Stanley Wei

CS180

Sarrafzadeh (F23)

Table of Contents

Overview

Algorithms:

Stable Matching

Majority Problem

Graph Traversal

- BFS
- DFS
- Bipartite graphs: Graph coloring (BFS)

Topological Sorting

Greedy Paradigm

- Interval Scheduling

Shortest Path: Dijkstra's algorithm

Minimum Spanning Tree

- Prim's algorithm
- Kruskal's algorithm

Crossings Problem: Counting inversion

Paradigms:

Remove Vertices: Majority Problem

Greedy: Interval Scheduling, Shortest Path, Minimum Spanning Tree

Divide and Conquer: Merge sort

10/1/23 (missed)

- Serial model of computation; von Neumann model of computing
- Sorting - heapsort (n log n)
    - Special case: limited time sorting (ex: only 0s and 1s to sort -> count # of 0s, create new array [linear time]

<u>Stable Matching</u>

Definition (*Matching*): given two groups of n elements, a *<u>complete matching</u>* is a mapping such that every element in one group mapped to exactly one element in the other.

- A complete matching is called "*<u>stable</u>*" if there do not exist any cases in which two elements not matched with each other both prefer each other over their current matches.
    - Alternatively: Both have each other higher in their ranking than their current match.

Problem: Every element in each group has a priority list/complete ranking of which elements in the other group it prefers to match with; find the matching that maximizes the priority of elements selected (i.e. that tries to pick the most-preferred match for each element in every group)

- To represent an element taking multiple matches from the other set, put multiple copies of the element in its set.

Notation: draw a matching as two sets of dots (elements), with lines between dots representing an individual match

- Indicate an element's preference of another element with an arrow from the preferrer to the preferred

Algorithm (Gale-Shapley)

1. Pick arbitrary x in group 1. Match x with its highest-priority match.
2. Continue picking unmatched elements x from group 1 until all elements have been matched. For each element from group 1, pick the highest-ranked match it has not already considered ("asked").
    a. If that match is either currently unmatched or has the new element higher in its ranking than its current match, match the new element and the match (and unmatch the match's former match).

b. Otherwise, pick the element's next-highest match.

Proof of Correctness

Proof: *No elements will be unmatched.*

Assume, for the sake of contradiction, that there is an element in group 1 that is unmatched when the algorithm terminates.

- If there is an element in group 1 that is unmatched, then there must also be an element in group 2 that is unmatched.
- The only case in which the element in group 2 would not match with the element in group 1 is if the element in group 2 already found a better match; but the element in group 2 is unmatched (contradiction).

Proof: *The algorithm produces a stable matching.*

Assume, for the sake of contradiction, that there is not a stable matching.

- Then, there is an element x in group 1 and an element y in group 2 such that y is higher in x's ranking than x's current match and x is higher in y's ranking than y's current match.
    - *Case 1*: x has not already asked y. This is not possible: that x has matched with an element in group 2 of lower ranking than y would mean x has already asked every element in group 2 ranked higher than its current match, including y, per the algorithm (contradiction).
    - *Case 2*: x already asked y. This would mean y matched with a higher-ranked element in group 1. x is higher in y's ranking than y's; therefore y moved from a higher-ranked match to a lower-ranked match (its current match). But a characteristic of the algorithm is that an element in group 2 will never move from a higher-ranked match to a lower-ranked match (contradiction).

Implementation

Store both groups as linked lists

- Store group 1 as a linked list, and advance one element whenever it is matched; if an element is unmatched, insert it back into the list
    - Store priority rankings as an array for each element
        - Since priority ranking is never backtracked, can be made to “search” (find next not asked) in constant time
- Searching group 2’s priority ranking: store as vector (linear search -> O(n); overall O(n^3)) or pre-compute and store as hash map: mapping number -> rank (O(n^2) before iterations begin; overall O(n^2)+O(n^2))

Time Complexity

- Best case: n asks
- Worst case: n^2 “asks” (proof of termination)

10/5/23

- Big O: ignore constant terms; just consider asymptotic behavior
- Proving big O: given big O behavior of # of iterations (e.g. O(n)), must also prove each iteration runs in constant time to determine the behavior of the overall algorithm
    - E.g. # of iterations: O(n), length/time of iterations: O(n) -> overall: O(n^2)
- Steps of Gale-Shapely:
    - (1) Identify free element in group 1 -> (2) identify highest rank not yet approached -> (3) for each potential match, determine if it already has a match (-> (4) if so, search its priority ranking to determine if the new element is higher priority than the current match; otherwise, match)
        - Write algorithms & proofs in bullet points; don’t make it too complicated
    - Memory complexity

Majority Problem

Problem: Given a total of n votes for m candidates, we want to find whether any single candidate has a majority (strictly more than half the votes).

- Votes presented in the form of an n-length array of numbers 1, 2, …, m
- Condition: constant amount of extra storage

Approach (Problem Reduction):

A given candidate m must have at least $\frac{n}{2}$ votes in the original array to win. If we remove 2 votes from the array (one vote for m, and one vote not for m), then m needs to have at least $\frac{n}{2} - 1$ votes in the new (n-2)-length array

- Any other candidate (aside from the other candidate whose vote was removed) still needs $\frac{n}{2}$ votes in the new array

Algorithm:

1. Take the 1st element (vote) as our temporary majority candidate, and initialize its vote counter to be 1.
2. Advance a pointer through the array. At each element:
    a. *Case 1*: The element is the same as the temporary majority. Then increment the majority vote counter by 1.
    b. *Case 2*: The element is different from the temporary majority. Then decrement the majority candidate vote counter by 1.
        i. Analogous to implicitly removing one element of the temporary majority + the just-found element
    c. If the majority vote counter reaches 0, there is no current majority; set the temporary majority to be the next element and continue
3. When the loop finishes:

a. *Case 1*: The temporary majority is nonzero, and has at least 1 vote. Then that specific element is the only one that might have a majority; rescan all the votes to determine if that element has a majority.
b. *Case 2*: The temporary majority is null. Then there is no majority.

Implementation

Only 4 pieces of extra storage needed:

1. A pointer to start of array
2. Our current index in the array
3. One variable denoting the [temporary] majority candidate
4. One variable containing the # of votes for that majority candidate

10/6/23 [Disc]

- Famous person problem: given n people, define a famous person to be a person who does not know anyone, but is known by all other people (where “know” is a one-way relation). Design an algorithm to find the famous person.
    - Paradigm - find a way to eliminate vertices

10/10/23 [missed]

Graphs

- Matrix vs linked-list representations of graphs
    - Matrix representations for dense graphs
        - Edge weights set to 0 or 1 for unweighted graphs; continuous values for weighted graphs
    - Linked list representations for sparse graphs
- Graphs
    - Notation: n nodes, m edges
    - Paths, cycles, connectivity
    - Path - a sequence of nodes/vertices $v_1, ..., v_k$ of a graph, where for any v_n, v_{n+1} there exists an edge (v_n, v_{n+1})
        - Simple path - a path where all vertices are distinct
        - Cycle - a path where the endpoint is the same as the start point
    - Connected - a graph is connected if, for any two vertices in the graph, there exists a path between them
        - Strongly connected - a digraph is connected if, for any two vertices in the graph, there exists a path from either node to the other
    - The distance between two nodes is defined as the length of the shortest path between them
- Tree - a graph is a tree if it is connected and does not contain a cycle

    - Nodes are referred to as descendants of a root node, with parent/child relationships
    - An n-node tree has exactly n-1 edges
    - Is the minimum network without a cycle
- Graph connectivity
    - S-t Connectivity Problem: the problem of determining, for a pair of nodes s and t, whether there exists a path between s and t
- Bipartite graph - a graph where the nodes can be partitioned into two sets X and Y, where every edge in the graph is from a node in X to a node in Y
    - Bipartite graphs cannot have odd cycles
    - Coloring (determining bipartiteness): color a node red, color its neighbors blue, color their neighbors red, etc.; once finished, the graph is bipartite if there is no edge between nodes of the same color
        - Can be implemented with BFS (O(m+n) [m nodes, n edges])
- Directed graphs
    - Mutually reachable - two nodes in a graph are mutually reachable if there exists a path from each to the other
- Directed acrylic graph (DAG) - directed graph with no cycles
    - Topological ordering - an ordering of nodes v_1, v_2, … such that for any edge v_i to v_j, j>1
        - If a digraph has a topological ordering, then it is a DAG; a DAG must have at least one node with no incoming edges
- Complexity of representing graph with n vertices, e edges: O(n+e)
- Weighted graphs - edges with higher weights called larger/heavier
- Directed vs undirected graphs
    - Similar in terms of definition: main difference is definition of adjacency/neighboring vertices
    - Only considering simple paths

Traversal: BFS & DFS

Two main algorithms for traversing unweighted graphs: **BFS** and **DFS**

Breadth-first search (BFS): *"Explore things that are close together first"*

- Begin with a root node
- Upon visiting a node: look at all of its neighbors (that have not already been looked at); once all neighbors of the current node have been explored, move to a different neighboring node
    - From the root node (called *layer 0*): look at all neighbors (*layer 1*), then neighbors of neighbors (*layer 2*), and so on
        - Layer of a node is equal to the distance [length of the shortest path] from that node to the root node
            - Can be proven
- Call any edge used when first visiting a node, a tree edge
    - Any edge not a tree edge will create a cycle in the graph if included
    - End result of BFS is called a BFS tree (not necessarily binary)
- Multiple components - once one component has been fully explored, jump to a root node of another component

Depth-first search (DFS): *"Explore everything in one direction first"*

- Upon visiting a node: pick a neighboring node that has not already been visited (selection can be arbitrary) and visit it
    - Continue doing so until we reach a node with no unvisited neighbors, in which case we backtrack until we reach a node with unvisited neighbors
- When DFS concludes: the graph composed of all the nodes, containing only the edges that were taken in the DFS traversal, is a DFS tree
    - Any edges in the graph not in the DFS tree form a cycle with the edges of the DFS tree

BFS vs DFS ("orthogonal algorithms")

- Breadth-first search good for finding distances
- Depth-first search good for finding cycles

Time Complexity

- BFS:
    - For a graph with N nodes, for each node, at most (N-1) neighbors to search through
    - Overall: O(n^2); more specifically O(E) [E=number of edges]
        - Graph with N connected components: O(N+E) [max: N connected components = n vertices -> O(n+e), linear in e and n]
            - Most number of edges: every node connected -> n^2 edges -> O(n^2)
    - Same complexity produces both exponential and linear complexity (different methods of accounting)
        - O(m+n) - linear in terms of input size
- DFS:
    - Same deal: O(n^2), O(n+e), O(n!)

Implementation

- BFS: store nodes-to-visit in a FIFO queue
- DFS: store neighbors of current node in a FILO stack

Graph Coloring

Bipartite graph - a graph where the nodes can be partitioned into two sets X and Y, where every edge in the graph is from a node in X to a node in Y

- Assumed undirected, connected
- Bipartite graph - no odd cycles
- Equivalent term - graph is 2 colorable
- Individual components of a graph may each be bipartite/not bipartite

Problem: Given graph G=(V,E) with one component, determine whether G is bipartite
Approach (Coloring): Assign each vertex a color (either red or blue).

- Given a red vertex, then all of its neighbors must be blue (and vice versa) if the graph is bipartite

Algorithm

1. Assign a random node to be the root node of a BFS, and color it red.
2. Use BFS to generate a BFS tree of the graph.
3. Per our coloring algorithm, any odd level of the tree will be blue, and any even level of the tree will be red.
    a. The graph is bipartite if and only if there are no edges between two nodes of the same color (i.e. in the same layer).

Time Complexity
BFS: O(m+n)

Strong Connectedness

Definition: A digraph G is called "*strongly connected*" if, for any two nodes u and v in G, there exists a path from each to the other

Problem: Determine if a digraph G is strongly connected.

Algorithm

Let G be a directed graph.

1. Pick an arbitrary vertex s; from s, run BFS on G. If every node is included in the resulting BFS tree, then there is a path from s to every node in the tree.
2. From s, run BFS on the graph Grev (G reverse = G with the direction of edges reversed). If every node is included in the resulting BFS tree, then there is a path from every node in the tree to s.
3. If there is a path from s to every node, and a path from every node to s, then G is strongly connected.

Topological Sorting

Problem: Given a directed graph G(V, E), give an ordering of the vertices such that every edge is from a node higher in the ordering to a node later in the input

- Solution space: a given ordering for a graph is not always unique
    - Can have anywhere between a unique [graph is a single path] to an exponential number of possible orderings
- A topological ordering only exists if there is no cycle in the graph: graph is a directed acyclic graph (DAG)

Applications:

- Assigning registers in a computer - need a topological sorting to determine order of assignment if register values are interdependent
- Construction: tasks may be ordered via topological ordering (some things, e.g. permitting, may need to happen before others)

Algorithm

1. Initialize an empty topological ordering.
2. While not all nodes have been added to the ordering:
    a. Pick any node with no incoming edges [a source], and add it as the first node in our ordering.
        i. Number of incoming edges = in-degree; number of outgoing edges = out-degree; out-degree = 0 [sink]
        ii. Any source node can be the first node in the graph (since for each of them, there is no node that must be before them)
            1. There exists at least one source in a DAG
    b. Delete the just-added node from the graph, and any edges including it)
        i. Graph is still a DAG, since we did not add any edges, only deleted them; therefore, it still has a source

c. Add any source node in the new graph, and add it as the next node in our ordering; delete this node (and any edges including it) from the graph

Implementation

- Determining in-degree: loop through edges, increment in-degree of nodes for each edge
    - $O(e)=O(n(n+1))=O(n^2)$ [different accounting methods]
- Finding a source: find all sources by looping through all vertices; store as queue/stack
- Pop a source, add to ordering, delete from graph
    - Each source is processed one time; total of $O(n)$
    - Remove all edges including (i.e. originating from) the source, and decrement the in-degree of the node on the other side of the edge; if this causes a node's in-degree to become 0, add it to the queue/stack
        - Each edge is processed one time; step runs in a total of $O(e)$
- Continue until the queue/stack is empty
- Total runtime: $O(n+e)$
- Algorithm works regardless of number of connected components

Greedy Paradigm

- Graph as an adjacency list
- Greedy paradigm for local optimization - making simplifying assumptions about a problem/solution space (without exploring the entire space) for speed and efficiency
    - Assumptions carried forward to a solution, that may or may not be globally optimal
    - May require a proof of global optimality to generalize the solution

Interval Scheduling

- Interval scheduling - greedy plane sweep (problem is geometric -> geometric solution)
    - Left to right scan: at each stage, take the interval that ends the earliest (remove all overlapping intervals) and continue
    - Proof of optimality:
        - Stay-ahead: given a solution set from the greedy algorithm and a solution set not from the greedy algorithm, prove that the first k intervals from the greedy solution set end at the same time at, or before, the first k intervals from the other solution set for all k (prove by induction)
    - Implementation: one array sorting intervals by end time (O(nlogn) - heapsort, e.g.), one array sorting intervals by start time

10/24/23

- Articulation point - a vertex in a connected graph such that removing it disconnects the graph
    - Can be found via BFS, BFS tree
- Shortest path problems
    - BFS for finding minimum distance fails for graphs with weighted edges (i.e. where not all edges are of unit length)

Shortest Path Problem

- **Problem:** Given a graph with weighted edges, find the minimum weight path from a vertex a to vertex b
  - Also called “minimum-weight path”
- Assumption: all weights non-negative (otherwise, algorithm fails)
  - Algorithms do exist for negative weights, but not negative cycles

Algorithm (Dijkstra’s)

Assume all weights are non-negative; (otherwise, the algorithm fails.

1. Start at vertex a.
2. Find the neighbor x of a that is closest to a. Then the shortest path $a \rightarrow x$ is the edge from a to x.
    a. Assign $d(a, x)$ to be the length of this edge.
3. Find next-closest vertex y to a. This vertex will be either a neighbor of a or a neighbor of x.
    a. Fix $d(a, y)$ as the length of the edge (a, y), or the length of the edge (x, y) plus the distance from x to a. (Or the minimum of the two). Then $d(a, y)$ is the weight of the minimum-weight path $a \rightarrow y$.
4. Continue this process until the distance from all vertices to a has been determined.
    a. Represent a vertex v as belonging to one of three categories:
        i. *Processed*: A minimum-weight path $a \rightarrow v$ has been found.
        ii. *Intermediate*: A path $a \rightarrow v$ has been found.
        iii. *Unprocessed*: v has not yet been seen.

Implementation

Two implementations for storing processed/intermediate vertices

1. Implementation 1: Store intermediate vertices in an array
    a. N steps, at most $(N - 1)$ vertices modified per step -> O(N^2)
2. Implementation 2: Store every intermediate vertex in a heap

    a. O(1) to find the minimum-weight vertex
    b. O($\log N$) time for insertion, maximum E insertions -> O($E \log N$)

- If e leq n^2/logn, pick elogn; else, pick n^2

Minimum Spanning Tree

Definition: A ***subgraph*** of a graph G is a subset of the nodes/edges of G

- A subgraph is called a ***tree subgraph*** if it has a tree structure
    - Tree subgraphs always contain (n-1) edges

Definition: A subgraph is called a ***spanning tree*** if it is a tree subgraph, is connected, and touches every vertex in the graph

- Spanning trees are the minimum network of edges in the graph without cycles; any network with a cycle can be reduced to a spanning tree without one
- BFS, DFS trees are themselves spanning trees

Definition: A ***minimum spanning tree (MST)*** of a graph is the spanning tree of minimum total weight

Problem: Given a graph G, find a minimum spanning tree of G.

MST Theorem

Theorem: For every MST on the graph, for any partition of G into two sets, the edge of minimum weight between partitions will be included in the MST.

Proof (MST Theorem):

Begin with an arbitrary MST and an arbitrary partition. Assume, for the sake of contradiction, that the edge of minimum weight between partitions is not in the MST.

- Let the edge of minimum weight be between vertices v and w.
- Then there is a path between v and w in the MST, and that path must take an edge of larger weight between partitions.
- Then the MST would be of smaller total weight if we replaced that edge with the edge (v, w).
- Then the MST is not the smallest spanning tree.

Algorithm (Prim's)

1. Partition the vertices of G into two sets, such that one set (left) contains only 1 vertex and the other set (right) contains the other (n-1) vertices.
2. Take the minimum edge between the two sets and add it to our subgraph.
    a. If there are two minimum edges of equal weight, pick only one.
3. The minimum edge is between a vertex in the left set and a vertex in the right set. Move the vertex in the right set to the left set.
4. Take the minimum edge between the two sets and add it to our subgraph. Move the right-side vertex on the edge, to the left set.
5. Continue until (n-1) edges have been added.

Prim's Algorithm

- Requires no proof of correctness (relies only on MST Theorem proof)
- MST returned is unique iff we make the assumption of unique edge weights
- Runtime identical to Dijkstra's: $O(N^2)$ / $O(E \log N)$

Algorithm (Kruskal's)

1. Sort all edges by weight
2. Add the shortest edge e_1 to the graph
    a. Let e_1 = (v, w). If we take a partition that takes v as one side, and all other nodes (including w) on the other; then the e_1 is the shortest between partitions, and therefore can be added per MST theorem
3. Add the next-shortest edge:
    a. *Case 1*: The edge is between two nodes (i, j) not in e_1. Then, if we take a partition that takes i as one side, and all other nodes (including j, e_1) as the other; then e_2 is the shortest between partitions, and therefore can be added.
    b. *Case 2*: The edge is between two nodes (i, v), where v is in e_1. Then, if we take a partition that takes i as one side, and all other nodes (including v, e_1) as the

other; then e_2 is the shortest between partitions, and therefore can be added.

4. Continue looking at next-shortest edges e_k until (n-1) edges have been added:
    a. *Case 1*: The edge e_k is between two nodes (a, b), where at least one of a, b has not already been added to our graph. Then we can draw a partition with e_k as the minimum edge, therefore we can add e_k to our graph.
    b. *Case 2*: The edge e_k is between two nodes (a, b) that are both already in our graph (i.e. e_k creates a cycle. Then there is no way to draw a partition with e_k as the minimum edge, since every other edge in the cycle is shorter than e_k; therefore we do not add it to our graph.

Time Complexity

- Kruskal via arrays: O(e) -> O(n^2)

<u>Union-Find</u>

Problem (Union-Find): Given n elements partitioned into k sets, we want to be able to perform two operations efficiently:

1. *<u>Find</u>*: given two elements, determine if they are in the same set
2. *<u>Union</u>*: given two sets, replace them with their union

Solution: represent each set as a *<u>rooted tree</u>*

- Convention: call each tree by the name of its root
- Operations:
    - *Find*: visit both elements; they are in the same set if and only if the respective roots of their trees are the same
    - *Union*: we can simply append one tree to the end of another
        - Keep track of first, last elements of each tree -> O(1)

Rooted Tree Configurations:

- Storing each tree as a path [a la linked list] - O(n) find, O(1) union
- Storing each tree by having all elements in the tree point toward a single root (all paths of length 1) - O(1) find, O(n) union
- Want a balanced tree, i.e. where tree height is logarithm of vertices in the tree
    - Deeper paths than star-shaped configuration; shorter paths than straight-line configuration
    - Find: O(log n) [maximum path length is height= log n]
    - Union: O(1)
        - Given two rooted trees, simply change the root of the shorter tree to point toward the root of the taller tree
            - Trees of unequal height: Height of new tree will just be height of taller tree
            - Trees of equal height: height of new tree will be height before union + 1 (still logarithmic relative to # vertices)

Kruskal's (Union-Find)

- Start by placing each vertex (in Kruskal) in its own Union-Find set of height 1
    - Union-Find sets as rooted trees
- When an edge is added to the MST, perform a Union of the trees of the elements on each side of the edge.
    - Union-Find helps keep track of connected components
- To check edges are not creating cycles:
    - An edge creates a cycle only when: both nodes on the cycle are connected by a path <=> nodes in the same rooted tree (perform a Find to verify)

Time Complexity

- Kruskal (Union-Find): O(m log m)
    - O(m log m) to sort all m edges
    - O(m log n) for Kruskal Union-Find loop
        - m edges; 2 Finds (log(n) each) per edge

Notes (MSTs)

- 3rd algorithm - ***<u>Reverse deletion algorithm</u>***
    - Sort the edges of the graph in descending order. Loop through the edges: at each edge, check if we can delete the edge without disconnecting the graph (if so, we delete the edge).
    - Relies on MST Theorem
- Unique edges: our algorithms fail if edge weights are not unique
    - Solution: if two edges have the same weight, add a small epsilon to one so they become unique (epsilon = 1/E, e.g.)
        - May result in different MSTs depending on which edge is chosen; MSTs only unique if edge weights are all unique
    - May be an additional step: add epsilon, 2epsilon, 3epsilon to first, second, third vertex, etc. (O(n))

<u>Divide and Conquer</u>

Divide and Conquer: *If a problem is too difficult to solve directly, partition it into sub-problems*

- Works for any problem, though it may not result in the most optimal solution

Merge Sort

Sorting: merge sort (recursive partitioning)

Algorithm

Given an array to sort, that is too long to sort directly:

1. Divide the array into some number of subarrays of equal size.
2. Divide subarrays into even smaller subarrays as needed. Keep dividing until the divided subarrays are small enough to sort directly.
3. Sort all subarrays.
4. *Merging step*: Once each subarray is sorted, then we can keep combining (*merge*) our subarrays into larger subarray until we have returned to the original array.
5. Combination: Construct new subarray by continuously picking the minimum of the next element for each subarray
    a. Let p, q = size of two subarrays -> combined to size p+q
    b. Each element of the output array is added in O(1) -> total runtime O(p+q)
        i. Alt: each element of an input array is seen at most q times -> O(pq)

Time Complexity

- Runtime: time to sort n elements T(n) = 2T(n/2)+O(n); T(1) = O(1)
    - -> T(n) = 2T(n/2)+Cn [for some constant C - time requirement of merging]
    - -> $T(n) = 2(2T(n/4)+Cn/2)+Cn=2^2T(n/2^2)+2Cn$
    - -> $T(n)=2^iT(n/2^i)+iCn$
    - -> We divide recurse $\log_2(n)$ times until we reach T(1) [$n/2^i=1$]
    - -> $T(n)=2^{log\_2(n)}T(n/2^{log\_2(n)})+2Cn\log_2(n)$
    - -> T(n) = O(n log(n))

Misc

- Master theorem
    - When T(n) <= qT(n/2)+cn, T(2)<=c -> p(n)=O(nlogq) when q>2

- Lower bound for general sorting (elements in random order, not bounded): linear time (look at every element once)
    - Eventual proof: O(NlogN) is optimal

Counting Inversions

Crossing Problem: given two parallel lines between which there are total n lines (potentially overlapping), find an algorithm to determine the number of crossings

- Trivial: look at every pair of lines; if the start & end ranges of the lines overlap, then they cross (n^2 total possible pairs)

Solution

Assign each line endpoint on one side numerical labels 1,2,...

Assign each endpoint on the other side, the label assigned on the first side

If any two numbers are out of order on the other side, then the two lines cross; then the number of crossings is equal to the number of out-of-order indices (*inversions*).

Solution: Count the number of inversions

Algorithm

1. Divide the sequence into subsequences
2. Perform merge sort; at every merge, keep track of number of crossings
    a. *Case 1*: # on right larger than # on left -> no crossing (for the current left number, and any smaller left numbers)
    b. *Case 2*: # on right smaller than # on left -> # crossed current left number, and every larger left number
        i. Can compute the number of crossings from the current right number in constant time

Time Complexity

Runtime: O(NlogN) [merge sort]

<u>Closest Pair Problem</u>

Problem: given a n set of points in R^i, want to find the closest pair of points

- Closest pair using Euclidean/L1 norm
    - Alternate metrics: Manhattan/L1: horizontal + vertical displacement

- Know: solution between O(n) [constant time per point] vs $O(n^2)$ [see all possible pairs of points]
    - 1D: sort points by coordinate in some dimension (e.g. x-axis)
        - Compare distance between each adjacent pair of points to find local minimum distance (O(N log N) for sorting, O(N) for comparison)
    - Multiple dimensions: more difficult
        - Greedy attempts (e.g. projection onto axes, polar coordinates) fail
    - Divide and conquer: divide points into partitions
        - Closest points will either be closest points in one of the partitions, or be two points in different partitions