

Algorithmic Thinking

Turn a problem into a clear, testable sequence of steps.

Beginner lesson · Estimated reading time: 8–12 minutes

Introduction

Algorithmic thinking means defining a problem precisely, breaking it into manageable parts, choosing a procedure, and checking that the procedure works. It is useful before writing code: a good algorithm can be implemented in many programming languages.

Learning objectives

- Describe an algorithm and its input/output contract.
- Decompose a problem and identify relevant information.
- Trace pseudocode and reason about branches and loops.
- Estimate simple time complexity and test edge cases.

1. What is an algorithm?

An algorithm is a finite, unambiguous sequence of steps that transforms input into output. A useful specification states what input is allowed, what result is expected, and which assumptions apply. An algorithm is not tied to a particular programming language.

Example: to find the largest value in a non-empty list, keep a candidate maximum. Visit each value; if it is larger than the candidate, replace the candidate. When the list ends, the candidate is the maximum.

2. Decomposition and abstraction

Decomposition splits a large task into smaller tasks with clear responsibilities. Abstraction keeps the details needed for the current question and temporarily ignores irrelevant detail. For a student grade calculator, separate reading grades, validating them, computing an average, and displaying a result.

Ask: What information do I receive? What must I produce? What rules constrain the answer? What smaller steps can be tested independently?

3. Writing and tracing pseudocode

Pseudocode describes logic without requiring the syntax of a specific language. Indentation shows which steps belong to a condition or loop.

```
largest ← A[0]
FOR each value x in A
  IF x > largest THEN
    largest ← x
RETURN largest
```

Trace for $A = [4, 2, 9, 5]$: largest begins at 4; 2 changes nothing; 9 replaces 4; 5 changes nothing; output is 9. This algorithm assumes A contains at least one value.

4. Correctness, edge cases, and testing

A program can work on the example you tried and still fail elsewhere. Test ordinary cases, boundary cases, and invalid inputs. For maximum: test a one-element list, a list of negative values, repeated maxima, and an empty list. The stated algorithm needs a non-empty list; either reject an empty list or define a separate result for it.

An invariant is a statement that remains true during a loop. Here, after processing the first k values, 'largest' is the largest among those k values. Each new value either becomes the maximum or leaves it unchanged.

5. A first look at time complexity

Time complexity describes how the number of operations grows as input size n grows. Finding a maximum in a list examines every value once, so its running time grows linearly: $O(n)$. This does not promise an exact number of seconds; hardware and implementation also matter.

One loop over n values is commonly $O(n)$. Two nested loops each visiting n values may be $O(n^2)$. Ignore constants and lower-order terms when describing asymptotic growth: $3n + 8$ is $O(n)$.

6. Common mistakes

- Starting with an arbitrary value such as zero when all inputs may be negative.
- Forgetting empty input or other boundary cases.
- Mixing the problem specification with language-specific syntax.
- Testing only one happy-path example.
- Claiming an algorithm is correct without explaining why its steps preserve the required result.

Practice exercises

EX1: Trace the maximum algorithm for $[-3, -8, -1, -1]$. What is the final value of largest?

Answer: -1 . The candidate starts at -3 , then becomes -1 when that value is encountered.

EX2: Write three test cases for an algorithm that checks whether a number is even.

Answer: For example: 0 (even), 7 (odd), and -4 (even). These cover zero, an odd value, and a negative value.

EX3: A loop examines every item in a list once. What is its time complexity in terms of n ?

Answer: $O(n)$, linear time.