JAVA

- Use `Object.equals(Object o)` to compare
- String concatenation takes $O(\text{length})$

BIG-O NOTATION

Upper bound $T(n) = O(f(n))$ if for some $c > 0$ and $n_0 > 0$,

$$T(n) \leq cf(n)$$

for all $n > n_0$.

Lower bound $T(n) = \Omega(f(n))$ if for some $c > 0$ and $n_0 > 0$,

$$T(n) \geq cf(n)$$

for all $n > n_0$.

Tight bound $T(n) = \Theta(f(n))$ if

$$T(n) = O(f(n)) \quad \text{and} \quad T(n) = \Omega(f(n))$$

Properties Let $T(n) = O(f(n))$ and $S(n) = O(g(n))$.

1. If $T(n)$ is a polynomial of degree k then
$$T(n) = O(n^k)$$
2. Addition: $T(n) + S(n) = O(f(n) + g(n))$
3. Multiplication: $T(n) \times S(n) = O(f(n) \times g(n))$
4. Max: $\max(T(n), S(n)) = O(f(n) + g(n))$
5. Composition: $T(S(n)) = O(f \circ g(n))$ only if both functions are increasing

Overview

$$1 < \log n < \sqrt{n} < n < n \log n$$
$$< n^2 < n^3 < 2^n < 2^{2n} < n! < n^n$$

Stirling's approximation $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Used to show that $\log(n!) = O(n \log n)$

Geometric series

$$S_n = a + ar + ar^2 + \cdot + ar^{n-1} = \frac{a(1-r^n)}{1-r}$$

Harmonic series $\sum_{i=1}^{\infty} \frac{1}{i} = O(\log n)$

Logarithm change of base $\log_b a = \frac{\log_x b}{\log_x a}$

Master theorem Comparing $f(n)$ with $n^{\log_b(a)}$ as polynomials, for $a \geq 1$ and $b > 1$,

$$T(n)$$
$$= aT\left(\frac{n}{b}\right) + f(n)$$
$$= \begin{cases} \Theta(n^{\log_b(a)}) & \text{if } f(n) < n^{\log_b(a)} \\ \Theta(n^{\log_b(a)} \log n) & \text{if } f(n) = n^{\log_b(a)} \\ \Theta(f(n)) & \text{if } f(n) > n^{\log_b(a)} \end{cases}$$

ALGORITHMS

(with binary search as example algo)

Precondition Fact before algorithm runs. e.g. Array is sorted

Postcondition Fact when algorithm ends. e.g. If element in array, then `A[begin] = key`

Invariant Relationship between variables that is always true

Loop invariant Invariant at beginning/end of loop e.g. `A[begin] <= key <= A[end]`, i.e. key is in range

Peak finding

Assume `A[-1]` and `A[n]` are `-INT_MAX`.

```
FindPeak(A, n)
  mid = n/2
  if A[mid+1] > A[mid] then recurse on
       right half
  elif A[mid-1] > A[mid] then recurse on
       left half
  else A[mid] is a peak
```

If we recurse into right half, then:

- Given that `A[mid] < A[mid+1]` (condition for recursing into right half)
- Assuming no peak, then `A[mid] < A[mid+1] < A[mid+2] < ... < A[n-1] > A[n] = -INT_MAX`, i.e. `A[n-1]` is a peak
- Hence peak must exist

Sorting

Bubble compare adjacent and swap to make right > left

Selection find smallest element and swap it to end of sorted region

Insertion swap each new element until it is correctly placed within sorted region

Small arrays Insertion sort is stable, works fast for nearly sorted arrays, has little overhead.

Merge (recursive) sort each half and merge

Quicksort (recursive) partition about a pivot

In-place partitioning $O(n)$

1. Choose a random pivot (for good quicksort performance), and swap it to the start
2. Increase `lptr` while element at left < pivot
3. Decrease `rptr` while element at right > pivot
4. If not yet at centre, swap pointers and repeat

Quicksort variations

- Is stable only if the partitioning is stable. Requires $O(n)$ extra space to store initial indices
- If pivot splits by a fraction, good enough
- 3-way partitioning: pack duplicates in middle to eliminate duplicate elements worst case
- Paranoid: force a good pivot
- k-pivots: $O(k \log k)$ to sort pivots + $O(n \log k)$ to binary search the pivots to choose correct bucket to place the new item. $T(n) = O(n \log_k n \log k) = O(n \log n)$, i.e. no performance improvements
- 2-pivot quicksort is in practice better than 1-pivot quicksort, because of cache performance

Quickselect Like quicksort, but only recurse on relevant half of partitioned array

Probability

Linearity of expectation For random variables X and Y: $E(aX + bY) = aE(X) + bE(Y)$

Expected trials If an event X has probability of success p, then an expected $\frac{1}{p}$ trials is required for 1 success.

TREES

Binary trees

- A binary tree is either empty, or a node pointing to two binary trees.
- In a balanced tree, $h = O(\log n)$
- Height is no. of edges on longest path to leaf

The following operations are all $O(\log n)$:

searchMin keep going left

searchMax keep going right

search, insert go left/right depending on comparison of new key with cur key

successor

- If node has right child: `right.searchMin()`
- Else traverse upwards until ancestor contains node in left subtree, then return ancestor

predecessor

- If node has left child: `left.searchMax()`
- Else traverse upwards until ancestor contains node in right subtree, then return ancestor

delete

- 0 children: remove node
- 1 child: remove node, connect parent to child
- 2 children: delete successor (at most 1 child), replace node value with successor value

The following operations are all $O(n)$:

in-order left, self, right

pre-order self, left, right

post-order left, right, self

level-order decreasing height (BFS)

Ancestor Node x is an ancestor of node $y \Leftrightarrow x$ comes before y in pre-order, AND y comes before x in post-order.

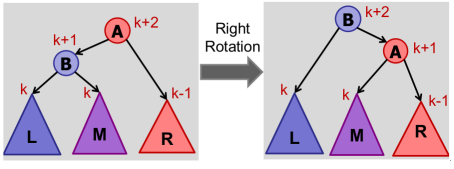
AVL trees

- A node v is height balanced if `|v.L.height - v.R.height| <= 1`
- Consider a height balanced tree with height h and n nodes.
  - **At most** $h < 2 \log n$ (actually, approximately $\frac{1}{\log \phi} \log n$)
  - **At least** $n > 2^{h/2}$ nodes
- Can build AVL tree in $O(N)$ given sorted array (keep selecting median as root of subtree)

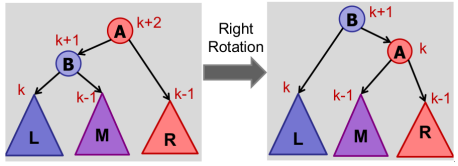
Balancing Assume A is left-heavy. Otherwise, if A is right-heavy, substitute ALL left, right with right, left

Define B as left child of A.

Case 1: B is **balanced:** `rightRotate(A)`



Case 2: B is **left-heavy:** `rightRotate(A)`



$O(1)$ work

$$T(n) = T(\tfrac{n}{k}) + O(1) = O(\log n)$$
$$T(n) = kT(\tfrac{n}{k}) + O(1) = O(n)$$
$$T(n) = kT(\tfrac{n}{2k}) + O(1) = O(\sqrt{n})$$

Not $O(1)$ work

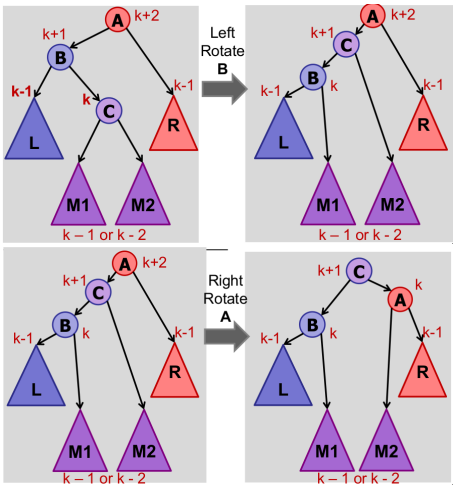
$$T(n) = T(\tfrac{n}{k}) + O(n) = O(n)$$
$$T(n) = kT(\tfrac{n}{k}) + O(n) = O(n \log n)$$
$$T(n) = kT(\tfrac{n}{k}) + O(n \log n) = O(n \log^2 n)$$

Subtract in recurrence

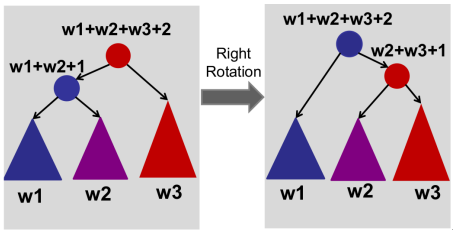
$$T(n) = T(n-c) + O(n) = O(n^2)$$
$$T(n) = 2T(n-1) + O(1) = O(2^n)$$
$$T(n) = T(n-1) + T(n-2) + O(1) = O(\phi^n)$$

Sort	Best	Average	Worst	Stable	Memory	Invariant (after k iterations)
Bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$	last k elements in correct final position
Selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$	first k elements in correct final position
Insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$	(original) first k elements in relative sorted order
Merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	$O(n)$	subarray is sorted
Quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	×	$O(1)$	pivot is in correct final position

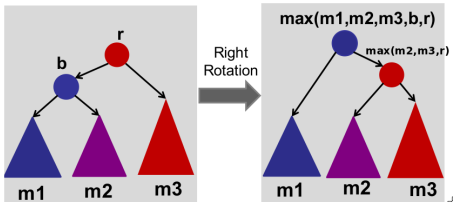
Case 3: B is **right-heavy:**
`leftRotate(A.left); rightRotate(A)`



Update weights after `rightRotate(A)`



Update max after `rightRotate(A)`



- Insertion: max 2 rotations
- Deletion: max $O(\log n)$ rotations

Rank of node Position in-order

```
rank(node):
  rank = node.L.weight + 1
  while node != null
    if node is right child
      rank += node.parent.L.weight + 1
    node = node.parent
  return rank
```

Trie

search, insert: $O(L)$, space: $O\left(\sum L + \text{overhead}\right)$

Order Statistics

Find kth smallest element Quicksort but only recurse on relevant side - $O(n)$

Dynamic Augment with size of subtree at each node, use rank algo

Interval Tree

Each node stores an interval (left endpoint as key), augmented with max right endpoint in subtree

Search $O(\log n)$

1. If value in node interval, return node
2. If `value > node.left.max`, recurse right
3. Else recurse left

All-overlaps If we want to find all k intervals that contain value - $O(k \log n)$

1. Search for interval
2. Add to list and delete interval

Orthogonal Range Searching

- Leaf nodes contain points
- Internal nodes contain max in left subtree
- Build tree: $O(n \log n)$, Space: $O(n)$

Search Find number of points in range $[a, b]$ - $O(k + \log n)$, where k is number of points found

1. Find split node, highest node with key in range $[a, b]$
2. Do left child traversal
   - If node in query range, then add entire right subtree to list, and recurse left
   - Else recurse right
3. Do right child traversal (similar)

n dimensional search

- Recursively store $d-1$ dim range tree in each node of a 1D range tree
- Build tree: $O(n \log^{d-1} n)$, Query: $O(\log^d n + k)$
- Space: $O(n \log^{d-1} n)$, 2D tree Rotate: $O(n)$

k-d tree

- Stores coordinates in $x - y$ plane
- Levels alt. between splitting plane by x or y

Search node $O(\log n)$

1. If horizontal split, compare x-coordinate
2. If vertical split, compare y-coordinate
3. $O(h)$ time

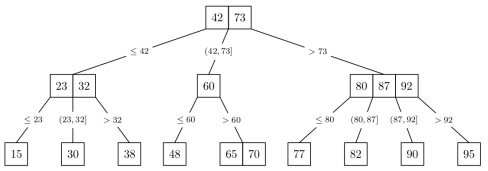
Search min $O(\sqrt{n})$ (e.g. min x)

1. If horizontal split, recurse left child
2. If vertical split, recurse on both children
3. $T(n) = 2T(n/4) + O(1)$

Build $O(n \log n)$

1. Choose either x or y.
2. Quickselect median of x or y: $O(n)$
3. Split array into two halves using median Partioning: $O(n)$

(a, b)-tree



Sample $(2, 4)$-tree

Rules

1. (a, b) child policy

	# Keys		# Children	
Node	Min	Max	Min	Max
Root	1	$b-1$	2	b
Internal	$a-1$	$b-1$	a	b
Leaf	$a-1$	$b-1$	0	0

2. A non-leaf node must have one more child than its number of keys
3. All leaf nodes must all be at the same depth

Definitions

- Key range: Range of keys allowed in a subtree (wrt parent)
- Key list: List of keys in node (assume sorted)
- Tree list: List of children

B-tree simply $(B, 2B)$ trees

Search $O(\log n)$:

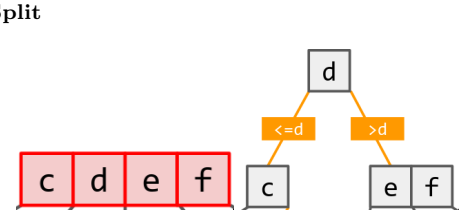
- $O(\log b)$ binary search keylist for subtree containing the key to search
- Repeat along height of $O(\log_a n)$

Insert $O(\log n)$: Like search, then perform split/merge as necessary

- Proactive: preemptively split nodes at full capacity (only applies if $b \geq 2a$)
- Passive: insert then check (potentially splitting all the way to root)

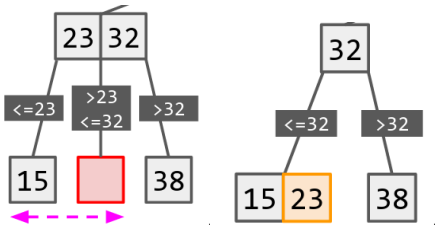
Delete $O(\log n)$: Like search, then perform split/merge as necessary

Split



1. Pick median key of overfull range z as new split key k
2. Put k into parent
3. Split z into LHS and RHS of k
4. If parent is overfull, `split(parent)`

Merge



Let d be deleted node, and l be left sibling of d. Assume keylist of l and d have $< b-1$ keys in total. Otherwise use share below

1. Pick key k from parent, on left of d
2. Move k to keylist of l
3. Merge d keylist, treelist into l
4. Delete d

Share `merge(l, d)`; then split newly combined node

HASHING

Hash functions

- Maps universe to keys in $\{1, \cdots, m\}$
- Store item with key k in bucket $hash(k)$
- Since universe size larger, collisions inevitable by pigeonhole principle

Simple uniform hashing assumption

1. Each key has equal probability of being mapped to each bucket
2. Each key is mapped independently to each bucket

Uniform hashing assumption

- Stronger version of SUHA property 2: Every key is mapped independently to every permutation.

Java implementation

`java.util.Map` interface

- Does not allow duplicate or mutable keys

hashCode

- Always returns the same value, if the object hasn't changed
- If two objects are equal, then they return the same hashCode
- Must redefine `.equals()` to be consistent with hashCode

equals

- Reflexive: $x = x$
- Symmetric: $x = y \Rightarrow y = x$
- Transitive: $(x = y) \wedge (y = z) \Rightarrow x = z$
- Consistent: always returns same answer
- Null: `x.equals(null)== false`

Chaining

Assume n keys have been inserted into hash table of size m.

- Each bucket c stores linked list of items with $hash(k) = c$
- Total space: $O(n + m)$

Insert

- Allow duplicate keys: $O(1 + hash) = O(1)$
- Don't allow dupe keys $\Rightarrow$ search on insert

Search

- Without SUHA (assume all items have same hash): $O(n + hash) = O(n)$
- With SUHA (consider expected case): $O(1 + hash + n/m)$
- Choose good m, e.g. $m = 2n$: $O(1 + hash)$

Expected max cost of n inserts

- $O(\log n) = \Theta(\frac{\log n}{\log \log n})$

Open addressing

Properties

- NULL = empty, DELETED = deleted
- Insert: probe a sequence of buckets until NULL/DELETED
- Search: probe same sequence until found/NULL
- Delete: find key to delete, set bucket to DELETED

Linear probing

Define new hash function h such that

$$h(key, i) = k(key, 1) + i \pmod m$$

where i is the number of collisions.

- SUHA property 1: $h(key, i)$ is a permutation of $\{1, \cdots, m\}$
- Does NOT satisfy stronger version of SUHA property 2, because linear probing requires a circular permutation of $\{1, \cdots, m\}$

Performance

- Clustering: If table is 1/4 full, $\exists$ clusters of size $\Theta(\log n)$, which ruins constant-time performance
- In practice, caching makes this much faster
- Expected cost of an operation, $E[\#\text{ probes}] \leq \frac{1}{1-\alpha}$
- α is average no. of items per bucket, n/m.

Double hashing

Given two hash functions f, g, define:

$$h(k, i) = f(k) + i \cdot g(k) \pmod m$$

- If $g(k)$ is relatively prime to m, then $h(k, i)$ hits all buckets.
- If $m = 2^r$, then choose $g(k)$ odd.

Table size

Let n be the number of elements in the hash table. Let m be the size of the hash table.

Growing table Time: $O(m_{\text{old}} + m_{\text{new}} + n)$

growth	resize	insert n items
increment	$O(n)$	$O(n^2)$
double	$O(n)$	$O(n)$
square	$O(n^2)$	$O(n^2)$

Shrinking table

- If we choose to shrink when table at half capacity, we have a scenario. Consider $n = 200, m = 100$. Delete causes a shrink, insert causes a grow, repeat.
- Hence, we choose the following:
  - If $(n == m)$, then $m = 2m$
  - If $(n < m/4)$, then $m = m/2$
  - Growth $\Rightarrow \geq m/2$ new items are added $\Rightarrow$ can pay for growing
  - Shrink $\Rightarrow \geq m/4$ items are deleted $\Rightarrow$ can pay for shrinking

AMORTIZED ANALYSIS

An op has amortized cost $T(n)$ if for every integer k, the cost of k ops is $\leq kT(n)$.

Hash table resizing Inserting k elements into a hash table with resizing takes $O(k)$, so insert has amortized $O(1)$ cost.

Binary counter Increment has amortized $O(1)$ cost

GRAPHS

Graph

- Nonempty set of objects (nodes) with a set of relations (edges)
- Each edge connects two nodes

Multigraph Two nodes may be connected by more than one edge

Hypergraph Each edge connects ≥ 2 nodes

(Simple) path

- Set of edges connecting two nodes
- Path intersects each node at most once

Cycle "Path" where first node = last node

Degree of node Number of adjacent edges

Degree of graph Max degree over all nodes

Diameter of graph Max distance between two nodes in a graph, using shortest path

Types of graphs

Connected graph Every pair of nodes is connected by a path

Disconnected graph Some pair of nodes is not connected by a path

Tree Connected graph with no cycles

Forest Graph with no cycles

Star One central node, all edges connect center to other nodes

Clique (complete graph) All pairs of nodes are connected with an edge

Line Tree with degree 2

Cycle Entire graph is a cycle

Bipartite

- Nodes can be divided into two sets, with no edges between nodes of the same set
- Does not have odd-length cycles

Type	Degree	Diameter
Star	$n-1$	2
Clique	$n-1$	1
Line	2	$n-1$
Cycle	2	$n/2$ or $n/2-1$

Planar graphs

Can be drawn on 2D plane without edges crossing

Face Area bounded by edges (outer, infinite area is also a face)

Euler's formula $V - E + F = 2$

DAGs

Directed graph Graph with directed edges

In-degree Number of incoming edges

Out-degree Number of outgoing edges

DAG Directed graph with no cycles

Representation

Adjacency list Space: $O(V + E)$

- Nodes: stored in array, edges: linked list per node
- Only stores outgoing edges for directed graphs

Adjacency matrix Space: $O(V^2)$

- 2D array, where $A[v][w] = e(v, w)$
- A^k represents all length k paths
- Symmetric $\iff$ undirected graph

Searching (Unweighted)

BFS $O(V + E)$, using adjacency list

- Uses a queue
- Each node is visited once, and each neighbour is only enumerated once.
- BFS parent edges form a shortest path tree (from root)

DFS $O(V + E)$, using adjacency list

- Uses a stack
- DFS called once per node, and each neighbour is only enumerated once.
- DFS parent edges form a tree

Topological ordering

Given a DAG, find a total ordering of nodes, where all edges point forwards.

Idea 1 DFS with post-order processing: $O(V+E)$

- Perform usual DFS
- Add a node to the topo-order (in reverse order) during the post-order.

Idea 2 Kahn's algorithm: $O(V + E)$

- Let S = all nodes in G that have no incoming edges. S is nonempty because G is a DAG.
- Dequeue v from S
  - Add v to topo-order
  - Remove edges adjacent to v
  - If any neighbours have no incoming edges, add to S
- Repeat until G is empty

We can apply the same idea considering in-degree instead.

Connected components

Undirected graph

- Vertex v and w are in the same connected component $\iff$ there is a path from v to w
- There is a set $\{v_1, v_2, \cdots, v_k\}$ where there is no path from v_i to v_j $\iff$ there are k connected components

Strongly connected components (Only applies to directed graphs) A component is strongly connected if for every vertex v and w, there is a path from v to w, and from w to v.

Graph of strongly connected components If we replace each strongly connected component with a supernode, then the graph of the supernodes is acyclic.

Shortest paths

Δ inequality $\delta(S, C) \leq \delta(S, A) + \delta(A, C)$

Bellman-Ford

Algorithm $O(VE)$

- Maintain distance estimate for every node
- Repeat $|V| - 1$ times: relax every edge
- If one iteration does not relax any edge, we can terminate early

```
void relax(int u, int v) {
  if (dist[v] > dist[u] + weight(u,v))
    dist[v] = dist[u] + weight(u,v);
}
```

Properties

- k-hop estimate for node v is correct after k iterations, only if v is on the shortest path tree rooted at source s
- After k iterations, gives all shortest paths within k hops from start
- Works with negative edges
- Does not work with negative cycle, but if run for $|V|$ iterations, and estimate changes in the last iteration, then negative weight cycle exists
- Given a set of distance estimates, we can run 1 iteration of Bellman-Ford ($O(E)$), and if the distance estimates do not change, then the distance estimates are accurate
- If edge weights are all same, then use BFS and multiply the answer by the edge weight

Dijkstra

Algo $O((E + V) \log V)$ with AVL-based PQ

- Maintain distance estimate for every node
- Begin with empty shortest path tree
- Repeat until destination reached:
  - Let v be the node with the minimum distance estimate so far
  - Add v to shortest path tree, mark v as visited
  - Relax all outgoing edges from v

Time complexity

- `insert`: $|V|$ times of $O(\log V)$
- `deleteMin`: $|V|$ times of $O(\log V)$
- `decreaseKey`: $|E|$ times of $O(\log V)$
- Total is $O((V+E)\log V) = O(E \log V)$

Properties

- Does not work with negative edges
- cf. BFS/DFS, here we use a PQ

DAGs

Algorithm $O(V+E)$

- Generate topological order - $O(V+E)$
- Relax edges in topological order - $O(E)$

Longest path Negate edges. Shortest path in negated graph is the longest path in original graph

MSTs

Spanning tree is an acyclic subset of the edges that connects all nodes

Minimum spanning tree is a spanning tree with minimum total edge weight

Cut of a graph is a partition of the vertices into two disjoint subsets

- An edge crosses a cut if it has one vertex in each of the two sets

Properties

1. No cycles
2. Cut an MST, the two pieces are both MSTs.
3. In each cycle, the max weight edge is not in the MST. Min weight edge is inconclusive.
4. In each cut, min weight edge across cut in MST.

Generic algorithm

- Red rule: If C is a cycle with no red arcs, then color the max weight edge in C red.
- Blue rule: If D is a cut with no blue arcs, then color the min weight edge in D blue.
- Apply red rule or blue rule to all edges. The blue edges are the edges of a MST.

Prim's algorithm $O(E \log V)$

- Maintain a cut, and store its cut-edges in a PQ.
- Remove min cut-edge, growing the cut by this edge (and node)
- Each vertex added/removed once from PQ - $O(V \log V)$
- Each edge has one decreaseKey - $O(E \log V)$

Small edge weights $O(V+E)$ - Use array of size k as PQ

Kruskal's algorithm $O(E \log V)$

- Sort edges, and keep adding edge to set only if it does not cause a cycle
- Sorting - $O(E \log E) = O(E \log V)$
- Find/Union - $O(\log V)$ per edge

Small edge weights $O(\alpha E)$ - Use array of size k to sort

All edges have same weight

- DFS or BFS, any spanning tree is a MST
- MST has $V-1$ edges, cost is $k(V-1)$, where k is the edge cost

Directed MST $O(E)$

- If we have a DAG with one root, for every node except the root, add min weight incoming edge

Maximum spanning tree

- Negate edge weights and run MST, or
- Kruskal but sort descending

UNION-FIND

Quick find `int[] componentId`, flat trees

- Object ID is the array index (hash table + open addressing)
- Component ID is the array value
- $O(1)$ Find: check if two objects have same `componentId`
- $O(n)$ Union: iterate through array and update relevant `componentId`

Quick union `int[] parentId`, tall (unbalanced) trees

- $O(n)$ Find: check if same root
- $O(n)$ Union: join one root to the other root

Weighted union `int[] parentId`, `int[] size`

- $O(\log n)$ Find: check if same root
- $O(\log n)$ Union: join root of smaller tree to the other root
- Height only increases when total size doubles, so tree of height k has size at least 2^k
- Height of tree of size n is at most $\log n$

Path compression `int[] parentId`

- $O(\log n)$ Find: check if same root
- $O(\log n)$ Union: after finding root, set the parent of each traversed node to the root

Weighted union + path compression

- For m union/find operations on n objects, expected $O(n + m \cdot \alpha(m,n))$
- For a single find operation, worst case is $O(\log n)$, if tree not compressed yet

DYNAMIC PROGRAMMING

Properties

1. **Optimal sub-structure** Optimal solution can be constructed from optimal solutions to smaller sub-problems (DP, divide and conquer)
2. **Overlapping subproblems** The same smaller problem is used to solve multiple different bigger problems (DP)

Recipe

CS2040S Identify optimal substructure, Define subproblems, Solve problem using subproblems (recurrence)

6.006 Subproblem, Relate (recursively), Topo order, Base cases, Original problem, Time analysis

Longest increasing subsequence

Subproblem `S[i]` is length of LIS using indices 1 to i, ending at `A[i]`

Recurrence

$S[1] = 0 \qquad S[i] = 1 + \max_{(j<i, A_j<A_i)} S[j]$

Time $O(n)$ subproblems, $O(n)$ each: $O(n^2)$

Prize collecting

Find longest path in graph, with exactly k edges

Subproblem `P[v,k]` is maximum prize you can collect starting at v, taking exactly k steps

Recurrence

- $P[v,0] = 0$
- $P[v,k] = \max_{\forall i}\{P[w_i, k-1] + e(v,w_i)\}$, where w_i are neighbours of v

Time

- $O(kV)$ subproblems $\times$ $O(V)$ time each: $O(kV^2)$
- Or using table, we have k rows and $O(E)$ cost to solve all problems in a row, giving $O(kE)$

Vertex cover on a tree

Given tree, find minimum set of nodes where every edge is adjacent to at least one node

Subproblem

- $S[v,0]$ is size of vertex cover in subtree rooted at v, if v is NOT covered
- $S[v,1]$ is size of vertex cover in subtree rooted at v, if v IS covered

Recurrence

- $P[\text{leaf}, 0] = 0$ and $P[\text{leaf}, 1] = 1$
- $S[v,0] = \sum_{\forall i} S[w_i, 1]$, where w_i are adjacent to v
- $S[v,1] = 1 + \sum_{\forall i} \min(S[w_i,0], S[w_i,1])$

Time

- $2V$ subproblems, each edge explored once
- $O(V)$ time to solve all subproblems

All Pairs Shortest Paths

SSSP on all

- Run Dijkstra on all: $O(VE \log V)$
- (Identical weight) BFS: $O(V(E+V)) = O(VE)$

Floyd-Warshall

Optimal substructure If P is the shortest path $u \to v \to w$, then P contains the shortest paths $u \to v$ and $v \to w$.

Subproblem `S[v,w,P_i]` is the shortest path from v to w that only uses intermediate nodes in $\{1, \cdots, i\}$

Recurrence

$$S[v,w,P_{i+1}] = \min(S[v,w,P_i], \\ S[v,i+1,P_i] + S[i+1,w,P_i])$$

Time $O(V^3)$ subproblems, $O(1)$ each: $O(V^3)$

Reconstructing path In `arr[i][j]` store the first hop on the shortest path from i to j

HEAPS

Properties

- Key of every node $\geq$ Key of node's children
- Complete binary tree, i.e. every level (except last) must be full. Leaf nodes are as far to the left as possible.

Operations

Height is $\lfloor \log n \rfloor$, so all operations are $O(\log n)$.

bubbleUp compare with parent (recursively) and swap if necessary

bubbleDown compare with children (recursively) and swap if necessary (in maxheap, if there are two children, swap with higher priority)

increaseKey/decreaseKey modify key and `bubbleUp()`/`bubbleDown()`

insert insert as leftmost leaf, `bubbleUp()`

delete swap node with rightmost leaf, delete node, `bubbleDown()` swapped node

extractMax `delete(root)`

Heap as array

	1-based index	0-based index
`u.left`	$i \times 2$	$(i+1) \times 2 - 1$
`u.right`	$i \times 2 + 1$	$(i+1) \times 2$
`u.parent`	$\lfloor i/2 \rfloor$	$\lfloor (i+1)/2 \rfloor - 1$

Heapify $O(n)$

- Given array, `bubbleDown()` in reverse
- `bubbleUp()` from leaf is $O(\log n)$, but `bubbleDown()` from arbitrary node is $O(h)$, where h is the height of subtree at that node

HeapSort $O(n \log n)$

- $O(n)$ - heapify
- $O(n \log n)$ - `extractMax()` n times

Dijkstra PQ implementations

PQ impl	insert/decreaseKey	deleteMin	total
Array	1	V	$O(V^2)$
AVL Tree	$\log V$	$\log V$	$O(E \log V)$
d-way Heap	$d \log_d V$	$d \log_d V$	$O(E \log_{E/V} V)$
Fibonacci Heap	1	$\log V$	$O(E + V \log V)$

Knuth-Shuffle $O(n)$

```
for (i from 2 to n) do
  Choose r = random(1, i)
  Swap(A, i,r)
end
```

NaiveShuffle++ $O(n)$

```
for (int i=n-1; i>=1; i--) {
    // random number in [0, i)
    int j = Random.nextInt(i);
    swap(i, j);
}
```