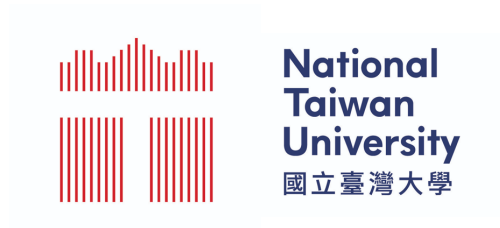


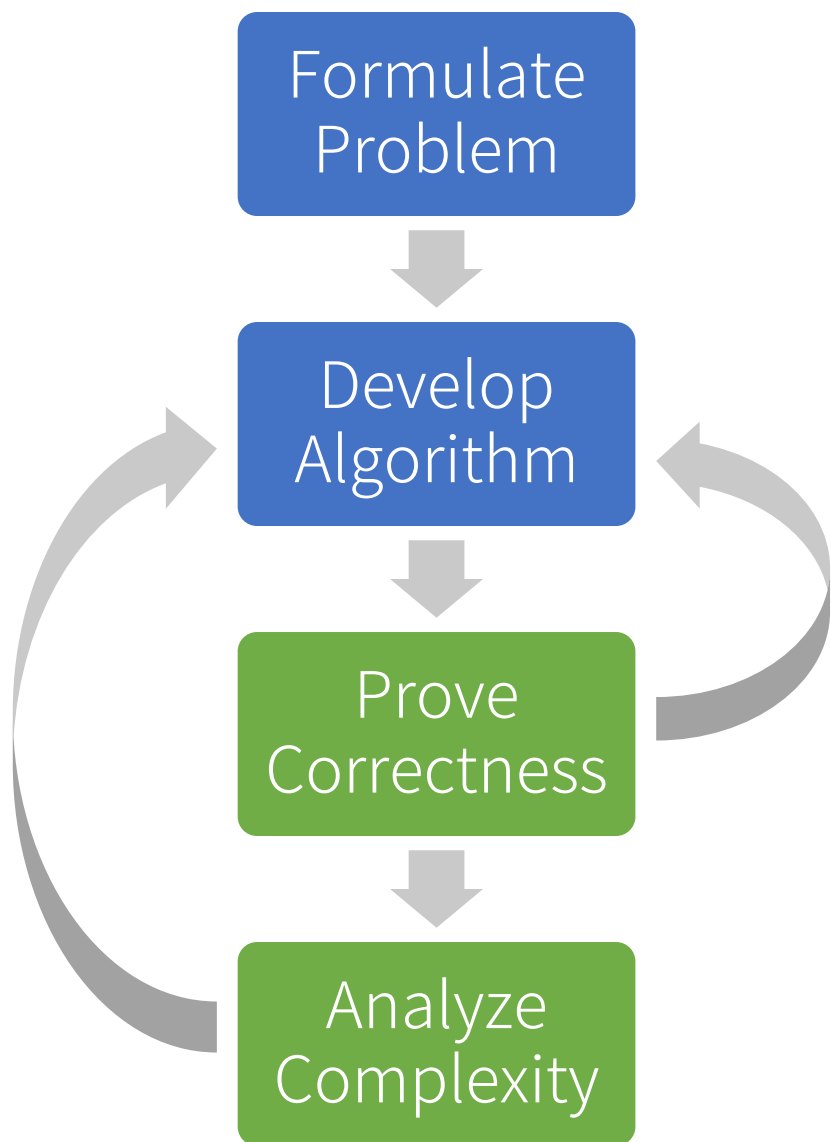
CSIE 2136 Algorithm Design and Analysis, Fall 2022



Review

Hsu-Chun Hsiao

演算法設計與分析的流程



Computational Problem

排序 (sorting) 問題：要怎麼把 n 個數字由小排到大？

Input: an unsorted array (E.g., 8, 5, 7, 2, 3, 9)

Output: a sorted array (E.g., 2, 3, 5, 7, 8, 9)

Algorithm

E.g., 插入排序法 (Insertion Sort)

Correctness

程序會停止嗎？

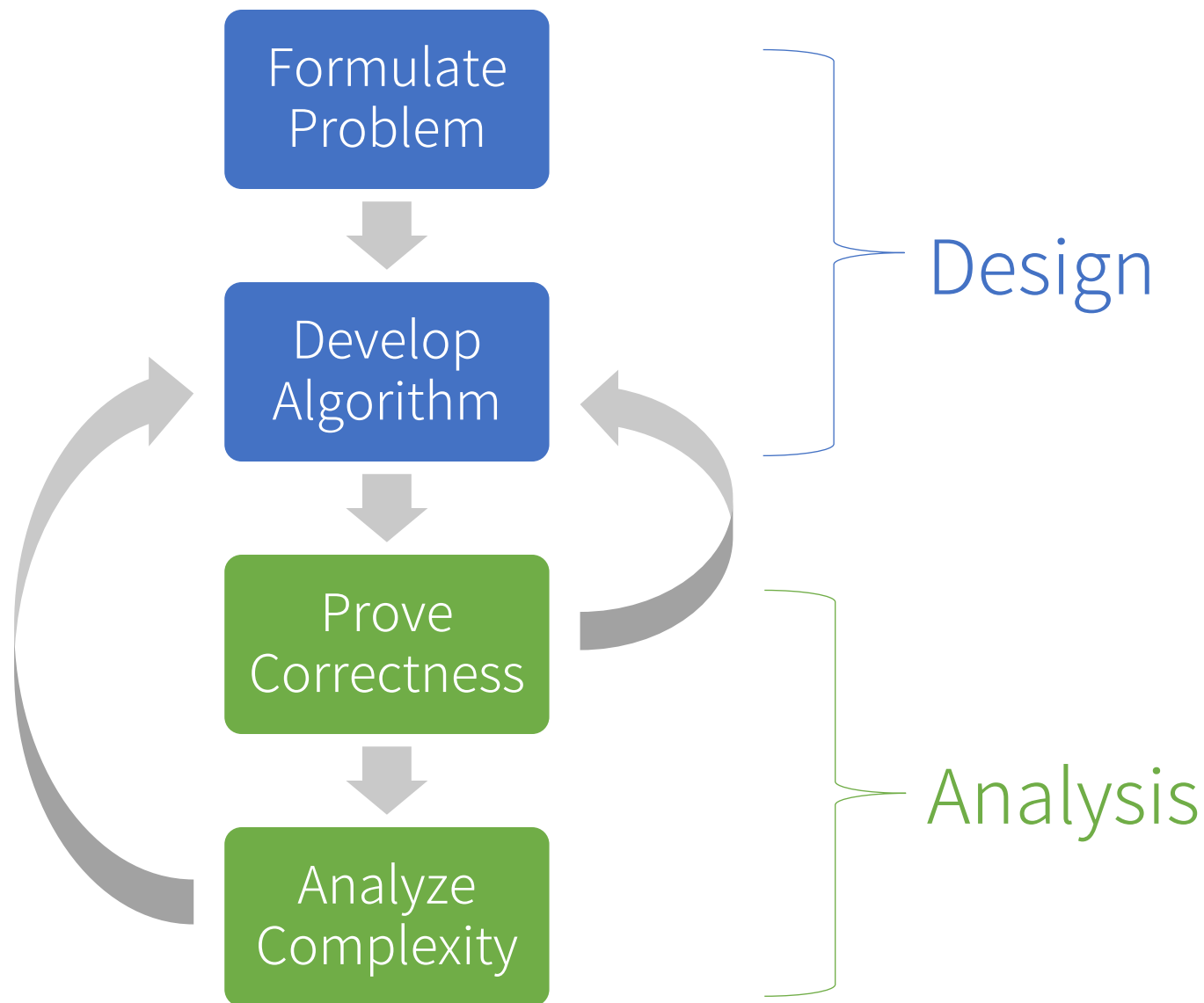
對每個 input 都能找出正確的結果嗎？

Efficiency

執行時間是多久？

要使用多少儲存空間？

演算法設計與分析的技巧💡



- Brute-force search
- Divide and conquer
- Dynamic programming
- Greedy algorithms
- Graph algorithms
- Reduction

- Induction
- Proof by contradiction
- Proof by contraposition
- Exchange argument
- Asymptotic analysis
- Amortized analysis

暴力搜尋法 Brute-force search

- Systematically enumerate & check all possible solutions
- Inefficient when the search space is large, even infinite

Q: When might we use brute-force search?

- When problem size is small (e.g., the base case in divide-and-conquer)
- When there are problem-specific heuristics to reduce the search space

分治法 Divide and Conquer

基本精神：Divide -> Conquer -> Combine

Base case:

當問題足夠小時，
直接解決

Recursive case:

1. Divide
分割

把大問題切割成較小的同樣問題



2. Conquer
各個擊破

遞迴呼叫自己解決小問題



3. Combine
合擊

有時候需整合小問題的答案
以重建大問題的答案

分治法 Divide and Conquer

True/False: The following D&C algorithm correctly finds an MST:

Divide – Given a graph G , partition G into two parts by using a cut.

Conquer – Find an MST for each part

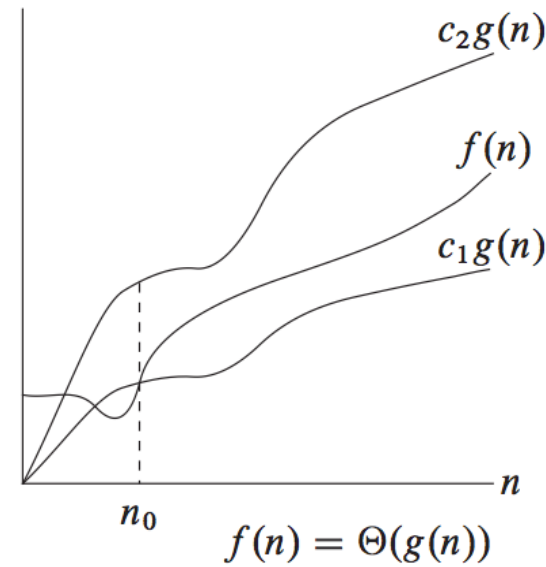
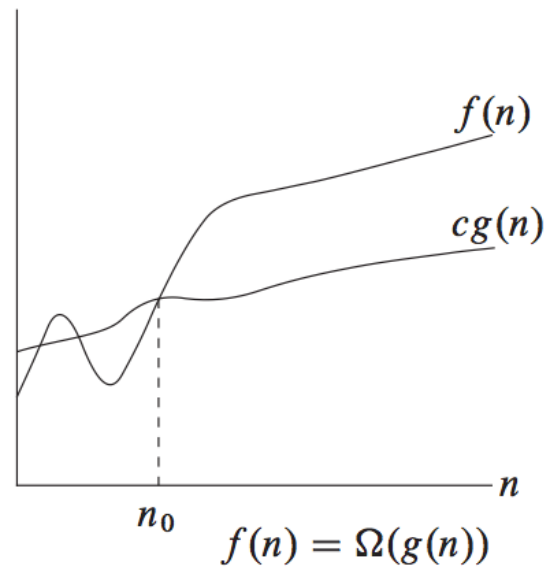
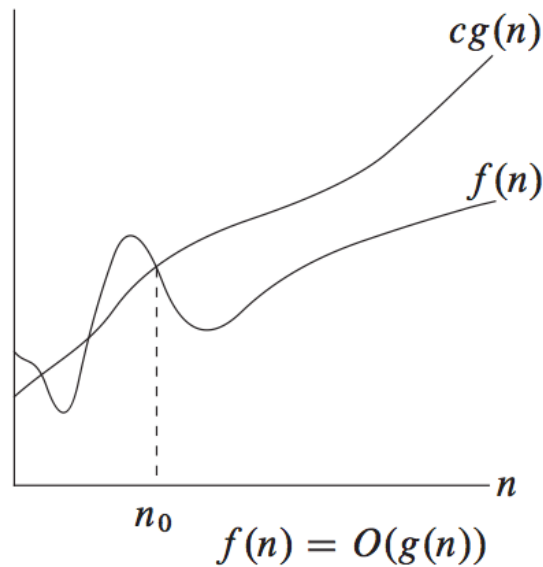
Combine – Combine the two parts using a minimum edge of the cut

False. The MST might require more than one edge in the cut!

Consider a graph of 3 vertices and $w(A, B) = w(A, C) = 1, w(B, C) = 2$, and a cut $\{A\}, \{B, C\}$.

漸進分析 Asymptotic analysis

- $f(n)$ = time or space of an algorithm for an input of size n
- Asymptotic analysis: focus on the **growth** of $f(n)$ as $n \rightarrow \infty$



True/False: If $f(n) = O(g(n))$, then $g(n) = \Omega(f(n))$

Divide and Conquer (DC)

Dynamic Programming (DP)

把一個問題分解成數個性質相同的小問題。
解答可以用小問題的解答遞迴表示。

- Work best when subproblems are **independent, or disjoint**
- Blindly recompute overlapping subproblems
- When subproblems are **dependent, or overlapping**
- Two equivalent ways to avoid recomputation
 - Top-down with memoization
 - Bottom-up method

DP and optimization problems

- To apply DP, an optimization problem must exhibit two key properties:
 - **Overlapping subproblems** - solutions to same subproblems are used repeatedly
 - **Optimal substructure** – an optimal solution can be constructed from optimal solutions to subproblems

Q: Why these two properties are required?

Without overlapping subproblems, DP saves no time.

Without optimal substructure, we need to consider non-optimal solutions to a subproblem, and thus hardly reduce the search space.

Pseudo-polynomial time

- Running time of DP-based knapsack algorithm is $\Theta(nW)$
 - n = # of objects
 - W = knapsack's capacity, W is a non-negative integers
- Running time is **pseudo-polynomial**, not polynomial, in input size
 - Pseudo-polynomial time: “if its running time is **polynomial in the numeric value of the input**, but is **exponential in the length of the input** – the number of bits required to represent it.”
- The size of the representation of W is $\lg W$
 - $\Theta(nW) = \Theta(n2^k)$, where $k = \lg W$

Dynamic programming: 4 steps

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. Construct an optimal solution from computed information

①

②

③

④

True/False: DP solves each subproblem at most once.

True/False: The running time of a DP algorithm is $O(\# \text{ of subproblems})$.

Dynamic programming

Greedy algorithms

Both require optimal substructure

- Make an informed choice **after** getting optimal solutions to subproblems
- Overlapping subproblems

- Make a greedy choice **before** solving the resulting subproblem
- No overlapping subproblem
 - Each round selects only one subproblem
 - Sizes of subproblems decrease

貪心演算法 Greedy algorithms

- Try to solve optimization problems by **greedy** choices
- Always make a choice that looks best at the moment
- Make a **locally optimal choice** in hope of getting a **globally optimal solution**
- Many greedy algorithms to same problem
 - Easy to invent one
 - Do not always yield optimal solutions
 - Hard to find one that actually works and prove its optimality

貪心演算法 Greedy algorithms

To yield an optimal solution, the problem should exhibit

1. Greedy-choice property

- Making locally optimal (greedy) choices leads to a globally optimal solution

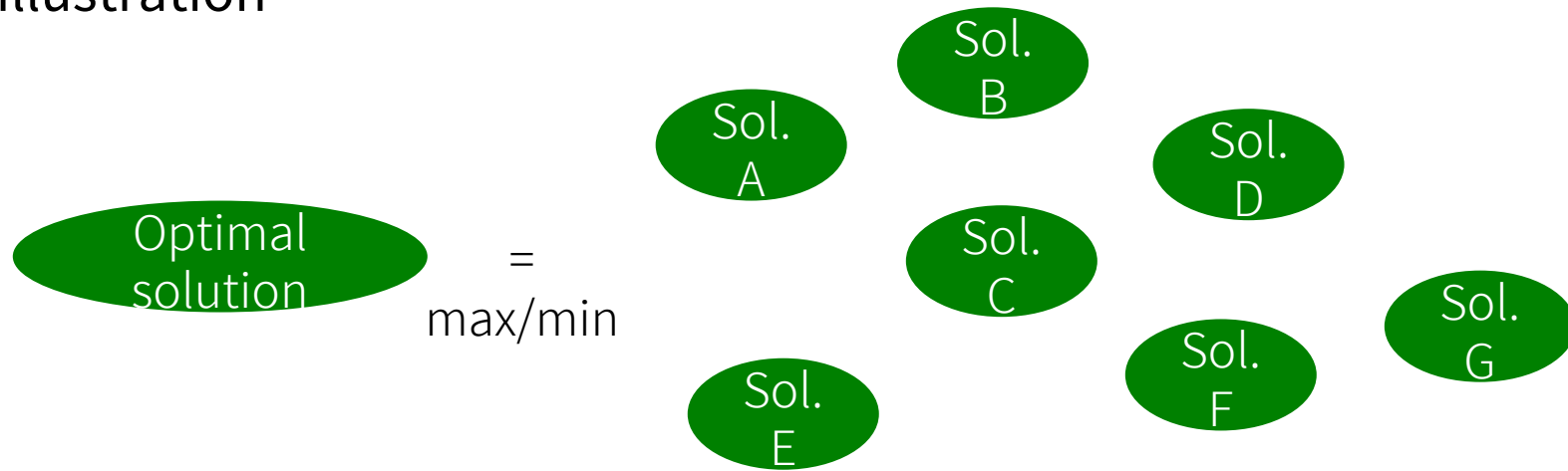
2. Optimal substructure

- An optimal solution to the problem contains within it optimal solutions to subproblems

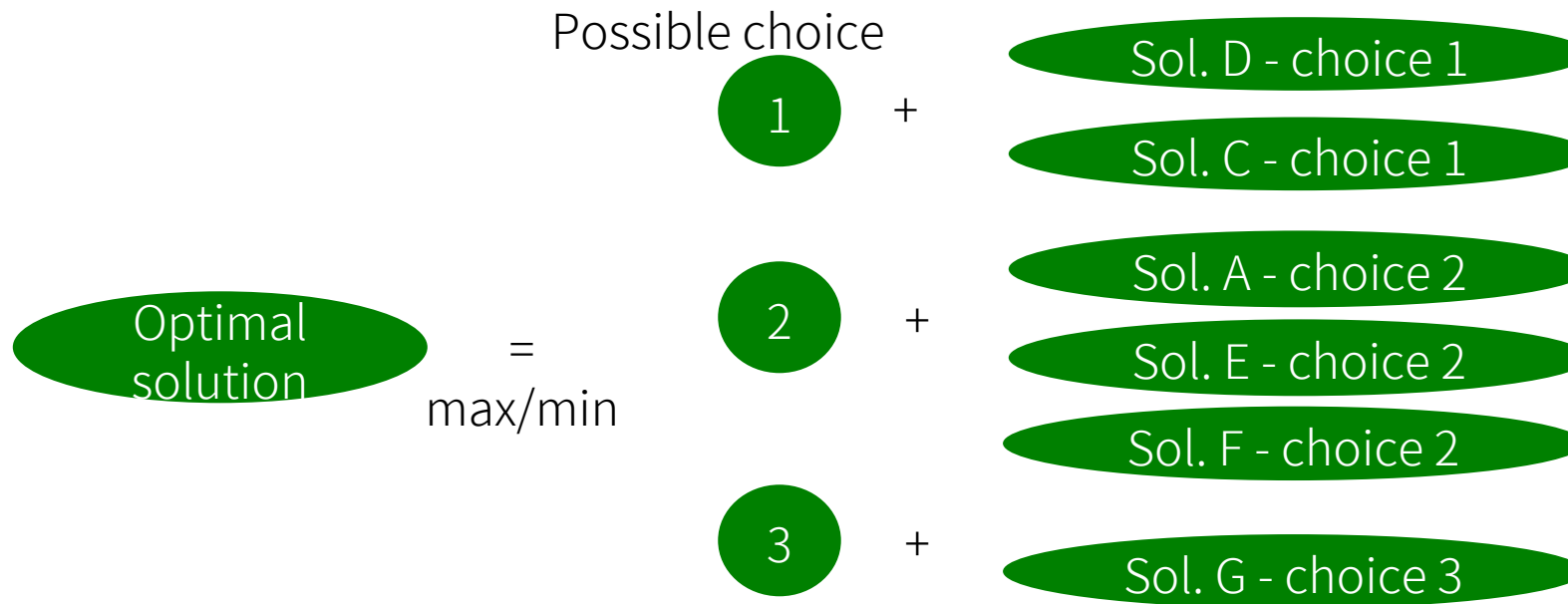
True/False: Kruskal's algorithm and Prim's algorithm are greedy algorithms.

True/False: Greedily selecting a vertex covering the most (uncovered) edges can yield a 2-approximate vertex cover algorithm.

Brute-force illustration

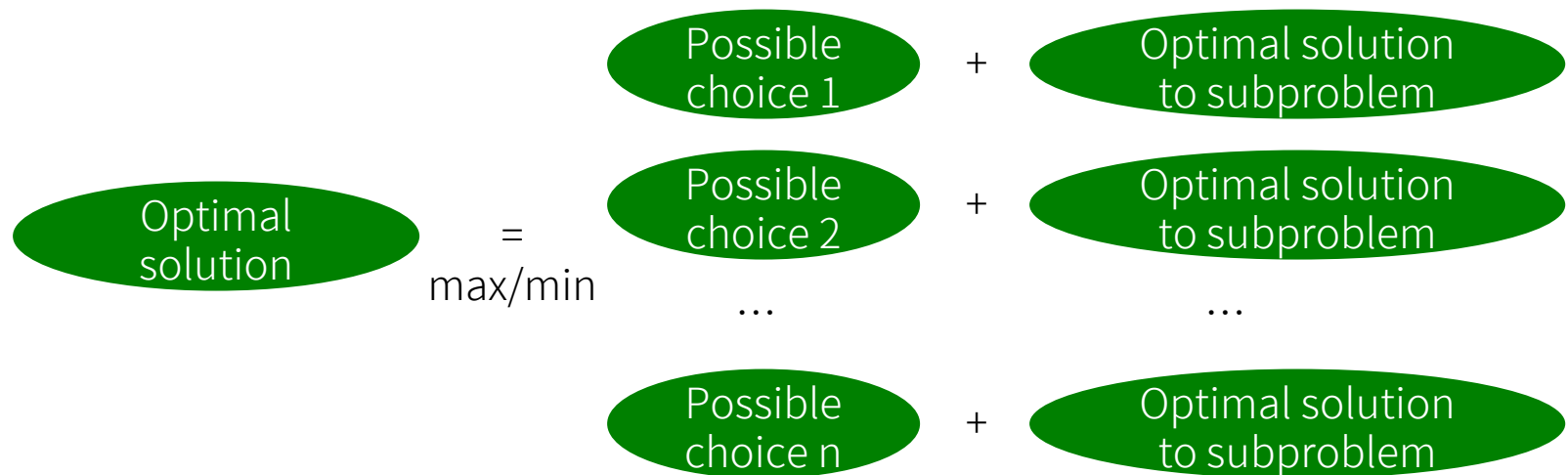


Same brute-force illustration (grouping solutions by exclusive choices)



DP illustration

Optimal substructure property ensures that we only need to consider an optimal solution to each subproblem



對每個可能的情況，只要考慮相對應的subproblem的一個 optimal解就可以了。
其他的solution都不用考慮了！

Greedy illustration

Greedy-choice property ensures that we only need to consider one greedy choice (among all possible choices)



非greedy choice的狀況
都不用考慮了！

Graph algorithms

- **Graph basics**
 - The origin of graph theory
 - Graph terminology [B.4, B.5]
 - Real-world applications
 - Graph representations [Ch. 22.1]
- **Graph traversal**
 - Breadth-first search (BFS) [Ch. 22.2]
 - Depth-first search (DFS) [Ch. 22.3]
- **DFS applications**
 - Topological sort [Ch. 22.4]
 - Strongly-connected components [Ch. 22.5]
- **Minimum spanning trees** [Ch. 23]
 - Kruskal's algorithm
 - Prim's algorithm
- **Single-source shortest paths** [Ch. 24]
 - Dijkstra algorithm
 - Bellman-Ford algorithm
 - SSSP in DAG
- ~~◦ **All-pairs shortest paths** [Ch. 25]~~
 - ~~◦ Floyd-Warshall algorithm~~
 - ~~◦ Johnson's algorithm~~

* Out of scope: You're not expected to know these terms, but you may be asked to derive/reason about them based on your knowledge and the provided information.

Graph traversal (or graph search)

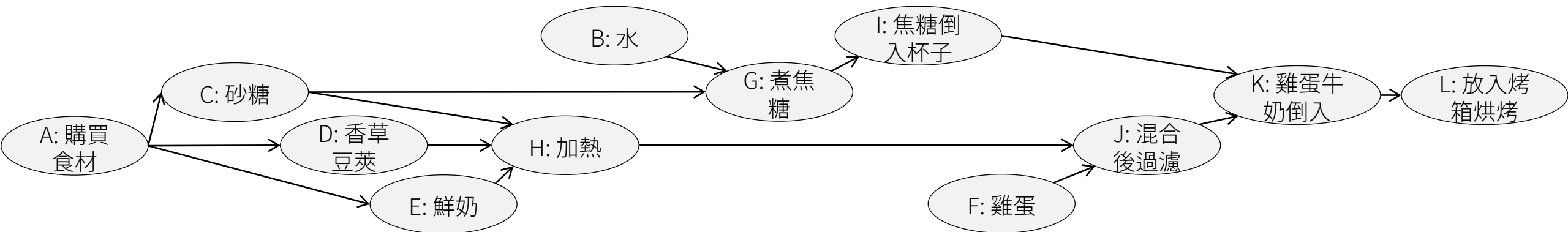
- From a given source vertex s , systematically find all reachable vertices
- Useful to discover the **structure** of a graph
- Standard graph-search algorithms
 - **Breadth-first Search** (BFS, 廣度優先搜尋)
 - **Depth-first Search** (DFS, 深度優先搜尋)

True/False: A connected undirected graph has at least $|V| - 1$ edges.

Q: Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

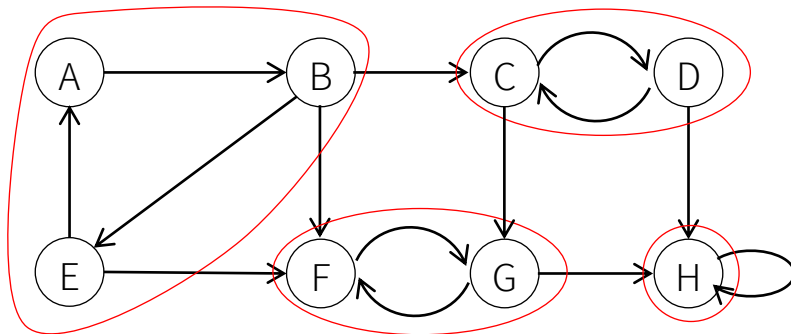
Topological Sort

- Input: a **directed acyclic graph (DAG)** $G = (V, E)$
 - Often indicates precedence among events (**X must happen before Y**)
- Output: a linear ordering of all its vertices such that for all edges (u, v) in E , u precedes v in the ordering
- Alternative view: a vertex ordering along a horizontal line so that **all directed edges go from left to right**



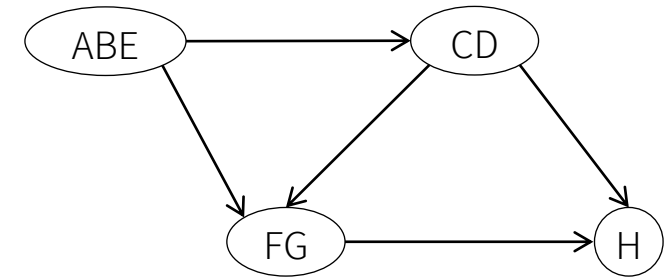
Decomposing a directed graph

- A directed graph is a **DAG of its SCC**



$G = (V, E)$

Contract each SCC
into one vertex



Component graph $G^{scc} = (V^{scc}, E^{scc})$

Finding SCC: the Kosaraju-Sharir algorithm

Strongly-Connected-Components(G)

```
1  call  $DFS(G)$  to compute finishing times  $u.f$  for each vertex  $u$ 
2  compute  $G^T$ 
3  call  $DFS(G^T)$ , but in the main loop of DFS, consider the vertices in order of
   decreasing  $u.f$  (as computed in line 1)
4  output the vertices of each tree in the DFS forest formed in line 3 as a
   separate strongly connected component
```

- Input: a directed graph $G = (V, E)$
- Output: strongly connected components
- Time complexity
 - 2 DFS executions
 - $\Theta(V + E)$ using adjacency lists

True/False: The number of SCCs in a graph always decreases after a new edge is added.

Minimum spanning tree (MST)

- Finding an MST is an **optimization** problem
- Two **greedy** algorithms compute an MST:
 - **Kruskal's algorithm**: consider edges in **ascending order of weight**. At each step, select the next edge as long as it does not create cycle.
 - **Prim's algorithm**: start with any vertex s and **greedily grow a tree from s** . At each step, add the edge of the least weight to connect an isolated vertex.

True/False: MST is unique if all edge weights are distinct.

True/False: Kruskal's (or Prim's) algorithm may output an incorrect result if there exist negative edges.

True/False: Finding a maximum spanning tree is NP-hard.

True/False: Suppose in a graph G , each edge weight value appears at most twice. Then, there are at most two minimum spanning trees in G .

Single-source shortest-path algorithms

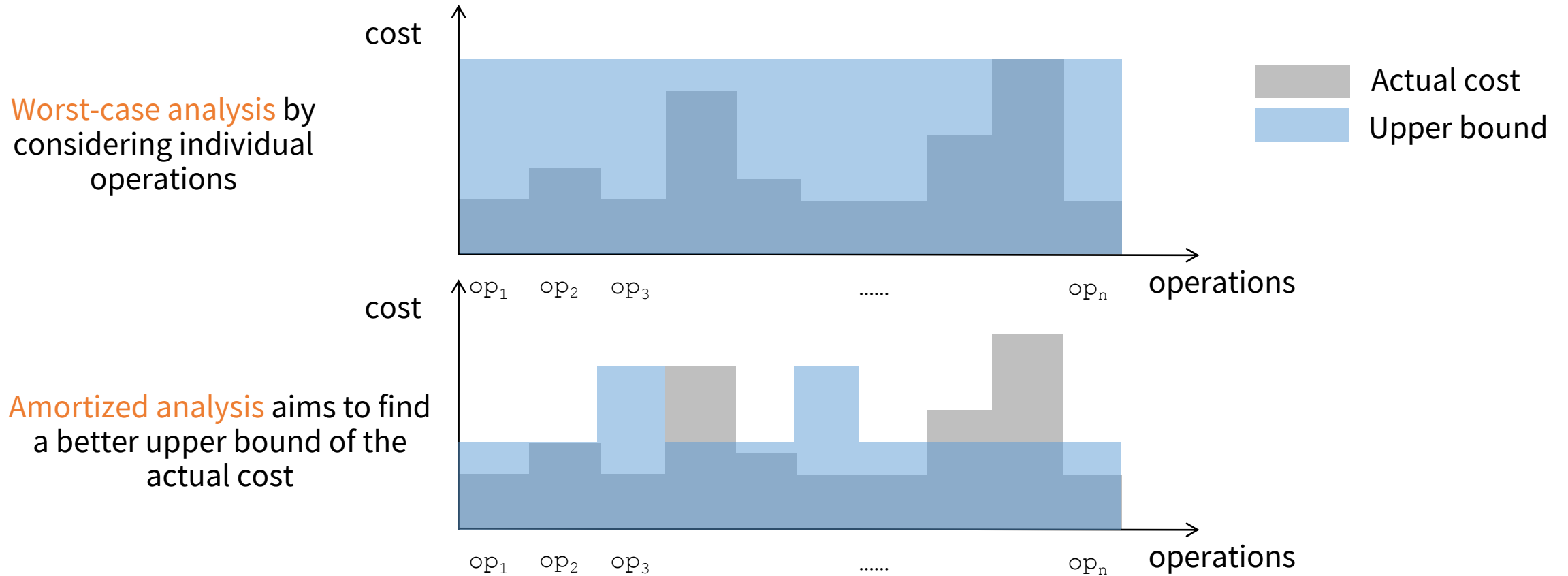
- Given a graph $G = (V, E)$ and a **source** vertex s in V , find the minimum cost paths from s to every vertex in V
- **Dijkstra algorithm**
 - Greedy
 - Requiring that all edge weights are **nonnegative**
- **Bellman-Ford algorithm**
 - Dynamic programming
 - General case, edge weights **may be negative**
- Both on a weighted, directed graph

True/False: Dijkstra algorithm may not terminate if there exist negative cycles.

True/False: Dijkstra algorithm may output incorrect results when the graph has negative-weight edges.

Goal of amortized analysis

- Obtain an **asymptotic worst-case bound** for a **sequence of n operations**



True/False: If every operation of a data structure has an amortized cost $O(1)$, then the cost of any sequence of i th to $(i + n - 1)$ -th operations is bounded by $O(n)$.

Amortized analysis: 3 common techniques

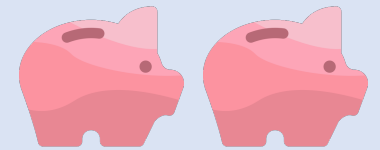
Aggregate method (聚集方法)

- Determine an upper bound on the cost over any sequence of n operations, $T(n)$
- The average cost per operation is then $T(n)/n$
- All operations have the same amortized cost



Accounting method (記帳方法)

- Each operation is assigned an amortized cost (may differ from the actual cost)
- Each object of the data structure is associated with a credit
- Need to ensure that every object has sufficient credit at any time



Potential method (位能方法)

- Similar to accounting method; each operation is assigned an amortized cost
- The data structure as a whole maintains a credit (i.e., potential)
- Need to ensure that the potential level is nonnegative at any time



Note: these are **for analysis purpose only**, not for implementation!

NPC & Approximation

- NP-Completeness Overview
 - Warm up: graph coloring
 - Complexity classes
 - Decision problems
 - Reduction
 - Appendix: P-time solving vs. verification
- Proving NP-Completeness
 - Formula satisfiability
 - 3-CNF satisfiability
 - Clique
 - Vertex cover
 - Independent set
 - Traveling salesman
 - Hamiltonian cycle

- What is approximation?
- Vertex cover
 - Approximate vertex cover
 - Approximate weighted vertex cover
- Traveling salesman problem
 - Proving NP-completeness
 - Approximation & inapproximability
- Randomized approximation
 - 3-CNF-SAT
 - MAX-CUT

Complexity classes

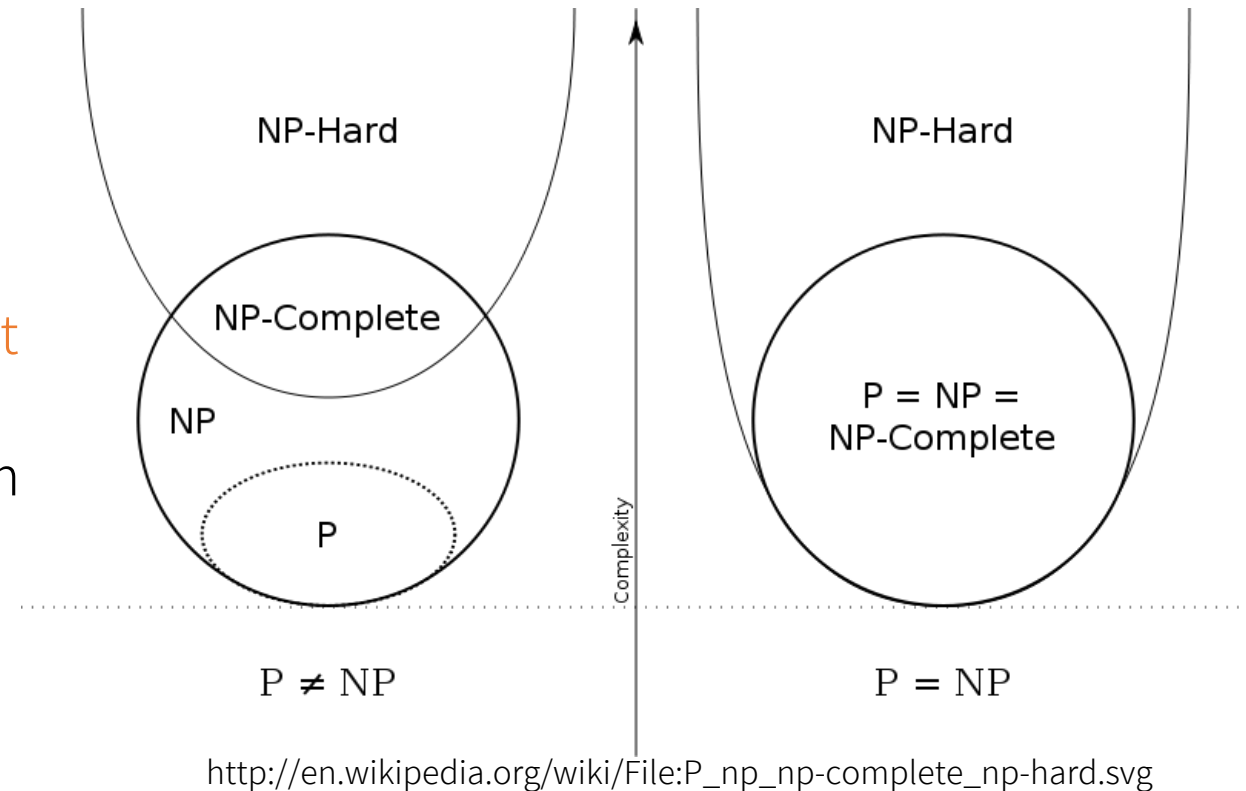
Class P: class of problems that can be solved in $O(n^k)$

Class NP: class of problems that can be verified in $O(n^k)$

Class NP-hard: a class of problems that are "at least as hard as the hardest problems" in NP

Class NP-complete (NPC): class of problems in both NP and NP-hard

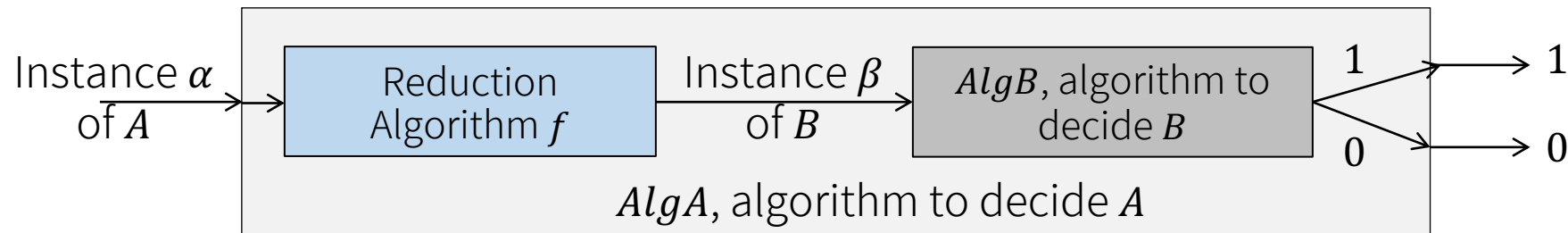
Hardness relationship can be determined via [polynomial-time reduction](#)



True/False: If $P = NP$, every NP-hard problem can be solved in polynomial time.

Reduction

- A **reduction** f is an algorithm for **transforming every instance** of a problem A into an instance of another problem B , and, for all α , $AlgA(\alpha) = 1$ **if and only if** $AlgB(f(\alpha)) = 1$
 - Thus, we can use $AlgB$ to construct $AlgA$ for solving problem A
- A **polynomial-time reduction** ($A \leq_p B$) is a **polynomial-time algorithm** for transforming every instance of a problem A into an instance of another problem B
 - Can help determine the hardness relationship between problems (within a polynomial-time factor)
 - $A \leq_p B$ implies **A is no harder than B** ; equivalently, **B is at least as hard as A**



True/False: If $A \leq_p B$ and there is an $O(n^3)$ algorithm for B , then there is an $O(n^3)$ algorithm for A .

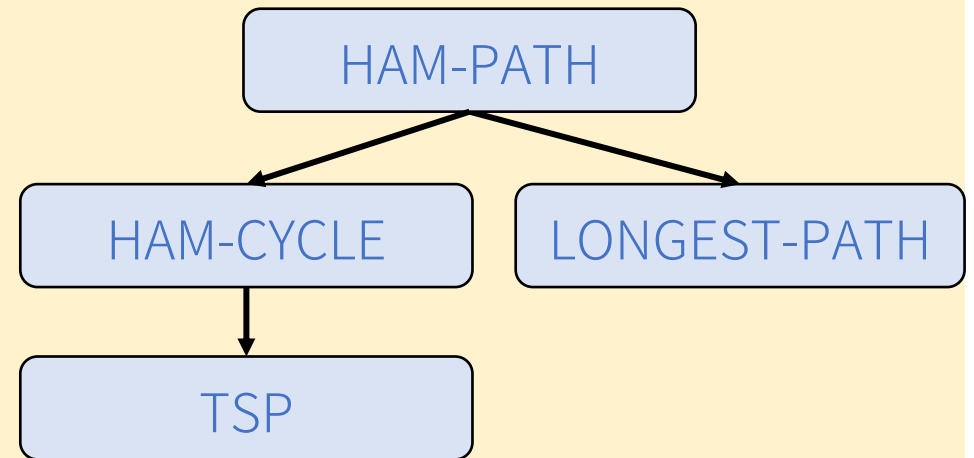
True/False: If $A \leq_p B$ and there is a polynomial-time algorithm for B , then there is a polynomial-time algorithm for A .

Construct a polynomial-time reduction from HAM-PATH to LONGEST-PATH

Hint: What is the length of a Hamiltonian path, if exists?

Construct a polynomial-time reduction from HAM-PATH to HAM-CYCLE

Hint: add a vertex that links to all vertices

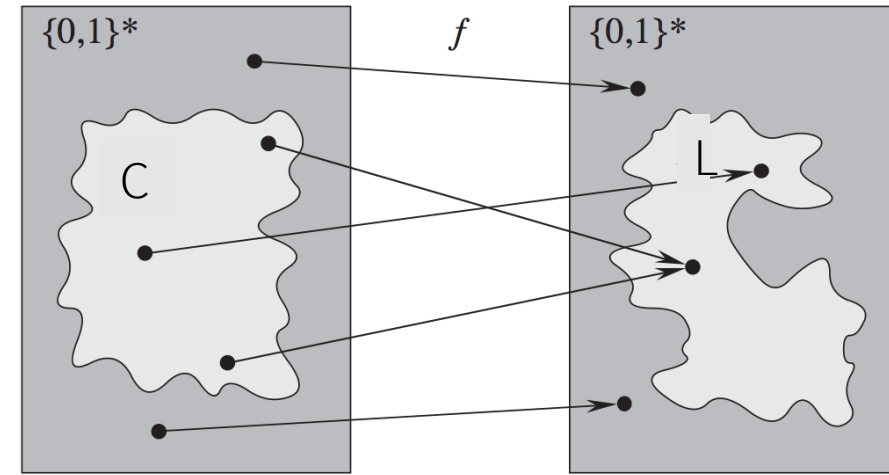


Proving NP-Completeness

$L \in \text{NP-Complete} \Leftrightarrow L \in \text{NP} \text{ and } L \in \text{NP-hard}$

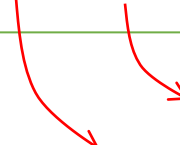
Step-by-step approach for proving L in NPC:

1. Prove $L \in \text{NP}$
2. Prove $L \in \text{NP-hard}$
 - ① Select a **known NPC problem C**
 - ② **Construct a reduction f** transforming every instance of C to an instance of L
 - ③ Prove that x in C **if and only if** $f(x)$ in L for all x in $\{0,1\}^*$
 - ④ Prove that f is a **polynomial-time transformation**



Short answer: Suppose we know problem X is NP-complete, and we want to show that problem Y is NP-hard. Should we reduce X to Y or reduce Y to X ?

Approximation algorithms

- $\rho(n)$ -approximation algorithm
 - **Efficient**: guaranteed to run in polynomial time
 - **General**: guaranteed to solve every instance of the problem
 - **Near-optimal**: guaranteed to find solution within a factor of $\rho(n)$ of the cost of an optimal solution
 - Approximation ratio $\rho(n)$
 - n : input size
 - C^* : cost of an optimal solution
 - C : cost of the solution produced by the approximation algorithm
- $$\max \left(\frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n)$$
- 

Maximization problem: $\frac{C^*}{C} \leq \rho(n)$

Minimization problem: $\frac{C}{C^*} \leq \rho(n)$

True/False: If $A \leq_p B$, and there is a 2-approximation algorithm for B , then there is a 2-approximation algorithm for A .

Where can we go from here?

Applying what you've learned or studying advanced techniques

- 生物序列分析演算法、量子演算法、財務演算法、基因遺傳演算法、演算法的數學解析、高等演算法、最佳化演算法…

83449	財金所	Fin7051		財務演算法
65425	電機系	EE4033		演算法
86508	資工系	CSIE1212	01	資料結構與演算法
22516	資工系	CSIE1212	02	資料結構與演算法
23661	生醫電資所	EE5048		演算法
23661	電子所	EE5048		演算法
23661	電機所	EE5048		演算法
81953	智慧醫療學程	EE5145		基因遺傳演算法
81953	電機所	EE5145		基因遺傳演算法
24677	資工所	CSIE7008		演算法的數學解析
24059	網媒所	CSIE7134		財務演算法
24059	資工所	CSIE7134		財務演算法
23566	資工所	CSIE5410		最佳化演算法
23072	生資國際學程	TIGBPB8009		高等演算法
23072	生資國際學程	TIGBPB8009		高等演算法

52932	生物機電系	BME5010		資料結構與演算法實務
52932	生物機電所	BME5010		資料結構與演算法實務
52932	大數據學程	BME5010		資料結構與演算法實務
13706	生物機電所	BME5938		生物資訊演算法
13706	生物機電系	BME5938		生物資訊演算法
57411	資管系	IM2009		演算法
77283	電機系	EE4033	01	演算法
29083	電機系	EE4033	02	演算法
15840	電機系	EE4033	03	演算法
39705	資工系	CSIE2136	01	演算法設計與分析
37972	資工系	CSIE2136	02	演算法設計與分析
48321	電子所	EE5048		演算法 停開
48321	電機所	EE5048		演算法 停開
84071	生醫電資所	CSIE5028		生物序列分析演算法
84071	資工所	CSIE5028		生物序列分析演算法
87361	資工所	CSIE5132		量子演算法
87361	網媒所	CSIE5132		量子演算法
98269		TB10310605	01	計算機演算法導論
96958		TB10420122	01	通用啟發式演算法

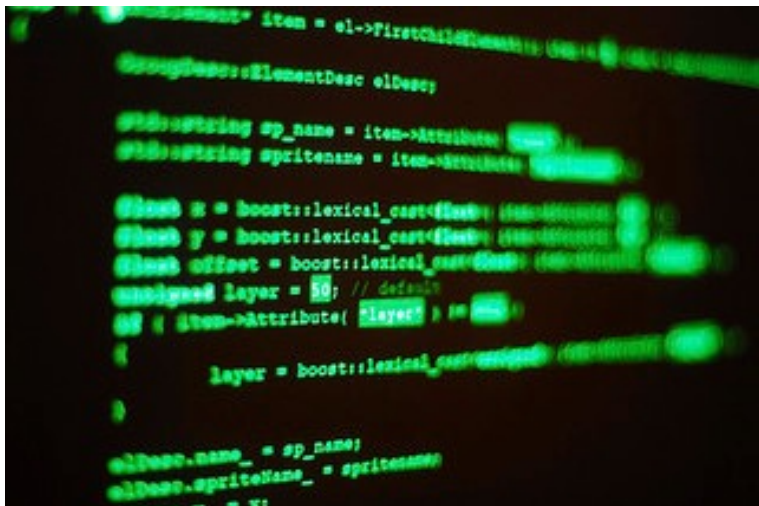
Algorithms in the real (cybersecurity) world



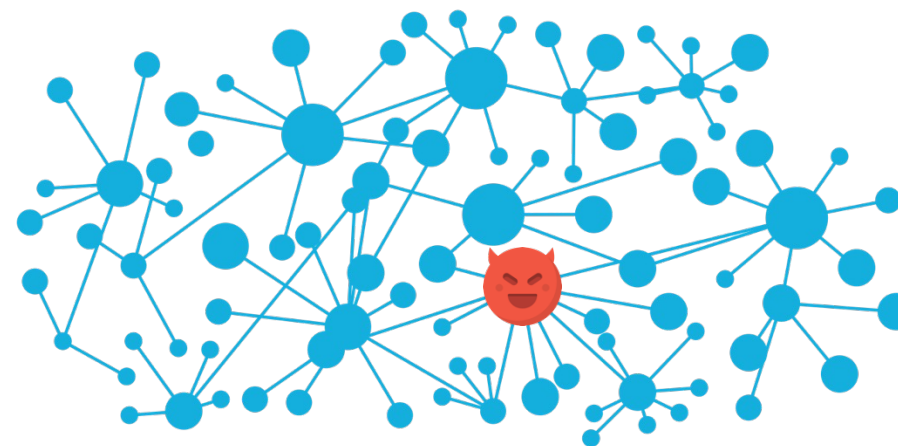
Public-key cryptography & how to break it



Detecting attack flows in real-time with very limited memory



Improving performance of automated bug finding



Detecting malicious switches by sending a minimum number of probe packets

Course objective: This course will provide you with intellectual tools for **designing** and **analyzing** algorithms, so that you know how to solve your own computational problems in the future.

You are now equipped with **powerful tools** for **solving big, important problems**.

