Contents

% Chapter 04 — Randomized Algorithms % CS161 Reader (Plotkin F10) %

1 Chapter 04 — Randomized Algorithms

Randomization is a *proof technique disguised as an implementation trick*. The key discipline is to state **exactly** what is random, what is adversarial, and what kind of guarantee you are proving.

Primary patterns: expectation (indicators + linearity), amplification (repeat/boost), induction (recurrences).

Executable witnesses: `cs161lab.algorithms.sorting.rand_quicksort`, `cs161lab.algorithms.sorting.qui` `cs161lab.algorithms.mincut.karger`.

1.1 4.1 Model and guarantee types

We analyze algorithms in the randomized RAM model: the algorithm may draw independent random bits at any step.

1.1.1 Las Vegas vs Monte Carlo

- **Las Vegas**: always correct; randomness affects running time (Randomized QuickSort).
- **Monte Carlo**: fixed running time; randomness affects correctness (Karger's min-cut).

1.1.2 Common guarantee traps

Expected-time bounds do **not** automatically imply "fast with high probability." When you need high probability, use explicit tail bounds or amplification (repetition plus a correctness filter or "best-of" selection).

1.2 4.2 Linearity of expectation and indicator variables

1.2.1 Theorem 4.1 (Linearity of expectation)

For any random variables $X_1, \ldots, X_m$ (independence not required),

$$E\Big[\sum_{i=1}^{m} X_i\Big] = \sum_{i=1}^{m} E[X_i].$$

Proof. Expand $E[\cdot]$ by definition (sum/integral) and swap summations/integrals. □

1.2.2 Lemma 4.2 (Indicators)

If I is an indicator (0/1) for an event E, then $E[I] = P(E)$.

Proof. $E[I] = 0 \cdot P(I = 0) + 1 \cdot P(I = 1) = P(I = 1) = P(E)$. □

This is the standard CS161 move: reduce expected counts to sums of event probabilities.

1.3 4.3 Randomized QuickSort

1.3.1 Algorithm

Given n distinct keys: 1. Pick a pivot uniformly at random from the subarray. 2. Partition around the pivot. 3. Recurse on both sides.

Executable witness: `cs161lab.algorithms.sorting.rand_quicksort`
Trace events: `pivot`, `partition`.

1.3.2 Theorem 4.3 (Expected comparisons)

Let C_n be the number of comparisons. Then

$$E[C_n] = 2(n+1)H_n - 4n = O(n \log n),$$

where $H_n = \sum_{k=1}^{n} \frac{1}{k}$ is the n-th harmonic number.

1.3.2.1 Proof (indicator-variable proof) Label keys in sorted order $z_1 < z_2 < \cdots < z_n$. For each $i < j$, define $I_{ij} = 1$ iff QuickSort compares z_i and z_j at some point. Then

$$C_n = \sum_{i<j} I_{ij} \quad \Rightarrow \quad E[C_n] = \sum_{i<j} E[I_{ij}] = \sum_{i<j} P(I_{ij} = 1).$$

Consider the set $S_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$. Keys z_i and z_j are compared **iff** the first pivot chosen from S_{ij} is one of the endpoints (z_i or z_j). If the first pivot in S_{ij} is some z_k with $i < k < j$, the recursion separates z_i and z_j forever.

The first pivot drawn from S_{ij} is uniformly distributed among its $|S_{ij}| = j - i + 1$ elements, so

$$P(I_{ij} = 1) = \frac{2}{j - i + 1}.$$

Therefore,

$$E[C_n] = \sum_{i<j} \frac{2}{j-i+1} = \sum_{\ell=2}^{n} \sum_{i=1}^{n-\ell+1} \frac{2}{\ell} = \sum_{\ell=2}^{n} \frac{2(n-\ell+1)}{\ell} = 2(n+1)(H_n - 1) - 2(n-1) = 2(n+1)H_n - 4n.$$

□

1.4 4.4 Randomized Selection (QuickSelect)

QuickSelect finds the k-th smallest element by partitioning around a random pivot and recursing on one side.

Executable witness: `cs161lab.algorithms.sorting.quickselect`
Trace events: `pivot`, `partition`, `shrink`, `found`.

1.4.1 Theorem 4.4 (Expected linear time)

The expected running time of QuickSelect is $O(n)$.

1.4.1.1 Proof (good pivot with constant probability)

Partitioning costs cn. A pivot is **good** if its rank lies in the middle half (between $n/4$ and $3n/4$), which happens with probability at least $1/2$. With a good pivot, the recursion continues on size at most $3n/4$. With a bad pivot, size is at most $n-1$. Hence,

$$E[T(n)] \le cn + \frac{1}{2}E[T(3n/4)] + \frac{1}{2}E[T(n-1)].$$

A standard induction shows this solves to $E[T(n)] \le an$ for a sufficiently large constant a. □

1.5 4.5 Amplification and the union bound

1.5.1 Theorem 4.5 (Union bound)

For events $E_1, \ldots, E_m$,

$$P\Big(\bigcup_{i=1}^{m} E_i\Big) \le \sum_{i=1}^{m} P(E_i).$$

1.5.2 Amplification lemma

If an independent trial succeeds with probability at least $p > 0$, repeating t times yields failure probability at most $(1-p)^t \le e^{-pt}$. To get failure at most δ, choose $t \ge \frac{1}{p}\ln(1/\delta)$.

1.6 4.6 Karger's randomized min-cut

Karger's algorithm is Monte Carlo: one run can fail, but it is fast, and repetition makes failure negligible.

Executable witness: `cs161lab.algorithms.mincut.karger`
Trace events: `contract`, `self_loop`, `trial`, `best`, `result`.

1.6.1 Algorithm (Random Contraction)

While more than 2 supernodes remain: 1. pick a uniformly random edge, 2. contract its endpoints, 3. delete self-loops. Return the cut induced by the final 2 supernodes.

1.6.2 Theorem 4.6 (One-run success probability)

For an n-vertex multigraph with min-cut value λ, one run returns a min-cut with probability at least $\frac{2}{n(n-1)}$.

1.6.2.1 Proof

Fix a particular minimum cut C of size λ. The run succeeds if no contraction ever contracts an edge crossing C. When there are k supernodes left and we have not crossed C, the cut still has exactly λ crossing edges. Also, the number of edges m_k satisfies $m_k \ge k\lambda/2$ because the minimum degree is at least λ (otherwise we'd have a smaller cut). Thus,

$$P(\text{avoid } C \text{ at stage } k) = 1 - \frac{\lambda}{m_k} \ge 1 - \frac{\lambda}{k\lambda/2} = 1 - \frac{2}{k}.$$

Multiply from $k = n$ down to 3:

$$P(\text{avoid } C \text{ for all contractions}) \ge \prod_{k=3}^{n}\left(1 - \frac{2}{k}\right) = \prod_{k=3}^{n}\frac{k-2}{k} = \frac{2}{n(n-1)}.$$

□

1.6.3 Corollary 4.7 (Repetition schedule)

After t independent runs and taking the best cut,

$$P(\text{all fail}) \le \left(1 - \frac{2}{n(n-1)}\right)^t \le \exp\left(-\frac{2t}{n(n-1)}\right).$$

To make failure $\le \delta$, it suffices to take $t = \Theta(n^2 \log(1/\delta))$.

1.7 4.7 What can go wrong

- Running a Monte Carlo algorithm without amplification (and then believing it).
- Assuming tail bounds from expectations without proof.
- Non-uniform pivot selection (implementation bias).
- PRNG seeding mistakes that destroy independence assumptions.

1.8 Appendix — Trace events as proof witnesses

- QuickSort: `pivot`, `partition` support the indicator proof narrative (who gets compared).
- QuickSelect: `shrink` is a termination witness; the good-pivot argument predicts frequent shrinkage.
- Karger: `contract` and `trial` let you empirically validate the repetition schedule.

% Chapter 05 — Graph Search and Topological Order % CS161 Reader (Plotkin F10) %

2 Chapter 05 — Graph Search and Topological Order

Graph algorithms are where *invariants* become unavoidable: without them DFS/BFS are just vibes.

Primary patterns: invariant, induction (structural recursion).

Executable witnesses: `cs161lab.algorithms.graphs.dfs`, `cs161lab.algorithms.graphs.toposort`.

2.1 5.1 Graph model and notation

Let $G = (V, E)$ be a directed graph with $|V| = n$ and $|E| = m$, represented with adjacency lists.

A directed graph is a **DAG** iff it has no directed cycle.

2.2 5.2 DFS (Depth-First Search)

2.2.1 Algorithm

Maintain a state for each vertex: white (unvisited), gray (active), black (finished). When exploring v: 1. mark gray 2. for each neighbor u: - if white: DFS(u) - if gray: found a back edge (cycle witness) 3. mark black; append v to a finishing list

Trace events: `enter`, `edge`, `back_edge`, `exit`.

2.2.2 Invariant 5.1 (Stack invariant)

At any time, gray vertices are exactly those on the current recursion stack. Any edge from a gray vertex to a gray vertex is to an ancestor.

Proof. Immediate from the definition of when vertices become gray/black. □

2.2.3 Lemma 5.2 (Back edge implies cycle)

If DFS discovers an edge (v, u) with u gray, then the graph contains a directed cycle.

Proof. Since u is on the recursion stack, there is a directed path from u to v using DFS tree edges. Adding (v, u) closes a cycle. □

2.3 5.3 Topological ordering

A topological ordering is an ordering $(v_1, \ldots, v_n)$ such that every edge points forward.

2.3.1 Theorem 5.3 (DFS finishing times yield topo order)

If G is a DAG, sorting vertices by **decreasing finishing time** produces a topological order.

Proof. Consider any edge (v, u). When DFS explores (v, u): - if u is white, DFS must finish u before finishing v; - if u is gray, that's a back edge implying a cycle, impossible in a DAG; - if u is black, then u already finished. Thus $f(v) > f(u)$ for all edges (v, u), so decreasing finishing time orders edges forward. □

Executable witness: `cs161lab.algorithms.graphs.toposort`.

2.4 5.4 Complexity

DFS runs in $O(n + m)$ time and uses $O(n)$ extra space (recursion stack + state).

2.5 Appendix — Trace events as proof witnesses

- `back_edge` is a concrete cycle witness.
- finishing order corresponds to the topological theorem's $f(v) > f(u)$ condition.

% Chapter 06 — Shortest Paths (Dijkstra & Bellman–Ford) % CS161 Reader (Plotkin F10) %

3 Chapter 06 — Shortest Paths

Shortest-path algorithms are relaxation machines: they maintain **upper bounds** and improve them until no improvement is possible. Dijkstra is fast but needs **nonnegative weights**; Bellman–Ford is slower but handles negative edges and detects negative cycles.

Primary patterns: relaxation, invariant.

Executable witnesses: `cs161lab.algorithms.sssp.dijkstra`, `cs161lab.algorithms.sssp.bellman_ford`.

3.1 6.1 Problem definition

Let $G = (V, E)$ be a directed graph with weights $w : E \to \mathbb{R}$ and a source s. Define

$$\delta(s, v) = \min_{p: s \rightsquigarrow v} w(p),$$

where $w(p)$ sums edge weights along p; if no path exists, $\delta(s, v) = +\infty$.

If there is a reachable negative cycle, shortest paths are ill-defined for vertices reachable from it (distance $-\infty$).

We maintain estimates $d[v]$ and predecessors $\pi[v]$.

3.2 6.2 Relaxation and the upper-bound invariant

3.2.1 Definition (Relaxation)

Relaxing edge (u, v) sets

$$d[v] \leftarrow \min\{d[v],\ d[u] + w(u, v)\}.$$

3.2.2 Invariant 6.1 (Upper-bound invariant)

At all times,

$$d[v] \geq \delta(s, v) \quad \forall v.$$

Proof. True at initialization. If $d[u] \geq \delta(s, u)$ then $d[u] + w(u, v) \geq \delta(s, u) + w(u, v) \geq \delta(s, v)$, so taking a min preserves the inequality. □

3.2.3 Lemma 6.2 (Witness paths via predecessors)

Whenever $d[v]$ is finite, predecessors reconstruct an $s \to v$ path of weight $d[v]$.

Proof. Set $\pi[v] = u$ exactly when relaxation improves $d[v]$ using u. Following predecessors yields a path. □

3.3 6.3 Dijkstra (nonnegative weights)

Assume $w(e) \geq 0$ for all edges.

3.3.1 Algorithm

Initialize $d[s] = 0$, others $+\infty$. Maintain finalized set $S = \emptyset$. Repeatedly extract $u \notin S$ with minimum $d[u]$, add to S, relax outgoing edges.

Trace events: `finalize`, `relax`.

3.3.2 Theorem 6.3 (Finalization lemma)

When Dijkstra finalizes a vertex u, then $d[u] = \delta(s, u)$.

Proof. Let S be finalized vertices before choosing u. Consider a shortest path from s to u and let (x, y) be the first edge crossing from S to $V \setminus S$. When x was finalized, (x, y) was relaxed, so

$$d[y] \leq d[x] + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y).$$

With Invariant 6.1, $d[y] = \delta(s, y)$. Since remaining edges on the path are nonnegative, $\delta(s, u) \geq \delta(s, y) = d[y]$. As u minimizes d outside S, $d[u] \leq d[y] \leq \delta(s, u)$. With Invariant 6.1, $d[u] \geq \delta(s, u)$, hence equality. □

3.3.3 Corollary 6.4

At termination, Dijkstra outputs correct distances for all vertices reachable from s.

3.3.4 Complexity

Using a binary heap: $O((n + m) \log n)$.

3.4 6.4 Bellman–Ford (negative edges allowed)

3.4.1 Algorithm

Initialize $d[s] = 0$, others $+\infty$. Repeat $n - 1$ passes: relax every edge. Do one extra pass; if any edge relaxes, report a reachable negative cycle.

Trace events: `iteration`, `relax`, `neg_cycle`.

3.4.2 Lemma 6.5 (k-edge optimality)

After k full passes, $d[v]$ is at most the shortest-path weight among all $s \to v$ paths using at most k edges.

Proof (induction). Each pass allows paths that extend a k-edge best path by one edge. □

3.4.3 Theorem 6.6 (Correctness without negative cycles)

If there is no reachable negative cycle, then after $n - 1$ passes, $d[v] = \delta(s, v)$ for all v.

Proof. A shortest path can be chosen simple, hence has at most $n - 1$ edges. Apply Lemma 6.5 with $k = n - 1$ and combine with Invariant 6.1. □

3.4.4 Theorem 6.7 (Negative-cycle detection)

If an edge can still be relaxed after $n - 1$ passes, then a reachable negative cycle exists.

Proof. A strict improvement implies an improving walk of length at least n edges, hence repeats a vertex; the repeated segment is a negative cycle. □

3.5 Appendix — Trace events as proof witnesses

- Dijkstra `finalize` events correspond to Theorem 6.3 checkpoints.
- Bellman–Ford `iteration(k)` tracks the Lemma 6.5 induction parameter.
- `neg_cycle` is a concrete witness of a violated "no negative cycle" assumption.