CS 170 Final Cheat Sheet

Asymptotic Notation

$f(n) = O(g(n))$	There exists $c > 0$ such that for large enough n, $f(n) \leq c \cdot g(n)$.
$f(n) = \Omega(g(n))$	There exists $c > 0$ such that for large enough n, $f(n) \geq c \cdot g(n)$.
$f(n) = \Theta(g(n))$	$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Limit Rule

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{then } f(n) = O(g(n)) \\ c > 0 & \text{then } f(n) = \Theta(g(n)) \\ \infty & \text{then } f(n) = \Omega(g(n)). \end{cases}$$

Divide and Conquer

High Level Approach:

1. Break a problem into subproblems
2. Recursively solve these subproblems
3. Combine the results

Master Theorem: Useful if solving a size-n problem requires recursively solving a size-$\frac{n}{b}$ subproblems and $O(n^d)$ work to prepare/combine the results from the subproblems. In recurrence form:
If

$$T(n) = aT(n/b) + O(n^d) \text{ for } a > 0, b > 1, \text{ and } d \geq 0,$$

where $T(n)$ is the work to solve a problem of size n, then,

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b(a) \\ O(n^d \log n) & \text{if } d = \log_b(a) \\ O(n^{\log_b(a)}) & \text{if } d < \log_b(a) \end{cases}.$$

Mergesort

Main Idea: Break down the list into several sublists until each sublist consists of a single element. Merge the sublists to return the sorted list.
Recurrence: $T(n) = 2T(n/2) + O(n)$. Runtime: $T(n) = O(n \log n)$.

Fast Fourier Transform

Goal: Given a degree d polynomial $P(x) = p_0 + p_1 x + \ldots + p_d x^d$ evaluate P on the nth roots of unity $1, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}$, where $\omega_n = e^{\frac{2\pi i}{n}}$ (n is the smallest power of 2 greater than or equal to $d+1$).
Main Idea: Let $E(x) = p_0 + p_2 x + p_4 x^2 + \ldots$ and $O(x) = p_1 + p_3 x + p_5 x^2 + \ldots$, We can represent $P(x) = E(x^2) + xO(x^2)$.
$P(x) = E(x^2) + x\,O(x^2)$
$P(-x) = E(x^2) - x\,O(x^2)$
The problem reduces to evaluating two degree $n/2 - 1$ polynomials, E and O, on the $n/2$ roots of unity and combining the results.
Runtime: $O(n \log n)$.
Matrix View: We speed up this computation (where n is a power of 2):

$$\begin{bmatrix} P(1) \\ P(\omega_n) \\ P(\omega_n^2) \\ \vdots \\ P(\omega_n^{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega_n^1 & \omega_n^2 & \ldots & \omega_n^{(n-1)} \\ 1 & \omega_n^2 & \omega_n^4 & \ldots & \omega_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{(n-1)} & \omega_n^{2(n-1)} & \ldots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_{n-1} \end{bmatrix}$$

Inverse FFT: Given evaluation of a degree $d \leq n-1$ polynomial $P(x)$ on the nth roots of unity, find P.

Inversion Formula: $M_n(\omega)^{-1} = \frac{1}{n} M_n(\omega^{-1})$

Strongly Connected Components (SCC)

To find an SCC, run DFS on $G^{Reverse}$ to get the post-order of the graph
Run DFS on G starting at the max post-order of $G^{Reverse}$ (sink of G)
When there are no more vertices to explore = one SCC.
Runtime: $O(|V| + |E|)$.

Graph Traversals (DFS & BFS)

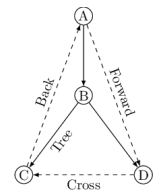
	Pseudocode	Explanation

DFS	`def explore(G,v):`
	`  visited(v) = true`
	`  previsit(v)`
	`  for each edge(v,u) in E:`
	`  if not visited(u):`
	`  explore(u)`
	`  postvisit(v)`
	
	`def dfs(G):`
	`  for all v in V:`
	`  if not visited(v):`
	`  explore(v)`
		- Runtime: $O(\|V\| + \|E\|)$
		- Uses a stack
		- pre(v) = the "time" when v was about to be explored.
		- post(v) = the "time" when v has already been explored.
		- **Topological sort:** If G is a DAG and $(u, v) \in E$, then post(u) > post(v).

BFS	`def bfs(G, s):`
	`  for all u in V :`
	`  dist(u) = infinity`
	`  dist(s) = 0`
	`  Q = [s]`
	`  while Q is not empty:`
	`  v = Q.eject()`
	`  for each edge(v,u) in E:`
	`  if dist(u) == infinity:`
	`  Q.inject(u)`
	`  dist(u) = dist(v) + 1`
		- Runtime: $O(\|V\| + \|E\|)$
		- Uses a queue
		- Treats all edges as having the same length
		- All vertices not connected to start vertex are ignored

Types of Edges

Tree Edge: part of the DFS forest
Forward Edge: leads to a non-child descendent in the tree
Back Edge: leads to an ancestor in the tree
Cross Edge: leads to a node that's neither descendant nor ancestor in the tree



An edge $(u, v) \in E$ is a:
Tree/Forward Edge if
pre(u) < pre(v) < post(v) < post(u) [[ ]] ($uvvu$)
Back Edge if
pre(v) < pre(u) < post(u) < post(v) [[ ]] ($vuuv$)
Cross Edge if
pre(v) < post(v) < pre(u) < post(u) [ ] [ ] ($vvuu$)

Shortest Paths

Dijkstra's Pseudocode	Explanation

`def dijkstra(G,l,s):`
`  for all u in V:`
`  dist(u) = infinity`
`  prev(u) = nil`
`  dist(s) = 0`
`  H = makequeue(V)`
`  while H is not empty:`
`  u = deletemin(H)`
`  for all edges (u,v) in E:`
`  if dist(v) > dist(u) + l(u,v)`
`  dist(v) = dist(u) + l(u,v)`
`  prev(v) = u`
`  decreasekey(H,v)`
	- Runtime: $O((\|V\| + \|E\|) \log(\|V\|))$
	- Calculates shortest path from s to every other vertex
	- Assumes no negative edges

Bellman-Ford Pseudocode	Explanation

`def bellman-ford(G,l,s):`
`  for all u in V :`
`  dist(u) = infinity`
`  prev(u) = nil`
`  dist(s) = 0`
`  repeat |V| - 1 times:`
`  for all e in E:`
`  update(e)`
 	
`def update(e = (u, v)):`
`  d(v) = min(d(v), d(u) + l(u, v))`
	- Runtime: $O(\|V\| \cdot \|E\|)$
	- Procedure updates all edges $\|V\| - 1$ times
	- Detect negative cycles with one extra round of BF, if dist value changes

DAG Shortest Path	Explanation

`def dag-shortest path(G,l,s):`
`  Linearize G`
`  for each u in V in linear order:`
`  for all edges (u, v) in E:`
`  update(u, v)`
	- Runtime: $O(\|V\| + \|E\|)$
	- Works for negative edges
	- Visits vertices in topological order

CS 170 Final Cheat Sheet

Greedy Algorithms

High Level Approach: Greedy algorithms make the locally optimal choice at each step. Hence, greedy algorithms work for problems where making locally optimal choices yields a global optimum.

Minimum Spanning Trees (MST)

Goal: Given a weighted undirected graph $G = (V, E)$, find the lightest weight tree that connects all vertices V.
Cut Property: The lightest edge across a cut is in some MST.
Cut Property 2: Suppose edges X are part of some MST. Let $(S, V \setminus S)$ be any cut for which edges in X do not cross the cut, and let e be the lightest edge across the cut. Then edges $X \cup \{e\}$ are part of some MST.

Kruskal's Algorithm
Main Idea: Repeatedly add the next lightest edge that doesn't produce a cycle.
Runtime: $O(|E| \log(|V|))$.

Prim's Algorithm
Main Idea: On each iteration, pick the lightest edge between a vertex in the current subtree S and a vertex outside S.
Runtime: $O(|E| \log(|V|))$.

Huffman Coding

Goal: Given characters/frequencies $\{(i, f_i)\}$, encode each character in binary so the final encoding is maximally efficient.
Main Idea: Find the two symbols with the smallest frequencies, say i and j, and make them children of a new node (meta-node), which then has frequency $f_i + f_j$, and repeat until one node remaining.
Cost (average bits/character): $\sum_{i=1}^{n} f_i l_i$, where l_i is the number of bits to encode the ith character.

Set Cover (Greedy Approximation)

Goal: Find the minimum number of subsets that cover a set $U = \{1, 2, \ldots, n\}$.
Main Idea: Pick the set S_i with the largest number of uncovered elements. Repeat until all vertices are covered.
Performance: Not optimal, but pretty good: $k_g \leq k_o \cdot \ln(n) + 1$, where k_g is number of subsets output by greedy and k_o is optimal.

Dynamic Programming

High Level Approach: Define subproblems s.t. the solution to a big problem can be easily derived from the solutions to its subproblems. Solve all subproblems from small to large, using results from previous subproblems to solve the current subproblem. Both recursion with memoization (top-down) and iteration (bottom-up) approaches exist.

Shortest Path in a DAG

Goal: Given a DAG, find the length of the shortest path from s to t.
Main Idea: For every $v \in V$, we define $dist(v)$ as the length of the shortest path from s to v. Order of subproblems: topological order. Base cases: $dist(s) = 0$, and $dist(v) = \infty$ if $v \neq s$ is a source. To calculate the shortest path to v, we look at edges (u, v) going into v, we use all the pre-computed $dist(u)$ + the length of the edge (u, v) and find the minimum:

$$dist(v) = \min_{(u,v) \in E} \{dist(u) + l(u, v)\}.$$

Knapsack

Goal: Given knapsack with max capacity W and n items of weight $w_1, \ldots, w_n$ and dollar value $v_1, \ldots, v_n$, find the most valuable combination of unique items to fit in this knapsack
Main Idea: Define $f(i, u)$ as the max value achievable with a knapsack of a capacity u and items $1, ..., i$. The optimal solution to $f(i, u)$ has two cases: to include item i, or to not include it in the knapsack. If we include it, we need to subtract its capacity. Thus, the recurrence is:

$$f(i, u) = \max(f(i - 1, u), f(i - 1, u - w_i) + v_i)$$

Runtime: Storing our $f(i, u)$ values in a 2-D array of $n + 1$ rows and $W + 1$ columns, the algorithm fills this array from left to right, with each entry taking $O(1)$ steps, giving total runtime of $O(nW)$.

Tips

1. Define an appropriate subproblem with relevant parameters, ensuring that the parameters fully determine the subproblem.
2. Given access to all previously solved subproblems, develop an appropriate relation to solve the current subproblem.
3. Test on smaller cases

Max Flow

Network Flow:
A flow f from s to t in a directed graph G whose edges have capacities $c_e \geq 0$ has the following properties:
- Flow on each edge f_e must satisfy: $0 \leq f_e \leq c_e$.
- Conservation of flow: flow entering vertex = flow leaving vertex
Goal: Find maximum flow val(f^*) where val(f) = $\sum_{(s,i) \in E} f_{(s,i)}$.

Ford-Fulkerson Algorithm

Residual graph: A directed graph R whose edge weight is denoted as residual capacity $r(u, v)$:

- $r(u, v) = c(u, v) - f(u, v)$ if $(u, v) \in E$ and $f(u, v) < c(u, v)$
- $r(u, v) = c(u, v) + f(v, u)$ if $(v, u) \in E$ and $f(v, u) > 0$

where $c(u, v)$ is capacity of (u, v) on G ($c(u, v) = 0$ if edge doesn't exist), $f(u, v)$ is flow on (u, v). For Example: if $c(u, v) = 3$, $c(v, u) = 4$ and $f(u, v) = 2$ then $r(u, v) = 1$ and $r(v, u) = 6$.
Algorithm:

1. Find a path from s to t and route max flow through this path
2. Update capacities by updating the residual graph
3. Repeat 1 and 2 until algorithm halts (no path from s to t is found)

Explanation:
- Intuition: The back edges in residual graph can undo suboptimal flows in some edges for future iterations.
- Runtime: $O(|V| \cdot |E|^2)$ if using BFS to find paths.

Max-Flow Min-Cut Theorem

An (s, t)-cut partitions V into two subsets S, T, where $s \in S$, $t \in T$.
Capacity of (s, t)-cut C: val(C) = $\sum_{e \in \{(u,v) | u \in S, v \in T\}} c_e$.
Theorem: val(f^*) = val(C^*). Maximum flow is equal to minimum cut. In the residual graph, the set of vertices, X, reachable from s, yields the cut $(X, V \setminus X)$ whose capacity equals the max flow.

Linear Programming

High Level Approach: Define your problem as an objective function with a set of linear constraints while following the given convention, and the Simplex algorithm does the rest. If a linear program has a bounded optimum, then so does it's dual, and the two optimum values coincide.
Generic Linear Program:

Primal:	Dual:
$\max \mathbf{c}^T \mathbf{x}$	$\min \mathbf{y}^T \mathbf{b}$
$\mathbf{A}\mathbf{x} \leq \mathbf{b}$	$\mathbf{y}^T \mathbf{A} \geq \mathbf{c}^T$
$\mathbf{x} \geq 0$	$\mathbf{y} \geq 0$

Constraint Transformations:

Changing Objective	Inequality to Equality
$\max \mathbf{c}^T \mathbf{x} = \min -\mathbf{c}^T \mathbf{x}$ $\min \mathbf{c}^T \mathbf{x} = \max -\mathbf{c}^T \mathbf{x}$	$ax \leq b \quad \rightarrow \quad ax + s = b,$ $s \geq 0$
Equality to Inequality	**Unrestricted Variable**
$ax = b \quad \rightarrow \quad ax \leq b,$ $ax \geq b$	$\mathrm{x} \in \mathbb{R} \quad \rightarrow \quad \mathrm{x} = \mathrm{x}_+ - \mathrm{x}_-$ $\mathrm{x}_+, \mathrm{x}_- \geq 0$

Zero-Sum Games

Setup: Given a 2D matrix where the rows are player A's moves, the columns are player B's moves, and the values are A's reward for each move. Whichever player goes first has to announce their strategy first.
Strategy is the vector probabilities of playing specific moves
Main Idea: If player A goes first, player B will pick the move that minimizes A's reward. So A's best strategy will be the maximum of the minimum of B's moves. If player B goes first, player A will pick the move that maximizes A's reward. So B's best move will be the minimum of A's moves. These two equations come out to be the dual of each other.

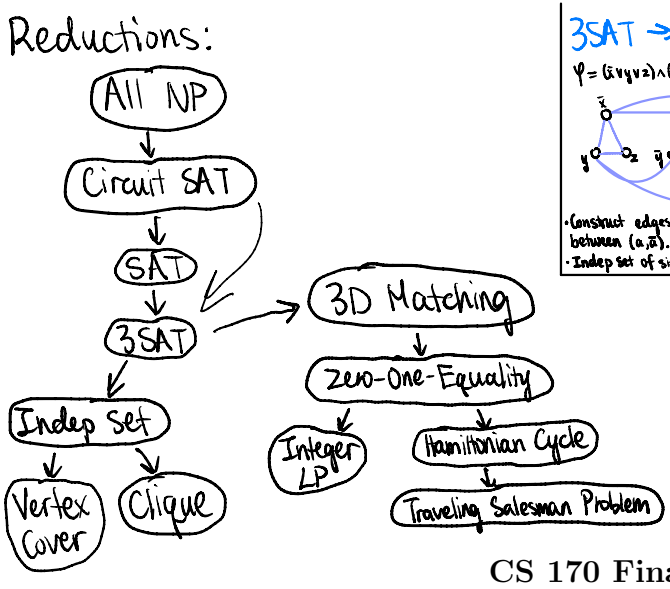
Example:

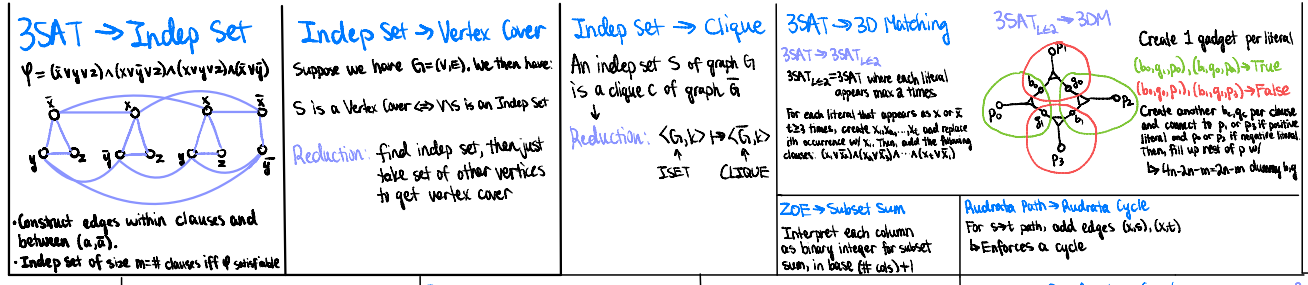
	$B1$	$B2$
$A1$	a	c
$A2$	b	d

A's strategy is $[x_1, x_2]$
B's strategy is $[y_1, y_2]$
A picks $[x_1, x_2]$ to maximize the value $\min\{ax_1 + bx_2, cx_1 + dx_2\}$
B picks $[y_1, y_2]$ to minimize the value $\max\{ay_1 + cy_2, by_1 + dy_2\}$
Min-Max Theorem: $\max_x \min_y \sum_{i,j} G_{ij} x_i y_j = \min_y \max_x \sum_{i,j} G_{ij} x_i y_j$

Simplex

Main Idea: The optimal point is at an intersection of the geometrical representation of the constraints. If the chosen vertex is not optimal, then the objective function can be improved by following an outgoing edge from this vertex.





CS 170 Final Cheat Sheet

Multiplicative Weights

Main Idea: Create algorithm performing approximately as well as following best of n experts that we know beforehand

Setup: At each time step $t = 1, \ldots, n$ we choose n possible strategies, producing array $x^{(t)} = (x_1^{(t)}, \ldots, x_n^{(t)})$, where $x_i^{(t)}$ is the probability we follow/fraction of budget allocated in strategy i. After choosing $x^{(t)}$, we are presented loss array $l^{(t)} = (l_1^{(t)}, \ldots, l_n^{(t)})$ $(0 \le l_i^{(t)} \le 1)$, and therefore our loss at time t is

$$\sum_{i=1} x_i^{(t)} \cdot l_i^{(t)}$$

We then define regret R_T at time T as such:

$$R_T = \sum_{t=1}^{T} \sum_{i=1}^{n} x_i^{(t)} \cdot l_i^{(t)} - \min_i \sum_{t=1}^{T} l_i^t$$

Multiplicative Weight Algorithm

Use parameter $0 \le \epsilon \le 1/2$, maintaining at each time step t a set of weights

$$w_i^{(0)} = 1, w_i^{(t+1)} = w_i^{(t)} \cdot (1-\epsilon)^{l_i^{(t)}}$$

At each step, algorithm chooses allocations proportionally to weights

$$x_i^{(t)} = \frac{w_i^{(t)}}{\sum_j w_j^{(t)}}$$

Implications: This Multiplicative Weight Algorithm guarantees

$$R_T \le 2\sqrt{T \ln n}$$

Approx Algorithms:

TSP (Metric) 2-Approximation

1. Compute MST T^*
2. Traverse T^* via DFS pre-order tour

} $n \log n$ runtime

Karger Contraction Algorithm (approximates min-cut)

1. Pick an edge uniformly at random and contract along that edge (combine ends of edge into a metavertex)
2. Repeat step 1 until there are two meta-vertices remaining; these meta-vertices represent the cut

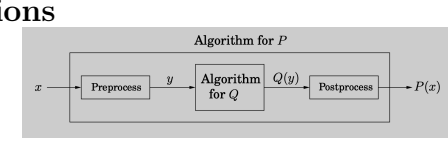
} $O(rn^2)$ runtime where r=# of times we repeat these steps

$\mathbb{P}[\text{Fail to find Mincut}] \le (1 - \frac{1}{\binom{n}{2}})^r \le e^{-\frac{r}{\binom{n}{2}}}$

Probability Stuff:

- $\mathbb{E}[X] = \sum_{x \in X} x\,\mathbb{P}[X=x]$
- Markov's Ineq: $\mathbb{P}[X > k] \le \frac{\mathbb{E}[X]}{k}$
- Chebyshevs Ineq: $\mathbb{P}[|X - \mathbb{E}[X]| \ge \epsilon] \le \frac{\mathrm{Var}(X)}{\epsilon^2}$

Reductions



Reducing from P to Q $\equiv$ $P \to Q$ $\equiv$ $P \le Q$.
Both pre-processing and post-processing run in polynomial time. Implication: If there exists an efficient algorithm for problem Q, then there exists an efficient algorithm for P. If there's no efficient algorithm for problem P, then there is no efficient algorithm for Q.

Example: Reducing Bipartite Matching to Max-Flow

Goal: Find maximum number of matching pairs (pairs of vertices with connected edges) in a bipartite graph $G = (L, R, E)$.

1. Pre-Process: add a source node s and a sink node t, and connect s to all vertices in L and all vertices in R to t. Give capacities 1 to all edges, and direct them from s to L to R to t.
2. Run Max-Flow algorithm.
3. Post-Process: assuming flow is integral, take all edges saturated by the flow that are between L and R.

NP-Completeness

P: if this problem can be **solved** in polynomial time
NP: if a solution of this problem can be **verified** in polynomial time
NP-Hard: if all problems in NP reduce to this problem
NP-Complete: if this problem is in NP and NP-Hard
How to Prove that Problem A is NP-complete:

1. Show that A is in NP by designing a verification algorithm
2. Choose an NP-complete problem B
3. Show that A is NP-Hard by reducing B to A
4. 1 and 3 shows that A is NP-Complete

Examples of NP-Complete problems

3SAT: Given a set of clauses, each containing between 1 and 3 literals, for example $(\bar{x} \vee y \vee \bar{z})(x \vee \bar{y} \vee z)(\bar{x} \vee \bar{y})$, find a satisfying truth assignment.
Independent Set: Given a graph and a number g, find a set of g pairwise non-adjacent vertices.
Vertex Cover: Given a graph and a budget b, find b vertices such that an endpoint of every edge is in the cover set.
Integer Linear Programming: Given a system of linear inequalities, find a feasible integer solution.
Rudrata Path: Given a graph g and two vertices s and t, find a path starting at s and ending at t going through each vertex exactly once.
Set Cover: Given a set of elements E and and several subsets of it $S_1, \ldots, S_m$ with budget b, select b of these subsets s.t. their union is E.

Approximation Algorithm

Setting: Find an approximately optimal solution to an optimization problem A.
Approximation Ratio: An approx algorithm with an approx ratio α to problem A means that the solution found by this algorithm is between $OPT(A)$ and $\alpha \cdot OPT(A)$. $\alpha > 1$ for minimization problems and $\alpha < 1$ for maximization problems.
Example: 2-Approximation for Vertex Cover

- While there are edges: select any edge e and delete all edges that share endpoints with e.
- Approximated vertex cover is the endpoints of the selected edges.
- In order to cover all the edges, the optimal vertex cover must be at least the number of edges selected, which is at least one half of the number of endpoints from the approximation algorithm above.
- Therefore the result has at most twice as much as the optimal solution, and achieves approximation ratio $\alpha = 2$.

Hashing and Streaming

Main Idea: Map a large domain to n buckets using modular arithmetic.

Universal Family of Hash Functions:

Definitions:
1. For all $x \ne y$, exactly $\frac{|\mathcal{H}|}{n}$ hash functions from family $\mathcal{H}$ map x and y to the same value.
2. For all $x \ne y$ it holds that $P[h_a(x) = h_a(y)] \le \frac{1}{n}$

Note: n is often prime.
Examples:
1-var: $\mathcal{H} = \{h_a : a \in \{0, 1, \ldots n-1\}\}$ i.e. $h_a(x) = a \cdot x \bmod n$
m-vars: $\mathcal{H} = \{h_a : a \in \{0, 1, \ldots n-1\}^m\}$ i.e. $h_a(\vec{x}) = \sum_{i=1}^{m} a_i \cdot x_i \bmod n$

Streaming:

Main Idea: Algorithms that work in sub-linear space to handle an indefinite sequence of data
Frequency Moments: m_i = # of times i appeared in the stream
$F_0 = \sum_i m_i^0$ = # of distinct elements
$F_1 = \sum_i m_i^1$ = Total number of elements
$F_2 = \sum_i m_i^2 \approx$ variance

Counting Distinct Elements:

$N = F_0$; if $N \le k$: return "small"; if $N \ge 2k$ return "large"
Algorithm:
Main idea: if any number in stream hashes to zero, N is relatively large
1. Choose pairwise-independent $h_{a,b}(x) = ax + b \bmod B$, B is O(k)
2. If any x_i in stream has $h(x_i) = 0$, output "large", else output "small"

Analysis:
$\Pr(large | N \le k) = \frac{k}{B} = \frac{1}{4}$ if $B = 4k$
$\Pr(large | N \ge 2k) \ge \frac{2k}{B}(1 - \frac{k}{B}) = \frac{3}{8}$ if $B = 4k$ (by Inclusion-Exclusion).

Las Vegas: always correct, runtime random.
↳ We want to put an upper bound on $\mathbb{E}_r[T(A(x,r))]$ $\forall x$ (i.e worst-case x)

Monte Carlo: always fast, correctness random
↳ We put a lower bound on $\mathbb{P}_r[A(x,r)$ is correct$]$ (worst correctness rate)

Static: query only
Dynamic: insert, delete, query

Counting:
Naive: Store a variable x that's incremented every time
Memory: $O(\log N)$ bits
Morris's: Run R copies in parallel for estimator, and change base for exponentiation in increment probability
Memory: $O(\log\log N + \log\frac{1}{\varepsilon} + \log\frac{1}{\delta})$ bits

Unique Counting: Might be repeats
Stream of m items coming from some universe $\{1, \ldots, n\}$, want to find number of distinct elements in this stream
Algorithm: $h \in$ 2-wise independent H
init(): X=1, pick random $h: [n] \to [0,1]$
update(i): $X \to \min(X, h(i))$
query: $\frac{1}{X} - 1$
Memory: $O(\frac{1}{\varepsilon^2} + \log n)$