Midterm Review

Nov 9th, 2017

Algorithm Design and Analysis

YUN-NUNG (VIVIAN) CHEN HTTP://ADA17.CSIE.ORG

國立臺灣大學
National Taiwan University

Midterm!!!

- **Date: 11/16 (Thursday)**
- **Time: 14:20-17:20** (3 hours)
- **Location: R103** (check the seat assignment before entering the room)
- Content
  - Recurrence and Asymptotic Analysis
  - Divide and Conquer
  - Dynamic Programming
  - Greedy
- Based on slides, assignments, and some variations (practice via textbook exercises)
- Format: Yes/No, Multiple-Choice, Short Answer, Prove/Explanation
- Easy: ~60%, Medium: ~30%, Hard: ~10%
- Close book

Algorithm Design & Analysis Process

1) Formulate a **problem**
2) Develop an **algorithm**
3) Prove the **correctness**
4) Analyze **running time/space** requirement

Design Step

Analysis Step

Algorithm Analysis

- Analysis Skills
  - Prove by contradiction
  - Induction
  - Asymptotic analysis
  - Problem instance
- Algorithm Complexity
  - In the worst case, what is the growth of function an algorithm takes
- Problem Complexity
  - In the worst case, what is the growth of the function the optimal algorithm of the problem takes

Algorithm Design Strategy

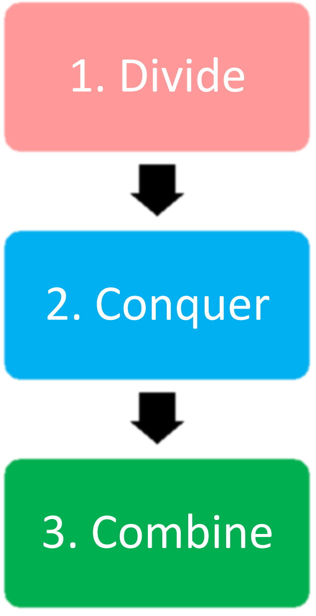
- Do not focus on "specific algorithms"
- But "some strategies" to "design" algorithms

- First Skill: Divide-and-Conquer (各個擊破)
- Second Skill: Dynamic Programming (動態規劃)
- Third Skill: Greedy (貪婪法則)

6 Divide-and-Conquer

What is Divide-and-Conquer?

- Solve a problem <u>recursively</u>
- Apply three steps at each level of the recursion
  1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem (比較小的同樣問題)
  2. **Conquer** the subproblems by solving them recursively If the subproblem sizes are *small enough*
     - then solve the subproblems — base case
     - else recursively solve itself — recursive case
  3. **Combine** the solutions to the subproblems into the solution for the original problem



How to Solve Recurrence Relations?

1. **Substitution Method** (取代法)
   - Guess a bound and then prove by induction
2. **Recursion-Tree Method** (遞迴樹法)
   - Expand the recurrence into a tree and sum up the cost
3. **Master Method** (套公式大法/大師法)
   - Apply Master Theorem to a specific form of recurrences

When to Use D&C?

- Analyze the problem about
  - Whether the problem with small inputs can be solved directly
  - Whether subproblem solutions can be combined into the original solution
  - Whether the overall complexity is better than naïve

10 Dynamic Programming

What is Dynamic Programming?

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems
  - 用空間換取時間
  - 讓走過的留下痕跡
- “Dynamic”: time-varying
- “Programming”: a *tabular* method

Dynamic Programming: planning over time

Algorithm Design Paradigms

- Divide-and-Conquer
  - partition the problem into **independent** or **disjoint** subproblems
  - repeatedly solving the common subsubproblems

  → more work than necessary

- Dynamic Programming
  - partition the problem into **dependent** or **overlapping** subproblems
  - avoid recomputation
    - ✓ Top-down with memoization
    - ✓ Bottom-up method

Dynamic Programming Procedure

- Apply four 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, typically in a **bottom-up** fashion
  4. Construct an optimal solution from computed information

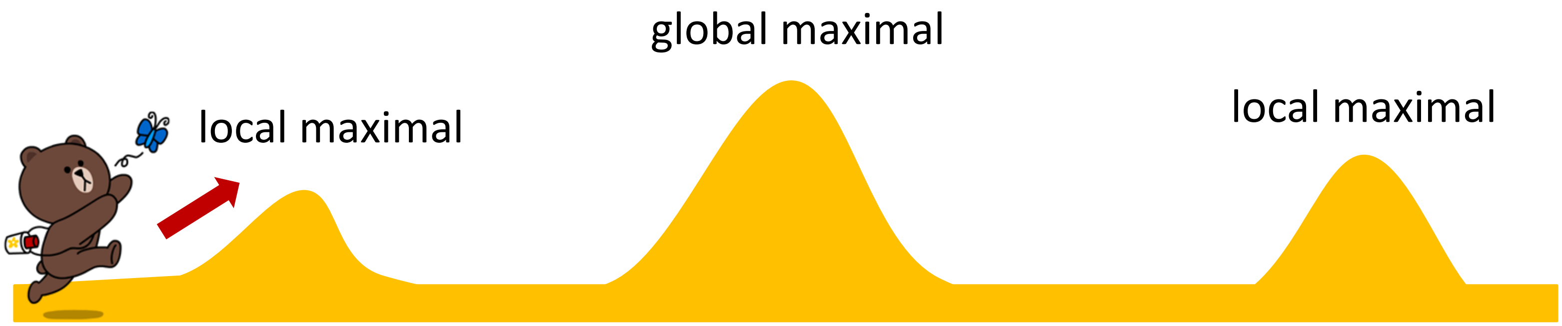
When to Use DP?

- Analyze the problem about
  - Whether subproblem solutions can combine into the original solution
  - When subproblems are overlapping
  - Whether the problem has optimal substructure
  - Common for optimization problem
- Two ways to avoid recomputation
  - Top-down with memoization
  - Bottom-up method
- Complexity analysis
  - Space for tabular filling
  - Size of the subproblem graph

15 Greedy Algorithms

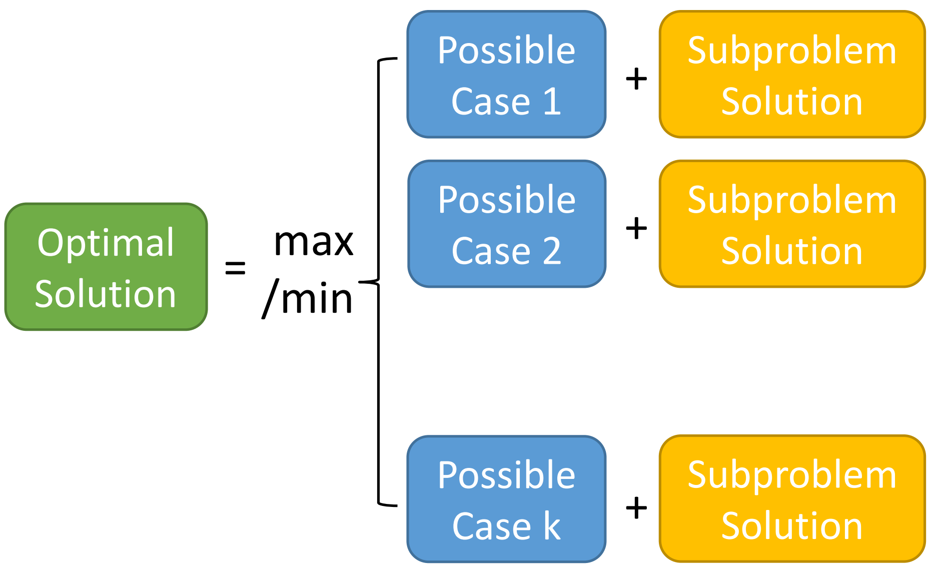
What is Greedy Algorithms?

- always makes the choice that looks best at the moment
- makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
  - not always yield optimal solution; may end up at local optimal

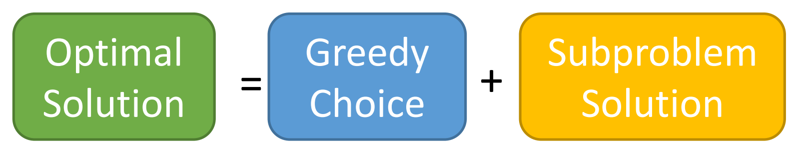


Algorithm Design Paradigms

- Dynamic Programming
  - has **optimal substructure**
  - make an informed choice after getting optimal solutions to subproblems
  - **dependent** or **overlapping** subproblems



- Greedy Algorithms
  - has **optimal substructure**
  - make a greedy choice before solving the subproblem
  - **no overlapping** subproblems
    - ✓ Each round selects only one subproblem
    - ✓ The subproblem size decreases

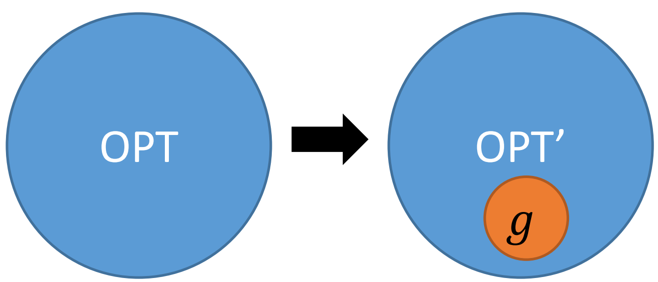


Greedy Procedure

1. Cast the optimization problem as one in which we make a choice and remain one subproblem to solve
2. Demonstrate the optimal substructure
   - ✓ Combining an optimal solution to the subproblem via greedy can arrive an optimal solution to the original problem
3. Prove that there is always an optimal solution to the original problem that makes the greedy choice

Proof of Correctness Skills

- Optimal Substructure : an optimal solution to the problem contains within it optimal solutions to subproblems
- Greedy-Choice Property : making locally optimal (greedy) choices leads to a globally optimal solution
  - Show that it exists an optimal solution that “contains” the greedy choice using **exchange argument**
  - For any optimal solution OPT, the greedy choice g has two cases
    - g is in OPT: done
    - g not in OPT: modify OPT into OPT’ s.t. OPT’ contains g and is at least as good as OPT



✓ If OPT’ is better than OPT, the property is proved by contradiction

✓ If OPT’ is as good as OPT, then we showed that there exists an optimal solution containing g by construction

When to Use Greedy?

- Analyze the problem about
  - Whether the problem has optimal substructure
  - Whether we can make a greedy choice and remain only one subproblem
  - Common for optimization problem

Optimal Solution = Greedy Choice + Subproblem Solution

21 Exercises

Short Answer Questions

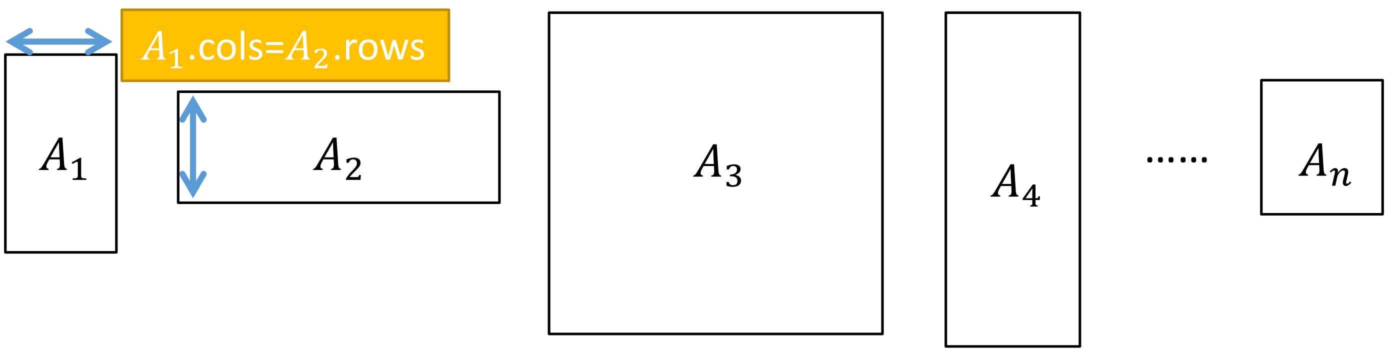
- True or False: To prove the correctness of a greedy algorithm, we must prove that every optimal solution contains our greedy choice.

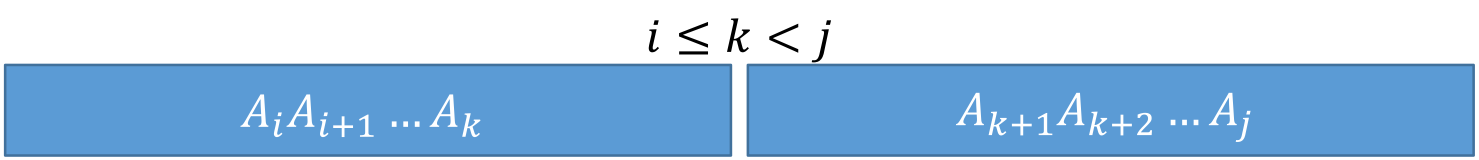
- Given the following recurrence relation, provide a valid traversal order to fill the DP table or justify why no valid traversal exists.

$$A(i, j) = F(A(i-2, j+1), A(i+1, j-2))$$

Matrix-Chain Multiplication

- Input: a sequence of integers $l_0, l_1, \dots, l_n$
  - l_{i-1} is the number of rows of matrix A_i
  - l_i is the number of columns of matrix A_i
- Output: a order of performing $n-1$ matrix multiplications in the maximum number of operations to obtain the product of $A_1 A_2 \dots A_n$





Q: Does optimal substructure still hold?

Huffman Coding

- In a binary Huffman code, the number of symbols that can be used to form a codeword is 2 (0 and 1). We can use a n-ary Huffman code, in which the number of symbols that can be used to form a codeword is n.

1) Please prove that the decoding tree which represents an optimal code for a file must be a *full* n-ary tree, i.e., all non-leaf nodes in the tree have n children.

2) Given the following information for a file, what are the lengths of the codewords for the characters 'a' and 'c' respectively, in a 3-ary Huffman code derived for this file?

Character	a	b	c	d	e	f	g
Frequency	700	400	200	100	1300	2400	100

Question?

Important announcement will be sent to @ntu.edu.tw mailbox
& post to the course website

Course Website: http://ada17.csie.org

Email: ada-ta@csie.ntu.edu.tw