CS 341 - W24
Algorithms
Full Course Notes

With Prof Armin Jamshidpey

I took in-class notes and mostly avoided writing examples. I hope this high-level summary helps!

Josiah Plett

CS 341 Notes $\boxed{1}$

(1) Intro Lecture

★ Second half of the course is harder!

★ Every single question comes with proof of <u>correctness</u> & <u>runtime</u>.

Syllabus

- divide&conquer, master theorem
- breadth-first, depth-first
- greedy algorithms
- dynamic programming
- NP-completeness

$T(I)$ = runtime on input I

$T(n) = \max_{I \text{ of size } n} T(I)$

★ Great idea to memorize formulae for $\underline{O(n)}$, $\underline{\Omega(n)}$, $o(n)$, $\omega(n)$, $\underline{\Theta(n)}$

(2) Solving Recurrences

In this class, we use the artificial computation model; ignore memory, data locality etc.

Also, index 0 → index 1!

Multiple Parameters ★

$f(n,m) \in O(g(n,m))$ if $\exists C, m_0, n_0$ s.t. $f(n,m) \le C g(n,m)$ for $n \ge n_0$ <u>and/or</u> $m \ge m_0$

Computational Model: Word RAM

- word size $\ge \log(n)$ for any n
- operations + accesses are $\boxed{\text{unit cost}}$

Solving Problems

1. Understand the problem Definition.
2. Come up with "a solution" (eg. brute force) and expect to fail.
3. Improve your approach until you have GREATNESS!

Important Tools

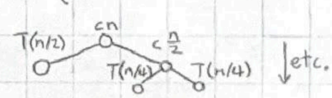
- Define mutually exclusive <u>cases</u>
- "Recursive" approach
- Dynamic: divide into sub-problems

This is inexact; if we want a formal proof for non-power values of N, then we can use the MASTER THEOREM. ★

Recurrence: $T(n) = \begin{cases} T(\lceil \frac{n}{2} \rceil) \times 2 + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$

Sloppy Recurrence: $T(n) = \begin{cases} 2T(n/2) + cn & \text{if } n > 1 \\ d & \text{if } n = 1 \end{cases}$

Recursion Tree:



Runtime: $\underbrace{cn + cn + \dots + cn}_{\log(n)} + d \cdot n = \Theta(n \log n)$

Master Theorem ★

Suppose $a \ge 1$ and $b > 1$ from: $\boxed{T(n) = aT(\frac{n}{b}) + \Theta(n^y)}$

Denote $\boxed{x = \log_b a}$ Then: $T(n) \in \begin{cases} \Theta(n^x) & \text{if } y < x \\ \Theta(n^y \log n) & \text{if } y = x \\ \Theta(n^y) & \text{if } y > x \end{cases}$

(3) Divide-and-Conquer

Substitution Method ★

To solve a recurrence relation:

1. Guess the solution (or its form)
2. use induction to prove it.

Divide-and-Conquer ★

Divide: split problem into mutually exclusive subprobs.

Conquer: Solve subproblems (recursively) w/same alg.

Combine: Use subproblem result to derive final result.

- <u>Inversion</u>: (i,j) where $i<j$ and $A[i] > A[j]$

(4) Algorithms We Should Know

Karatsuba's (polynomial multiplication)

$\in \Theta(n^{\log_2(3)})$ by combining $F_0G_1 + F_1G_0$ into $(F_0+F_1)(G_0+G_1) - F_0G_0 - F_1G_1$

Toom-Cook

(better Karatsuba's)

$\in \Theta(n^{\log_k(2k-1)})$

Strassen's (matrix multiplication)

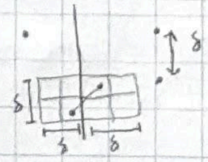
$\in \Theta(n^{\log_2(7)})$ via 7 multiplications!

Linear Quick Select (median of medians)

1. Median of medians ($n/5$ groups of 5)
2. Median select recursively! $\boxed{T(n) \in O(n)}$

Closest Pairs ★ $\Theta(n \log n)$

1. Split in half
2. Recurse
3. Solve transverse pairs:
   a. Boxes! compare to next 7 points ☺



(5) Breadth First Search

Undirected Graph ★

Def: $G=(V,E)$ with $|V|=n$, $|E|=m$

Data Structures:

- Adjacency list: $A[1...n]$ s.t. $A[v]$ is a linked list of v-connected edges.
  Size: $\boxed{\Theta(n+m)}$ Edge exists: $\omega(1)$
- Adjacency matrix: binary matrix of size $n \times n$ w/ $M[v,w]=1$ iff $\{v,w\} \in E$.
  Size: $\boxed{\Theta(n^2)}$ Edge exists: $O(1)$

You should know these from Math239:

path: $v_1 \dots v_k$ s.t. $\forall \{v_i, v_{i+1}\} \in E$

connected graph: $\forall v,w \in V$, $\exists$ path $v \rightsquigarrow w$

cycle: path $v_1 \dots v_k, v_1$ with $k \geq 3$ and no repeat vertices.

tree: connected graph with no cycles.

rooted tree: tree with special vertex "root".

subgraph: strictly inside G :) ya know

connected component: maximal connected subgraph of G.

BFS Tree ★

$O(n+m)$

Definition: subgraph made of:
- all w s.t. parent$[w] \neq$ NIL
- all $\{w, \text{parent}[w]\}$, for w as above (except $w=s$)

(10) Greedy Algorithms

Hamiltonian Paths ★

simple path that visits every vertex exactly once.

for undirected G: np-complete

for directed G: $O(n+m)$

Greedy Paradigm ★

"Don't think ahead"

EXAMPLES
- Interval Scheduling
- Interval colouring
- Minimizing completion time
- Dijkstra's algorithm
- Min spanning trees.

(6,7) Depth First Search

DFS Algorithm ★

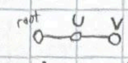
idea: as deep as possible, then backtrack

structure: use a stack (recursion/queue)

shortest paths? no

runtime: $O(n+m)$

ancestor/descendant:



u is ancestor of v; v is descendant of u

Back edge: edge in G that's not a tree edge in our DFS tree.

Cut vertex: vertex v in connected G s.t. $G \setminus v$ is disconnected.

Cut Vertex Claim ★

For any $v \neq s$, v is a cut vertex iff it has a child w with

$m(w) \geq \text{level}[v]$ ($m \to$ smallest descendent neighbour)

↳ see bottom of this page.

★ A Directed Graph is a DAG of disjoint strongly connected components.

Kosaraju's Alg ★

$O(n+m)$

For a directed graph G, the transpose graph $G^T = (V, E^T)$ is the graph with edges reversed.

Compute strongly connected components:

```
SCC(G):
1    run DFS on G, record finish times.
2    run DFS on G^T, with vertices
     ordered in decreasing finish time.
3    return trees in DFS forest of G^T.
```

Exchange Argument ★

Essentially: prove any swap between elements cannot create a more optimal solution

(8) Directed Graphs

DAG: directed, acyclic graph.

(9) Connectedness

Testing Acyclicity

G has a cycle IFF there is a back edge in its DFS forest.

Topological Ordering

Suppose $G=(V,E)$ is a DAG.

A topological order is an ordering ($<$) of V s.t. $\forall e=(v,w) \in E$, we have $v<w$.

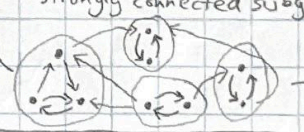
Strong Connectivity

iff $\forall v,w \in G$, $v \sim w$

Testing Strong Connectivity: Explore twice from $s \in V$, once with edges reversed.

Strongly Connected Component

- Subgraph of G
- which is strongly connected
- not contained in another strongly connected subgraph



(11) Greedy Algorithm Correctness

Greedy Alg Stays Ahead ★

1. Define an optimal outcome and a greedy outcome.
2. Prove the greedy outcome always returns an equal or better result than optimal.

↳ induction ↳ contradiction

$a(v) = \min(\text{level}[w])$, where $\exists \{v,w\} \in E$

$m(v) = \min(a(w))$, where w descendant of v

CS 341 Notes 2

12 Minimum Spanning Trees

Min Spanning Tree: spanning tree with minimal weight.

Kruskal's Alg

$O(m \log(m) + n^2)$

Greedy alg for minimum spanning T:

1. Take smallest edge
2. If made cycle, undo!

Exchanging Edges ★

Let (V, A) & (V, T) be spanning trees.
Let $e \in T$, $e \notin A$.
Then, $\exists e' \in A, e \notin T$, such that $(V, T) + e' - e$ is still a spanning trees.
Also, e and e' are on same cycle.

Merging Connected Sets

To find if we made a cycle! Use a list of linked lists implementing "give each node a colour representing its group."

13 Dynamic Programming

Tool: Dynamic Programming ★

1. Break problem down into subproblems
2. Store subproblem solutions
   - ↳ because they'll overlap; avoid recomputing!

14 More Dynamic Programming (LEC 12 SLIDES)

DP on various substring optimization subproblems.

15 Dynamic Programming Part 3

Longest Common Subsequence ★

i \ j	0	1	2	3	4
0	0	0	0	0	0
1	0	1	1	1	1
2	0	1	1 → 2	2	
3	0	1	1	2	2

j | darn
i | dry

$M[i,j]$ = MAX OF:
- ↳ $M[i, j-1]$
- ↳ $M[i-1, j]$
- ↳ IF $A[i] == B[j]$, $1 + M[i-1, j-1]$

Spellchecker

3 issues: add, delete, change.
Input: 2 arrays of chars, A and B
Output: Min # of modifications to turn A into B.

16 DP Part 4: Bellman-Ford

★ Bellman-Ford 2.0

(runtime: $O(mn)$)

Slowly "relaxes" all edges at once, until every vertex has it's minimum cost from the root

Solves Which Problem? Dijkstra but negative edges

- ★ $O(m)$ to check for a reachable negative cycle at the end
- ★ Implementable via walking every edge once (in any order), n times.

Independent Sets

Independent set S of $G=(V,E)$ is $S \subseteq V$ s.t. there are no edges between nodes of V.

★ Excercise for Finals practice!

17 DP Part 5: Floyd-Warshall

Floyd-Warshall Algorithm ★

Compute shortest path between every pair of nodes in a weighted graph (which may have negative weights). ⟹

$$D = \begin{bmatrix} 0 & \infty & \infty & & \\ \infty & 0 & \infty & & \\ \infty & \infty & 0 & \ddots & \\ & \ddots & & 0 & \\ & & & & 0 \end{bmatrix}$$

where $D[i][j]$ is shortest known path from $i \to j$, initialized to the weight of a direct edge (if exists)

Algorithm: $\forall k \in V$, update all $D[i][j]$ with $i \to k \to j$

18 Polynomial Time Reduction

Decision Problem: Given instance I, answer "yes" or "no".
$Size(I)$ = # of bits required to specify instance I.

Polynomial Time Reduction

A is polynomial time reducible to a decision problem B if there exists an algorithm F that transforms all I_A into an instance I_B such that

I_A is YES $\Leftrightarrow$ $F(I_A) = I_B$ is YES ★

(we write $A \leq_p B$ if such a polynomial time reduction exists.)

19 Simple Reductions

Simple Reduction: Proving one problem is equivalent to another by conventional proof means.

Examples

All equivalent re: polynomial solvability

- Is there a fully-connected clique in G of $\geq k$ vertices?
- Is there an independent set in G of $\geq k$ vertices?
- Is there a vertex cover in G of $\geq k$ vertices?

Equivalent ($HC =_p HP$)

- Does G have a Hamiltonian Cycle?
- Does G have a Hamiltonian Path?

20-21 NP-Completeness

cont'd next page

Proving NP ★

1. Understand your input S & certificate t
2. Provide polynomial-time alg for: $ALG(S, t) \to$ YES/NO
3. Remember, this is only verification

Definitions:

Co-NP: set of problems whose no-instances are NP.

P: set of problems solvable in polynomial time.

NP-Complete: hardest problem in NP.

Principle behind NP-C

If X is NP (verified), then it's the hardest problem in NP iff

$Y \leq_p X$ for all $Y \in NP$

(20-22) NP-Completeness

NP-Complete Problem Bank

- 3SAT, SAT $((x \vee y \vee \bar{z}) \wedge (\bar{x} \vee y \vee z))$
- Independent set [of size k]
- Vertex Cover, Clique [of size k]
- Hamitonian Cycle & Path (directed)
- Travelling Salesman
- Subset sum ($\sum a_i = k$ from array)
- Binary Knapsack (item: $\frac{\text{weight}}{\text{value}}$)

Proving NP-Completeness

Given: Decision problem X

1. Prove X is NP
   ↳ Give polynomial-time verification.
2. Choose Q from NPC
3. Show Q reduces to X ($Q \leq_p X$)
   ↳ Given I_Q, generate I_X. (in polynomial time)
   ↳ Prove that if I_Q is a yes-instance, so is I_X, and same for no-instances.

(or, YES iff YES, or NO iff NO).

NP-Completeness Examples

Circuit-SAT

Input: DAG with labelled vertices.

Decision: choice of booleans x_i that make v true.

Proof: Look, every problem in the world can be built using a DAG with labelled verices! Since every problem (ever) reduces to it, it's NP-Complete.

Directed Hamiltonian Cycle

What to reduce to: 3-SAT

Proof idea:
- row of nodes for each variable
- directed edges: left=T, right=F
- node for each clause; connect it smartly :)

(23-24) NPC Examples

Perfect 3D Matchings

Input: 3 disjoint sets X, Y, Z of size n and a family of hyper-edges $E \subseteq X \cdot Y \cdot Z$

Decision: is there a perfect matching, which is a full, disjoint-nodes cover?

Proof idea: make fidget spinners? My takeaway is: construct mini-tools that translate a subproblem of the chosen NPC problem into a subproblem of what you're trying to prove

NOTE

We are not required to understand these proofs for the final!

(I only include them for inspiration)

↑ us at the front of class holding numbers for armin's demonstrations ↑