Dijkstra's Shortest path from Source

Claim 1 $\forall v \in V$, $d(v)$ is non-increasing through the iterations
Claim 2 If u is added to R in iteration i, then $d_i(u) = d_{i-1}(u)$, or that once u is added to R, $d(u)$ does not change
Claim 3 At the end of iteration i, if $d_i(v) = k \neq \infty$, then there is an $R - path$ (path within R) $s \to v$ of weight k
Claim 4 If $d(u) = \infty$ when u is added to R, then we have no path from s to u in the graph.
Claim 5 If u is added to R in iteration i, then $d_i(u) = \delta(u)$, or that $d(u)$ has correct value.
Running Time: $O(m \log n)$, but $O(n^2 \log n)$ if graph is dense

Fractional Knapsack

Sort items in value/wt ratio and take entire items until the next item exceeds C, then take fractional of the next item. Let G be the fractional amount to take each $i-th$ item, a sample output of the solution could be $G = (1, 1, \ldots x_k, 0, 0, \ldots, 0)$

Huffman

Code maps a symbol to a binary string. **Fixed-length code** all have the same length. **Variable-length code** may have different lengths. However, we need that no code is a prefix of another. Represent this using a **tree** where going to L-child equates to 0 and going to R-child equates to 1. All symbol then, will be placed on the leaf nodes.
Give each symbol a frequency f and a depth, then the average depth (where we expect a typical symbol to be) of the tree $AD(T) = \sum_{a \in \Gamma} f(a) \cdot depth_T(a)$, then we would expect $AD(T)$ bits to encode the alphabet. Huffman **minimizes** this $AD(T)$, but know that Huffman does not give a **unique** tree, but does give an optimal tree in $O(n \log n)$.
Algorithm: order nodes of the tree in non-decreasing frequency into a list. Take 1st and 2nd node from this list and combine them into a tree such that the root node has $f(1+2) = f(1) + f(2)$. Now remove nodes 1 and 2 and place this root node back into the list (according to its frequency)and repeat the process until we have a full tree.

DC Algorithm

If (n is small) then solve directly. Else split input into a pieces, each of size $\frac{n}{b}$ where ($a \geq 1, b > 1$ are constants). Recursively solve each of the a subproblems, then combine soln to these subproblems into soln of original problem.

Master Theorem

$T(n) = a \cdot T\left(\frac{n}{b}\right) + c \cdot n^d$, If $a < b^d$, then $T(n) = O(n^d)$, If $a = b^d$, then $T(n) = O(n^d \log n)$, If $a > b^d$, then $T(n) = O(n^{\log_b a})$
Notice that $a \cdot T\left(\frac{n}{b}\right)$ is the time to solve a instances each of size $\frac{n}{b}$, and $c \cdot n^d$ is merging time for each recursive call

DC Algorithm Run times

Karatsuba is $O(n^{\log_2 3})$ where n is number of bits. Strassen's is $O(n^{\log_2 7})$ for $n \times n$ matrix

Closest Pair of Points

Running Time: $T(n) = 2 \cdot T\left(\frac{n}{2}\right) + cn = O(n \log n)$

Order Statistics

Ordering elements in non-decreasing order like $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$, then we have the median is $a_{\pi(\lceil \frac{n}{2} \rceil)}$
SELECT(A, k) finds the k$^{\text{th}}$ smallest element in A using $O(n)$ worst case by carefully choosing a splitter.

DP Algorithms

LIS: Longest inc subseq in a seq. $L(i) = 1 + max\{L(j) : j < i \cap a_j < a_i\}$ if $\exists j < i$ s.t. $a_j < a_i$ len of LIS ending at pos i.
Weighted Interval Scheduling: Feasible subset of intervals w/ max value. $V(i) = max\{V(i-1), V(p(i) + v(i))\}$
ED: Given 2 strings, min edits to make them the same. $ED(m,n) = min\{ED(i-1, j-1) + diff(i,j), ED(i-1, j) + 1 \ldots\}$
Optimal BST: $C[i,j] = min_{i \leq k \leq j}(C[i, k-1] + C[k+1, j]) + \sum_{i \leq u \leq j} P(u) \to$ Optimal cost for tree w/ nodes i to j
Discrete Knapsack: With $c \leq C$, $i \leq n$, $K(i,c) = max\{K(i-1,c), v(i) + K(i-1, c-w(i))\}$ Take or don't take.
Weighted: $W(i,v) = W(i-1, max(0, v - v_i)) + w_i$ $v > V_{i-1}$ $W(i,v) = min(W(i-1,v), W(i-1, max(0, v-v_i)) + w_i)$ ow

Bellman-Ford (Single Source)

Input: Directed graph, source node s, weight function. **Output**: wt of shortest $s \to u$ path $\forall u \in V$
(†): $L[u,k] = min_{v:(v,u) \in E}\{L[v, k-1] + wt(v,u)\}$ where $L[u,k]$ is the wt of the shortest $s \to u$ path with at most k edges.
Fact 1: If G has no neg-wt cycles reachable from s, then $\forall u$ $\exists$ a shortest $s \to u$ path that has at most $n-1$ edges.
Fact 2: G has no neg-wt cycles reachable from $s \iff \forall u$, $L[u,n] = L[u, n-1]$. (Detect neg-wt cycles reachable from s)
Fact 3: If $L[u,n] \neq L[u, n-1]$ then if P is a min wt $s \to u$ path using at most n edges, then P has cycle with $wt < 0$.
Running Time: $O(n^3)$ for adj matrix. $O(n \cdot m)$ for sparce graph, $O(n^3)$ for dense graph with adj list.

Floyd Warshall (All Pairs)

Input: Directed graph without neg-wt cycles, weight function. ***Output***: $\forall u, v \in V$ give $\delta(u,v) =$ min wt of $u \to v$ path.
(†) : $C[i,j,k] = min\{C[i,j,k-1], C[i,k,k-1] + C[k,j,k-1]\}$ where $C[i,j,k]$ is similar to BF and k is intermediate node.
Running Time: Trivially $O(n^3)$ with space complexity of $O(n^3)$
Fact 4: G has neg-wt cycle $\iff \exists u : C[u,u,n] < 0$ (Use to detect neg-wt cycle)
Transitive Closure: If $C[i,j,n] \neq 0$, then $\exists$ $i \to j$ path and thus j is called by i

Johnson's Algorithm (All-pairs Non-negative Edges) $\to$ **[make edges positive so we can use Dijkstra's]**

Define new weight $wt'(u,v) = wt(u,v) + x_u - x_v$ where $\forall u \in V$, $x_u =$ wt of shortest $s \to u$ path in G', and G' has one extra node s that has a di-edge of wt 0 to every other edge in V. Get x_u using $BF(G', s, wt)$. Run Dijkstra on all nodes.
Running Time: $O(n \times m \log n)$ for sparse graphs. (So running Dijkstra's n times) - **Don't forget to revert edges**

Max Flow & Min Cut Problem - Ford Fulkerson

Flow Network: $F = (G, s, t, c)$ source, target, and capacity for each di-edge. Each edge is defined by flow/capacity

Flow: $f \in F$ a map $f : E \to R^{\geq 0}$ that satisfies **capacity** and **conservation**
Max Flow Problem (Ford-Fulkerson): Given a flow network F, find max flow f (counted as sum of **flow** out of s)
Cut: Cut of F is a pair (S, T), partition of nodes in V where $s \in S$, $t \in T$ and $S \cap T = \emptyset$
Min Cut Problem: Given a flow network F, find cut (S, T) of minimum capacity (**capacity** of edges out of S into T)
Finding Min Cut: Consider the residual graph for max flow f, then $S \leftarrow$ nodes reachable from s in G_f and $T \leftarrow$ Rest
Residual Graph G_f: For every edge in G create FW edge (u, v) with residual $c(u, v) - f(u, v)$ (able to be pushed forward) and create BW edge (v, u) with residual $f(u, v)$ (able to be pushed back). **Ford-Fulkerson**: for every $s \to t$ path on the residual graph, push back the bottle-neck amount (minimum FW/BW residual on the path) until no more $s \to t$ paths.
Running Time: $O(mC)$ where $C =$ sum of capacities out of s. Picking path with max bottleneck each time: $O(m^2 \log C)$ (Widest - Polynomial). Picking path with minimum edges each time: $O(m^2 n)$ (Shortest - Strongly Polynomial).
Max-Flow Min-Cut Thm: V(max flow)=C(min cut). In addition, $V(f) \leq C(S, T)$ for any flow and cut
Integrality Thm: If capacities are integers, then $\exists$ max flow where every edge has an integer traffic. **(Integral Flow)**

Bipartite Matching → [Reduction of Bipartite Matching to Max Flow]
Bipartite Graph is a undirected graph $G = ((X, Y), E)$ and all edges connect $x \in X$ to $y \in Y$ but not within themselves
Matching is $M \subseteq E$ s.t. no node appears in 2 edges in M. We want maximum cardinality M as output.
Algorithm: Using di-edges, create s and connect s to each $x \in X$ and connect each $y \in Y$ to t. For each edge from $X \to Y$, convert into di-edge and assign it capacity of 1. Run FF and by integral flow, we have integer flow. Then $M = \{\{x, y\} : x \in X, y \in Y, f(x, y) = 1\}$
Running time: equal to $\Theta(m \cdot C) = \Theta(m \cdot n)$.
Max Matching Theorem: Integral flow of value k in $F \iff$ Matching of size k in G

Min Vertex Cover in Bipartite Graph
Vertex Cover: Given undirected graph $G = (V, E)$, vc is $V' \subseteq V$ s.t. every edge $e \in E$ has at least one endpoint in V'
Algorithm: Construct F from G like previously, find max flow $f \in F$, find min cut $(S, T) \in F$ and return $(X \cap T) \cup (Y \cap S)$
Running Time: $O(mC) = O(mn)$
Max-Match Min-Cover Thm: Cardinality of Max Match $|M|$ = Cardinality of Min VC $|R|$ and $|M| \leq |R|$
Hall's Thm: $\exists X' \subseteq X$ where $|N(X')| < |X'|$ where $N(X') = \{y : (x, y) \in E \& x \in X'\} \iff G$ has no perfect matching

Max Set of Edge-Disjoint Paths
Input: directed graph $G = (V, E)$. ***Output***: Max cardinality set of edge-disjoint $s \to t$ paths (no edge in common).
Edge-Disj Thm: integral flow of value k in $F \iff$ set of k edge-disj simple $s \to t$ paths in G
Algorithm: Reduce to max flow problem with capacity 1 for each edge. Trace flow from $s \to t$. **1.** If you reach t, then you found a path, so remove flow along the edges of the path. $V(f) = V(f) - 1$ **2.** If you found a loop, remove the flow along that loop. $V(f) = V(f)$. Repeat until $V(f) = 0$
Running Time: $O(m \cdot C) = O(m \cdot n)$ (Since there is at most m paths and n edges on each path)

Linear Programming
Variables: $x_1, \ldots, x_n$ amount of something to take/make/do. ***Obj Function (linear)***: Max/min $\sum_{i=1}^{n} c_i x_i$ for const c.
Constraints: Define **linear** constraints on the variables. Set of feasible solutions is a **convex polygon** with max # of sides equal to # of constraints. Optimal soln exists → Optimal soln on the corner of the feasible region (basic).
Unbounded/Infeasible: Unbounded means no max/min (∞ or $< / >$). Infeasible means no soln satisfies constraints.
Linear Algebra: Max/min $c_1 x_1 + c_2 x_2 + \ldots + c_n x_n = \vec{c} \cdot \vec{x}$ such that $A\vec{x}[\geq / \leq]\vec{b}$ (Satisfy constraints)
Reduction to LP: Line fitting (define error function). Abs val obj fcn: define y_i: $|x_i| \to y_i \geq x_i$ and $y_i \geq -x_i$ constraints
0-1 LP: NP-hard problem that has $x_j \in \{0, 1\}$ as a constraint (non-linear constraints)

Approximation
c-approximation: for $c > 1$, $opt \leq$ soln $\leq c \cdot opt$ (minimization) and $opt \geq$ soln $\geq \frac{1}{c} \cdot opt$ (maximization)
PTAS: for $\epsilon > 0, c = 1 + \epsilon$, $opt \leq$ soln $\leq (1 + \epsilon) \cdot opt$ (minimization) and $opt \geq$ soln $\geq \frac{1}{1+\epsilon} \cdot opt$ (maximization) $O(? \cdot (\epsilon))$
Min VC (0-1 LP): Relax $x_v \in \{0, 1\} \to x_v \geq 0, x_v \leq 1$ (linear). Round to 1 if $x_v \geq \frac{1}{2}$ 0 o/w. 2-approx (1.361 best)
Set Cover: Cover universe with least amount of sets (special case is vertex cover). Greedily choose sets that covers the most uncovered elements at each step. $L \leq$ soln size $\leq L \cdot \ln n$ where L is optimal size ($c = \ln n$)
Weighted Set Cover: Maximize the ratio of most uncovered elements to weight of set. Same approximation ratio.
Min Makespan (Load Balancing): Given machines and jobs (takes time t to complete). Want to balance the load (total time) between every machine so we minimize the max load on any machine. Do this by putting next job on least loaded machine at each step. $c = (2 - \frac{1}{m})$ where m is the # machines. Can also sort jobs in decreasing length ($\frac{4}{3} - \frac{1}{3m}$).
k-centre Problem: Given set S of sites under a metric space M and k, want C set of centres to minimize radius of the centres (center to most inconvenienced site). Approximate by choosing random site as centre, then at each step, assign the most inconvenienced site as the new centre, shrink radius according to the new most inconvenienced. (2-approximation)
Discrete Knapsack: Perform the weighted DP solution, but chop off least significant digits from the wt of each item. Can choose ϵ so that $c = 1 - \epsilon$ and running time is $O(\frac{n^3}{\epsilon})$. Scale weight using $\hat{v_i} \leftarrow \left\lfloor \frac{v_i}{v_{max}} \cdot \frac{n}{\epsilon} \right\rfloor$
Max Cut: Given undirected graph, partition $V \to (X, Y)$ so that the cut has max # of edges. Use **local search**, move nodes across the cut and see if we increase the net # of cross edges. Terminate when there isn't a node we can move to increase the net # of cross edges. 2-approximation with running time of $O(m(n + m))$. Weighted - pseudopolynomial