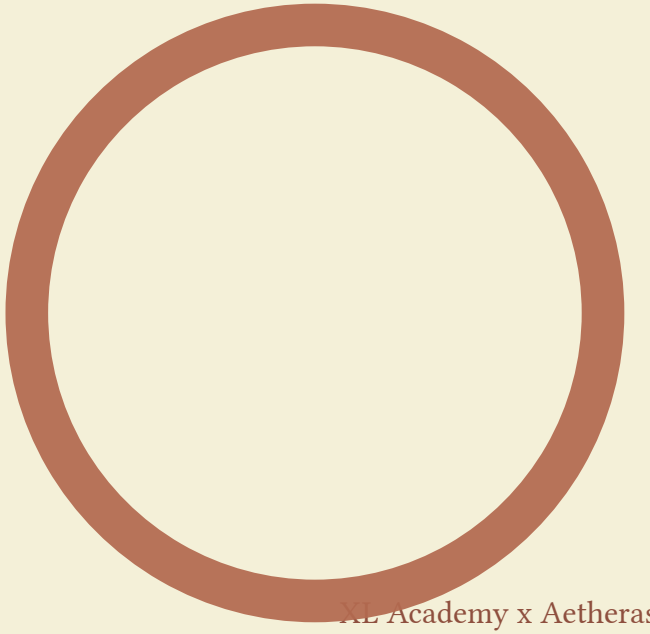
L14

AI Game Development & Computer Science Summer Program

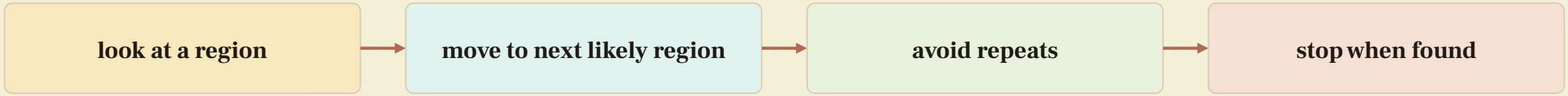
Lecture 14: Algorithms I: Correctness and Efficiency

Precise procedures, expression, language independence, and correctness.



A search path can be an algorithm

Optimizing a path is still a precise sequence of choices.



The problem

- Find a hidden target quickly in a crowded scene.

The metric

- Minimize wasted scanning distance or repeated work.

The lesson

- An algorithm can optimize behavior without changing the goal.

Algorithms: Definitions

First, name what counts as an algorithm.

An algorithm is a precise sequence of instructions

A computer can execute it, but the idea is older than computers.

Algorithm

- Takes input, when the task needs input.
- Follows a precise sequence of steps.
- Uses decisions, repetition, or recursion when needed.
- Produces output or a visible effect.
- Eventually stops.

Not enough

- A goal with no steps.
- A single example.
- A line of code with no stated contract.
- A process that can run forever.

A clear algorithm should be understandable by a person and executable by a program.

Algorithms existed before programming languages

Many real-world procedures already have the shape of an algorithm.

Recipes

- Ordered steps
- Branch on ingredients
- Stop when done

Construction

- Input materials
- Sequence actions
- Check constraints

Games

- Rules decide legal moves
- Turns repeat
- A win condition stops play

Nature

- Cells follow biochemical procedures that transform inputs.

Programming gives us a way to express algorithms precisely enough for a computer.

You already know several algorithms

This course has used algorithms even when we were not naming them yet.

Task	Algorithm shape	TypeScript example
length of a word	count characters	word.length
target in a list	check each item	items.includes(target)
ask a user	prompt, validate, respond	form submit handler
sort a list	compare and reorder	items.toSorted(...)
combine values	accumulate across items	reduce(...)

Standard algorithms show up in products

Recognizing the standard problem helps you choose a tested strategy.

Validation

- Check whether an ID or card number has a valid digit pattern.

Compression

- Store repeated data using shorter encodings.

Ranking

- Score candidates, then sort or filter by relevance.

Routing

- Search possible paths and prefer cheaper ones.

Course habit: name the abstract problem before writing the implementation.

Algorithms use a few building blocks

Complex behavior comes from combining control patterns carefully

Block	What it means	TypeScript shape
sequencing	do each step in order	statements
selection	choose a path using a condition	if / else
iteration	repeat while a condition or items remain	for, while, for...of
recursion	solve a smaller version of the same problem	a function calls itself

Composition

- Existing correct algorithms can become reliable building blocks for larger algorithms.

Check: which statement is false?

Pick the claim that does not belong.

A.

- Algorithms can be valuable.

B.

- You learned procedures before you could program.

C.

- Proving algorithms are correct is easy.

D.

- Algorithms can adapt when the procedure includes feedback.

After you choose

- The false claim is that proving correctness is easy.
- We can test implementations, but correctness for every valid input takes careful reasoning.

Algorithms: Properties and Expression

Next, make procedures precise enough to share.

Algorithms can be combined and reused

A known correct algorithm reduces the risk inside a larger design.

Combine

- Filter valid moves.
- Score each move.
- Choose the maximum.

Reuse

- Use a standard search, sort, or validation algorithm when it matches the contract.

Improve

- A new algorithm can unlock a better product experience or a new capability.

You do not need to reinvent a standard algorithm every time; you do need to know its contract.

Ambiguous instructions cause surprising behavior

Code follows the literal rule you gave it, not the intention you had in mind.

Ambiguous

- Buy one loaf of bread.
- If there are eggs, buy twelve.

Algorithm-ready

- Buy one loaf of bread.
- If eggs are available, also buy twelve eggs.
- If eggs are not available, buy only the bread.

The fix is not more clever code. The fix is a more precise procedure.

Algorithms can be expressed many ways

Choose the notation that helps the audience understand and execute the idea

Expression	Best use	Limit
natural language	communicating the idea	can be ambiguous
pseudocode	reasoning before syntax	not directly executable
TypeScript	typed browser and game code	syntax can distract from the idea
diagram or table	tracing state changes	may omit edge cases

Strong algorithm work often moves through several representations before final code.

An algorithm is not the same as one function

The algorithm is the strategy; a function is one implementation.

Problem

- Find the maximum value in a non-empty list.

Pseudocode

- Set max to the first value.
- Scan each remaining value.
- If a value is larger, replace max.
- Return max.

```
const maxValue = (values: readonly
number[]): number => {
  let max = values[0];
  for (const value of values.slice(1)) {
    if (value > max) max = value;
  }
  return max;
};
```

A contract must say this input is non-empty, or the implementation needs an empty-case policy.

Different languages express different algorithms well

Language family	Good fit	Tradeoff
C / C++	systems and performance-critical code	manual control and sharper edges
Java / C#	portable application code	more ceremony
Python	quick scripts and data exploration	runtime checks
TypeScript	typed browser apps and game UI	types and tooling up front
visual blocks	learning and demos	less like production code

Check: which language is most powerful?

Assume each language has enough memory and time for the task.

A.

- C++

B.

- Java

C.

- Python

D.

- TypeScript

E.

- They can express the same computable algorithms.

After you choose

- The important answer is equivalence of expression.
- Some languages make an algorithm clearer or faster to implement, but the algorithm itself is language-independent.

Pipeline algorithms compose simple steps

Filter, map, sort, and slice are reusable algorithms for transforming collections.

```
type Player = { name: string; score: number; active: boolean };

const leaderboard = players
  .filter((player) => player.active)
  .map((player) => ({ name: player.name, score: player.score }))
  .toSorted((a, b) => b.score - a.score)
  .slice(0, 5);
```

What is being combined?

- filter keeps valid items.
- map transforms each item.
- sort orders by a comparison rule.
- slice takes a prefix.

AI assist: useful for proposing pipeline alternatives, but verify each step's contract and edge cases.

Algorithms: Correctness

Finally, ask what it means for a procedure to be right.

Correct means every valid input works

A correct algorithm returns the promised output and does not run forever.

Correctness claim

- For every valid input, the output matches the contract.
- The algorithm terminates.

Incorrect can mean

- Wrong answer.
- Crash or invalid state.
- Never finishes.
- Works only for the examples you tried.

A program can be useful while still relying on an incorrect or incomplete algorithm.

Testing and reasoning answer different questions

Tests are essential, but they do not prove every possible input.

Tool	What it can show	Limit
unit tests	selected examples behave correctly	cannot cover every input
debugging	one failure's cause	does not prove other cases
formal reasoning	why the contract holds in general	requires a precise algorithm

AI assist: ask for counterexamples and test ideas, then decide which contract risk each one targets.

An invariant is a plain-language proof tool

A useful fact should stay true after each repetition.

```
let max = values[0];
for (const value of values.slice(1)) {
  if (value > max) max = value;
}
```

Invariant

- After each checked value, max is the largest value seen so far.

Checked values	max
4	4
4, 9	9
4, 9, 2	9
4, 9, 2, 7	9

Exercise: find a counterexample

A small bad input can reveal the hidden assumption.

```
const maxScore = (scores: readonly number[]): number => {
  let max = 0;
  for (const score of scores) {
    if (score > max) max = score;
  }
  return max;
};
```

Questions

- What input breaks this function?
- What contract was assumed but not stated?
- How should it handle an empty list?
- What invariant would be true after the fix?

Summary: algorithms are precise, reusable, and checkable

The point is not just writing code; it is writing procedures we can reason about.

Keep

- An algorithm is a finite, precise procedure.
- A language expresses the algorithm; it does not define what algorithms exist.
- Known algorithms can be reused as building blocks.
- Correctness needs a contract, tests, and reasoning.
- Tradeoffs become visible only after the task is stated precisely.

Practice: write the algorithm before the code

Choose a small feature and make the procedure exact.

```
type Move = { name: string; score: number; legal: boolean };

const chooseMove = (moves: readonly Move[]): Move | undefined => {
  // 1. State the contract.
  // 2. Define tie and empty-list behavior.
  // 3. Implement the scan.
  // 4. Explain the invariant.
};
```

Deliverable

- One contract.
- One TypeScript implementation.
- Three tests.
- One correctness explanation.

Proof checkpoint

Move on when your algorithm has a contract, tests, and a correctness reason.

Exit ticket

- One algorithm stated without syntax.
- One ambiguity rewritten as a precise rule.
- One test that attacks the contract.
- One invariant or correctness reason.
- One place where language choice affects expression, not solvability.