Unit 1A: Computational Complexity

• Course contents:
  – Computational complexity
  – NP-completeness
  – Algorithmic Paradigms

• Readings
  – Chapters 3, 4, and 5

<table>
<thead>
<tr>
<th>Time</th>
<th>Big-Oh</th>
<th>( n = 10 )</th>
<th>( n = 100 )</th>
<th>( n = 10^3 )</th>
<th>( n = 10^6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>( O(1) )</td>
<td>5 ( \times 10^{-7} ) sec</td>
<td>5 ( \times 10^{-7} ) sec</td>
<td>5 ( \times 10^{-7} ) sec</td>
<td>5 ( \times 10^{-7} ) sec</td>
</tr>
<tr>
<td>3( n )</td>
<td>( O(n) )</td>
<td>3 ( \times 10^{-7} ) sec</td>
<td>3 ( \times 10^{-6} ) sec</td>
<td>3 ( \times 10^{-6} ) sec</td>
<td>3 ( \times 10^{-6} ) sec</td>
</tr>
<tr>
<td>( n \log n )</td>
<td>( O(n \log n) )</td>
<td>3 ( \times 10^{-7} ) sec</td>
<td>2 ( \times 10^{-7} ) sec</td>
<td>3 ( \times 10^{-6} ) sec</td>
<td>0.001 sec</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>( O(n^2) )</td>
<td>1 ( \times 10^{-7} ) sec</td>
<td>1 ( \times 10^{-5} ) sec</td>
<td>0.001 sec</td>
<td>1 sec</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>( O(n^3) )</td>
<td>1 ( \times 10^{-7} ) sec</td>
<td>0.001 sec</td>
<td>1 sec</td>
<td>16.7 min</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>( O(2^n) )</td>
<td>1 ( \times 10^{-6} ) sec</td>
<td>3 ( \times 10^{17} ) cent.</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
<tr>
<td>( n! )</td>
<td>( O(n!) )</td>
<td>0.003 sec</td>
<td>( \infty )</td>
<td>( \infty )</td>
<td>( \infty )</td>
</tr>
</tbody>
</table>
\textbf{O: Upper Bounding Function}

- \textbf{Def:} \( f(n) = O(g(n)) \) if \( \exists c > 0 \) and \( n_0 > 0 \) such that \( 0 \leq f(n) \leq cg(n) \) for all \( n \geq n_0 \).
  - Examples: \( 2n^2 + 3n = O(n^2) \), \( 2n^2 = O(n^3) \), \( 3n \log n = O(n^2) \)
- Intuition: \( f(n) \) “\( \leq \)” \( g(n) \) when we ignore constant multiples and small values of \( n \).
**Big-O Notation**

- How to show $O$ (Big-Oh) relationships?
  
  \[
  f(n) = O(g(n)) \text{ iff } \lim_{n \to \infty} \frac{f(n)}{g(n)} = c \text{ for some } c \geq 0.
  \]

- “An algorithm has worst-case running time $O(f(n))$”: there is a constant $c$ s.t. for every $n$ big enough, every execution on an input of size $n$ takes at most $cf(n)$ time.
Computational Complexity

• **Computational complexity**: an abstract measure of the time and space necessary to execute an algorithm as function of its “input size”.

• **Input size examples**:
  - sort $n$ words of bounded length $\Rightarrow n$
  - the input is the integer $n \Rightarrow \lg n$
  - the input is the graph $G(V, E) \Rightarrow |V|$ and $|E|$

• **Time complexity** is expressed in *elementary computational steps* (e.g., an addition, multiplication, pointer indirection).

• **Space Complexity** is expressed in *memory locations* (e.g. bits, bytes, words).
Asymptotic Functions

• Polynomial-time complexity: \( O(n^k) \), where \( n \) is the input size and \( k \) is a constant.

• Example polynomial functions:
  - 999: constant
  - \( \lg n \): logarithmic
  - \( \sqrt{n} \): sublinear
  - \( n \): linear
  - \( n \lg n \): loglinear
  - \( n^2 \): quadratic
  - \( n^3 \): cubic

• Example non-polynomial functions
  - \( 2^n, 3^n \): exponential
  - \( n! \): factorial
### Running-time Comparison

- Assume 1000 MIPS (Yr: 200x), 1 instruction /operation

<table>
<thead>
<tr>
<th>Time</th>
<th>Big-Oh</th>
<th>(n = 10)</th>
<th>(n = 100)</th>
<th>(n = 10^3)</th>
<th>(n = 10^6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>(O(1))</td>
<td>(5 \times 10^{-7}) sec</td>
<td>(5 \times 10^{-7}) sec</td>
<td>(5 \times 10^{-7}) sec</td>
<td>(5 \times 10^{-7}) sec</td>
</tr>
<tr>
<td>(3n)</td>
<td>(O(n))</td>
<td>(3 \times 10^{-8}) sec</td>
<td>(3 \times 10^{-7}) sec</td>
<td>(3 \times 10^{-6}) sec</td>
<td>(0.003) sec</td>
</tr>
<tr>
<td>(n \log n)</td>
<td>(O(n \log n))</td>
<td>(3 \times 10^{-8}) sec</td>
<td>(2 \times 10^{-7}) sec</td>
<td>(3 \times 10^{-6}) sec</td>
<td>(0.006) sec</td>
</tr>
<tr>
<td>(n^2)</td>
<td>(O(n^2))</td>
<td>(1 \times 10^{-7}) sec</td>
<td>(1 \times 10^{-5}) sec</td>
<td>(0.001) sec</td>
<td>(16.7) min</td>
</tr>
<tr>
<td>(n^3)</td>
<td>(O(n^3))</td>
<td>(1 \times 10^{-6}) sec</td>
<td>(0.001) sec</td>
<td>(1) sec</td>
<td>(3 \times 10^5) cent.</td>
</tr>
<tr>
<td>(2^n)</td>
<td>(O(2^n))</td>
<td>(1 \times 10^{-6}) sec</td>
<td>(3 \times 10^{-17}) cent.</td>
<td>(\infty)</td>
<td>(\infty)</td>
</tr>
<tr>
<td>(n!)</td>
<td>(O(n!))</td>
<td>(0.003) sec</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>(\infty)</td>
</tr>
</tbody>
</table>
Optimization Problems

- **Problem:** a general class, e.g., “the shortest-path problem for directed acyclic graphs.”
- **Instance:** a specific case of a problem, e.g., “the shortest-path problem in a specific graph, between two given vertices.”
- **Optimization problems:** those finding a legal configuration such that its cost is minimum (or maximum).
  - MST: Given a graph $G=(V, E)$, find the cost of a minimum spanning tree of $G$.
- An instance $I = (F, c)$ where
  - $F$ is the set of *feasible solutions*, and
  - $c$ is a *cost function*, assigning a cost value to each feasible solution $c : F \rightarrow R$
  - The solution of the optimization problem is the feasible solution with optimal (minimal/maximal) cost
- c.f., **Optimal** solutions/costs, optimal (*exact*) algorithms (Attn: optimal $\neq$ exact in the theoretic computer science community).
The Traveling Salesman Problem (TSP)

- TSP: Given a set of cities and that distance between each pair of cities, find the distance of a "minimum tour" starts and ends at a given city and visits every city exactly once.
**Decision Problem**

- **Decision problems:** problem that can only be answered with “yes” or “no”
  - MST: Given a graph \( G=(V, E) \) and a bound \( K \), is there a spanning tree with a cost at most \( K \)?
  - TSP: Given a set of cities, distance between each pair of cities, and a bound \( B \), is there a route that starts and ends at a given city, visits every city exactly once, and has total distance at most \( B \)?

- A decision problem \( \Pi \), has instances: \( I = (F, c, k) \)
  - The set of of instances for which the answer is “yes” is given by \( Y_{\Pi} \).
  - A subtask of a decision problem is *solution checking*: given \( f \in F \), checking whether the cost is less than \( k \).

- Could apply binary search on decision problems to obtain solutions to optimization problems.
- NP-completeness is associated with decision problems.
The Circuit-Satisfiability Problem (Circuit-SAT):

- **Instance:** A combinational circuit $C$ composed of AND, OR, and NOT gates.
- **Question:** Is there an assignment of Boolean values to the inputs that makes the output of $C$ to be 1?

A circuit is satisfiable if there exists a set of Boolean input values that makes the output of the circuit to be 1.

- Circuit (a) is satisfiable since $<x_1, x_2, x_3> = <1, 1, 0>$ makes the output to be 1.
Complexity Class P

- **Complexity class** $P$ contains those problems that can be **solved** in polynomial time in the **size of input**.
  - **Input size**: size of encoded “binary” strings.
  - Edmonds: Problems in P are considered **tractable**.

- The computer concerned is a **deterministic Turing machine**
  - **Deterministic** means that each step in a computation is predictable.
  - A **Turing machine** is a mathematical model of a universal computer (any computation that needs polynomial time on a Turing machine can also be performed in polynomial time on any other machine).

- MST is in $P$. 
Complexity Class NP

• Suppose that solution checking for some problem can be done in polynomial time on a deterministic machine ⇒ the problem can be solved in polynomial time on a nondeterministic Turing machine.
  — Nondeterministic: the machine makes a guess, e.g., the right one (or the machine evaluates all possibilities in parallel).

• The class NP (Nondeterministic Polynomial): class of problems that can be verified in polynomial time in the size of input.
  — NP: class of problems that can be solved in polynomial time on a nondeterministic machine.

• Is TSP ∈ NP?
  — Need to check a solution in polynomial time.
    ■ Guess a tour.
    ■ Check if the tour visits every city exactly once.
    ■ Check if the tour returns to the start.
    ■ Check if total distance ≤ B.
  — All can be done in \(O(n)\) time, so TSP ∈ NP.
NP-Completeness

• An issue which is still unsettled:
  \[ P \subset NP \text{ or } P = NP? \]

• There is a strong belief that \( P \neq NP \), due to the existence of \( NP \)-complete problems.

• The class **NP-complete (NPC):**
  - All problems in NPC have the same degree of difficulty: Any NPC problem can be solved in polynomial time \( \Rightarrow \) **all** problems in NP can be solved in polynomial time.

![Diagram showing the relationship between P, NP, and NPC with P being most likely case and if P ≠ NP being NPC with if P = NP being P = NP = NPC](image-url)
**Polynomial-time Reduction**

- **Motivation:** Let $L_1$ and $L_2$ be two decision problems. Suppose algorithm $A_2$ can solve $L_2$. Can we use $A_2$ to solve $L_1$?

- **Polynomial-time reduction $f$ from $L_1$ to $L_2$: $L_1 \leq_P L_2$**
  - $f$ reduces input for $L_1$ into an input for $L_2$ s.t. the reduced input is a “yes” input for $L_2$ iff the original input is a “yes” input for $L_1$.
    - $L_1 \leq_P L_2$ if $\exists$ polynomial-time computable function $f$: $\{0, 1\}^* \rightarrow \{0, 1\}^*$ s.t. $x \in L_1$ iff $f(x) \in L_2$, $\forall x \in \{0, 1\}^*$.
    - $L_2$ is at least as hard as $L_1$.

- $f$ is computable in polynomial time.
Significance of Reduction

• Significance of $L_1 \leq_P L_2$:
  - $\exists$ polynomial-time algorithm for $L_2 \Rightarrow \exists$ polynomial-time algorithm for $L_1$ ($L_2 \in P \Rightarrow L_1 \in P$).
  - $\forall$ polynomial-time algorithm for $L_1 \Rightarrow \forall$ polynomial-time algorithm for $L_2$ ($L_1 \notin P \Rightarrow L_2 \notin P$).

• $\leq_P$ is transitive, i.e., $L_1 \leq_P L_2$ and $L_2 \leq_P L_3 \Rightarrow L_1 \leq_P L_3$.
Polynomial-time Reduction

- The Hamiltonian Circuit Problem (HC)
  - **Instance:** an undirected graph \( G = (V, E) \).
  - **Question:** is there a cycle in \( G \) that includes every vertex exactly once?

- TSP (The Traveling Salesman Problem)

- How to show HC \( \leq_p \) TSP?
  1. Define a function \( f \) mapping any HC instance into a TSP instance, and show that \( f \) can be computed in polynomial time.
  2. Prove that \( G \) has an HC iff the reduced instance has a TSP tour with distance \( \leq B \) \( (x \in \text{HC} \iff f(x) \in \text{TSP}) \).
HC $\leq_p$ TSP: Step 1

1. Define a reduction function $f$ for HC $\leq_p$ TSP.
   - Given an arbitrary HC instance $G = (V, E)$ with $n$ vertices
     - Create a set of $n$ cities labeled with names in $V$.
     - Assign distance between $u$ and $v$
       \[
       d(u, v) = \begin{cases} 
       1, & \text{if } (u, v) \in E, \\
       2, & \text{if } (u, v) \notin E.
       \end{cases}
       \]
     - Set bound $B = n$.
   - $f$ can be computed in $O(V^2)$ time.
HC $\leq_p$ TSP: Step 2

2. $G$ has an HC iff the reduced instance has a TSP with distance $\leq B$.

- $x \in HC \Rightarrow f(x) \in TSP$.
  - Suppose the HC is $h = <v_1, v_2, \ldots, v_n, v_1>$. Then, $h$ is also a tour in the transformed TSP instance.
  - The distance of the tour $h$ is $n = B$ since there are $n$ consecutive edges in $E$, and so has distance 1 in $f(x)$.
  - Thus, $f(x) \in TSP$ ($f(x)$ has a TSP tour with distance $\leq B$).
2. $G$ has an HC iff the reduced instance has a TSP with distance $\leq B$.

- $f(x) \in \text{TSP} \Rightarrow x \in \text{HC}$.
  - Suppose there is a TSP tour with distance $\leq n = B$. Let it be $<v_1, v_2, \ldots, v_n, v_1>$.
  - Since distance of the tour $\leq n$ and there are $n$ edges in the TSP tour, the tour contains only edges in $E$.
  - Thus, $<v_1, v_2, \ldots, v_n, v_1>$ is a Hamiltonian cycle ($x \in \text{HC}$).
NP-Completeness and NP-Hardness

• **NP-completeness**: worst-case analyses for decision problems.

• **L is NP-complete if**
  - \( L \in \text{NP} \)
  - **NP-Hard**: \( L' \leq_p L \) for every \( L' \in \text{NP} \).

• **NP-hard**: If \( L \) satisfies the 2nd property, but not necessarily the 1st property, we say that \( L \) is **NP-hard**.

• Suppose \( L \in \text{NPC} \).
  - If \( L \in P \), then there exists a polynomial-time algorithm for every \( L' \in \text{NP} \) (i.e., \( P = \text{NP} \)).
  - If \( L \notin P \), then there exists no polynomial-time algorithm for any \( L' \in \text{NPC} \) (i.e., \( P \neq \text{NP} \)).
Proving NP-Completeness

• Five steps for proving that $L$ is NP-complete:
  1. Prove $L \in \text{NP}$.
  2. Select a known NP-complete problem $L'$.
  3. Construct a reduction $f$ transforming every instance of $L'$ to an instance of $L$.
  4. Prove that $x \in L'$ iff $f(x) \in L$ for all $x \in \{0, 1\}^*$.
  5. Prove that $f$ is a polynomial-time transformation.

• We have shown that TSP is NP-complete.
Coping with NP-hard problems

- **Approximation algorithms**
  - Guarantee to be a fixed percentage away from the optimum.
  - E.g., MST for the minimum Steiner tree problem.

- **Pseudo-polynomial time algorithms**
  - Has the form of a polynomial function for the complexity, but is not to the problem size.
  - E.g., $O(nW)$ for the 0-1 knapsack problem.

- **Restriction**
  - Work on some subset of the original problem.
  - E.g., the longest path problem in directed acyclic graphs.

- **Exhaustive search/Branch and bound**
  - Is feasible only when the problem size is small.

- **Local search:**
  - Simulated annealing (hill climbing), genetic algorithms, etc.

- **Heuristics:** No guarantee of performance.
Spanning Tree v.s. Steiner Tree

- **Manhattan distance:** If two points (nodes) are located at coordinates \((x_1, y_1)\) and \((x_2, y_2)\), the Manhattan distance between them is given by \(d_{12} = |x_1 - x_2| + |y_1 - y_2|\).

- **Rectilinear spanning tree:** a spanning tree that connects its nodes using Manhattan paths (Fig. (b) below).

- **Steiner tree:** a tree that connects its nodes, and additional points (Steiner points) are permitted to used for the connections.

- The minimum rectilinear spanning tree problem is in P, while the minimum rectilinear Steiner tree (Fig. (c)) problem is NP-complete.
  - The spanning tree algorithm can be an approximation for the Steiner tree problem (at most 50% away from the optimum).
Exhaustive Search v.s. Branch and Bound

- TSP example
Algorithmic Paradigms

- **Exhaustive search:** Search the entire solution space.
- **Branch and bound:** A search technique with pruning.
- **Greedy method:** Pick a locally optimal solution at each step.
- **Dynamic programming:** Partition a problem into a collection of sub-problems, the sub-problems are solved, and then the original problem is solved by combining the solutions. (Applicable when the sub-problems are **NOT independent**).
- **Hierarchical approach:** Divide-and-conquer.
- **Mathematical programming:** A system of solving an objective function under constraints.
- **Simulated annealing:** An adaptive, iterative, non-deterministic algorithm that allows “uphill” moves to escape from local optima.
- **Tabu search:** Similar to simulated annealing, but does not decrease the chance of “uphill” moves throughout the search.
- **Genetic algorithm:** A population of solutions is stored and allowed to evolve through successive generations via mutation, crossover, etc.
Dynamic Programming (DP) v.s. Divide-and-Conquer

- Both solve problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms
  - Partition a problem into **independent** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
  - Inefficient if they solve the same subproblem more than once.
- Dynamic programming (DP)
  - Applicable when the subproblems are **not independent**.
  - DP solves each subproblem just once.
Example: Bin Packing

- **The Bin-Packing Problem** \( \Pi \): Items \( U = \{u_1, u_2, \ldots, u_n\} \), where \( u_i \) is of an integer size \( s_i \); set \( B \) of bins, each with capacity \( b \).
- **Goal**: Pack all items, minimizing # of bins used. (**NP-hard!**)
Algorithms for Bin Packing

- Greedy approximation alg.: First-Fit Decreasing (FFD)
  \[ FFD(\Pi) \leq \frac{11 \text{OPT}(\Pi)}{9} + 4 \]


- Mathematical Programming: Use **integer linear programming (ILP)** to find a solution using \(|B|\) bins, then search for the smallest feasible \(|B|\).
ILP Formulation for Bin Packing

- 0-1 variable: $x_{ij}=1$ if item $u_i$ is placed in bin $b_j$, 0 otherwise.

$$
\begin{align*}
\text{max} & \quad \sum_{(i,j) \in E} w_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{i \in U} w_{ij} x_{ij} \leq b_j, \forall j \in B \quad / \{ \text{capacity constraint} \} \quad (1) \\
& \quad \sum_{j \in B} x_{ij} = 1, \forall i \in U \quad / \{ \text{assignment constraint} \} \quad (2) \\
& \quad \sum_{ij} x_{ij} = n \quad / \{ \text{completeness constraint} \} \quad (3) \\
& \quad x_{ij} \in \{0,1\} \quad / \{ \text{0-1 constraint} \} \quad (4)
\end{align*}
$$

- **Step 1:** Set $|B|$ to the lower bound of the # of bins.
- **Step 2:** Use the ILP to find a feasible solution.
- **Step 3:** If the solution exists, the # of bins required is $|B|$. Then exit.
- **Step 4:** Otherwise, set $|B| \leftarrow |B| + 1$. Goto Step 2.
CAD Related Conferences/Journals

- Important Conferences:
  - ACM/IEEE Design Automation Conference (DAC)
  - IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)
  - IEEE Int'l Test Conference (ITC)
  - ACM Int'l Symposium on Physical Design (ISPD)
  - ACM/IEEE Asia and South Pacific Design Automation Conf. (ASP-DAC)
  - ACM/IEEE Design, Automation, and Test in Europe (DATE)
  - IEEE Int'l Conference on Computer Design (ICCD)
  - IEEE Custom Integrated Circuits Conference (CICC)
  - IEEE Int'l Symposium on Circuits and Systems (ISCAS)
  - Others: VLSI Design/CAD Symposium/Taiwan

- Important Journals:
  - IEEE Transactions on Computer-Aided Design (TCAD)
  - ACM Transactions on Design Automation of Electronic Systems (TODAES)
  - IEEE Transactions on VLSI Systems (TVLSI)
  - IEEE Transactions on Computers (TC)
  - IEE Proceedings – Circuits, Devices and Systems
  - IEE Proceedings – Digital Systems
  - INTEGRATION: The VLSI Journal