SAMPLE

Building the Algorithms of Tomorrow

Building the Proven Procedures and Policies of Tomorrow

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-the-algorithms-of-tomorrow-ebook-pdf-1465661508

Gumroad

Ko-fi

ElevenReader

Payhip

Contents

Introduction

Chapter 1: What an Algorithm Precisely Is

Chapter 2: The Toolbox

Chapter 3: Optimization

Chapter 4: Approximation and Heuristics

Chapter 5: Randomization

Chapter 6: Learning

Chapter 7: Matching and Markets

Chapter 8: Scheduling and Fairness

Chapter 9: The Failure Modes

Chapter 10: The Operating Discipline

This free sample contains the Introduction and Chapter 1. The complete book contains the full contents above.

Introduction

Civilization runs on recipes. The oldest ones humans wrote down were not about food: they were about counting — procedures, executed step by step, that turned a ledger into a sum, a boundary into a field's measure, a sky's motion into a calendar. The word the book is built around arrived later, attached to a mathematician's name, and detached from him almost immediately: the algorithm — a finite, unambiguous, terminating procedure that transforms an input into an output, the smallest unit of reliable thought ever specified. Everything this series has built — the models, the factories, the grids, the machine's ethics and its hallucinations — rests on this layer, and this volume descends to it: the algorithms of tomorrow, which are, mostly, the algorithms of yesterday — sorting, searching, matching, routing, optimizing — now running the world at a scale and in places their inventors never priced.

The structural claim of the book, stated plainly: the algorithm is the purest infrastructure there is. A bridge specifies a load; an algorithm specifies a behavior — and unlike the bridge, the specification can be proven, the failure modes enumerated, the costs bounded in symbols before a single machine runs. That purity is also the layer's danger, and the book's second theme: an algorithm is a decision made once, for every case it will ever see, by an author who could not see the cases — a frozen policy, executed tirelessly, invisibly, and at a scale that turns small design choices into the shape of populations' lives. The queue that orders the emergency room, the matching that pairs students to schools, the ranking that decides what a city sees, the feed that decides what it believes: none of these are neutral math, and none of their authors are exempt from the accountability this series demands of every infrastructure.

Ten chapters, in the series' standing shape. The first three build the foundation: what an algorithm precisely is (and the properties — correctness, termination, complexity — that make one trustworthy), the canonical toolbox every system since the computing era actually runs on, and optimization's mathematics — what "best" means, and why finding it is a different discipline than defining it. Chapters four and five treat the disciplines that make hard problems tractable: approximation and heuristics, the art of good-enough where exact is impossible; and randomization, probability as an engineering material. Chapters six and seven follow the algorithmic turn outward: learning — the shift from rules written to behavior trained, with its own physics; and matching and markets — the allocation algorithms that quietly run the economy's assignments.

Chapter eight treats scheduling and fairness — time as a resource, and the

competing definitions of distributing it. Chapter nine is the failure-mode catalog: bias, feedback loops, adversarial inputs, the complexity cliffs, the consequences no objective function contained. And the tenth chapter assembles the operating discipline — how an organization chooses, audits, and lives with its algorithms — and closes, as every volume does, with the questions.

A note on method, consistent with the series: evergreen and structural, no vendors, no benchmark numbers that rot, no mathematics beyond the reader's intuition — the symbols this book trades in are trade-offs, and each chapter ends with a closing argument, a checklist, and a diagnostic, because an algorithm that cannot be inspected is a policy nobody signed.

Chapter 1: What an Algorithm Precisely Is

The word is loose in public speech — any computational anything — and the discipline begins where this book's chapters always begin: with the definition tightened until it does work. An algorithm is a finite procedure: a bounded list of steps, each unambiguous to its executor, that begins with an input, ends after finitely many steps, and produces an output that stands in a specified relation to the input. The definition's five load-bearing properties, each worth a sentence. Finiteness: the procedure is bounded — it ends, on every input it accepts; the program that sometimes runs forever is not an algorithm, it is a hazard wearing one.

Unambiguity: each step means one thing to its executor — whether the executor is a machine or a careful human, there is no interpretive gap between the author's intent and the behavior; this is what "formal" means, and why the algorithm is the smallest unit of reliable thought. Input and output: the procedure transforms one into the other, and the transformation is the algorithm's entire identity — the same input, the same output, every time (or with the randomness priced, per Chapter 5). Effectiveness: each step is doable — the pencil-and-paper tradition's requirement, kept because it separates the procedure that can run from the poetry that cannot.

And determinism of specification: what is specified is the behavior, not the implementation — the same algorithm in two languages, on two machines, at two scales, is the same algorithm, which is why this book can discuss the infrastructure of tomorrow without naming a single vendor of today.

The reader should fix the boundary cases now, because the public vocabulary blurs them and the blurs matter downstream. A program is not an algorithm: a program is an expression — one implementation, possibly containing several algorithms, possibly containing none of the good ones; the algorithm is the procedure the program implements. A model is not an algorithm either: a trained model (this series' earlier volumes built the distinction) is a frozen artifact — a set of parameters — and the algorithm is the procedure that trained it, the procedure that runs it, or the procedure that serves it; conflating the artifact with the procedure is how the accountability questions of Chapter 9 get lost, because the procedure has an author and the artifact's behavior escapes attribution.

And a heuristic is not — yet — the chapter's subject comes later, but the boundary deserves its place here: the algorithm guarantees its output's relation to its input; the heuristic promises only its effort. Both are infrastructure; they are different kinds, and the book will price the difference

honestly in Chapter 4.

What makes an algorithm trustworthy is not that it works — everything works until it doesn't — but that its working is established in one of the discipline's three currencies, and the reader should learn the currencies in order of strength. Correctness: the procedure provably produces outputs in the specified relation to its inputs, for every input — the proof tradition's gold, established by induction, by invariant, by construction; the sorting procedure that is proven to sort, always, for every list, is civilization's cheapest reliable component, and the reader should understand why the proof matters: it holds for the inputs the author never imagined, which is the entire value of proof over testing.

Termination: the procedure provably ends — the halting analysis that shows the loop's measure decreases, the recursion that bottoms out; termination is the property the public never thinks about and the engineer always does, because the algorithm that does not end is not slow, it is a lock. And complexity: the resource profile — time and space, as functions of input size — that prices the procedure before it ever runs. The complexity statement is the algorithm's economics, and the discipline's deepest habit: the difference between the procedure that scales linearly with the data and the one that scales quadratically is not a matter of polish; it is the difference between the system that survives the decade's growth and the one that dies quietly in its third year, at ten times the data, discovered by pager.

The three currencies compose into the chapter's central lesson: an algorithm is a guarantee, and the guarantee is written before the machine. This is the layer's purity, from the introduction, made precise: a bridge's load rating is established by testing and inspection, at cost, per bridge; an algorithm's correctness is established once, by proof, for every instance that will ever exist — including the instances its author will never see, in the systems of tomorrow, by the engineers of tomorrow, who will never know they are standing on it. That is the infrastructure claim in its strongest form: the sorting procedures proven in the discipline's youth still run, unmodified, under everything the reader touched today — and the chapter closes by planting the book's running theme: guarantees that strong are also policies that frozen, and the chapters that follow follow both sides of that.

The chapter's last section turns the lens from mathematics to governance, because the definition has a consequence the discipline spends little time on and this book must: an algorithm is a decision made once, for every case. The procedure's author — the person who chose the objective, the ordering, the tie-break — legislated: the choice of which candidate the matching prefers,

which task the scheduler serves first, which node the route visits next, is a rule of behavior, applied without exception, at machine speed, to every instance the future delivers. The legislation is usually invisible — the author was solving a problem, not writing a law — but the effect is legislative in every way that matters: it binds parties who never agreed, at a scale no parliament reaches, and it persists after the author has left, the objective function's values outliving their intent like every law's letter does.

The reader who has finished this series' ethics volume will recognize the pattern arriving at the mathematical layer: policy is wherever a choice is made for others, and the algorithm is the purest, least examined form of it.

The chapter therefore closes with the position the book will hold throughout: the algorithm deserves both of its reputations. As mathematics: the most inspectable, provable, trustable artifact civilization produces, and the reader should be slower than the headlines to distrust it — the proof is real, the guarantee is real, and everything downstream in this book leans on both. As policy: among the least accountable forms of decision-making in institutional life — rarely signed, rarely reviewed, its values rarely defended in front of the populations it governs — and the reader should be quicker than the discipline to demand its accountability. The chapters that follow honor both: the toolbox (Chapter 2) is presented with the respect its guarantees deserve; the failure catalog (Chapter 9) is presented with the severity its unexamined policies deserve; and the operating discipline (Chapter 10) exists to reconcile the two, which is the book's whole task.

- [ ] The organization's vocabulary distinguishes algorithms, programs, models, and heuristics, with the guarantee-difference stated.

- [ ] Every load-bearing algorithm has its correctness/termination/complexity currency identified and documented. - [ ] The policy embedded in each algorithm's objective, ordering, and tie-breaks is named, owned, and signed. - [ ] No system treats a model's behavior as attributable without locating the procedures that trained and serve it.

The layer has a history worth one honest section, because the history explains the layer's peculiar character: it is the oldest engineering discipline that behaves like the youngest. The recipes are ancient — the earliest surviving mathematics is procedure: step-by-step methods for dividing fields, compounding interest, resolving disputes by calculation; the Babylonian scribes wrote algorithms, unambiguously, for executors they would never meet. The word itself traveled a strange road — from a mathematician's name, attached to decimal procedures, migrating through centuries of popular arithmetic books — and the modern era's turn was not the invention of the

procedure but the formalization of it: the recognition that a procedure could be an object of mathematics, examined, proven, and classified — a thing one could know things about, rather than merely run.

The formalization gave the discipline its central results: the discovery that some well-specified problems cannot be solved by any procedure at all (the unsolvability results, the field's genuine limits, proven like any theorem); the discovery of the complexity classes (the tractability boundary of Chapter 3); and the universal executor result — the proof that one fixed machine can run any procedure whatsoever, given its description, which is the founding fact of the entire computing era and the reason software is a thing at all.

The history's lesson for the program is the one this book keeps finding: the formal layer is only as accountable as the institutional habits around it. The procedures were proven in the discipline's youth precisely because the machines were slow — correctness was affordable and compute was not — and the eras since have inverted the economics: compute became cheap, correctness-expense became optional, and a great deal of the modern world runs on unproven, unexamined, thoroughly hyped procedure. The reader should hold the arc without nostalgia: the proofs exist, the proof skills exist, and the choice to use them is an institutional one, made per system, at the objective-review meeting this book keeps convening; the discipline's youth shows what the discipline is capable of, and the present shows what it usually sells.

The diagnostic, then as now, is the currency question: for any load-bearing procedure, is there a proof, and if not, is the absence chosen (and priced) or drifted into? The answer distinguishes the two computing eras more cleanly than any hardware generation does.

The chapter's second section teaches the reader how correctness is actually established — not as ceremony but as craft, because the technique is one of the most beautiful tools civilization has, and the reader who understands it can read a proof the way this series' readers read a blueprint. The central device is the invariant: a statement about the procedure's state that is true at the loop's start, preserved by every step of the loop, and — when the loop ends — delivers the conclusion. The sorting procedure's invariant: at every step, the elements already processed are in order; the step preserves it (each new element is inserted where order survives); the loop's end delivers it (all elements processed, all in order — the procedure sorts, proven, for every list).

The reader should see the invariant's real power: it does not test the procedure on instances, it covers every instance the future will ever deliver — the proof is a guarantee about infinity, purchased with a finite argument, and

there is nothing else in the engineering repertoire that buys anything like it. The technique's siblings complete the craft: the termination measures (the quantity that decreases each step and cannot decrease forever — the loop must end, and the reader should register that this is how "the algorithm terminates" is established: not by running it, but by finding the clock inside it); the contradiction arguments (the assumption that a wrong answer exists, walked to an impossibility — the standard shape of optimality proofs, including the proofs that no procedure can do better); and the construction proofs (the algorithm is the proof: the artifact and its correctness certificate in one).

The section's practical yield is the review craft: the invariant is the code review's missing instrument, and the mature program asks for it by name. The pull request that says "fixed the bug" is a report; the one that says "the loop's invariant was wrong — it held only for nonempty inputs; here is the corrected invariant and the case that failed it" is an engineering record, and the difference compounds exactly like every other documentation debt this series has priced. The invariant discipline also disciplines the tester: the test suite exercises instances; the invariant covers the space — and the program that has both knows things the program with only tests cannot (the boundary cases the suite missed, which the invariant names), and the program with only the proof knows nothing about its implementation's accidents (the proof covers the algorithm, the tests cover the code, and the gap between them is where the deployed systems actually fail — which is why the book's checklists demand both: the proof for the procedure, the tests for the code, and the boundary between them stated honestly).

The chapter closes by widening the definition's quietest term: the executor. The classical definition is executor-neutral — a finite, unambiguous procedure for any agent that can follow steps — and the neutrality is not a technicality; it is the definition's deepest social fact, because humans execute algorithms too, and the reader lives inside several. The checklist is an algorithm — the surgical safety list, the pilot's pre-flight, the emergency response card: finite, unambiguous, terminating, and proven in the field's own way (the safety traditions' results are as real as the mathematical ones, measured in outcomes); the bureaucratic procedure is an algorithm — the application process, the escalation path, the grievance ladder — executed by institutions at human speed; and the reader should notice the two executor classes' mirror-image failure modes: the machine executes perfectly and its procedures are under-specified (the specification's values never written down), while the human executes imperfectly and its procedures are over-specified (the steps written for the machine's fidelity and executed by the tired, the hurried, and the distracted — the checklist that gets skipped exactly when it matters, which

is the human executor's characteristic cliff, and the safety tradition's response is the honest one: design the procedure for its executor's real properties, as this series designed every layer for its real substrate).

The two executor classes are converging, and the section's close prices the convergence. The machine-executed procedures are acquiring human-facing languages (the natural-language instruction, the model-driven workflow — procedures specified in prose and executed by machines that interpret it), which imports the human executor's ambiguity problem into the machine's reliability: the prose procedure run by the interpreting machine is neither the mathematical algorithm nor the human checklist, and its guarantee chain runs through the interpreter — Chapter 6's measurement currency, arriving at the specification layer. And the human-executed procedures are acquiring machine verification (the checklist instrumented, the compliance check automated, the escalation path observed) — which imports the mathematical layer's inspectability into the institution's recipes: the procedure whose execution is provable, in the audit sense, for the first time.

The reader who holds both directions can see the decade's executor landscape whole: the recipes are merging, the fidelity requirements are diverging, and the program's question for any procedure is the same in both worlds — who executes, what fidelity, and what happens when the executor errs? The answer is the procedure's real specification, whatever its form says.

The chapter's fourth section prices the guarantee's own limit, which the proof tradition states plainly and the deployment world often forgets: correctness is relative to a specification, and the specification is written by people. The proven algorithm does exactly what the spec says — sorts the records, meets the deadlines, allocates the shares — and if the spec itself encodes the wrong order, the wrong deadlines, or the wrong shares, the proof certifies the wrongness with the same rigor it would certify the rightness; the guarantee's strength amplifies whatever it is pointed at, which is the book's oldest theme arriving at its sharpest form: the proof is only as good as the specification, and the specification is a policy document.

The field's honest response to this limit is not despair but separation of duties, and the mature program institutionalizes it: the specification's author is named and its review is a human act (the objective review of the operating chapter, upstream of every proof), while the proof's establishment is a mechanical act (checked by machinery, trusted by mathematics, never asked for judgment). The two acts never substitute for each other: the proof that certifies an unexamined specification is diligence performed in the wrong place, and the specification that relies on the proof for its values has delegated the und

delegable — the classic confusion, found in incident reviews with depressing regularity, usually phrased as "but the algorithm was correct," which was true, and beside the point.

The section's deliverable is the specification audit, run before the proofs are trusted: for any load-bearing algorithm, the spec is read as a policy document and asked the policy questions — whose interests does this ordering serve, what does this constraint encode, what world was this requirement made for (the legacy register's question, arriving at the intake), and who signed it? The reader should run the audit's logic on the chapter's own examples to feel its reach: the sorting procedure proven correct was proven about a comparison function — and the comparison function (which candidate ranks first) is the entire policy; the scheduler proven to meet deadlines was proven about the deadlines — and the deadlines are the institution's values; the proof is the machinery's fidelity to the spec, and the audit is the civilization's fidelity to its own intent, and both are needed, always, in that order.

The chapter's last section widens the definition once more, to the case the classical theory set aside and the deployed world cannot: the distributed procedure — steps executed not by one neutral executor but by several parties, each with its own state, its own failures, and its own interests. The reader has met the infrastructure (the consensus machinery of the randomization chapter, the allocation mechanisms of the matching chapter), and the section's purpose is to name what changes at the definitional level, because the change is profound: when the executor has interests, the procedure's steps are no longer merely followed — they are chosen, strategically, and the algorithm's design must therefore make the desired behavior the safe behavior for every party: the mechanism-design turn, in which the procedure's rules and the parties' incentives are designed together, so that the truthful, cooperative play is the dominant strategy and the mechanism holds even though no one is neutral.

The reader should appreciate the resulting discipline's character: it is game theory and algorithm design fused — the same proofs, the same guarantees, but the guarantee is now about behavior under incentives, not merely execution of steps — and its results are among the field's most consequential (the auction chapter's truthful bidding, the matching chapter's stable assignments, the consensus chapter's agreement despite faulty members, the cryptographic layer's verification machinery: all of them procedures with interests, proven to hold anyway, which is one of the deepest things the mathematical tradition has ever built).

The section's close connects the two definition widenings the chapter has

made — the human executor and the interested executor — into the reader's single reading frame: a deployed algorithm in the real world is almost always both, a procedure executed by parties with interests at stake, at some layer of the stack: the employees executing the process, the departments feeding the pipeline, the platforms invoking each other's services, the populations living in the queues. The classical definition — the neutral executor, the trusted steps — is the pure core the theory stands on, and this book has kept returning to it because the proofs are real; the deployed reality wraps that core in interests, and the program's discipline is to design and audit the wrap as deliberately as the core: the specification's authors signed (the previous section), the parties' incentives designed (this section), and the executor's fidelity assumed honestly — the three audits, run together, are what the operating discipline of this book finally assembles.

An algorithm is a finite, unambiguous, terminating procedure with a guaranteed relation between input and output — the smallest unit of reliable thought, and the only infrastructure whose safety can be proven before it is built. The same purity makes it the purest form of policy: a rule authored once, executed forever, for every case the author never saw. Trust the mathematics; distrust the unaudited values; and demand of every algorithm what this book will demand of each one in turn: its guarantee, its cost, and its author's signature.

End of free sample. The full book is available at putrabooks.github.io

ABOUT THE AUTHOR

Aiman JoeHanz writes the Building of Tomorrow series, a volume-per-technology account of the systems defining this decade -- from the algorithms and their ethics to the cities, medicine, and stations they will build. Each book is a structural treatment: the parts, the failure modes, the economics, and the program. In this volume the author descends to the layer everything else runs on -- the algorithm itself -- and treats it the way the series treats every infrastructure: as engineering, with proofs, policies, failure modes, and a discipline a builder can actually run.