Conclusion

The End

- This is an appropriate place to end a course on algorithms!

Final Review

What this course was all about...

Understanding

Analyzing

Designing

Algorithms to solve fundamental problems

What this course was all about...

By the end of the course, you would learn the following skills:

Check if an algorithm is correct
If not, debug it
Calculate the efficiency of an algorithm
If inefficient, modify it to make it more efficient
Given a problem, see if you can reuse an existing algorithm
If not, design a new one

Appreciate the “beauty” of algorithms

Algorithm Properties

- Correctness: An algorithm is correct if and only if it produces correct output for ALL valid/legal inputs

- Efficiency:
  - Time: How much time (or how many basic steps) are executed to solve a problem when it is specified with an input of a certain size
  - Space: How much space (or how many memory locations) are needed to solve a problem when it is specified with an input of a certain size

Correctness of algorithms

- What does it mean?

- How to prove it?
  - By counterexample
  - By contradiction
  - By induction
  - Loop invariants

Complexity of Algorithms

- Growth functions: Big-Oh, Omega, Theta, small-oh, small-omega
- Computing T(n) of non-recursive algorithms
- Determining an algorithm's order of complexity
- Comparing algorithms with different complexities
- Determining the recurrences of recursive algorithms
- Solving Recurrences
  - Recursion tree method
  - Master method
  - Substitution method
  - Changing variable method
  - Backward substitution method
  - Forward substitution method

Algorithm Design

- Recursive
- Non-recursive (Iterative)
- Divide and conquer
- Decrease and conquer

Consider efficiency at the time of recursive characterization

Recursive characterization of the solution of an optimization problem varies in two ways:

1. how many subproblems have to be solved within an optimal solution to the original problem
2. how many choices we have in determining which subproblem(s) to use in an optimal solution.

The efficiency of the corresponding dynamic programming algorithm depends on:

- The number of subproblems = the number of recursive calls
- The number of choices for each subproblem = the number of times the maximizing/minimizing loop (within which the recursive calls occur) has to execute

Two ways of designing a DP algorithm

Given a problem first check if it is amenable to dynamic programming (does it satisfy the two properties?), then you can either

1. Take the recursive algorithm resulting from the solution characterization step and revise it to add memoization.

OR

2. Develop an iterative dynamic programming algorithm with table lookup.

Which approach is better?

If all subproblems must be solved at least once, then the iterative DP algorithm will be more efficient because it avoids the overhead of recursion

If all subproblems need not be solved to get at an optimal solution, the recursive DP algorithm will be more efficient because the bottom up table approach will systematically solve all subproblems (because it fills the table completely) whereas the recursive one will only solve those that are needed as determined by the recursive executions that actually occur.

Reading Assignments

- 15.1: Rod Cutting
- Assembly Line Scheduling 作业里面有
- 15.2: Matrix Multiplication
- 15.3: Elements of DP
- 15.4: Longest Common Subsequence
- 16.1: Rod Cutting
- 16.2: Elements of the greedy strategy
- 16.3: Huffman Coding

Optimizing: Dynamic programming

- When is it applicable?
- How to apply it?
- The notion of memoization
- When dynamic programming does not work (e.g., finding longest path in a direct graph)

Dynamic programming

- Rod cutting
- Assembly-line scheduling
- Matrix-chain multiplication
- Longest common subsequence

Optimizing: Greedy algorithms

- Elements of the Greedy Strategy
- Differences and similarities between the greedy approach and dynamic programming
- Activity selection
- Huffman codes

Landscape of problems

- From constant time to combinatorial
- P, NP, NP-Complete and PTAS/FPTAS
- Introduction to approximation algorithms
- Network flow and bipartite matching
- Introduction to approximation algorithms

Final Exam

- Coverage
  - All materials we have learned in the class
  - All the reading assignments and homeworks
  - All the problems you can solve with the methods you have learned

Outline

- Divide and Conquer
- Greedy Algorithms, Dynamic Programming and Shortest Paths
- NP Completeness, Approximation Algorithms
- Network Flows, Bipartite Matching
- Randomized Algorithms

Final Exam

1. Understand an optimization problem
2. Come up with a recursive characterization of its solution
3. Check to see if it has optimal substructure
4. Produce a recursive algorithm
5. Check to see if it suffers from overlapping subproblems
6. Memoize the algorithm
7. Understand the problem-subproblem dependencies through a subproblem graph or recursion tree
8. Decide what size table needs to be filled in what order to construct the solution bottom up
9. Design a corresponding table lookup nonrecursive DP algorithm
10. Be able to determine whether the memorized recursive or nonrecursive table lookup algorithm will be more efficient
11. See if a recursive characterization in which only one subproblem results is possible
12. See if one of the choices in that characterization can be identified as a greedy choice
13. Can you prove that the greedy choice is indeed a correct choice?
14. Develop a corresponding greedy algorithm

Final Exam

15. Network flow, s-t flow, capacity/flow, augmenting path, Max-flow Mini-cut theorem.
16. Bipartite matching, augmenting path (different definition), algorithm.
17. NP, P, NP-hard, NPC, reduction.
18. Clique, vertex cover, subset-sum etc. are NP-Complete.
19. Approximation Ratios, PTAS, FPTAS, α-Approximation.
20. 2 approx for vertex cover, 2 approx for max-cut,
21. Randomized algorithm, Las Vegas algorithm, Monte-Carlo Algorithm
22. Quicksort, Karger's Randomized Min-Cut Algorithm