Chapter 7 & 8

Computability
and complexity theory

Computability Theory

Computability theory answers:
What problems can computers solve?

- Why we want to know if a problem is unsolvable?
  - to realize that the problem must be simplified or altered before you can find an algorithmic solution.

Complexity theory

Measuring Complexity

Suppose that we have a decider D for some language L.

How might we measure the complexity of D?

- Number of states.
- Size of tape alphabet.
- Size of input alphabet.
- Amount of tape required.
- Number of steps required. Or ?

Time complexity

- A **step** of a Turing machine is one event where the TM takes a **transition**.
- Running a TM on different inputs might take a **different number of steps**.
- The number of steps a TM takes on some input is sensitive to
  - The structure of that input. T
  - he length of the input.

How can we come up with a **consistent measure** of a machine's runtime?

Time Complexity

The time complexity of a TM M is a **function** (typically denoted **f(n)**) that measures the **worst-case** number of steps M takes on any input of length n.

By convention, **n** denotes the **length** of the **input**.

Example: $L = \{ w \in \Sigma^* \mid w$ has the same number of 0s and 1s $\}$
Solution: M = "On input w: –

Scan across the tape until a 0 or 1 is found.
If none are found, accept.
If one is found, continue scanning until a matching 1 or 0 is found.
If none is found, reject.
Otherwise, cross off that symbol and repeat."

What is the time **complexity** of M?

Example

Scan across the tape until a 0 or 1 is found. At most n steps
If none are found, accept. At most 1 step
If one is found, continue scanning until ….. At most n step
a matching 1 or 0 is found.
If none is found, reject. …… At most 1 step
Otherwise, cross off that symbol and
go back to start of tape…….. At most n step
repeat.” ……n/2
Accept…….atmost 1 step

Hence $f(n) = n/2(n+n) + 1 = n^2 + 1$

An Easier Approach

In complexity theory, we rarely need an **exact** value for a TM's time complexity.
Usually, we are curious with the **long-term growth rate** of the time complexity.

For example,

if the time complexity is $3n + 5$, then doubling the length of the string **roughly doubles** the worst-case runtime.

If the time complexity is $2^n – n^2$, since 2^n grows much more quickly than n^2 , for large values of n, increasing the size of the input by 1 doubles the worst-case running time

Big-O Notation

Ignore everything **except** the **dominant growth term**, including constant factors.

Examples:

$4n + 4 = \mathbf{O(n)}$

$137n + 271 = \mathbf{O(n)}$

$n^2 + 3n + 4 = \mathbf{O(n2)}$

$2^n + n^3 = \mathbf{O(2^n)}$

$137 = \mathbf{O(1)}$

$n^2 \log n + \log 5\ n = \mathbf{O(n^2 \log n)}$

Big-O Notation, Formally

Let $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$.
Then **f(n) = O(g(n)) iff** there exist constants **c** $\in \mathbb{R}$ and $\mathbf{n_0} \in \mathbb{N}$ such that

For any $n \geq n_0$, **f(n) ≤ cg(n)**

Intuitively, as n gets "large" (greater than n0), f(n) is **bounded from above** by some multiple (determined by **c**) of **g(n)**

Properties of Big-O Notation

Theorem: If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) + f_2(n) = O(g_1(n) + g_2(n))$.

Intuitively: If you run two programs **one after another**, the big-O of the result is the big-O of the sum of the two runtimes.

Theorem: If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n)f_2(n) = O(g_1(n)g_2(n))$.

Intuitively: If you run **one program** some **number of times**, the big-O of the result is the big-O of the program times the big-O of the number of iterations.

This makes it substantially easier to analyze time complexity, though we do lose some precision.

Classes of problems
P Vs. NP

Efficiency

- What does it mean for an algorithm/ TM to be efficient?
- We define running time as the function of input length n
- When an algorithm has a running time $O(n^2)$ it means that for a long enough inputs the algorithm takes no more than quadratic time.
- An algorithm is efficient if its running time is polynomial. More precisely a running time is polynomial if it is $O(n^c)$ for some constant c.
  - Polynomial: n^2, n^{100},$n\log^n$
  - Non polynomial: 2^n,$n!$,$2^{\log n}$

NP-Hard and NP-Complete problems

- **P** = the set of problems that are solvable in polynomial time.
- If the problem has size n, the problem should be solved in $n^{O(1)}$.
- **NP** = the set of decision problems solvable in nondeterministic polynomial time.

Decision Problems

- The output of these problems is a YES or NO answer.
  - Example: is N prime? Do sequences x and y have common subsequences of length>k?
  - For many problems there is an associated decision problem.
- Nondeterministic refers to the fact that a solution can be guessed out of polynomially many options in $O(1)$ time.
- If any guess is a YES instance, then the nondeterministic algorithm will make that guess.
- In this model of nondeterminism, we can assume that all guessing is done first.
- This is equivalent to finding a polynomial-time verifier of polynomial-size certificates for YES answers.

NP

Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y?)

NP

Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y?)

There are many decision problems where we can efficiently verify solutions, but can't efficiently find them.

Example: Hamiltonian Cycle: In the graph G is there a simple cycle that goes through every vertex?

NP

Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y?)

There are many decision problems where we can efficiently verify solutions, but can't efficiently find them.

Example: Hamiltonian Cycle: In the graph G is there a simple cycle that goes through every vertex?

There is no known poly-time algorithm for this but it is easy to verify if a given cycle is a Hamiltonian cycle.

NP

Given a solution to an instance of a decision problem, we want to verify if it actually is a solution (is this sequence actually a subsequence of x and y?)

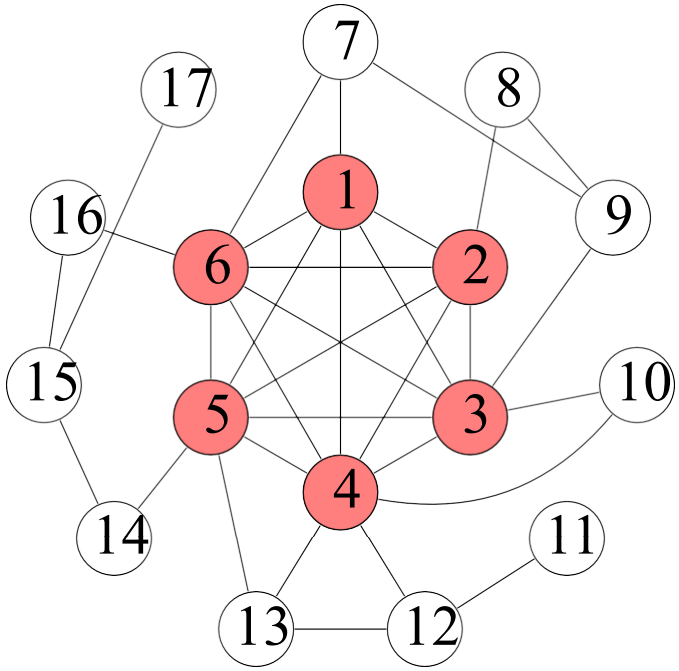
There are many decision problems where we can efficiently verify solutions, but can't efficiently find them.

Example: Hamiltonian Cycle: In the graph G is there a simple cycle that goes through every vertex?

There is no known poly-time algorithm for this but it is easy to verify if a given cycle is a Hamiltonian cycle.

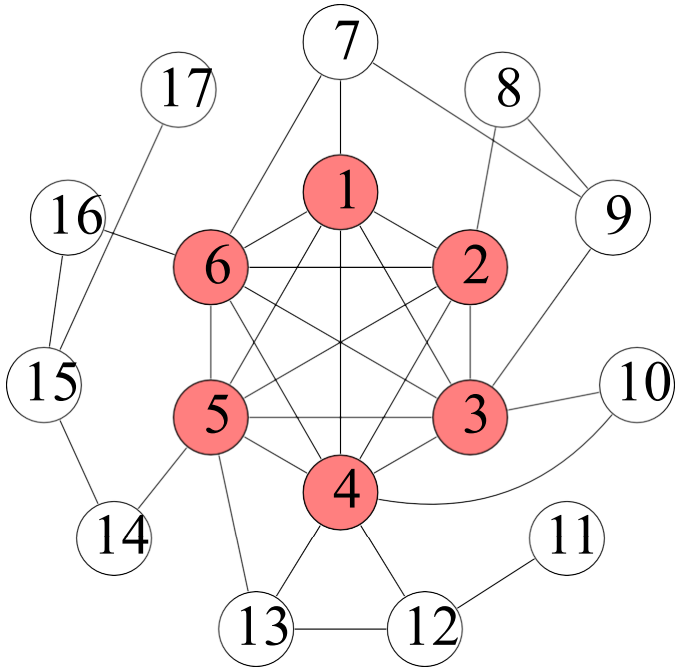
NP is the class of decision problems where if the answer is YES then there is a short "solution" which can be easily verified.

Maximum Clique



Given a graph G and a number K, are there K vertices in G that are all pairwise adjacent (this is called a clique)?

Maximum Clique

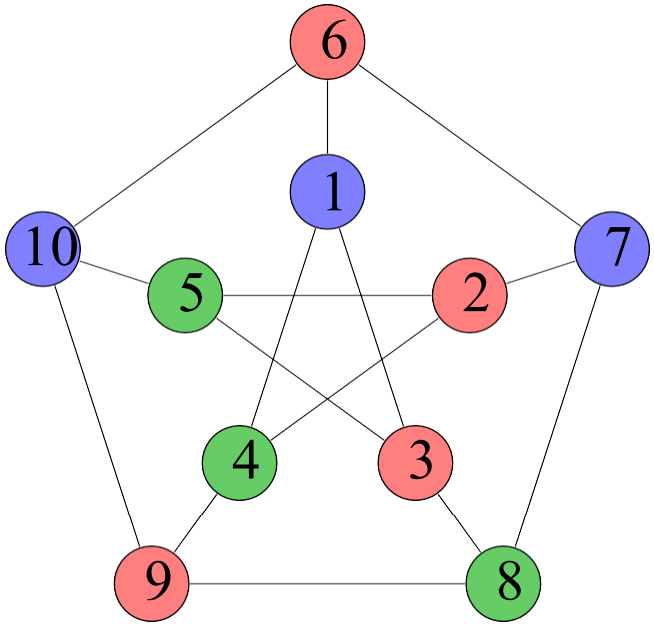


Given a graph G and a number K, are there K vertices in G that are all pairwise(very 2 of w/c are adjacent) adjacent (this is called a clique)?

Easy to verify since given the vertices we only need check that they are all adjacent.

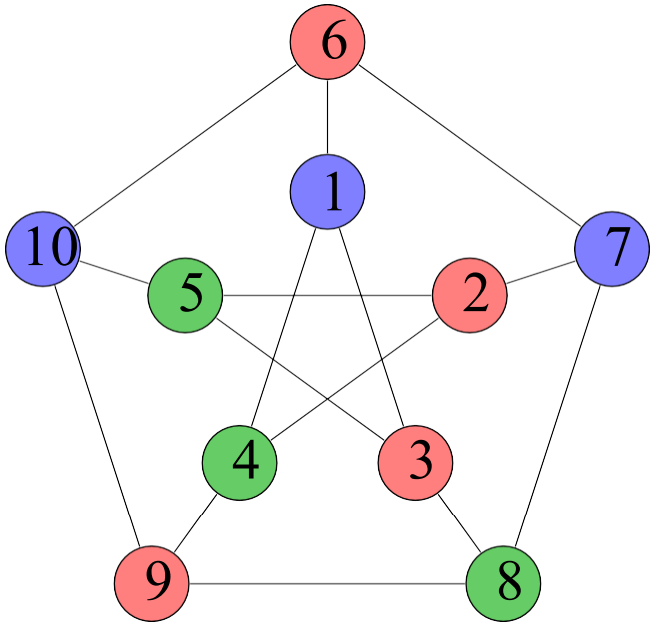
Useful for finding groups of mutual friends in social networks.

3-Coloring



Given a graph G, can we assign 3 colors to its vertices so that any pair of adjacent vertices have different colors?

3-Coloring

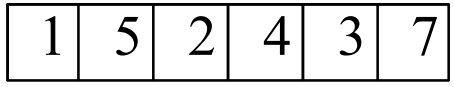


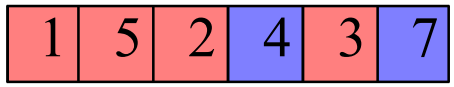
Given a graph G, can we assign 3 colors to its vertices so that any pair of adjacent vertices have different colors?

Easy to verify a coloring by examining all edges so it is in NP.

Useful for allocating transmission frequencies to radio stations to avoid interference.

Partition Problem

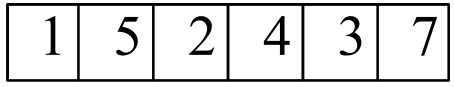


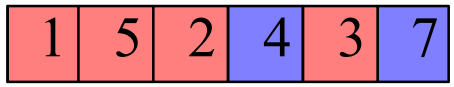


?

Given n numbers, can they be partitioned into 2 sets such that the sums of the numbers in the sets are equal

Partition Problem





?

Given n numbers, can they be partitioned into 2 sets such that the sums of the numbers in the sets are equal

Easy to verify given the two sets, so it is in NP.

P v NP

Recall that for a decision problem in NP, if the answer is yes for a given input then the "solution" can be verified efficiently

P v NP

Recall that for a decision problem in NP, if the answer is yes for a given input then the "solution" can be verified efficiently

But can we compute the solution efficiently?

P v NP

Recall that for a decision problem in NP, if the answer is yes for a given input then the "solution" can be verified efficiently

But can we compute the solution efficiently?

We don't know! This is the $P =^{?} NP$ question.

Hard Problems

One approach to settling P = ? NP: Identify the "hardest" problems in NP and focus on solving them in poly-time

Hard Problems

One approach to settling $P \stackrel{?}{=} NP$: Identify the "hardest" problems in NP and focus on solving them in poly-time

But how do we know which problems are hard?

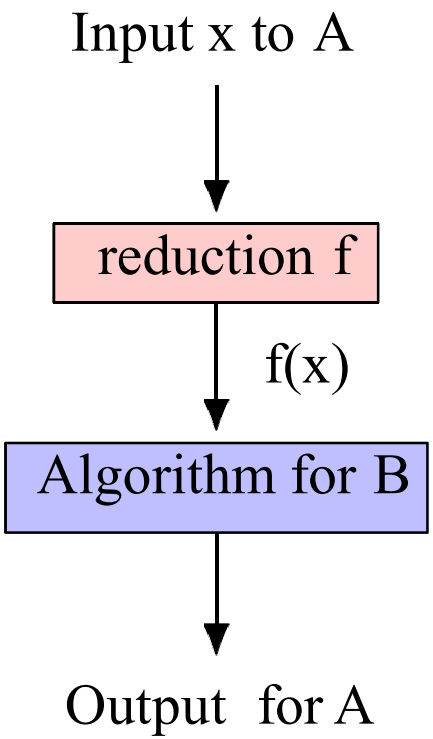
Hard Problems

One approach to settling P = ? NP: Identify the "hardest" problems in NP and focus on solving them in poly-time

But how do we know which problems are hard?

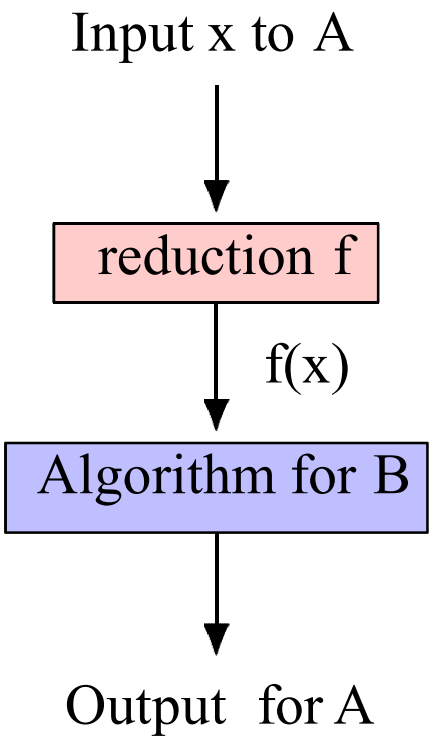
Idea: Reductions

Reductions



We can reduce problem A to problem B if we can use a solution to B to solve A.

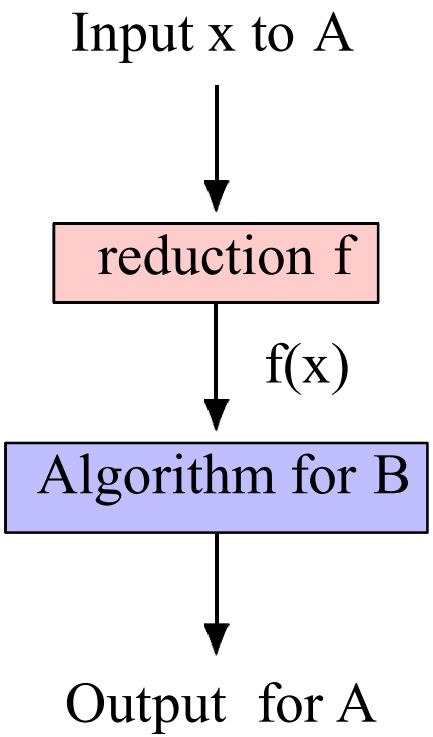
Reductions



We can reduce problem A to problem B if we can use a solution to B to solve A.

Example: Problem A is finding the median of n elements and Problem B is sorting n elements.

Reductions



We can reduce problem A to problem B if we can use a solution to B to solve A.

Example: Problem A is finding the median of n elements and Problem B is sorting n elements.

A reduces to B since we can take the input to A, sort it using the solution to B, and then recover the solution to A by looking at the middle element in sorted order.

Reductions cont.

The median and sorting example illustrates reductions, but it is a bad example since medians can be computed directly faster than sorting.

Reductions cont.

The median and sorting example illustrates reductions, but it is a bad example since medians can be computed directly faster than sorting.

However if we go the other way we can use the median finding algorithm to make an efficient sorting algorithm.

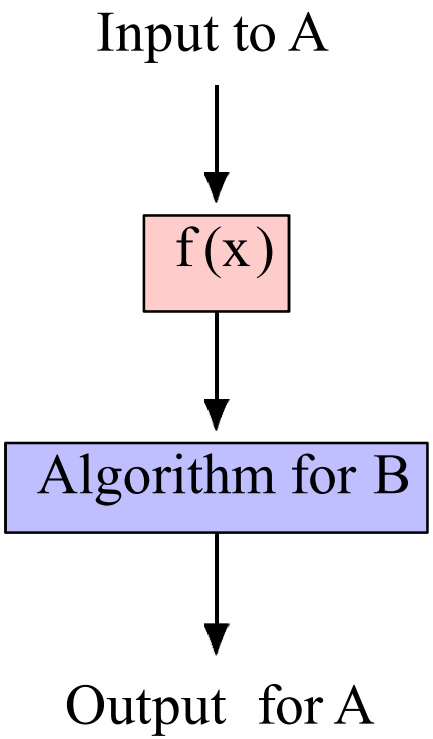
Reductions cont.

The median and sorting example illustrates reductions, but it is a bad example since medians can be computed directly faster than sorting.

However if we go the other way we can use the median finding algorithm to make an efficient sorting algorithm.

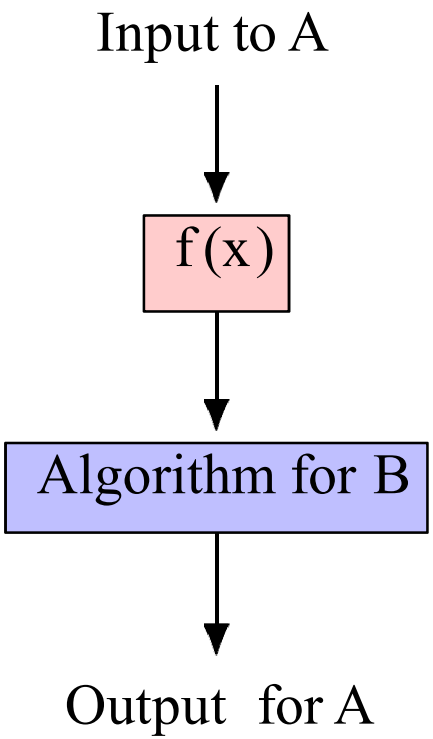
Sort n elements, compute the median using the black box for A
Use the median as a pivot like in quicksort and recurse.
We will always have a perfect partition so the algorithm will be efficient.

Reduction Definition



We need a more rigorous definition of a reduction to identify the hardest problems in NP.

Reduction Definition



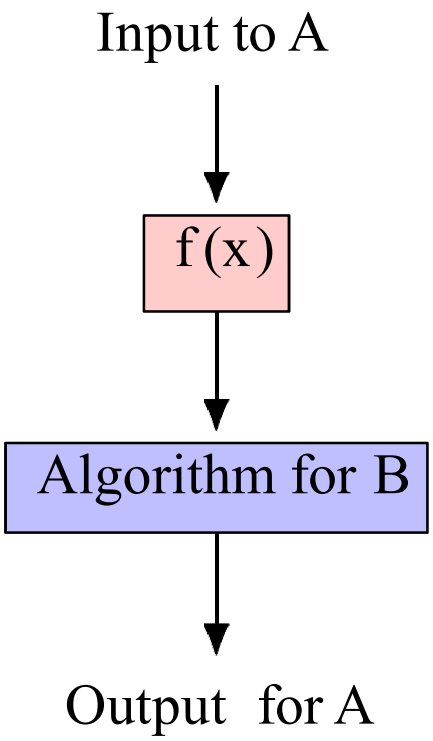
We need a more rigorous definition of a reduction to identify the hardest problems in NP.

We say Decision Problem A reduces to Decision problem B if there is a function f mapping inputs of A to inputs of B such that:

If x is a YES input for A, then f (x) is a YES input for B

IF x is a NO input for A, then f (x) is a NO input for B

Reduction Definition



We need a more rigorous definition of a reduction to identify the hardest problems in NP.

We say Decision Problem A reduces to Decision problem B if there is a function f mapping inputs of A to inputs of B such that:

If x is a YES input for A, then f (x) is a YES input for B
IF x is a NO input for A, then f (x) is a NO input for B

The reduction is f itself.

Poly-Time Reductions

If f (x) can be computed in polynomial time, we say it is a polynomial-time reduction.

Note: If f is a polynomial-time reduction, then |f (x)| is polynomial in the length of x.

Poly-Time Reductions

If $f(x)$ can be computed in polynomial time, we say it is a polynomial-time reduction.

Note: If f is a polynomial-time reduction, then $|f(x)|$ is polynomial in the length of x.

Median finding and sorting are not decision problems, but otherwise the median finding to sorting reduction fits this definition.

What is $f(x)$? Is it in computable poly-time?

However, the other direction does not fit since we repeatedly use the median finding algorithm

Poly-Time Reduction Implications

What does it mean if there is a poly-time reduction from A to B?

Poly-Time Reduction Implications

What does it mean if there is a poly-time reduction from A to B?

If B is solvable in poly time, then so is A. Just map the input to A to an input to B, and use the method for solving B

(recall that if f (x) and g (x) are polynomials then f (g (x)) is also a polynomial).

Poly-Time Reduction Implications

What does it mean if there is a poly-time reduction from A to B?

If B is solvable in poly time, then so is A. Just map the input to A to an input to B, and use the method for solving B

(recall that if f (x) and g (x) are polynomials then f (g (x)) is also a polynomial).

If A is not solvable in polynomial time, then neither is B. This statement is actually equivalent to the original one.

Example

Suppose we know that if one could travel faster than the speed of light, then one could travel back in time.

Example

Suppose we know that if one could travel faster than the speed of light, then one could travel back in time.

Using our language, the problem of traveling back to the past reduces to the problem of traveling faster than the speed of light

Example

Suppose we know that if one could travel faster than the speed of light, then one could travel back in time.

Using our language, the problem of traveling back to the past reduces to the problem of traveling faster than the speed of light

If we manage to build a faster-than-light vehicle, then we can go back to the past

But if we prove that is impossible to travel back in time, then we immediately know it is impossible to build a faster-than-light vehicle.

NP−Completeness Definition

Suppose we have a decision problem A in NP such that for every problem in NP there is a polynomial time reduction to A.

NP−Completeness Definition

Suppose we have a decision problem A in NP such that for every problem in NP there is a polynomial time reduction to A.

If we knew A is solvable in polynomial time, then every problem in NP can be solved in polynomial time.

Equivalently, $A \in P \implies P = NP$.

NP−Completeness Definition

Suppose we have a decision problem A in NP such that for every problem in NP there is a polynomial time reduction to A.

If we knew A is solvable in polynomial time, then every problem in NP can be solved in polynomial time.

Equivalently, $A \in P \implies P = NP$.

In a sense, A is a "hardest" problem in NP.

We say a problem A is NP-complete if

$A \in NP$

Every problem in NP reduces to A

Satisfiability

How could we ever show that every problem in NP reduces to a problem A?

Satisfiability

How could we every show that every problem in NP reduces to a problem A?

Cook and Levin did exactly this for the Satisfiability problem.

Satisfiability: Given a boolean formula, is there a way to set the variables such that the formula evaluates to true?

Satisfiability

How could we every show that every problem in NP reduces to a problem A?

Cook and Levin did exactly this for the Satisfiability problem.

Satisfiability: Given a boolean formula, is there a way to set the variables such that the formula evaluates to true?

For example, $((\neg a \vee \neg b \vee c) \cap (a \vee c) \cap (\neg c \vee b)) \vee (\neg a \cap b \cap c)$

is satisfied by $a \longmapsto 1, b \longmapsto 0, c \longmapsto 0$.

Satisfiability

How could we every show that every problem in NP reduces to a problem A?

Cook and Levin did exactly this for the Satisfiability problem.

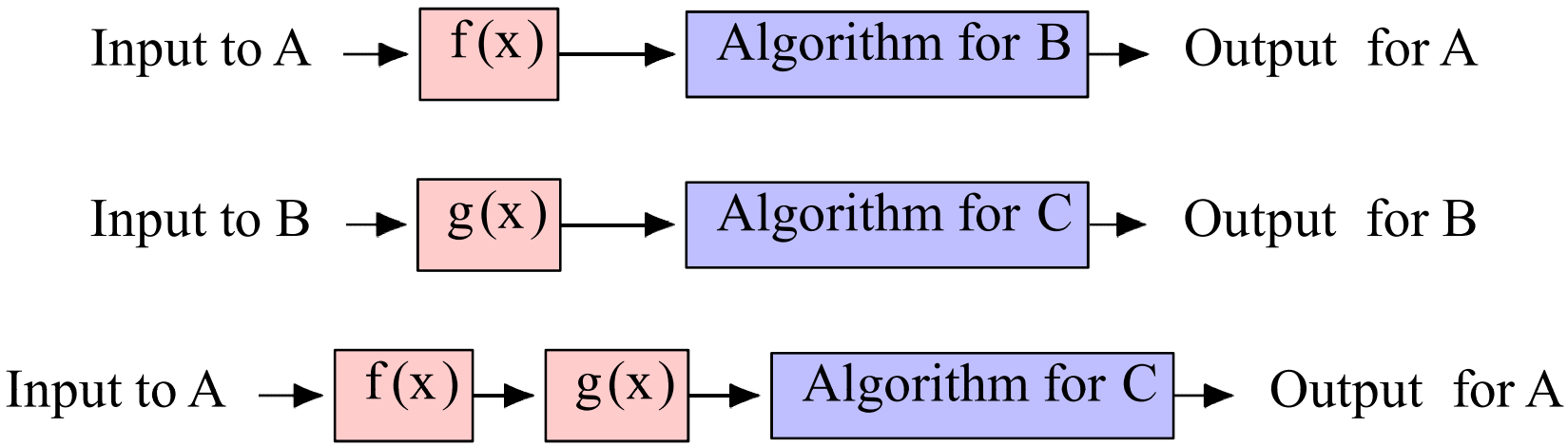
Satisfiability: Given a boolean formula, is there a way to set the variables such that the formula evaluates to true?

For example, $((\neg a \vee \neg b \vee c) \wedge (a \vee c) \wedge (\neg c \vee b)) \vee (\neg a \wedge b \wedge c)$

is satisfied by $a \mapsto 1, b \mapsto 0, c \mapsto 0$.

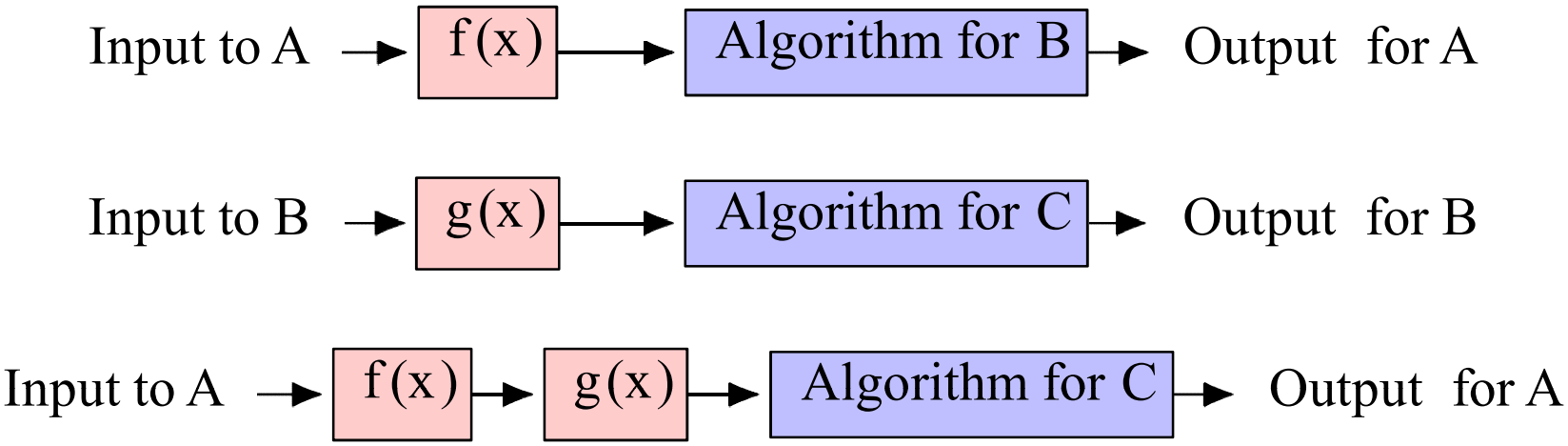
Cook-Levin uses the fact that every problem in NP has a poly-time verifier to construct a formula that is satisfiable if and only if there is a "solution" that the verifier will accept.

Properties of Reductions



Polynomial-time reductions are transitive: If A reduces to B and B reduces to C, then A reduces to C

Properties of Reductions



Polynomial-time reductions are transitive: If A reduces to B and B reduces to C, then A reduces to C

After Cook-Levin, to show a problem X is NP-complete we need only show that $X \in NP$ and that Satisfiability or another NP-complete problem reduces to X .