CHECKLIST

Algorithms Checklist: Verify Correctness, Performance, and Safety Before Production

A practical algorithms checklist for technical reviews. Verify problem fit, correctness, complexity, edge cases, security, observability, and production readiness before release.

Programming - July 03, 2026

Purpose

Algorithms fail in production for predictable reasons: the wrong problem model, hidden edge cases, non-obvious complexity blowups, unsafe input assumptions, or insufficient operational testing. An algorithms checklist gives reviewers a repeatable way to confirm that an implementation is correct, efficient enough for its workload, and safe to run under real conditions.

Use this checklist to decide whether an algorithm is ready for production use, what evidence to collect during review, and which gaps must be closed before release. It is written for engineers who need practical validation rather than theory.

How to use this checklist

Review the checklist in order, from problem definition through operational readiness. For each phase, confirm the listed checks, capture the required evidence, and compare the result against the acceptance criteria. If a phase fails, stop and fix the issue before moving on.

Treat this as a production review artifact. It works best when one owner is accountable for algorithm design, one for implementation, and one for validation. If the algorithm is part of a broader release process, align this review with a readiness method such as the NET Checklist: Operational Readiness Review for Production Use when the algorithm is embedded in a service, or use the Learning Checklist for AI and Machine Learning Projects if the algorithm is part of an ML pipeline.

Phase 1: Problem definition and scope

The goal of this phase is to confirm that the algorithm solves the right problem, under the right constraints, with the right inputs and outputs.

Checklist items

- ? Confirm the problem statement is specific, measurable, and bounded.
- ? Review the input domain, including data types, ranges, ordering, nullability, and maximum size.
- ? Validate the expected output format, determinism requirements, and tie-breaking rules.
- ? Document performance constraints such as latency, throughput, memory usage, and batch size.
- ? Assign ownership for algorithm logic, data assumptions, and runtime validation.
- ? Review whether the algorithm must be exact, approximate, stable, or idempotent.

Evidence to capture

- Written problem statement and acceptance criteria
- Input/output contract or interface specification
- Known constraints from product, data, or platform owners
- Estimate of workload volume and peak behavior

Acceptance criteria

- The problem statement is unambiguous and testable.
- All required input and output constraints are documented.
- The algorithm?s required behavior is clear under normal and abnormal inputs.
- The implementation scope is narrow enough to validate independently.

Owner and review cadence

Owner: algorithm designer or service owner. Review at design time and again before implementation freeze.

Common mistakes

- Treating a vague business request as a complete algorithm specification
- Failing to define how duplicates, missing values, or unordered inputs are handled
- Ignoring memory limits or expected peak input size

Phase 2: Algorithm choice and correctness model

The goal of this phase is to validate that the chosen approach is suitable and that its correctness can be reasoned about, not merely observed on a few examples.

Checklist items

- ? Confirm the selected algorithm matches the problem class and constraints.

- ? Review whether the approach depends on sorted input, graph structure, recursion depth, or probabilistic assumptions.
- ? Validate that invariants are documented and preserved at each step.
- ? Document the termination condition and prove or explain why the algorithm cannot loop indefinitely.
- ? Review whether the algorithm produces the same result for the same input when determinism is required.
- ? Assign a reviewer to challenge the algorithm with counterexamples and boundary cases.

Evidence to capture

- Design notes, proof sketch, or correctness argument
- Invariants and termination conditions
- Counterexamples considered during review
- Dependency assumptions, including ordering and data structure requirements

Acceptance criteria

- The correctness argument is understandable and internally consistent.
- Required assumptions are explicit and verified elsewhere in the system.
- No hidden dependency undermines the algorithm?s correctness.
- Failure modes are identified where correctness cannot be guaranteed.

Owner and review cadence

Owner: algorithm designer. Review during design review and again after peer review.

Common mistakes

- Choosing a familiar algorithm without checking whether its assumptions hold
- Relying on empirical success instead of a correctness argument
- Ignoring the difference between functional correctness and acceptable approximation

Phase 3: Complexity and resource analysis

The goal of this phase is to confirm that the algorithm is efficient enough for the expected workload and that resource use is bounded.

Checklist items

- ? Review the time complexity for best, average, and worst cases.
- ? Validate the space complexity, including auxiliary structures, recursion stack, and buffering.
- ? Confirm that complexity claims match the actual implementation path, not just the intended design.
- ? Test whether pathological inputs trigger unacceptable behavior.
- ? Document any trade-offs between CPU, memory, and I/O.
- ? Assign a performance owner to verify the algorithm against production-like data.

Evidence to capture

- Complexity analysis with assumptions
- Memory profile or sizing estimate
- Performance test results on representative and worst-case inputs
- Notes on pathological or adversarial input patterns

Acceptance criteria

- Complexity is compatible with expected scale and peak load.
- Resource consumption stays within platform limits with margin.
- Worst-case behavior is known and acceptable.
- Any expensive operation is justified and bounded.

Owner and review cadence

Owner: performance engineer or service owner. Review before load testing and before production approval.

Common mistakes

- Assuming average-case complexity is sufficient without checking worst-case behavior
- Underestimating the cost of recursion, copying, or repeated scanning
- Ignoring hidden costs from serialization, allocation, or cache misses

Phase 4: Edge cases and failure handling

The goal of this phase is to verify that the algorithm behaves predictably when inputs are incomplete, malformed, empty, extreme, or adversarial.

Checklist items

- ? Validate behavior for empty, single-item, duplicate, and maximum-size inputs.
- ? Review handling for null, missing, NaN, overflow, underflow, and out-of-range values where applicable.
- ? Test boundary transitions such as off-by-one limits and exact threshold values.
- ? Confirm the algorithm fails safely or returns a defined result for invalid input.

- ? Document fallback behavior, retries, or error propagation rules.
- ? Assign negative test cases that intentionally break assumptions.

Evidence to capture

- Edge-case test matrix
- Validation and sanitization rules
- Error handling examples or logs
- Overflow, underflow, and boundary test results

Acceptance criteria

- Every documented edge case has an expected result.
- Invalid input produces a safe and explicit failure mode.
- No edge case produces undefined behavior or silent corruption.
- Input validation occurs before dangerous operations whenever possible.

Owner and review cadence

Owner: implementation owner. Review during unit test completion and before integration testing.

Common mistakes

- Testing only representative ?happy path? data
- Allowing silent truncation, overflow, or implicit coercion
- Assuming upstream systems will always sanitize input correctly

Phase 5: Implementation quality and code review

The goal of this phase is to ensure the code matches the design, is readable enough to maintain, and does not introduce avoidable defects.

Checklist items

- ? Confirm the implementation matches the documented algorithm step for step.
- ? Review naming, structure, and comments for maintainability.
- ? Validate that helper functions, recursion, and loops are bounded and purposeful.
- ? Test whether refactoring changed behavior or complexity.
- ? Document any language-specific pitfalls such as integer overflow, iterator invalidation, or reference aliasing.
- ? Assign a peer reviewer who did not author the original code.

Evidence to capture

- Code review record and approval status
- Static analysis output where available
- Comparison between design notes and implementation
- Notes on language-specific hazards

Acceptance criteria

- The code follows the reviewed design without unapproved deviations.
- Readability is sufficient for future maintenance and incident response.
- Static analysis or code review findings are resolved or explicitly accepted.

- No implementation shortcut weakens correctness or safety.

Owner and review cadence

Owner: implementation owner with peer review. Review at pull request time and after any substantial refactor.

Common mistakes

- Optimizing before correctness is proven
- Hiding algorithm logic inside unclear helper layers
- Missing language-specific edge cases that change runtime behavior

Phase 6: Verification and test coverage

The goal of this phase is to confirm the algorithm behaves correctly across unit, property-based, integration, and regression tests.

Checklist items

- ? Confirm unit tests cover normal paths, edge cases, and failure paths.
- ? Review property-based or randomized tests for invariant preservation where suitable.
- ? Validate regression tests for previously observed defects.
- ? Test integration points where algorithm output feeds other systems or calculations.
- ? Document test data provenance and any synthetic data assumptions.
- ? Assign a reviewer to confirm that tests fail when the algorithm is intentionally broken.

Evidence to capture

- Test plan and coverage summary
- Failing and passing test examples
- Regression history and defect references
- Integration test results with downstream consumers

Acceptance criteria

- Test coverage is meaningful, not just high in number.
- Critical invariants are tested directly.
- Regression tests exist for known failure modes.
- Test results are reproducible and traceable to the reviewed version.

Owner and review cadence

Owner: QA engineer, developer, or validation owner. Review per release candidate and after any algorithm change.

Common mistakes

- Relying on a single sample dataset
- Treating line coverage as proof of correctness
- Skipping negative tests because the algorithm ?works in practice?

Phase 7: Security and abuse resistance

The goal of this phase is to verify that the algorithm does not become a denial-of-service vector, data exposure path, or trust boundary failure.

Checklist items

- ? Review whether untrusted input can cause excessive CPU, memory, or recursion usage.
- ? Validate that the algorithm does not leak sensitive data through logs, errors, or side channels.
- ? Confirm that access control checks occur before any sensitive operation the algorithm performs.
- ? Test whether attacker-controlled input can trigger worst-case complexity intentionally.
- ? Document any cryptographic, hashing, or randomness dependencies and verify they are appropriate for the use case.
- ? Assign security review for algorithms used in authentication, authorization, deduplication, ranking, or anomaly detection.

Evidence to capture

- Threat model notes or abuse-case analysis
- Security review findings
- Resource exhaustion test results
- Logging and error-handling samples

Acceptance criteria

- The algorithm fails safely under malicious or malformed input.
- No sensitive state is exposed through operational output.
- Worst-case resource usage is understood and controlled.
- Security-sensitive algorithms receive explicit review and approval.

Owner and review cadence

Owner: security reviewer with implementation owner. Review before production exposure and after threat model updates.

Common mistakes

- Ignoring algorithmic complexity as a security issue
- Logging input values that contain secrets or identifiers
- Using weak or unsuitable randomness where predictability matters

Phase 8: Operational readiness and rollout

The goal of this phase is to ensure the algorithm can be observed, rolled back, and safely operated after deployment.

Checklist items

- ? Confirm the implementation exposes meaningful metrics for correctness, latency, error rate, and saturation.
- ? Review whether logs and traces are sufficient to diagnose algorithm-specific failures.
- ? Validate that feature flags, configuration guards, or rollout controls exist where needed.
- ? Test rollback behavior if the algorithm changes output or workload characteristics.
- ? Document versioning for model parameters, thresholds, lookup tables, or rulesets if applicable.
- ? Assign an operational owner for post-deployment monitoring and incident response.

Evidence to capture

- Monitoring dashboard or metric definitions

- Rollout and rollback procedure
- Change log for thresholds, parameters, or rule updates
- On-call ownership and escalation path

Acceptance criteria

- The algorithm is observable in production terms, not just test terms.
- Rollback is practical and verified.
- Configuration and version changes are tracked.
- Operational ownership is explicit.

Owner and review cadence

Owner: service owner or operations lead. Review before staged rollout and after production incident review.

Common mistakes

- Deploying an algorithm without monitoring for its failure modes
- Allowing untracked parameter changes to alter behavior
- Assuming rollback is possible without validating state compatibility

Simple readiness scoring

Use this lightweight scoring method to summarize review outcomes. It is not a substitute for evidence, but it helps decide whether release can proceed.

Score each phase

- 0 = Not started or no evidence
- 1 = Partial coverage, significant gaps remain
- 2 = Mostly complete, minor gaps remain
- 3 = Complete with satisfactory evidence and no open blockers

Interpreting the score

- 18?24 points: Ready for production review or limited rollout
- 12?17 points: Conditional readiness; fix gaps before wider release
- 0?11 points: Not ready; do not release

Pass/fail rule

A single failure in correctness, security, or rollback safety is a fail, even if the total score is high. If the algorithm is mathematically elegant but operationally unsafe, it is still not ready.

Review checklist summary

Before approving production use, confirm these outcomes together:

- The problem definition is precise and scoped.
- The chosen algorithm is correct for the assumptions it depends on.
- Complexity is acceptable for the real workload.
- Edge cases and invalid inputs are covered.
- The code matches the design and passes meaningful tests.
- Security and abuse cases are considered.
- Monitoring, rollback, and ownership are in place.

If any of these are unresolved, keep the algorithm out of production until the gap is closed and revalidated. A strong algorithm is not just one that works in a notebook or unit test; it is one that remains correct, bounded, and supportable when real traffic, real failures, and real attackers arrive.

Use this guidance together with learning implementation roadmap checklist to connect the workflow with related operational context already available on the site.

Related guides in this cluster

- How to Measure Algorithms Maturity
- Algorithms Reporting Template FAQ: Build a Reliable Output Record

Related guides in this cluster

- Algorithms Implementation Roadmap Checklist
- How to Optimize Dijkstra’s Algorithm for Shortest Paths

Related guides in this cluster

- Dijkstra?s Algorithm for Fast Shortest Path Routing Optimization

Related guides in this cluster

- Efficient Dijkstra?s Algorithm for Large-Scale Network Routing

Related guides in this cluster

- Dijkstra’s Algorithm Explained for Network Path Optimization

- Space-Efficient Dijkstra Variants for Large-Scale Graphs

Related guides in this cluster

- A* Search Algorithm for Optimal Pathfinding in Weighted Graphs

Related guides in this cluster

- Dijkstra’s Algorithm for Fastest Path Routing in Networks

Related guides in this cluster

- Branch and Bound Algorithm for Optimal Resource Scheduling

Related guides in this cluster

- How to Implement Dijkstra’s Algorithm for Shortest Paths

Related guides in this cluster

- A* Search Optimization for Real-Time Pathfinding Systems

Related guides in this cluster

- Dijkstra?s Algorithm for Secure Network Path Optimization
- Fastest Path Algorithms for Dynamic Network Routing Optimization

Related guides in this cluster

- What Is Programming? Algorithms Insights for Technical Teams

Related guides in this cluster

- Efficient Dijkstra Variants for Weighted Graph Shortest Paths

Related guides in this cluster

- Dijkstra’s Algorithm for Shortest Path in Weighted Graphs