Algorithms

Lecture Notes for CS310 / CS5102

Dr. Mudassir Shabbir

January 22, 2026

Contents

Chapter 1

Introduction to Algorithms and Runtime Analysis

1.1 Overview and Philosophy

Algorithms are at the heart of computer science. An algorithm is not merely a piece of code—it is a *recipe or blueprint* for solving a computational problem. This course focuses on both the design and analysis of algorithms. By the end, you should be able to:

- identify appropriate algorithmic strategies for a problem,
- reason about their correctness, and
- predict and compare their efficiency rigorously.

The study of algorithms separates the *idea* of a solution from any particular implementation. This abstraction allows us to analyze the scalability of a method independently of programming language, hardware, or system-specific factors.

Key Questions in Algorithmic Thinking:

1. How do we systematically design algorithms for new problems?
2. How do we prove that an algorithm is correct?
3. How do we measure and compare algorithmic efficiency?

1.2 Instructor, Teaching Team, and Learning Environment

The course is taught by **Dr. Mudassir Shabbir**, with co-instructor **Aamina Jamal Khan** and a team of TAs.

The teaching team spans academic research and industrial experience. Students are encouraged to engage actively, ask questions freely, and collaborate with peers. The course uses a mixture of lectures, in-class worksheets, homework assignments, and online participation.

1.3 Communication and Professional Conduct

1.3.1 Communication Channels

Communication occurs primarily through the LMS and Slack. Students are expected to:

- check announcements regularly,
- participate in discussion threads, and
- help peers when possible.

1.3.2 Email Etiquette

Professional communication is an essential skill. Always:

- include "CS310" in the subject line,
- clearly state your name, student ID, and section,
- ask specific and focused questions.

Example: Instead of writing "I don't understand HW3," write: "I am confused about why the loop invariant in Algorithm 2 guarantees correctness after iteration 3. I tried X and Y, but I am unsure if my reasoning is correct."

1.4 Course Policies and Evaluation

1.4.1 Classroom Expectations

The classroom is a shared intellectual space. Students are expected to:

- arrive on time,
- silence electronic devices,
- avoid side conversations, and
- participate actively.

Confusion is normal; silence is not a virtue. Curiosity is.

1.4.2 Grading Philosophy

Assessment consists of exams, homework, worksheets, and participation. While approximate grade ranges are provided, final grades are determined *relatively*. The purpose of grading is to measure understanding, not to rank students.

1.4.3 Recommended References

Primary textbooks:

- **Cormen, Leiserson, Rivest, Stein**, *Introduction to Algorithms*, MIT Press, 4th edition.
- **Kleinberg and Tardos**, *Algorithm Design*, Pearson.

CLRS emphasizes formal proofs, rigorous analysis, and comprehensive coverage. Kleinberg-Tardos emphasizes design paradigms and problem-solving intuition. Supplementary reading for intuition: *Grokking Algorithms* by Bhargava.

1.5 What Is an Algorithm?

An algorithm is more than just a piece of code; it is a precise **step-by-step procedure** to solve a computational problem. The instructions must be **unambiguous**, **finite**, and guaranteed to **terminate** with a correct solution for all valid inputs.

Formal Definition (CLRS, Ch. 1). An *algorithm* is a finite sequence of well-defined instructions, each of which can be executed in a finite amount of time, that transforms input into a desired output.

Key Properties of Algorithms

- **Input:** An algorithm has zero or more inputs, taken from a specified domain.
- **Output:** It produces at least one output—a solution or a result.
- **Definiteness:** Each step is clearly and unambiguously defined.
- **Finiteness:** It must terminate after a finite number of steps.
- **Effectiveness:** Every step can be carried out in a finite amount of time using basic operations.

Example: Sorting vs Selection Understanding the distinction between a problem statement and an algorithm is crucial:

- **Problem statement:** "Sort the array." This specifies the goal, but it gives no clue about how to achieve it.
- **Algorithm:** "Repeatedly select the smallest element in the unsorted portion of the array and move it to the front." This is concrete, step-by-step, and can be executed on a machine.

Another Example: Maximum of a List Problem: Find the largest number in a list of n numbers. Algorithm:

1. Initialize `largest` to the first element.
2. For each remaining element, compare it with `largest`.
3. If the current element is larger, update `largest`.
4. After scanning all elements, return `largest`.

This algorithm is finite, unambiguous, and terminates with the correct output.

Remark Notice that the algorithm is independent of the programming language used to implement it. This abstraction allows us to reason mathematically about its efficiency and correctness.

1.6 Why Do We Analyze Algorithms?

Even if two implementations solve the same problem correctly, their efficiency may vary dramatically. For example:

- Alice writes a Python program to compute the sum of all elements in a list.
- Bob writes the same algorithm in C++.

Although both programs compute the correct sum, Bob's implementation may run faster because C++ executes more efficiently at the machine level. Conversely, an inefficient algorithm in Python could outperform a clever C++ implementation for very small inputs.

This demonstrates that **implementation details and hardware differences can obscure the true efficiency of an algorithm**. To make meaningful comparisons, we abstract away these differences and focus on the **algorithm itself**.

1.6.1 Goals of Algorithm Analysis

Algorithm analysis helps answer:

- How does runtime grow with input size n? (scalability)
- Which algorithm is asymptotically faster for large n?
- How do time and space requirements trade off?
- How much memory does an algorithm require in addition to input storage?

By formalizing runtime analysis, we can identify the **dominant operations** and predict performance for very large inputs without running experiments.

1.6.2 Historical Note: Knuth and the RAM Model

Donald Knuth, in his seminal work *The Art of Computer Programming*, introduced a rigorous approach to analyzing algorithms. Key ideas include:

- Counting **primitive operations** instead of measuring wall-clock time.
- Using the **Random Access Machine (RAM) model**:
  - Each basic arithmetic or comparison operation takes exactly one unit of time.
  - Accessing any memory location takes one unit of time.
  - There are no hidden delays from caches, pipelines, or hardware specifics.
- This abstraction allows **language-independent** analysis and **mathematical reasoning** about algorithm efficiency.

Intuition Under the RAM model, an algorithm's runtime is essentially a function of input size n and the number of primitive steps it executes. This leads naturally to **asymptotic notation**, which describes growth rates while ignoring constants and lower-order terms.

Example (CLRS 1.2) The linear scan algorithm for finding the maximum in a list of n elements:

$$T(n) = n - 1$$

primitive comparisons. Even if implemented in Python, C++, or pseudocode, the **asymptotic behavior** remains $O(n)$.

Historical Context Knuth's focus on rigorous analysis led to:

- the development of precise asymptotic notation (Big-O, Big-Ω, Big-Θ),
- a framework for comparing algorithms independently of hardware, and
- a methodology for understanding scalability of algorithms that is still standard in modern textbooks, including CLRS and Kleinberg-Tardos.

This abstraction enables rigorous, machine-independent reasoning about efficiency.

1.7 Primitive Operations

Primitive operations are the basic steps whose number dominates runtime for large inputs. Examples:

- finding the maximum: comparisons,
- sorting: comparisons,
- matrix multiplication: additions and multiplications.

> *We abstract away machine-dependent details to reveal the algorithm's intrinsic complexity.*

1.8 Example: Finding the Maximum Element

1.8.1 Problem Statement

Given a list of n distinct numbers, find the largest.

1.8.2 Algorithm (Linear Scan)

1. Initialize `largest` to the first element.
2. For each remaining element, compare with `largest` and update if larger.

1.8.3 Analysis

Number of comparisons $= n - 1$ (exact). This is independent of language or hardware.

Alternative Approach: Sort the list and pick the last element. Sorting takes $O(n \log n)$ comparisons. Linear scan is faster asymptotically.

1.9 Asymptotic Notation

We formalize growth of runtime using:

- Big-O: upper bounds
- Big-Ω: lower bounds
- Big-Θ: tight bounds

Example: Finding the maximum: $T(n) = O(n)$, Sorting first: $T(n) = O(n \log n)$.

1.10 Concluding Remarks

Chapter 1 establishes the algorithmic mindset:

- think abstractly,
- reason mathematically,
- separate ideas from implementation.

> *Algorithms are not about code—they are about ideas.*

This foundation will support the rigorous analysis and design of algorithms in subsequent chapters.