MATH REVIEW

- **Logarithm rules**
  - Definition: $\log_b x = y \Leftrightarrow b^y = x$.
  - Product: $\log_b xy = \log_b x + \log_b y$.
  - Quotient: $\log_b \frac{x}{y} = \log_b x - \log_b y$.
  - Power: $\log_b x^y = y \log_b x$.
  - Base swap: $\log_b c = \frac{1}{\log_c b}$.
  - Base change: $\log_b x = \frac{\log_c x}{\log_c b}$.
  - Base what the fuck rule: $b^{\log_c a} = a^{\log_c b}$
- **Exponent rules**
  - Product: $a^n a^m = a^{n+m}$, $a^n b^n = (ab)^n$
  - Quotient: $\frac{a^n}{a^m} = a^{n-m}$, $\frac{a^n}{b^n} = \left(\frac{a}{b}\right)^n$.
  - Negative exponent: $b^{-n} = \frac{1}{b^n}$.
  - Power rule 1: $(b^n)^m = b^{nm}$
  - Power rule 2: $b^{n^m} = b^{(n^m)}$
  - Power rule 3: $\sqrt[m]{b^n} = b^{\frac{n}{m}}$
  - Power rule 4: $b^{\frac{1}{n}} = \sqrt[n]{b}$
- **Sequences and series**
  - Linearity: finite sequences can be rearranged: $\sum_{k=1}^{n}(ca_k + b_k) = c\sum_{k=1}^{n} a_k + \sum_{k=1}^{n} b^k$.
  - **Arithmetic series**: $\sum_{k=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} = \Theta(n^2)$.
  - Series of squares: $\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
  - Series of cubes: $\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}$
  - **Geometric series**: $1 + x^2 + x^3 + \cdots + x^n = \sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1}$ for finite n.

    For infinite n and |x| < 1: $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$. If |x| > 1, series diverges.

    For infinite n and |x| < 1: $\sum_{i=0}^{\infty} ix^{i-1} = 1/(1-x)^2$.
  - **Harmonic series**: $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k} = \ln n + O(1)$
  - **Telescoping series**: $\sum_{k=1}^{n}(a_k - a_{k-1}) = a_n - a_0$ due to repeated cancelling of terms.
- **Induction** proves that a predicate is true for a base case. Then, assuming that the predicate holds for n, the proof must show that it holds for n+1. Common pattern: split $\sum_{k=1}^{n+1} f(k)$ into $\sum_{k=1}^{n} f(k) + f(n+1)$, then substitute in inductive hypothesis for $\sum_{k=1}^{n} f(k)$, then show sum obeys the predicate.
- **Combinations**: # of ways to choose k elements from set of n is $n!/(k!\ (n-k)!)$.
- **Permutations**: # of ways to arrange k elements from a set of n is $n!\ /\ (n-k)!$.
- **Bernoulli Trials:** if prob. p success, expected trials until success = 1/p; until kth success = k/p.
- **Markov's Inequality**: $P(X \geq a) \leq E(X)/a$ where a is a positive constant.
- **Euler's Number**: $\lim_{n\to\infty}(1 + 1/n)^n = e$ and $\lim_{n\to\infty}(1 - 1/n)^n = \frac{1}{e}$.

ASYMPTOTIC NOTATION

- **Big-Oh (upper bound)**: T(n) ∈ O(f(n)) if ∃ constants c, n_0 > 0 s.t. T(n) ≤ cf(n) for all n > n_0.
- **Big-Omega (lower bound)**: T(n) ∈ Ω(f(n)) if ∃ constants c, n_0 > 0 s.t. T(n) ≥ cf(n) for all n > n_0.
- **Big-Theta**: T(n) ∈ Θ(f(n)) if T(n) ∈ O(f(n)) and T(n) ∈ Ω(f(n)), i.e. they differ by a constant factor.
- **Little-Oh**: T(n) ∈ o(f(n)) if ∀ constants c > 0, ∃ n_0 > 0 s.t. T(n) < cf(n) for all n > n_0.
- **Little-Omega**: T(n) ∈ ω(f(n)) if ∀ constants c > 0, ∃ n_0 > 0 s.t. T(n) > cf(n) for all n > n_0.
- Polynomial growth will always dominates logarithmic growth (to any power).
- **Upper bound** performance is **sufficient** for all inputs, even the very worst possible input.
  - Loose upper bounds can be determined by rounding up (i.e. highballing) recurrence terms.
  - Tighter upper bounds closer to the "true complexity" are lower.
- **Lower bound** performance is **necessary** for all inputs. Must apply to *all possible algorithms*.
  - Tighter lower bounds closer to the "true complexity" are higher than looser lower bound.
  - One technique is to consider the minimum value for half of the terms in a recurrence. Example: T(n)=cn+T(n-1) has n/2 terms each ≥ cn/2. Sum is (n/2)*(cn/2) = $cn^2/4$ → $\Theta(n^2)$.

RECURRENCES AND SUMMATIONS

- **Recursion trees** visualize the recurrence of the form T(n) = a * T(n/b) + cn^k.
  - Root node does cn^k work; a child nodes do $c(n/b)^k$; a^2 grandchildren do $c(n/b^2)^k$ work, etc.
  - a = "splitting factor", i.e. the number of children that each node splits to; often equals b.
  - b = the degree to which the problem size is reduced for the child; often equals a.
  - L = $\lfloor \log_b(n) \rfloor$ + 1, the number of levels in recursion tree, including the root.
  - M = a^L - 1, the number of nodes in recursion tree, including the root.
- **Unrolling** decomposes the recursive calls into a single line, from which it may be possible to determine a summation (or at least lower/upper bounds) by inspection.
- **Guess and inductive proof** requires making a guess and then proving the guess is correct by induction. If the guess was wrong, it may have clues to a better guess.
  - Example: T(n) = 2T(n/2) + $\log_2$(n); T(1) = 0. For each of the 2n-1 nodes, upper bound with O(logn) to get T(n) = O(nlogn); lower bound with 1 to get T(n) = Ω(n).
- **Master Formula** is based on recurrence form T(n) = a * T(n/b) + cn^k, i.e. algorithm does cn^k work up front, then divides the problem into a pieces of size n/b, solving each one recursively.
  - If a < b^k then $T(n) \in \Theta(n^k)$, i.e. problem is subdivided by b more than splitting factor a, so T(n) is dominated by the n^k work done at higher levels.
  - If a = b^k then $T(n) \in \Theta(n^k \log n)$, i.e. problem is subdivided by b at the same rate as the splitting factor a, so T(n) is contributed to equally by all logn levels.
  - If a > b^k then $T(n) \in \Theta(n^{\log_b a})$, i.e. problem is subdivided by b less than splitting factor a, so T(n) is dominated by the work done by many nodes at lower levels.
- Examples:
  - Quiz 1: T(n) = 3T(n-1) + 3^n for n > 1; T(n) = 1 for n ≤ 1. Summation: $\Theta(n3^n)$ by unrolling as a tree: each level has total cost 3^n and there are n levels, thus T(n) = n*3^n = $\Theta(n3^n)$.
  - Quiz 1: $T(n) = \left\lfloor \sqrt{T(n-1)} \right\rfloor = \Theta(1)$ since $\lfloor \sqrt{2} \rfloor$ is always 1.
  - HW1: T(n) = 2T(n/2) + 1 for n > 1; T(n) = 1 for n ≤ 1. Summation: Θ(n).
  - HW1: T(n) = 3T(n/2) + n^3 for n > 1; T(n) = 1 for n ≤ 1. Summation: $\Theta(n^3)$.
  - HW1: T(n) = $T(n^{1/2})$ + 1 for n > 2; T(n) = 1 for n ≤ 2. Summation: Θ(log(log(n))).

ORDER STATISTIC AND SORTING ALGORITHMS

- **Median finding** algorithms aim to find the ith **order statistic**, i.e. the ith smallest element.
  - **QuickSelect algorithm** has expected O(n) time. It selects a random pivot, separates array into two subarrays with elements < pivot and > pivot. Median value must be in the larger of the two subarrays, which is the argument to a recursive call. Worst-case running time is still $O(n^2)$ if pivot is always largest. Expected number of comparisons is at most 4n.
  - **Worst-case O(n) time** algorithm ensures that at least 3/10 of array is ≥ p and ≤ p. It does this by subdividing the array into subarrays of 5 elements, each of which is sorted so the median is found. The median of these medians must be ≥ and ≤ at least 3/10 of the array.
- **Maximum finding** has tight upper/lower bound of n-1 comparisons.
  - Upper bound proof is a simple scan, keeping track of largest element so far.
  - Lower bound proof models elements as nodes in graph with comparisons as edges: less than n-1 comparisons results in a graph with ≥ two components, invalid by contradiction.
  - Tournament algorithm finds 2nd largest element with tight upper/lower bound of n + lgn - 2 comparisons. Proof is by fact that 2nd largest element must have "lost" to largest.
- **Comparison-based sorting** has tight upper/lower bound of Θ(n logn).
  - Proof is by "information theoretic" argument: log(n!) bits of info about input are needed to find sorted sequence out of n! permutations; log(n!) < n logn and log(n!) > n/2*log(n/2).
  - If the number of permutations x ≠ n!, then the height of the binary tree is still log(x).
- **Exchange-based sorting** has tight Θ(n-1) upper/lower bound. Proof for upper bound is trivial.
  - Proof for lower bound models elements as nodes in directed graph. n self-loops denote correct positioning. Worst-case has one cycle, requiring n-1 exchanges to reach n self-loops.
- **Evasiveness of connectivity** relates to determining graph connectivity from adjacency matrix. Tight upper/lower bound of n(n-1)/2 requires querying every pair via adversarial argument.

AMORTIZED ANALYSIS

- **Amortized analysis** finds tighter upper bounds for a sequence of operations whose costs are not equal. The cost of the expensive operation is *amortized* over the inexpensive operations.
  - **Aggregate analysis** divides sum of individual operations by their count: T(n)/n.
  - **Accounting method** "overcharges" some operations, saving "prepaid credit" on the data structure elements for later operations that would otherwise be expensive.
  - **Potential method** also "overcharges" some operations, but with a single pool of credit.
- **Amortized cost**: a = c_i + Φ_{final} - $\Phi_{initial}$ = c_i + ΔΦ. c_i is actual cost and Φ is potential function.
  - **Potential function** defines the amount of "credit" banked based on data structure state.
  - $\Sigma a_i = \Sigma c_i + \Phi_{final} - \Phi_{initial}$, thus $\Sigma c_i = \Sigma a_i + \Phi_{initial} - \Phi_{final}$. Therefore, if $\Phi_i \geq 0$ for all i, then $\Phi_{initial} - \Phi_{final} \leq 0$, then sum of amortized costs is an upper bound of the sum of the true costs.
  - **Splitting problems into cases** can be useful when reasoning about potential functions.
  - When reasoning about potential functions, divide the cost of the expensive operation by the *number* of inexpensive operations over which the expensive operation has to be paid.
- **Binary counter** examples: Φ = number of 1 bits in the current counter state.
  - Lecture: Flipping any bit costs 1: cumulative cost = 2n, amortized cost = 2.
  - Recitation: Flipping ith bit costs 2^i: cumulative cost = Θ(nlogn) due to cost-frequency proportionality, amortized cost = Θ(logn).
  - Recitation: Flipping ith bit costs i+1: cumulative cost = 4n, amortized cost = 4.
  - Practice: nth operation costs largest power of 2 that divides n, amortized cost = Θ(logn).
- Growing a table examples.
  - Lecture: Φ = 0 if size s ≤ n/2, else 4(s - n/2). Cumulative cost ≤ 5m, amortized cost = 5. This is because m can be as small as N/2 + ϵ , 2N can be as large as 4m.
  - Quiz: `grow` triples table size and costs *old* table size. Just after grow to size S, there will be (2/3)S inserts until next grow, which will cost S. $((2/3)S + S)/((2/3)S) = 2.5$.
  - Quiz: `grow` triples table size and costs *new* table size. Just after grow to size S, there will be (2/3)S inserts until next grow, which will cost 3S. $((2/3)S + 3S)/((2/3)S) = 5.5$.

HASHING

- **Hash tables** support O(1) `Insert`, `Search`, and `Delete` in average case; Θ(n) in worst case.
- Load factor α for hash table T with M slots that stores N elements is N/M.
- **Universal hash function** h is drawn randomly from a universal hash function family H. Each h is universal iff for distinct x and y: $\Pr_{h\leftarrow H}[h(x) = h(y)] \leq 1/M$, i.e. chance of collision ≤ 1/M.
- **Expected number of collisions** between x and N elements already in table hashed by universal h ∈ H is N/M. By linearity of expectation, this implies O(1) time operations for N ≤ M.
- **Hash family tables**: M = number of possible values, N = number of keys/columns.
  - Determine max number of rows/functions for each pair of distinct columns/keys with a collision. Hash family is universal iff count ≤ 1/total number of rows.
- **Matrix Method** visualizes hash function family as m×u matrix A with random bits
  - M = 2^m since m bits are required to index M slots; |U| = 2^u for similar reasons.
  - Key is u×1 matrix of bits that "selects" columns to produce m×1 hash table index matrix.
- **ℓ-universal/independent hash functions** imply a uniform distribution over ℓ-length bit strings.
  - 1-universal implies probability of each key hashing to each value is *exactly* $1/M^1$. For binary outputs where M=2 (e.g. quiz), this implies a uniform distribution over 1-length bit strings 0, 1.
  - 2-universal implies probability of every pair of 2 distinct keys hashing to 2 particular values is *exactly* $1/M^2$. For binary outputs where M=2, this implies a uniform distribution over 4 bit strings 00, 01, 10, 11.
- **Perfect Hashing** picks h ∈ H that will produce zero collisions; keys must known and static.
  - Naïve approach: if M = N^2, then $\Pr_{h\leftarrow H}(\text{no collisions, i.e. perfect hash}) \geq 1/2$.
  - More space efficient approach: top array has size N; bins with c collisions point to overflow bins of size c^2, rehashed using method above with ≥ 1/2 probability of no collisions.

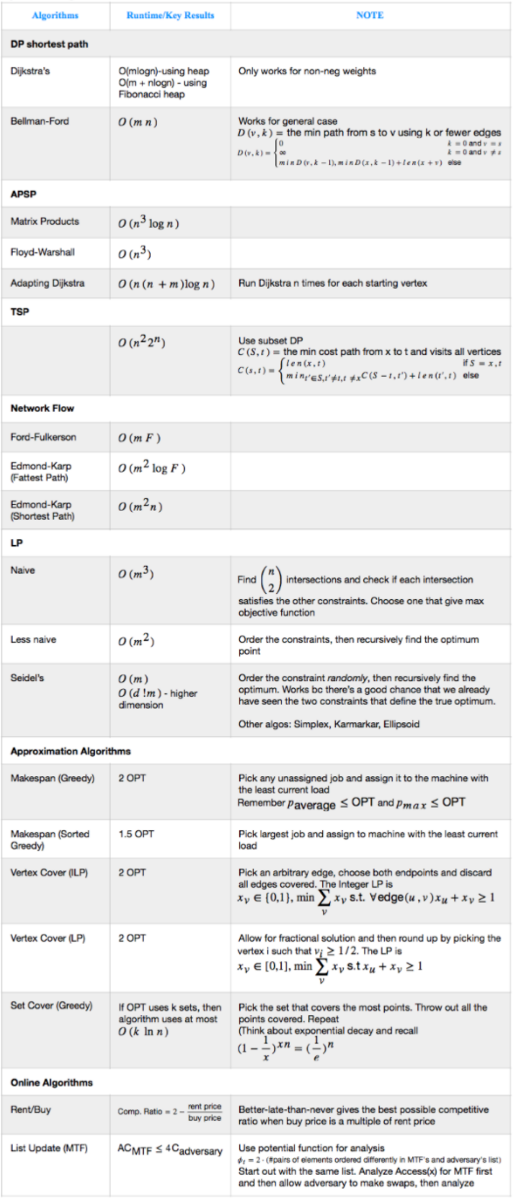
STREAMING AND STRING ALGORITHMS

- **ϵ-heavy hitters** want to know if an element appears ***strictly*** **greater than ϵ** of the time.
  - False negatives won't happen. False positives are OK; preventing them needs Ω(n) space.
  - Array T[k] stores elements from stream, C[k] stores integers where k = ⌈1/ϵ⌉ - 1.
  - If new element a in T[i], then increment C[i]. Else if a not in T and C[i] = 0, then set T[i] to a and C[i] to 1. Else decrement all entries in C.
  - Proof based on difference between estimate in C and true value, which is from decrements. Upper bound on number of decrements is t / (k + 1) ≤ ϵt where t is number of elements.
  - **Heavy hitters with deletions** require hashing. Number of errors (from collisions) is ≤ $|S_t|$/k where S_t is the active set and k is the number of bins. Space usage is k + O(logk * logΣ).
  - This can be improved by using m independent hash functions and m counters. The probability that all counters have large (i.e. 2x) error is $(1/2)^m$.
- **Missing numbers problem** has stream of n-1 elements from 1 through n missing one element.
  - Finding missing element is done by storing sum and subtracting it from n(n+1)/2.
  - Finding two missing elements a and b is done by storing sum and sum of squares. By knowing a+b and $a^2 + b^2$, you can solve for them.
- **Prime numbers**. π(n) = number of primes in a sequence of ℕ from 1 to n.
  - π(n) > n/logn; for n ≥ 60k, π(n) > n/ln(n). Probability that i is prime for 1 ≤ i ≤ n is ≥ 1 / logn.
  - To have at least k ≥ 4 primes between 1 and n, it suffices to have n ≥ 2k*logk.
  - Any natural number D ≤ 2^N has at most N prime divisors. If we seek sN ≥ 2sN*log(sN) primes, then probability of a collision is 1/s.
- **String equality problem**: Alice and Bob want to check that they have the same message x.
  - Instead of sending x = N bits, send prime number p and x mod p = O(logN) bits.
  - Approach 1 has linear decrease in error probability: pick a prime from a larger range.
  - Approach 2 has exponential decrease in error probability: repeat the process independently.
- **Karp-Rabin algorithm** relies a rolling hash function to speed up hashing substrings of text.
  - Rolling hash function can be recomputed rapidly as the window slides: subtract the value of the (old) high-order digit, then add the value of the (new) low-order digit, mod p.
  - Two distinct values can collide as a false positive, but there will never be a false negative.
  - Initial hashing is O(n) time, each of m rolling hashes is O(1) time; total = O(m + n).
  - Probability of one false positive is m*(1/s) = m/s. Setting s = 100m ensures 1/100 probability of one false positive. Thus M = (200mn)*log(200mn), prime p is O(logm + logn) bits.
- Generalized Karp-Rabin "String Matching Oracle" can check for substring equality in O(1) time.
  - h(s) = ($s_0 b^{n-1}$ + $s_1 b^{n-2}$ + … + $s_{n-1} b^0$) mod p, where s = string, n = |s|, b is base > |Σ|.
  - Precomputed arrays: r[i] = b^i; a[i] = ($t_0 b^{i-1}$ + $t_1 b^{i-2}$ + … + $t_{n-1} b^0$) mod p, where t = pattern, i = |t|.
  - h($t_{i,j}$) = a[j + 1] - a[i]*r[j - i + 1].

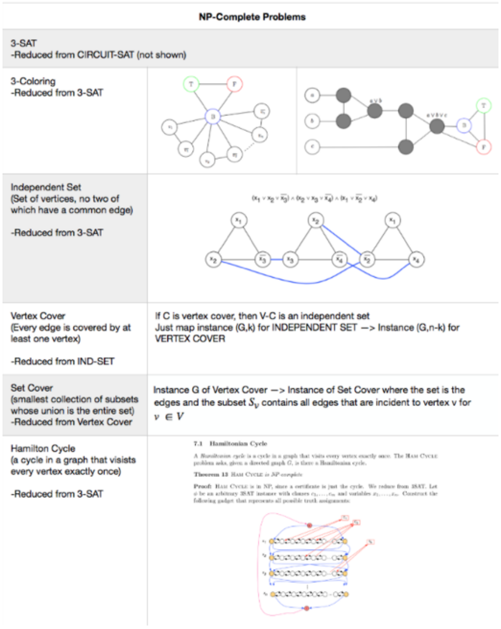
DYNAMIC PROGRAMMING

- Dynamic programming solves problems by breaking them into a reasonable (polynomial) number of subproblems that overlap, i.e. they are called in recursion repeatedly. Consider:
  - 1) What is the return value? This will usually be the return value of the recursive function.
  - 2) What are the choices? Example: include/exclude marginal item, consider n items, etc. The choices are typically aggregated using a min/max (for optimality) or perhaps sum.
  - 3) What are the subproblems? This follows from choices and return value.
  - 4) What are the parameters of each subproblem? This defines memo array and traversal.
  - 5) What are the base cases? Consider corner cases for parameters.
- **Memoization** stores subproblem solutions in an array/hash table for O(1) lookup on repeats.
- **Rod-cutting** seeks max. revenue from cutting a rod into integral lengths, each with a price.
  - i is length; p_i is the price of a length i rod segment, r_n is maximum revenue of length n rod.
  - Every possible cut position i is considered: $r_n = \max_{1\leq i\leq n}(p_i + r_{n-i})$.
- **Longest common subsequence (LCS) problem** seeks longest sequence of (non-contiguous) characters appearing in order in strings S and T where i is index of S and j is index of T.
  - If i = 0 or j = 0, then no substring is possible: LCS[i,j] = 0.
  - Else if S[i] ≠ T[j], then need to ignore either S[i] or T[j]: LCS[i,j] = max(LCS[i-1, j], LCS[i, j-1])
  - Else S[i] = T[j], then can't hurt to include S[i] and T[j]: LCS[i,j] = 1 + LCS[i-1, j-1].
  - Memoization grid has dimensions |S|*|T|. Overall runtime is O(|S|*|T|).
  - To reconstruct string, walk from lower-right of grid. If above or left cell have same value, move there (i.e. ignoring S[i] or T[j]). Else move to above-left cell (i.e. including S[i], T[j]).
- **Knapsack problem** seeks max combined value choice of n items with total weight ≤ w.
  - w_i is weight of item i; v_i is value of item i; K(w) = max value achievable with capacity w.
  - If repetition of items is allowed: $K(w) = \max_{i:w_i\leq w}\{k(w - w_i) + v_i\}$.
  - If repetition of items is not allowed, then need parameter 0 ≤ j ≤ n. Max value achievable with capacity w and items 1 … j $= K(w,j) = \max\{K(w - w_j, j-1) + v_j, K(w, j-1)\}$.
  - Memoization grid has dimensions n*w. Overall runtime is O(nw).
  - To reconstruct items, walk from lower-right of grid. If arr[k][B] = arr[k-1][B] then kth item was not used, go up to arr[k-1][B]. Else kth item was used, so output it and go to arr[k-1][B-s_k].
- **Making change** seeks minimum number of differing denominations to make C change.
  - v_i is value of ith denomination; l_i is number of bills available; j is number of bills used.
  - Primary recurrence: $B[C', i] = \min\{B[C - jv_i, i-1] + j : 0 \leq j \leq l_i, C' - jv_i \geq 0\}$.
- Independent set is a subset of vertices S ⊆ V where no edge has both ends in S. If each vertex v has a weight w_v, **Max-Weight Independent Set (MWIS)** seeks independent set with max weight. This is a hard problem, but O(n) time on trees. C(v) is of children of vertex v.
  - Weight of MWIS in v's subtree including v $= U(v) = w(v) + \sum_{u\in C(v)} N(u)$.
  - Weight of MWIS in v's subtree excluding v $= N(v) = \sum_{u\in C(v)}[\max\{N(u), U(u)\}]$.
  - Memoization grid has dimensions |V|*|V|???
- **Optimal binary search tree** orders n keys by frequency f_x so most frequent keys are near top.
  - Dynamic programming approach considers keys in range i through j and root k in that range.
  - $C_{i,j}$ is the cost of optimum tree using keys i through j. Base cases: $C_{i,j}$ = 0 if i > j; $C_{i,j}$ = f_i if i ==j.
  - Recursive case: $C_{i,j} = \min_{i\leq k\leq j}(f_{i,j} + C_{j,k-1} + C_{k+1,j})$. This considers all possible roots k.
  - Overall runtime is $O(n^3)$ in basic implementation, but can be improved to $O(n^2)$.

Bipartite Matching < Network Flow < Min-Cost Max-Flow < LP



Algorithms	Runtime/Key Results	NOTE
DP shortest path		
Dijkstra's	O(mlogn)-using heap O(m + nlogn) - using Fibonacci heap	Only works for non-neg weights
Bellman-Ford	$O(m\ n)$	Works for general case $D(v,k)$ = the min path from s to v using k or fewer edges $D(v,k) = \begin{cases} 0 & k = 0 \text{ and } v = s \\ \infty & k = 0 \text{ and } v \neq s \\ min\ D(v,k-1), min\ D(x,k-1) + len(x+v) & \text{else} \end{cases}$
APSP		
Matrix Products	$O(n^3 \log n)$	
Floyd-Warshall	$O(n^3)$	
Adapting Dijkstra	$O(n(n+m)\log n)$	Run Dijkstra n times for each starting vertex
TSP		
	$O(n^2 2^n)$	Use subset DP $C(S,t)$ = the min cost path from x to t and visits all vertices $C(s,t) = \begin{cases} len(x,t) & \text{if } S = x,t \\ min_{t'\in S, t'\neq t, t\neq x} C(S-t,t') + len(t',t) & \text{else} \end{cases}$
Network Flow		
Ford-Fulkerson	$O(m\ F)$	
Edmond-Karp (Fattest Path)	$O(m^2 \log F)$	
Edmond-Karp (Shortest Path)	$O(m^2 n)$	
LP		
Naive	$O(m^3)$	Find $\binom{n}{2}$ intersections and check if each intersection satisfies the other constraints. Choose one that give max objective function
Less naive	$O(m^2)$	Order the constraints, then recursively find the optimum point
Seidel's	$O(m)$ $O(d\,!m)$ - higher dimension	Order the constraint *randomly*, then recursively find the optimum. Works bc there's a good chance that we already have seen the two constraints that define the true optimum. Other algos: Simplex, Karmarkar, Ellipsoid
Approximation Algorithms		
Makespan (Greedy)	2 OPT	Pick any unassigned job and assign it to the machine with the least current load Remember p_{average} ≤ OPT and p_{max} ≤ OPT
Makespan (Sorted Greedy)	1.5 OPT	Pick largest job and assign to machine with the least current load
Vertex Cover (ILP)	2 OPT	Pick an arbitrary edge, choose both endpoints and discard all edges covered. The Integer LP is $x_v \in \{0,1\}$, $\min \sum_v x_v$ s.t. ∀edge(u,v) $x_u + x_v \geq 1$
Vertex Cover (LP)	2 OPT	Allow for fractional solution and then round up by picking the vertex i such that $v_i \geq 1/2$. The LP is $x_v \in [0,1]$, $\min \sum_v x_v$ s.t $x_u + x_v \geq 1$
Set Cover (Greedy)	If OPT uses k sets, then algorithm uses at most $O(k\ \ln n)$	Pick the set that covers the most points. Throw out all the points covered. Repeat (Think about exponential decay and recall $(1 - \frac{1}{x})^{xn} = (\frac{1}{e})^n$
Online Algorithms		
Rent/Buy	Comp. Ratio = $2 - \frac{\text{rent price}}{\text{buy price}}$	Better-late-than-never gives the best possible competitive ratio when buy price is a multiple of rent price
List Update (MTF)	$AC_{MTF} \leq 4C_{adversary}$	Use potential function for analysis ϕ_t = 2 · (#pairs of elements ordered differently in MTF's and adversary's list) Start out with the same list. Analyze Access(x) for MTF first and then allow adversary to make swaps, then analyze



NP-Complete Problems	
3-SAT -Reduced from CIRCUIT-SAT (not shown)	
3-Coloring -Reduced from 3-SAT	
Independent Set (Set of vertices, no two of which have a common edge) -Reduced from 3-SAT	$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$
Vertex Cover (Every edge is covered by at least one vertex) -Reduced from IND-SET	If C is vertex cover, then V-C is an independent set Just map instance (G,k) for INDEPENDENT SET —> Instance (G,n-k) for VERTEX COVER
Set Cover (smallest collection of subsets whose union is the entire set) -Reduced from Vertex Cover	Instance G of Vertex Cover —> Instance of Set Cover where the set is the edges and the subset S_v contains all edges that are incident to vertex v for $v \in V$
Hamilton Cycle (a cycle in a graph that visits every vertex exactly once) -Reduced from 3-SAT	7.1 Hamiltonian Cycle A *Hamiltonian cycle* is a cycle in a graph that visits every vertex exactly once. The HAM CYCLE problem asks, given a directed graph G, is there a Hamiltonian cycle. **Theorem 13** HAM CYCLE *is NP-complete* **Proof:** HAM CYCLE is in NP, since a certificate is just the cycle. We reduce from 3SAT. Let φ be an arbitrary 3SAT instance with clauses $c_1, \ldots, c_m$ and variables $x_1, \ldots, x_n$. Construct the following gadget that represents all possible truth assignments:

GRAPH ALGORITHMS

- **s-t shortest path** seeks shortest path from node s to node t.
- **Single-Source Shortest Path (SSSP)** seeks shortest path from node s to every node in G. Equivalent **Single-Sink Shortest Path** is the same but from every node to node t.
- **All-Pairs Shortest Path (APSP)** seeks shortest path between every pair of nodes in G.
- **Dijkstra's algorithm** solves SSSP in O(mlogn) time, or O(m + nlogn) with Fibonacci heaps.
  - Dijkstra's does not accommodate negative-weight edges (use Bellman-ford instead).
  - APSP can be solved by repeated runs of a modified Dijkstra's. Runtime is O(n(n+m)logn). Beats $O(n^3)$ if m ≤ $O(n^2/\log n)$. Need to run Bellman-Ford to compute potential for each node.
- **Bellman-Ford algorithm** solves SSSP using dynamic programming. D(v,k) is the minimum path length from vertex s to vertex v using k or fewer edges. Negative-weight edges are fine.
  - If k = 0 and v = s, then this is the start node: D(v,k) = 0.
  - Else if k = 0 and v ≠ s, then no zero-length path exists: D(v,k) = ∞.
  - Else path goes from s to x, a neighbor of v, via k-1 edges, then one last edge on its way to v. Length is : $D(v,k) = \min\{D(v,k-1), \min_{(x,v)\in E}[D(x,k-1) + len(x,v)]\}$.
  - Overall runtime for n vertices and m edges is O(mn) for SSSP, $O(mn^2)$ for APSP.
  - To reconstruct path, move from v at distance d[v] to neighbor x where d[x] + len(x,v) = d[v].
- **"Matrix Product" algorithm** performs $O(n^3 \log n)$ APSP by increasing max number of edges.
  - Square matrix A has vertices along each axis. Diagonal A[i,i] = 0, otherwise elements are ∞.
  - Iterate through edges: A[i,j] = len(i,j) for edge e from i to j. This is length via 1 or fewer edges.
  - Shortest path length via 2 or fewer edges B[i,j] = min(A[i,k] + A[k,j]). Calculated by performing "matrix multiplication" except sum instead of multiply and minimize instead of sum.
- **Floyd-Warshall** algorithm performs $O(n^3)$ APSP by increasing set of allowable vertices.
  - A[i,j] = min(A[i,j], A[i,k] + A[k,j]) ∀ neighbors k of j. We either go through k or we don't.
- **Traveling Salesman Problem** seeks optimal tour visiting each node once, returning to start.
  - Each subproblem is distance from x to t', a neighbor of t. Number of subproblems is number of sets of vertices times number of ending vertices: $O(2^n)$*O(n) = $O(n2^n)$. Overall $O(n^2 2^n)$.
  - C(S,t) is minimum cost path from x to t hitting all vertices in S along the way.
  - If S = {x,t}, then this is the base subproblem: C(S,t) = len(x,t).
  - Else, consider all choices of t': $C(S,t) = \min_{t'\in S, t'\neq t, t'\neq x}[C(S-t,t') + len(t',t)]$.

GAME THEORY

- **Game** consists of **players** (participants), each with a set of **actions** (choices on how to behave). The combined behavior of players leads to a **payoff** for each player.
- **Payoff matrix** has tuple of payoffs to (row, column) players for given choice of actions by each.
- **Zero-sum games** have payoffs sum to 0 within each tuple. Thus payoff matrix can be collapsed to **row-payoff matrix**, only showing payoffs to the row player; **column-payoff matrix** is inverse.
- **Pure strategy** is a single action. **Mixed strategy** is probability distribution over actions. Vectors p and q are mixed strategies for row and column players respectively. Values always sum to 1.
- Row player picks p* maximizing payoff over all q: $lb = \max_p \min_q V_r(p,q)$.
- Column player picks q* minimizing opponents payoff over all p: $ub = \min_q \max_p V_r(p,q)$.
- When solving for p* and q*, can assume a pure strategy for opponent, and then set terms of min() or max() equal to each other.
- **Von Neumann's Minimax Theorem**: $lb = ub$ for finite, 2-player, zero-sum games.
- **Randomized algorithm lower bounds** use payoff matrix R to formalize adversarial argument.
  - Columns of R are various algorithms for the problem (sorting in this case).
  - Rows of R are all n! possible inputs to the (sorting) algorithms.
  - Entry R_{ij} is cost of algorithm j on input i (the number of comparisons).
  - Deterministic, good worst-case runtime is a column with all small entries.
  - Randomized, good expected runtime is a probability distribution q over columns (i.e. a mixed strategy) such that expected cost for each row i is small. This is an upper-bound.
  - Best randomized algorithm is minimax-optimal q*.
  - Lower bound for randomized algorithms is a distribution p over rows (i.e. a mixed strategy) such that for every column (algorithm j) expected cost of j is high.
- **General-Sum Two-Player Games** are not purely competitive: there are win-win and lose-lose.
  - **Nash Equilibrium** is a stable set of (mixed) strategies for the players, where neither has an incentive to unilaterally switch to a different strategy.
  - Every finite player game with a finite number of strategies has at least one Nash Equilibrium.

NETWORK FLOWS

- An **s-t cut** partitions vertices into sets A and B where s ∈ A, t ∈ B. **Capacity** of the cut is the sum of the capacities of the cut edges that go from A to B; an upper bound on flow from A to B.
- **Residual graph** describes remaining capacity after each **augmentation** (i.e. iteration).
- **Skew symmetry** means flow can be "undone" in residual graph: f(u,v) = -f(v,u).
- **Maxflow-Mincut Theorem**: maximum s-t flow equals the capacity of the minimum s-t cut.
- **Integral-Flow Theorem**: If all capacities are integral, the maximum flow is integral.
- **Ford-Fulkerson (FF)** repeatedly pushes as much flow through a path with capacity > 0.
  - Nodes reachable (e.g. via DFS) from s after FF are in set A; unreachable nodes are in set B.
  - Correctness proof uses fact that augmenting path with k flow traverses A→B once more than B→A, thus cut's residual capacity decreases by k, which is the same as the increase in flow.
  - Runtime of Ford-Fulkerson is O(F(m+n)) where F is maximum s-t flow. This is exponential!
- **Edmonds-Karp #1 (EK1)** is FF except it picks the largest capacity "maximum bottleneck" path.
  - Maximum bottleneck path is computed using modified Dijkstra's in O(mlogn) with minimum of weights in a path (rather than sum); this repeats on residual graph after each augmentation.
  - Graph with maximum s-t flow F must have a path with capacity ≥ F/m.
  - Runtime: O(mlogF) iterations * O(mlogn) = $O(m^2 \log n \log F)$; can be lowered to $O(m^2 \log F)$.
- **Edmonds-Karp #2 (EK2)** is FF except it picks the shortest path in the residual graph.
  - In other words, convoluted paths are avoided unless they are needed.
  - Runtime is O(mn) iterations times O(m+n) BFS = $O(nm^2 + mn^2) = O(nm^2)$ if m ≫ n.
- **Dinic's Algorithm** is EK2 except it performs batch augmentation of all paths of min. length.
  - **Blocking flow** is batch flow on all paths of minimum length, as determined from BFS tree. Every path has a saturated edge on a blocking flow.
  - After augmenting with blocking flow, all paths with capacity in residual graph are longer.
  - Runtime is O(n) iterations times O(nm) per iteration to find blocking flow = $O(n^2m)$.
- **Bipartite matching** seeks assignment (edges) from left set of nodes L to right set of nodes R.
  - A **matching** is a set of edges with no common endpoints; **maximum matching** is has the maximum number of edges; **perfect matching** connects all nodes in L and R.
  - This reduces to a flow problem: create dummy node s connecting to nodes in L and t that nodes in R connect to. Running FF (since edge weight is 1) finds maximum matching since it can undo "bad" paths via skew symmetry.
  - When run with Dinic's, first blocking flow is at least ⌈ m/2 ⌉, runtime is O(n+m).
- **Min-cost max flow** can model **min-cost matching** (i.e. matching with preferences/weights).
  - Among all the ways to achieve precisely the max flow, we want the one with lowest cost.
  - Edges now have weights in addition to capacities. Skew symmetry applies to weights too.
  - One approach is modified Ford-Fulkerson that always chooses least cost path from s to t. This uses Bellman-Ford instead of Dijkstra for finding path. Runtime is similar to FF.
- Network flow **does not enforce mutual exclusion**. To enforce this, it may be needed to **duplicate nodes** (e.g. CATS problem, agents in practice exam).

LINEAR PROGRAMMING

- Linear programming (LP) is general problem solving tool: n **variables**, m **linear (in)equalities**, and **objective function**. Solve for variables that maximize/minimize objective and satisfy constraints.
  - Constraints and objective function must be linear, i.e. no multiplication of variables.
  - **Infeasible** LPs have no points satisfying the constraint.
  - **Feasible and bounded** LPs have a feasible point of maximum objective value.
  - **Feasible and unbounded** LPs have feasible point of arbitrarily large objective function value.
  - **Integral solutions** can't be forced using LP; integer linear programming is much harder.
- **Standard form** for LPs maximizes c^Tx subject to Ax ≤ b and x ≥ 0.
  - **x** is column vector of the d variables that the LP is solving for.
  - **c** is column vector with coefficients of variables (from x) in objective function.
  - **A** is the matrix where each row is coefficients of variables (from x) of one constraint.
  - **b** is column vector of constants describing bounds (i.e. right hand side) of each constraint.
  - Equality constraints can be replaced with ≥ and ≤.
  - Negation and reordering translates inequalities into ≤ relationship.
  - Variable x_i that can be negative is substituted with two variables: $x_i = x'_i - x''_i$.
  - Important matrix identity: $(AB)^T = B^TA^T$
- Modeling **max flow as LP** has one variable f_{uv} for each edge (u,v) ∈ E.
  - Objective function is to maximize flow into t (minus any out of t): $\sum_u f_{ut} - \sum_u f_{tu}$.
  - Constraint for capacity: for all edges: 0 ≤ f_{uv} ≤ c(u,v) where c() is capacity.
  - Constraint for flow conservation: $\forall v \notin \{s,t\}, \sum_u f_{uv} = \sum_u f_{vu}$.
  - **Min-cost max flow as LP** first solves maximum flow F ignoring costs. Then, add constraint that flow must equal F and change objective function to minimize $\sum_{u,v\in E} w(u,v) f_{uv}$.
- Modeling **2-player zero-sum games as LP** has one variable p_i for the probability of each action in a minimax optimal mixed strategy. Payoff matrix m_{ij} for row/column player actions i/j.
  - Constraint for probability distribution: p_i ≥ 0 and $\sum_i p_i = 1$.
  - Constraint for opponent minimization: extra variable v denotes the minimization done by the opponent. The expected payoff for each column is at least v: $\sum_i p_i m_{ij} \geq v$.
  - Objective function is to maximize v.
- Modeling **matching problems as LP** has one variable e_{uv} for each edge (u,v) ∈ E.
  - Constraint for binary presence: 0 ≤ e_{uv} ≤ 1, either edge is included (1) or its not (0).
  - Constraint for edges touching distinct nodes: $\sum_{v:(u,v)\in E} e_{uv} \leq 1$ for all vertices u ∈ V.
  - Objective function is: $\max \sum_{(u,v)\in E} e_{uv}$.
  - Maximum objective function Z_G ≤ maximum matching M_G. If G is bipartite (i.e. graph with no odd cycles) then $Z_G = M_G$, but this does not hold generally (leads to non-integral solutions).
- Modeling **s-t shortest path as LP** has either a) a variable for every vertex and constraints for every edge, or b) a variable for every edge and constraints for every vertex.
- **Convex polytope** is feasible region: an intersection of half-spaces (one per constraint).
  - Point q is a vertex of polytope P if q ∈ P and vector v ∈ $\mathbb{R}^d$ ≠ 0, either q+v ∉ P or q-v ∉ P.
  - If a finite optimal point exists, a finite optimal point exists at a vertex of polytope P.
  - The intersection of convex sets is always convex. Thus you can't get stuck at a local maximum, have to go "backwards" to less optimal solution, and then "forwards" to true maximum.
- **Simplex Algorithm** for solving LPs uses fact that optimum solution is always at a vertex.
  - Starting at a polytope vertex, go to neighboring vertex with greatest objective function value.
  - There can be an exponential number of corners, so runtime can be exponential.
- **Ellipsoid Algorithm** only solves feasibility problem, but in a polynomial number of steps.
- **Interior Point Algorithm** have aspects of Simplex and Ellipsoid.
- **Seidel's Algorithm** iteratively adds constraints in a random order, checking old optimum x*.
  - Case 1: If x* satisfies new constraint C_m, then x* is still optimal. Test performed in O(d) time.
  - Case 2: If x* does not satisfy C_m, new optimum will be on C_m. To find new optimum, iterate through constraints, finding intersection point of each with C_{i+1}. Optimum lies at one of the intersections where constraints transition from "right facing" to "left facing".
  - Probability that optimum is defined by a given constraint is 2/m for m constraints.
  - Runtime: O(d!m) expected, $O(d!m^2)$ worst case. Great for large m but low d.

	Good in practice?	Good in theory?	Explainable?	Polynomial time?
Simplex	Yes		Yes	
Ellipsoid		Yes	Yes	Yes (plus binary search)
Interior point	Yes	Yes		Yes
Seidel's	Yes (low dimensions)	Yes	Yes	Yes in m (exponential in D)

- **Dual** of a **primal** LP problem is the linear combination of constraints.
  - Primal *maximizes* c^Tx subject to Ax ≤ b, x ≥ 0. Dual *minimizes* y^Tb subject to $y^TA \geq c^T$, y ≥ 0.

Primal
max($2x_1 + 3x_2$)
y_1: $4x_1 + 8x_2 \leq 12$
y_2: $2x_1 + 1x_2 \leq 3$
y_3: $3x_1 + 2x_2 \leq 4$
$x_1, x_2 \geq 0$

Dual
min($12y_1 + 3y_2 + 4y_3$)
x_1: $4y_1 + 2y_2 + 3y_3 \geq 2$
x_2: $8y_1 + 1y_2 + 2y_3 \geq 3$
$y_1, y_2, y_3 \geq 0$

  - The dual of the dual is the primal. Dual and primal are the best upper/lower bounds.
  - **Weak Duality**: If primal and dual are both feasible, primal optimal ≤ dual optimal = $c^Tx \leq y^Tb$.
  - **Strong Duality**: feasible, *bounded* primal implies a feasible, *bounded* dual. The maximum of the primal equals the minimum of the dual.

		Dual		
		Infeasible	Finite	Unbounded
Primal	Infeasible	Possible	Impossible	Possible
	Finite	Impossible	Primal = Dual	Impossible
	Unbounded	Possible	Impossible	Impossible

  - Min vertex cover primal: $\min \sum_{v\in V} x_v$ s.t. $x_u + x_v \geq 1\ \forall \{u,v\} \in E$ and $x_v \geq 0\ \forall v \in V$. This has one variable per vertex and one constraint per edge, enforcing rules of vertex cover. Dual is max matching: $\max \sum_{e\in E} y_e$ s.t. $\sum_{e\in N(v)} y_e \leq 1\ \forall v \in V$ and $y_e \geq 0\ \forall e \in E$. This has one variable per edge and one constraint per vertex, enforcing rules of matching.

NP-COMPLETENESS AND REDUCTIONS

- **Polynomial (poly) time algorithms** run in $O(n^c)$ for some constant c and input n.
  - Problem A is **poly-time reducible** to problem B, i.e. $A \leq_p B$, if A can be solved in poly time given a poly time black box algorithm for B. Poly-time equivalent $A =_p B$ if: $A \leq_p B$ and $B \leq_p A$.
  - $A \leq_p B$ can be thought of as "A is no harder than B, up to polynomial factors."
- **Karp Reduction** from problem A to problem B is a function f computed in poly time: if x is YES-instance of A, f(x) is a YES-instance of B; if x is a NO-instance of A, f(x) is a no instance of B.
- **P** is the set of **decision problems** (i.e. YES or NO answers) solvable in poly time. **Search problem** (i.e. specific answer) is solved with binary search of decision problem.
- **NP** is the set of decision problems with poly time verifiers V(I,X) where I is a YES- or NO-instance, X is the witness/certificate, and X is polynomial in the size of I.
- **NP-complete** includes problem Q if Q is in NP and for any other problem Q' in NP, $Q' \leq_p Q$.
  - Must show that I is YES-instance of Q' ⇔ f(I) is a YES-instance of Q and f is poly time.
  - If the first condition does not hold, then it is **NP-hard**.
- **CIRCUIT-SAT**: given a circuit of NAND gates with a single output and no loops, is there a setting of the inputs that causes the circuit to output 1?
- **3-SAT**: given a conjunctive normal form (CNF) formula, i.e. an AND of OR clauses, over n variables where clauses have ≤ 3 variables, is there a satisfying variable assignment?
- **Independent set**: given a graph G, is there an independent set of ≥ k vertices? Reduction from 3SAT: triangular clauses with terms as nodes, edges between negations of same terms.
- **CLIQUE**: given a graph G, does G contain a clique (complete subgraph) of size ≥ k?
- **Vertex cover**: given a graph G, is there a set of ≤ k vertices that cover i.e. touch every edge? Reduction from Independent Set: IS of size ≥ k ⇔ VC of size ≤ n-k.
- **Set cover**: given items U, set S of subsets of U, is there a subset of ≤ k sets ∈ S covering U? Reduction from Vertex Cover: if sets are like edges that cover 2 vertices, VC ≤ k ⇔ SC ≤ k.
- **Hamiltonian cycle**: is it possible to visit every vertex in a graph exactly once with no repeats? **Hamiltonian path** is related but not a cycle; reduced by duplication any node in the cycle.
- **Subset sum**: given a set of integers, is there a subset that sums exactly to T?

APPROXIMATION ALGORITHMS

- **Vertex cover** is NP-complete, but there are many algorithms that are 2-approximations:
  - Pick vertices of one edge, discard covered edges, repeat. Proof is by matching.
  - LP relaxation: variable per vertex 0 ≤ x_i ≤ 1, minimize $\sum_i x_i$, round x ≥ 1/2 to 1. Proof is by inequality: raw LP ≤ optimal integer LP ≤ rounded LP ≤ raw LP*2.
  - DFS (from HW7): perform DFS, then discard leaf nodes.
  - Runtime for trivial algorithm is n^k for n vertices, faster algorithm runs in $O(2^{k(2 \lg k + 1)})$ time.
- **Set cover** is NP-complete, but can get ln(n)-approximation by greedily picking remaining set that covers the most remaining points. (Even tighter bound is actually k*ln(n/k) + k.)
  - OPT must have a set covering n*(1/k) points; next set must cover n*(1-1/k); the tth set must cover $n*(1-1/k)^t$ points. After t = k*ln(n), there are $n*(1-1/k)^{k*\ln(n)} < n(1/e)^{\ln(n)} = 1$ points left.
- **Metric TSP** means distances are symmetric and obey the triangle inequality; it is NP-complete.
  - MST algorithm computes MST, then visits cities in preorder traversal, shortcutting where possible. This is 2-approximation: MST ≤ OPT - 1 edge ≤ OPT ≤ ALG ≤ 2*MST ≤ 2*OPT.
  - Christofide's algorithm computes MST T, then minimum-weight perfect matching M between odd-degree vertices in T, then constructs spanning Eulerian multigraph G = T ∪ M. Shortcutting G gives C: cost(C) ≤ cost(G) = cost(T) + cost(M) ≤ cost(OPT) + 1/2*cost(OPT) = 3/2*cost(OPT).
- **Makespan job scheduling** wants to minimize cumulative load on most-loaded machine i.
  - Greedy algorithm assigns arbitrary job to machine with least load; 2-approximation.
  - Sorted greedy algorithm assigns largest job to machine with least load; 4/3-approximation.

ONLINE ALGORITHMS

- **Competitive ratio** of online algorithm ALG on worst-case sequence σ is ALG(σ)/OPT(σ).
- **Rent or buy problem** has rental cost r and purchase cost p.
  - **Better-late-than-never (BLTN) algorithm** has competitive ratio < 2. If went skiing < ⌈p/r⌉ times, BLTN = OPT. Else BLTN paid r(⌈p/r⌉ -1) + p < 2p; 2p - r if p is integer multiple of r so ratio is 2 - r/p. Worst case is when you stop skiing just after you bought skis.
  - When r ≪ p this is very close to being 2-competitive; with randomization, e/(e-1)-competitive.
- **List update problem** has list of n items; access cost is item's index and swapping costs 1.
  - Do Nothing, Single Exchange, and Frequency Count have Ω(n) competitive factors.
  - **Move to Front algorithm** is 4-competitive. Potential function Φ = 2*number of inversions between MTF and OPT, analysis partitions items before x in MTF by before/after x in OPT.
- **Paging problem** has a disk with N pages and a cache with k < N pages.
  - LRU is k-competitive: adversary can always request item just evicted from cache.
  - Marking algorithm marks page if in cache, else (unmark all if all are marked), then evicts a random unmarked page. Competitive ratio is H_k (kth harmonic number ≤ 1 + ln(k)) for N = k+1. For N > k + 1, competitive ratio is 2*H_k.
- **Missing keys problem** scans ℤ from origin until objective at unknown value n.
  - Linearly increasing scan: 1, -1, 2, -2, etc. requires $\Omega(n^2)$ travel, Ω(n) competitive ratio.
  - Doubling scan 1, -1, 2, -2… = $2(1+1+2+2+\ldots+2^i+2^i)+2*2^{i+1}+|X| = 4(2^{i+1}-1)+2*2^{i+1}+X = 12*2^i + |X| - 4$. With $|X| = -(2^i + \epsilon)$, ratio is 12 + 1 - ϵ = 13.
  - Faster doubling scan: 1, -2, 4, -8… = $2(1+2+\ldots+2^{i+1})+|X| = 2(2^{i+2} - 1) + X = 8*2^i + X - 2$. Competitive ratio is 8 + 1 - ϵ = 9.

EXPERT ADVICE

- **Expert advice** problems ask n entities yes/no question, we make prediction based on answers.
- **Majority-and-halving** goes with majority of remaining experts, excludes expert after one mistake, each mistake excludes half of remainder, we make $\lceil \log_2 n \rceil$ mistakes versus perfect.
  - If best expert makes M mistakes, run majority-and-halving, but reset experts in new "phase" after all eliminated. We made $\log_2 n$ + 1 mistakes per phase, or (M + 1)($\log_2 n$ + 1) overall.
- **Multiplicative weights algorithm** lowers weight to expert by (1 - ϵ) after each mistake. Then sum the weights of experts making each prediction, and pick the weightier prediction.
  - Sum of weights $\Phi = \sum_{i=1}^n w_i$. For ϵ = ½, $\Phi_{new} \leq \frac{3}{4}\Phi_{old}$ and $\Phi_{final} \leq n(3/4)^M$. If best expert i* made M mistakes, $\Phi_{final} \geq w_{i*} = (1/2)^M$. Thus $(1/2)^M \leq n(3/4)^m \Rightarrow (4/3)^m \leq n2^M$ for m algorithm mistakes. Log both sides: $m \log_2(4/3) \leq \log_2 n + M$ or $m \leq 2.41(M + \log_2 n)$.
  - Assuming ϵ ≤ 1/2: As (1 - ϵ) approaches 1, ratio approaches 2, but cannot go below 2.
  - Randomized version chooses prediction with probability equal to weight instead of majority. Expected number of mistakes $m \leq (1+\epsilon)M + \ln n/\epsilon$.

GRADIENT DESCENT

- **Convex Set** A set K is convex if $\lambda x_1 + (1-\lambda)x_2 \in K$.
- **Convex Function** A function f is convex iff $f(\lambda x_1 + (1-\lambda)x_2 \leq \lambda f(x_1) + (1-\lambda) f(x_2)$ (ie. The function should always be less than the secant line)

  Or $f(y) \geq f(x) + < grad(f(x), y - x >$ (ie. The function should always be above the tangent line)

  Basic framework: $x_{t+1} \leftarrow x_t - \eta(grad(f(x_t))$ and return $\hat{x} = \frac{1}{T}\sum_{t=0} x_t$

  For the analysis to work, we need to bound the diameter $D := \|x_0 - x^*\|$ where x^* is the optimal solution and the gradient $\|grad(f(x))\| \leq G$. If $T = \left(\frac{GD}{\epsilon}\right)^2$ $and\ \eta = \frac{D}{G\sqrt{T}}$ then $f(\hat{x}) - f(x^*) \leq \epsilon$.
- **Online gradient descent**: $\sum f_t(x_t) \leq \sum f_t(x^*) + \frac{\eta}{2}G^2T + \frac{1}{2\eta}D^2$. The regret (difference between our solution and the optimal) is $O(\epsilon T) + O\left(\frac{1}{\epsilon}\right)$

AUCTIONS, VCG, MATCHING MARKET

- **VCG standard version** Given a vector of reported valuation functions v
  - Let f(v) be the allocation that maximizes social welfare with respect to v.
  - Let $p_{i(v)} = \max_a(\sum_{j \backslash not\backslash equal\ i} v_j(a) - \sum_{j \backslash not \backslash equal\ i} v_j(f(v))$
- **Market-Clearing Prices Algorithm** St
  - Start with p_i = 0 for all rooms.
  - Build "preferred" graph G
  - If G has perfect matching, done. If no perfect matching, there exists a set of people S such that |S| > |N(S)| → raise price on N(S) by 1 each
  - Special step: If all prices > 0, reduce all by 1

- This algorithm is not poly-time, if the maximum valuation is Vmax, takes $\Omega(Vmax)$to finish. Can be modified and raise the price by the least price instead of 1 (Hungarian algorithm) This can achieve poly time.

GRAPH COMPRESSION

SKETCHING

- **Estimating Euclidean norm** Hash x_1,…, x_n to k buckets using 2-wise hash functions and then multiply each coordinate with a random sign {-1,1} from a 4-wise hash function σ. The linear sketch is $S: R^n \rightarrow R^k$where row is of S is a hash bucket and (Sx)_i is the value in the bucket. Output |Sx|^2. $E[|Sx|^2] = \cdots |x|^2\ and\ Var[|Sx|^2] = O(\frac{|x|^4}{k})$. By Chebyshev's inequality $\Pr[||Sx|^2 - |x|| > \epsilon|x|^2] \leq \frac{Var}{\epsilon^2|x|^4k}$. Set k = 10/e^2 we can estimate |x|^2 up to a (1+epsilon) with probability at least 9/10
- **Finding a non-zero coordinate of a vector:** Given d+1 pairs of coordinates, there is at most one polynomial P(x) of degree at most d that spans these points.
- **1-Sparse Recovery Algorithm:**
  - Maintain $A = \sum x_i\ \ B = \sum x_i \cdot (i)\ and\ C = \sum x_i z^i mod\ p$ where p is a random prime and z is a random integer mod p
  - If B/A is not in {1,2,…n} output FAIL
  - Else, if $C = A \cdot z^{\frac{B}{A}}\ mod\ p$ output N/A. Otherwise output FAIL
- **k-Sparse Recovery Algorithm:** If x has k non-zero entries, hash x to 10k buckets using a 2-universal has function, in each bucket, run 1-sparse finder. Since h is 2-unviersal, with probability at least 1-k/10k = 9/10, one of buckets is 1-sparse
- **Subsampling:** to reduce the space. Uniformly sample half of the coordinates each time and run the k-sparse algorithm in parallel. Note that there are at most log_2(n) subsamples. It can be shown that if we run a k=96-sparse algorithm on every l = log_2k – 5 subsamples, with probability at least 4/5, we output a non-zero item. Space required is O(logn)^2

POLYNOMIALS

- **Few-Roots Theorem** Any non-zero polynomial of degree at most d has at most d roots
- **Corollary:** Given d+1 pairs of coordinates, there is at most one polynomial P(x) of degree at most d that spans these points.
- **Lagrange interpolation example**
- **Error correcting codes:**
  - Suppose k numbers are corrupted → send P(0), P(1), … P(d + k) numbers
  - The receiver will get back at least d+1 numbers, which can uniquely specify P(x)