EECS3101 Algorithms (Summer'26)

Digest of Tutorial 7, July 2

Iterative Algorithms: Precondition, Postcondition, Loop Invariants

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas

- Iterative algorithms can be specified and reasoned about using three contracts: a precondition Q, a postcondition R, and a loop invariant I.
- The loop invariant must be *established* after initialization and *preserved* after every loop iteration; it is the key bridge from the loop's intermediate state to the final result.
- When a loop exits, its correctness argument must show that $\neg B \wedge I \Rightarrow R$, where B is the stay condition of the loop.
- The `findMax` example showed how to write a complete postcondition: the result must dominate every array element and must itself be an array element.
- A plausible-looking invariant can still be wrong. Tracing the proposed invariant on $[20, 10, 40, 30]$ exposed a violation caused by incrementing the loop counter before the invariant is checked.
- Tutorial 7 prepares you to reason about the loop invariant for Dijkstra's algorithm in the next lecture. Assignment 2 is due July 8; the Written Test is July 13 and Programming Test 2 is July 20.

Roadmap

Course timeline and the purpose of this tutorial

Jackie situated the tutorial between the introduction to Dijkstra's algorithm and the upcoming graph material. The immediate technical goal was not another graph trace: it was to build the correctness language needed for iterative algorithms, especially for Dijkstra next week. Continue working on Assignment 2, which is due July 8, and use the test-free week to prepare for the Written Test on July 13 and Programming Test 2 on July 20.

The contract structure of an iterative algorithm

The notes introduced the general pattern $Q\ \{S_{init};\ \texttt{while}(B)\ \{S_{body}\}\}\ R$. Here, Q is the input precondition, S_{init} prepares the state for iteration, B is the boolean stay condition, and R is the promised input–output relation after termination. The additional contract I, the loop invariant, is checked after initialization and after each execution of the loop body.

How the flowchart locates correctness failures

The flowchart separates several ways an execution can fail. Illegal input violates Q before the algorithm begins. After initialization, I must already be true; otherwise it was not established. At the end of every iteration, I must still be true; otherwise it was not preserved. When B becomes false, the loop exits and the final state must satisfy R. The intended safe state has $\neg B$, I, and R all true.

A small trace: `testLI`

The counter example used precondition $i = 1$, invariant $1 \le i \le 6$, stay condition $i \le 5$, loop body `i = i + 1`, and postcondition $i = 6$. It illustrates two important habits: the invariant remains true even after the final iteration, and a postcondition may be false during intermediate iterations without being a postcondition violation. The postcondition is checked only after exit.

Deriving public contracts for `findMax`

For a non-null, non-empty array `a`, `findMax` returns `result`. Jackie first developed the requirement that `result` is at least every array entry, then showed why that alone is insufficient: a faulty implementation could return an arbitrary value larger than every element. A complete postcondition must also require that `result` occurs in the input array. Express array properties by quantifying over valid indices, not by treating an array as a mathematical set.

Tracing a candidate loop invariant

The proposed invariant said that `result` is at least every `a[j]` for $0 \le j \le i$. On $[20, 10, 40, 30]$, it is established initially and survives the first iteration, but fails after the second iteration: `i` becomes 2 while `result` remains 20, so the condition wrongly includes `a[2]`$= 40$ before that entry has been processed. The point is to trace a candidate invariant carefully rather than trust its wording. The follow-up tutorial solution gives the corrected invariant and the formal correctness argument.

Key Terms, Notation, and Structures

- **Precondition Q:** A boolean condition on valid input values that must hold before the algorithm begins.
- **Postcondition R:** A boolean input–output relation promised when the algorithm terminates normally.
- **Loop invariant I:** A boolean correctness property that must hold after initialization and after every completed loop iteration.
- **Initialization S_{init}:** Statements before the loop that prepare the first iteration and must establish I.
- **Stay condition B:** The boolean condition in `while (B)`. When B is true, another iteration runs; when $\neg B$ is true, the loop exits.
- **Loop body S_{body}:** The statements executed in one iteration. They must preserve I.
- **Established / preserved:** I is *established* if it holds after initialization; it is *preserved* if every loop body execution keeps it true.

- **Safe final state:** A correct exit state satisfies $\neg B$, I, and R. The central proof obligation is $(\neg B) \wedge I \Rightarrow R$.
- **Loop variant:** An integer-valued measure used to prove termination. It is different from a loop invariant, which is a boolean predicate used for correctness.
- **Universal and existential quantifiers:** $\forall$ means "for every" and $\exists$ means "there exists." For arrays, quantify over legal indices such as $0 \leq j <$ `a.length`.
- **Public contracts vs. internal state:** Q refers to inputs; R refers to inputs and outputs. A loop invariant may additionally refer to loop counters and local variables because it describes intermediate progress.

Key Examples and Demonstrations

The `testLI` contract trace

Setup: Start with $i = 1$, use an empty initialization, repeatedly increment `i` while $i \leq 5$, and maintain $1 \leq i \leq 6$.

Main point: After the last iteration, $i = 6$: the invariant still holds, the stay condition is false, and the postcondition $i = 6$ holds. This is the intended exit pattern, not an exception to the invariant.

What to practise: Fill in the trace table yourself for initialization and all five iterations. At each row, evaluate I, B, and R, and distinguish "R is currently false" from a genuine postcondition violation.

A complete postcondition for `findMax`

Setup: For a valid array `a`, `findMax` returns `result`. The notes used $[20, 10, 40, 30]$ as the running example.

Main point: A complete postcondition combines the following two claims:

$$R_1: \quad \forall j \bullet \big(0 \leq j < \texttt{a.length} \Rightarrow \texttt{result} \geq \texttt{a[j]}\big)$$
$$R_2: \quad \exists k \bullet \big(0 \leq k < \texttt{a.length} \wedge \texttt{result} = \texttt{a[k]}\big)$$

R_1 says the result is at least every array value; R_2 rules out values that are not actually in the array.

What to practise: Explain why R_1 alone would incorrectly allow a result such as $+\infty$, then rewrite R_2 using an index variable rather than informal set-membership notation.

Finding the flaw in a proposed invariant

Setup: Consider the candidate $\forall j \bullet (0 \leq j \leq i \Rightarrow \texttt{result} \geq \texttt{a[j]})$ for the given `findMax` loop, where `i` is incremented at the end of the loop body.

Main point: The invariant's range is one step too large after `i` is incremented. On the lecture array, it fails when `i` = 2 and `result` = 20, because it now requires $20 \geq$ `a[2]` $= 40$.

What to practise: Recreate the initialization, first iteration, and second iteration in a table. State exactly which value of j witnesses the violation, then use the posted exercise solution to compare against the correct invariant and proof.

Common Pitfalls and Misconceptions

Confusing the stay condition with the loop invariant

Pitfall: Treating B as the invariant because both are boolean expressions checked during execution.

Why it causes problems: B only decides whether another iteration occurs. It does not usually provide the stronger progress information needed to derive the postcondition at exit.

How to avoid it: Ask whether $\neg B$ together with the proposed I is strong enough to imply R. If not, the proposed invariant is not useful for the correctness proof.

Expecting the postcondition to hold throughout the loop

Pitfall: Declaring an error because R is false during an intermediate iteration.

Why it causes problems: R describes the final output after the loop exits, not a promise about partial progress.

How to avoid it: Check R only on the exit path, after B becomes false. During iterations, focus on whether I is established or preserved.

Writing an incomplete postcondition for `findMax`

Pitfall: Specifying only that `result` is at least every array entry.

Why it causes problems: A value not stored in the array could still satisfy that condition, so the contract would permit an incorrect implementation.

How to avoid it: Combine a universal dominance condition with an existential membership condition over valid array indices.

Using loop counters in public contracts

Pitfall: Defining Q or R using an internal loop counter such as `i`.

Why it causes problems: Preconditions and postconditions should state the public input and output behaviour, independent of one particular implementation.

How to avoid it: Use input variables in Q, input and output variables in R, and reserve counters or local variables for I.

Treating an array as a set

Pitfall: Writing an informal membership assertion such as `result` $\in$ `a`.

Why it causes problems: An array is an indexed sequence, so the condition does not state which array positions are being considered.

How to avoid it: Use $\exists k$ with the bounds $0 \leq k <$ `a.length`, then compare `result` with `a[k]`.

Trusting a plausible invariant without tracing it

Pitfall: Assuming the proposed $j \leq i$ invariant is correct because it resembles the final maximum condition.

Why it causes problems: The invariant is checked after `i` changes, so it can include an element that the loop body has not processed yet.

How to avoid it: Trace the exact state after initialization and after each iteration, paying attention to the order of assignments and the value of each quantified bound.

What You Should Be Able to Do

After this tutorial, you should be able to:

- identify Q, R, I, S_{init}, B, and S_{body} in a simple iterative algorithm;
- explain the difference between a loop invariant used for correctness and a loop variant used for termination;
- trace a loop and determine whether an invariant is established, preserved, or violated;
- explain why a correct exit state must satisfy $\neg B$, I, and R, and apply $(\neg B) \wedge I \Rightarrow R$;
- write the precondition for a non-null, non-empty input array;
- formulate the two essential parts of a complete `findMax` postcondition using $\forall$ and $\exists$ over array indices;
- distinguish public contracts from implementation-specific variables; and
- diagnose the failure of a candidate loop invariant by giving a concrete trace and counterexample state.

Suggested Practice

- Rewatch the Tutorial 7 flowchart discussion and redraw the paths for: precondition violation, invariant not established, invariant not preserved, postcondition violation, and safe termination.
- Recreate the `testLI` trace with columns for `i`, I, B, and R. Explain why R may be false before termination.
- For `findMax`, write Q, R_1, and R_2 formally, then test whether each condition rejects deliberately faulty implementations.

- Trace the proposed $j \leq i$ invariant on $[20, 10, 40, 30]$ and identify the exact iteration and array index at which it fails.
- Watch the Tutorial 7 exercise-solution video for the correct `findMax` invariant and the proof using the exit condition and invariant.
- Review predicate and propositional logic, especially $\forall$, $\exists$, $\Rightarrow$, $\wedge$, and negation, before the Written Test.
- Review the Lecture 16 Dijkstra trace so that you are ready to apply the loop-invariant framework to Dijkstra's algorithm in Lecture 17.
- Continue Assignment 2 before its July 8 deadline; its work also supports Programming Test 2 preparation.

Where to Review

- **Tutorial 7 iPad/PDF notes, page 2:** the general syntax Q, R, I, S_{init}, B, and S_{body}, plus the flowchart of contract checks and violations.
- **Tutorial 7 notes, page 3:** the `testLI` trace and the exit reasoning $(\neg B) \wedge I \Rightarrow R$.
- **Tutorial 7 notes, pages 4–6:** `findMax`, the precondition, the two-part postcondition, index-based quantification, and the public/private distinction for contract variables.
- **Tutorial 7 notes, page 7:** the proposed incorrect invariant and its failure on $[20, 10, 40, 30]$.
- **Graph lecture slides 62–67:** the formal loop-correctness material, including the flowchart on slide 65.
- **Tutorial 7 page on the lecture site:** the session recording and the posted exercise-solution video for the corrected loop invariant and formal proof.
- **Lecture 16 notes and recording:** weighted graphs and the Dijkstra trace that this tutorial prepares you to revisit in Lecture 17.

Closing Reminder

Loop correctness is not learned by memorizing names such as precondition or invariant. Revisit the notes and recording, trace the examples carefully, write the predicates yourself, and check the exact state at initialization, after each iteration, and at exit. Use the posted Tutorial 7 solution to verify your reasoning, then apply the same habits when Dijkstra's algorithm returns next week.