1 Problem Types & Algorithms

1.1 Median Find (lec2)

Uses divide-and-conquer to find median. Picks a pivot x such that its rank is not extreme, to avoid unbalanced recursion (high running time). Rank is the number of items less than x in the set S.

- Splits S into groups of size $n/5$ and finds the median of the medians of these 5 groups.
- Uses this as the pivot x, then iterates through S to find the actual rank of x.
- If median, done, otherwise calculate the difference from the median and find the element of required rank in either the set of items $< x$ or $> x$.

operation	**runtime**
MEDIAN-FIND(S)	$O(n)$

1.2 Union Find (lec4)

Maintains a dynamic collection of pairwise disjoint sets, with an arbitrary representative element. Forest of trees method uses a tree to represent each set, with the root as the representative.

- MAKE-SET initiates a new tree with the element specified.
- FIND-SET follows the tree up from the given node to find the root, returns that.
  - As we walk up the tree to the root, redirect pointers of every node we encounter to point to the root. Requires walking up and back down, but this doesn't change the runtime.
- UNION follows calls FIND-SET on each argument, sets root of shorter tree as the child of the root of the taller tree (maintains rank of each set).

Runtime: Proved in textbook that a sequence of m operations, n of which are MAKE-SET has worst-case running time $O(m\alpha(n))$. Therefore, with this sequence:

operation	**runtime**
FIND-SET(x)	$O(\alpha(n))_a$
MAKE-SET(x)	$O(\alpha(n))_a$
UNION(x, y)	$O(\alpha(n))_a$

1.3 Minimum Spanning Tree (lec8, rec4)

A *spanning tree* of a graph is an acyclic subset of the edges in G that connects all V. Given a connected, undirected graph $G = (V, E, w)$, a *minimum* spanning tree is a spanning tree where the sum of the weights of the edges is minimized.

1.3.1 Properties

- **Cut Property:** If S is some non-empty subset of the vertices, for any cut $(S, V\S)$ any least-weight edge crossing the cut must belong to some MST.
  - A unique least-weight edge is in *all* MSTs.
- **Cycle Property:** If a cycle C in the graph G contains an edge e whose weight is strictly larger than every other edge in C, then e cannot be in any MST.
- **Optimal Substructure:** (rec4 p5) If an edge e is in some MST $T*$ of G, and T' is some MST of $G' = G/e$, then $T = T' \cup \{e\}$ is an MST of G. In other words (I think), you can swap out any edge between two nodes that is part of an MST with a different edge that is a part of another MST, and the new tree will still be an MST.

1.3.2 Kruskal's Algorithm (lec8 pg4, rec4 pg6)

This algorithm finds an MST of a graph $G = (V, E, w)$ by initially containing each vertex in a separate set, then choosing minimum-weight edges to connect disjoint sets (taking advantage of the cut property). We store these sets using the Union Find data structure.

- Initialize $|V|$ sets, one for each $v \in V$.
- Sort all edges by increasing weight order. This allows us to always take the first edge in E that crosses each cut.
- For every edge (u, v), check if u and v are in distinct sets. If so, UNION the sets and add e to the tree T.

Sorting the edges takes $O(|E| \log |E|) = O(|E| \log |V|)$ (using $|E| = O(|V|^2)$). We do at most $|E|$ UNION ops, and $2|E|$ FIND-SET ops, and this takes $O(|E|\alpha(|V|)$ by the Union-Find runtimes. Thus:

operation	**runtime**
KRUSKAL($G = (V, E, w)$)	$O(\|E\| \log \|V\|)$

1.3.3 Prim's Algorithm (lec8 p6, rec4 p5)

Prim's algorithm grows a tree by selecting the lowest weight edge that connects the tree to a vertex not in the tree yet.

- Initialize the tree with edges $T_E = \varnothing$ and $T_V = \{v\}$ where $v \in V$ is some single vertex.
- Until the tree includes all vertices, find the lowest weight edge (u, v) such that $u \in T_V$, $v \notin T_V$, and update T_V and T_E.

To improve the runtime, we store the edges that connect to other vertices in a priority queue, which we update on each step. See recitation notes–basically, select the vertex u closest to the tree and add it, then update its neighbors' keys to be the weight of the edge (u, v). This edge is the lightest weight edge connecting v to some vertex in the tree, or else we would have added v instead of u. Thus updating maintains the PQ. We can implement the PG with a fibonacci heap. With the PQ, we examine each edge only once. Then, our overall runtime is $O(|E| \cdot T_{Decrease-Key} + |V| \cdot T_{Extract-Min}$. Thus:

operation	**runtime**
PRIM($G = (V, E, w)$)	$O(\|E\| + \|V\| \log \|V\|)$

2 Concepts & Techniques

2.1 Divide and Conquer

Split a problem into subproblems—modeled by a recurrence. Use the tree method to solve this, sum the work across each level and multiply it by the number of levels, basically. Alternatively use **master theorem** to solve (see 006 cheat sheet).

2.1.1 Common Recurrences

- Merge Sort: $T(n) = 2T(n/2) + \Theta(n) \rightarrow n \log n$. Expands into a tree with $\log_2 n + 1$ levels, and each level has a total of $n/2$ work across it.

2.2 Randomized Algorithms

Allow us to spread worst-case runtimes of certain cases out by randomizing the decisionmaking.

- **Monte Carlo**: worst-case polynomial runtime, but not always correct
- **Las Vegas**: *expected* polynomial runtime, but always correct

2.3 Amortized Analysis (lec4, rec2)

Using a sequence of events to analyze cost over a sequence of events. An operation has *amortized cost* $T(n)$ iff any k operations have cost $\leq k \cdot T(n)$.

2.3.1 Aggregate Method

Compute total cost of k operations and divide by k. Useful for simple analyses.

2.3.2 Potential Method (rec2)

Use a potential function to shift work from expensive operations and distribute it across multiple cheaper operations when analyzing. For any operation in the sequence, we define *make-believe cost* (for purposes of analysis) as $\hat{c}$, *true cost* as c, and the change in potential energy from the previous step to this step as $\Delta\Phi$. We then have:

$$\hat{c} \equiv c + \Delta\Phi$$

We can compute the cost of a sequence of n operations as well, in terms of the final and initial potential:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

As long as $\Phi(D_n) \geq \Phi(D_0)$, the total amortized cost ($\sum \hat{c}_i$) gives an upper bound on the total actual cost. The potential function must then be chosen so that the value of the potential function never drops below its initial value.

If choosing Φ, choose it such that $\Delta\Phi$ decreases by a big step when expensive operations happen.

2.4 Competitive Analysis (lec5, rec3)

Used to compare an online algorithm that processes requests as they come in to an optimal offline algorithm that knows the full sequence of requests beforehand. An algorithm A is α-competitive if, for any sequence of requests R, the total cost is at most α times the optimal (plus a constant k):

$$C_A(R) \leq \alpha \cdot C_{OPT}(R) + k$$

This can be done using the potential method as well, and it can be useful to approach it from an amortized standpoint, using again $\hat{C}_i = C_i + \Delta\Phi$.

$$C_{A,i}(R) + \Delta\Phi) \leq \alpha C_{OPT,i}(R)$$

If we can prove this for every case depending on the decisions of C_A and C_{OPT}, we can then multiply this amortized cost by $|R|$ to prove the initial inequality.

2.5 Hashing (lec6, lec7, rec3)

A hash function takes an input in the set U of possible keys, and returns an output on the range $\{0, \ldots, m-1\}$. This can be used in conjunction with a direct access array of size m to build a hash table.

2.5.1 Universal Hash Family (lec6, rec3)

Hash functions by definition have collisions. We can distribute these collisions by generating a random hash function for every hash table, but this uses $O(|U|)$ space. A UHF allows us to achieve the same goal with much less space.

A family $\mathcal{H}$ of hash functions h_i, each of which maps a key in U to one of m slots, is a Universal Hash Family if an arbitrary pair of keys (k, k') collide with probability at most $\frac{1}{m}$ when we pick a random hash function from $\mathcal{H}$:

$$\Pr_{h \in \mathcal{H}}[h(k) = h(k')] \leq \frac{1}{m} \text{ for all } k \neq k'$$

2.5.2 Perfect Hashing (lec7)

Perfect hashing is hashing without collisions—it allows worst-case $O(1)$ lookups instead of expected. Implemented with randomized algorithms–we generte the data structure in expected time, however.

3 Sorting Algorithms

Algorithm	Time $O(\cdot)$	In-place?	Stable?	Comments
Insertion Sort	n^2	Y	Y	$O(nk)$ for k -proximate
Selection Sort	n^2	Y	N	$O(n)$ swaps
Merge Sort	$n \log n$	N	Y	stable, optimal comparison
AVL Sort	$n \log n$	N	Y	good if also need dynamic
Heap Sort	$n \log n$	Y	N	low space, optimal comparison
Counting Sort	$n + u$	N	Y	$O(n)$ when $u = O(n)$. $u = max(vals)$
Radix Sort	$n + n \log_n u$	N	Y	$O(n)$ when $u = O(n^c)$

Selection Sort finds largest number in remainder of the list, then move it to the front. Recurse.

Merge Sort recursively sorts halves of the list, then merges them with an $O(n)$ two finger alg.

Insertion Sort Recursively sorts the prefix, then adds and sorts in one item at a time.

Counting Sort basically uses a direct access array, with a chain in each slot. Insert then read back.

AVL Sort inserts into a set AVL and reads it back. Any set data structure defines a sorting alg.

Heap Sort is priority queue sort but with a max heap. Build then repeatedly delete.

Radix Sort uses tuple sort with auxillary counting sort to sort tuples. Sort from least significant key to most significant key. If every key $< n^c$ for some $0 < c = \log_n(u)$, at most c digits base n. Then write it as a c element tuple and sort each digit in $O(c)$ time. If c is constant so each key is $\leq n^c$, linear time.

4 Data Structures

4.1 Sequences

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	`build(x)`	`get_at()` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i,x)` `delete_at(1)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n
Sequence AVL	n	$\log n$	$\log n$	$\log n$	$\log n$

4.2 Sets

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(x)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n \log n$	$\log n$	n	1	$\log n$
Direct Access	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$	$\log n$

4.3 Priority Queues

Priority Queue Data Structure	Operations $O(\cdot)$			
	`build(x)`	`insert(x)`	`delete_max()`	`find_max()`
Dynamic Array	n	$1_{(a)}$	n	n
Sorted Dyn. Array	$n \log n$	n	$1_{(a)}$	1
Set AVL	$n \log n$	$\log n$	$\log n$	$\log n$
Binary Heap	n	$\log n_{(a)}$	$\log n_{(a)}$	1

5 Binary Trees

5.1 Traversal Order

For every node x, every node in x's left subtree is before x. Every node in x's right subtree is after x. Recursive.

5.2 Binary Search Trees

These have a key for each node, and are sorted by these keys. For every node i, all keys in left subtree $\leq i$'s key, which is less than or equal to every key in the right subtree. This way, they are returned in order when you traverse.

5.3 Rotations

A node is height-balanced if its **skew** = height(right subtree) - height(left subtree) is -1, 1, or 0. AVL trees can be balanced using rotations! Rotations preserve the traversal order while changing the depth of the subtrees. Rotate right is moving from left to right below, and rotate left is moving right to left.

5.4 Augmentations

Augmentations need to be based on the subtree nodes, and be stores at each node. To augment, state the subtree property $P(x)$ you want to store at each node x, and show how to compute $P(x)$ based on augmentations of x's children in $\mathbf{O(1)}$ time.

6 Recurrences

6.1 Master Theorem

Used for solving recurrences where recursive calls decrease by a constant factor. Given a recurrence relation of the form $T(n) = aT(n/b) + f(n)$, we can apply the below, given a few restrictions:

- branching factor $a \geq 1$
- problem size reduction factor $b > 1$
- asymptotically non-negative $f(n)$.

case	solution	conditions
1	$T(n) = \Theta\left(n^{\log_b a}\right)$	$f(n) = O\left(n^{\log_b a - \varepsilon}\right)$ for some constant $\varepsilon > 0$
2	$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$	$f(n) = \Theta\left(n^{\log_b a} \log^{k} n\right)$ for some constant $k \geq 0$
3	$T(n) = \Theta(f(n))$	$f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$ for some constant $\varepsilon > 0$ and $af(n/b) < cf(n)$ for some constant $0 < c < 1$

When $f(n)$ is polynomial, such that the recurrence has the form $aT(n/b) + \Theta(n^c)$ for some constant $c \geq 0$:

case	solution	conditions	intuition
1	$T(n) = \Theta\left(n^{\log_b a}\right)$	$c < \log_b a$	Work done at leaves dominates
2	$T(n) = \Theta\left(n^c \log n\right)$	$c = \log_b a$	Work balanced across the tree
3	$T(n) = \Theta\left(n^c\right)$	$c > \log_b a$	Work done at root dominates

6.2 Substitution

Basically just guess and plug in your guess times a constant for $T(n)$. Like if you're guessing $O(n)$, put cn for $T(n)$ and $c(\frac{n}{2})$ for $T(\frac{n}{2})$. Guessing $T(n) = O(n \log n)$ for $T(n) = 2T(n/2) + O(n)$:

$$T(n) = n \log n \longrightarrow cn \log n = O(n) + 2c(\frac{n}{2}) \log \frac{n}{2} \longrightarrow cn \log 2 = O(n)$$

7 Graph Algorithms

Restrictions		SSSP Algorithm		
Graph	Weights	Name	Running Time $O(\cdot)$	Notes
DAG	Any	DAG Relaxation	$\|V\| + \|E\|$	Relax in topo ordering
General	Unweighted	BFS	$\|V\| + \|E\|$	
General	Non-negative	Dijkstra	$\|V\| \log \|V\| + \|E\|$	With Fibonacci heap PQ
General	Any	Bellman-Ford	$\|V\| \cdot \|E\|$	Duplicate $\rightarrow$ DAG Relax
General	Any	Johnson's (**APSP**)	$\|V\|^2 \log \|V\| + \|V\|\|E\|$	B-F then $\|V\|$ Dijkstras

BFS searches in levels of increasing distance, and does not revisit seen nodes. Does not explore all paths. Returns parent tree of shortest paths.

DFS touches all nodes reachable from s, does not revisit nodes. Returns parent tree, but not shortest. Can find cycles by checking if edges match topo order.

DAG Relaxation finds a topo ordering, then relaxes edges in that order. Relax (u, v, x) means $\delta(u, v)$ becomes min of $\delta(u, v)$ and $\delta(u, x) + w(x, v)$. Enforces triangle ineq.

Dijkstra explores all edges from source, then from each subsequent node in order of node's distance from source. Order of node exploration mimics water flowing. Whenever source node is found, must be shortest path. $O(|V|^2)$ w/ array PQ, $O(|E| \log |V|)$ w/ bin. heap PQ.

Bellman-Ford structures graph into a form usable by DAG relaxation. Duplicates $|V|+1$ times (max $|V|$ edges), connects each v to self and adj. vert. in next level. Prunes G to reachable subgraph for $|V| = O(|E|)$. If $\delta_{|V|} < \delta_{|V|-1}$, must be neg. wt. cycle.

Johnson's runs B-F from supernode x with $w(x, v) = 0 \ \forall v \in V$ to find *potential* of each node $h(v) = \delta(S, v)$. Removes x and reweights $w(u, v) = w(u, v) + h(u) - h(v)$ for all edges. Weights now non-neg., so run Dijkstra from each v to find δ', then $\delta(u, v) = \delta'(u, v) - h(u) + h(v)$.

7.1 Problem Types

- **Single Source Reachability**: finds which vertices are accessible from a source. Solved by BFS/DFS.
- **Cycle Detection**: determining whether a cycle exists in a graph. Solved by DFS looking for back edges.
- **Connected Components**: find subgraphs of G that are connected. Solved by Full-DFS/Full-BFS.
- **Single Source Shortest Paths (SSSP)**: finds the shortest paths to every v in a graph G from a source s.
- **All Pairs Shortest Paths (APSP)**: finds the shortest paths between every pair of nodes u, v in G.

7.2 Definitions

- **Simple Path**: a path that does not repeat any vertices.
- **Full-{DFS/BFS}** repeats DFS/BFS until all nodes have been touched.
- **Connected Graph**: an undirected graph in which there is a path between every pair of nodes.
- **Strongly Connected**: every node is reachable from every other. Note this implies $|E| = \Omega(|V|)$.
- **Dense Graph**: defined by $|E| = \Omega(|V|^2)$.
- **Sparse Graph**: defined by $|E| = O(|V|)$. **Planar Graphs** are sparse.
- **Directed Acyclic Graph (DAG)**: Contains no directed cycle.
- **Finishing Order**: The order in which a Full-DFS *finishes* visiting each vertex in G.
- **Topological Order:** an ordering f such that every edge $(u, v) \in E$ satisfies $f(u) < f(v)$.
  - The reverse of a *finishing order* is a topological order.

7.3 Tricks

- **Graph Duplication**: for any problem with *state*, graph duplication is a good idea. Create one copy of the vertexes for each state, and connect them according to state transitions. Then, run shortest path algorithm.
- **Supernodes**: creating a "supernode" with edges to mulptiple other nodes in the graph can speed up many algorithms instead of running a SP alg from each node. However, lose specificity.
- **Constant Bounded Weights**: if the weights in a graph are bounded by $O(1)$, SSSP can be solved with BFS by "splitting" edges into multiple edges between dummy vertices to create strictly weight-1 edges.
- **Reversing Edges**: to find shortest paths *to* a single node, can reverse all path directions and run SSSP.

8 Dynamic Programming

8.1 General Approach

Dynamic programming problems can be described using the SRTBOT acronym.

Subproblem: Define the subproblem(s) that must be calculated in English.

- this should always be like a number or T/F value, not the path to get there.
- if using multiple subproblems, it can often be simpler to combine them into a single subproblem with an additional parameter.
- often relies on an "index" parameter.
- typically prefix/suffix/substring

Relate: Define the recursive relationship between subproblems mathematically; the way that each subproblem relies on a smaller subproblem.

Topological Order: Show that the graph formed by the subproblems' dependencies is acyclic.

- often enough to say "i is strictly decreasing" or "$i+j$ is strictly increasing".

Base Cases: Define the cases where you know the answer to the subproblem without relying on other subproblems. Typically when the "index" reaches the end/beginning.

Original Problem: Give the subproblem or operation between multiple subproblems that give the answer to the original problem.

- can be like a max over all possible starting positions
- specify top-down or bottom-up
- specify using parent pointers if need the actual path.

Time Analysis: Find the time of the whole operation

- # of subproblems times time per subproblem
- make sure to account for solving original

8.2 Examples

8.2.1 Longest Common Subsequence

Given two strings A and B, find a longest subsequence of A that is also a subsequence of B.
Idea: keep starting indices in A and B, check first letters. If match, in subsequence, if not, try next letter in each.

9 Complexity

Focus on **decision problems**: assignment of input to yes/no. A decision problem is **decidable** if there exists constant length code to solve the problem in finite time.

9.1 Classifications

We classify decision problems into three main classes where $\mathbf{R} \subsetneq \mathbf{EXP} \subsetneq \mathbf{P}$, with input size n:

- R: broadest, all problems decidable in finite time.
- EXP: problems decidable in exponential time $2^{n^{O(1)}}$
- P: problems decidable in polynomial time $n^{O(1)}$

There is another class of problems called **NP** for Nondeterministic Polynomial Time. Regardless of how long it takes to solve the problem, problems in **NP can be verified in polynomial time**:

- Given an input consisting of an instance I of the problem, and a certificate c with length $O(|I|^k)$, there exists an algorithm to verify c, returning whether the instance is a YES or a NO in $O(|I|^k)$ time.
- For example, the certificate of checking whether a shortest path with length $< d$ exists would be the shortest path itself.
  - If YES and c is correct, c can be verified as a correct solution easily
  - If YES but c is invalid (ie a path longer than d), return no
  - If NO, all c are invalid.
- $\mathbf{P} \subset \mathbf{NP}$, but unknown if $\mathbf{P} = \mathbf{NP}$. So we don't know if there are problems that can be checked in polynomial time, but cannot be solved in polynomial time.

9.2 Reductions

Problem A can be solved by converting it into a problem B you know how to solve. Called **reducing** A to B.

- The solution to some instance of B can be used to find a solution to A.
- B must be **at least as hard** as A then.
- Some problems can be reduced to each other, meaning they are equal in hardness.

9.3 NP-Complete and NP-Hard

A problem is **NP-Hard** if all problems in NP can be reduced to that problem, and the reduction itself takes polynomial time. That means it's **at least as hard** as every problem in NP.
If a problem is in NP **and** is NP-Hard, it is termed NP-Complete.

- All NP-Complete problems are reducible to each other, equivalent in hardness.
- NP-Complete problems are the hardest problems in NP.