Algorithms and Complexity
UCLA-CS180-S18

Quentin Truong
Taught by Professor Meka

Spring 2018

Contents

1 Ch5: Divide and Conquer

1.1 Introduction

- Divide and conquer
  - Class of algorithmic techniques in which one breaks input into several parts, solves subproblems recursively, and recombines into overall solution
- Analyze running time
  - Generally will use recurrence relations
  - For divide and conquer problems, the brute force solution is typically polynomial; we are trying for a lower polynomial

1.2 A First Recurrence: The Mergesort Algorithm

- Mergesort
  - Sort list of numbers by dividing into two equal halves, solving each half recursively, and recombining solutions
  - $T(n) \leq 2T(n/2) + O(n)$
  - Ignore ceiling and floor issues for odd numbers, b/c not actually impactful
- Approaches to Solving Recurrences
  - Unrolling the recurrence into a graph allows us to see how many operations are performed at each level
    - Identify the pattern, and sum over all levels
  - Substituting a solution into the recurrence
    - Requires a guess
  - Partial substitution can determine the constants
    - Useful for determining exact constants if we know the general form of the solution

1.3 Further Recurrence Relations

- 5.3 $T(n) \leq qT(n/2) + cn$
- 5.4 Any function T() satisfiyng 5.3 with q >2 is bounded by $O(n^{\log_2 q})$
- 5.5 Any function T() satisfying 5.3 with q=1 is bounded by O(n)
- 5.6 $T(n) \leq 2T(n/2) + cn^2$

1.4 Counting Inversions

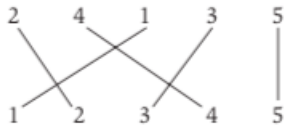

Figure 5.4 Counting the number of inversions in the sequence 2, 4, 1, 3, 5. Each crossing pair of line segments corresponds to one pair that is in the opposite order in the input list and the ascending list—in other words, an inversion.

- Application of Counting Inversions
  - Analysis of rankings
    - Collaborative filtering, to match preferences to those of other people on the Internet

    - Recommend things according to what other similar people like
  - Meta-search tools
    - Execute same query on different search engines
    - Measure how "out of order" the different orderings are
  - Generally
    - Given a sequence of distinct numbers, measure how far the list is from being in ascending order
    - Number should increase as more scrambled
- More formal definition of Counting Inversions
  - Count the number of indices i <j which form an inversion, $a_i > a_j$
- Algorithm
  - Divide list in two halves, recursively sort and count inversions for each
  - To recombine two halves, take min of each sorted half
  - If take min from first half, no inversions are added
  - If take min from the second half, then add the number of remaining elements from the first half to the inversion count

1.5 Finding the Closest Pair of Points

- 1D Algorithm
  - Sort; min must be adjacent to each other, so traverse
- 2D Algorithm
  - P_x are the points sorted with respect to x
  - P_y are the points sorted with respect to y
  - L is the line dividing P_x in half
  - Q are first half of points in P_x
  - R are second half of points in P_x
  - Recursively determine closest pair of points in Q; then in R
  - $\delta = \min(d(q_0^*, q_1^*), d(r_0^*, r_1^*))$
  - If there exists q ∈ Q and r ∈ R for which d(q, r) $<\delta$, then each of q and r lies within a distance of δ of L
  - If s, s′ have d(s, s′), then s and s′ are within 15 positions of each other in the sorted S_y
    - Each box has sidelength (δ/2), and at most one point (b/c if had more than one point, *delta* would be different)
    - 16 boxes leads to 16 possible positions
    - Can reduce down to 5 possible positions using packing arguments
    - Linear time to try/fail find a position in S
  - Bruteforce for $P \leq 3$
  - Satisfies recurence found in 5.1 to achieve O(nlogn)

1.6 Integer Multiplication

- Multiplication
  - Adding up partial products works the same in base-10 as in base-2
- Karatsuba's Algorithm
  - Based on breaking up partial sums
  - $xy = (x_1 * 2^{n/2} + x_0)(y_1 * 2^{n/2} + y_0$
  - $= x_1y_1 * 2^n + (x_1y_0 + x_0y_1) * 2^{n/2} + x_0y_0$
  - This achieves $T(n) \leq 4T(n/2) + cn = O(n^2)$
  - $(x_1 + x_0)(y_1 + y_0) = x_1y_1 + x_1y_0 + x_0y_1 + x_0y_0$
  - Subtract away x_1y_1 and x_0y_0
- Analysis
  - $T(N) \leq 3T(n/2) + O(n)$
  - $O(n^{log_2 3})$

1.7 Convolutions and the Fast Fourier Transform

- Convolution
  - Take two vectors of length n and produce a vector of 2n - 1 coordinates
  - $\sum_{(i,j):i+j=k;i,j<n} a_i b_j$
  - Many applications, especially in signal processing
- Fast Fourier Transform
  - Can calculate convoution in nlogn
  - Product of polynomials is equivalent to convolution
  - Reconstruct C from values on the $(2n)^{th}$ roots of complex roots of unity
  - Coefficients of C are coordinates of convolution vector

2 Ch3: Graphs

2.1 Basic Definitions and Applications

- Graph is a collection V of nodes and collection E of edges
- Undirected Graph
  - Edge is two-element subset of V
- Directed Graph
  - Edge is an ordered pair of two vertices
- Applications
  - Transportation networks: airplanes and trains
  - Communication networks: wired and wireless
  - Information networks: WWW as directed graph
  - Social networks: used to study interactions among people, etc
  - Dependency networks: university course offerings with prereqs, food web, software modules
- Paths and Connectivity
  - Path in undirected graph is sequence of consecutive nodes joined by edges
  - Path is simpe if all vertices are distinct
  - Cycle is sequence of 3 or more vertices where first and last vertex are the same
  - Undirected graph is connected if there is a path between all pairs of nodes
  - Directed graph is strongly connected if for all nodes u and v, there is path uv and vu
  - Distance between nodes is minimum number of edges
  - Trees are undirected, connected graphs without cycles
    - Orient edges away from each other, such that nodes are descendants of ancestors
    - (3.1) Every n-node tree has exactly n-1 edges
  - Rooted trees show a hierarchy
- (3.2) Let G be undirected graph on n nodes. Any two implies third
  - G is connected
  - G does not contain a cycle
  - G has n-1 edges

2.2 Graph Connectivity and Graph Traversal

- S-T Connectivity (Maze-solving)
  - Use BFS and DFS
- Breadth-First Search (BFS)
  - (3.3) For each j $\geq$ 1, layer L_j produced by BFS consists of all nodes at distance exactly j from s. There is a path from s to t iff t appears in some layer
  - (3.4) Nodes X and Y in BFS tree in layers L_i and L_j are connected by an edge, then i and j differ by at most 1
- Exploring a Connected Component
  - BFS discovers the set of reachable nodes
  - Could discover this set in many ways
- Depth-First Search (DFS) (Maze-exploration)
  - Explore as deeply as possible, retreat only when necessary
  - Maintain global knowledge of which nodes have been explored
- Set of all Connected Components
  - (3.8) For any two nodes, their connected components are either identical or disjoint

2.3 Implementing Graph Traversal Using Queues and Stacks

- Representing Graphs
  - Adjacency Matrix
    - n x n matrix where A[u, v] is 1 if contains edge

        - $\Theta(n^2)$ space
        - $\Theta(n)$ time to find incident edges
    - Adjacency List
        - Reecord for each node v, containing list of nodes to which v has edges
        - Total length of al lists is 2m = O(m)
        - Much better for sparse
        - O(m+n) space where m is edges n is nodes
- BFS - Queue
- DFS - Stack
- Finding Set of All Connected Components
    - BFS/DFS a node, repeat on unconnnected nodes until find all groups

2.4 Testing Bipartiteness: Application of BFS

- (3.14) Bipartite graph cannot contain odd cycle
- Red-Blue
    - Color node red, BFS, color all those blue, BFS, swap back to red
    - Once all colored, scan all edges, check for edges with same colors on both sides of edge

2.5 Connectivity in Directed Graphs

- Directed Graphs
    - Adjacency list has two list
        - Nodes to which it has edges
        - Nodes from which it has edges
- Graph Search Algorithms
    - BFS/DFS will traverse set of nodes to which starting node is connected to
    - If want set of nodes with paths to s (not set of nodes to which s has paths), reverse direction of every edge, then run BFS/DFS from this new s
- Strong Connectivity
    - For every two nodes, there is path from u to v and v to u
- Mutually Reachable
    - Path from u to v and v to u
- (3.17) For any two nodes, their strongly connected component is either identical or disjoint

2.6 Directed Acyclic Graphs and Topological Ordering

- Undirected Graph without cycles
    - Each connected component is a tree
- Directed Acylic Graphs
    - Used often as graph for dependencies
- Topological Ordering
    - Every edge points forward in the ordering
    - Ordering of nodes, such that for every edge (v_i, v_j), we have i <j
    - Topological ordering can provide immediate visual proof of no cycles
    - Topological ordering implies DAG
    - DAG implies topological ordering
- Computing Topological Ordering
    - Find node with no incoming edges, place it first
    - Delete that node
    - Recursively compute topological ordering for rest of graph
    - Append deleted node to front of remaining topological ordeing
    - Complexity Analysis
        - $O(n^2)$ - Trivially search

– O(m+n) - If track number of incoming edges for each node and set of nodes without incoming edges

3 Ch4: Greedy Algorithms

3.1 Introduction

- Greedy
  - Builds up solution in samll steps, choosing decision at each step to myopically optimize some underlying criterion
  - Sometimes can even create algorithms with greedy rules which are guaranteed to be close, but not equal, to optimum
- Proving
  - Show step-by-step that greedy is better than any other algorithm, including optimal
  - Gradually exchange parts of any solution with the greedy solution, showing that the greedy never hurts the quality of solution

3.2 Interval Scheduling: The Greedy Algorithm Stays Ahead

- Interval Scheduling Problem Statement
  - Set of requests, each with starting and finishing time
  - Compatible (no two overlap in time) sets of maximum size is called optimal
- Designing a Greedy
  - Use simple rule to select first request i_1, then reject any requests which are not compatible with i_1
  - Many wrong solutions
  - Smallest Finishing Time is a good rule
  - Intuition is that we want our resource to become free as soon as possible
- Analyzing the Algorithm
  - All requests will definitely be compatible
  - Cannot show that this produces the optimal, because there are many optimal algorithms
  - Simply show that this algorithm contains same number of intervals as the optimal
  - Do this by comparing partial solutions from greedy to the optimal and show that greedy is doing as good or better
- Proof
  - Let Greedy Algorithm produce set of requests A = $i_1, ..., i_k$
  - Let Optimal set O = $j_1, ..., j_m$
  - Greedy rule guarantees $f(i_1) \leq f(j_1)$
  - Goal is to show $k = m$
  - (4.2) Lemma: For all indices $r \leq k$ we have $f(i_r) \leq f(j_r)$
    - Proof by induction
    - This is true for $r = 1$
    - Suppose true for $r - 1$, prove for r
    - Wrt one step, greedy would only finish after optimal if it chose a later-finishing interval.
    - Formally, $f(j_{r-1}) \leq s(j_r)$ by compatibility
    - $f(i_{r-1}) \leq s(j_r)$ bc $f(i_{r-1}) \leq f(j_{r-1})$
    - So greedy can always pick the same job as optimal
  - (4.3) Greedy algorithm returns an optimal set
    - Since next request from optimal starts after j_k and thus after i_k, Greedy is able to pick it
- Interval Partitioning Problem
  - Ex: Lectures at predefined start/end times need to be scheduled in classrooms, want to minimize number of classrooms used
  - Depth is the maximum number of set of intervals that pass over any single point in time
  - Number of resources needed is at least the d = depth of set of intervals
  - Assign label to each interval, label is in range from $1, ..., depth$
  - Basically, just sweep through interval left to right, assigning nonconflicting labels to intervals
  - (4.5) Every interval will be assigned a label and no two overlapping intervals will receive same label
  - Proof

- Consider one interval I_j and t intervals earlier that overlap it
- Then there are $t + 1 \leq d$ and so $t \leq d - 1$
- So there is at least one label which can be assigned to I_j
- No intervals overlap because no labels are reassigned in same interval by definition

3.3 Shortest Paths in a Graph

- Single-source Shortest-paths Problem
  - Dijkstra
  - Maintain set S of vertices u for which we have determined a shortest-path distance
  - Extend set by finding vertex with path through explored set with minimum cost
  - (4.14) Consider set S at any point in algorithm's execution. For each $u \in S$ the path P_u is a shortest path $s - u$ path
  - Proof by induction
    - Base case is trivial
    - Suppose true for $|S| = k$, prove for $k + 1$ by adding node v using edge (u, v) to complete path P_v
    - By induction, P_u is shortest from $s - u, u \in S$
    - Any other path $s - v$ must be at least as long as P_v because it is already at least as long as P_v by the time it leaves set S
    - This is because Dijkstra had already considered adding other nodes, but chose not to do that because they are not the smallest path available
    - Formally, let P_1 is subpath of P from $s - x$
    - Let P_x be shortest $s - x$ path,
    - Then, $l(P_1) \geq l(P_x) = d(x)$
    - So subpath out of P to node Y is length $l(P_1) + l(x, y) \geq d(x) + l(x, y) \geq d'(y)$

3.4

-

4 Ch4: Greedy Algorithms

4.1 Basic Definitions and Applications

–