Analysis Of Algorithms

LECTURE 03

Analysis Of Algorithms

LECTURE 03

PROVING CORRECTNESS OF ALGORITHM USING LOOP INVARIANTS

Dr. Shamila Nasreen

Introduction to Algorithms

- Why algorithms?
  - Consider the problems as special cases of general problems.
    - Searching for an element in any given list
    - Sorting any given list, so its elements are in increasing/decreasing order

Reasoning (formally) about algorithms

- 1. I/O specs: Needed for correctness proofs, performance analysis.
  - INPUT: A[1..n] - an array of integers
  - OUTPUT: an element m of A such that A[j] ≤ m, 1 ≤ j ≤ length(A)
- 2. CORRECTNESS: The algorithm satisfies the output specs for EVERY valid input
- 3. ANALYSIS: Compute the running time, the space requirements, number of cache misses, disk accesses, network accesses,….

Algorithms Correctness

- Algorithm correctness is crucial in ensuring that an algorithm functions as expected for all valid inputs.
- One powerful method for proving correctness is **loop invariants**, which help verify the correctness of iterative algorithms.
- A **loop invariant** is a property that holds true before and after each iteration of a loop. It serves as a **mathematical assertion** that remains unchanged throughout the loop's execution.
  - It provides a structured way to reason about the correctness of iterative algorithms.

Key Aspects of Loop Invariants:

Holds **before** the first loop iteration (initialization).

Holds **before and after** each loop iteration (maintenance).

Ensures correctness when the loop terminates (termination).

Mathematical Formulation

Formally, a loop invariant $P(i)$ is a property that holds at the start of each loop iteration i.

If:

1. **Base Case (Initialization)**: $P(0)$ is true before the loop starts.
2. **Inductive Step (Maintenance)**: If $P(k)$ is true before the k-th iteration, then $P(k+1)$ remains true.
3. **Termination**: When the loop exits at $i = n$, $P(n)$ guarantees the desired result.

Then, by **mathematical induction**, the invariant holds for all iterations, proving the algorithm's correctness.

Steps to Prove Algorithm Correctness Using Loop Invariants

- To prove the correctness of an algorithm using loop invariants, we follow the **three-step method**:

Step 1: Initialization

- Show that the invariant holds **before** the loop begins.
- This establishes a base case that validates the algorithm's starting conditions.

Step 2: Maintenance

- Show that if the invariant is true before an iteration, it remains true after the iteration.
- This step ensures that the loop invariant is preserved throughout the loop execution.

Step 3: Termination

- Show that when the loop terminates, the invariant, along with the loop termination condition, provides a **useful property** that leads to the correctness of the algorithm.

How to Formulate a Loop Invariant from a Problem Statement?

To derive a loop invariant, follow these steps:

Step 1: Understand the Goal of the Algorithm

- Identify what the algorithm is trying to achieve (e.g., sorting, searching, finding the maximum element).

Step 2: Identify the Loop's Functionality

- Determine what the loop does at each iteration (e.g., inserting an element, checking a condition, swapping values).

Step 3: Define a Property that Remains True Throughout the Loop

- The property should reflect the progress made toward solving the problem.
- It must hold **before** the first iteration, stay true **throughout** execution, and ensure correctness **upon termination**.

Loop Invariants: Examples

Example: Insertion Sort

- Problem Statement: Sort an array in ascending order.
- Loop Functionality: Each iteration inserts one element into the correct position.
- Loop Invariant: At the start of each iteration, the subarray A[0:j] is sorted.

Example: Finding Maximum Element

- Problem Statement: Find the maximum value in an array.
- Loop Functionality: Keep track of the maximum found so far.
- Loop Invariant: At the start of each iteration, max_val holds the maximum value among the first i elements.

Example 1: Finding the Maximum Element in an Array

```
def find_max(A):
    max_val = A[0]
    for i in range(1, len(A)):
        if A[i] > max_val:
            max_val = A[i]
    return max_val
```

- Algorithm: Maximum Finding
- **Invariant:** At the start of each iteration of the loop, max_val contains the maximum value among the first i elements of A.

Example 1: Finding the Maximum Element in an Array

```python
def find_max(A):
    max_val = A[0]
    for i in range(1, len(A)):
        if A[i] > max_val:
            max_val = A[i]
    return max_val
```

Proof Using Loop Invariant:

Step 1: Initialization

Before the loop starts (when `i = 0`):

- `max_val = A[0]`, which means it holds the maximum value for the first element.
- Since `A[0]` is the only value considered, the invariant holds.

Step 2: Maintenance

- Suppose before the `i`-th iteration, `max_val` correctly stores the maximum value of the first `i` elements (`A[0]` to `A[i-1]`).
- In the `i`-th iteration, we compare `A[i]` with `max_val`:
  - If `A[i] > max_val`, we update `max_val = A[i]`.
  - Otherwise, `max_val` remains unchanged.
- This ensures that `max_val` always holds the largest value among `A[0]` to `A[i]`.

Step 3: Termination

- The loop stops when `i = n`, meaning all elements have been checked.
- By the invariant, `max_val` contains the maximum value of the entire array.
- The algorithm is correct.

Example 2: Summing the First n Elements of an Array

Loop Invariant:

At the start of each iteration, `sum_val` contains the sum of the first `i` elements of `A`.

Proof Using Loop Invariant:

Step 1: Initialization

Before the loop starts (`i = 0`):

- `sum_val = 0`, which correctly represents the sum of an empty set of elements (`A[0:0]`).
- Thus, the invariant holds.

Step 2: Maintenance

- Assume before iteration `i`, `sum_val` holds the sum of the first `i` elements (`A[0]` to `A[i-1]`).
- In the `i`-th iteration, we add `A[i]` to `sum_val`.
- After addition, `sum_val` correctly represents the sum of the first `i+1` elements (`A[0]` to `A[i]`).
- Thus, the invariant holds for the next iteration.

Step 3: Termination

- The loop stops when `i = n`, meaning all elements have been processed.
- At this point, `sum_val` contains the sum of all elements from `A[0]` to `A[n-1]`.
- The algorithm is correct.

```python
def sum_array(A):
    sum_val = 0  # Step 1: Initialize sum_val to 0
    for i in range(len(A)):
        sum_val += A[i]  # Add the current element to sum_val
    return sum_val
```

Example 3: Insertion Sort

- Loop Invariant for Insertion Sort:

Invariant: At the start of each iteration of the outer loop, the subarray A[0:j] contains the same elements that were originally in A[0:j], but in sorted order.

Step 1: Initialization

- Before the first iteration (j=1), A[0:1] (i.e., a single-element subarray) is trivially sorted.
- Thus, the invariant holds initially.

Step 2: Maintenance

- Suppose A[0:j] is sorted at the start of iteration j.
- The loop inserts A[j] into the correct position by shifting larger elements to the right.
- After insertion, A[0:j+1] remains sorted.
- Thus, the invariant holds after every iteration.

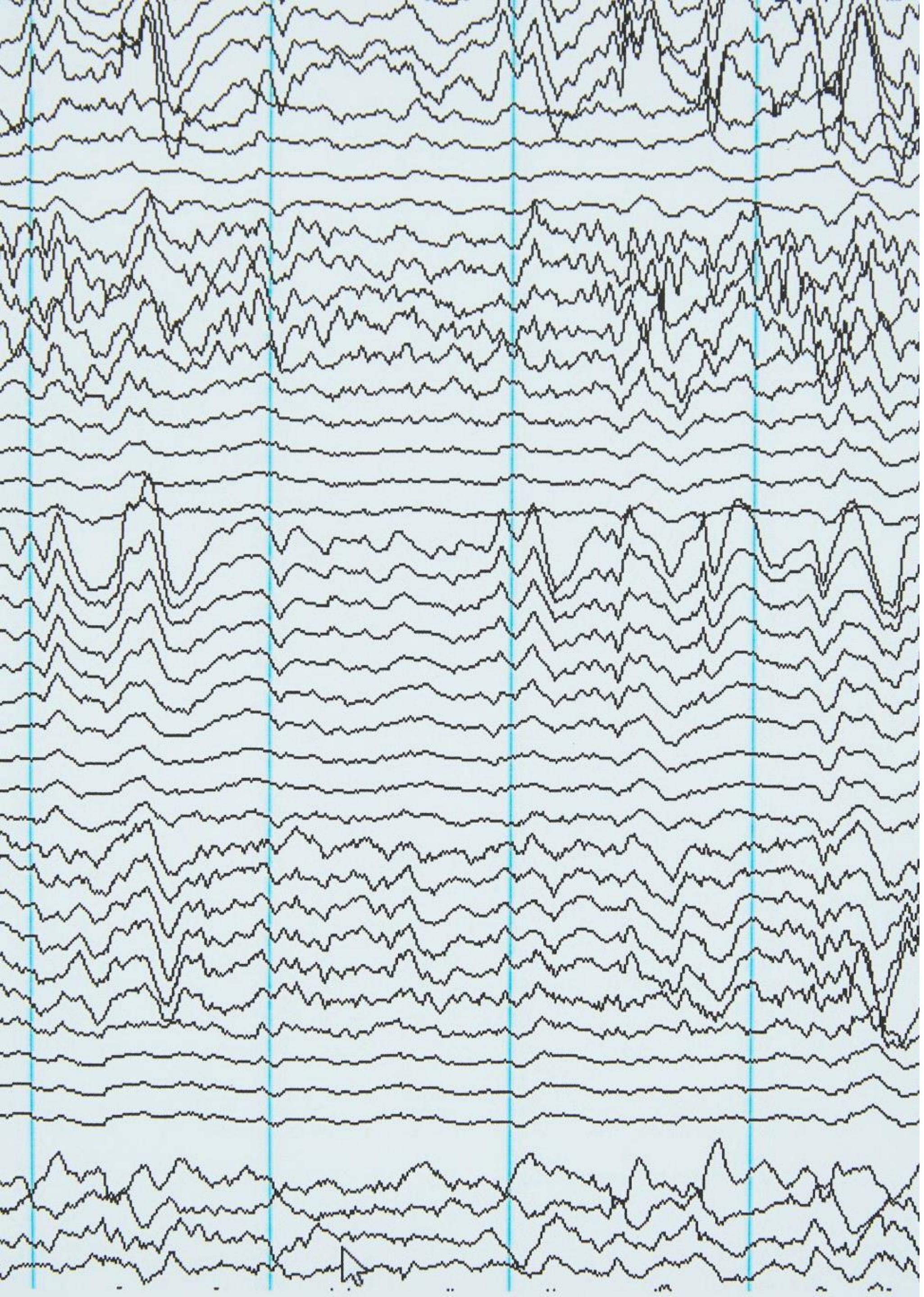
Step 3: Termination

- The outer loop terminates when j = n, meaning all elements have been inserted into their correct positions.
- The loop invariant ensures that A[0:n] is sorted at the end.
- Thus, the algorithm is correct.

```python
def insertion_sort(A):
    for j in range(1, len(A)):
        key = A[j]
        i = j - 1
        while i >= 0 and A[i] > key:
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
```

Why Use Loop Invariants?

- Provides a **systematic method** for proving correctness.
- Helps **understand and debug** iterative algorithms.
- Essential in **formal verification** and proving program properties.
- Useful in competitive programming and algorithm design.



Class Activity (Submitted on 28-03-2025)

- Computing Factorial Using Iteration
- Prove the correctness of the **Binary Search** algorithm using loop invariants.
  - Identify the loop invariant property
  - Define the three steps for both Problems