Mathematical shorthands
$\min(a_1, \dots, a_n) \Rightarrow \min_{i:i \leq n}(a_i)$
Given values $a_1, \dots, a_n$ find a set S of $\{1, \dots, n\}$ such that $\sum_{i \in S} a_i$

Greedy

Inductive proof concepts: Show that greedy is optimal for a set of sub-cases, show that large problems can be broken down into those sub-cases. Often, proof is done by contradiction (suppose optimal < greedy …)

Dijkstra's algo: Initialize set of explored nodes S and array storing shortest path cost to those nodes d. Repeatedly choose an unexplored node $v \notin S$ where $\pi(v) = \min_{\mathrm{e=(u,v):u \in S}} d[u] + w(u, v)$, add v to s and set $d[v] = \pi(v)$

Cashier's algo: Prove optimality for sub-cases, then extend to larger cases

Huffman encoding: Understand bottom-up construction of tree starting with the least occurring nodes, combining as you go. Repeat until all nodes have been combined under a single tree

Divide & conquer

Master's Theorem: Applies to recurrences of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1, b > 1$ are constants and $f(n)$ is an asymptotically growing function (Note: Unless otherwise stated, log is base 2)
1. If $f(n) = \mathrm{O}\left(n^{\log_b (a-\epsilon)}\right)$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta\left(n^{\log_b a}(\log^k n\right)$ with $k \geq 0$, then $T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$ – note that $\log^k n = (\log n)^{\mathrm{k}}$
3. If $f(n) = \Omega\left(n^{\log_b (a+\epsilon)}\right)$ for some $\epsilon > 0$, and $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n), c < 1, \forall n > n'$ then $T(n) = \Theta(f(n))$

Median of medians: Recursive calling between 2 functions – *SELECT()* and *CHOOSEPIVOT(). SELECT()* performs sorting and selection if array is below a certain size. Otherwise, it calls *CHOOSEPIVOT()*, which splits the array into $m = \left\lceil \frac{n}{5} \right\rceil$ groups and chooses the median out of each of them. Overall complexity is $\mathrm{O}(n)$

Dynamic programming

Iterative approach: Usually involves 2D arrays and using solutions to smaller problems to solve larger ones
Recursive approach: Backtrack with memorization
Both involve identifying an optimal substructure and coming up with a recursive formulation

Knapsack problems: 0-1, unbounded

Flow

Ford-fulkerson (greedy): Given graph G and flow f, residual graph G_f has the same nodes as G. For any edge $(u, v) \in G$, include edge (u, v) with capacity $c'_{u,v} = c_{u,v} - f_{u,v}$ and (v, u) with capacity $c'_{v,u} = f_{u,v}$ if $f_{u,v} > 0$. Continuously try to push more flow from s to t in residual graph until t is unreachable from s

Complexity is $\mathrm{O}(|E|f)$ where f=optimal flow value. DFS is $\mathrm{O}(|V| + |E|)$ but in a connected graph $|E| \geq |V| - 1$, each iteration improves flow by at least 1.

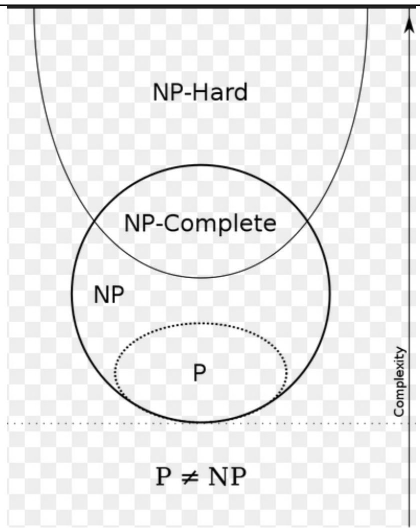
Dummy nodes can be introduced for cases of multiple sources & sinks. Applicable to problems like bipartite matching and max circulation

Edmonds-karp: Uses BFS to find augmenting path, runs in $\mathrm{O}(|V||E|^2)$

Max-flow, min-cut: Max flow for a graph is equivalent to min-cut

Computational limits

Reduction: We can reduce problem X to problem Y by doing a transformation from X to Y, and using a solver for Y to find a solution, then transforming the solution back to a version in X.
- X reduces to Y is denoted as $X \leq_P \ Y$ (transitive
Optimization: Minimize or maximize certain metrics
Search: Answer a question with evidence
Decision: Give a boolean answer to a question
Optimization>Search>Decision
P problems are solvable in polynomial time
NP problems have solutions that can be *verifiable* in polynomial time *(but we cannot verify the absence of a solution in polynomial time)*
NP-complete if all other NP problems that can be reduced to them
NP-hard problems are at least as hard as NP-complete problems, and are not necessarily in NP



Linear programming
General form: Maximize/minimize a certain metric while conforming to constraints. Inequalities in constraints can be converted to equalities using *slack variables*. Solution is guaranteed to lie at a point on the feasible region, if one exists
ILP (Integer Linear Programming): Restricted to integers/binary instead of real numbers
Duality: Drawing on the concepts from linear algebra, the *dual* of an LP can be derived from the *primal* as such: - Each variable in the primal becomes a constraint in the dual - Each constraint in the primal becomes a variable in the dual - Objective direction is inversed, maximum in the primal becomes minimum in the dual **Symmetry and asymmetry**

Primal (Maximize)	*Dual (Minimize)*
ith constraint $\leq$	ith variable ≥ 0
ith constraint $\geq$	ith variable ≤ 0
ith constraint $=$	ith variable unrestricted
jth variable ≥ 0	jth constraint $\geq$
jth variable ≤ 0	jth constraint $\leq$
jth variable unrestricted	jth constraint $=$

Strong duality if both the primal and the dual have optimal solutions. If x^* and y^* are solutions to the primal and dual, then $c^T x = b^T y$
Approximation
k-approximation: For maximization, for any problem instance l, the algo produces a solution $A(l)$ such that $OPT(l) \geq A(l) \geq \frac{1}{r} \cdot OPT(l)$. For minimization, $OPT(l) \leq A(l) \leq r \cdot OPT(l)$
Relaxation in the context of ILPs means allowing real numbers instead of integer values. Will give lower minima and higher maxima in feasible region.
TSP approximation: (For TSP problem instances where the triangle inequality holds) - Find an MST of the graph, choose an arbitrary vertex as root, and return a preorder walk (2-approximation)
Heuristics and randomization
Local search: Define a neighbourhood around a solution, involving slight permutations. A *k-opt* neighbourhood might involve making *k* adjustments to the current solution to form a new solution. This can be done multiple times to make incremental improvements
Max-cut local search: Arbitrarily partition vertices into 2 sets. Loop through all vertices. If a vertex can be switched to the other side to increase crossing edges, do so. Repeat until no improvements can be made. $O(\|V\|\|E\|)$
Randomized algos
Finding median: Upon randomly sampling S containing $\frac{1}{\epsilon}\log\, n$ elements from an input array, we can sort S with $O(\frac{1}{\epsilon}\log n \cdot \log(log\, n))$ complexity, and guarantee an element between $(1-\epsilon)\frac{n}{2}$ to $(1+\epsilon)\frac{n}{2}$ with error probability n^{-2}. Note that as n grows larger, error probability decreases
Las Vegas solutions have a *deterministic* output and *variable* runtime **Monte Carlo** solutions have a *variable* output and *deterministic* runtime
Bernoulli trial: For a Bernoulli distribution with probability of success p, expected tries for first success is $\frac{1}{p}$
Law of total expectation: $E[X] = \sum_i P(A_i)E[X\|A_i]$