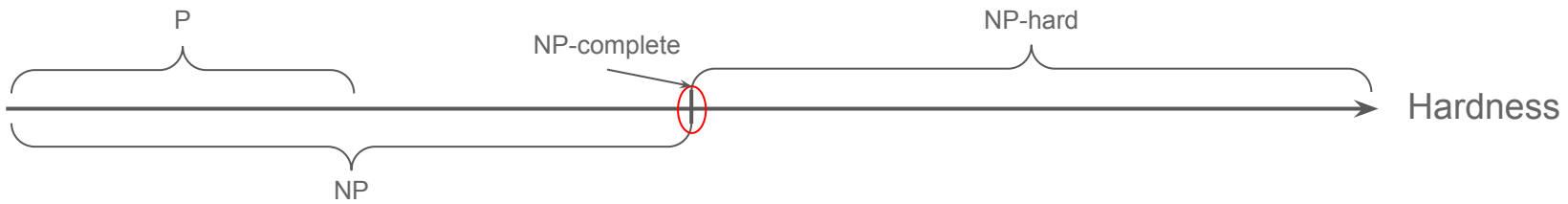


Showing that problem is NP-complete

Showing that problem is NP-complete

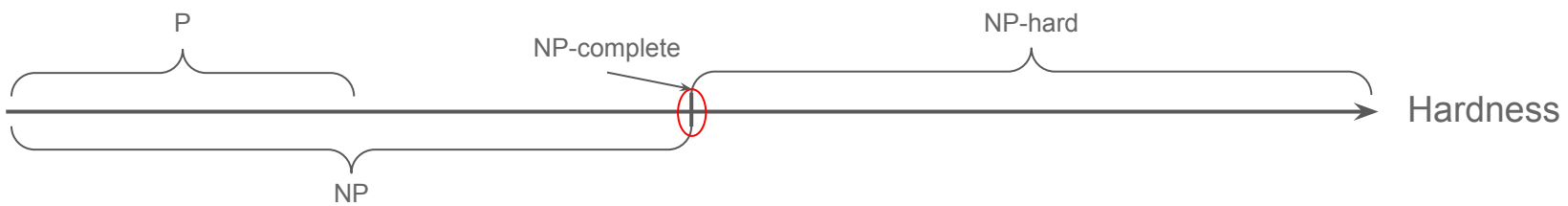
- You have a problem X and want to know whether it is an NP-complete

- The steps are actually very routine!
  - Not easy to do, but they follow the same procedure



Showing that problem is NP-complete

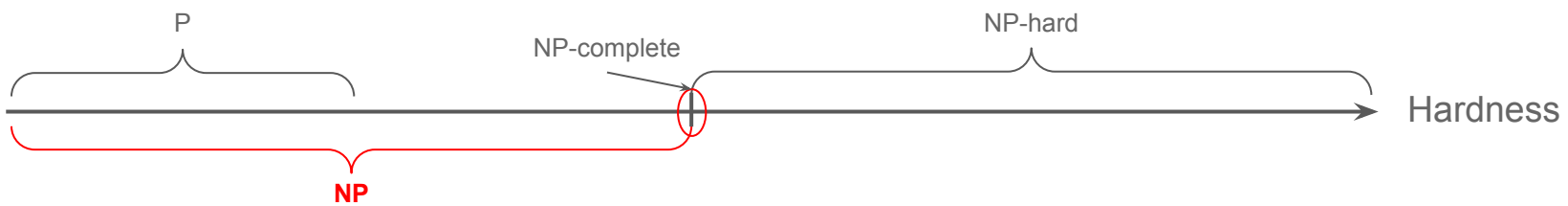
Recall: NP-complete means both in NP and NP-hard



Showing that problem is NP-complete

Recall: NP-complete means both in NP and NP-hard

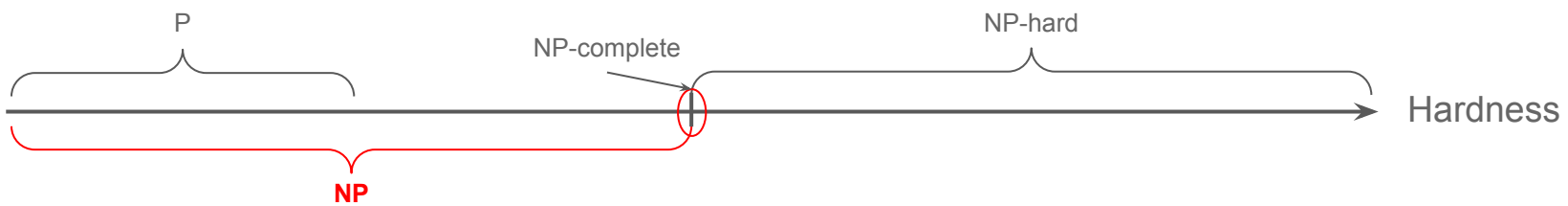
1. Show that X is in NP:



Showing that problem is NP-complete

Recall: NP-complete means both in NP and NP-hard

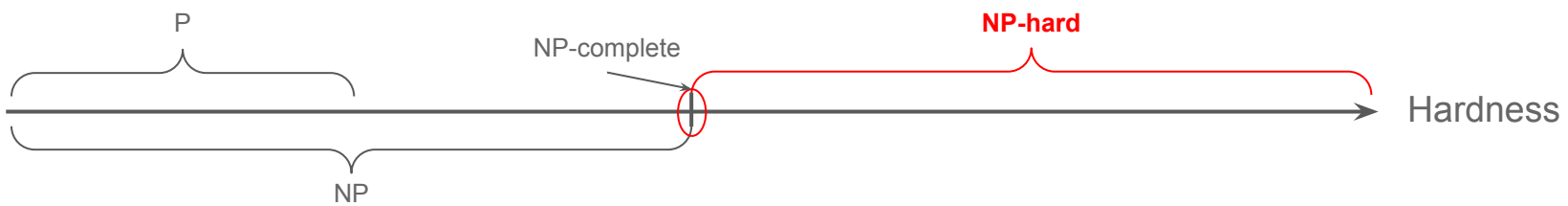
1. Show that X is in NP:
   a. Give a polynomial-sized certificate format for X (proposed solution)
   b. Give a polynomial-time algorithm to **verify** the certificate (how to check proposed solution is legal)



Showing that problem is NP-complete

Recall: NP-complete means both in NP and NP-hard

2. Show that X is NP-hard:
   a. Take a problem A **known to be NP-complete**
   b. Show that $A \leq_p X$



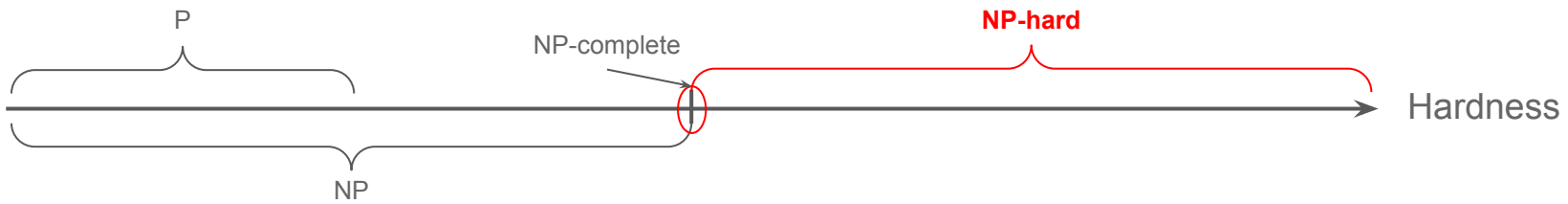
Showing that problem is NP-complete

Recall: NP-complete means both in NP and NP-hard

2. Show that X is NP-hard:
   a. Take a problem A **known to be NP-complete**
   b. Show that $A \leq_p X$

$A \leq_p X$ is a **polynomial time reduction** between decision problems A and X, when it transforms `input_A` of problem A to `input_X` of problem X such that:

- `input_A` is YES-instance → `input_X` is also a YES-instance
- `input_X` is YES-instance → `input_A` is also a YES-instance
- The transformation takes polynomial time in the size of `input_A`



IMPORTANT

Please get the direction of reduction right!

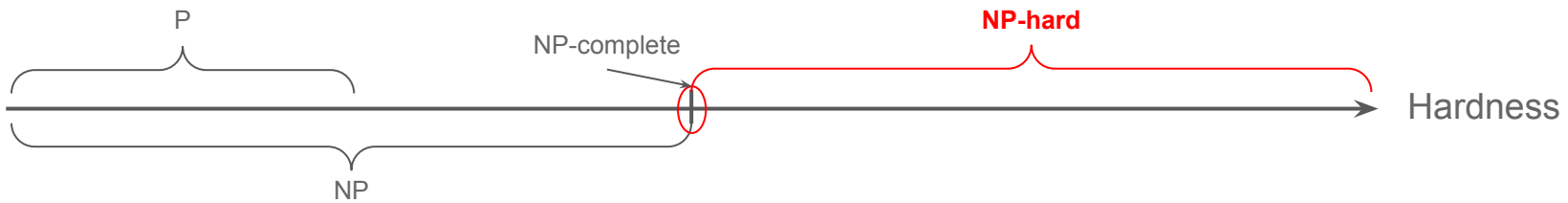
To show a problem is “hard”, you need to reduce **FROM** a “hard” problem

Recall: NP-complete means both in NP and NP-hard

2. Show that X is NP-hard:
   a. Take a problem **A known to be NP-complete**
   b. Show that $A \leq_p X$

$A \leq_p X$ is a **polynomial time reduction** between decision problems A and X, when it transforms `input_A` of problem A to `input_X` of problem X such that:

- `input_A` is YES-instance → `input_X` is also a YES-instance
- `input_X` is YES-instance → `input_A` is also a YES-instance
- The transformation takes polynomial time in the size of `input_A`

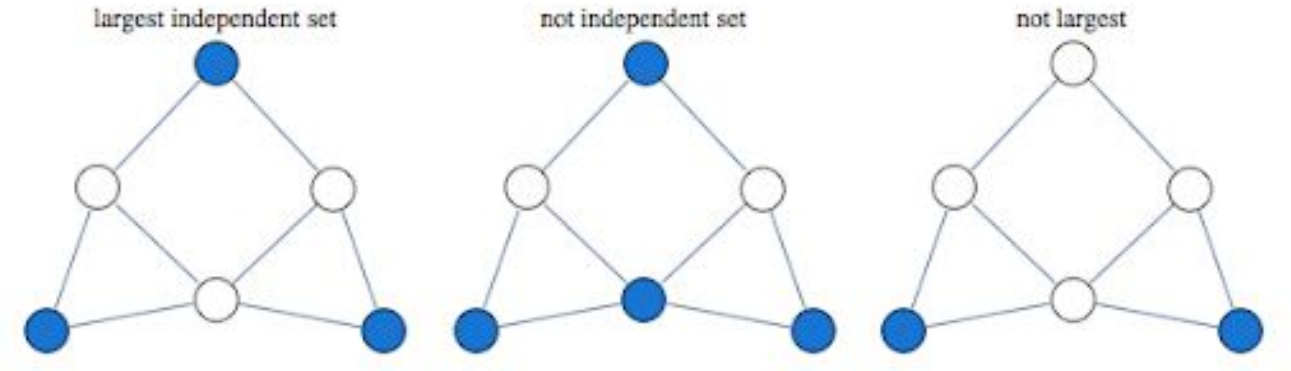


Question 3:
INDEPENDENT-SET to CLIQUE

INDEPENDENT-SET Problem

Given a graph *G = (V, E)*, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge

INDEPENDENT-SET (IS) problem: Given a graph *G* and integer *k*, is there an independent set of size at least *k*?



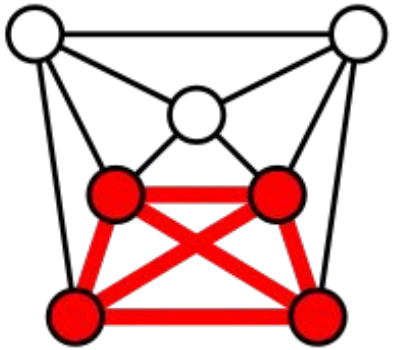
IS of size 3

IS of size 2

CLIQUE Problem

A set of vertices *U* of a graph *G* is a **clique** if every pair of vertices in *U* has an edge in *G*. Intuitively, a subgraph of G is a complete graph.

CLIQUE problem: Given a graph G and integer k, is there a clique of size at least k?



A graph with
clique of size 4

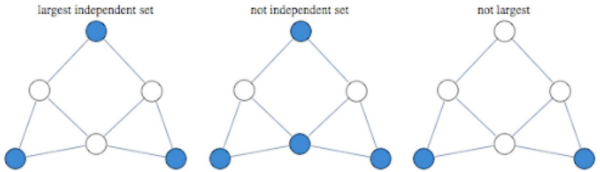
Question

Show that the CLIQUE problem is NP-complete. (Try a reduction from Independent Set)

INDEPENDENT-SET Problem

Given a graph *G = (V, E)*, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge

INDEPENDENT-SET (IS) problem: Given a graph *G* and integer *k*, is there an independent set of size at least *k*?



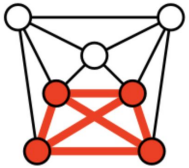
IS of size 3

IS of size 2

CLIQUE Problem

A set of vertices *U* of a graph *G* is a **clique** if every pair of vertices in *U* has an edge in *G*. Intuitively, a subgraph of G is a complete graph.

CLIQUE problem: Given a graph G and integer k, is there a clique of size at least k?



A graph with clique of size 4

Quick check

To show that CLIQUE is NP-complete, we will do a reduction from INDEPENDENT-SET.

Does this mean we are going to design an input for INDEPENDENT-SET problem or design an input for CLIQUE problem?

Quick check

To show that CLIQUE is NP-complete, we will do a reduction from INDEPENDENT-SET.

Does this mean we are going to design an input for INDEPENDENT-SET problem or design an input for CLIQUE problem?

We start from an input of INDEPENDENT-SET, and design an input for the CLIQUE problem!

INDEPENDENT-SET $\leq_p$ CLIQUE

Recall the pseudocode:

INDEPENDENT-SET $\leq_p$ CLIQUE

Recall the pseudocode:

```
def INDEPENDENT-SET(G, k) -> bool:
1.  G', k' = reduction(G, k)
2.  yes_or_no: bool = CLIQUE(G', k') # magically given
3.  return yes_or_no
```

G' and k' may or may not be the same as the original input

INDEPENDENT-SET $\leq_p$ CLIQUE

G' and k' may or may not be the same as the original input

Recall the pseudocode:

```
def INDEPENDENT-SET(G, k) -> bool:
1.  G', k' = reduction(G, k)
2.  yes_or_no: bool = CLIQUE(G', k') # magically given
3.  return yes_or_no
```

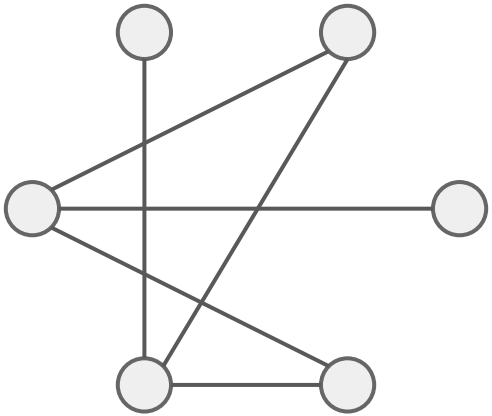
What to show for a correct reduction:

- `(G, k)` is YES-instance → `(G', k')` is also a YES-instance
- `(G', k')` is YES-instance → `(G, k)` is also a YES-instance
- The transformation takes polynomial time in the size of `(G, k)`

Given a graph *G = (V, E)*, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge
INDEPENDENT-SET (IS) problem: Given a graph *G* and integer *k*, is there an independent set of size at least *k*?

Building up the idea for the reduction

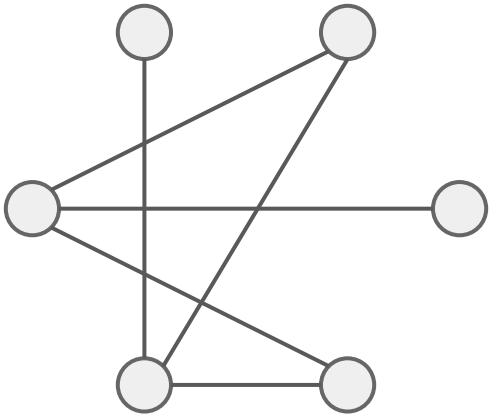
Let’s say the following graph is the input to INDEPENDENT-SET, with k = 4



Given a graph $G = (V, E)$, **independent set** is a subset of vertices V such that no two vertices in the graph is connected by an edge
INDEPENDENT-SET (IS) problem: Given a graph G and integer k, is there an independent set of size at least k?

Building up the idea for the reduction

Let's say the following graph is the input to INDEPENDENT-SET, with k = 4



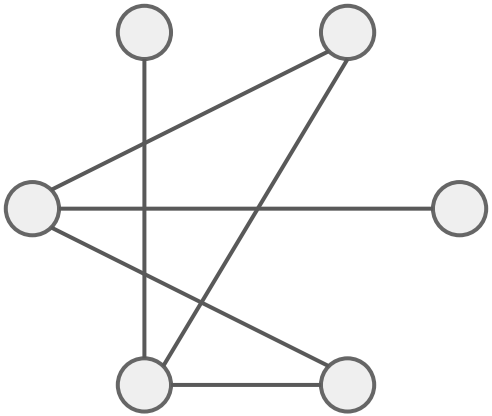
Questions to ask: Should we modify this graph for the input to CLIQUE? Should we modify k?

Given a graph $G = (V, E)$, **independent set** is a subset of vertices V such that no two vertices in the graph is connected by an edge
INDEPENDENT-SET (IS) problem: Given a graph G and integer k, is there an independent set of size at least k?

Building up the idea for the reduction

Let's say the following graph is the input to INDEPENDENT-SET, with k = 4

If this was a YES-instance to IS, can it be / how can it be modified to be a YES-instance to CLIQUE?



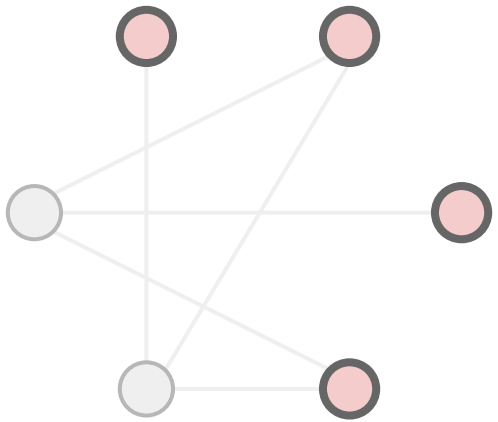
Questions to ask: Should we modify this graph for the input to CLIQUE? Should we modify k?

Given a graph $G = (V, E)$, **independent set** is a subset of vertices V such that no two vertices in the graph is connected by an edge
INDEPENDENT-SET (IS) problem: Given a graph G and integer k, is there an independent set of size at least k?

Building up the idea for the reduction

Let's say the following graph is the input to INDEPENDENT-SET, with k = 4

If this was a YES-instance to IS, can it be / how can it be modified to be a YES-instance to CLIQUE?



Indeed, this is a YES-Instance to IS!

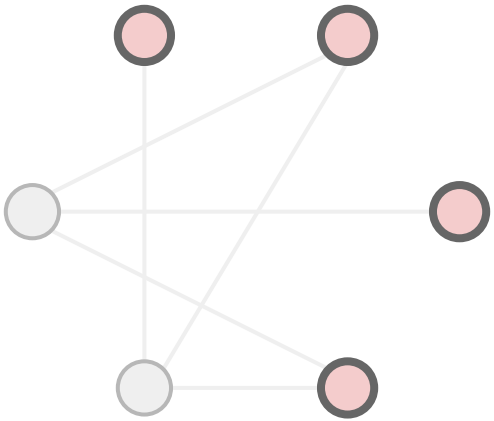
Questions to ask: Should we modify this graph for the input to CLIQUE? Should we modify k?

Given a graph $G = (V, E)$, **independent set** is a subset of vertices V such that no two vertices in the graph is connected by an edge
INDEPENDENT-SET (IS) problem: Given a graph G and integer k, is there an independent set of size at least k?

Building up the idea for the reduction

Let's say the following graph is the input to INDEPENDENT-SET, with k = 4

If this was a YES-instance to IS, can it be / how can it be modified to be a YES-instance to CLIQUE?



Indeed, this is a YES-Instance to IS!

The 4 nodes that form the independent set is the "total opposite" of a complete graph (for clique problem)

Questions to ask: Should we modify this graph for the input to CLIQUE? Should we modify k?

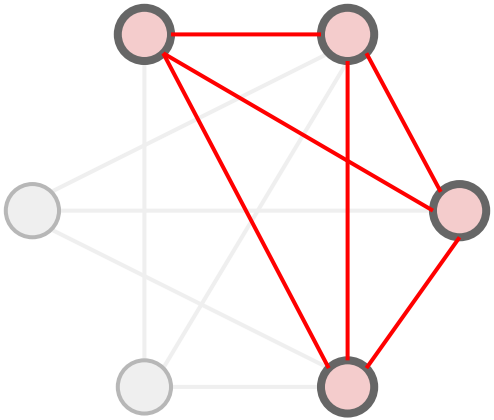
Given a graph $G = (V, E)$, **independent set** is a subset of vertices V such that no two vertices in the graph is connected by an edge
INDEPENDENT-SET (IS) problem: Given a graph G and integer k, is there an independent set of size at least k?

Building up the idea for the reduction

Let's say the following graph is the input to INDEPENDENT-SET, with k = 4

If this was a YES-instance to IS, can it be / how can it be modified to be a YES-instance to CLIQUE?

Graph that is a YES-instance to CLIQUE



Note: We haven't specified how exactly to modify yet

Indeed, this is a YES-Instance to IS!

The 4 nodes that form the independent set is the "total opposite" of a complete graph (for clique problem)

Idea: Somehow modify graph to add the edges to form clique of size 4.

Questions to ask: Should we modify this graph for the input to CLIQUE? Should we modify k?

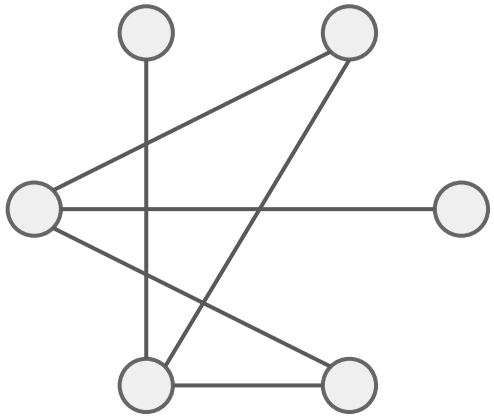
Idea for reduction

- We can deduce that the integer *k* in the IS problem and CLIQUE problem is the **same**, since the modified graph is “derived from” IS of size *k*

- We also know that from the original graph, we should somehow add all edges to the independent set to form a clique
  - But how exactly?

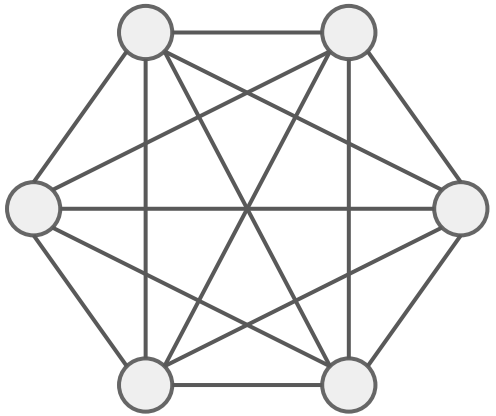
Reduction Attempt 1

Don’t know where to add the edges? Just add edges to the entire graph!



k = 4

Input to INDEPENDENT-SET problem



k = 4

Input to CLIQUE problem

Reduction Attempt 1

Recall, one of the conditions for reduction to be correct is:

Input to CLIQUE is a YES-Instance → Input to IS is a YES-Instance

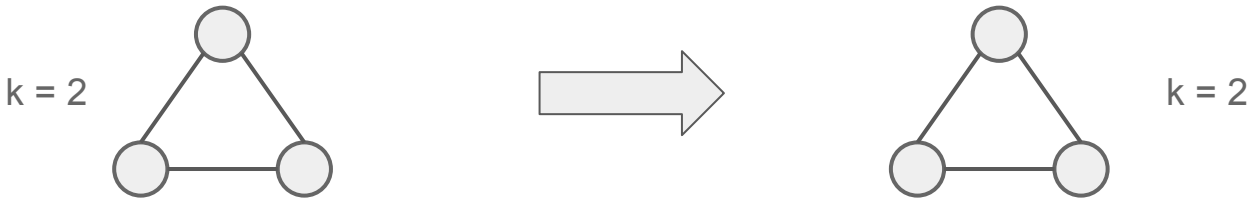
Reduction Attempt 1

Recall, one of the conditions for reduction to be correct is:

Input to CLIQUE is a YES-Instance → Input to IS is a YES-Instance

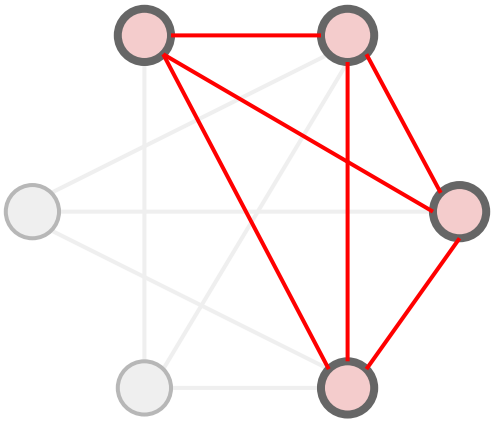
Problem with the current approach: The graph construction will always result in a YES-Instance for the CLIQUE problem, regardless of the initial graph.

Example: If input to IS problem is a complete graph, it is a NO-instance. But it is a YES-instance for CLIQUE problem



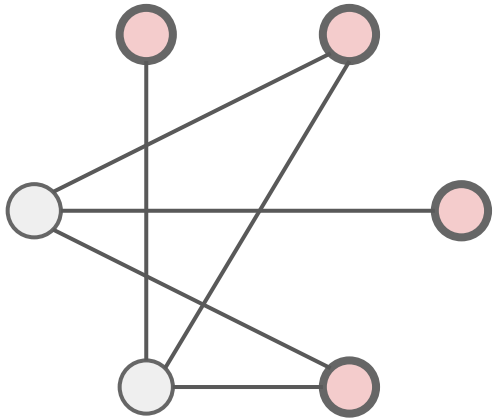
Reduction Attempt 2

Maybe we added too many edges? Notice how adding edges to the vertices involved in IS is sufficient



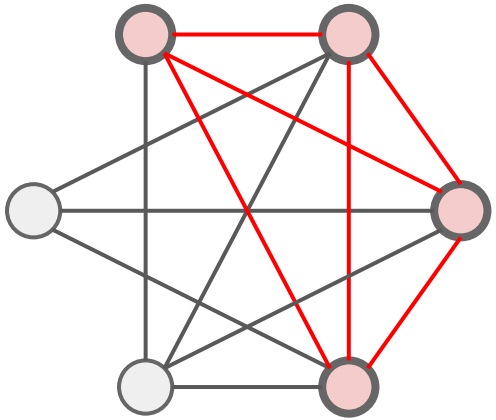
Reduction Attempt 2

1. Find vertices involved in independent set
2. Add edges to those vertices if found



k = 4

Input to INDEPENDENT-SET problem



k = 4

Input to CLIQUE problem

Reduction Attempt 2

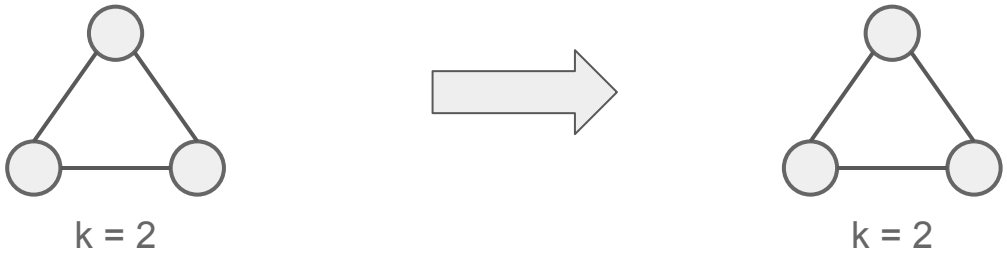
Problem #1 with this attempt: We want our reduction to run in polynomial time.

Finding the relevant vertices is itself ***the*** NP-complete Independent Set problem

Reduction Attempt 2

Problem #1 with this attempt: We want our reduction to run in polynomial time. Finding the relevant vertices is itself ***the*** NP-complete Independent Set problem

Problem #2 with this attempt: We can still form YES-instances to CLIQUE from a NO-instance of INDEPENDENT-SET.



Fixing the attempts

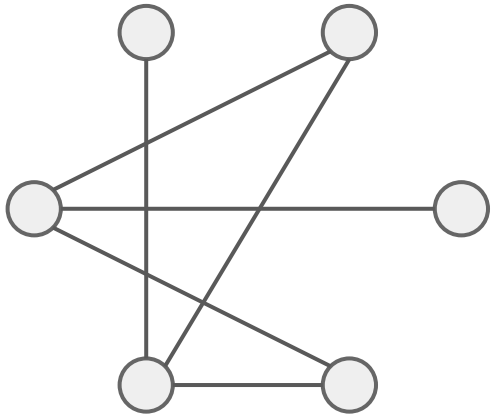
Lessons from the previous attempts:

- Aside from adding edges, might need to **remove some edges** (so that a complete graph as input to IS can be a NO-Instance to CLIQUE)

- We need to do the reduction **without any information about the existence** (or non-existence) of the Independent Set (otherwise it's like solving for IS in the first place)

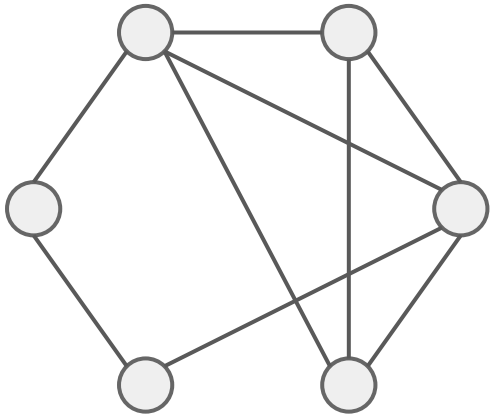
Fixed Reduction Attempt

Take the complement graph (i.e., for any two distinct nodes u and v, add the edge if no edge exists between them and remove them otherwise). Set k’ = k



k = 4

Input to INDEPENDENT-SET problem



k = 4

Input to CLIQUE problem

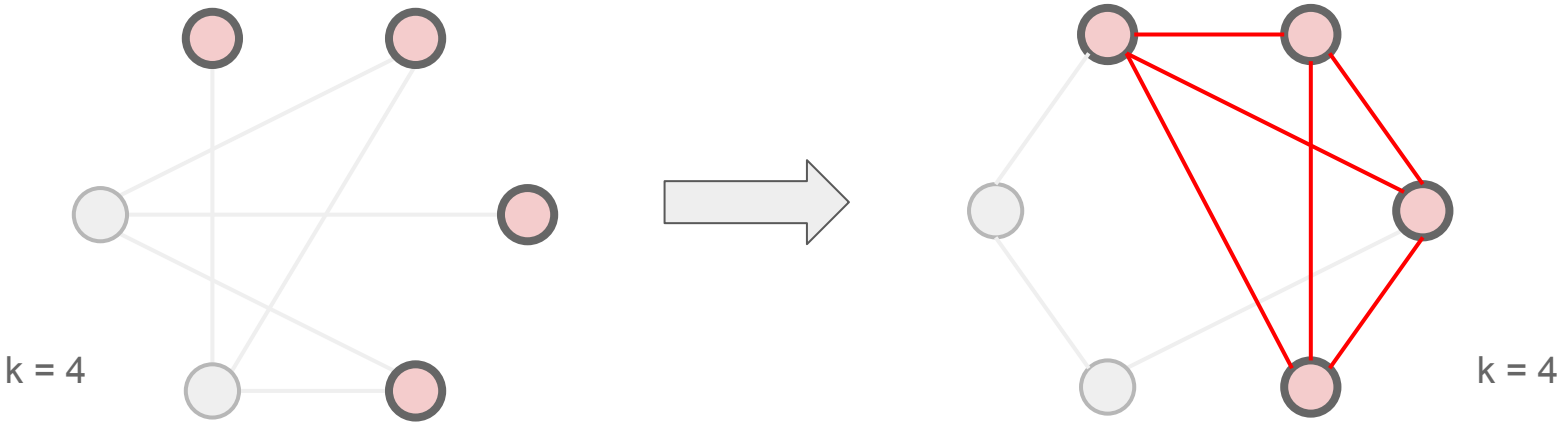
Correctness of Reduction (1)

Input to IS *(G, k)* is YES-Instance → Input to CLIQUE *(G', k')* is YES-Instance

Correctness of Reduction (1)

Input to IS *(G, k)* is YES-Instance → Input to CLIQUE *(G', k')* is YES-Instance

- Consider the independent set *S* of size at least k in *G*
- No two nodes in *S* share an edge in *G* (definition of independent set)
- All pairs in *S* share an edge in *G'* (because of complement graph)
- *S* is a clique in *G'* also of size at least k (and we set k' = k)



Correctness of Reduction (2)

Input to CLIQUE *(G', k')* is YES-Instance → Input to IS *(G, k)* is YES-Instance

Correctness of Reduction (2)

Input to CLIQUE *(G', k')* is YES-Instance → Input to IS *(G, k)* is YES-Instance

Same argument as before:

- Consider the clique *S* of size at least *k'* in *G'*
- All pairs in *S* share an edge in *G'* (definition of clique)
- No two nodes in *S* share an edge in *G* (*G'* was constructed by taking the complement of *G*)
- *S* is an independent set in *G* also of size at least k' (and we set k' = k)

Correctness of Reduction (2)

Input to CLIQUE *(G', k')* is YES-Instance → Input to IS *(G, k)* is YES-Instance

Same argument as before:

- Consider the clique *S* of size at least *k'* in *G'*
- All pairs in *S* share an edge in *G'* (definition of clique)
- No two nodes in *S* share an edge in *G* (*G'* was constructed by taking the complement of *G*)
- *S* is an independent set in *G* also of size at least k' (and we set k' = k)

In general, not all reductions will have identical arguments in both directions.
E.g. reduction from PARTITION to KNAPSACK in the previous tutorial

Correctness of Reduction (3)

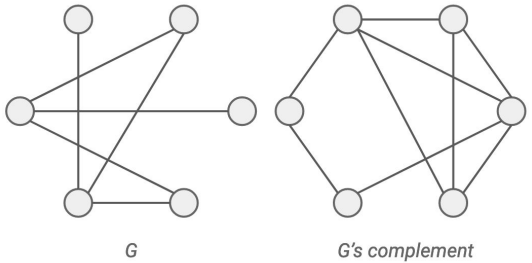
Reduction runs in polynomial time with respect to the input to IS *(G, k)*

Correctness of Reduction (3)

Reduction runs in polynomial time with respect to the input to IS *(G, k)*

In particular, it takes time $O(|V|^2)$, since we are considering all possible edges in the graph and we are either adding or removing it.

(Note that if you combine the set of edges in a graph and its complement, you will obtain all the edges -- i.e. it can form a complete graph)



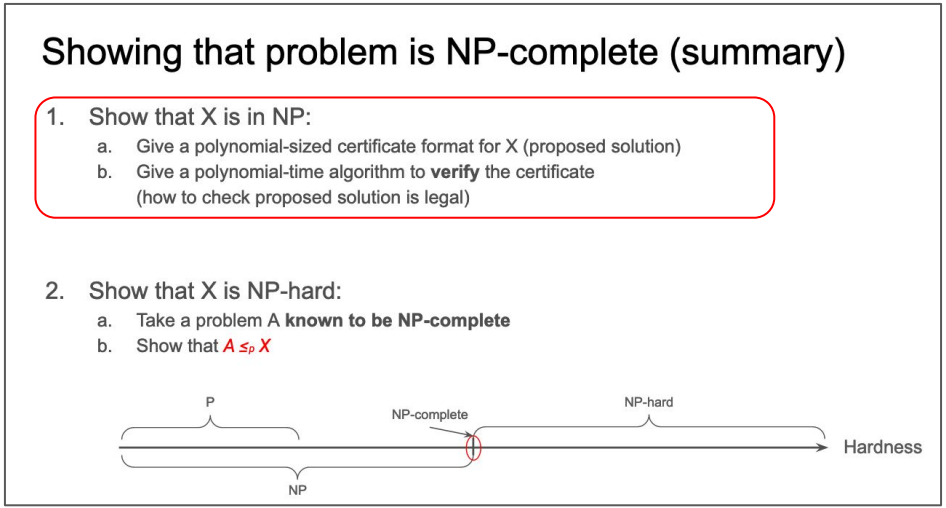
What have we shown?

Did we show that CLIQUE is NP-complete?

What have we shown?

~~Did we show that CLIQUE is NP-complete?~~ **No!**

- We only showed that CLIQUE is NP-hard by a reduction from IS
- Need to show that CLIQUE is in NP as well



A set of vertices *U* of a graph *G* is a **clique** if every pair of vertices in *U* has an edge in *G*. Intuitively, a subgraph of G is a complete graph.
CLIQUE problem: Given a graph G and integer k, is there a clique of size at least k?

Showing CLIQUE is in NP

- Certificate (proposed solution): A set of vertices *C*

A set of vertices *U* of a graph *G* is a **clique** if every pair of vertices in *U* has an edge in *G*. Intuitively, a subgraph of G is a complete graph.
CLIQUE problem: Given a graph G and integer k, is there a clique of size at least k?

Showing CLIQUE is in NP

- Certificate (proposed solution): A set of vertices *C*
- Verifier:
  - Verifies that the size of *C* is ≥ *k*
  - Verifies that every pair of nodes in *C* has an edge in *G* (it forms a clique)

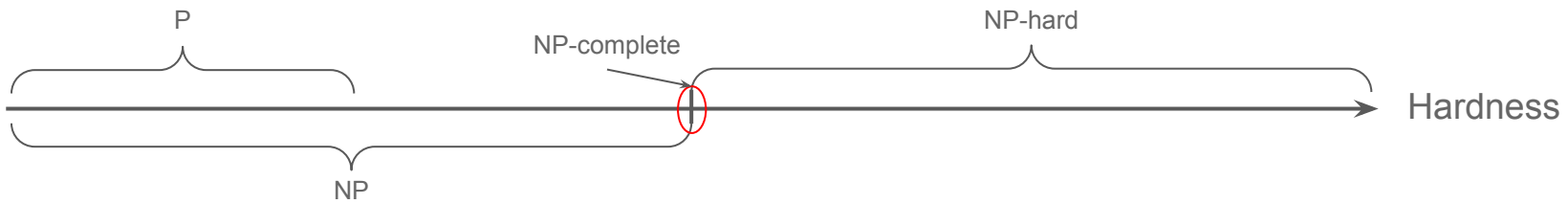
We have a polynomially-sized certificate, as well as a verifier that runs in polynomial time

Therefore, CLIQUE is also in NP

Showing that problem is NP-complete (summary)

1. Show that X is in NP:
   a. Give a polynomial-sized certificate format for X (proposed solution)
   b. Give a polynomial-time algorithm to **verify** the certificate
      (how to check proposed solution is legal)

2. Show that X is NP-hard:
   a. Take a problem A **known to be NP-complete**
   b. Show that $A \leq_p X$



That’s all for CS3230!
All the best for finals :)