Lecture 19
P and NP

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Lecture Outline

- Different types of problems
    - Tractable vs intractable
    - Solvable vs unsolvable
    - Decision problems
    - P and NP
- Polynomial-time reduction
- NP-completeness, NP-hardness

Different time complexities

Different algorithms can have different time complexities.

Some common complexity classes	Notation (input size $= n$)
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Log-linear	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(e^n)$
Factorial	$O(n!)$
Doubly-exponential	$O(e^{e^n})$

Polynomial time

We say an algorithm runs in **polynomial time** if its time complexity is **at most** $O(n^c)$ for some constant c.

Exponential time

We say an algorithm runs in **exponential time** if its time complexity is **at most** $O(e^{n^c})$ for some constant c.

Some common complexity classes	Notation (input size $= n$)
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Log-linear	$O(n \log n)$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(e^n)$
Factorial	$O(n!)$
Doubly-exponential	$O(e^{e^n})$

Exponential time

Question: Can you see why e^{e^n} is not $O(e^{n^c})$ for any constant c?

Challenge: Explain why $n!$ is $O(e^{n^2})$, but $n!$ is **not** $O(e^n)$?

Tractable problems

Given a problem A, there could be many possible solutions, with possibly different time complexities.

- We say A **can be solved in exponential time** if A has at least one algorithmic solution that runs in **exponential** time.
- We say A **can be solved in polynomial time** if A has at least one algorithmic solution that runs in **polynomial** time.

Definition: A **tractable problem** is a problem that can be solved in polynomial time.

- In this course so far, we have seen many tractable problems:
  - Sorting problem
  - Single source shortest path problem
  - Etc.

Intractable problems

An intractable problem is a problem that **cannot** be solved in **polynomial time**.

- This is a very strong condition!
- A problem, without any known solution that is able to run in polynomial time, may not necessarily be intractable.
  - Perhaps someone smart enough would one day be able to come up with a solution that runs in polynomial time.
- An intractable problem is a problem that is **guaranteed to have no possible solutions that runs in polynomial time**.

There are many problems that we suspect are intractable, but we still cannot rule out the existence of solutions (still not yet discovered) that run in polynomial time.

Why we care about polynomial time

A general philosophy in Computer Science:

- Computational problems that cannot be solved in polynomial time are "hard problems".
  - "hard" in the sense that it is **hard to complete the computation in a reasonable amount of time**.

Intuition for tractable versus intractable problems

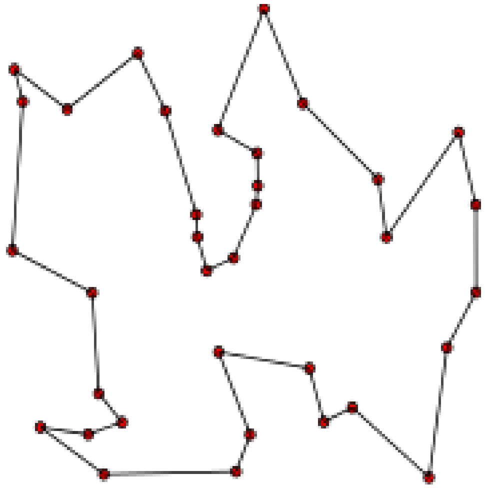
- Problem A takes $100n^3$ steps to solve, for an input of size n.
- Problem B takes 0.01×2^n steps to solve, for an input of size n.
- If each step takes one microsecond (1 millionth of a second), then for an input of size 100, Problem A takes ______________ to solve, while Problem B takes roughly __________________ to solve.

The traveling salesman problem

Problem: The traveling salesman problem

Given a list of cities and the distances between each pair of cities, what is a shortest possible route that visits each city exactly once and returns to the origin city?

- If there are n cities, then the "best" known solution uses dynamic programming and has time complexity $O(n^2 2^n)$.
- "best" solution ≈ brute-force search + dynamic programming



This problem is <u>suspected</u> to be not solvable in polynomial time.

- We still do not know whether the problem is tractable or intractable.

Unsolvable problems?

Question: Are there unsolvable computational problems?

There are examples of unsolvable problems.

- The most famous one is called the **halting problem**.

The Halting Problem:

Given a computer program P and some input I, determine whether P will terminate when executed with input I.

- This is a yes/no problem. The answer to the halting problem is either yes or no.
  - Yes, if P terminates.
  - No, if P runs forever (e.g. enters an infinite loop).
- If I is not a valid input for P, then P executed with input I will terminate with an error message.

Understanding the halting problem

The Halting Problem: Given a computer program P, and an input I, determine (yes or no) whether the program will terminate when executed with input I.

- Consider the following Python functions.

```
def funcOne($x$):
    while $x > 0$:
        $x \leftarrow 5$
    return $x$
```

```
def funcTwo($x$):
    $x \leftarrow 2$
    while $x < 10$:
        $x \leftarrow x^2$
    return $x$
```

```
def funcThree($x$):
    print(“Hello World)
    print(“This code is correct”
```

- funcOne runs forever if $x > 0$, and terminates if either $x \leq 0$ (returns x) or x is not a numerical input (returns error).
- funcTwo always terminates and returns the value 16.
- funcThree always terminates with a syntax error message.

Understanding the halting problem

Suppose the halting problem can be solved, i.e. there is an algorithm $\mathbf{halt}(P, I)$ that takes in any program P and any input I, and gives an output either true or false.

- Output true represents “yes, program terminates on I”, output false represents “no, program does not terminate on I”.

Examples:

```
def $\mathrm{funcOne}(x)$:
    while $x > 0$:
        $x \leftarrow 5$
    return $x$
```

```
def $\mathrm{funcTwo}(x)$:
    $x \leftarrow 2$
    while $x < 10$:
        $x \leftarrow x^2$
    return $x$
```

```
def $\mathrm{funcThree}(x)$:
    print(“Hello World)
    print(“This code is correct”
```

- $\mathbf{halt}(\mathrm{funcOne}, 7) = \mathrm{false}$ (runs forever).
- $\mathbf{halt}(\mathrm{funcTwo}, 5) = \mathrm{true}$ (terminates and returns value 16).
- $\mathbf{halt}(\mathrm{funcThree}, 99) = \mathrm{true}$ (terminates with syntax error).

Why the halting problem is unsolvable

We have assumed that the halting problem is solvable, which means there is an algorithm $\mathbf{halt}(P, I)$ that solves the problem.

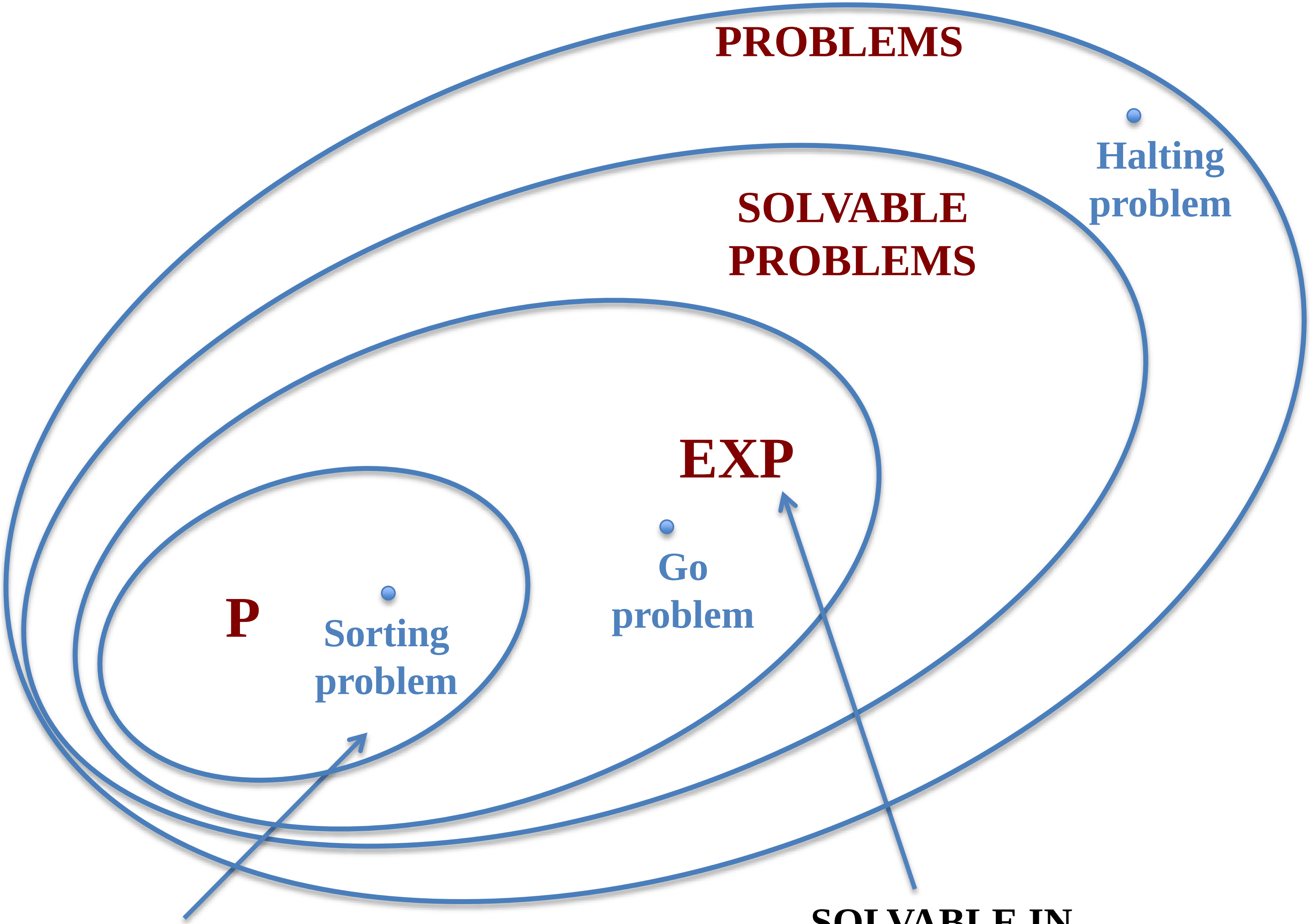
Key Idea: Consider the following Python function.

```
def funny(P):
  if halt(P, P) then:
    loop_forever()
```

Question: What is the output of $\text{funny}(\text{funny})$?

- If $\mathbf{halt}(\text{funny}, \text{funny}) = \text{true}$, then when running $\text{funny}(\text{funny})$, we enter the 'if loop' and loop forever, which means the program funny does not terminate when executed with input funny, so by the definition of **halt**, we conclude that $\mathbf{halt}(\text{funny}, \text{funny}) = \text{false}$.
- If $\mathbf{halt}(\text{funny}, \text{funny}) = \text{false}$, then when running $\text{funny}(\text{funny})$, we do not enter the 'if loop', which means the program funny terminates when executed with input funny, so by the definition of **halt**, we conclude that $\mathbf{halt}(\text{funny}, \text{funny}) = \text{true}$.

Different kinds of problems



SOLVABLE IN POLYNOMIAL TIME
(these are the tractable problems)

SOLVABLE IN
EXPONENTIAL TIME

Useful terminology (P and EXP)

We say a problem is **in P** if it can be solved in **p**olynomial time.

- E.g. the sorting problem is in P.
- E.g. the single source shortest path problem is in P.
- The notation **P** refers to the class of all problems that are solvable in polynomial time, i.e. the class of **tractable problems**.
  - **Note:** This class P consists of problems, not solutions or algorithms.

We say a problem is **in EXP** if it can be solved in **exp**onential time.

- E.g. the Go problem is in EXP, and not in P.
- E.g. the sorting problem is in EXP (and in fact in P).
- E.g. the traveling salesman problem is in EXP, but we do not know whether it is in P or not in P.
- **Note:** The class P is a subclass of EXP.
  - **Any problem in P is also a problem in EXP.**

Decision Problems

Decision problems

A problem is called a **decision problem** if its solution has two possible outcomes

- Either “Yes” or “No”.
- Either “True” or “False”.
- Either 1 or 0.
- Either “Accept” or “Reject”
- ..

Examples of decision problems:

- Chess problem
  - The white player can force a win: Yes or No?
- Halting problem
  - Program P will terminate when executed with input I: Yes or No?

Optimization problems

A problem is called an **optimization problem** if we want to find a "best" solution, given some input, and some constraints that the solution must satisfy. (There may be more than one "best" solution.)

Examples of optimization problems:

- Single source shortest path problem
  - Find a shortest path from a source vertex to each other vertex.
- Longest common subsequence problem
  - Find a longest common subsequence of two given sequences.
- 0/1 Knapsack problem
  - Choose a subset of items so that their total value is maximized, while still satisfying the constraint that their total size doesn't exceed max capacity.
- Traveling salesman problem
  - Find a shortest route that visits each city exactly once and returns to the origin city.

How do we show a problem is not in P?

Recall: When we say a problem is not in P, we mean that any possible solution, even those not yet discovered, cannot possibly run in polynomial time.

- How can we prove that a problem is not in P?
    - In other words, how can we prove that a problem is intractable?
- **Short answer:** For many problems, we don't know how!

Current Status: We do not know of any general method that works on all problems, that can prove that a problem is not in P.

- In fact, we do not even know of any general method that can prove that a problem is not solvable in linear time.
- However, for some specific problems (e.g. the Go problem), we are able to prove that they are intractable.

Verification algorithms

Decision Problem: Decide if input I satisfies a set of conditions.

Question: How do we verify an answer to the decision problem?

- An algorithm to verify an answer is called a **verification algorithm**.
- We can have two verification algorithms.
  - One verification algorithm to verify that a "**yes**" answer is correct.
  - One verification algorithm to verify that a "**no**" answer is correct.

A verification algorithm takes in two inputs I and E.

- I is the given input to the decision problem.
- E represents "evidence" that we provide to the algorithm.
  - If E is sufficient "evidence", then the algorithm outputs 1 ("accept"), and we say that E is a **certificate** for the verification algorithm.
  - If E is insufficient "evidence", then the algorithm outputs 0 ("reject").

Example: The partition problem

Problem: Given a set of integers S, determine (yes/no) if S can be partitioned into two subsets A and B, such that the integers in each of A and B have exactly the same sums.

Observation: If the input set is $S = \{1,2,4,6,7\}$, then one possible certificate for a "yes" answer to this decision problem is the pair $A = \{1,2,7\}$ and $B = \{4,6\}$.

- We can check that indeed $1 + 2 + 7 = 4 + 6$, so we can verify that the answer to the decision problem is indeed "yes".

Note: A certificate is not a direct answer "yes" or "no".

- Instead, a certificate is "enough information" so that you can verify for yourself the answer for the given input set S.
- If instead we are given the pair $A = \{1,2,4\}$ and $B = \{6,7\}$, then this pair is not a certificate for any of the two answers.

Non-deterministic algorithms and NP problems

Guessing solutions to a problem

Decision problem: Given input I, does there exist an outcome E (dependent on I) that satisfies some given constraints? Yes or no?

- (e.g. Partition problem: Given an input set I, does there exists a partition E of I into two subsets with the same sum? Yes or no?)

Consider a verification algorithm $\boldsymbol{A}(I, E)$ that verifies a "yes" answer to this decision problem. (i.e. output $= 1$ for "yes", output $= 0$ for "inconclusive")

A different strategy for solving a decision problem:

1) Randomly generate a valid outcome E for $\boldsymbol{A}(I, E)$. *(guessing)*
2) Compute the value of $\boldsymbol{A}(I, E)$. *(verifying the guess)*
3) If $\boldsymbol{A}(I, E) = 1$, then return "yes" as the solution to the decision problem. Otherwise, go back to Step 1.

Note: Once we have found some E such that $\boldsymbol{A}(I, E) = 1$, then this E is a certificate for a "yes" answer to the decision problem.

Solving the partition problem by guessing

Partition problem: Given an input set I, does there exists a partition E of I into two subsets with the same sum? Yes or no?

- Suppose we are given the input set $I = \{1,3,4,5,6,7\}$.

Let $\boldsymbol{A}(I, E)$ be a verification algorithm to verify a "yes" answer.

Guessing potential certificates:

1) Randomly generate a valid outcome E for $\boldsymbol{A}(I, E)$. *(guessing)*
2) Compute the value of $\boldsymbol{A}(I, E)$. *(verifying the guess)*
3) If $\boldsymbol{A}(I, E) = 1$, then return "yes"; Otherwise, go back to Step 1.

Best case scenario: If the first randomly generated E consists of $A = \{3,4,6\}, B = \{1,5,7\}$, then we can check that A and B have the same sums, conclude that $\boldsymbol{A}(I, E) = 1$, and return "yes". *(a lucky guess)*

Worst case scenario: If each randomly generated E keeps on yielding $\boldsymbol{A}(I, E) = 0$, then the process does not end. *(bad guesses one after another)*

Non-deterministic algorithms

A non-deterministic algorithm for verifying a "yes" answer:

Input: I

1) Randomly generate a valid outcome E for $\boldsymbol{A}(I, E)$. *(guessing)*
2) Compute the value of $\boldsymbol{A}(I, E)$. *(verifying the guess)*
3) If $\boldsymbol{A}(I, E) = 1$, then return "yes"; Otherwise, go back to Step 1.

- An algorithm is called **non-deterministic** if it is possible to have different behaviors on different runs, even for the same input I.
- In contrast, an algorithm is called **deterministic** if it behaves exactly the same on different runs, for the same input I.

Important Remark:

Step 1 is usually "fast", so in the best case scenario, the time complexity of this non-deterministic algorithm depends on the time complexity of Step 2, i.e. the **verification of a certificate** for a "yes" answer.

What is NP?

A **decision problem** is said to be in class **NP** if there is at least one non-deterministic algorithm that is able to verify a “**yes**” answer to the decision problem, such that in the best case scenario, this non-deterministic algorithm runs in **polynomial time**.

- NP ≈ **solvable in Non-deterministic Polynomial time**.
- “problem in NP” = “problem in class NP” = “NP problem”.

Another equivalent definition for NP

A **decision problem** is in **NP** if for every input whose corresponding correct answer should be “yes”, there is a certificate for a “yes” answer that can verified in polynomial time.

- **Important Note:** NP does **not** say anything about the verification of a “no” answer. NP only concerns the verification of a “yes” answer!

Intuition for NP

For a decision problem, we need to decide whether the correct answer is “yes” or “no”, given some input I.

- Whatever answer that we decide, we have to verify that our answer is indeed correct. One way is to provide a certificate that can verify our answer.

Suppose we are only interested in verifying a “**yes**” answer.

We say that the **decision problem** is in **NP** if we can guess a certificate for a “**yes**” answer, such that it takes polynomial time to verify that your guess indeed certifies that the correct answer should be “**yes**”.

- Guessing is a non-deterministic process, that’s why NP stands for “**N**on-deterministic **P**olynomial time”. We can verify a “**yes**” answer non-deterministically in polynomial time.

NP: Polynomial time in terms of what?

Consider a decision problem. (Input = I, Output = yes/no)

- **Assumptions:**
  - Input I requires n bits of memory for storing it.
  - Correct answer for decision problem with input I is “yes”.

We say that the decision problem is in NP to mean the following:

- There is a “short” certificate E for a “yes” answer to the decision problem with this given input I.
- This certificate E is “short”, meaning that it requires at most $O(n^c)$ bits of memory for storing it, for some constant c.
- Verifying this “short” certificate takes polynomial time, i.e. there is some verification algorithm $\boldsymbol{A}$, such that running $\boldsymbol{A}(I, E)$ would take at most $O(n^c)$ steps, for some constant c.

Note: By considering “polynomial time” in terms of the number of bits required to store the input, the class NP becomes much more general and includes many interesting decision problems.

An example of an NP problem

Fact: The **partition problem** is in NP.

- Problem: Given an input set I of non-negative integers, determine (yes/no) if I can be partitioned into two subsets of equal sums.

Example: Consider $I = \{1,3,4,5,6,7\}$.

- Suppose I requires n bits of memory for storing it.
  - If we consider each element of I as an 8-bit unsigned integer, then we can store I as an array that takes up $6 \times 8 = 48$ bits of memory.
- A certificate for a "yes" answer to the partition problem with input set I is the partition consisting of two subsets $\{3,4,6\}$ and $\{1,5,7\}$.
  - We can store this certificate as a list consisting of two arrays, each with three 8-bit unsigned integers. Hence, storing this list takes $O(n)$ bits of memory.
- The verification of this certificate involves computing the two sums $3 + 4 + 6$ and $1 + 5 + 7$, and comparing if the two sums have equal values. This verification process takes $O(n)$ steps.

Polynomial-time reductions

The idea of reductions

There are so many different computational problems that we may want to solve.

- These problems can seem very different! Do we have to solve every single one of these problems from scratch?

Key Idea of reductions

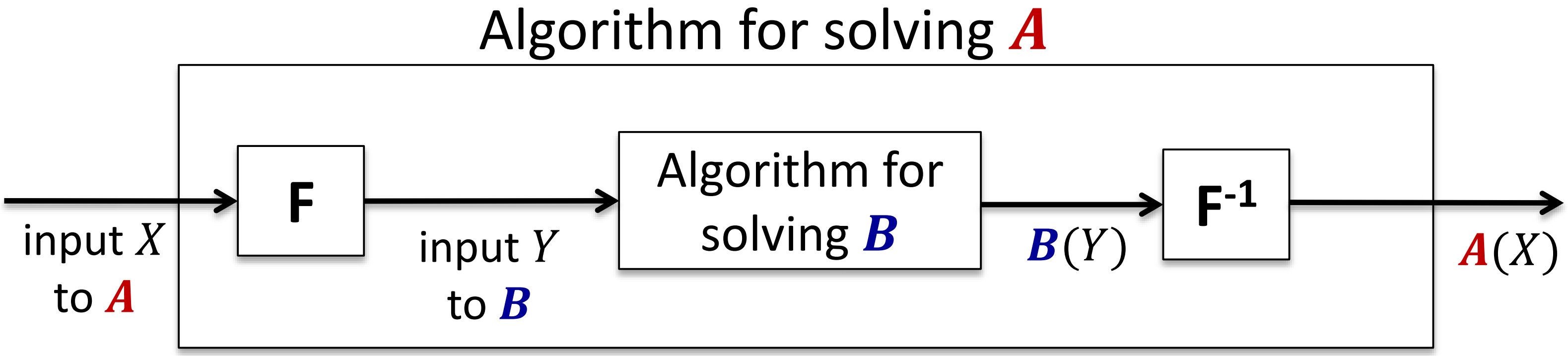
Given a Problem A that we want to solve, and suppose there is another Problem B that we already know to solve.

- Suppose we can reformulate Problem A to “look like” Problem B, so that by starting with a solution to Problem B, we are able to solve Problem A.
  - This reformulation could be as simple as a change in notation, or it could be an algorithmic process to use computed solutions to Problem B, do further computations, so as to generate one or more solutions to Problem A.
- Then we say that we have **reduced** Problem A to Problem B.

How do reductions work?

Assumptions: We have two problems $\boldsymbol{A}$ and $\boldsymbol{B}$.

- Problem $\boldsymbol{A}$: We want to design an algorithm to solve it.
- Problem $\boldsymbol{B}$: We already know an algorithm for solving it.

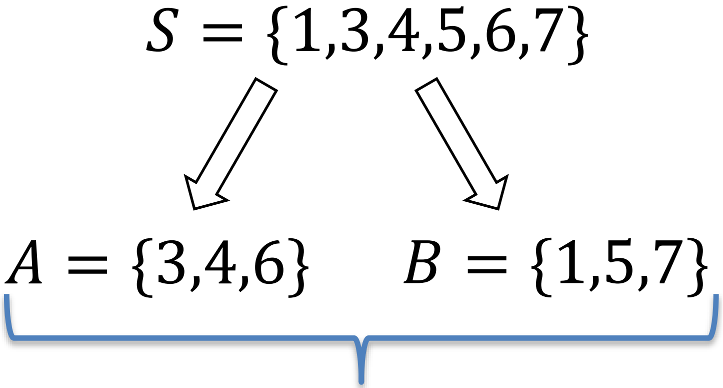

Algorithm for solving $\boldsymbol{A}$
input X to $\boldsymbol{A}$
F
input Y to $\boldsymbol{B}$
Algorithm for solving $\boldsymbol{B}$
$\boldsymbol{B}(Y)$
F-1
$\boldsymbol{A}(X)$

- **F**: An algorithm that transforms an input for Problem $\boldsymbol{A}$ to an input for Problem $\boldsymbol{B}$.
- **F^{-1}**: An algorithm that transforms a solution to Problem $\boldsymbol{A}$ to a solution to Problem $\boldsymbol{B}$.
- Using **F** and **F^{-1}**, we have reduced Problem $\boldsymbol{A}$ to Problem $\boldsymbol{B}$.

Example: Reduction of the partition problem

Fact: The **partition problem** can be reduced to the **knapsack problem**.

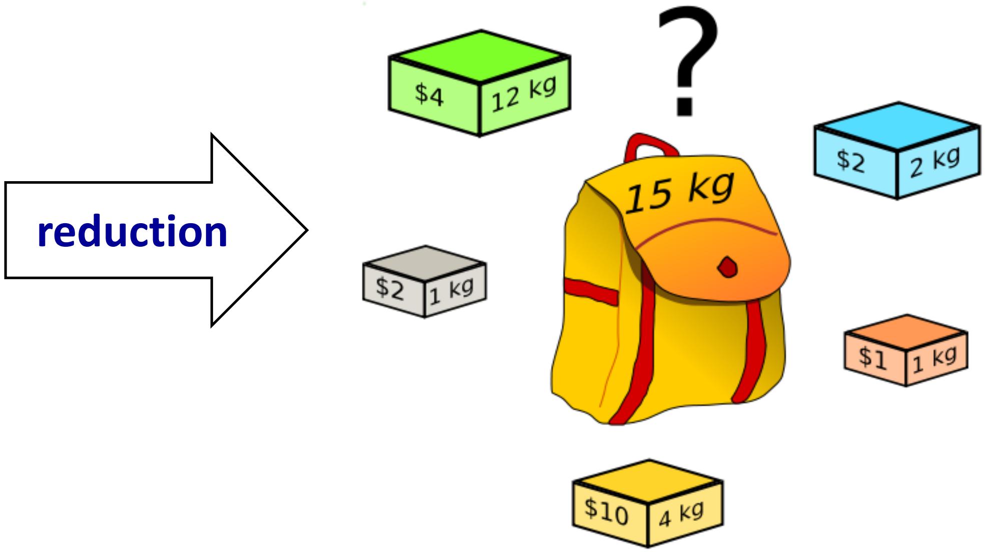
- Partition problem: Given a set S of integers, can we partition S into two subsets whose elements have equal sums?
- Knapsack problem (L11.02): Given a knapsack of maximum capacity m, and given n items to choose from to put inside the knapsack, where the i-th item has size s_i and value v_i, find the maximum total value possible of the chosen items, so that their total size is $\leq m$.



$S = \{1,3,4,5,6,7\}$

$A = \{3,4,6\}$ $B = \{1,5,7\}$

A and B have equal sums.



Example: Reduction of the partition problem

Input to the partition problem: A set $S = \{a_1, \dots, a_n\}$ of integers.

First observation: If $a_1, \dots, a_n$ do not add up to an even integer, then the answer to the partition problem is "no".

- Remaining case: $a_1 + \cdots + a_n = 2k$ for some integer k.

Reformulation of the partition problem: Is there a subset $T \subseteq S$ such that the integers in T add up to k?

- If reformulated problem has answer "yes", then S can be partitioned into two subsets T and $S \backslash T$, each with the same sum k.
- Conversely, if partition problem has answer "yes", then either of the two subsets can be a certificate for the reformulated problem.
  - "yes" to reformulated problem if and only if "yes" to partition problem.

Question: Can you see how this reformulated partition problem can be viewed as a knapsack problem?

Example: Reduction of the partition problem

Reformulated partition problem: Is there a subset $T \subseteq S = \{a_1, \ldots, a_n\}$ such that the integers in T add up to k?

Key Idea: Consider the knapsack problem with maximum capacity k, and n items, such that the i-th item has size $s_i = a_i$ and value $v_i = a_i$.

- Our goal for this knapsack problem is to find the maximum possible total value of any subset of the n items that does not have a total size exceeding k.

Two possibilities:

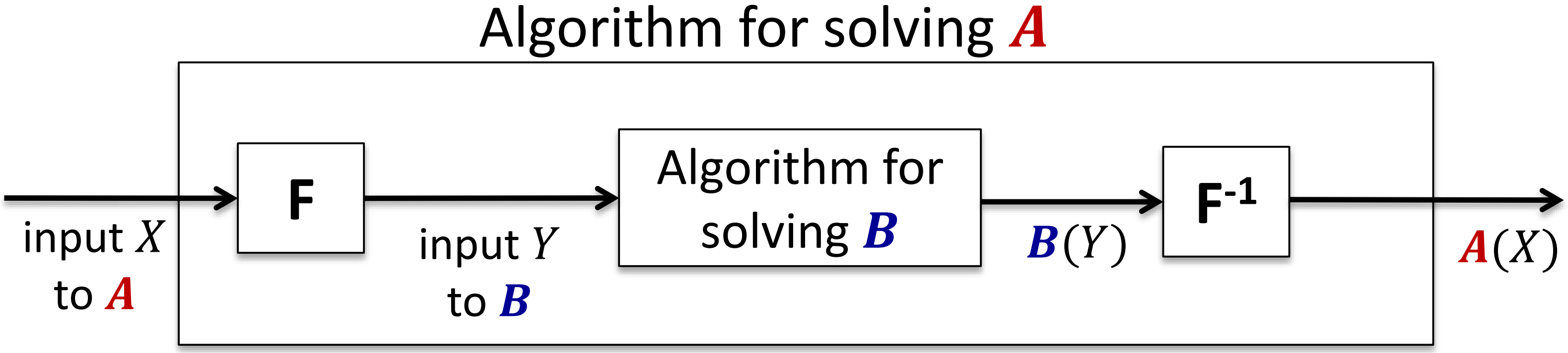
- If this maximum possible total value is k, then we can find a subset of $\{a_1, \ldots, a_n\}$ whose sum is exactly k.
- If this maximum possible total value is $< k$, then every possible subset of $\{a_1, \ldots, a_n\}$ whose sum is $\leq k$ must have a sum $< k$.

Conclusion: By solving this knapsack problem, we can solve the reformulated partition problem, and thus solve the partition problem.

Polynomial-time reductions

Assumptions: We have two problems $\boldsymbol{A}$ and $\boldsymbol{B}$.

- Problem $\boldsymbol{A}$: We want to design an algorithm to solve it.
- Problem $\boldsymbol{B}$: We already know an algorithm for solving it.

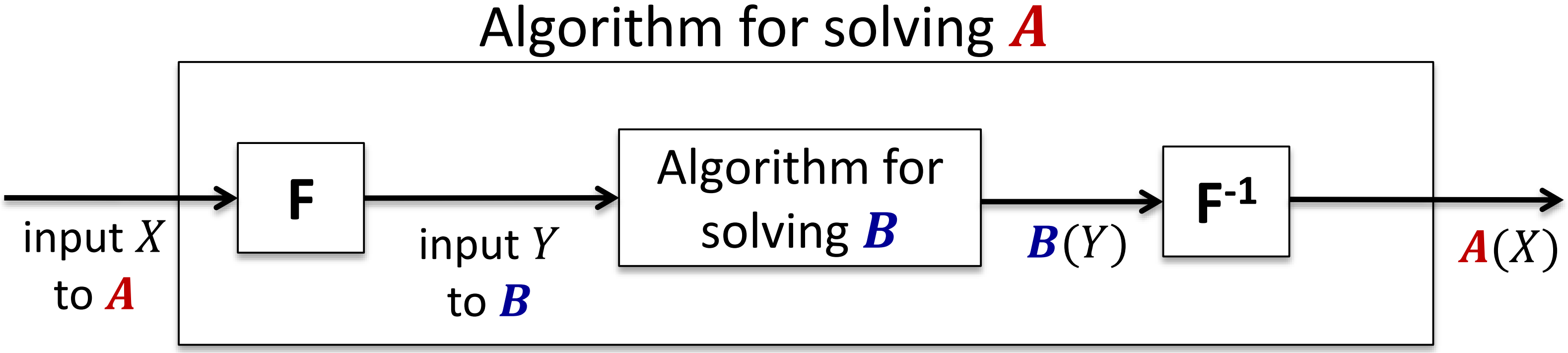

Algorithm for solving $\boldsymbol{A}$
input X to $\boldsymbol{A}$
F
input Y to $\boldsymbol{B}$
Algorithm for solving $\boldsymbol{B}$
$\boldsymbol{B}(Y)$
F^{-1}
$\boldsymbol{A}(X)$

- We use **F** and **F^{-1}** to reduce Problem $\boldsymbol{A}$ to Problem $\boldsymbol{B}$.

Definition: If both algorithms **F** and **F^{-1}** run in polynomial time, then we say that the reduction of Problem $\boldsymbol{A}$ to Problem $\boldsymbol{B}$ is a **polynomial-time reduction**.

Polynomial-time reductions are important

If there is a polynomial-time reduction of Problem $\boldsymbol{A}$ to a known tractable Problem $\boldsymbol{B}$, then Problem $\boldsymbol{A}$ is tractable!


Algorithm for solving $\boldsymbol{A}$
input X to $\boldsymbol{A}$
F
input Y to $\boldsymbol{B}$
Algorithm for solving $\boldsymbol{B}$
$\boldsymbol{B}(Y)$
$\mathbf{F}^{-1}$
$\boldsymbol{A}(X)$

- So we can solve a seemingly “hard” problem by finding a polynomial-time reduction to an “easier” problem we already know how to solve.

Such polynomial-time reductions are very common, so we have the useful notation $\boldsymbol{A} \leq_p \boldsymbol{B}$ to mean there is polynomial-time reduction from Problem $\boldsymbol{A}$ to Problem $\boldsymbol{B}$.

- If $\boldsymbol{A} \leq_p \boldsymbol{B}$ and $\boldsymbol{B} \leq_p \boldsymbol{A}$, then we write $\boldsymbol{A} \cong_p \boldsymbol{B}$.

Polynomial-time reductions: Consequences

Suppose $\boldsymbol{A}$ and $\boldsymbol{B}$ are problems such that $\boldsymbol{A} \leq_p \boldsymbol{B}$.

- If $\boldsymbol{B}$ is in P, then $\boldsymbol{A}$ is also in P.
- Contrapositive: If $\boldsymbol{A}$ is not in P, then $\boldsymbol{B}$ is also not in P.
  - Indeed, if $\boldsymbol{A}$ is not solvable in polynomial time, then $\boldsymbol{B}$ cannot possibly be solvable in polynomial time, otherwise we can combine such a polynomial time solution for $\boldsymbol{B}$ with the polynomial-time reduction to get a polynomial time algorithm that solves $\boldsymbol{A}$.
- If we believe that $\boldsymbol{A}$ is intractable, then we should also believe that $\boldsymbol{B}$ is also intractable.

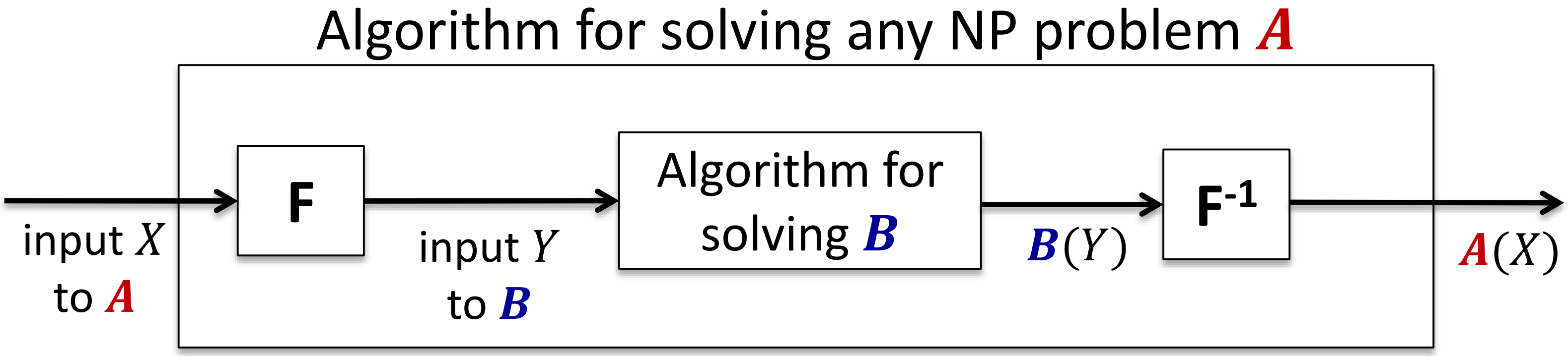
Intuition: $\boldsymbol{A} \leq_p \boldsymbol{B}$ means that $\boldsymbol{B}$ is “at least as hard as” $\boldsymbol{A}$.

- After all, if we have reason to believe that $\boldsymbol{A}$ is “hard” (intractable), then we should also believe that $\boldsymbol{B}$ is also “hard” (intractable).
- We have a reduction of $\boldsymbol{A}$ to $\boldsymbol{B}$, so if we are stuck after the reduction (i.e. stuck at solving $\boldsymbol{B}$), then solving $\boldsymbol{B}$ is at least as hard as solving $\boldsymbol{A}$.

What is NP-hard?

A problem $\boldsymbol{B}$ is said to be **NP-hard** if every problem $\boldsymbol{A}$ in NP has a polynomial-time reduction to $\boldsymbol{B}$.

- **Note:** A deterministic algorithmic solution for $\boldsymbol{B}$ can be used as a subroutine to solve every problem in NP deterministically.


Algorithm for solving any NP problem $\boldsymbol{A}$
input X to $\boldsymbol{A}$
F
input Y to $\boldsymbol{B}$
Algorithm for solving $\boldsymbol{B}$
$\boldsymbol{B}(Y)$
F^{-1}
$\boldsymbol{A}(X)$

Intuition: If we can find a deterministic algorithmic solution for $\boldsymbol{B}$ that runs in polynomial time, then this solution can be used to design a polynomial time algorithmic solution for every problem in NP. Therefore, solving an NP-hard problem should be "hard".

What is NP-complete?

A problem is said to be **NP-complete** if it is both an NP problem and an NP-hard problem.

- NP ≈ solvable in non-deterministic polynomial time
- NP-hard ≈ at least as hard as every NP problem

Intuition: An NP-complete problem is a “hardest” problem in NP.

Theorem (Cook–Levin, 1971): The SAT problem is NP-complete.

(This was the first time ever that a problem was shown to be NP-complete.)

- **SAT** (Satisfiability): given a boolean formula, can you make it TRUE;

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3)$$

There are many NP-complete problems!

- Some examples of NP-complete problems:
  - SAT problem, 3-SAT problem, graph coloring problem, 4-way matching problem, vertex cover problem, Hamiltonian path problem, longest path problem, clique problem, independent set problem, multiprocessor scheduling problem, max-cut problem, quadratic programming, integer linear programming, etc.
  - These are all the joint “hardest” problems in NP.
- For many decades, nobody has found a polynomial time algorithm to solve any of these NP-complete problems.

Important Note: A polynomial time algorithm to solve just **one** NP-complete problem would imply polynomial time algorithms to solve **all** NP-complete problems.

P versus NP problem

The P versus NP problem

The P versus NP problem: Is P $=$ NP?

- Most famous unsolved problem in computer science!
- The Clay Mathematics Institute (based in the United States) offers a prize of US$1 million for anyone who can solve this “P versus NP problem”.

Understanding the problem:

P $=$ set of all problems solvable in (deterministic) polynomial time

NP $=$ set of all problems solvable in non-deterministic polynomial time

Technicality: To have a fair comparison, the input size of problems in P should be in terms of the number of bits used to store the input, just like in the case of NP.

Fact: P $\subseteq$ NP.

- For any problem in P, there is by definition a polynomial time algorithm to solve the problem, so we can go ahead and solve the problem in polynomial time, and use this solution to verify any certificate for a “yes” answer to the “converted” decision problem.

Remarks on the P versus NP problem

Intuition: P represents a set of relatively easy problems, while NP includes some very hard problems.

Case: $P \neq NP$

- Consequence: NP-complete problems are indeed intractable!
- Philosophical implication: Solving hard problems is harder than verifying solutions to hard problems.

Case: $P = NP$.

- Philosophical Implication: All the seemingly hard problems (including the long list of NP-complete problems) actually have relatively easy solutions that have eluded humans for decades!
- Practical consequences (if an explicit polynomial time algorithm is found):
  - Efficient solutions for all NP-complete problems
  - Cryptographic hashing will no longer be secure.

So, what do you think?
P $=$ NP or P $\neq$ NP?

(Or do you think the P versus NP problem is unsolvable?)