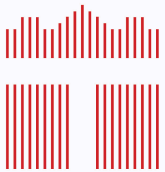
Slido: #ADA2020

CSIE 2136 Algorithm Design and Analysis, Fall 2020



National Taiwan University
國立臺灣大學

Review

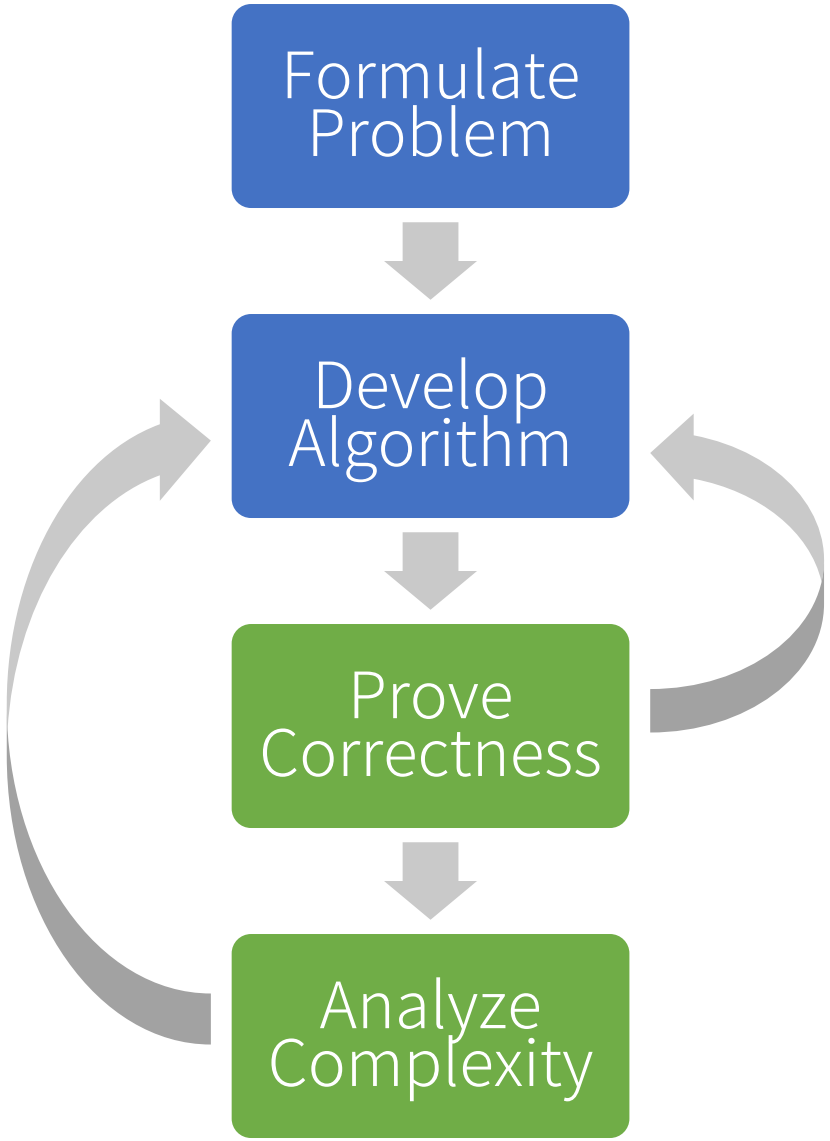
Hsu-Chun Hsiao

Closed-book final exam: 1/14

- Scope
  - Topics before midterm: roughly 1/3
  - Topics after the midterm: roughly 2/3
  - Easy: ~60%, medium: ~30%, hard: ~10%

- Exam questions may have different difficulty levels, move on if you get stuck!

演算法設計與分析的流程 



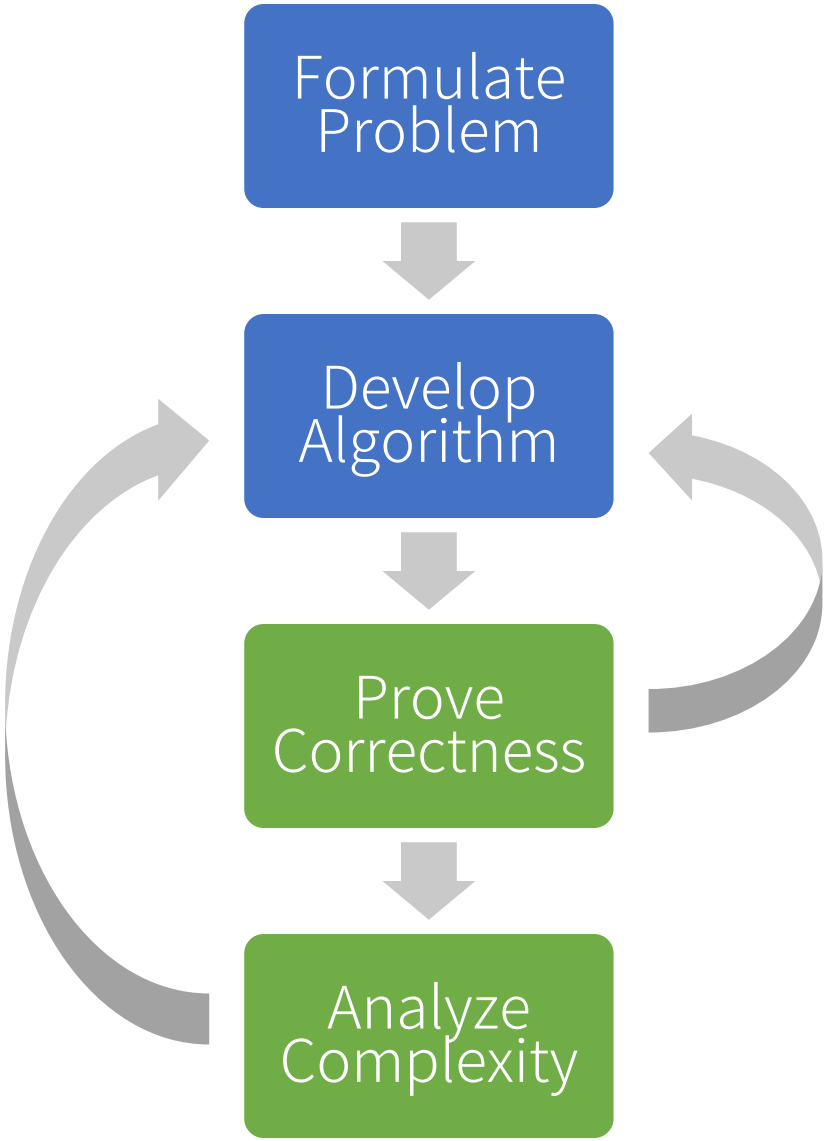
Computational Problem
排序 (sorting) 問題：要怎麼把 n 個數字由小排到大？
Input: an unsorted array (E.g., 8, 5, 7, 2, 3, 9)
Output: a sorted array (E.g., 2, 3, 5, 7, 8, 9)

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



Design

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

Analysis

- 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?

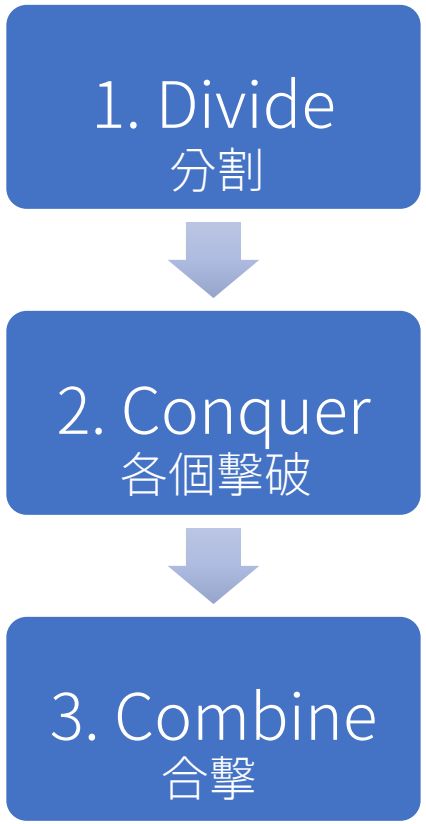
- When problem size is small (e.g., the base case in divide-and-conquer)
- When there are problem-specific heuristics to reduce the search space

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:

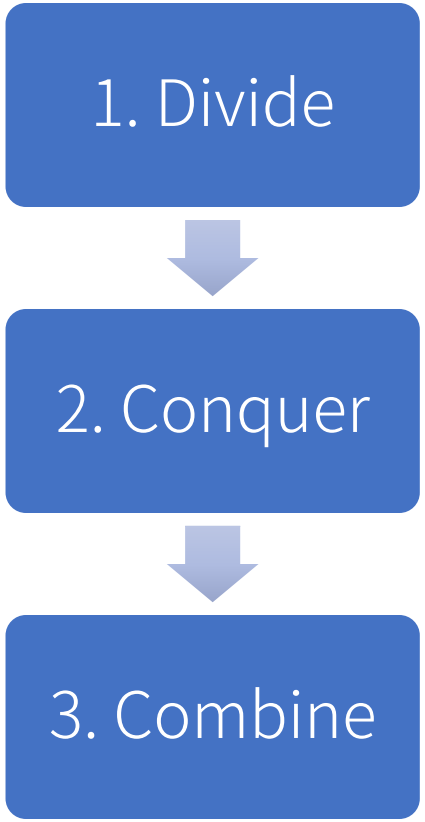


把大問題切割成較小的同樣問題

遞迴呼叫自己解決小問題

有時候需整合小問題的答案
以重建大問題的答案

Divide and Conquer 思路



- 是否適合用 D&C?
  - 當 input 很小時，可以直接解決小問題嗎？
  - 若有小問題的解，可以組合成大問題的解嗎？
  - Overall 的時間複雜度是否有比 naïve 方法好？
- 如果上述任一答案為否：
  - 試著稍作修正或增加額外資訊
  - 試著重新切割
  - 放棄使用 D&C

分治法 Divide and Conquer

Q: 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 $\{A\}, \{B,C\}$.

D&C 演算法之時間複雜度

$T(n)$ = running time for input size n

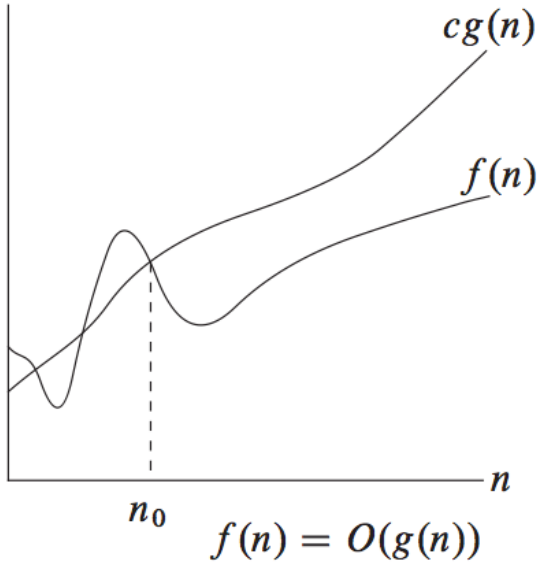
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \text{ ,} \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

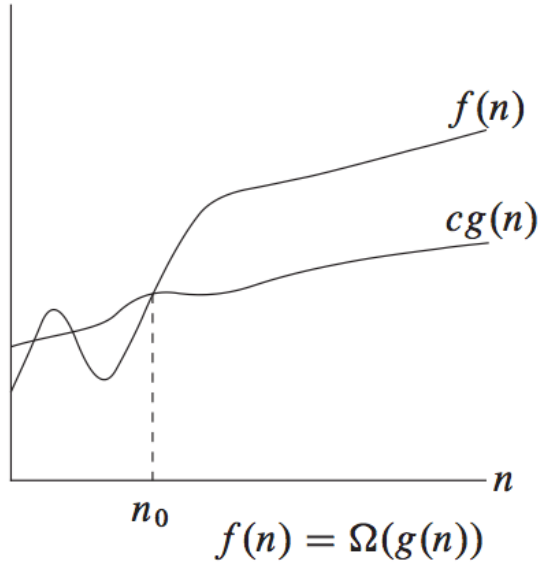
Q: What do a, b, $D(n)$, and $C(n)$ stand for?

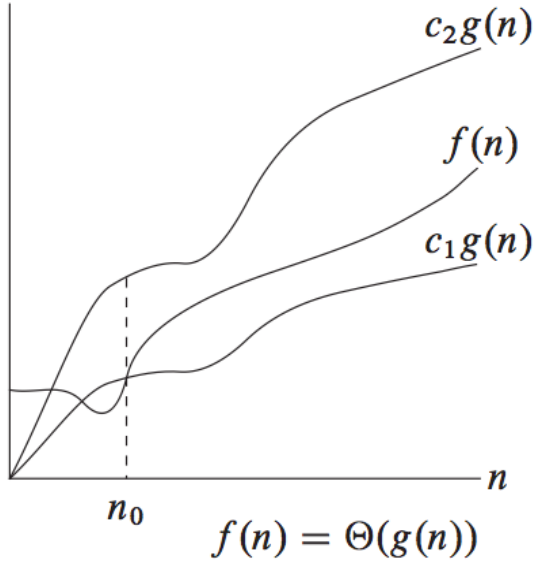
- $D(n)$ = time of Divide for input size n
- $C(n)$ = time of Combine for input size n
- a = # of subproblems
- n/b = size of each subproblem

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 $n \to \infty$







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

遞迴式 Recurrence relations

- How to solve recurrence relations?
- Substitution method (取代法)
  - Make a guess and then prove by induction
- Recursion-tree method (遞迴樹法)
  - Expand the recurrence into a tree and sum up the cost
- Master method (套公式大法)
  - Apply Master Theorem to a specific form of recurrences

Q: True/False: The master method can solve every recurrence.

Master method

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

只有這種形式才能套master theorem

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

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

- Dynamic programming are often applied to solving optimization problems (最佳化問題)
  - 從問題的多個解之中，選出最佳的
  - 最佳的解可能有很多個，找出一個就好了
- Examples of optimization problems
  - 從兩個字串中，找出最長的共同子字串
  - 給一個背包和一堆物品，找出背包最多能裝多少物品
  - …

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 $k = \lg W$

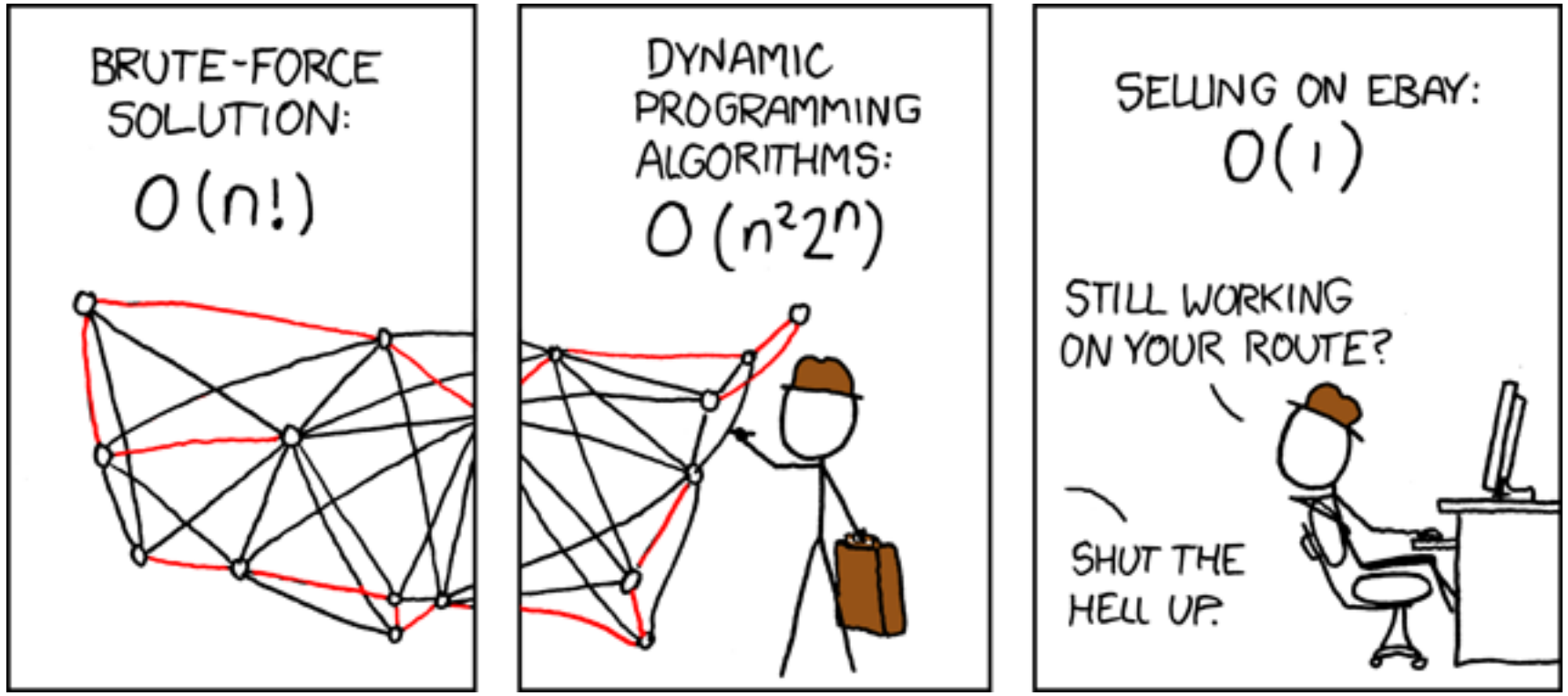
Dynamic programming: 4 steps

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

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

Q: **True/False**: The running time of a DP algorithm is O(# of subproblems).

Practice: DP for TSP



https://xkcd.com/399/

Dynamic programming

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

Greedy algorithms

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

Both require optimal substructure

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

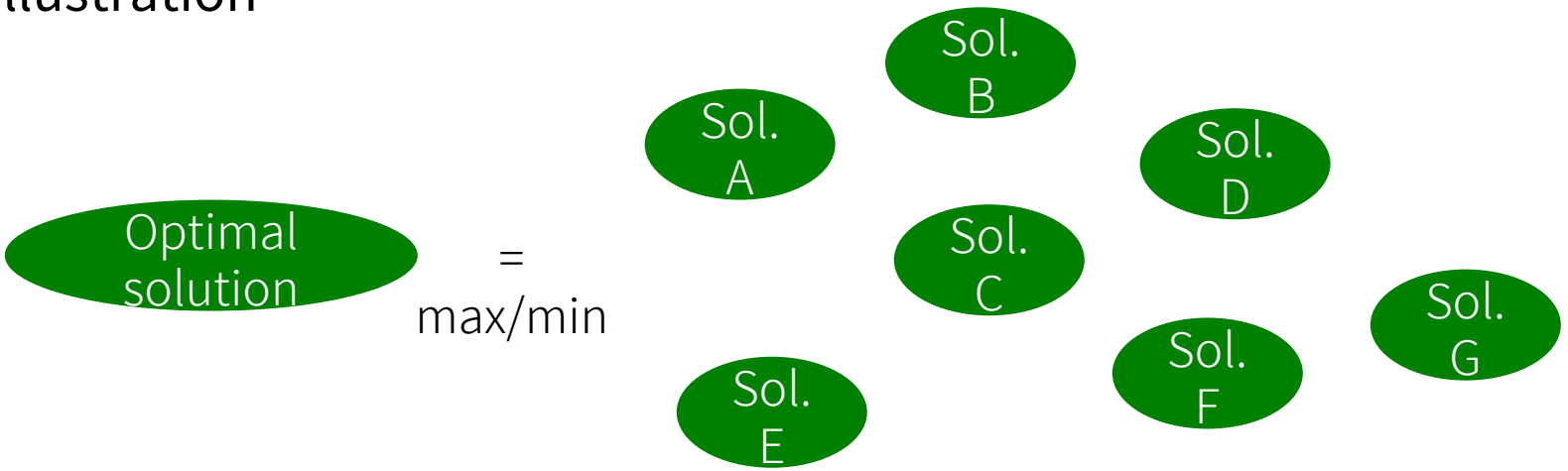
To yield an optimal solution, the problem should exhibit

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

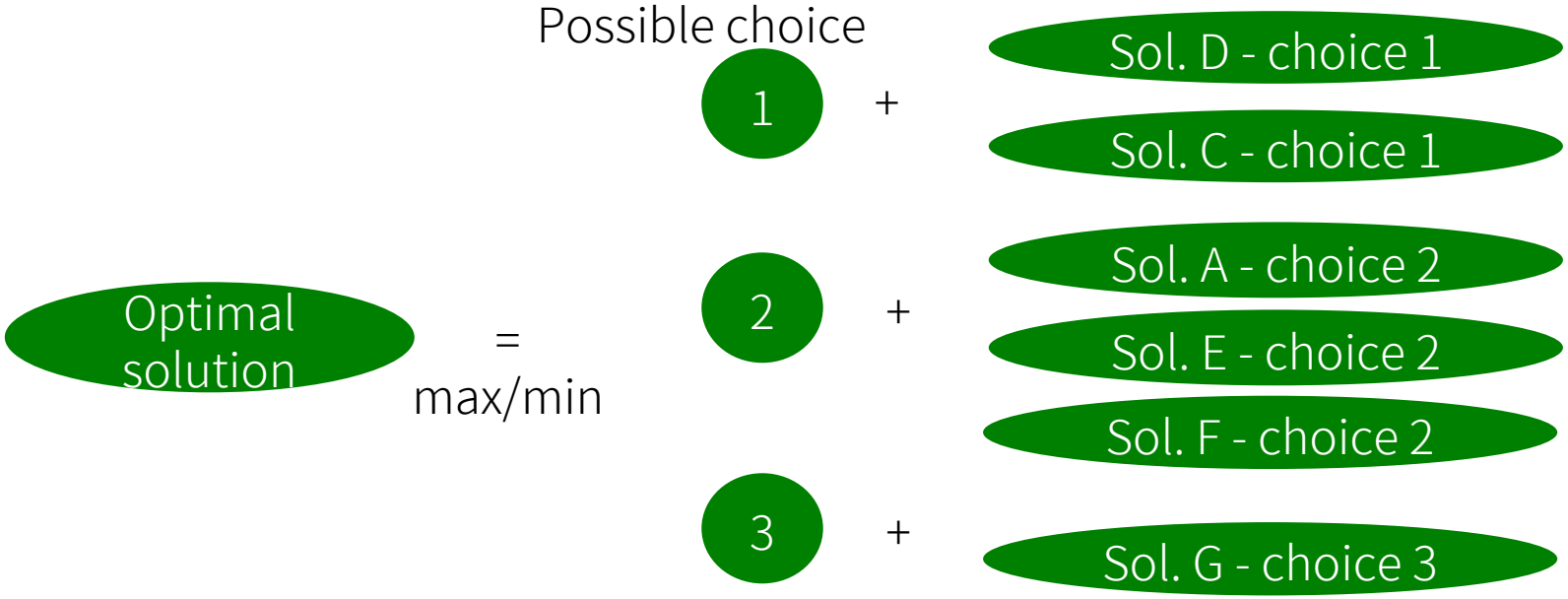
Q: **True/False**: Kruskal's algorithm and Prim's algorithm are greedy algorithms.

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

Brute-force illustration

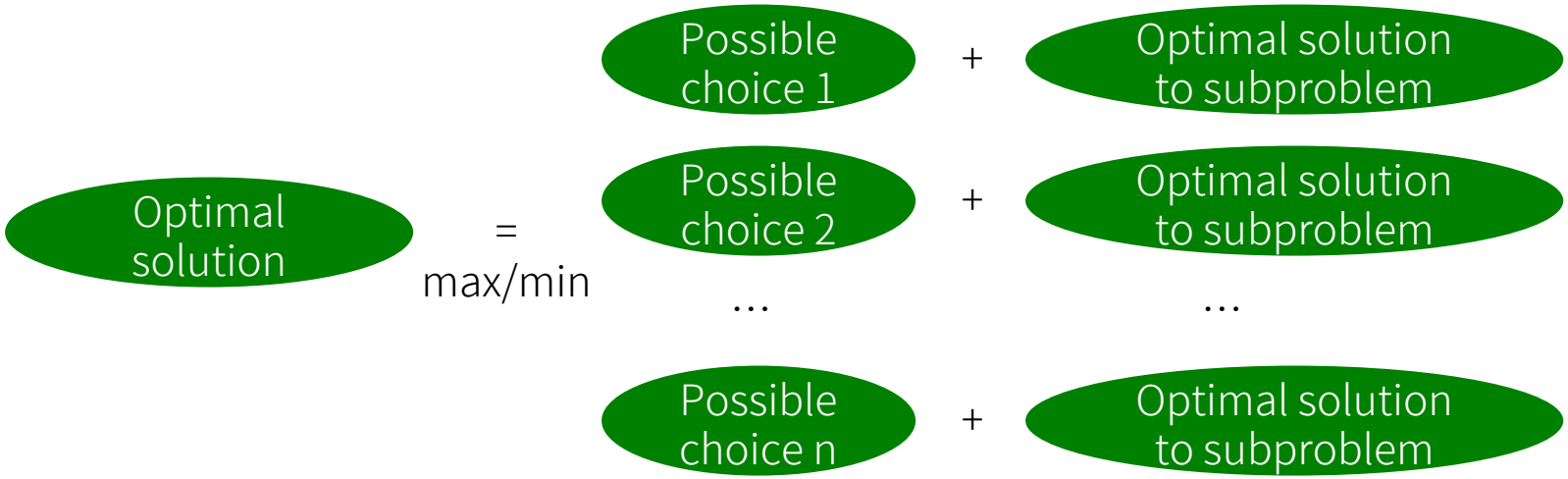


Same brute-force illustration (grouping solutions by exclusive choices)



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
  - 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 searching)

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

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 $|E|$.

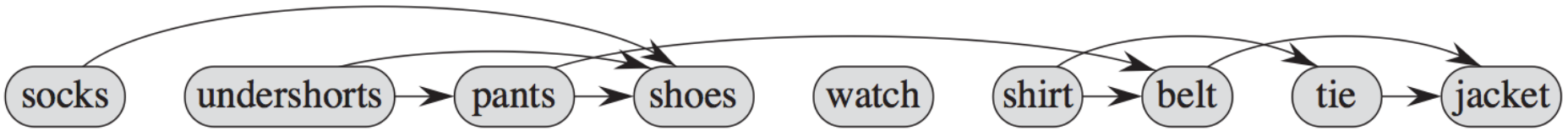
Minimum spanning tree (MST)

- 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: Kruskal's algorithm and Prim's algorithm always output the same MST.
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.

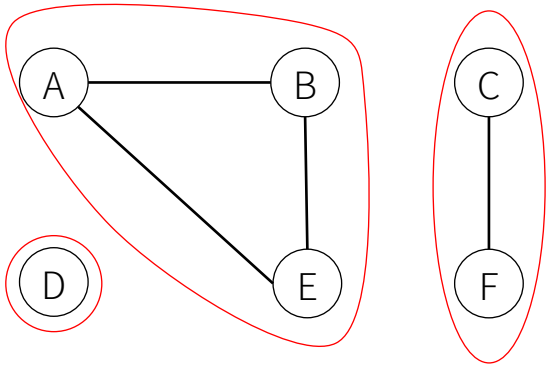
Topological Sort

- **Input**: a DAG $G = (V, E)$
- **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
- A DAG can have multiple valid topological orders
  - E.g., watch can be placed anywhere in the following example



Connected components of an **undirected graph**

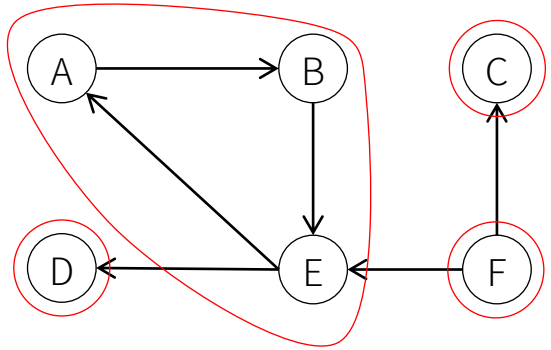
The connected components of an undirected graph are the equivalence classes of vertices under the "is reachable from" relation.



3 connected components: {A,B,E}, {C,F}, {D}

Strongly connected components of a **directed graph**

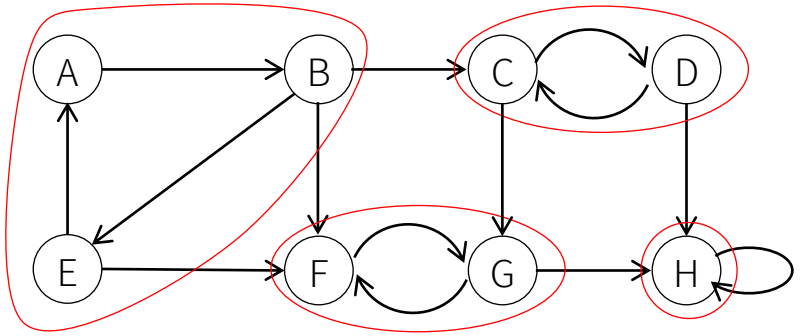
The strongly connected components of a directed graph are the equivalence classes of vertices under the "mutually reachable" relation.
That is, a strong component is a maximal subset of mutually reachable nodes.



4 strongly connected components: {A,B,E}, {C}, {D}, {F}

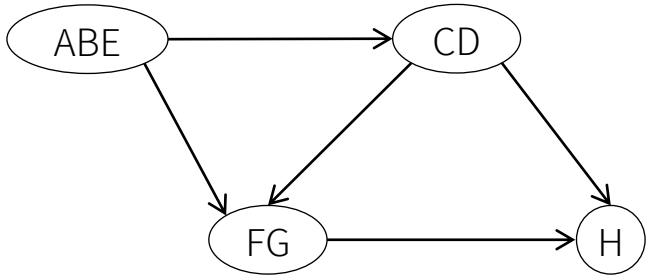
Decomposing a directed graph

- A direct graph is a DAG of its strongly connected components



$G = (V, E)$

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
```

- 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.

Single-source shortest-path algorithms

- 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 work incorrectly with negative-weight edges.
True/False: Given a graph with positive edge weights, the Bellman-Ford algorithm and Dijkstra algorithm may produce different shortest-path trees.

Priority-first search

- Maintain a set of explored vertices S
- Grow S by exploring highest-priority edges with exactly one endpoint leaving S

Q: What's the priority in each variant (BFS, DFS, Prim and Dijkstra)?

BFS: edges from vertex discovered least recently

DFS: edges from vertex discovered most recently

Prim: edges of minimum weight

Dijkstra: edges to vertex closest to *s*

Types of running-time analysis

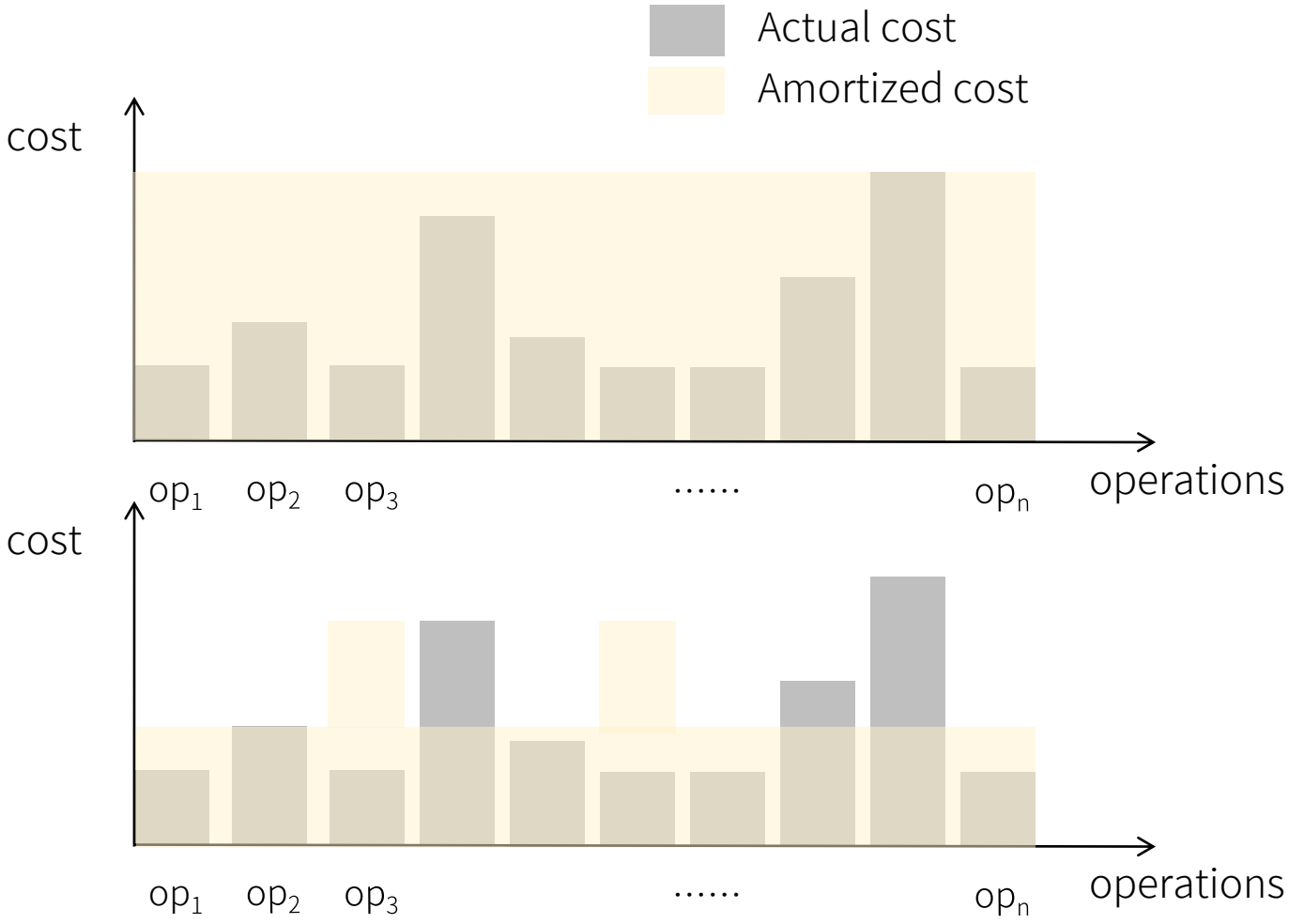
Worst case	Running time guarantee for any input of size n
Average case	Expected running time for a random input of size n
Probabilistic	Expected running time of a randomized algorithm
Amortized	Worst-case running time for a sequence of n operations

worst-case analysis considers the worst-case of one op

The the yellow area is clearly an upper bound the grey area

Amortized analysis aims to find a better upper bound

Need to explain why the yellow area will always ≥ the grey area for any sequence of n ops



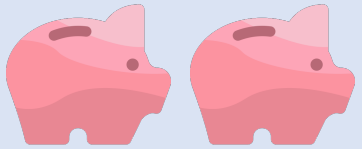
Amortized analysis: 3 common techniques

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: four color problem
  - Decision vs. optimization
  - Complexity classes
  - Reduction
  - P-time solving vs. verification
- Proving NP-Completeness
  - Formula satisfiability problem
  - 3-CNF-SAT
  - The clique problem
  - The vertex-cover problem
  - The independent-set problem
  - Traveling salesman problem
  - Hamiltonian cycle

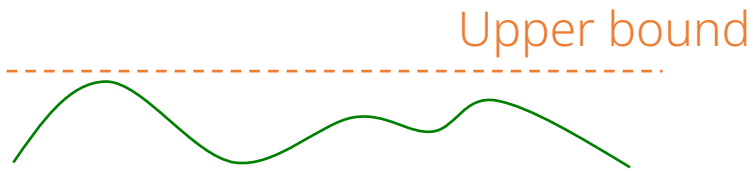
- Approximation algorithms
  - Vertex Cover
  - TSP
  - 3-CNF-SAT
- ~~Randomized algorithms~~
  - ~~Karger's min-cut algorithm~~
  - ~~Probabilistic data structures~~

* 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.

Computational complexity theory

Algorithm design

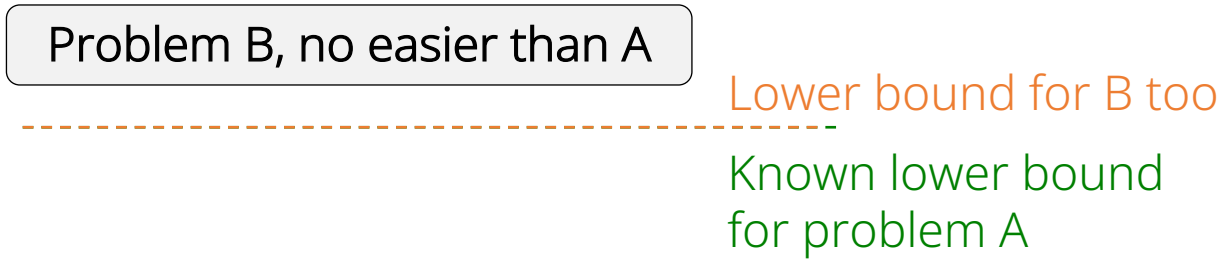
- Design algorithms to solve computational problems
- Mostly concerned with upper bounds on resources



E.g., Bellman-Ford is designed to find shortest paths in $O(VE)$ time

Computational complexity theory

- Classify problems based on their difficulty and identify relationships between those classes
- Mostly concerned with lower bounds on resources



E.g., Solving Knapsack is no easier than solving SAT, which is known intractable, so Knapsack is intractable too

Complexity classes

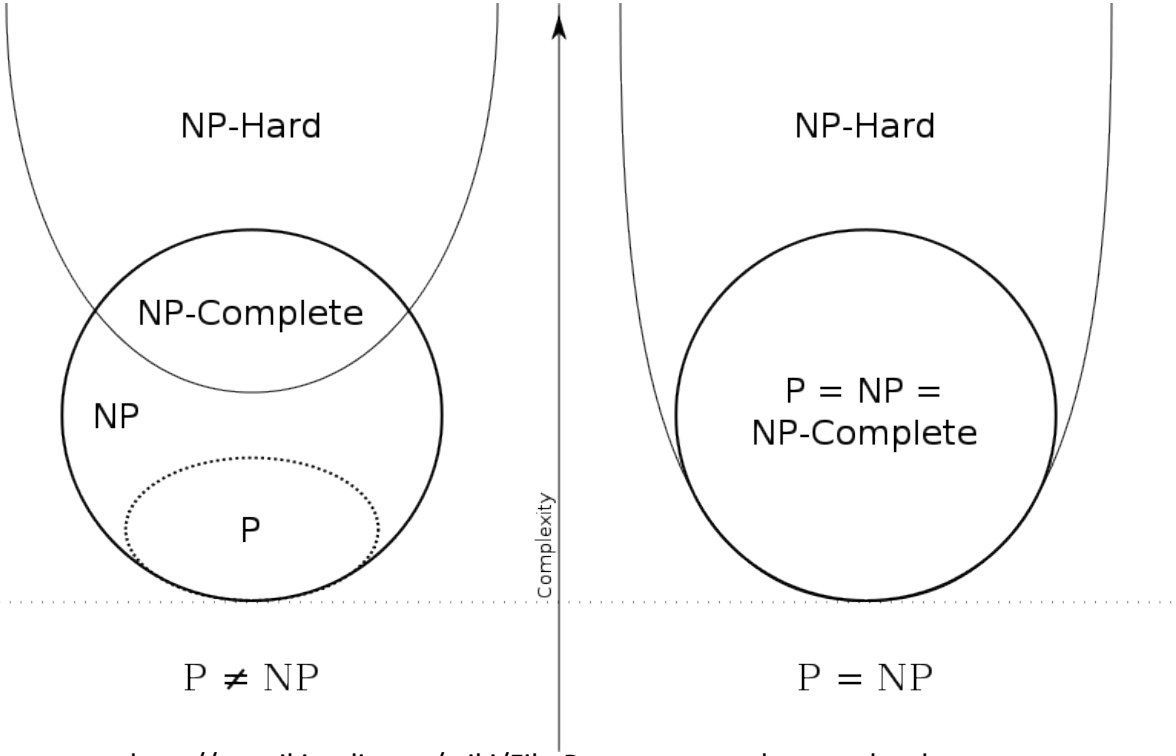
Class P: class of problems that can be solved in $O(n^k)$

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

- Hardness relationship can be determined via polynomial-time reduction

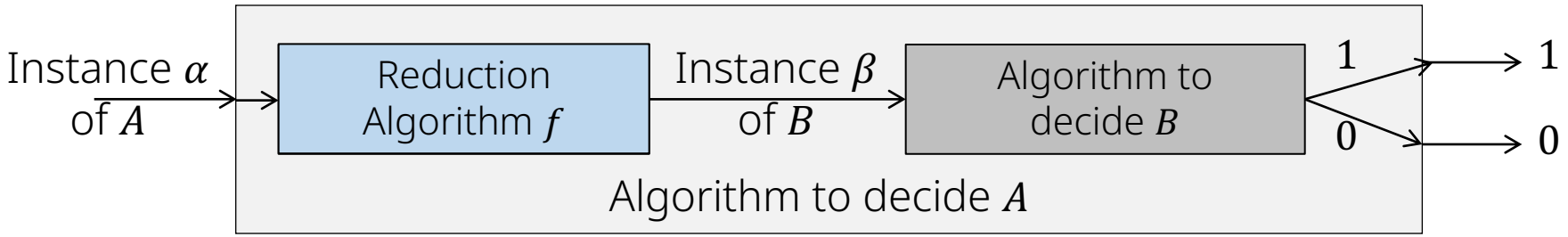
Class NP-complete (NPC): class of problems in both NP and NP-hard



http://en.wikipedia.org/wiki/File:P_np_np-complete_np-hard.svg

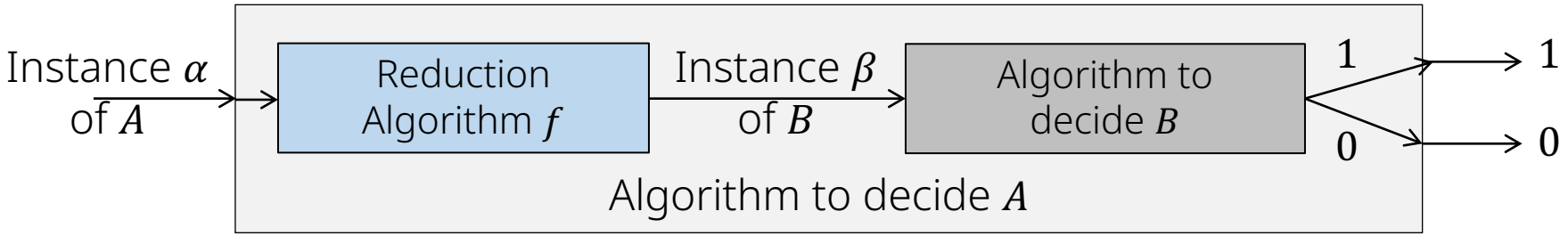
Reduction

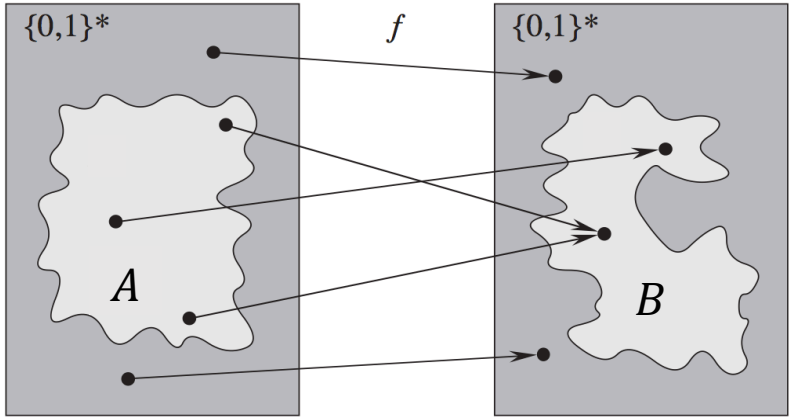
- A reduction is an algorithm for transforming every instance of a problem A into an instance of another problem B
  - We focus on decision problems here
- A polynomial-time reduction can help determine the hardness relationship between problems
  - We write $A \leq_p B$ if A can be reduced to B in p-time
  - $A \leq_p B$ implies A is no harder than B



Reduction

- To show the correctness of a p-time reduction $A \leq_p B$:
  - Show that B outputs 1 if and only if A outputs 1
  - That is, for all α, $AlgA(\alpha) = 1$ iff $AlgB(f(\alpha)) = 1$
  - Show that the reduction algorithm is in polynomial time



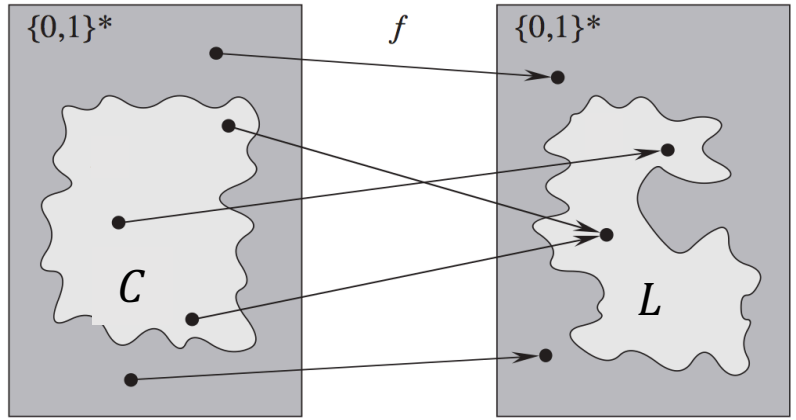


Note that $\{f(\alpha) | AlgA(\alpha) = 1\}$ is a subset of $\{\beta | AlgB(\beta) = 1\}$

Proving NP-Completeness

- $L \in$ NP-Complete $\Leftrightarrow L \in$ NP and $L \in$ NP-hard
- Step-by-step approach for proving L in NPC:

1. Prove $L \in$ NP
2. Prove $L \in$ NP-hard ($C \leq_p L$)
   - 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



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

- Algorithms in the real world!
  - 922 U0270 財務演算法 (Principles of Financial Computing)
  - 921 U9560 分子演算法 (Molecular Algorithms)
  - 922 U4020 圖形演算法及生物資訊應用 (Graph Algorithms and Their Applications in Bioinformatics)
  - 922 U0400 生醫資料探勘演算法 (Data Mining Algorithms for Bioinformatics)
  - 922 U0290 生物序列分析演算法 (Algorithms for Analyzing Biological Sequences

Studying advanced algorithmic theories and techniques

- Here are some advanced courses offered:
  - 922 U1810 演算法設計方法論 (Design Strategies for Computer Algorithms)
  - 921 U9400 基因遺傳演算法 (Genetic Algorithms)
  - 922 U3080 隨機演算法 (Randomized Algorithms)
  - 922 U3060 圖形演算法特論 (Special Topics on Graph Algorithms)
  - 922 M1250 演算法的數學解析 (Mathematical Analysis of Algorithms)
- And many many more
  - Parallel algorithms
  - Streaming algorithms
  - Algorithmic game theory
  - …

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.