

# Algorithms and Complexity

## UCLA-CS180-S18

Quentin Truong  
Taught by Professor Meka

Spring 2018

### Contents

<b>1</b>	<b>Algorithm Design: Ch5: Divide and Conquer</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	A First Recurrence: The Mergesort Algorithm . . . . .	2
1.3	Further Recurrence Relations . . . . .	2
1.4	Counting Inversions . . . . .	2
1.5	Finding the Closest Pair of Points . . . . .	3
1.6	Integer Multiplication . . . . .	3
1.7	Convolutions and the Fast Fourier Transform . . . . .	4
<b>2</b>	<b>Ch3: Graphs</b>	<b>5</b>
2.1	Basic Definitions and Applications . . . . .	5
2.2	Graph Connectivity and Graph Traversal . . . . .	5
2.3	Implementing Graph Traversal Using Queues and Stacks . . . . .	5
2.4	Testing Bipartiteness: Application of BFS . . . . .	6
2.5	Connectivity in Directed Graphs . . . . .	6
2.6	Directed Acyclic Graphs and Topological Ordering . . . . .	6
<b>3</b>	<b>Ch3: Graphs</b>	<b>8</b>
3.1	Basic Definitions and Applications . . . . .	8

# 1 Algorithm Design: 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

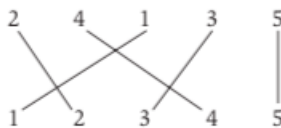
## 1.2 A First Recurrence: The Mergesort Algorithm

- 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  $T()$  satisfying 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  $q \in Q$  and  $r \in R$  for which  $d(q, r) < \delta$ , then each of  $q$  and  $r$  lies within a distance of  $\delta$  of  $L$
  - If  $s, s'$  have  $d(s, s') < \delta$ , then  $s$  and  $s'$  are within 15 positions of each other in the sorted  $S_y$ 
    - Each box has sidelength  $(\delta/2)$ , 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  $S$
  - Brute force for  $P \leq 3$
  - Satisfies recurrence found in 5.1 to achieve  $O(n \log n)$

## 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 convolution in  $n \log n$
  - 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 simple 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
    - $n \times n$  matrix where  $A[u, v]$  is 1 if contains edge

- $\Theta(n^2)$  space
- $\Theta(n)$  time to find incident edges
- Adjacency List
  - Record for each node  $v$ , containing list of nodes to which  $v$  has edges
  - Total length of all 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 unconnected 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 Acyclic 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 ordering
  - 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 Ch3: Graphs

### 3.1 Basic Definitions and Applications

—