AnW Speedrun (FS25) - Shivram Sambhus @ cs.shivi.io

1 TOC

2 Introduction: Probability in CS

Probability is a core tool in modern CS, acting as a "superpower" for:

- **Efficiency:** Designing faster/simpler algorithms (e.g., Randomized QuickSort, Hashing).
- **Symmetry Breaking:** Solving contention in distributed systems where deterministic approaches get stuck.
- **Modeling Uncertainty:** Handling noisy data in machine learning and statistics.
- **Hard Problems:** Creating efficient probabilistic tests for problems where deterministic solutions are slow (e.g., Primality Testing), or randomized approximations for NP-hard problems.

3 Core Algorithms & Data Structures (Recap)

3.1 Sorting Algorithms

Comparison-Based Sorting

- **Bubble Sort ($O(n^2)$):** Repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The largest unsorted element "bubbles" to its correct position in each pass. Simple but highly inefficient.
- **Selection Sort ($O(n^2)$):** In each pass, finds the minimum element from the unsorted part and puts it at the beginning of the unsorted part. Makes the minimum number of swaps, but still requires quadratic comparisons.
- **Insertion Sort ($O(n^2)$):** Builds the final sorted array one item at a time. It iterates through the input elements and inserts each element into its correct position in the sorted part of the array. Efficient for small or nearly-sorted lists because it does little work if the element is already close to its final position.

Divide & Conquer Sorting

- **Merge Sort ($O(n \log n)$):** Divides the array into two halves, recursively sorts them, and then merges the two sorted halves. The "merge" step is the key operation. Stable and guarantees $O(n \log n)$ performance, but requires extra space.
- **Quick Sort ($O(n \log n)$ average):** Picks a 'pivot' element and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. Its performance hinges entirely on choosing good pivots; a random pivot makes the worst case very unlikely. Worst case is $O(n^2)$ if pivots are chosen poorly.
- **Heap Sort ($O(n \log n)$):** Uses a Max-Heap data structure. First, it builds a max-heap from the input data. Then, it repeatedly extracts the maximum element from the heap (the root) and places it at the end of the sorted array. It's an in-place sort with guaranteed $O(n \log n)$ performance.

3.2 Core Data Structures

Heap

A binary tree satisfying the **heap property**: the value of each parent node is greater than or equal to the value of its children (for a Max-Heap). Stored efficiently as an array.

- **Extract-Max ($O(\log n)$):** Swap root with last element, remove last, then "sift down" the new root to restore the heap property. The work is proportional to the height of the tree.
- **Insert ($O(\log n)$):** Add element at the end, then "sift up" to restore the heap property. Again, work is proportional to tree height.
- **Heapify ($O(n)$):** Build a heap from an unsorted array in linear time by sifting down from the last non-leaf node up to the root. **Intuition for $O(n)$:** While it's $\frac{n}{2}$ calls to sift-down (which can be $O(\log n)$), most nodes are near the bottom of the tree and sift down very little. A tight analysis shows the total work is linear.

Binary Search Tree (BST)

A binary tree where for each node, all values in the left subtree are less than the node's value, and all values in the right subtree are greater.

- **Search, Insert, Delete:** Average time $O(\log n)$. Worst case $O(n)$ if the tree is unbalanced (skewed), becoming a linked list.
- **Deletion:** The tricky case is deleting a node with two children. Replace it with its **in-order successor** (the smallest node in its right subtree), then delete the successor. **Why the successor?** It's the next largest value, so it maintains the BST property with minimal disruption.

AVL Tree (Balanced BST)

A self-balancing BST. It maintains the **AVL Property**: for every node, the heights of its two child subtrees differ by at most one.

- **Rebalancing:** After an insertion or deletion that violates the property, balance is restored via **rotations** (single or double). Rotations are local restructuring operations that change parent-child relationships to fix height imbalances while preserving the BST property. This keeps all operations at a guaranteed worst-case $O(\log n)$.

Union-Find

Tracks a collection of disjoint sets. Key for algorithms like Kruskal's.

- **`find(x)`**: Determines the representative (root) of the set containing x.
- **`union(x, y)`**: Merges the sets containing x and y.
- **Optimizations:**
  1. **Union by Size/Rank:** Always attach the smaller tree to the root of the larger tree to keep the trees shallow. This avoids creating long, spindly trees.
  2. **Path Compression:** When running `find(x)`, make every node on the path from x to the root point directly to the root. This dramatically flattens the tree structure for future `find` operations in that subtree.

With both optimizations, the amortized time per operation is nearly constant, $O(\alpha(n))$.

3.3 Dynamic Programming (DP)

SRTBOT Framework

A structured way to approach DP problems.

1. **Subproblem:** Define smaller, overlapping versions of the problem (e.g., `dp[i]`, `dp[i][j]`).
2. **Recurrence:** Express the solution to a subproblem in terms of smaller subproblems.
3. **Topological Order:** Solve subproblems in an order that ensures dependencies are met (e.g., small `i` to large `i`).
4. **Base Case:** Define the solution for the smallest subproblems.
5. **Original Problem:** Express the final answer in terms of solved subproblems.
6. **Time Complexity:** Analyze `#subproblems` $\times$ `work per subproblem`.

Common DP Patterns

- **Longest Common Subsequence ($O(nm)$):** `dp[i][j]` = LCS of `A[:i]` and `B[:j]`. Recurrence considers if `A[i]==B[j]` (match and extend) or not (take max of dropping one character from either string).
- **Edit Distance ($O(nm)$):** `dp[i][j]` = min cost to convert `A[:i]` to `B[:j]`. Recurrence takes min of insert, delete, substitute operations, representing the three ways to resolve a mismatch.
- **Knapsack ($O(nW)$):** `dp[i][w]` = max profit using items `1..i` with capacity `w`. Recurrence decides whether to include item `i` (if it fits) or not, a classic "take it or leave it" choice.
- **Longest Increasing Subsequence ($O(n^2)$):** `dp[i]` = length of LIS ending at index `i`. Recurrence looks at all previous elements `j<i` to find the best one to extend. (Can be optimized to $O(n \log n)$).

3.4 Graph Traversal & Shortest Paths

Traversal

- **Breadth-First Search (BFS):** Explores layer by layer using a queue. It's a "wide and shallow" search. Finds shortest paths in **unweighted** graphs. $O(V + E)$.
- **Depth-First Search (DFS):** Explores as deeply as possible using recursion (a stack). It's a "deep and narrow" search. Useful for topological sort, finding cycles, and connectivity. $O(V + E)$.

Single-Source Shortest Path (SSSP)

- **Dijkstra's Algorithm:** For graphs with **non-negative** edge weights. Uses a priority queue to greedily select the closest unvisited node. **Intuition:** Since edge weights are non-negative, once we declare a shortest path to a node `u`, we know we can't find a cheaper path later, because any detour would only add more weight.
- **Bellman-Ford Algorithm:** For graphs that may have **negative** edge weights. Relaxes all edges $|V| - 1$ times. **Intuition:** A shortest path can have at most $|V| - 1|$ edges. The algorithm finds all shortest paths of length 1, then length 2, and so on, up to $|V| - 1|$. It can detect negative-weight cycles if distances still improve on the $|V|$-th iteration. Runtime $O(VE)$.

All-Pairs Shortest Path (APSP)

- **Floyd-Warshall ($O(V^3)$):** DP approach. `dist[i][j]` is iteratively improved by considering each vertex `k` as an intermediate point. **Intuition:** "Is the path from `i` to `j` shorter if we go through `k`?" It asks this for all `i,j,k`.
- **Johnson's Algorithm ($O(VE + V^2 \log V)$):** For sparse graphs. Uses Bellman-Ford to re-weight edges to be non-negative, then runs Dijkstra from every vertex. A clever way to make a graph "Dijkstra-safe".

3.5 Minimum Spanning Tree (MST)

Greedy MST Algorithms

All MST algorithms are based on a greedy approach that is proven correct by the "Cut Property": for any cut in the graph, the minimum-weight edge crossing the cut must be in some MST. **Intuition:** To connect two sets of vertices (the two sides of the cut), you must use an edge between them. To build a **minimum** spanning tree, you should always choose the **cheapest** such connecting edge.

- **Prim's Algorithm ($O(E \log V)$):** Grows a single tree by adding the cheapest edge that connects a vertex in the tree to a vertex outside the tree. Similar to Dijkstra.

- **Kruskal's Algorithm ($O(E \log E)$):** Sorts all edges by weight and adds them to the MST if they don't form a cycle (checked using Union-Find). It builds a "forest" of components that gradually merge.

4 Graph Theory & Algorithms

4.1 Graph Connectivity

This section covers how to measure the "robustness" of a graph's connections. We formalize this using vertex and edge connectivity, which are deeply linked to the number of independent paths between nodes by Menger's Theorem.

Definitions

- **k-vertex-connected**: $|V| \geq k+1$, graph remains connected after removing fewer than k vertices.
- **k-edge-connected**: Graph remains connected after removing fewer than k edges.
- **Cut Vertex**: A single vertex whose removal disconnects the graph.
- **Bridge**: A single edge whose removal disconnects the graph.

Menger's Theorem (1927)

This is the fundamental theorem linking connectivity to paths.

- **Vertex Version**: k-vertex-connected $\iff$ k internally vertex-disjoint paths exist between any two distinct u, v.
- **Edge Version**: k-edge-connected $\iff$ k edge-disjoint paths exist between any two distinct u, v.

TA Tip: Think of this as network reliability. To cut k phone lines (k-edge-connected), there must be k independent routes that don't share any lines.

Connectivity Hierarchy

For any graph G:

$$\kappa(G) \leq \lambda(G) \leq \delta(G)$$

. **Why?** Removing all edges from a vertex of min-degree $\delta(G)$ disconnects it, so $\lambda(G) \leq \delta(G)$. An edge cut can be "simulated" by removing vertex endpoints, so $\kappa(G) \leq \lambda(G)$.

Finding Cut Vertices/Bridges ($O(|V| + |E|)$)

This algorithm uses a single Depth-First Search (DFS) to find all cut vertices and bridges. The core idea is to determine if any subtree in the DFS tree is "trapped" and can only connect to the rest of the graph through its single parent node or edge in the tree.

The Intuition: `dfs` vs. `low` numbers

During the DFS, we assign two numbers to each vertex v:

- **`"dfs"[v]` (Discovery Time):** A counter for when we first visit v. This defines the DFS tree structure. Ancestors always have smaller `"dfs"` numbers than descendants.
- **`"low"[v]` (Low-Link Value):** The lowest discovery time reachable from v (including from its entire subtree) by following tree edges down and then at most **one** back edge up.

Key Idea: The `"low"[v]` value tells us the "highest" ancestor (smallest `"dfs"` number) that vertex v or any of its descendants can reach using a "shortcut" (a back edge). It essentially answers the question: **"Can the subtree at v escape upwards without using the parent-child tree edge?"**

The Calculation: During DFS

We compute `"low"[v]` as we traverse:

1. **Initialization:** When we discover v, its best known shortcut is to itself. So, we initialize $\text{low}[v] = \text{dfs}[v]$.
2. **Exploring from u to neighbor v:**
   - **If (u, v) is a tree edge (v is a new child):** We must first explore v's subtree by calling `dfs(v)`. After the call returns, `low[v]` holds the highest ancestor reachable from the child's subtree. The parent `u` can also reach that high, so we update: `"low"[u] = min("low"[u], "low"[v])`.
   - **If (u, v) is a back edge (v is an ancestor):** We've found a direct shortcut from u to ancestor v. This shortcut lets u reach a node with discovery time `"dfs"[v]`. We update: `"low"[u] = min("low"[u], "dfs"[v])`.

The Conditions: Identifying Critical Points

After the recursive call `dfs(v)` returns to its parent `u`, we have the final `low[v]` value and can check for critical connections.

- **Cut Vertex Condition (for non-root u):** u is a cut vertex if it has a child v such that

  $$\text{low}[v] \geq \text{dfs}[u]$$

  Justification: This means the best shortcut from v's entire subtree can only reach back to u itself, or somewhere below u. It **cannot** find a path to an ancestor **above** u. Therefore, u is the sole connection point for the entire subtree of v to the rest of the graph. Removing u separates this subtree.

- **Bridge Condition:** The tree edge (u, v) is a bridge if

  $$\text{low}[v] > \text{dfs}[u]$$

  Justification: This is a stricter condition. It means the best shortcut from v's subtree cannot even reach back to its parent u via another path. If it could (`"low"[v] == "dfs"[u]`), the edge (u, v) would be part of a cycle and not a bridge. The strict inequality ensures that (u, v) is the **only** path connecting v's subtree to the rest of the graph.

- **Root as a Cut Vertex:** The root of the DFS tree is a cut vertex if it has more than one child. This is because the subtrees of its children are only connected via the root.

```
# Core logic inside dfs(u) after recursive call for
child v
dfs(v)
low[u] = min(low[u], low[v])
if low[v] >= dfs_num[u]: # u is a potential cut
vertex
    ...
if low[v] > dfs_num[u]: # (u,v) is a bridge
    ...
```

Blocks & Block-Cut Tree

- **Block**: A maximal 2-edge-connected subgraph. Conceptually, these are the "robust" components of a graph. Two edges are in the same block if they lie on a common cycle. Blocks are the "islands of stability" where removing any single vertex doesn't disconnect other parts of the block.
- **Block-Cut Tree**: A high-level structural view of the graph. It's a bipartite tree with nodes for cut vertices and nodes for blocks, showing how the robust components are hinged together by the critical cut vertices. It reveals the graph's skeleton.

4.2 Cycles: Eulerian vs. Hamiltonian

This section contrasts two fundamental types of cycles. Eulerian cycles are about traversing edges and are computationally easy. Hamiltonian cycles are about visiting vertices and are famously hard, representing a cornerstone of NP-completeness.

Eulerian Tours (Edges)

A closed walk visiting every **edge** exactly once. Exists $\iff$ the graph is connected and every vertex has **even degree**. **Intuition:** For every time you enter a vertex via an edge, you must leave via a different edge. This pairs up the edges at each vertex, requiring an even degree. Found in $O(|E|)$ time (Hierholzer's).

Hamiltonian Cycles (Vertices)

A cycle visiting every **vertex** exactly once. NP-complete.

- **Dirac's Thm:** If $n \geq 3$ and min degree $\delta(G) \geq \frac{n}{2}$, a Hamiltonian cycle exists.
- **DP Solution ($O(n^2 2^n)$):** The state `P[S,x]` solves the subproblem of finding a path that covers a specific subset `S` and ends at `x`. The algorithm builds longer paths from shorter ones. **Intuition:** The state `dp[(mask, u_idx)]` asks "Is there a path from the start node to node `u_idx` that visits exactly the set of nodes represented by `mask`?" The recurrence tries to extend existing paths: if we have a path to `u` covering `mask`, we can extend it to a neighbor `v` (if `v` is not in `mask`) to form a path to `v` covering `mask | {v}`.

```
# Conceptual pseudocode for Hamiltonian Cycle DP
def has_hamiltonian_cycle(graph):
    n = len(graph)
    # dp[(subset_mask, last_node_idx)] -> bool
    dp = {}
    start_idx = 0
    dp[(1 << start_idx, start_idx)] = True # Base
case

    for mask in range(1, 1 << n):
        for u_idx in range(n):
            if (mask & (1 << u_idx)) and
dp.get((mask, u_idx)):
                # Try to extend path from u to
neighbor v
                for v_idx in range(n):
                    if not (mask & (1 << v_idx)) and
is_neighbor(u_idx, v_idx):
                        dp[(mask | (1 << v_idx),
v_idx)] = True

    # Final check: is there a full path ending at a
neighbor of start?
    full_mask = (1 << n) - 1
    for u_idx in range(n):
        if dp.get((full_mask, u_idx)) and
is_neighbor(u_idx, start_idx):
            return True
    return False
```

4.3 Traveling Salesman Problem (TSP)

The classic problem of finding the shortest route visiting a set of cities. While the general version is NP-hard and inapproximable, the practical Metric TSP variant (where distances obey the triangle inequality) allows for good approximation algorithms.

Problem & Complexity

Find the shortest tour visiting all cities. NP-hard. **Metric TSP:** Distances satisfy triangle inequality ($l(x, z) \leq l(x, y) + l(y, z)$). This prevents arbitrarily long "detours" and makes approximation possible.

2-Approximation for Metric TSP

The strategy is to build a cheap "skeleton" that connects all points (an MST), then make it traversable (an Eulerian tour) and finally convert it to a valid TSP tour.

```
def tsp_2_approx(graph):
    # 1. Compute MST (a low-cost connected subgraph)
    mst = minimum_spanning_tree(graph)
    # 2. Get a tour by traversing the MST (e.g.,
preorder)
    preorder_nodes = preorder_traversal(mst)
    # 3. Shortcut the path by skipping visited
nodes.
    tour = list(dict.fromkeys(preorder_nodes))
    tour.append(tour[0])
    return tour
```

Analysis: $\text{Length}(\text{Tour}) \leq 2 * \text{Length}(\text{MST}) \leq 2 * \text{Length}(\text{Optimal Tour})$. **Why?** (1) A preorder walk on the MST traverses every edge twice. The final tour only shortcuts this walk, so its length is at most $2 * \text{Length}(\text{MST})$. (2) An optimal TSP tour is a connected subgraph. The MST is the **minimum weight** connected subgraph, so $\text{Length}(\text{MST}) \leq \text{Length}(\text{Optimal Tour})$.

Christofides' 1.5-Approximation

This algorithm improves on the 2-approximation by being smarter. Instead of doubling all edges, it only adds the cheapest possible edges (a perfect matching) to fix the "odd-degree" vertices.

1. Compute MST T.
2. Find vertices X with odd degree in T.

3. Find a minimum-weight perfect matching M on X.
4. Combine T and M (all degrees are now even).
5. Find an Eulerian tour and shortcut it.

Analysis: $\text{Length(Tour)} \le \text{l}(T) + \text{l}(M) \le \text{opt} + 0.5 * \text{opt} = 1.5 * \text{opt}$. The key step is showing $\text{l}(M) \le 0.5 * \text{opt}$. This relies on the triangle inequality applied to the optimal tour's path among the odd-degree vertices.

4.4 Matchings in Graphs

Matching is a fundamental pairing problem. The key to finding optimal matchings is the concept of augmenting paths, which provide a way to iteratively improve a non-optimal solution.

Definitions & Core Idea

- **Matching M**: A set of edges with no common vertices.
- **M-Augmenting Path**: Alternating path starting and ending at unmatched vertices.
- **Berge's Thm:** M is maximum $\iff$ no M-augmenting path exists.

TA Tip: An augmenting path is a "chain reaction" for improvement. By flipping the status of edges along the path, you remove k matched edges but add $k+1$, for a net gain of one.

Bipartite Matching

- **Hall's Marriage Thm:** In $G = (A \bigcup B, E)$, a matching covering A exists $\iff$ **Hall's Condition** holds: $\forall X \subset A, |N(X)| \ge |X|$. **Intuition:** For any group of suitors X from set A, their combined set of potential partners $N(X)$ in set B must be at least as large as their own group. If not, there aren't enough partners to go around for that group.
- **Hopcroft-Karp ($O\left(\sqrt{|V|}|E|\right)$):** An efficient algorithm that, in each phase, finds a **maximal set of shortest vertex-disjoint augmenting paths** using BFS (to find length) and DFS (to find paths), then augments them all at once. The path length strictly increases, limiting the number of phases. It's a "smarter" version of augmenting one path at a time.

4.5 Graph Coloring

Assigning colors to vertices so no neighbors share a color is a model for many resource allocation problems (e.g., scheduling). The problem is computationally hard, so we often rely on heuristics.

Definitions & Complexity

- **k-Coloring**: Assignment of k colors to vertices so no adjacent vertices share a color.
- **Chromatic Number $\chi(G)$**: Minimum k for a k-coloring.

Complexity: 3-Colorability is NP-complete.

Greedy Coloring

The idea is simple: iterate through vertices and assign each the first available color. The performance depends heavily on the order.

```python
def greedy_coloring(graph, vertex_order):
    coloring = {}
    for v in vertex_order:
        neighbor_colors = {coloring.get(n) for n in graph.get(v, [])}
        color = 1
        while color in neighbor_colors:
            color += 1
        coloring[v] = color
    return coloring
```

Performance: Uses at most $\Delta(G) + 1$ colors. A **smallest-last** ordering heuristic (saving low-degree vertices for last) often improves results. **Intuition:** By coloring low-degree vertices last, we save the "easy" cases for when color choices are most constrained. This gives the high-degree vertices (which are harder to color) more options early on.

4.6 Max-Flow & Min-Cut

This is a cornerstone of network optimization. The Max-Flow Min-Cut theorem provides a beautiful duality between the maximum throughput of a network and its narrowest bottleneck.

Definitions

- **Flow Network**: A directed graph with edge capacities, a source s, and a sink t.
- **Flow f**: Satisfies capacity constraints and flow conservation (inflow = outflow for intermediate nodes).
- **Cut (S,T)**: A partition of V with $s \in S, t \in T$. Its capacity $\text{cap}(S,T)$ is the sum of capacities of edges from S to T.

Max-Flow Min-Cut Theorem

Max value of an $s-t$ flow = Min capacity of an $s-t$ cut.
Proof Idea: When Ford-Fulkerson terminates, there are no more augmenting paths. The set of vertices still reachable from s in the residual graph forms a min-cut whose capacity equals the computed max flow.

Ford-Fulkerson Algorithm

The strategy is to iteratively find any path from source to sink with available capacity and push as much flow as possible along it. This process is repeated until no such paths can be found. **Intuition:** It's a greedy approach. Find a pipe with spare capacity and pump more water through it. The magic of the residual graph is that it allows the algorithm to "undo" bad choices by pushing flow backward, effectively rerouting flow to find a better overall solution.

```python
def ford_fulkerson(graph, s, t):
    # G_f is the residual graph, flow f is initially 0
    while (path := find_path_in_residual(G_f, s, t)):
        path_flow = min_capacity_on_path(path)
        # Augment flow along path and update residual graph
        for u, v in path_edges(path):
            f[u,v] += path_flow; f[v,u] -= path_flow
            G_f[u,v] -= path_flow # Decrease forward capacity
            G_f[v,u] += path_flow # Increase backward capacity
    return total_flow_from_s(f)
```

TA Tip: The residual graph is key. A forward edge means "push more flow." A backward edge means "cancel/reroute existing flow."

5 Probability & Randomized Algos

5.1 Probability Foundations

Basics

- **Conditional Probability:** $\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]}$.
- **Bayes' Thm:** $\Pr[B|A] = \frac{\Pr[A|B]\Pr[B]}{\Pr[A]}$. Updates prior beliefs with evidence.
- **Independence:** Events A, B are independent if $\Pr[A \cap B] = \Pr[A]\Pr[B]$.

Combinatorics: Counting Choices

Type	Order Matters?	Replacement?	Formula
Tuple	Yes	Yes	n^k
k-Permutation	Yes	No	$\frac{n!}{(n-k)!}$
Combination	No	No	$\binom{n}{k}$
Multiset	No	Yes	$\binom{n+k-1}{k}$

5.2 Random Variables (RVs)

An RV is a numerical summary of a random outcome. Expectation gives the average, while variance measures the spread. Linearity of Expectation is the single most important tool for analyzing randomized algorithms.

Expectation & Variance

- **Expected Value $E[X]$**: Weighted average. **Linearity:** $E[\sum a_i X_i] = \sum a_i E[X_i]$. **Always holds, even for dependent RVs!** This is a superpower because it lets us break down a complex random variable into a sum of simple indicator variables. **Indicator RVs:** $E[I_A] = \Pr[A]$.
- **Variance $\text{Var}[X]$**: $E[X^2] - (E[X])^2$. For **independent** RVs, $\text{Var}[X+Y] = \text{Var}[X] + \text{Var}[Y]$.

Common Distributions

Name	Models	$E[X]$	$\text{Var}[X]$
Bernoulli(p)	Single success/fail	p	$p(1-p)$
Binomial(n, p)	# succ. in n trials	np	$\text{np}(1-p)$
Geometric(p)	Trials to 1st succ.	$\frac{1}{p}$	$\frac{1-p}{p^2}$
Poisson(λ)	# rare events	λ	λ
Negative Binomial(k, p)	# trials to k succs	$\frac{k}{p}$	$\frac{k(1-p)}{p^2}$

5.3 Concentration Inequalities

These inequalities formalize the idea that a random variable is likely to be close to its expectation. The more information we have (just mean, or mean+variance, or sum of independents), the tighter the bound.

Markov, Chebyshev, Chernoff

Inequality	Bound	Requirements
Markov	$\Pr[X \ge t] \le \frac{E[X]}{t}$	$X \ge 0$
Chebyshev	$\Pr[[X - \mu| \ge t] \le \frac{\text{Var}[X]}{t^2}$	Finite μ, Var
Chernoff	$\Pr[[X - \mu| \ge \delta\mu] \le 2e^{-\delta^2 \frac{\mu}{c}}$	Sum of indep. Bernoullis

TA Tip: Markov is weak (polynomial decay). Chebyshev is better (also polynomial). Chernoff is extremely powerful (exponential decay), making it the tool of choice for analyzing sums of many independent random events.

5.4 Randomized Algorithm Types

Monte Carlo vs. Las Vegas

- **Monte Carlo**: Always fast, probably correct. Think of primality testing. (A fast but possibly blurry photo.)
- **Las Vegas**: Always correct, probably fast. Think of Randomized QuickSort. (A slow but guaranteed sharp photo.)

Error Reduction (Amplification)

- **Las Vegas**: Repeat until you get an answer. $N = O(\frac{1}{\varepsilon}\log(\frac{1}{\delta}))$ repeats reduce failure probability to δ.
- **MC (Two-sided error, correct w.p. $\ge \frac{1}{2} + \varepsilon$)**: Repeat $N = O(\frac{1}{\varepsilon^2}\log(\frac{1}{\delta}))$ times and take a **majority vote**.

6 Algorithm Highlights

6.1 Randomized QuickSort

Expected Comparisons: $O(n\ln(n))$

The key to analyzing QuickSort is to focus not on the depth of recursion, but on the probability of any two elements being compared.

Analysis using Indicator Variables:
1. Let $a_1 < \ldots < a_n$ be the sorted elements.
2. Total comparisons $X = \sum_{i<j} X_{\{i,j\}}$.

3. **Key Insight:** a_i and a_j are compared $\iff$ one of them is the **first** pivot chosen from the set $\{a_i, ..., a_j\}$. If a pivot **between** them is chosen first, they are separated into different subproblems forever.
4. $\Pr[a_i, a_j \text{ compared}] = \frac{2}{j-i+1}$.
5. $E[X] = \sum_{i<j} \frac{2}{j-i+1} \approx 2n \ln n$.

6.2 Primality Testing

Miller-Rabin Primality Test

The industry standard for primality testing. It is a strong Monte Carlo test with one-sided error (it never calls a prime "composite"). It's based on a stronger property than Fermat's Little Theorem: if n is prime, the only solutions to $x^2 \equiv 1 \bmod n$ are $x \equiv +-1$. The algorithm hunts for "non-trivial" square roots of 1. **Intuition:** If n is composite, there are many "witnesses" a that will expose its compositeness. The test picks a random a and checks if it's a witness. Each test that passes increases our confidence that n is prime.

```
# Test n with base a
def miller_rabin(n, a):
    # Write n-1 = 2^k * d where d is odd
    d, k = n - 1, 0
    while d % 2 == 0:
        d //= 2; k += 1

    x = pow(a, d, n) # x = a^d mod n
    if x == 1 or x == n - 1:
        return "probably prime"

    # Check for non-trivial square roots of 1
    for _ in range(k - 1):
        x = pow(x, 2, n)
        if x == n - 1: return "probably prime"
        if x == 1: return "composite"

    return "composite"
```

Error Bound: For any composite n, $\Pr[\text{test passes}] \leq \frac{1}{4}$.

6.3 Global Min-Cut

Karger's Randomized Contraction Algorithm

A beautifully simple algorithm that leverages the fact that a random edge is far more likely to be a "normal" edge than one of the few edges in the minimum cut. By repeatedly contracting random edges, we hope to preserve the min-cut until the end.

```
def karger_min_cut(graph):
    while graph.num_vertices() > 2:
        u, v = graph.choose_random_edge()
        graph.contract(u, v) # Merge v into u
    return graph.num_edges()
```

Analysis: The probability of success in one run is low, $\geq \frac{1}{\binom{n}{2}}$. **Sanity Check:** If the min-cut has size k and the graph has m edges, the chance of contracting a min-cut edge first is $\frac{k}{m}$. This is non-zero, so failure is possible at every step. We amplify this by running it $O(n^2 \log n)$ times and taking the minimum cut found. A faster recursive version (Karger-Stein) runs in $O(n^2 \text{ polylog}(n))$.

6.4 Geometric Algorithms

Smallest Enclosing Disk

Clarkson's randomized algorithm ($O(n \log n)$ expected). The core strategy is adaptive sampling. We repeatedly sample a small set of points R, compute their enclosing disk $C(R)$, and find any outliers. These outliers are crucial; by adding them back to the pool (effectively increasing their "weight"), we make them more likely to be sampled in the future, which quickly forces the algorithm to consider the true boundary-defining points. **Intuition:** The true smallest disk is defined by 2 or 3 points. An outlier **must** be one of these defining points. By finding and focusing on outliers, we quickly zero in on the correct set of defining points.

Convex Hull in 2D ($O(n \log(n))$)

The Graham Scan (or Local Repair) algorithm is an optimal, elegant method. After sorting points by x-coordinate, it builds the hull incrementally. For each new point, it performs a "local repair" by backtracking and removing points from the current hull that would create a non-convex corner ("right turn" for the lower hull), maintaining the convexity invariant at each step.

Intuition: As we add points from left to right, we maintain a convex chain. Adding a new point p might make the last segment a "dent" (a right turn). We pop points off the hull until the path to p makes a "left turn", restoring convexity.

```
def graham_scan(points):
    points.sort() # Sort by x-coordinate
    hull = []
    # Build lower hull (checks for counter-clockwise
turns)
    for p in points:
        while len(hull) >= 2 and
cross_product(hull[-2], hull[-1], p) <= 0:
            hull.pop()
        hull.append(p)
    # (A similar loop for the upper hull follows)
    return hull
```

6.5 Final Topics

Floyd's Cycle-Finding (Tortoise & Hare)

Finds a cycle in a functional graph (like $x \to a[x]$) in $O(n)$ time and $O(1)$ space.

1. **Phase 1 (Meet):** `tortoise` moves 1 step, `hare` moves 2 steps. They meet inside the cycle.
2. **Phase 2 (Find Start):** Reset one pointer to the start. Move both 1 step at a time. They meet at the cycle's entrance.

TA Tip: The key insight for Phase 2 is that the distance from the start to the cycle entrance is equal to the distance from the meeting point to the cycle entrance (modulo cycle length).

Bloom Filter

A space-efficient probabilistic set that answers membership queries. It has **no false negatives** but allows **false positives**.

Use Case: Check if a URL has been visited by a web crawler without storing all URLs. A rare "yes" for a new URL is acceptable if space is tight.

Structure: A bit array of size m and k hash functions. To insert an item, set all k corresponding bits to 1. To query, check if all k bits are 1. The trade-off is space vs. false positive rate.

Colorful Path DP ($O(2^k km)$)

Finds if a path of length $k-1$ exists where all k vertices have distinct colors. This is the deterministic core of the randomized Long-Path algorithm.

State: $P_i(v)$ = set of all color-sets of size $i+1$ for colorful paths of length i ending at v.

Intuition: The state tracks, for each vertex v, all possible "palettes" of colors that can be formed by a colorful path of a certain length ending at v.

Recurrence: Build $P_i(v)$ by trying to extend paths from neighbors x in $P_{i-1}(x)$, but only if v's color is new to the path.

In detail: To compute $P_i(v)$, look at each neighbor u of v. For each color set C in $P_{i-1}(u)$, if $\text{color}(v)$ is not in C, then add the new set $C \cup \{\text{color}(v)\}$ to $P_i(v)$.