Introduction to Algorithms and Data Structures

Introduction to NP-completeness

So far…

- We were given a problem A that we want to solve.

So far…

- We were given a problem A that we want to solve.
- We came up with an algorithm ALG^A that solves it.

So far…

- We were given a problem A that we want to solve.
- We came up with an algorithm ALG^A that solves it.
- We argued about the correctness of ALG^A (sometimes).

So far…

- We were given a problem A that we want to solve.
- We came up with an algorithm ALG^A that solves it.
- We argued about the correctness of ALG^A (sometimes).
- We argued about its running time.

Running time hierarchy

$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^\alpha)$	$O(c^n)$
logarithmic	linear		quadratic	polynomial	exponential
The algorithm does not even read the whole input.	The algorithm accesses the input only a constant number of times.	The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions.	The algorithm considers pairs of elements.	The algorithm performs many nested loops.	The algorithm considers many subsets of the input elements.

constant	$O(1)$	superlinear	$\omega(n)$
superconstant	$\omega(1)$	superpolynomial	$\omega(n^\alpha)$
sublinear	$o(n)$	subexponential	$o(c^n)$

Running time hierarchy

Polynomial time

$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^\alpha)$	$O(c^n)$
logarithmic	linear		quadratic	polynomial	exponential
The algorithm does not even read the whole input.	The algorithm accesses the input only a constant number of times.	The algorithm splits the inputs into two pieces of similar size, solves each part and merges the solutions.	The algorithm considers pairs of elements.	The algorithm performs many nested loops.	The algorithm considers many subsets of the input elements.

constant	$O(1)$	superlinear	$\omega(n)$
superconstant	$\omega(1)$	superpolynomial	$\omega(n^\alpha)$
sublinear	$o(n)$	subexponential	$o(c^n)$

Efficient algorithms

Efficient algorithms

- An algorithm is typically called *efficient* if it runs in polynomial time.

Efficient algorithms

- An algorithm is typically called *efficient* if it runs in polynomial time.

- If we were not interested in efficiency, we could solve all of these problems in exponential time $O(c^n)$ using brute force.

Efficient algorithms

- An algorithm is typically called *efficient* if it runs in polynomial time.
- If we were not interested in efficiency, we could solve all of these problems in exponential time $O(c^n)$ using brute force.
- If its possible to design an efficient algorithm for a problem, we shouldn’t be satisfied with brute force.

Efficient algorithms

Efficient algorithms

- Is it possible to design a polynomial-time algorithm for *every* problem?

Efficient algorithms

- Is it possible to design a polynomial-time algorithm for *every* problem?
- Are there problems for which polynomial-time algorithms do not exist?

Reductions

Polynomial Time Reduction

- We are given a problem A that we want to solve.

Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can *reduce* solving problem A to solving some other problem B.

Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can *reduce* solving problem A to solving some other problem B.
- Assume that we had an algorithm ALG^B for solving problem B.

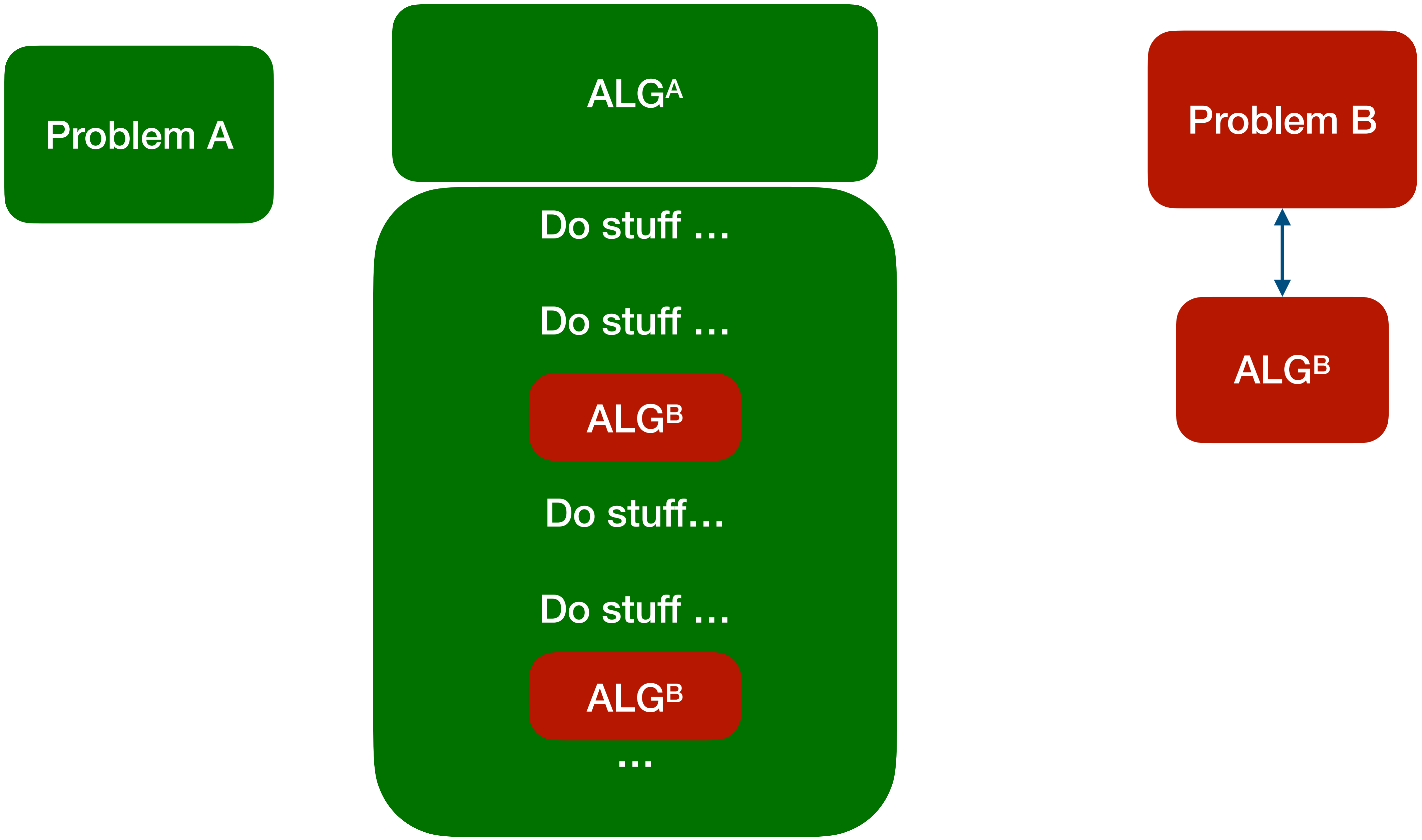
Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can *reduce* solving problem A to solving some other problem B.
- Assume that we had an algorithm ALG^B for solving problem B.
- We can construct an algorithm ALG^A for solving problem A, which uses calls to the algorithm ALG^B as a subroutine.

Polynomial Time Reduction

- We are given a problem A that we want to solve.
- We can *reduce* solving problem A to solving some other problem B.
- Assume that we had an algorithm ALG^B for solving problem B.
- We can construct an algorithm ALG^A for solving problem A, which uses calls to the algorithm ALG^B as a subroutine.
- If ALG^A is a polynomial time algorithm, then this is a *polynomial time reduction*.

Pictorially



Notation

- When problem A reduces to problem B in polynomial time, we write

  $A \leq^p B$

  We often say "there is a polynomial time reduction *from* A *to* B".

How to work with reductions

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction A $\leq^p$ B, which will give me a polynomial time algorithm for solving A.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
  - If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.

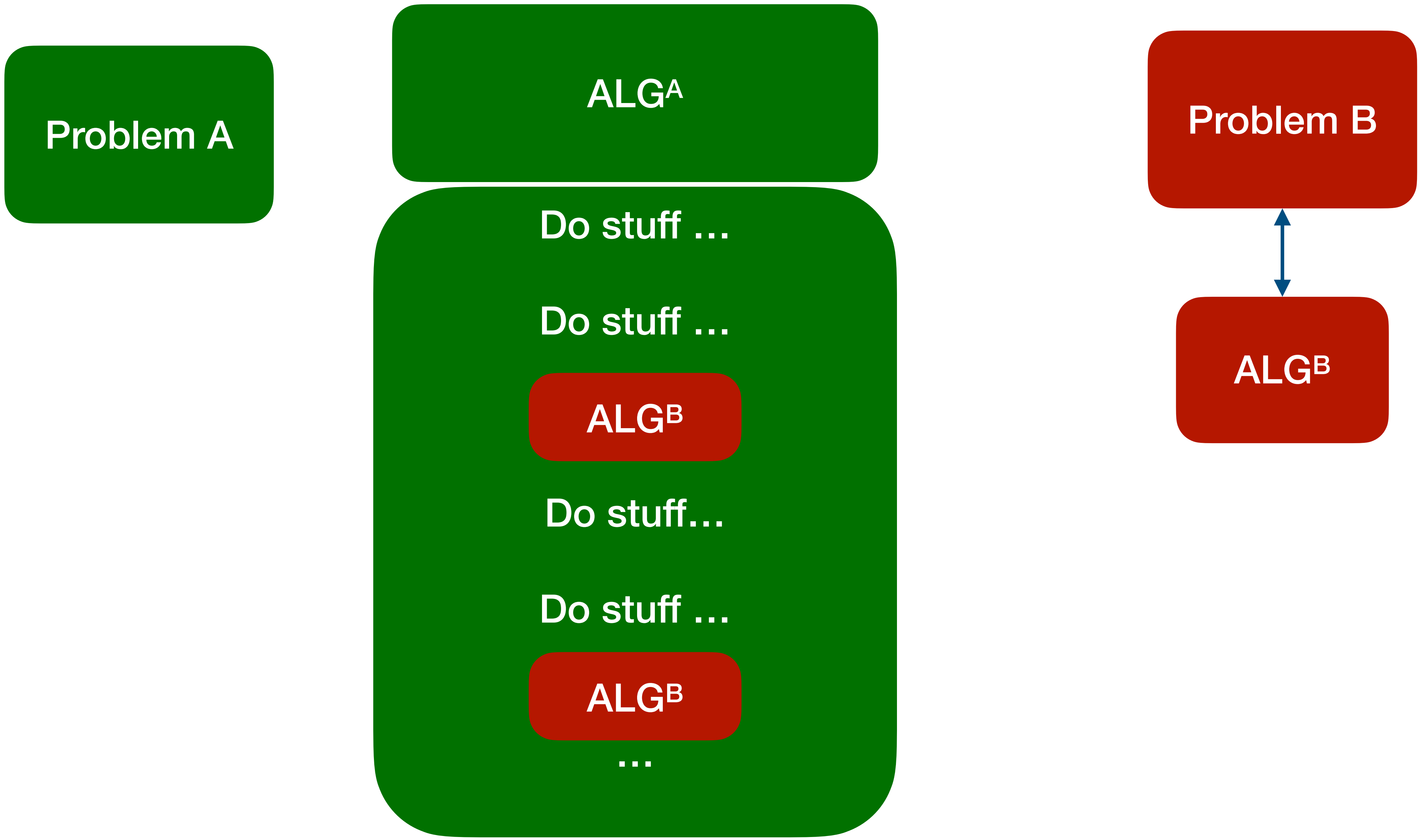
How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
  - If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.
  - B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

Types of reductions

- **Turing reduction:**
  - Notation: $A \leq_T B$
  - A reduction which solves problem A using (polynomially) many calls to an oracle (an algorithm) for solving problem B.
  - (Also known as Cook reduction).

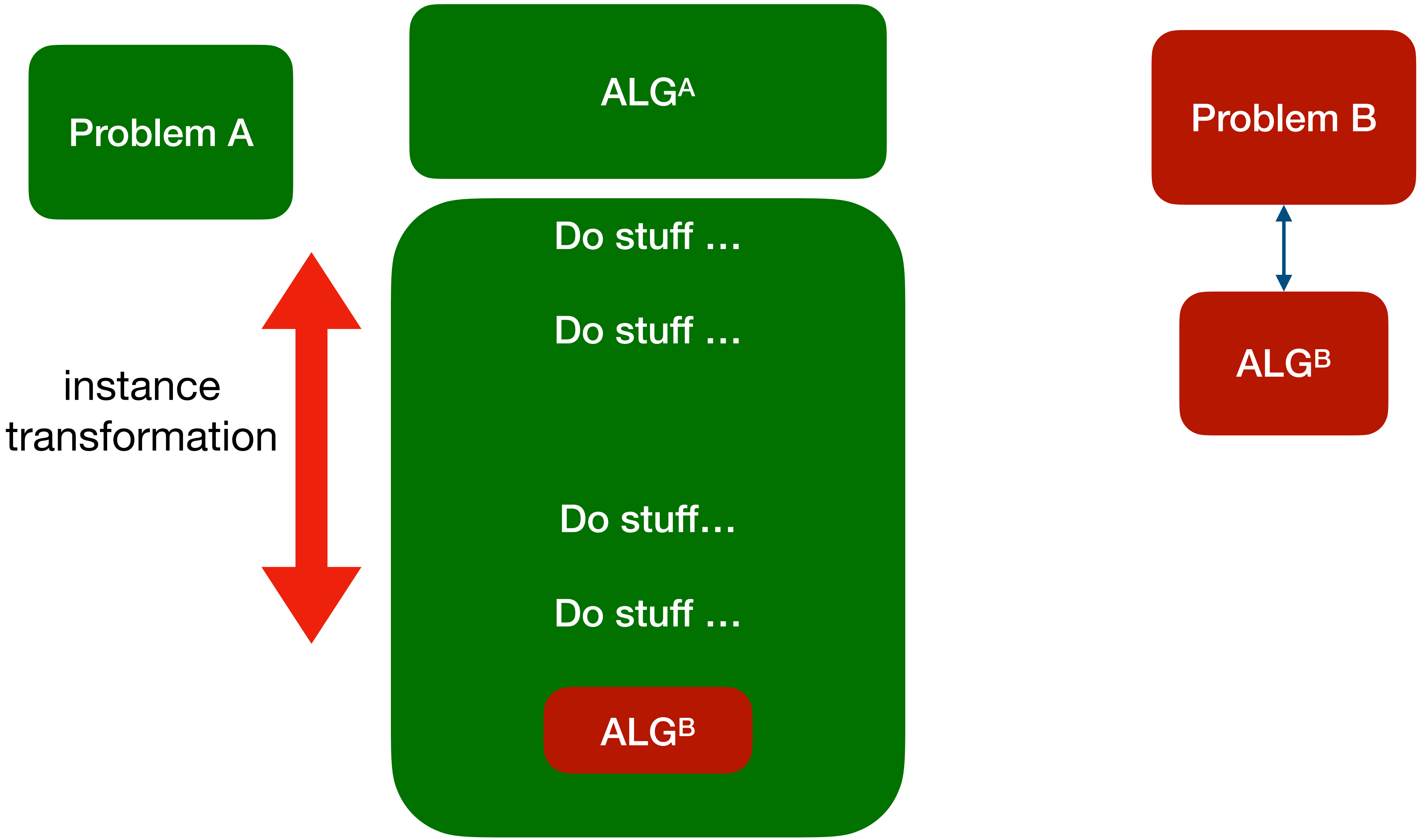
Pictorially



Types of reductions

- **Turing reduction:**
  - Notation: $A \leq_T B$
  - A reduction which solves problem A using (polynomially) many calls to an oracle (an algorithm) for solving problem B.
  - (Also known as Cook reduction).
- **Many-one reduction:**
  - Notation: $A \leq_m B$
  - A reduction which *converts instances* of problem A to *instances* of problem B.
  - (Also known as Karp reduction).

Pictorially



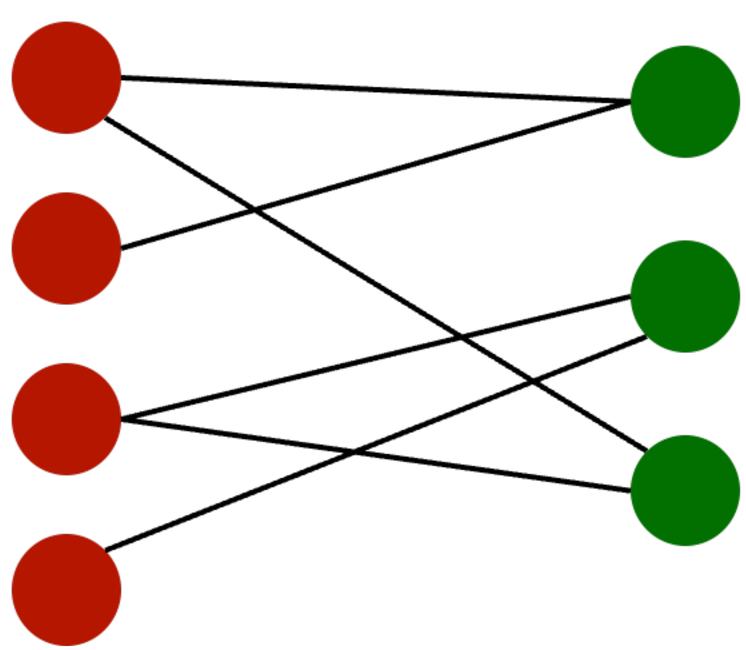
Types of reductions

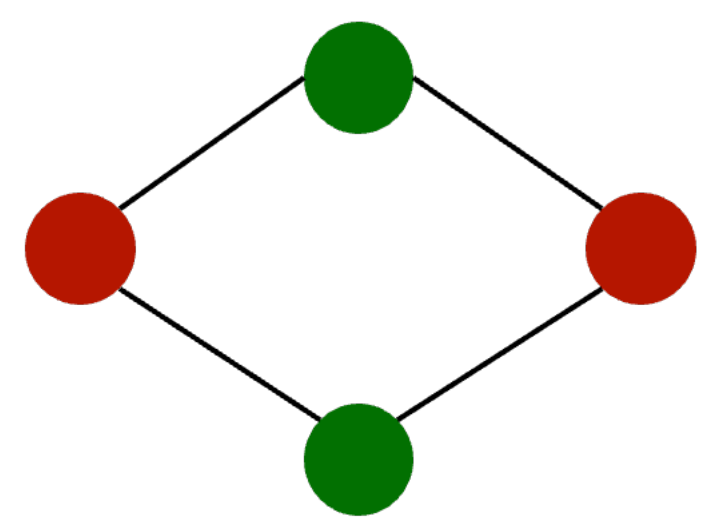
- **Turing reduction:**
  - Argument: Here is an algorithm which runs in polynomial time solving problem A, using polynomially many calls to an oracle for problem B.
- **Many-one reduction:**
  - Argument:
    - If z is a solution to instance I of problem A, then z' is a solution of instance f(I) to problem B.
    - If z is not a solution to instance I of problem A, then z' is not a solution of instance f(I) to problem B.
    - Equivalently: If z' is a solution of instance f(I) to problem B, then z is a solution to instance I of problem A.

Examples of reductions?

Bipartite graphs

- A graph G=(V,E) is bipartite *if any only if* it can be partitioned into sets A and B such that each edge has one endpoint in A and one endpoint in B.
  - Often, we write G=(A U B,E).





Deciding bipartiteness

Deciding bipartiteness

- Given a graph G, *decide* if it is bipartite or not.

Deciding bipartiteness

- Given a graph G, *decide* if it is bipartite or not.
- How did we solve this problem?

Deciding bipartiteness

- Given a graph G, *decide* if it is bipartite or not.
- How did we solve this problem?
- Given a a graph G decide if it is 2-colourable or not.

Deciding bipartiteness

- Given a graph G, *decide* if it is bipartite or not.
- How did we solve this problem?
- Given a a graph G decide if it is 2-colourable or not.
- We reduced the problem to deciding 2-colorability.

Deciding bipartiteness

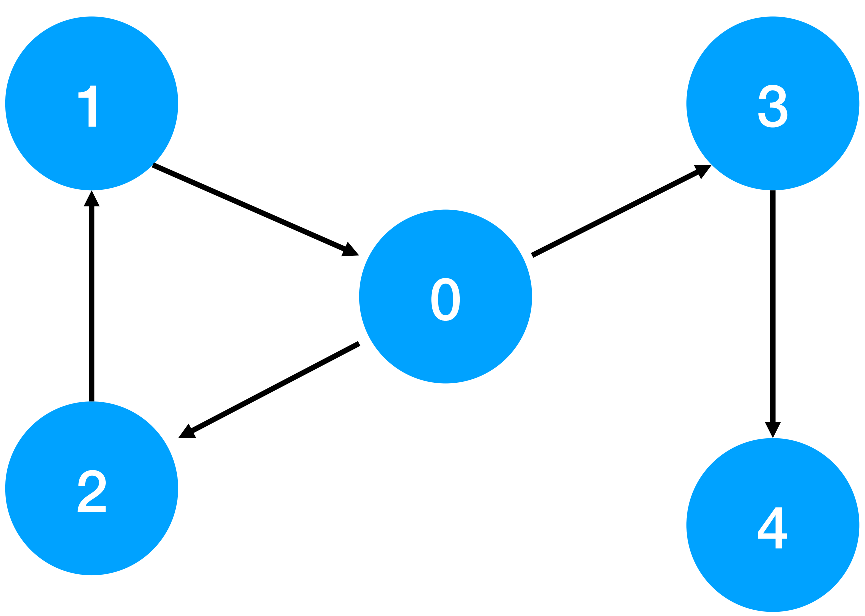
- Given a graph G, *decide* if it is bipartite or not.
- How did we solve this problem?
- Given a a graph G decide if it is 2-colourable or not.
- We reduced the problem to deciding 2-colorability.
  - And how did we solve that?

Deciding bipartiteness

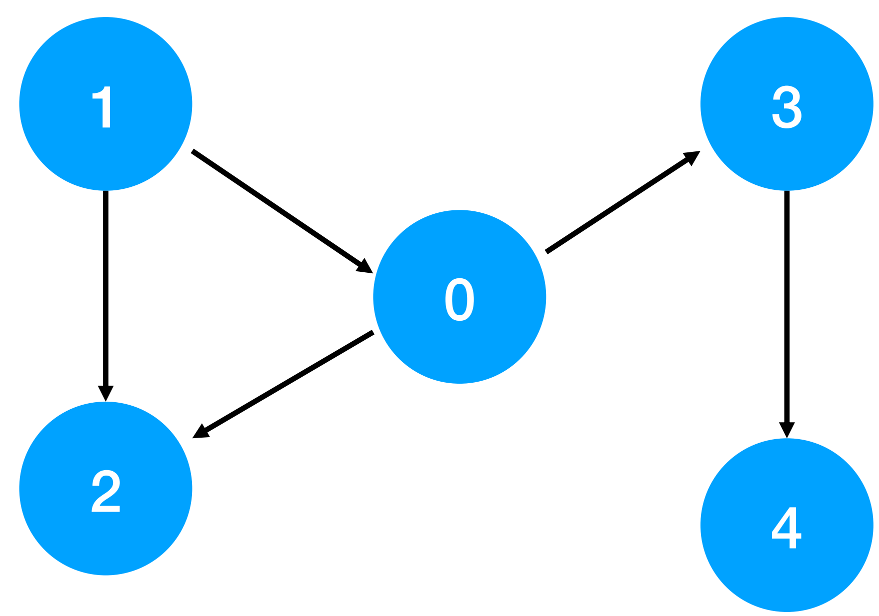
- Given a graph G, *decide* if it is bipartite or not.
- How did we solve this problem?
- Given a a graph G decide if it is 2-colourable or not.
- We reduced the problem to deciding 2-colorability.
    - And how did we solve that?
    - We reduced it to checking whether BFS colours two adjacent nodes with the same colour.

Directed Acyclic Graphs

- A directed acyclic graph (DAG) G is a graph that does not have any cycles.



not a DAG



a DAG

Deciding for DAGs

Deciding for DAGs

- Given a graph G, *decide* if it is a DAG.

Deciding for DAGs

- Given a graph G, *decide* if it is a DAG.
- How can we solve this problem?

Deciding for DAGs

- Given a graph G, *decide* if it is a DAG.
- How can we solve this problem?
- Given a graph G, *decide* if it has a topological ordering.

Deciding for DAGs

- Given a graph G, *decide* if it is a DAG.
- How can we solve this problem?
- Given a graph G, *decide* if it has a topological ordering.
- We reduced the problem to deciding whether the graph has a topological ordering.

Deciding for DAGs

- Given a graph G, *decide* if it is a DAG.
- How can we solve this problem?
- Given a graph G, *decide* if it has a topological ordering.
- We reduced the problem to deciding whether the graph has a topological ordering.
  - And how can we solve that?

Deciding for DAGs

- Given a graph G, *decide* if it is a DAG.
- How can we solve this problem?
- Given a graph G, *decide* if it has a topological ordering.
- We reduced the problem to deciding whether the graph has a topological ordering.
  - And how can we solve that?
  - We can develop an algorithm that finds a topological ordering, or returns that there is none.

Computational classes

Computational classes

Computational classes

- Every problem for which there is a known polynomial time algorithm is in the computational class P.
  - Searching, sorting, interval scheduling, graph traversal, …
  - The class P contains computational problems *that can be solved in polynomial time*.
    - We also say that they can be solved *efficiently*.

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?
  - Weighted interval scheduling?

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?
  - Weighted interval scheduling?
  - Subset sum?

Problems not in P

- Do you remember any problems from the lectures that we did not manage to prove that they lie in P?
  - Weighted interval scheduling?
  - Subset sum?
  - Knapsack?

The landscape of complexity

contains all problems that
can be solved in polynomial time.

The class NP

The class NP

- Stands for "*non deterministic polynomial time*".

The class NP

- Stands for "*non deterministic polynomial time*".
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.

The class NP

- Stands for "*non deterministic polynomial time*".
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.
- More intuitive definition:

The class NP

- Stands for "*non deterministic polynomial time*".
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.
- More intuitive definition:
    - Problems such that, *if a solution is given*, it can be *checked* that it is indeed a solution in polynomial time.

The class NP

- Stands for “*non deterministic polynomial time*”.
- Problems that can be solved in polynomial time by a non-deterministic Turing machine.
- More intuitive definition:
  - Problems such that, *if a solution is given*, it can be *checked* that it is indeed a solution in polynomial time.
  - *Efficiently verifiable*.

The subset sum problem

- We are given a set of n items {*1*, *2*, … , *n*}.
- Each item *i* has a non-negative integer weight w_i.
- We are given an integer bound W.
- Goal: Select a subset S of the items such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is maximised.

Equivalent formulation decision version

- We are given a set of n items {*1*, *2*, … , *n*}.
- Each item *i* has a non-negative integer weight w_i.
- We are given an integer bound W.
- Goal: Decide if there exists a subset S of the items such that

$$\sum_{i \in S} w_i = W$$

Subset Sum is in NP

- If we are given a candidate solution S, we can easily check whether the following holds or not:

$$\sum_{i \in S} w_i = W$$

Problem classification

Problem classification

- Problems in P:
  - Searching, sorting, graph traversal, maximum flow, minimum cut, Weighted Interval Scheduling, …

Problem classification

- Problems in P:
  - Searching, sorting, graph traversal, maximum flow, minimum cut, Weighted Interval Scheduling, …
- Problems in NP:
  - Subset Sum, Knapsack

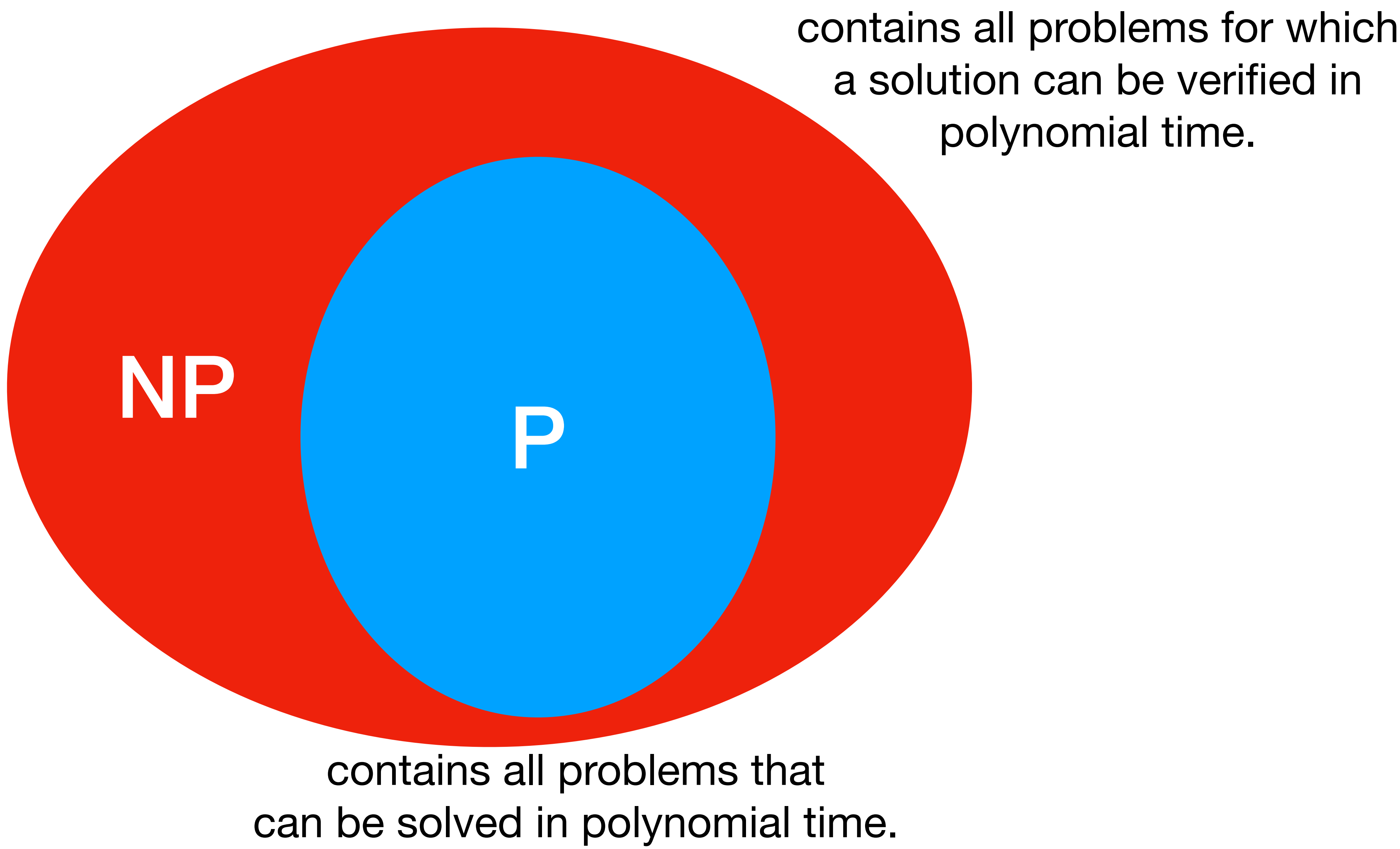
Problem classification

- Problems in P:
    - Searching, sorting, minimum spanning tree, graph traversal, Weighted Interval Scheduling, …
- Problems in NP:
    - Subset Sum, Knapsack, Weighted Interval Scheduling, searching, sorting, graph traversal, Weighted Interval Scheduling, …

The landscape of complexity

contains all problems that
can be solved in polynomial time.

The landscape of complexity



How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
  - If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.
  - B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

How to work with reductions

- Positive: Assume that I want to solve problem A and I know how to solve problem B in polynomial time.
  - I can try to come up with a polynomial time reduction $A \leq^p B$, which will give me a polynomial time algorithm for solving A.
- Contrapositive: Assume that there is a problem A for which it is unlikely that there is a polynomial time algorithm that solves it.
  - If I come up with a polynomial time reduction $A \leq^p B$, it is also unlikely that there is a polynomial time algorithm that solves B.
  - B is "*at least as hard to solve as*" A, because if I could solve B, I could also solve A.

NP-hardness

- A problem B is NP-hard if for every problem A in NP, it holds that A $\leq^p$ B.
  - If every problem in NP is “polynomial time reducible to B”.
  - This captures the fact that B is *at least as hard as the hardest problems* in NP.

NP-hardness

- A problem B is NP-hard if for every problem A in NP, it holds that $A \leq^p B$.

- To prove NP-hardness, it seems that we have to construct a reduction from every problem A in NP.

  - This is not very useful!

NP-completeness

- A problem B is NP-complete if

NP-completeness

- A problem B is NP-complete if
  - *It is in NP*.

NP-completeness

- A problem B is NP-complete if
  - *It is in NP*.
    - i.e., it has a polynomial-time verifiable solution.

NP-completeness

- A problem B is NP-complete if
    - *It is in NP*.
        - i.e., it has a polynomial-time verifiable solution.
    - *It is NP-hard*.

NP-completeness

- A problem B is NP-complete if
  - *It is in NP*.
    - i.e., it has a polynomial-time verifiable solution.
  - *It is NP-hard*.
    - i.e., every problem in NP can be efficiently reduced to it.

NP-completeness

NP-completeness

- Assume problem P is NP-complete.

NP-completeness

- Assume problem P is NP-complete.
  - Then every problem in NP is efficiently reducible to P. (why?)

NP-completeness

- Assume problem P is NP-complete.
  - Then every problem in NP is efficiently reducible to P. (why?)
- To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.

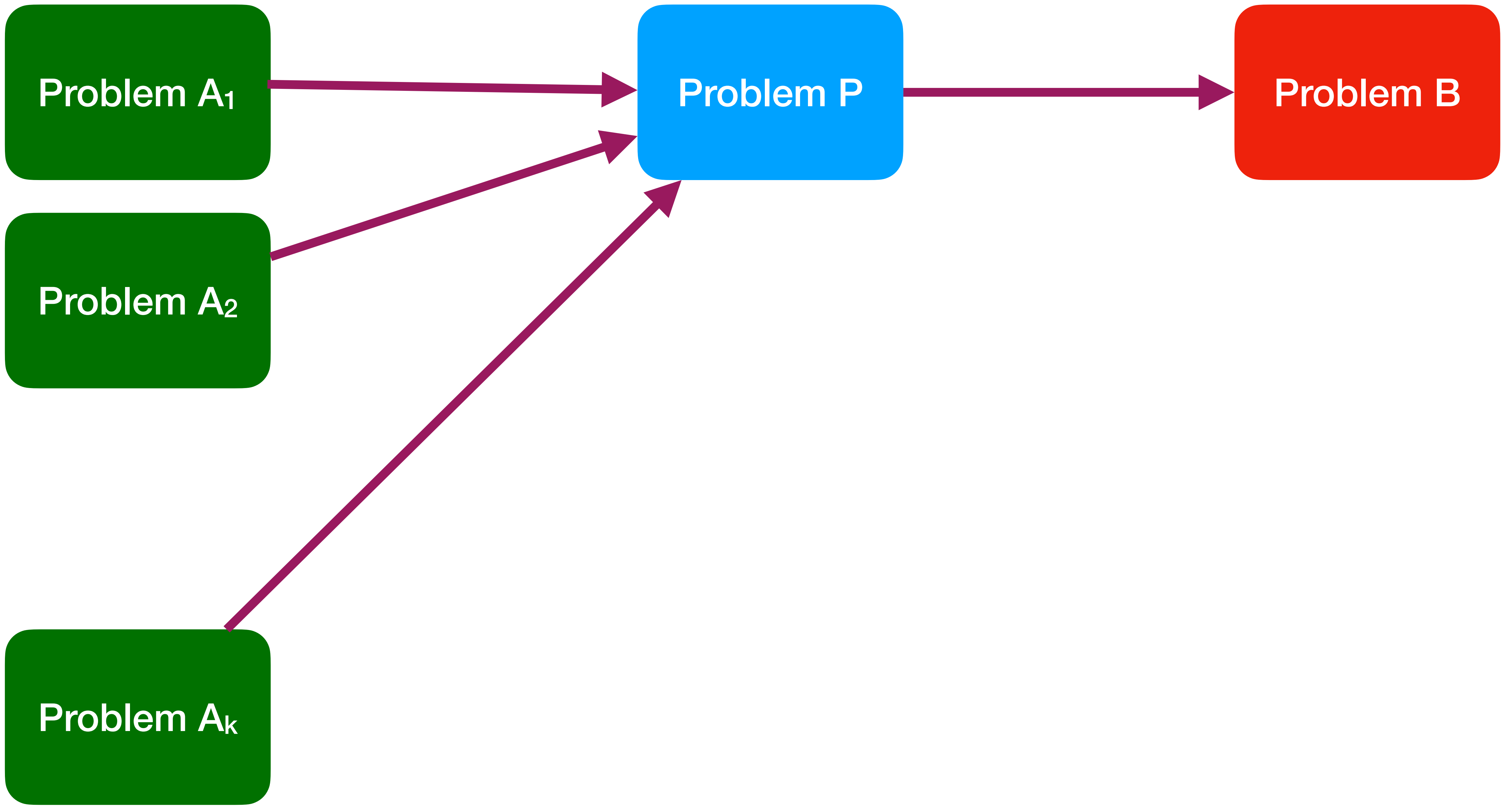
NP-completeness

- Assume problem P is NP-complete.
    - Then every problem in NP is efficiently reducible to P. (why?)
- To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.
    - Actually, it suffices to construct a reduction from P to B.

NP-completeness

- Assume problem P is NP-complete.
  - Then every problem in NP is efficiently reducible to P. (why?)
- To prove NP-hardness of problem B, it seems that we have to construct a reduction from every problem A in NP.
  - Actually, it suffices to construct a reduction from P to B.
  - A reduction from any other problem A to B goes “via” P.

NP-hardness via P



NP-completeness

NP-completeness

- Assume problem P is NP-complete.

NP-completeness

- Assume problem P is NP-complete.
- This all works if we have an NP-complete problem to start with.

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- ("An AND of ORs").

- Each clause has three literals.

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- ("An AND of ORs").

- Each clause has three literals.

- Truth assignment: A value in {0,1} for each variable x_i.

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- ("An AND of ORs").

- Each clause has three literals.

- Truth assignment: A value in {0,1} for each variable x_i.

- Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).

3 SAT

- A CNF formula with m clauses and k literals.

  $\phi = (x_1 \vee x_5 \vee x_3) \wedge (x_2 \vee x_6 \vee \neg x_5) \wedge \ldots \wedge (x_3 \vee x_8 \vee x_{12})$

- ("An AND of ORs").
- Each clause has three literals.
- Truth assignment: A value in {0,1} for each variable x_i.
- Satisfying assignment: A truth assignment which makes the formula evaluate to 1 (= true).
- Computational problem 3SAT : Decide if the input formula ϕ has a satisfying assignment.

3 SAT is NP-complete

3 SAT is NP-complete

- 3 SAT is in NP (why?)

3 SAT is NP-complete

- 3 SAT is in NP (why?)
- 3 SAT is NP-hard.

3 SAT is NP-complete

- 3 SAT is in NP (why?)
- 3 SAT is NP-hard.
- Remarks:
  - The first problem shown to be NP-complete was the SAT problem (more general than 3 SAT), and this reduces to 3SAT.
  - Several textbooks start from Circuit SAT, a version of the SAT problem defined on circuits with boolean gates AND, OR or NOT.

Proving NP-completeness

Proving NP-completeness

- Suppose that you are given a problem A and you want to prove that it is NP-complete.

Proving NP-completeness

- Suppose that you are given a problem A and you want to prove that it is NP-complete.
- First, prove that A is in NP.
    - Usually by observing that a solution is efficiently checkable.

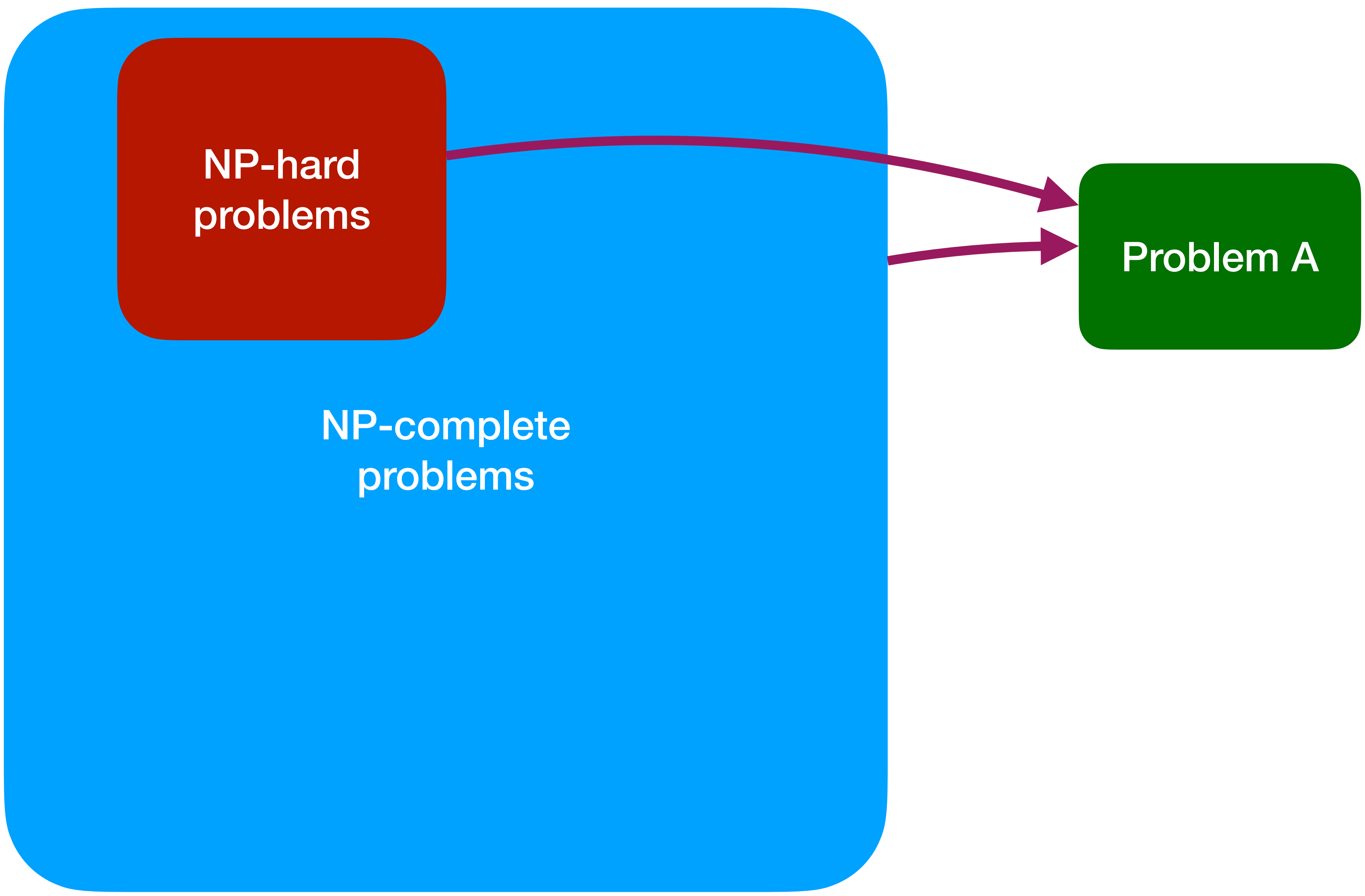
Proving NP-completeness

- Suppose that you are given a problem A and you want to prove that it is NP-complete.
- First, prove that A is in NP.
  - Usually by observing that a solution is efficiently checkable.
- Then prove that A is NP-hard.
  - Construct a polynomial time reduction from some NP-complete problem P.

In fact ...

- Suppose that you are given a problem A and you want to prove that it is NP-complete.
- First, prove that A is in NP.
  - Usually by observing that a solution is efficiently checkable.
- Then prove that A is NP-hard.
  - Construct a polynomial time reduction from some NP-hard problem P.

Pictorially



Enough with the definitions. Let’s see how it works.

Enough with the definitions. Let's see how it works.

Next time!