Data Structures and Algorithms

Session 27. May 4th, 2009

Instructor: Bert Huang

http://www.cs.columbia.edu/~bert/courses/3137

Annoucements and Today's Plan

- Final Exam Wednesday May 13th, 1:10 PM - 4 PM Mudd 633
- Course evaluation
- Review 2nd half of semester
- Lots of slides, I'll go fast but ask questions if you have them

Final Topics Overview

- Big-Oh definitions (Omega, Theta)
- Arraylists/Linked Lists
- Stacks/Queues
- Binary Search Trees: AVL, Splay
- Tries
- Heaps
- Huffman Coding Trees
- Hash Tables: Separate Chaining, Probing
- Graphs: Topological Sort, Shortest Path, Max-Flow, Min Spanning Tree, Euler
- Complexity Classes
- Disjoint Sets
- Sorting: Insertion Sort, Shell Sort, Merge Sort, Quick Sort, Radix Sort, Quick Select

Big Oh Definitions

- For *N* greater than some constant, we have the following definitions:

$$
\begin{aligned}
T(N) &= O(f(N)) \leftarrow T(N) \leq cf(N) \\
T(N) &= \Omega(g(N)) \leftarrow T(N) \geq cf(N) \\
T(N) &= \Theta(h(N)) \leftarrow \begin{array}{c} T(N) = O(h(N)), \\ T(N) = \Omega(h(N)) \end{array}
\end{aligned}
$$

- There exists some constant c such that cf(N) bounds T(N)

Big Oh Definitions

- Alternately, O(f(N)) can be thought of as meaning

$$T(N) = O(f(N)) \leftarrow \lim_{N \to \infty} f(N) \geq \lim_{N \to \infty} T(N)$$

- Big-Oh notation is also referred to as **asymptotic** analysis, for this reason.

Huffman's Algorithm

- Compute character frequencies
- Create forest of 1-node trees for all the characters.
- Let the **weight** of the trees be the sum of the frequencies of its leaves
- Repeat until forest is a single tree:
  Merge the two trees with minimum weight.
  Merging sums the weights.

Huffman Details

- We can manage the forest with a priority queue:
- **buildHeap** first,
  - find the least weight trees with 2 **deleteMin**s,
  - after merging, **insert** back to heap.
- In practice, also have to store coding tree, but the payoff comes when we compress larger strings

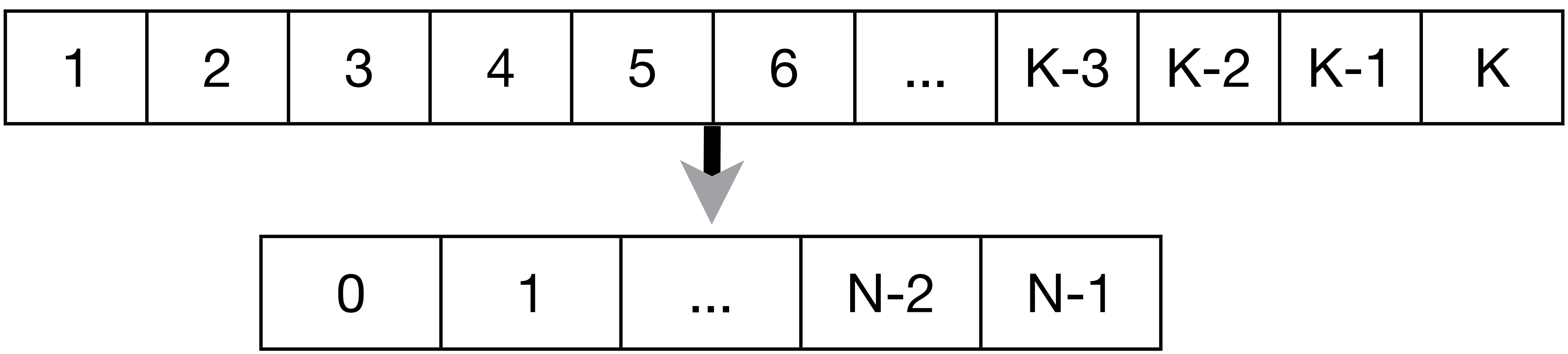
Hash Table ADT

- Insert or delete objects by **key**
- Search for objects by **key**
- **No** order information whatsoever

- Ideally O(1) per operation

Hash Functions

- A **hash function** maps any key to a valid array position
  - Array positions range from 0 to N-1
  - Key range possibly unlimited



Hash Functions

- For integer keys, (key mod N) is the simplest hash function
- In general, **any** function that maps from the space of keys to the space of array indices is valid
- but a good hash function spreads the data out evenly in the array;
- A good hash function avoids **collisions**

Collisions

- A **collision** is when two distinct keys map to the same array index
  - e.g., $h(x) = x \bmod 5$
    $h(7) = 2$, $h(12) = 2$
- Choose $h(x)$ to minimize collisions, but collisions are inevitable
- To implement a hash table, we must decide on collision resolution policy

Collision Resolution

- Two basic strategies
  - Strategy 1: Separate Chaining
  - Strategy 2: Probing; lots of variants

Strategy 1: Separate Chaining

- Keep a list at each array entry
  - Insert(x): find h(x), add to list at h(x)
  - Delete(x): find h(x), search list at h(x) for x, delete
  - Search(x): find h(x), search list at h(x)
- We could use a BST or other ADT, but if h(x) is a good hash function, it won’t be worth the overhead

Strategy 2: Probing

- If h(x) is occupied, try **h(x)+f(i) mod N** for i = 1 until an empty slot is found
- Many ways to choose a good f(i)
- Simplest method: Linear Probing
  - f(i) = i

Primary Clustering

- If there are many collisions, blocks of occupied cells form: **primary clustering**
- Any hash value inside the cluster adds to the end of that cluster
- (a) it becomes more likely that the next hash value will collide with the cluster, and (b) collisions in the cluster get more expensive

Quadratic Probing

- f(i) = i^2
- Avoids primary clustering
- Sometimes will never find an empty slot even if table isn't full!
- Luckily, if load factor $\lambda \leq \frac{1}{2}$, guaranteed to find empty slot

Double Hashing

- If $h_1(x)$ is occupied, probe according to

$$f(i) = i \times h_2(x)$$

- 2nd hash function must never map to 0

- Increments differently depending on the key

Rehashing

- Like ArrayLists, we have to guess the number of elements we need to insert into a hash table
- Whatever our collision policy is, the hash table becomes inefficient when load factor is too high.
- To alleviate load, **rehash**:
  - create larger table, scan current table, insert items into new table using new hash function

Graph Terminology

- A **graph** is a set of **nodes** and **edges**
  - nodes aka vertices
  - edges aka arcs, links
- Edges exist between pairs of nodes
  - if nodes x and y share an edge, they are **adjacent**

Graph Terminology

- Edges may have **weights** associated with them
- Edges may be **directed** or **undirected**
- A **path** is a series of adjacent vertices
  - the **length** of a path is the sum of the edge weights along the path (1 if unweighted)
- A **cycle** is a path that starts and ends on a node

Graph Properties

- An undirected graph with no cycles is a tree
- A directed graph with no cycles is a special class called a **directed acyclic graph (DAG)**
- In a **connected** graph, a path exists between every pair of vertices
- A **complete** graph has an edge between every pair of vertices

Implementation

- Option 1:
    - Store all nodes in an indexed list
    - Represent edges with **adjacency matrix**
- Option 2:
    - Explicitly store **adjacency lists**

Topological Sort

- Problem definition:
  - Given a directed acyclic graph G, order the nodes such that for each edge $(v_i, v_j) \in E$, v_i is before v_j in the ordering.
- e.g., scheduling errands when some tasks depend on other tasks being completed.

Topological Sort Better Algorithm

- 1. Compute all indegrees
- 2. Put all indegree 0 nodes into a Collection
- 3. Print and remove a node from Collection
- 4. Decrement indegrees of the node's neighbors.
- 5. If any neighbor has indegree 0, place in Collection. Go to 3.

Topological Sort Running time

- Initial indegree computation: O(|E|)
  - Unless we update indegree as we build graph
- |V| nodes must be enqueued/dequeued
- Dequeue requires operation for outgoing edges
- Each edge is used, but never repeated
- Total running time O(|V| + |E|)

Shortest Path

- Given **G = (V,E)**, and a node **s** $\in$ **V**, find the shortest (weighted) path from **s** to every other vertex in **G**.
- Motivating example: subway travel
  - Nodes are junctions, transfer locations
  - Edge weights are estimated time of travel

Breadth First Search

- Like a level-order traversal
- Find all adjacent nodes (level 1)
- Find *new* nodes adjacent to level 1 nodes (level 2)
- ... and so on
- We can implement this with a queue

Unweighted Shortest Path Algorithm

- Set node s' distance to 0 and enqueue s.
- Then repeat the following:
  - Dequeue node **v**. For unset neighbor **u**:
    - set neighbor **u**'s distance to **v**'s distance +1
    - mark that we reached **v** from **u**
    - enqueue **u**

Weighted Shortest Path

- The problem becomes more difficult when edges have different weights
- Weights represent different costs on using that edge
- Standard algorithm is **Dijkstra's Algorithm**

Dijkstra’s Algorithm

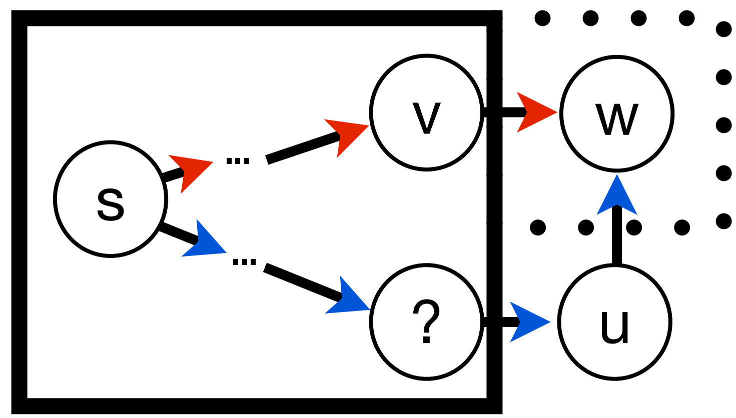
- Keep distance overestimates **D(v)** for each node **v** (all non-source nodes are initially infinite)
- 1. Choose node **v** with smallest *unknown* distance
- 2. Declare that **v**’s shortest distance is *known*
- 3. Update distance estimates for neighbors

Updating Distances

- For each of **v**'s neighbors, **w**,
- if min(**D(v)+ weight(v,w)**, **D(w)**)
  - i.e., update **D(w)** if the path going through **v** is cheaper than the best path so far to **w**

Proof by Contradiction (Sketch)

- Contradiction: Dijkstra's finds a shortest path to node **w** through **v**, but there exists an even shorter path



- This shorter path must pass from inside our known set to outside.
- Call the 1st node in cheaper path outside our set **u**
- The path to **u** must be shorter than the path to **w**
  - But then we would have chosen **u** instead

Computational Cost

- Keep a priority queue of all unknown nodes
- Each stage requires a **deleteMin**, and then some **decreaseKey**s (the # of neighbors of node)
- We call **decreaseKey** once per edge, we call **deleteMin** once per vertex
- Both operations are O(log |V|)
- Total cost: O(|E| log |V| + |V| log |V|) = O(|E| log |V|)

All Pairs Shortest Path

- Dijkstra's Algorithm finds shortest paths from one node to all other nodes
- What about computing shortest paths for all pairs of nodes?
- We can run Dijkstra's |V| times. Total cost: $O(|V|^3)$
- Floyd-Warshall algorithm is often faster in practice (though same asymptotic time)

Recursive Motivation

- Consider the set of numbered nodes **1** through **k**
- The shortest path between any node **i** and **j** using only nodes in the set {**1**, ..., **k**} is the minimum of
  - shortest path from **i** to **j** using nodes {**1**, ..., **k-1**}
  - shortest path from **i** to **j** using node **k**
- path(i,j,k) = min(path(i,j,k-1),
  path(i,k,k-1)+ path(k,j,k-1))

Dynamic Programming

- Instead of repeatedly computing recursive calls, store lookup table
- To compute path(i,j,k) for any i,j, we only need to look up path(-,-, k-1)
  - but never k-2, k-3, etc.
- We can incrementally compute the path matrix for k=0, then use it to compute for k=1, then k=2...

Floyd-Warshall Code

- Initialize `d` = weight matrix
-
```
for (k=0; k<N; k++)
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      if (d[i][j] > d[i][k]+d[k][j])
        d[i][j] = d[i][k] + d[k][j];
```
- Additionally, we can store the actual path by keeping a “midpoint” matrix

Transitive Closure

- For any nodes i, j, is there a path from i to j?
- Instead of computing shortest paths, just compute Boolean if a path exists
- path(i,j,k) = path(i,j,k-1) OR
  path(i,k,k-1) AND path(k,j,k-1)

Maximum Flow

- Consider a graph representing flow capacity
- Directed graph with **source** and **sink** nodes
- Physical analogy: water pipes
  - Each edge weight represents the **capacity**: how much “water” can run through the pipe from source to sink?

Max Flow Algorithm

- Create 2 copies of original graph: **flow graph** and **residual graph**
  - The flow graph tells us how much flow we have currently on each edge
  - The residual graph tells us how much flow is available on each edge
- Initially, the residual graph is the original graph

Augmenting Path

- Find any path in residual graph from source to sink
  - called an **augmenting path**.
- The minimum weight along path can be added as flow to the flow graph
- But we don't want to commit to this flow; add a reverse-direction undo edge to the residual graph

Running Times

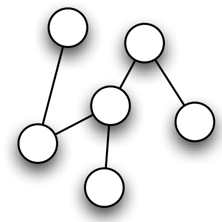
- If integer weights, each augmenting path increases flow by at least 1
- Costs O(|E|) to find an augmenting path
- For max flow f, finding max flow (Floyd-Fulkerson) costs $O(f|E|)$
- Choosing shortest unweighted path (Edmonds-Karp), $O(|V||E|^2)$

Minimum Spanning Tree Problem definition

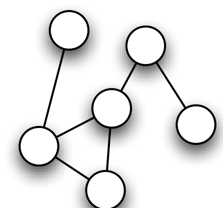
- Given connected graph **G**, find the connected, acyclic subgraph **T** with minimum edge weight
  - A tree that includes every node is called a **spanning tree**
- The method to find the MST is another example of a greedy algorithm

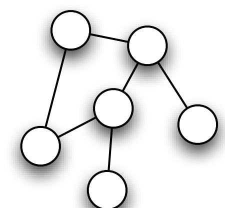
Motivation for Greed

- Consider any spanning tree

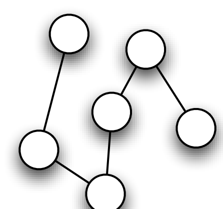


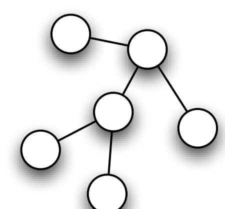
- Adding another edge to the tree creates exactly one cycle





- Removing an edge from that cycle restores the tree structure





Prim's Algorithm

- Grow the tree like Dijkstra's Algorithm
- Dijkstra's: grow the set of vertices to which we know the shortest path
- Prim's: grow the set of vertices we have added to the minimum tree
- Store shortest edge **D**[ ] from each node to tree

Prim's Algorithm

- Start with a single node tree, set distance of adjacent nodes to edge weights, infinite elsewhere
- Repeat until all nodes are in tree:
  - Add the node **v** with shortest known distance
  - Update distances of adjacent nodes **w**:
    D[w] = min(**D**[w], weight(**v**,**w**))

Prim’s Algorithm Justification

- At any point, we can consider the set of nodes in the tree **T** and the set outside the tree **Q**
- Whatever the MST structure of the nodes in **Q**, at least one edge must connect the MSTs of **T** and **Q**
- The greedy edge is just as good structurally as any other edge, and has minimum weight

Prim's Running Time

- Each stage requires one deleteMin O(log |V|), and there are exactly |V| stages
- We update keys for each edge, updating the key costs O(log |V|) (either an insert or a decreaseKey)
- Total time: O(|V| log |V| + |E| log |V|) = O(|E| log |V|)

Kruskal's Algorithm

- Somewhat simpler conceptually, but more challenging to implement
- Algorithm: repeatedly add the shortest edge that does not cause a cycle until no such edges exist
- Each added edge performs a union on two trees; perform unions until there is only one tree
- Need special ADT for unions (Disjoint Set... we'll cover it later)

Kruskal’s Justification

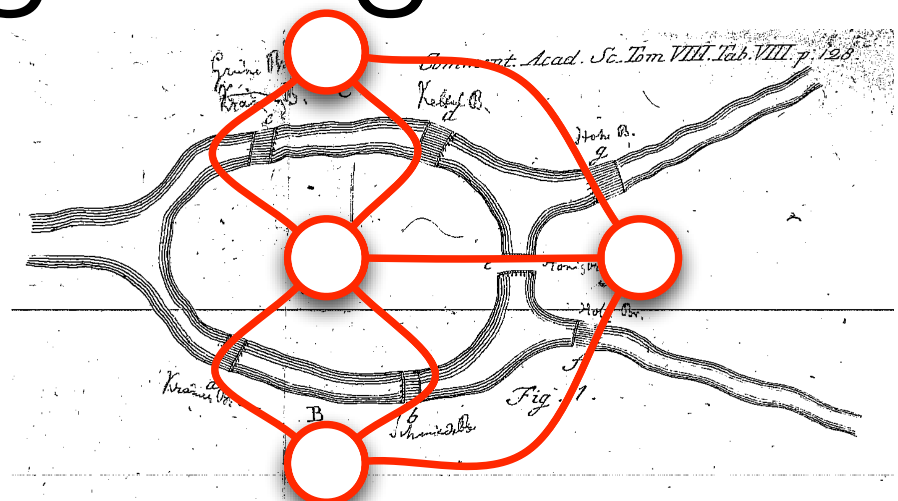
- At each stage, the greedy edge **e** connects two nodes **v** and **w**
- Eventually those two nodes must be connected;
  - we must add an edge to connect trees including **v** and **w**
- We can always use **e** to connect **v** and **w**, which must have less weight since it's the greedy choice

Kruskal's Running Time

- First, buildHeap costs O(|E|)
- In the worst case, we have to call |E| deleteMins
- Total running time O(|E| log |E|); but $|E| \leq | V|^2$

$$O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$$

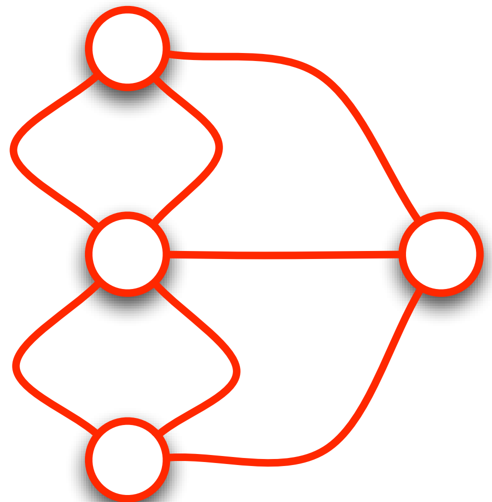
The Seven Bridges of Königsberg



http://math.dartmouth.edu/~euler/docs/originals/E053.pdf

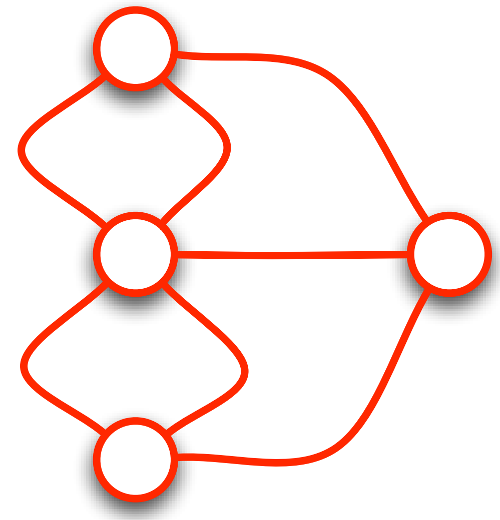
- Königsburg Bridge Problem: can one walk across the seven bridges and never cross the same bridge twice?
- Euler solved the problem by *inventing* graph theory

Euler Paths and Circuits



- Euler path – a (possibly cyclic) path that crosses each edge exactly once
- Euler circuit - an Euler path that starts and ends on the same node

Euler's Proof



- Does an Euler path exist? **No**
- Nodes with an odd degree must either be the start or end of the path
- Only one node in the Königsberg graph has odd degree; the path cannot exist
- What about an Euler circuit?

Finding an Euler Circuit

- Run a partial DFS; search down a path until you need to backtrack (mark edges instead of nodes)
- At this point, you will have found a circuit
- Find first node along the circuit that has unvisited edges; run a DFS starting with that edge
- Splice the new circuit into the main circuit, repeat until all edges are visited

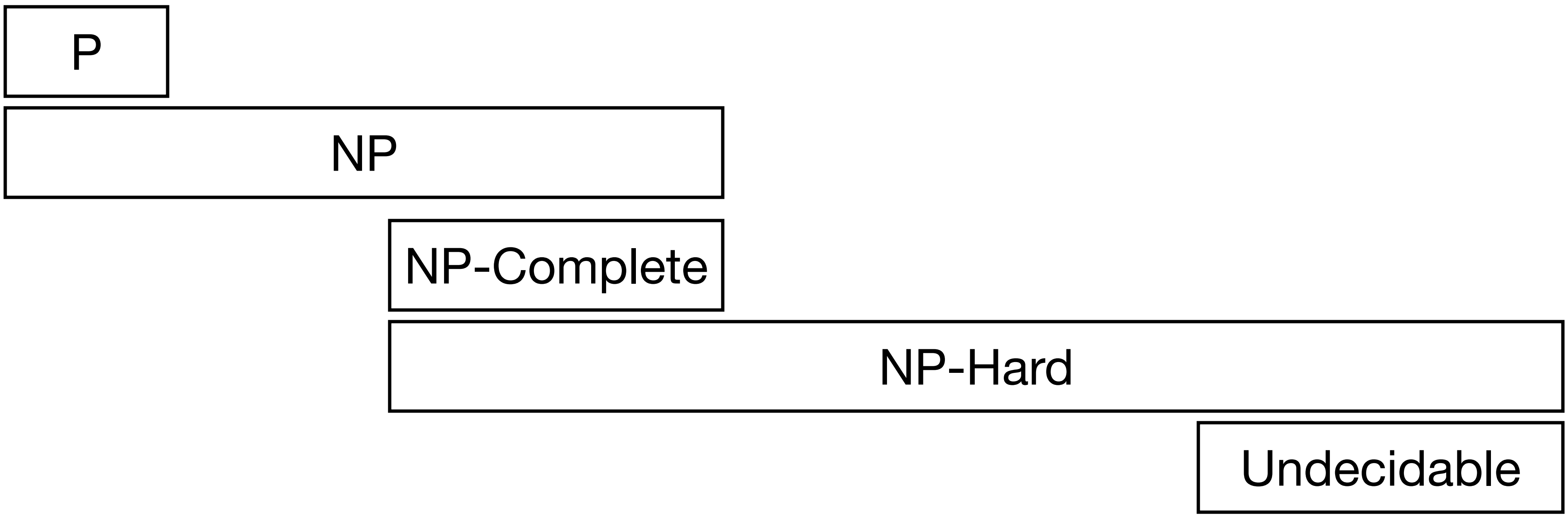
Euler Circuit Running Time

- All our DFS's will visit each edge once, so at least O(|E|)
- Must use a linked list for efficient splicing of path, so searching for a vertex with unused edge can be expensive
- but cleverly saving the last scanned edge in each adjacency list can prevent having to check edges more than once, so also O(|E|)

Complexity Classes

- **P** - solvable in polynomial time
- **NP** - solvable in polynomial time by a nondeterministic computer
  - i.e., you can check a solution in polynomial time
- **NP-complete** - a problem in NP such that any problem in NP is polynomially reducible to it
- **Undecidable** - no algorithm can solve the problem

Probable Complexity Class Hierarchy



Polynomial Time **P**

- All the algorithms we cover in class are solvable in polynomial time
- An algorithm that runs in polynomial time is considered **efficient**
- A problem solvable in polynomial time is considered **tractable**

Nondeterministic Polynomial Time **NP**

- Consider a magical nondeterministic computer
  - infinitely parallel computer
- Equivalently, to solve any problem, check every possible solution in parallel
  - return one that passes the check

NP-Complete

- Special class of NP problems that can be used to solve any other NP problem
- Hamiltonian Path, Satisfiability, Graph Coloring etc.
- NP-Complete problems can be **reduced** to other NP-Complete problems:
  - polynomial time algorithm to convert the input and output of algorithms

NP-Hard

- A problem is NP-Hard if it is at least as complex as all NP-Complete problems
- NP-hard problems may not even be NP

NP-Complete Problems Satisfiability

- Given Boolean expression of N variables, can we set variables to make expression true?
- First NP-Complete proof because Cook's Theorem gave polynomial time procedure to convert any NP problem to a Boolean expression
- I.e., if we have efficient algorithm for Satisfiability, we can efficiently solve any NP problem

NP-Complete Problems
Graph Coloring

- Given a graph is it possible to color with **k** colors all nodes so no adjacent nodes are the same color?
- Coloring countries on a map
- Sudoku is a form of this problem. All squares in a row, column and blocks are connected. **k** = 9

NP-Complete Problems Hamiltonian Path

- Given a graph with N nodes, is there a path that visits each node exactly once?

NP-Hard Problems Traveling Salesman

- Closely related to Hamiltonian Path problem
- Given complete graph **G**, find a path that visits all nodes that costs less than some constant **k**
- If we are able to solve TSP, we can find a Hamiltonian Path; set connected edge weight to constant, disconnected to infinity
  - TSP is NP-hard

Equivalence Relations

- An equivalence relation is a relation operator that observes three properties:
  - **Reflexive**: (a R a), for all a
  - **Symmetric**: (a R b) if and only if (b R a)
  - **Transitive**: (a R b) and (b R c) implies (a R c)
- Put another way, equivalence relations check if operands are in the same **equivalence class**

Equivalence Classes

- Equivalence class: the set of elements that are all related to each other via an equivalence relation
- Due to transitivity, each member can only be a member of one equivalence class
- Thus, equivalence classes are **disjoint sets**
  - Choose any distinct sets S and T, $S \cap T = \emptyset$

Disjoint Set ADT

- Collection of objects, each in an equivalence class
- **find**(x) returns the class of the object
- **union**(x,y) puts x and y in the same class
  - as well as every other relative of x and y
- Even less information than hash; no keys, no ordering

Data Structure

- Store elements in equivalence (general) trees
- Use the tree's root as equivalence class label
- **find** returns root of containing tree
- **union** merges tree
- Since all operations only search up the tree, we can store in an array

Implementation

- Index all objects from 0 to N-1
- Store a parent array such that **s[i]** is the index of **i**'s parent
- If **i** is a root, store the negative size of its tree*
- **find** follows **s[i]** until negative, returns index
- **union**(x,y) points the root of x's tree to the root of y's tree

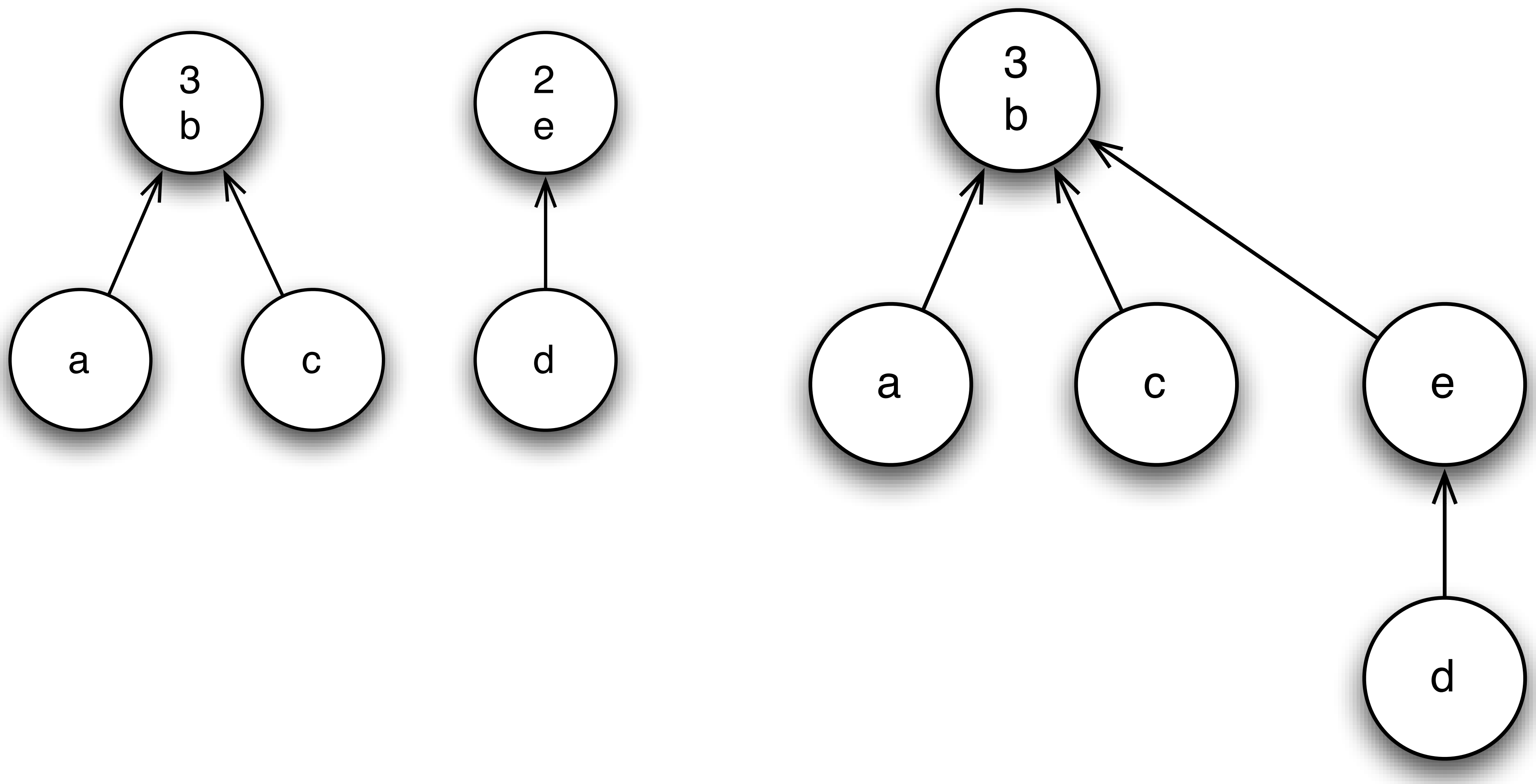
Analysis

- **find** costs the depth of the node
- **union** costs O(1) after **find**ing the roots
- Both operations depend on the height of the tree
- Since these are general trees, the trees can be arbitrarily shallow

Union by Size

- Claim: if we union by pointing the smaller tree to the larger tree's root, the height is at most log N
- Each union increases the depths of nodes in the smaller trees
- Also puts nodes from the smaller tree into a tree at least twice the size
  - We can only double the size log N times

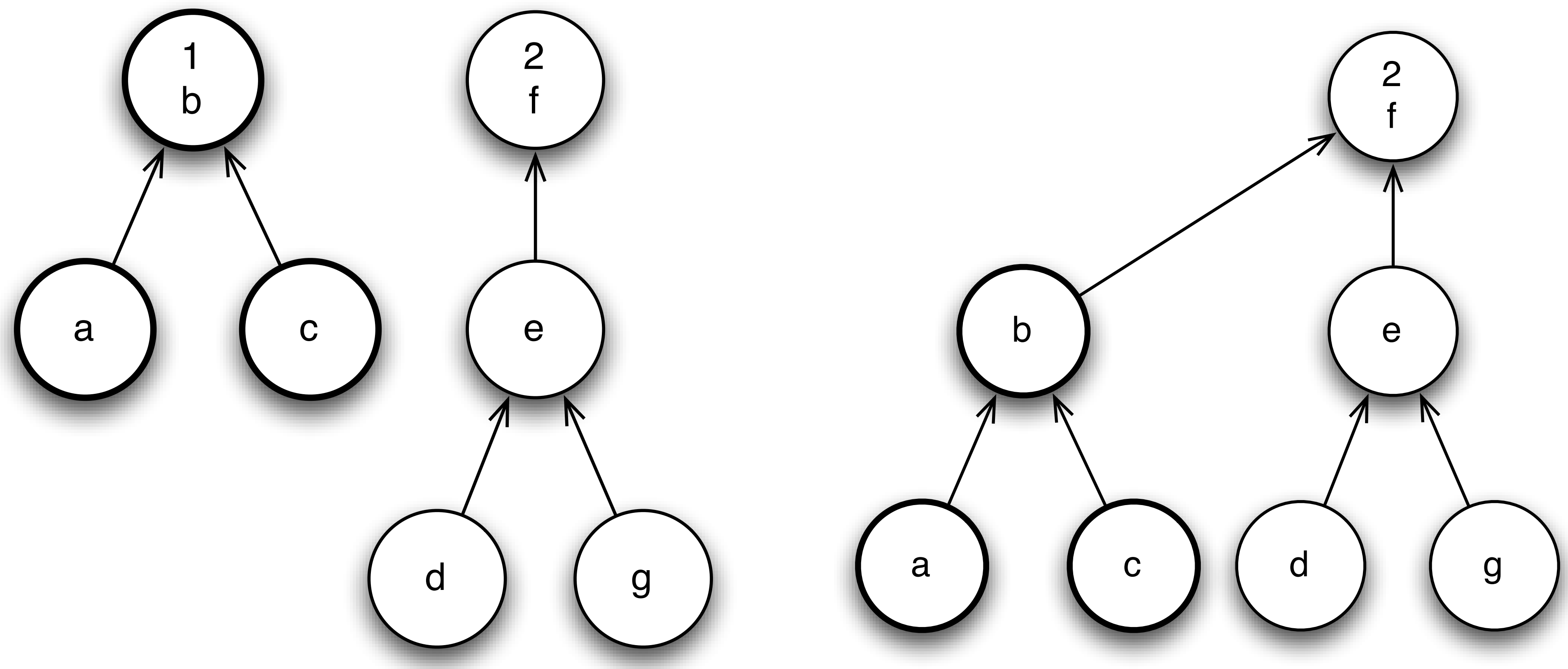
Union by Size Figure



Union by Height

- Similar method, attach the tree with less height to the taller tree
- Shorter tree’s nodes join a tree at least twice the height, overall height only increases if trees are equal height

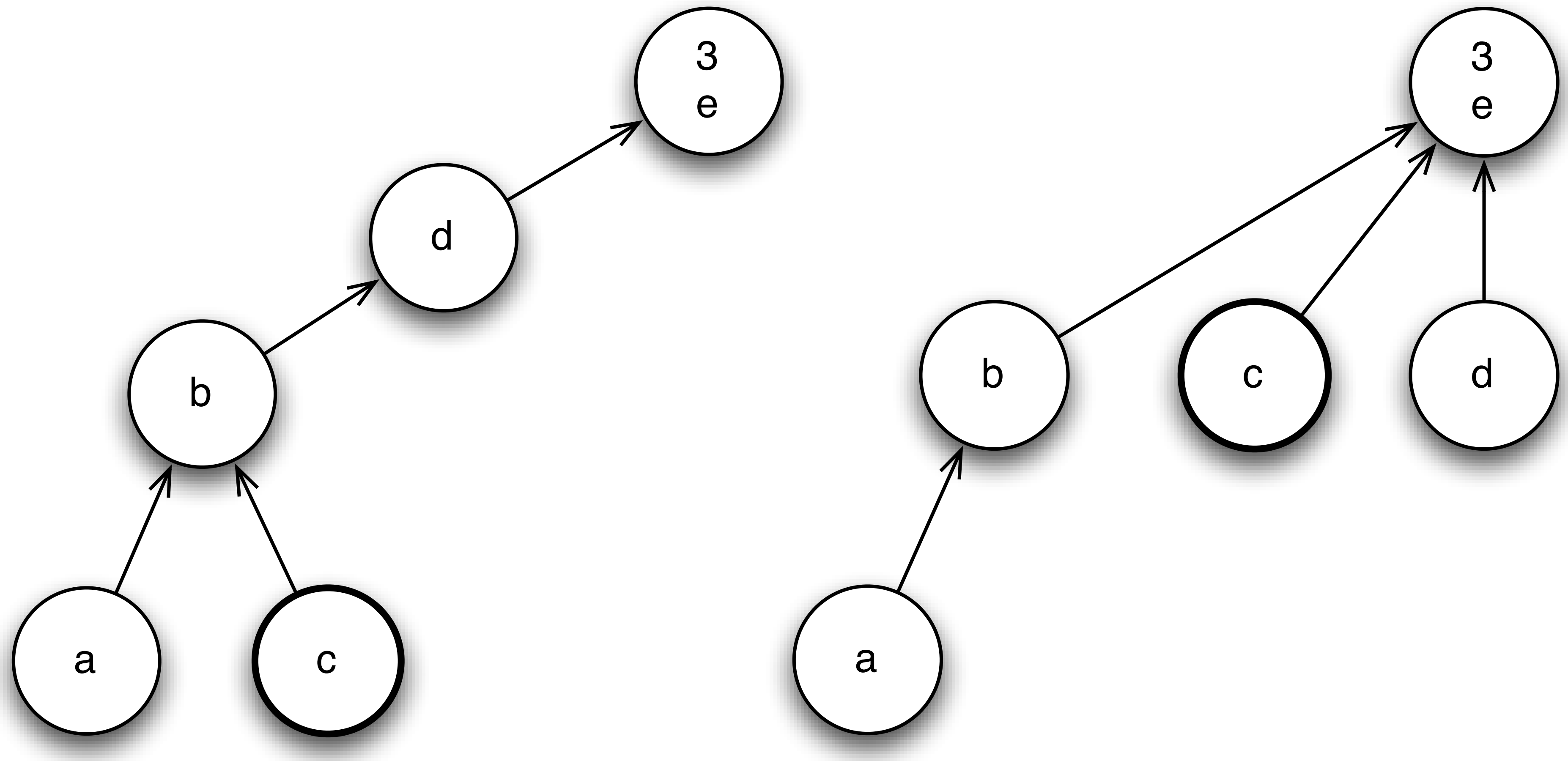
Union by Height Figure



Path Compression

- Even if we have log N tall trees, we can keep calling **find** on the deepest node repeatedly, costing O(M log N) for M operations
- Additionally, we will perform **path compression** during each **find** call
  - Point every node along the find path to root

Path Compression Figure



Union by Rank

- Path compression messes up union-by-height because we reduce the height when we compress
- We could fix the height, but this turns out to gain little, and costs **find** operations more
- Instead, rename to **union by rank**, where **rank** is just an overestimate of height
- Since heights change less often than sizes, rank/height is usually the cheaper choice

Worst Case Bound

- A slightly looser, but easier to prove/understand bound is that any sequence of $M = \Omega(N)$ operations will cost **O(M log* N)** running time
- log* N is the number of times the logarithm needs to be applied to N until the result is ≤ 1
- Proof idea: upper bound the number of nodes per rank, partition ranks into groups

Sorting

- Given array A of size N, reorder A so its elements are in order.
  - "In order" with respect to a consistent comparison function

The Bad News

- Sorting algorithms typically compare two elements and branch according to the result of comparison
- Theorem: An algorithm that branches from the result of pairwise comparisons must use $\Omega(N \log N)$ operations to sort worst-case input
- Proof via decision tree

Counting Sort

- Another simple sort for integer inputs
- 1. Treat integers as array indices (subtract min)
- 2. Insert items into array indices
- 3. Read array in order, skipping empty entries

Bucket Sort

- Like Counting Sort, but less wasteful in space
- Split the input space into **k** buckets
- Put input items into appropriate buckets
- Sort the buckets using favorite sorting algorithm

Radix Sort

- Trie method and CountingSort are forms of Radix Sort
- Radix Sort sorts by looking at one digit at a time
- We can start with the least significant digit or the most significant digit
  - least significant digit first provides a **stable** sort
  - trie's use most significant, so let's look at least...

Radix Sort with Least Significant Digit

- CountingSort according to the least significant digit
- Repeat: CountingSort according to the next least significant digit
- Each step must be **stable**
- Running time: **O(Nk)** for maximum of **k** digits
- Space: **O(N+b)** for base-**b** number system*

Comparison Sort Characteristics

- Worst case running time
- Worst case space usage (can it run in place?)
- Stability
- Average running time/space
- (simplicity)

Insertion Sort

- Assume first **p** elements are sorted. Insert **(p+1)**'th element into appropriate location.
  - Save **A[p+1]** in temporary variable **t**, shift sorted elements greater than **t**, and insert **t**
- Stable
- Running time $O(N^2)$
- In place **O(1)** space

Insertion Sort Analysis

- When the sorted segment is **i** elements, we may need up to **i** shifts to insert the next element

$$\sum_{i=2}^{N} i = N(N-1)/2 - 1 = O(N^2)$$

- Stable because elements are visited in order and equal elements are inserted after its equals
- Algorithm Animation

Shellsort

- Essentially splits the array into subarrays and runs Insertion Sort on the subarrays
- Uses an increasing sequence, $h_1, \ldots, h_t$, such that $h_1 = 1$.
- At phase **k**, all elements h_k apart are sorted; the array is called h_k-sorted
- for every **i**, $A[i] \leq A[i + h_k]$

Shell Sort Correctness

- Efficiency of algorithm depends on that elements sorted at earlier stages remain sorted in later stages
- Unstable. Example: 2-sort the following: [5 5 1]

Increment Sequences

- Shell suggested the sequence $h_t = \lfloor N/2 \rfloor$ and $h_k = \lfloor h_{k+1}/2 \rfloor$, which was suboptimal
- A better sequence is $h_k = 2^k - 1$
- Shellsort using better sequence is proven $\Theta(N^{3/2})$
- Often used for its simplicity and sub-quadratic time, even though **O(N log N)** algorithms exist
- Animation

Heapsort

- Build a **max** heap from the array: **O(N)**
- call deleteMax **N** times: **O(N log N)**
- **O(1)** space
- Simple if we abstract heaps
- Unstable
- Animation

Mergesort

- Quintessential divide-and-conquer example
- Mergesort each half of the array, merge the results
- Merge by iterating through both halves, compare the current elements, copy lesser of the two into output array
- Animation

Mergesort Recurrence

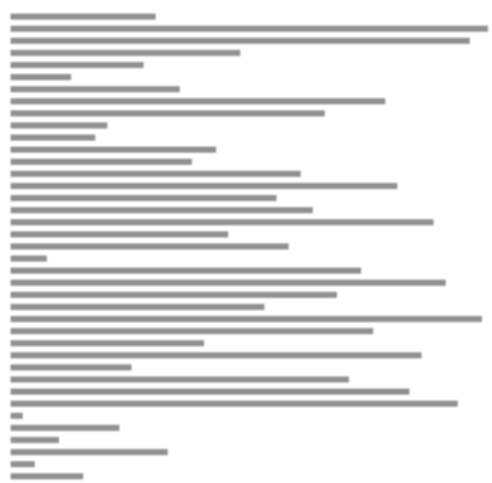
- Merge operation is costs **O(N)**
- **T(N) = 2 T(N/2) + N**
- We solved this recurrence for the recursive solutions to the homework 1 theory problem

$$= \sum_{i=0}^{\log N} 2^i c \frac{N}{2^i}$$

$$= \sum_{i=0}^{\log N} cN = cN \log N$$

Quicksort

- Choose an element as the **pivot**
- Partition the array into elements greater than pivot and elements less than pivot
- Quicksort each partition



Choosing a Pivot

- The worst case for Quicksort is when the partitions are of size zero and **N-1**
- Ideally, the pivot is the median, so each partition is about half
- If your input is random, you can choose the first element, but this is very bad for presorted input!
- Choosing randomly works, but a better method is…

Median-of-Three

- Choose three entries, use the median as pivot
- If we choose randomly, **2/N** probability of worst case pivots
- Median-of-three gives **0** probability of worst case, tiny probability of 2nd-worst case. (Approx. $2/N^3$)
- Randomness less important, so choosing (first, middle, last) works reasonably well

Partitioning the Array

- Once pivot is chosen, swap pivot to end of array. Start counters **i**=1 and **j**=**N-1**
- Intuition: **i** will look at less-than partition, **j** will look at greater-than partition
- Increment **i** and decrement **j** until we find elements that don't belong (**A[i]** > **pivot** or **A[j]** < **pivot**)
- Swap (**A[i]**, **A[j]**), continue increment/decrements
- When **i** and **j** touch, swap pivot with **A[j]**

Quicksort Worst Case

- Running time recurrence includes the cost of partitioning, then the cost of 2 quicksorts
- We don't know the size of the partitions, so let **i** be the size of the first partition
- **T(N) = T(i)+T(N-i-1) + N**
- Worst case is **T(N) = T(N-1) + N**

Quicksort Properties

- Unstable
- Average time O(N log N)
- Worst case time $O(N^2)$
- Space O(log N)/$O(N^2)$ because we need to store the pivots

Summary

	Worst Case Time	Average Time	Space	Stable?
Selection	$O(N^2)$	$O(N^2)$	$O(1)$	No
Insertion	$O(N^2)$	$O(N^2)$	$O(1)$	Yes
Shell	$O(N^{3/2})$	?	$O(1)$	No
Heap	$O(N \log N)$	$O(N \log N)$	$O(1)$	No
Merge	$O(N \log N)$	$O(N \log N)$	$O(N)/O(1)$	Yes/No
Quick	$O(N^2)$	$O(N \log N)$	$O(\log N)$	No

Selection

- Recall selection problem: best solution so far was Heapselect
    - Running time: **O(N+k log N)**
- We should expect a faster algorithm since selection should be easier than sorting
- Quick Select: choose a pivot, partition array, recurse on the partition that contains **k**'th element

Quickselect Worst Case

- Quickselect only recurses one one of the subproblems
- However, in the worst case, pivot only eliminates one element:
  - **$T(N) = T(N-1) + N$**
- Same as Quicksort worst case

External Sorting

- So far, we have looked at sorting algorithms when the data is all available in RAM
- Often, the data we want to sort is so large, we can only fit a subset in RAM at any time
- We could run standard sorting algorithms, but then we would be swapping elements to and from disk
  - Instead, we want to minimize disk I/O, even if it means more CPU work

MergeSort

- We can speed up external sorting if we have two or more disks (with free space) via Mergesort
- One nice feature of Mergesort is the merging step can be done online with streaming data
- Read as much data as you can, sort, write to disk, repeat for all data, write output to alternating disks
  - merge outputs using 4 disks

Simplified Running Time Analysis

- Suppose random disk i/o cost 10,000 ns
  - Sequential disk i/o cost 100 ns
  - RAM swaps/comparisons cost 10 ns
- Naive sorting: 10000 N log N
- Assume **M** elements fit in RAM.
  External mergesort:
  10 N log M + 100 N (# of sweeps through data)

Counting Merges

- After initial sorting, **N/M** sorted subsets distributed between 2 disks
- After each run, each pair is merged into a sorted subset twice as large.
  - Full data set is sorted after **log(N/M)** runs
- External sorting:
  10 N log M + 100 N log (N/M)