Lecture 1 - Chpt 1, Chpt 2

Harley Caham Combest

Fa2025 CS4413 Lecture Notes – Mk1

Chapter 1: The Role of Algorithms in Computing

Historical Context. The concept of an *algorithm* is ancient. Euclid's algorithm (ca. 300 BCE) for the greatest common divisor is one of the earliest known procedures equipped with a formal proof of correctness. Medieval scholars in India and the Islamic world advanced methods for arithmetic and algebra; indeed, the word "algorithm" derives from the name of the 9th-century Persian mathematician al-Khwarizmi.

In the modern era, algorithms govern every aspect of computing—from genome sequencing to cryptography to internet routing. They form the backbone of both theoretical computer science and practical computation.

Definition of Algorithms (1.1).

Definition 1 (Algorithm)**.** An **algorithm** is a well-defined computational procedure that takes some value(s) as input and produces some value(s) as output in a finite amount of time.

Equivalently, an algorithm is a tool for solving a computational problem:

- The *problem statement* specifies the desired input-output relation.
- The *algorithm* provides an explicit procedure to realize it.

Definition 2 (Correctness)**.** An algorithm is **correct** if, for every input instance, it halts in finite time and outputs the correct solution. An algorithm is **incorrect** if there exists at least one input for which it fails to halt, or halts with an incorrect output.

Calculative Example: The Sorting Problem.

- **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** A permutation $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ such that

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}.$$

Example. Input $\langle 31, 41, 59, 26, 41, 58 \rangle$ yields output $\langle 26, 31, 41, 41, 58, 59 \rangle$.

Algorithms as a Technology (1.2). If computers were infinitely fast and memory free, efficiency would not matter. But correctness would still be required. In reality, both time and space are limited, making algorithm design a central technological concern.

Proposition 1 (Algorithms as Core Technology)**.** *Algorithms are as fundamental to computing as hardware, networking, or user interfaces. They are decisive in domains such as:*

- Genome sequencing — *dynamic programming for sequence alignment.*
- Internet routing — *shortest-path algorithms.*
- E-commerce security — *public-key cryptography and digital signatures.*
- Data science and machine learning — *clustering, regression, and optimization.*

Calculative: Insertion Sort vs. Merge Sort. Insertion sort runs in $\Theta(n^2)$ time in the worst case. Merge sort runs in $\Theta(n \log n)$ time. While insertion sort may be faster for small n, merge sort outpaces it dramatically as n grows.

Lesson. Hardware advances matter, but the choice of algorithm often determines whether a problem is tractable at scale.

Concluding Remarks. Algorithms formalize computation and efficiency. *Correctness* ensures reliability. *Asymptotic efficiency* ensures scalability. As we proceed, we will refine the mathematical tools—summations, asymptotics, recurrences— that allow precise analysis of algorithm performance.

Chapter 2: Getting Started

Historical Context. Sorting problems have been studied since antiquity in contexts ranging from orderly arrangements of data to the efficient execution of arithmetic. With the rise of computing, sorting became a fundamental operation, appearing as a subroutine in countless algorithms. Early mechanical computers, such as Hollerith's tabulating machine, were built to handle sorting tasks. Today, sorting is a laboratory for algorithm design: many techniques—incremental, divide-and-conquer, randomized—are first understood through sorting.

Insertion Sort (2.1).

Definition 3 (Sorting Problem)**.**

- **Input:** A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
- **Output:** A permutation $\langle a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)} \rangle$ such that

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}.$$

Algorithm 1 Insertion-Sort(A, n)

for $i \leftarrow 2$ **to** n **do**
    $key \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j > 0 \land A[j] > key$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    end while
    $A[j+1] \leftarrow key$
end for

Loop Invariants and Correctness.

Proposition 2 (Loop Invariant for Insertion Sort)**.** *At the start of each iteration of the* `for` *loop with index i, the subarray $A[1:i-1]$ consists of the elements originally in $A[1:i-1]$, but in sorted order.*

Proof.

1. *Initialization.* Prior to the first iteration ($i = 2$), the subarray $A[1]$ trivially contains a single element, and is therefore sorted.

2. *Maintenance.* Assume $A[1 : i-1]$ is sorted at the start of iteration i. The while loop shifts larger elements to the right until the correct position for $A[i]$ is found. Inserting *key* there yields that $A[1 : i]$ is sorted. Thus the invariant is preserved.

3. *Termination.* When the loop terminates, $i = n + 1$. By the invariant, $A[1 : n]$ is sorted. Therefore the algorithm is correct.

□

Analyzing Algorithms (2.2).

Definition 4 (Running Time)**.** The running time $T(n)$ of an algorithm on input of size n is the number of primitive instructions executed, measured as a function of n.

Analysis of Insertion Sort.

- *Best case:* Input already sorted. Each inner while-loop runs only once, giving $T(n) = \Theta(n)$.

- *Worst case:* Input in reverse order. Each $A[i]$ is compared against all of $A[1 : i-1]$, giving $T(n) = \Theta(n^2)$.

Algorithm 2 Merge-Sort(A, p, r)

```
if p ≥ r then
    return
end if
q ← ⌊(p + r)/2⌋
MERGE-SORT(A, p, q)
MERGE-SORT(A, q + 1, r)
MERGE(A, p, q, r)
```

Algorithm 3 Merge(A, p, q, r)

```
copy A[p : q] into array L; copy A[q + 1 : r] into array R
i ← 0;  j ← 0;  k ← p
while i < |L| ∧ j < |R| do
    if L[i] ≤ R[j] then
        A[k] ← L[i];  i ← i + 1
    else
        A[k] ← R[j];  j ← j + 1
    end if
    k ← k + 1
end while
while i < |L| do
    A[k] ← L[i];  i ← i + 1;  k ← k + 1
end while
while j < |R| do
    A[k] ← R[j];  j ← j + 1;  k ← k + 1
end while
```

Merge Sort (2.3). Recurrence for Merge Sort.

$$T(n) = \begin{cases} \Theta(1), & n = 1, \\ 2T(n/2) + \Theta(n), & n > 1. \end{cases}$$

By the Master Theorem, $T(n) = \Theta(n \log n)$.

Concluding Remarks. Insertion sort illustrates correctness proofs via loop invariants and quadratic-time analysis. Merge sort exemplifies divide-and-conquer design, yielding $O(n \log n)$ performance. Together, they demonstrate the principle: algorithmic efficiency, not just hardware speed, determines scalability.