Bounds

$f(n) = O(g(n))$. if there exists a constant $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. $O(g(n))$ is actually a set of functions, more formally, $f(n) \in O(g(n))$

$f(n) = o(g(n))$. if there exists a constant $c > 0$, $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.

$f(n) = \Omega(g(n))$. if there exists a constant $c > 0$, $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

$f(n) = \omega(g(n))$. if there exists a constant $c > 0$, $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

$f(n) = \Theta(g(n))$. if there exists constants $c_1 > 0$, $c_2 > 0$, $n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

Proof for upperbound. Claim: $2n^2 = O(n^3)$. Proof: Let $f(n) = 2n^2$. Then $f(n) \leq n^3$ when $n \geq 2$. Set $c = 1, n_0 = 2$, then $f(n) = 2n^2 \leq c \cdot n^3$ for $n \geq n_0$.

$\lim_{x\to\infty} \frac{f(n)}{g(n)} = 0 => f(n) = o(g(n))$.

$\lim_{x\to\infty} \frac{f(n)}{g(n)} < \infty => f(n) = O(g(n))$.

$0 < \lim_{x\to\infty} \frac{f(n)}{g(n)} < \infty => f(n) = \Theta(g(n))$.

$\lim_{x\to\infty} \frac{f(n)}{g(n)} > 0 => f(n) = \Omega(g(n))$.

$\lim_{x\to\infty} \frac{f(n)}{g(n)} = \infty => f(n) = \omega(g(n))$

Transitive. $f(n) = O(g(n)), g(n) = O(h(n)) => f(n) = O(h(n))$.

Reflexive. Only the big notations are $f(n) = O(f(n))$.

Symmetric. For Θ, $f(n) = \Theta(g(n)) iff g(n) = \Theta(f(n))$.

Complementary. $f(n) = O(n)$ iff $g(n) = \Omega(f(n))$, $f(n) = o(n)$ iff $g(n) = \omega(f(n))$

Degree-k polynomials. $O(n^k), o(n^{k+1}), \omega(n^{k+1})$.

Expoential dominate polynomials (when base is greater than 1), polynomials dominate logs.

$O(2^{5n}) \neq O(2^n)$. $max(f(n), g(n)) = \Theta(f(n) + g(n))$.

$e^x \leq 1 + x$. $a^{log_b c} = c^{log_b a}$. $2^{lg\ lg\ n} = O(lg\ n)$.

$log(n!) = \Theta(nlogn)$. $\sqrt{n} = O(lg\ lg\ n)$

GP Sum. $\sum_{k=1}^{n} x^k = 1 + x + x^2 \ldots = \frac{x^{n+1}-1}{x-1}$

Harmonic series. $1 + \frac{1}{2} + \ldots + \frac{1}{n} = ln\ n + O(1)$

L'Hospitals Rule. $\lim_{x\to\infty} \frac{f(n)}{g(n)} = \lim_{x\to\infty} \frac{f'(n)}{g'(n)}$

Correctness of Iterative Algos

Find loop invariant.

Initialisation. Show invariant is true before entering loop.

Maintenance. Show invariant is maintained throughout the loop.

Termination. When exit, show invariant provide useful property for correctness.

Recursive Algos

Correctness. Use strong induction. Prove base cases and show algo works correctly assuming it works for small cases.

Running Time. Derive and solve a recurrence using recurrence tree, master and substitution.

General Form. $T(n) = aT(\frac{n}{b}) + f(n)$, where a is number of subproblems, b is size of subproblem and $f(n)$ time taken at each layer to solve problem of size n.

Substitution. Guess the form of the solution and proves it via induction.

Master

Must satisfy one of the case for it to be used by master.

Case 1. $T(n) = \Theta(n^{log_b a})$, where $f(n) = O(n^{log_b a - \epsilon})$, where $\epsilon > 0$.

Case 2. $T(n) = \Theta(n^{log_b a} log^{k+1} n)$, where $f(n) = \Theta(n^{log_b a} log^k n)$ and $k \geq 0$.

Case 3. $T(n) = \Theta(f(n))$, where $f(n) = \Omega(n^{log_b a + \epsilon})$, having $\epsilon > 0$ and $f(n)$ satisfy the regularity condition that $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$.

Regularity condition guarentees sum of subproblems is smaller than $f(n)$.

Common occurances that cannot use master: Trigometric functions, multiple unequal T terms, $\sqrt{n}$ terms.

Randomised QuickSort

Assumption: ties broken arbitrarily. e_i is ith smallest element in array. T_π be number of comparisons when for permutation π. T_π is a random variable.

$\mathbb{E}(n) = max(\mathbb{E}[T_\pi])$ be worse case expected run time.

$T(\pi) = n - 1 + T_{\pi_l} + T_{\pi_r}$. Randomly choose pivots and recursive sorting of left and right arrays.

Indicator Random Variables

$X = 1$ with probability p and $X = 0$ with probability $1 - p$. $E[X] = p$, expected value of IRV is the probability of $X = 1$. Common steps:

- Identify what to count in expectation, X
- Express as X as a sum of indicator random variables, $\sum_{i=1}^{n} X_i$
- Calculate $Pr(X_i = 1)$ to get $E[X_i]$
- Apply linearity of expection to get $E[X]$

Bayes Formula. $Pr(A_i \mid B) = \frac{Pr(B|A_i)Pr(A_i)}{Pr(B|A_1)Pr(A_1)+\ldots}$.

Independence. $Pr(A \cap B) = Pr(A)Pr(B)$.

Conditional. Probability of A given B, $Pr(A|B)$.

If $E[X] = \frac{n}{m} + \frac{n}{m-1} \ldots + \frac{n}{m-q+1}$ rewrite to the following: $E[X] = (\frac{n}{m} + \frac{n}{m-1} \ldots + \frac{n}{m-q+1} + \frac{n}{m-q} + \ldots + \frac{n}{1}) - (\frac{n}{m-q} + \ldots + \frac{n}{1})$ and use harmonic sequence to solve. $E[cX] = cE[X]$.

Bernoulli Trial. Generalised coin flip, probability of success and failure must remain the same across different trials. Expectation of Bernoulli Trial is $E[X] = \frac{1}{p}$, where p is probability of success.

Hashing

Hash Function. Gives location of where to store element in universe in table.

Goal. Minimise collisions : $h(x) = h(y)$. Should map to random value. Minimise storage space : $M = O(N)$, where N is number of stored items. Randomly pick hash function from family of hash functions, hash function itself **not random**.

Universal Hashing

Suppose $\mathcal{H}$ is a set of hash functions mapping $\mathcal{U}$ to [M]. $\mathcal{H}$ is universal if for all $x \neq y$: $\frac{|h \in \mathcal{H} : h(x) = h(y)|}{|\mathcal{H}|} \leq \frac{1}{M}$. For any randomly chosen h, the probability of collision is $\frac{1}{M}$ if the hash family is universal.

Methodology. Count the number of hash functions that have collisions between 2 fixed variables. Take the number as a fraction over the number of hash functions and determine if it satisfy the condition. Given 2 universal hashing families, composing them may not result in a universal hashing function. Crossing them makes them universal though.

Pairwise Independence

$\mathcal{H}$ is pairwise independent if for any 2 distinct universe elements, x, y and for any 2 hash values i_1, i_2, $Pr[h(x) = i_1, h(y) = i_2] = \frac{1}{M^2}$. A pairwise independent family is always universal but not the other way.

Suppose $Pr[h(x) = i_1, h(y) = i_2] \leq \frac{1}{M^2}$, there cannot exist distinct $x, y \in \mathbb{U}$ and $i_1, i_2 \in [M]$ such that $Pr[h(x) = i_1, h(y) = i_2] < \frac{1}{M^2}$. The expected number of elements x_i such that $h(x_i) = 1$ is $\frac{N}{M}$ for a hash function drawn from an independent family. Upper bound for this would be N. Let $\mathbb{U}_{all}$ be the set of all functions from $\mathbb{U}$ to $1 \ldots M$, then $\mathbb{U}_{all}$ is pairwise independent. Expected number of elements such that there are no x_i such that $h(x_i) = z$ is $M(1 - \frac{1}{M})^N$

Properties.

- Universality does not imply anything about the hash function
- Uniformity means that for every element in $\mathcal{U}$ and for every hash value, $Pr(h(x) = i) \leq \frac{1}{M}$. There does exist a family $\mathcal{H}$ that is uniform but not universal. **Proof:** For every element in M, let there be a hash function h_i mapping all elements in $\mathcal{U}$ to i. The functions are uniform but not universal.
- For any element $x_i \in \mathcal{U}$, the expected number of collisions between x_i and other elements is $< \frac{N}{M}$. Proof by setting collision to be an indicator random variable, then linearity of expectations to get $\frac{(N-1)}{M} < \frac{N}{M}$
- For any sequence of N actions (insert, del, query), if $M > N$ then expected total cost of sequence is $O(N)$.
- Total number of collisions : if there are N insertions, expected number of pairs that collide is $< 2N$
- Maximum load bound (maximum number of elements hashed to a particular slot in table) : $\leq O(\sqrt{N})$

Construction of Universal Family

Purpose is to create a table that have all the buckets, ie size of $M = O(N)$. Suppose $\mathcal{U}$ indexed by u-bit strings, and $M = 2^m$. For any **binary matrix** A with m rows and u cols, $h_A(x) = Ax (mod\ 2)$ Ax is the matrix multiplication where A is the hash results of the respective functions and x is the element we intend to hash. The result would be the slot in the table to put x. $Pr(Ax = 0) \geq \frac{1}{M}$ is false since $x = 0$ always hash to 0.

Perfect Hashing

Using quadratic space to show that expected collisions would be less than 1. Instead, possible to use linear space with multi-level scheme. Choose a hash function, h from the family. Let L_k be number of x_i's for which $h(x_i) = k$. Choose second level hash function to hash without collisions.

Amortised Analysis

Aggregate Method. Calculate the average over all the operations. Does not give amortised cost of a particular operation.

Example - k-bit counter. Count total flips from 1 to 0 and vice versa. Want to get total number of flips during n increments, $T(n)$. Let $f(i)$ be number of times i bit flipped. Then $T(n) = \sum_{i=0}^{k-1} f(i)$. Since $f(i) = 2^{-i}$, then $T(n) < 2n$.

Example - Queues. Insert operation (1 element) and empty the entire queue. Number of deletes must be $\leq$ number of inserts. If there are k inserts then the total cost of all empty operations would be $\leq k$. Then total cost of n operations is $2k \leq 2n = O(1)$

Accounting Method. Amortised cost, $c(i)$, fixed for each operation while true cost, $t(i)$, varies depending on operation. $c(i)$ must satisfy : $\sum_{i=1}^{n} t(i) \leq \sum_{i=1}^{n} c(i)$ for all n. The total amortised cost provides an upper bound on the true cost. Normally $c(i)$ would be greater than $t(i)$ but occasionally the opposite is true. For cheap operations, pay more to accumulate credit in the bank. Upon reaching an expensive operation, pay using the bank. **Bank balance cannot drop below 0**. $c(i)$ can be 0 as long as the conditions are still satisfied.

Potential Method. ϕ : potential function of data structure, potential after i operation. Amortised cost, $c(i) = t(i) + \Delta\phi_i = t(i) + \phi(i) - \phi(i-1)$. $\phi(i) \geq 0$ for all values of i. Typically, $\phi(0) = 0$. During expensive operations, determine what quantity is decreasing. Use that quantity as part of the potential function! For multiple cases, the potential should be the same for all cases.

Dynamic Programming

Overlapping Subproblems. Recursive solution contains a small number of distinct subproblems repeated many times.

Optimal Substructure. Optimal solution to a problem instance that contains optimal solutions to subproblems.

Cut-and-Paste Argument. Suppose you came up with an optimal solution, to a problem by using suboptimal solutions to subproblems. Then, if you were to replace ("cut") those suboptimal subproblem solutions with optimal subproblem solutions (by "pasting" them in), you would improve your optimal solution. But, since your solution was optimal by assumption, you have a contradiction. What this means is that if S is our optimal solution, and S' is the optimal solution for some subproblem. Suppose that S' is not optimal for the subproblem and theres is something better, T. By improving T to T', we get $T' \neq S$, and this contradicts that S is optimal. Example using Knapsack: Suppose S is optimal and contains the ith item. Claim: Without i, $S - \{i\}$ is optimal for $n - 1$ items. Proof: Suppose T is optimal and weigh more than $S - \{i\}$. Then adding i to T would exceed the weight. So this contradict that T is optimal for subproblem.

Paradigm. Express the solution recursively. Overall there are only polynomial number of subproblems. But there is a huge overlap among the subproblems. So recursive algorithm takes exponential time (solving same subproblem multiple times). So compute recursive solution iteratively in a bottom-up

fashion. Avoid wastage of computation and leads to efficient implementation. Top-down fashion uses **memoisation** which writes intermediate results to a table and can be easily referenced.

Greedy Algorithms

Must have **Optimal Substructure** and **Greedy-choice Property**.

Greedy-choice Property. Possible to replace an item in the optimal solution and there would be no difference.

Cast the problem s.t. we have to make a choice and are then left with 1 subproblem to solve. Prove that there is always an optimal solution to the original problem that makes the greedy choice, so the greedy choice is safe. Use optimal substructure to show that we can combine an optimal solution to subproblem with the greedy choice to get an optimal solution to the original problem.

Simple Greedy Problem. Given n items with positive weights, find a set of k items that maximise total weights in the set.

Optimal Substruct. Suppose S is any optimal solution and S contains i. Claim: $S - \{i\}$ is optimal for the subproblem with i remove and size limit $k-1$. Proof: If T is optimal for the subproblem and weighs more than $S - \{i\}$, then adding i would weigh more than S. Contradict that T is optimal.

Greedy Choice. Suppose i has max weight. Claim: There exists an optimal solution that contains i. Proof: **Suppose otherwise**. Then can replace heaviest item in S with i. Total weight don't decrease so new set is optimal and contains i.

Flow Networks

Directed graph $G = (V, E)$ in which every edge has a capacity ≥ 0. Distinguised vertices are source and target. Assumption is that there are no self loops and no anti-parallel edges. To model anti-parallel edges, add a dummy node to the edge with incoming and outgoing flow to be the same. If vertex are allowed to have capacities, split the vertex into 2 and have a directed edge with that capacity.

Flow

Function that assigns a number $f(u, v)$ to each edge, flow from u to v.

Capacity. For every edge, the flow going through it cannot exceed the capacity and cannot be less than 0.

Flow conservation. For every vertex, the flow into it equals the flow out of it.

Max Flow. Total value of flow from start to every vertex.

Max Flow Problem. Given a network with capacities, find the maximum flow.

Matching. Subset of edges s.t. each vertex is part of at most 1 edge.

Bipartite Matching. Start by mapping node to every node on one side and end node maps to every node on other. Set all capacities to 1 and apply max flow algorithm. Result is the largest number of edges a matching can have. **Integer valued flow corresponds to some matching, and vice versa**.

Residual Capacities. Leftover flow.

Ford-Fulkerson

Find an path from start to end in the graph (all non-zero weights). Obtain the bottleneck weight of the path (lowest weight in the path). Reduce all capacities of the edges by bottleneck, add bottleneck to result. Efficiency depends on how path is chosen. If DFS, $O(|E| \cdot |f_{max}|)$, where f_{max} is maximal flow.

Proof of correctness. If f is a valid flow, it remains a valid flow after updating capacities. If there is no path left, then f is maximal flow.

Capacity on Vertex. Split node into in and out nodes. Then the edge connecting both contains the capacity of the vertex as the capacity of the edge.

Flows and Cuts

Split into vertices that can be reached by source, S and those that can't, T. Then the min-cut would be the sum of the capacities of edges going from S to T. Any cut is a bottleneck for any flow.

Linear Programming

All constraints are linear inequalities, objective is a linear function. Just need to solve optimisation problem. Optimal solution (if exists) can be found in poly(n,m) time. Normally not as fast the algorithm to solve the actual problem (using LP to solve max flow is slower than max flow algorithms).

Standard Form of LP. Objective is to **maximise**. **All variables** required to be **non-negative**. **All constraints** are less-than-or-equal-to. Tricks to convert to standard form:

- If objective is to minimise, flip the sign to maximise (multiply by -1)
- To convert $\geq$ to $\leq$, flip the sign
- To convert equality ($3x_1 + 5x_2 = 20$), result should be 2 inequalities, $3x_1 + 5x_2 \leq 20$ and $-3x_1 - 5x_2 \leq -20$.
- If a variable is not required to be non-negative, say x_2, replace x_2 with $(x_2' - x_2'')$. Then enforce that $x_2', x_2'' \geq 0$.
- To minimise $\sum_{i=1}^{n} |y_i - ax_i - b|$, introduce new variables $e_1 \dots e_n$ to get $min \ \sum_i e_i$ st $y_i - ax_i - b \leq e_i$, $y_i - ax_i - b \geq -e_i$ and $e_i \geq 0$.

Solving LP. Plot a graph with the variables as axis. Narrow the area down such that solutions within the area are possible solutions that satisfies the constraints. The resulting shape that satisfies all constraints is the **simplex**. The function that we want to maximise would be at the vertex (highest point) of the simplex. To determine this, draw a line and shift it until the vertex is reached, then take the result. Optimal feasible solution (if it exists) is always at vertex. If objective function keeps increasing, then we know that feasible solutions exists but no optimum, so LP is **unbounded**.If no points satisfies all constraints, then no feasible solutions exists. So LP is **infeasible**.

Simplex Algorithm. Start at feasible vertex and compute objective value. Then, find neighboring vertex with larger objective value. If none exist, then current is optimum. Otherwise, move to that vertex and repeat. If LP is unbounded, algorithm will notice unbounded vertex whose objective value is unbonded. Some pre-processing is needed to check if LP is feasible. If there are n variables, then simplex is n-dimensional convex polyhedron. Correctness follows from convexity - any local optimum is global optimum. Could take exponential time in worse case. Could construct a set of constraints that force algorithm to visit exponential number of points.

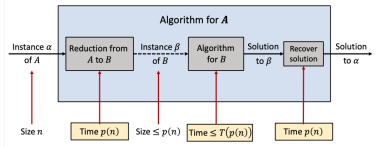
Reductions

Consider 2 problems A and B(blackbox). If we can use B to solve A, then **A reduces to B** or **reduction from A to B** or **A has reduction to B**. Convert A to instance of B, then solve B, and use solution to get solution for A.

Bounded Time Reductions

If possible to perform conversions in $p(n)$, where n is **size** of input, then **A has $p(n)$ time reduction to B**. For integers, $i, n = log(i)$. Matrixes, $n = N \times N$.

Running Time Composition. $p(n)$ reduction from A to B, $T(n)$ to solve B, then there is a $T(p(n)) + O(p(n))$ time algorithm to solve A. Size of B at most $p(n)$ as its not possible to create something bigger than $p(n)$ in $p(n)$ time.



If there is a $p(n)$ reduction from A to B, then $A \leq_P B$.

Pseudo-polynomial Algorithms. Algorithms that runs in polynomial in the **numerice value** of input but **exponential** in length of representation. If $A \leq_P B$, then if A cannot be solved in poly time, then neither can B. Composition is transitive. Example is knapsack which takes $O(nW)$.

Intractability

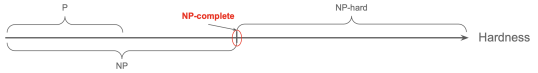
Decision Problems. Function that maps instance to either YES or NO.

Optimisation Problems. Solves for min/max of something we are interested in. Can use optimisation to solve decision problems, so $DEC \leq_P OPT$.

Reductions between Decision Problems. Possible such that a YES for A iff a YES for B and transformation takes polynomial time in size of A input.

Complexity Class

Refering to decision problems.



NP. Solutions that can be verified in polynomial time.

P. Set of problems that can be solved in polynomial time. Every problem in P is in NP (can just run algorithm in P to verify.)

co-NP. Counterexamples can be verified in polynomial time. Complement of any NP problem is in co-NP.

NP-hard. A is NP-hard if for any problem B in NP, $B \leq_P A$.

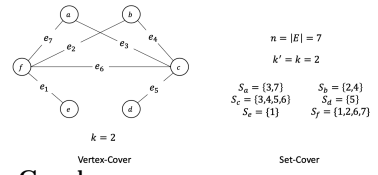
NP-Complete. If its in NP and also NP hard. To show, show that problem is in NP (give polynimial sized solution and polynomial time verification) and show that problem is in NP-Hard (Show that reduction from any NP complete problem to NP-hard problem.)

Reduction Tricks

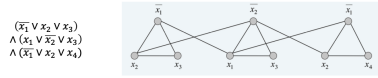
Independent Set. Vertices st no 2 are adjacent.

Vertex Cover. Vertices st each edge join to every other edge. Vertex cover is complement of Independent set.

Sets to Graph.



SAT to Graph.



LP. Fixed value constraints can be relaxed to a range instead, then perform rounding.

Directed to Undirected. Each node will node have an in, neutral and out nodes.

Indep Set to Clique. Complement graph.

Reducing from 3-SAT. Encode literals (each literal is either true or false), encode clause (if at least 1 literal is true, entire clause is true), encode CNF (put everything together).

Palindromes and LCS. Find longest palindromic subsequence. Reverse the string and pass the reversed and original to LCS to get the answer.

Matrix Mul and Square. From mul to square is easy. For square to mul, create a new matrix C with the inputs along the diagonal going from right to left and multiply together.

T-Sum & 0-Sum. 0-sum: sum of 2 elements is 0. T-sum: sum of 2 elements is T. $B[i] + B[j] - T = 0$, so create new array and each element in the array is of the form $B[i] - \frac{T}{2}$.

Approximating Optimisation Problems

Given an instance, find a solution that has min/max cost. If C^* is cost of optimal and C is cost of solution found by algorithm, then $\frac{C}{C^*}/\frac{C^*}{C}$ is the approximation ratio for min/max respectively. Ratio should be always larger than 1.

Approx Vertex Cover. Greedy algorithm that takes highest degree node everytime. Show that approximation is 2-Approx ($C \leq 2 \cdot C^*$). Let A be edge picked. For every node, vertex cover contain either vertice so size at least $|A|$. Size of the smallest would be at $|A| \leq C^*$. Since both vertices are added, so $C = 2 \cdot |A|$, which leads to $C \leq 2 \cdot C^*$.

Function Max Approx. Express suboptimal as optimal. Then take max of the ratios (sub/opt, opt/sub). For non-constant approx, express results as $1 + \epsilon + \_$ and show that ϵ present in the resulting term makes it non-constant.

Approximation Schemes

Poly-Time Approximation. An algorithm that given an instanc and an $\epsilon > 0$ runs in time $poly(n)f(\epsilon)$ for some function f and has approximation ratio $(1+\epsilon)$

Fully Poly-Time Approximation. An algorithm that given an instance and an $\epsilon > 0$, runs in time $poly(n, \frac{1}{\epsilon})$ and has approximation ratio $(1+\epsilon)$