Lesson 7:
Hard Problems

CSC325 – ADVANCED DATA STRUCTURES & ALGORITHMS | SPRING 2022

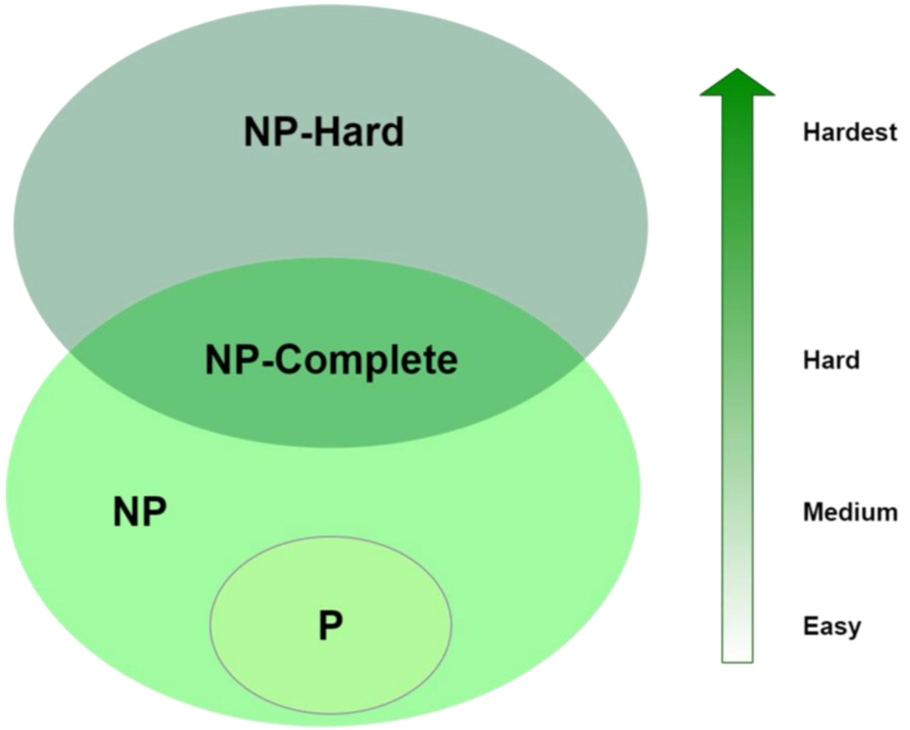
DR. ANDREY TIMOFEYEV

OUTLINE

- Introduction.
- Polynomial-time problems.
- Nondeterministic polynomial-time problems.
- Reduction.
- NP-hard and NP-complete problems.
- P = NP.

INTRODUCTION

- In **theoretical computer science, classification** & **complexity** of problems is defined by their "**hardness**".
  - Is there **any** algorithm to **solve** the problem?
  - Is there an **efficient** algorithm to **solve** the problem?
- **Computational problems classification:**
  - **Polynomial-time problems.**
    - P-problems.
  - **Nondeterministic polynomial-time problems.**
    - NP-problems.
  - **NP-complete problems.**
  - **NP-hard problems.**



Classes of computational problems

POLYNOMIAL-TIME PROBLEMS

- **Polynomial-time problems.**
  - All problems that are **solvable** in **polynomial time** based on some input size.
  - **Polynomial time** = $\mathrm{n}^{O(1)}$, where n = input size.
- **Polynomial-time problems** are considered as ***"easy"*** problems.
  - **Solvable** & **tractable** (solved in theory and practice).
- **Examples of polynomial-time problems:**
  - **Searching**.
    - Linear search – O(n).
    - Binary search – O(logn).
  - **Sorting**.
    - Insertion sort – O(n^2).
    - Heap sort – O(nlogn).
  - **Finding shortest path.**
    - Dijkstra's algorithm – O(V^2).

NONDETERMINISTIC POLYNOMIAL-TIME PROBLEMS

- **Nondeterministic polynomial-time problems.**
  - All problems that are **solvable** in **exponential time** but can be **verified** in **polynomial time**.
    - **Long** time to **solve**, **short** time to **verify**.
  - **Exponential time** = $O(1)^n$, where *n* = input size.
- **NP-problems** are considered as **“hard”** problems.
  - **Decision problems** solved by **nondeterministic machines**.
- **NP-problems** are treated as **decision** problems.
  - Output either **YES** or **NO**.
- **Nondeterministic** – solution can be **guessed** out of **polynomially** many options in **O(1)** time.
  - If **any** guess = **YES** -> **nondeterministic** algorithm will make **that guess**.

NP PROBLEMS: 3SAT

- **Example of NP problem: 3-satisfiability (3SAT) problem.**
  - Given a Boolean formula of a form: $(x_1 \text{ OR } x_2 \text{ OR } \overline{x_6}) \text{ AND } (\overline{x_6} \text{ OR } x_2 \text{ OR } \overline{x_7}) \text{ AND } \ldots$
  - Are there $\mathrm{x_1, x_2, \ldots, x_n = True/False}$, such that the entire formula evaluates to $\mathrm{True?}$
- **3SAT problem is NP.**
  - **Solving** requires **exponential time**.
  - **Verifying** the solution only requires **polynomial time**.
    - **Solution** = list of assignment of each variable.
    - **Verification** = check if statement is evaluated to true.

REDUCTION (1)

- In computational complexity, **reduction** allows **solving one problem** in **terms** of **another**.
  - Allows making **relative statements** about **upper** & **lower** bounds on the **cost** of a **problem**.
- **Reduction** - **polynomial-time algorithm** that **converts inputs** to one problem into **equivalent inputs** to another problem.
  - **Both** problems output the same **YES** or **NO** answer for the **input** and **converted** input.
- **Formal reduction process consist of three steps.**
  - **First problem** takes **input** I and transform it to **solution** SLN
  - **Second problem** takes **input** I' and transform it to **solution** SLN'.
  - **Reduction steps:**
    - **Transform** an arbitrary **instance** of the **first problem** to an **instance** of the **second** problem. $I \rightarrow I'$.
    - **Apply** an **algorithm** for the **second problem** to the **instance** I', yielding a **solution** SLN'.
    - **Transform** SLN' to the **solution** of I, known as SLN.
      - SLN must be the correct solution for I.

REDUCTION (2)

- **Reduction does not provide** an **algorithm** for **solving** either problem.
  - **Method** for **solving** the first problem given the **solution** to the second.
- **Reduction main goal** – **estimate bounds** of one problem in terms of another.
  - **Upper bound** of **first problem** is at **most upper bound** of the **second**.
  - **Lower bound** of **second problem** is at **least lower bound** of the **first**.
- **Example: reducing PAIRING into SORTING.**

SORTING:

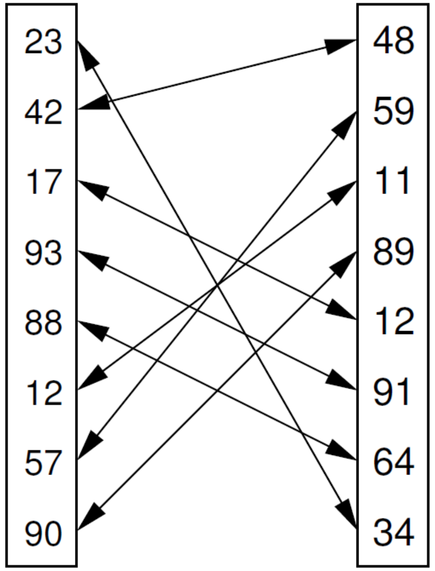
Input: A sequence of integers $x_0, x_1, x_2, ..., x_{n-1}$.

Output: A permutation $y_0, y_1, y_2, ..., y_{n-1}$ of the sequence such that $y_i \leq y_j$ whenever $i < j$.

PAIRING:

Input: Two sequences of integers $X = (x_0, x_1, ..., x_{n-1})$ and $Y = (y_0, y_1, ..., y_{n-1})$.

Output: A pairing of the elements in the two sequences such that the least value in X is paired with the least value in Y, the next least value in X is paired with the next least value in Y, and so on.



Pairing example

REDUCTION (3)

- **Example: number scrabble.**

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

REDUCTION (3)

- **Example: number scrabble.**

1	2	3	4	5	6	7	8	9

2	7	6
9	5	1
4	3	8

REDUCTION (3)

- **Example: number scrabble.**

1	2	3	4	5	6	7	8	9

2	7	6
9	5	1
4	3	8

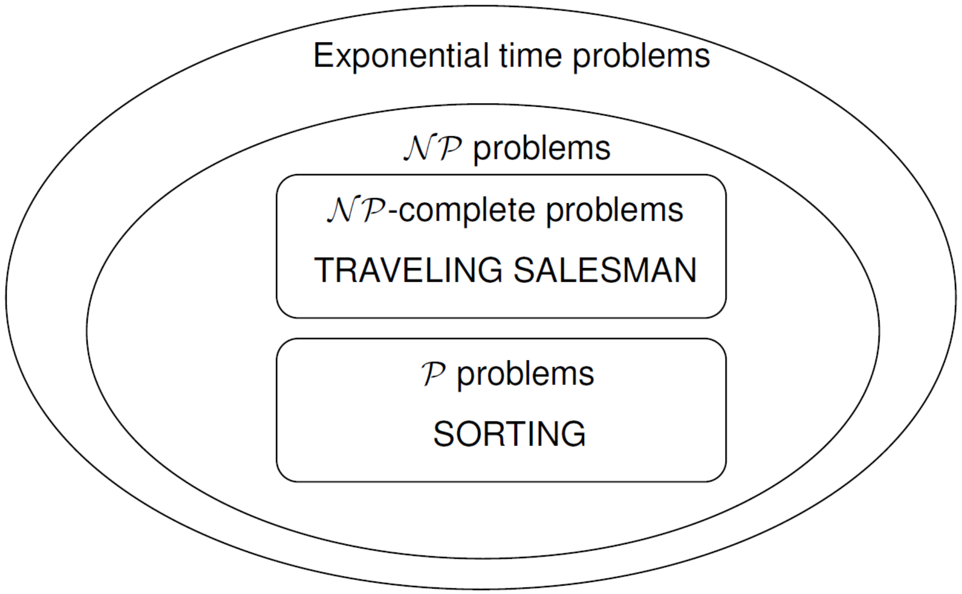
X	O	O
O	O	X
X	X	X

NP-COMPLETE PROBLEMS

- **Problem hardness** is defined by the **runtime** of the **algorithm** that solves it.
  - **Hard problem** = **best-known algorithm** to **solve** the problem is **expensive** in running time.
    - **Example**: Towers of Hanoi with $O(n^2)$ complexity.
- **NP-complete problems.**
  - Problem X is **NP-complete** if it is **NP problem** AND **NP-hard problem**.
  - **Hardest** of the **problems** to which solution can be **verified** in **polynomial time**.
- **Completeness.**
  - For any **NP problem** that is **complete**, there exists a **polynomial-time reduction algorithm** that can **transform** the problem into any other **NP-complete problem**.
- **Examples** of **NP-complete** problems:
  - Travelling salesman.
  - Knapsack.
  - Graph coloring.

NP-HARD PROBLEMS

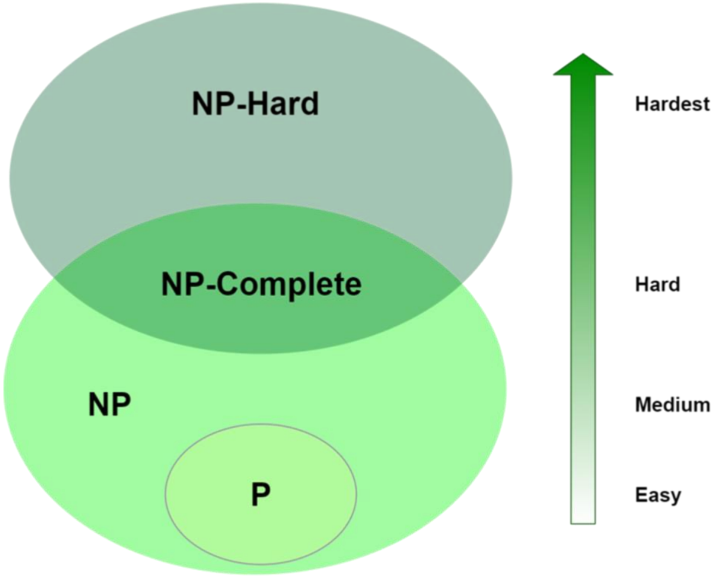
- **NP-hard problems.**
  - Problem X is **NP-hard** if every problem Y in NP can be **reduced** to X in polynomial time.
  - Given an **efficient** algorithm to solve **NP-hard problem** X, an **efficient algorithm** for ***ANY*** problem in **NP** can be **constructed**.
- **NP-hard problems** are **hard** to **solve** AND **hard** to **verify**.
  - At least as hard the hardest problems in NP.
- Assuming **P ≠ NP**, then **NP-hard** problems are **not** in P.
- **Examples** of **NP-hard** problems:
  - K-means clustering.
  - Traveling salesman.
  - Graph coloring.



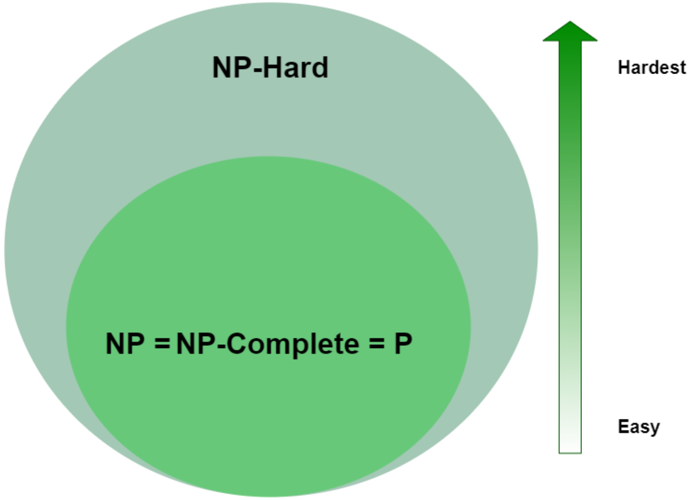
P & NP problems

P = NP OR P ≠ NP

- Major **unsolved problem** in computer science: **P vs NP**.
  - Can every problem that can be **verified** in **polynomial time** also be **solved** in **polynomial time**.
    - **Is P = NP?**
- If **P = NP**, then any **NP** or **NP-complete** problem can be **solved** in **polynomial time**.
  - Through reduction, if one NP-complete problem in P, then all NP-complete problems are in P.
- **Modern computer science** operates on **P ≠ NP** assumption.
- P vs NP is one of the seven **Millennium Prize Problems**.



P ≠ NP



P = NP