Algorithm II

# 1. Introduction

WU Xiaokun 吴晓堃

xkun.wu [at] gmail

# Algorithm: Design & Analysis

# What is an algorithm?

Simply put, algorithm is the instruction sequence for solving problem.

# What is an algorithm?

Simply put, algorithm is the instruction sequence for solving problem.

We can view an **algorithm** as a tool for solving a well-specified *computational problem*.

- (well formulated) desired input/output relationship.
- (can be implemented) specific computational procedure.

But before talking about algorithm, we need to think about the problem.

# Do we actually understand the problem?

One can do nothing based on a vague description.

- theoretically, we need mathematically clean core of the problem
- practically, structure of the problem decides algorithm choice

# Do we actually understand the problem?

One can do nothing based on a vague description.

- theoretically, we need mathematically clean core of the problem
- practically, structure of the problem decides algorithm choice

Proper formulation of an algorithmic problem is the first step.

# How to design the algorithm?

Or, how to solve the problem?

- guess the answer
- brute-force search
- resort to known techniques that solve similar problems
- innovate a new algorithm
  - creativity often comes from step-by-step improvements

# How to design the algorithm?

Or, how to solve the problem?

- guess the answer
- brute-force search
- resort to known techniques that solve similar problems
- innovate a new algorithm
  - creativity often comes from step-by-step improvements

Sufficient knowledge of *design patterns* makes the task easier.

# Why care about analyzing the design?

Think problem-solving as an *iterative procedure*:

1. design & analysis
2. implementation
3. verification

# Why care about analyzing the design?

Think problem-solving as an *iterative procedure*:

1. design & analysis
2. implementation
3. verification

Quantitative analysis requires an *uniform measure*.

- correctness
- time/space efficiency
  - room for improvements

# Why care about analyzing the design?

Think problem-solving as an *iterative procedure*:

1. design & analysis
2. implementation
3. verification

Quantitative analysis requires an *uniform measure*.

- correctness
- time/space efficiency
  - room for improvements

Rigorous mathematical proof can often be harder than the design itself.

- one can easily got lost while facing a final conclusion out-of-nowhere
- so use *progressive heuristics*, and bind analysis with design

# What do we discuss?

Computational problems, and Solving techniques.

- expose practical & interesting problems to you
- explain how other people design/solve them
- analyze/prove their correctness & efficiency

# What do we discuss?

Computational problems, and Solving techniques.

- expose practical & interesting problems to you
- explain how other people design/solve them
- analyze/prove their correctness & efficiency

This course is supposed to be taken fun

… in a painful way.

- A math course for CS students.

# Why it's useful to study this course?

Any problem-solving task can be benefited from this study.

- Whether the problem comes from CS or real life

# Why it's useful to study this course?

Any problem-solving task can be benefited from this study.

- Whether the problem comes from CS or real life
- problem database: relate problems in your future career to one that already known
- design patterns: choose the best tool that already understand
- analytical thinking: convince others that your solution is right

# What to expect?

Understand: when would we say a computational problem is *hard* (to solve on a computer)?

# What to expect?

Understand: when would we say a computational problem is *hard* (to solve on a computer)?

Be able to:

- if not, pick a provable good method to solve it.
- if so, prove it's hard and (if possible,) give a reasonable solution.

# What to expect?

Understand: when would we say a computational problem is *hard* (to solve on a computer)?

Be able to:

- if not, pick a provable good method to solve it.
- if so, prove it's hard and (if possible,) give a reasonable solution.

Basically, this is also the structure of this course.

# Course Overview

# Textbook

Primary textbook:

- Kleinberg & Tardos, *Algorithm Design*, 2006.

Not mandatory but recommended:

- Sedgewick & Wayne, *Algorithms*, 2011 4ed.
- Cormen et al., *Introduction to Algorithms*, 2009 3ed.

# Teaching plan I

Introduction & basic concepts recap:

- Computational Tractability
- Asymptotic Order of Growth
- Graphs

# Teaching plan II

Major algorithm design techniques:

- Greedy algorithms,
- Divide and conquer,
- Dynamic programming,
- Network flow.

# Teaching plan III

$\mathcal{NP}$ and PSPACE:

- Reducibility
- $\mathcal{P} \neq \mathcal{NP}$?
- NP-Complete
- $\mathcal{P} \neq PSPACE$?
- PSPACE-complete

# Teaching plan IV

Dealing with intractability:

- Identification of structured special cases,
- Approximation algorithms,
- Local search heuristics,
- Randomized algorithms.

# Teaching plan V

Invited talk:

- Tba

# Topics list

# Basics of Algorithm Analysis

Computational Tractability

- Worst-Case Running Times
- Polynomial Time

Asymptotic Order of Growth

- Upper Bounds ($O$), Lower Bounds ($\Omega$), Tight Bounds ($\Theta$).

Graphs

- Graph traversal: Breadth/Depth -First Search, implementation.

# Greedy Algorithms

Optimality

- "stays ahead".
- exchange argument.

Shortest Paths

- Dijkstra's Algorithm

Minimum Spanning Tree

- Kruskal's Algorithm
  - Union-Find data structure
- Prim's Algorithm
- Reverse-Delete Algorithm

Problems:

- Interval Scheduling

- Optimal Cache
- Clustering
- Huffman Codes and Data Compression.

# Divide and conquer

Recurrence relations

- Divide-by-2: Mergesort.
- Other divisions.

Approaches to Solving Recurrences

- Unrolling
  - Analyzing the first few levels
  - Identifying a pattern
  - Summing over all levels of recursion
- Substitution/Partial Substitution

Problems:

- Counting inversions
- Closest Pair of Points
- Integer multiplication

# Dynamic Programming

Principles

- Memoizing the Recursion
- Iteration over Subproblems

Problems:

- Weighted Interval Scheduling
- Segmented Least Squares
- Subset Sums and Knapsacks
- RNA Secondary-structure Prediction
- Sequence Alignment
- Shortest Paths
  - Minimum-Cost Path
  - Negative Cycle Detection

# Network Flow

Max-Flow Equals Min-Cut

- Ford-Fulkerson Algorithm
  - Augmenting Paths in a Residual Graph
  - Scaling Max-Flow Algorithm
- Preflow-Push Algorithm
  - Pushing and Relabeling

Problem:

- Bipartite Matching
- Disjoint Paths
- Minimum-Cost Perfect Matching

Application:

- Circulations with Demands (and Lower Bounds)
- Survey Design

- Airline Scheduling
- Image Segmentation
- Project Selection
- First Place Elimination

# NP and Computational Intractability

Reducibility

- Polynomial-time Reductions
  - Packing Problem
    - Vertex Packing (Independent Set)
    - Set Packing
  - Covering problem
    - Vertex Cover
    - Set Cover
- Reductions via “Gadgets”
  - Satisfiability Problem (SAT)
  - 3-Satisfiability (3-SAT)

$\mathcal{P} \neq \mathcal{NP}$?

- *Find* a solution vs. *Check* a solution.
- NP-Complete Problems
  - Circuit Satisfiability

## Important NP-Complete Problems

- Sequencing Problems
  - Traveling Salesman
  - Hamiltonian Cycle
    - Hamiltonian Path
- Partitioning Problems
  - 3-Dimensional Matching
  - Graph Coloring (k-Coloring)
    - Four-Color Conjecture
- Numerical Problems
  - Subset Sum
  - Scheduling with Release Times and Deadlines

## $\mathcal{NP} \neq \text{co-}\mathcal{NP}?$

- which implies: $\mathcal{P} \neq \mathcal{NP}$
- $\mathcal{P} = \mathcal{NP} \cap \text{co-}\mathcal{NP}?$

# PSPACE

$\mathcal{P} \neq PSPACE$?

- $\mathcal{P} \subseteq \mathcal{NP} \subseteq PSPACE$
- $\mathcal{P} \subseteq \text{co-}\mathcal{NP} \subseteq PSPACE$

PSPACE-complete

- Quantification: Quantified 3-SAT (QSAT)
- Games, e.g. Competitive Facility Location, Competitive 3-SAT
- Planning problems, e.g. Rubik's Cube, fifteen-puzzle

# Extending the Limits of Tractability

- Bound one of the parameters:
  - Finding Small Vertex Covers: divide-and-conquer
- Bound structure complexity of the graph:
  - Solving NP-Hard Problems on Trees:
    - Independent Set: greedy
    - Maximum-Weight Independent Set: dynamic programming
  - Ring, another simple topology:
    - Circular-Arc Coloring: dynamic programming

Tree Decompositions of Graphs

- Dynamic Programming over a Tree Decomposition
- Constructing a Tree Decomposition

# Approximation Algorithms

Greedy Algorithms

- Load Balancing
- Center Selection

Pricing method (Primal-dual method)

- Set Cover
- Vertex Cover
- Maximum Disjoint Paths

Linear Programming and Rounding

- Set Cover as an Integer Program
  - Rounding from Linear Programming
- Generalized Load Balancing

Dynamic programming with Rounding

# Local Search

The Landscape of an Optimization Problem

- Gradient Descent
- Choosing a neighboring solution at each step
  - Metropolis Algorithm
  - Simulated Annealing
- Choosing a Neighbor Relation
  - K-L heuristic
- Expectation Maximization

Applications

- Hopfield Neural Networks
- Maximum-Cut
- Image Segmentation (Multi-Labeling)
- Best-Response Dynamics and Nash Equilibria

# Randomized Algorithms

Random Variables and Their Expectations

Randomized Approximation Algorithm

Randomized Divide and Conquer

# Schedule

| Week | Date | Lecture |
|---|---|---|
| 1 | 2022/10/11 | Introduction |
| 2 | 2022/ | Algorithm Analysis & Graphs |
| 3 | 2022/ | Greedy Algorithm I |
| 4 | 2022/ | Greedy Algorithm II |
| 5 | 2022/ | Divide and Conquer I |
| 6 | 2022/ | Divide and Conquer II |
| 7 | 2022/ | Dynamic Programming I |
| 8 | 2022/ | Dynamic Programming II |
| 9 | 2022/ | Network Flow I |
| 10 | 2022/ | Network Flow II |

# Schedule (cont.)

| Week | Date | Lecture |
|---|---|---|
| 11 | 2022/ | Intractability |
| 12 | 2022/ | PSPACE |
| 13 | 2022/ | Extending Tractability |
| 14 | 2022/ | Approximation Algorithms |
| 15 | 2022/ | Local Search |
| 16 | 2022/ | Randomized Algorithms |

# Before-course Survey

What motivates you to learn algorithm design & analysis?

What are you expecting to learn from this course?

Have you ever thought of choosing your career in algorithm related positions? If so, in which specific area?