Course Orientation and Algorithmic Thinking

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why algorithmics matters

An algorithm is a finite description of executable rules for transforming an input into an output. For every allowed input, an always-correct algorithm must produce an output satisfying the specification and must eventually stop. A randomized algorithm may make random choices, but the possible steps and the guarantee being claimed must still be precise.

Algorithmics studies how to formulate computational problems, choose representations, design algorithms, prove their properties, compare their resource use, and test their implementations. Its central question is not merely “Did this program run once?” but:

What problem is being solved, why is the answer valid, and what happens when the input or workload changes?

2 Learning goals

After studying this note, you should be able to:

- distinguish a problem, an instance, an algorithm, and a program;
- state inputs, outputs, preconditions, postconditions, and frame conditions;
- explain how representation and computational model affect an analysis;
- prove a simple loop correct with an invariant and a termination argument;
- distinguish exact, randomized, approximate, and heuristic methods;
- separate proof, resource analysis, and empirical evidence; and
- explain the difference between solving and efficiently verifying a decision problem.

3 Prerequisites

You should be comfortable reading assignments, conditionals, loops, functions, arrays, and simple mathematical notation. No previous complexity theory is assumed. This note introduces the terminology used by the later readings.

4 Problems, instances, algorithms, and programs

A **computational problem** describes a family of permitted inputs and the required relationship between input and output. One particular input is an **instance**. For example, sorting is a problem; the array `[4,1,4,2]` is an instance; `[1,2,4,4]` is a valid output for it.

An **algorithm** is a language-independent method for solving every permitted instance. A **program** is an implementation of an algorithm, or a combination of algorithms, in a particular language and environment. Several programs can implement the same algorithm, and several algorithms can solve the same problem.

Inputs must have finite representations. Their **size** is a chosen measure of that representation: the number of array entries, vertices and edges, characters, or bits. The value of an integer and the length of its representation are different. The integer 2^k is large in value but requires only $k + 1$ binary digits. This distinction matters whenever arithmetic operands are not bounded machine words.

Algorithms also make different kinds of promises:

- an **exact deterministic** algorithm always returns a required answer without random choices;
- a **randomized** algorithm uses random bits, so its cost or answer may be probabilistic;
- an **approximation algorithm** returns a solution with a stated quality guarantee; and
- a **heuristic** seeks useful answers but may offer only empirical evidence rather than a universal guarantee.

These categories describe guarantees, not implementation quality. A randomized or heuristic method still needs a precise interface and an honest statement of what is and is not guaranteed.

They are also not mutually exclusive. For example, an approximation algorithm may use random choices, and a randomized algorithm may be exact with random running time or may instead permit a stated probability of error. Name each guarantee separately rather than forcing a method into one label.

5 From specification to implementation

A reliable solution passes through distinct layers:

1. **Problem:** what relationship must hold between input and output?
2. **Representation:** how are the objects encoded?
3. **Algorithm:** which abstract steps transform the representation?
4. **Implementation:** how are the steps expressed and executed?

5. **Evaluation:** do proof, analysis, and measurement support the claims?

A **specification** states what must be achieved without prescribing every step. For `binary_search(A,x)`, one possible contract is:

- **precondition:** `A` is sorted in nondecreasing order;
- **postcondition:** return an index `i` with `A[i] == x`, or `NOT_FOUND` if no such index exists;
- **frame condition:** do not modify `A`.

If duplicates are allowed, the postcondition must say whether any occurrence, the first occurrence, or the last occurrence is required. Calling binary search on an unsorted array violates its precondition; it is not merely a slower use of the same algorithm.

An **abstract data type** (ADT) specifies values and operations. A **data structure** is a representation implementing that ADT. A stack client can use `push`, `pop`, and `is_empty` without depending on whether the implementation uses an array or linked nodes.

6 Computational and cost models

Analysis counts operations in a stated abstract machine. In a **unit-cost RAM model**, reading or writing one memory word, comparing two words, and performing basic arithmetic on words each cost constant time. A common **word-RAM** assumption is that one word contains enough bits to address the input, typically $\Theta(\log n)$ bits.

This model is useful, not universal. Adding two fixed-width integers can reasonably be one operation. Adding million-bit integers cannot: its cost depends on their bit lengths. Disk access, communication, cache misses, or energy may be more relevant than instruction count in other settings.

The analysis must therefore name:

- the input-size parameter or parameters;
- the operations being counted;
- the case being claimed, such as worst or expected; and
- whether reported space includes the input and output or only auxiliary storage.

7 Correctness from first principles

Partial correctness means that if the algorithm terminates, its result satisfies the postcondition. **Total correctness** adds the requirement that it terminates on every permitted input.

For an iterative algorithm, a **loop invariant** is a statement that connects the program state with already completed work. A proof has three parts:

1. **Initialization:** the invariant holds before the first iteration.
2. **Maintenance:** one iteration preserves it.
3. **Termination:** when the loop stops, the invariant implies the postcondition.

A separate progress measure proves termination. For a finite scan, the number of unprocessed positions decreases. For recursion, calls must move to smaller instances under a well-founded measure.

Recursive correctness has the corresponding form: prove base cases, assume recursive calls correctly solve smaller instances, and prove that combining their answers solves the current instance.

8 Worked trace: counting ones

Problem. Given an array containing only `0` and `1`, return the number of entries equal to `1` without modifying the array.

```
count_ones(A):
    count = 0
    i = 0
    while i < length(A):
        if A[i] == 1:
            count = count + 1
        i = i + 1
    return count
```

For `A = [1,0,1,1]`, the states after successive iterations are:

Processed prefix	`i`	`count`
`[]`	0	0
`[1]`	1	1
`[1,0]`	2	1
`[1,0,1]`	3	2
`[1,0,1,1]`	4	3

The invariant is:

> At the start of each test, `count` equals the number of ones in `A[0..i)`, and $0 \leq i \leq n$.

It holds initially because the empty prefix contains no ones. An iteration inspects exactly `A[i]`, updates `count` precisely when that value is one, and then extends the processed prefix by one. At termination `i=n`, so the prefix is the whole array and `count` satisfies the postcondition. Since `n-i` decreases by one, termination is guaranteed.

The algorithm takes $\Theta(n)$ time and $\Theta(1)$ auxiliary space in the unit-cost RAM model. The lower bound is also linear for an always-correct algorithm in the array-query model: if some position were never inspected, changing only that position from zero to one would require a different answer while leaving the execution unchanged.

9 Solving, checking, and computational difficulty

The correctness argument above concerns one specified algorithm. Complexity classes ask a different question: whether **some** algorithm with a given resource bound exists for every instance of a problem.

A **decision problem** asks a yes-or-no question. An optimisation problem can often be paired with a decision version. Instead of asking for the smallest number of bins, for example, ask whether all items fit in at most k bins.

For **Subset Sum**, the input contains integers and a target. The decision question is whether some subset sums to that target. A proposed subset is a **certificate** for a yes-instance: add its elements and check the target. Verification can be much easier to describe than finding the subset.

The class P contains decision problems solvable in polynomial time by a deterministic algorithm. A decision problem is in NP when a polynomial-time verifier and a polynomial certificate-size bound exist such that an instance is a yes-instance exactly when **some** certificate within that bound is accepted. Thus $P \subseteq NP$: for a problem in P, a verifier may ignore the certificate and run the polynomial-time decision algorithm itself. Whether $P = NP$ is unknown.

The name NP means nondeterministic polynomial time, not "non-polynomial". Membership in NP alone is therefore not evidence that a problem is hard.

The class co-NP contains complements of problems in NP: its yes-instances correspond to no-instances of the original problem. "I found no solution" is not automatically an efficiently checkable certificate. These definitions do not by themselves prove that a particular problem is hard; reductions and completeness theory are needed for that.

10 Evidence, experiments, and trade-offs

Algorithmic claims use complementary evidence:

- **proof** establishes correctness or a mathematical guarantee;
- **analysis** relates resource use to input characteristics under a model; and
- **experiment** measures an implementation under stated conditions.

A timing plot cannot prove correctness or an asymptotic bound. Conversely, an asymptotically efficient algorithm may lose for practical input sizes because of constants, memory locality, setup costs, or implementation complexity.

Choosing a method is therefore workload-dependent. State what is being optimised: worst-case latency, expected throughput, memory, block transfers, implementation effort, simplicity, numerical accuracy, or solution quality. "Algorithm A is better" is not meaningful without this criterion and the relevant assumptions.

11 Connections

- *Growth Functions* develops asymptotic notation and experimental interpretation.
- Linear structures, trees, heaps, hashing, and graphs show how representation changes operation costs.
- Later notes develop divide and conquer, dynamic programming, randomized methods, exact search, approximation, and heuristics.
- The distinction between finding and verifying solutions motivates complexity theory and the treatment of difficult optimisation problems.

12 Common mistakes

- Starting to code before defining the input and required output.
- Treating successful examples or tests as a correctness proof.
- Proving partial correctness but never proving termination.
- Omitting assumptions about ordering, arithmetic, mutability, or input encoding.
- Calling a method "efficient" without naming the cost model and input size.
- Reporting timings without input generation, environment, repetitions, and variability.
- Choosing a familiar data structure before identifying the required operations.

13 Self-check

1. Distinguish a problem, an instance, an algorithm, and a program using one example.
2. Give a precise contract for merging two sorted arrays.
3. State an invariant and a progress measure for computing a prefix sum.
4. Why does the unit-cost assumption need qualification for arbitrary-size integers?
5. Explain the difference between an exact, randomized, approximation, and heuristic method.
6. Why does a polynomial-time verifier for yes-instances not immediately give a polynomial-time solver?
7. Which evidence supports correctness, scalability, and practical speed?

14 Revision summary

- Specify the problem, permitted instances, representation, and computational model before analysing code.
- Correctness needs a contract, preserved reasoning, and termination.
- ADTs define behaviour; data structures implement that behaviour with different costs.
- Input size measures a representation, not merely a numeric value.
- Proof, analysis, and experiment answer different questions.
- P concerns efficient solving; NP concerns efficient verification of yes-certificates.
- Clear algorithmic communication states guarantees, assumptions, costs, evidence, and trade-offs.

15 Implementations and hands-on exploration

- Hypothesis generates and shrinks Python test inputs from stated domains. Property-based testing can expose missing cases, but finitely many successful tests are not a proof.
- Python `timeit` repeats small code fragments with basic timing controls. Its default treatment of garbage collection and its microbenchmark setting may not represent an application workload.
- Frama-C checks ACSL contracts, loop invariants, and other properties of C programs with analysis plug-ins. A proof establishes the written specification under the selected model, so an incomplete contract can still describe the wrong problem.
- Dafny combines preconditions, postconditions, frame specifications, invariants, and termination metrics with static verification and executable compilation. Solver success does not validate requirements that were never specified.

15.1 Small experiments

1. Implement `count_ones`, generate every binary array of length at most 8, and compare the result with `sum(A)`. Insert one boundary bug and record the smallest counterexample.
2. Assert the loop invariant at the loop head and again after each complete iteration. Which parts of total correctness are exercised by exhaustive tests up to length 8, and which claims still require a general proof?

16 Sources and further study

- Algorithmics lecture materials, organisation and introduction decks, Jaak Vilo.
- (Cormen et al. 2022), Chapters 1-2.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.