# Data and Algorithm Analysis

## Chapter 2 — Getting Started

Thang Hoang

Department of Computer Science, Virginia Tech

# Outline

# Table of Contents

# Administrative Details

- **Syllabus**
- **Textbook:** Introduction to Algorithms (Fourth Edition), by Cormen, Leiserson, Rivest, and Stein
- **Website:** https://thanghoang.github.io/teaching/f23/cs4104/
- **Canvas**
  - Website
  - Office hours

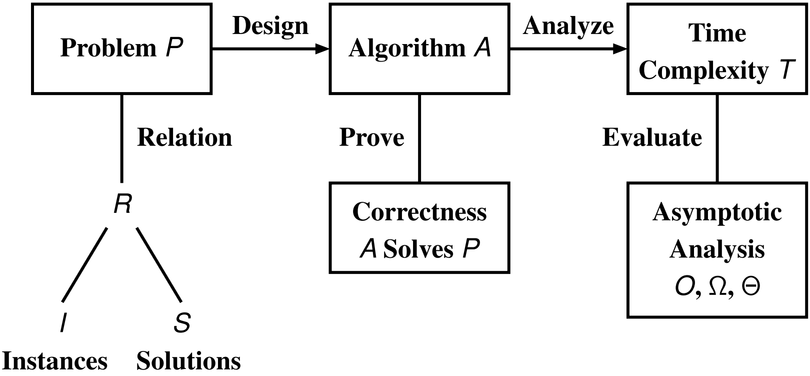
# Table of Contents

# Overview

- Use frameworks for describing and analyzing algorithms.
- Examine two sorting algorithms: insertion sort and merge sort.
- Learn how to present an algorithm with *pseudocode*.
- Understand asymptotic notation for running-time analysis.
- Learn “divide and conquer” technique with merge sort.

# Table of Contents

# Algorithms Solve Problems




$$R \subseteq I \times S$$

# Example — Sorting as a Formal Problem

SORTING

**Input**: Sequence $a_1, a_2, \ldots, a_n$ of integers.

**Output**: A permutation $b_1, b_2, \ldots, b_n$ of $a_1, a_2, \ldots, a_n$ such that

$$b_1 \leq b_2 \leq \cdots \leq b_{n-1} \leq b_n.$$

- For convenience of discussion, we often assume that the integers in the instance are distinct, though that is not strictly necessary.

# Example — Sorting

An input of SORTING as an array $A[1:n]$:

| $A[1]$ | $A[2]$ | $A[3]$ | $A[4]$ | $A[5]$ | $A[6]$ | $A[7]$ | $A[8]$ |
|---|---|---|---|---|---|---|---|
| 57 | 23 | 31 | 100 | 4 | 18 | 44 | 92 |

The output of SORTING as an array $B[1:n]$:

| $B[1]$ | $B[2]$ | $B[3]$ | $B[4]$ | $B[5]$ | $B[6]$ | $B[7]$ | $B[8]$ |
|---|---|---|---|---|---|---|---|
| 4 | 18 | 23 | 31 | 44 | 57 | 92 | 100 |

We will now **design algorithms** to solve SORTING.

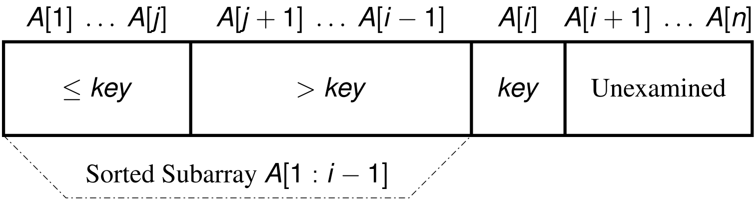
# Insertion Sort — A Simple Sorting Algorithm

Main idea is to build up a sorted array in place as follows:

- Proceed through $A[1:n]$ iteratively from left to right.
- Always maintain a sorted subarray on the left of $A[i]$.
- At $A[i]$, find the right place to insert $A[i]$ into the sorted subarray to its left.
- In the process, move larger integers to the right.

# Illustrating the Idea Behind Insertion Sort

$A[i] = key$ is the next integer to be inserted into the sorted subarray to its left.




We now express the INSERTION-SORT algorithm in **pseudocode.**

# CLRS Pseudocode

- Emphasis on human readability and comprehension
- Indentation for logical structure
- Keywords: `if`/`else`/`elseif`, `while`, `for`, `repeat`/`until`, `return`
- Assignment: $=$
- Comment: //
- Lines numbered for reference purposes
- Flexible semantics

# Pseudocode for INSERTION SORT

```
INSERTION-SORT(A, n)
    // A[1 : n] is an array of integers.
    // Returns a permutation of A in nondecreasing order.
    for i = 2 to n
        key = A[i]
        // Insert A[i] into the sorted subarray A[1 : i − 1].
        j = i − 1
        while j > 0 and A[j] > key
            A[j + 1] = A[j]
            j = j − 1
        A[j + 1] = key
    return A
```

# Algorithm Design Paradigms

- Incremental — Insertion Sort
- Divide and conquer — MergeSort — recursive
- Dynamic programming — Chapter 14
- Greedy — Chapter 15
- Randomized algorithms — Chapters 5 and 7
- Exhaustive search — an approach, later, for the Traveling Salesperson Problem

# Insertion Sort — Correctness

**Loop invariant:** At the start of iteration $i$ of the **for** loop (lines 3–10), the subarray $A[1 : i-1]$ is a sorted version of the original subarray $A[1 : i-1]$.

- **Initialization:** True before $i = 2$.
- **Maintenance:** True after loop body for $i$.
- **Termination:** Loop terminates with the array sorted when $i = n + 1$.

We conclude that the **for** loop actually sorts $A$.

# INSERTION SORT — Verify Loop Invariant

```
INSERTION-SORT(A, n)
    // A[1 : n] is an array of integers.
    // Returns a permutation of A in nondecreasing order.
    for i = 2 to n          // A[1 : i − 1] is already sorted
        key = A[i]
        // Insert A[i] into the sorted subarray A[1 : i − 1].
        j = i − 1
        while j > 0 and A[j] > key
            A[j + 1] = A[j]
            j = j − 1
        A[j + 1] = key
    return A
```

# Insertion Sort — Time Complexity Analysis

The final task is analyzing the time complexity of INSERTION-SORT.

This takes some discussion on analysis of algorithms, which is next.

# Analysis of Algorithms

- Why analyze?
  - Why not just code the algorithm, run it, and time it?
  - Analysis tells you how long the code takes to run on different settings, inputs, or programming language.
- Random-Access Machine (RAM) model
- What to analyze in an algorithm?
  - Time complexity
    - Worst case
    - Average case
  - Space complexity

# Worst Case Time Complexity

For algorithm $A$, *the worst case time complexity* $T_A(n)$ *of* $A$ is the **maximum** number of computational steps taken by algorithm $A$ on any instance of size $n$.

The parameter $n$ is often clear from context; we will discuss it in more detail when we study the theory of NP-completeness.

Worst-case time gives a guaranteed **upper bound** on the running time for any input.

# Average Case Time Complexity

Assume a probability distribution $f_n(I)$ on instances $I$ of size $n$.

The time for instance $I$ is $T(I)$.

For algorithm $A$, *the average case time complexity* $T(n)$ *of* $A$ is the **average** number of computational steps taken by algorithm $A$ on any instance of size $n$:

$$\begin{aligned} T(n) &= E[T(I) \mid \text{size of } I \text{ is } n] \\ &= \sum_{\text{size of } I \text{ is } n} f_n(I)T(I). \end{aligned}$$

# Order of Growth

Abstraction to ease analysis and focus on the important features.

Look only at the leading term of the running time formula

- Drop lower-order terms
- Ignore the constant coefficient in the leading term

Example: $an^2 + bn + c$

# Computing Worst Case $T(n)$

- Summation — for simple algorithms
- Recurrence — for divide and conquer algorithms

# INSERTION SORT — Just the **for** Loop

| *cost* | *times* | | |
|---|---|---|---|
| $c_3$ | $n$ | 3 | **for** $i = 2$ **to** $n$ |
| $c_4$ | $n-1$ | 4 |     $key = A[i]$ |
| $c_6$ | $n-1$ | 6 |     $j = i - 1$ |
| $c_7$ | $\sum_{i=2}^{n} t_i$ | 7 |     **while** $j > 0$ and $A[j] > key$ |
| $c_8$ | $\sum_{i=2}^{n} t_i - 1$ | 8 |         $A[j+1] = A[j]$ |
| $c_9$ | $\sum_{i=2}^{n} t_i - 1$ | 9 |         $j = j - 1$ |
| $c_{10}$ | $n-1$ | 10 |     $A[j+1] = key$ |

To get $T(n)$, need to multiply *cost* $\times$ *times* and sum it all up.

# Insertion Sort — Computing $T(n)$

$$
\begin{aligned}
T(n) &= c_3 n + c_4(n-1) + c_6(n-1) + c_7\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_8\left(\frac{n(n-1)}{2}\right) + c_9\left(\frac{n(n-1)}{2}\right) + c_{10}(n-1) \\
&= \left(\frac{c_7}{2} + \frac{c_8}{2} + \frac{c_9}{2}\right) n^2 \\
&\quad + \left(c_3 + c_4 + c_6 + \frac{c_7}{2} - \frac{c_8}{2} - \frac{c_9}{2} + c_{10}\right) n \\
&\quad - (c_4 + c_6 + c_7 + c_{10})
\end{aligned}
$$

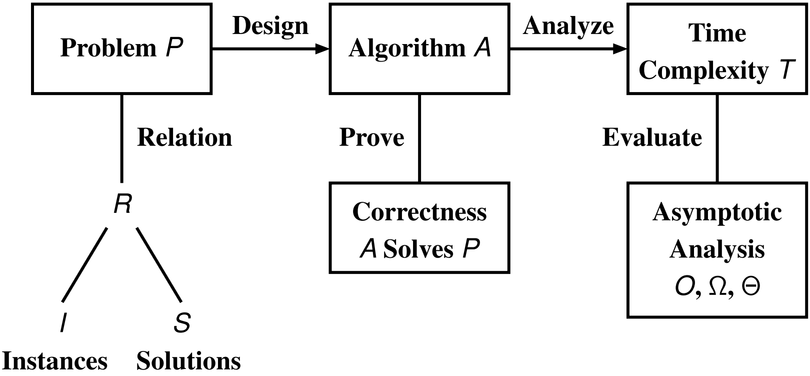
So, $T(n)$ is a **quadratic polynomial.**

# Asymptotics of $T(n)$

- Asymptotics are expressed using **asymptotic notation,** including $\Theta$.
- For INSERTION SORT, we have $T(n) = \Theta(n^2)$; more details on asymptotic notation in Chapter 3.
- That completes the analysis of INSERTION SORT.

# Table of Contents

# Algorithms Solve Problems




$$R \subseteq I \times S$$

# MergeSort — An Alternate Sorting Algorithm

- MergeSort: divide and conquer algorithm
- Recursive algorithm MERGE-SORT$(A, p, r)$
- Represents instance as an array $A[1:n]$
- Initial call MERGE-SORT$(A, 1, n)$
- **Divide and Conquer** — Splits array in two and recursively sorts each subarray
- Uses a MERGE algorithm to merge two sorted subarrays into one sorted array
- Pseudocode follows

# MERGE-SORT Algorithm

MERGE-SORT$(A, p, r)$

// $A[p : r] = a_p, a_{p+1}, \ldots, a_r$ is an array of integers.

// Returns an in-place permutation of $A[p : r]$

// in non-decreasing order.

**if** $p < r$

  $q = \lfloor (p + r)/2 \rfloor$

  MERGE-SORT$(A, p, q)$

  MERGE-SORT$(A, q + 1, r)$

  MERGE$(A, p, q, r)$

# MERGE Algorithm

MERGE($A, p, q, r$)
// $A[p:r] = a_p, a_{p+1}, \ldots, a_r$ is an array of integers
// with sorted subarrays $A[p:q]$ and $A[q+1:r]$.
// Returns an in-place permutation of $A[p:r]$
// in non-decreasing order.
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1:n_1+1]$ and $R[1:n_2+1]$ be new arrays
**for** $i = 1$ **to** $n_1$
    $L[i] = A[p+i-1]$
**for** $j = 1$ **to** $n_2$
    $R[j] = A[q+j]$

# MERGE Algorithm (Continued)

$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
**for** $k = p$ **to** $r$
    **if** $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    **else** $A[k] = R[j]$
        $j = j + 1$

# Time Complexity Analysis

- The MERGE algorithm (conquer step) takes linear time.
- The divide step takes at most linear time.
- So, the nonrecursion time at any call to MERGE-SORT is at most $c_1 n$, for some positive constant $c_1$.
- For a call to MERGE-SORT that does not result in recursion, we say the time complexity is some other positive constant $c_2$.

# Recurrences for Time Complexity

- MergeSort recurrence:

$$T(n) = \begin{cases} 2T(n/2) + c_1 n & n > 1 \\ c_2 & n = 1 \end{cases}$$

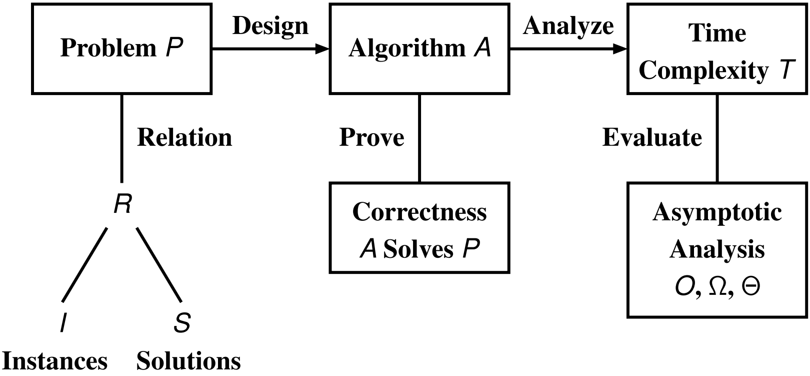
- Solving recurrence by recursion tree yields

$$T(n) = c_1 n \lg n + c_2 n.$$

- Details on recursion trees in Chapter 4.
- **Asymptotics —** $T(n) = \Theta(n \lg n)$

# Table of Contents

# Algorithms Solve Problems




$$R \subseteq I \times S$$

# Bonus Example — Traveling Salesperson Problem

TRAVELING SALESPERSON PROBLEM (TSP)

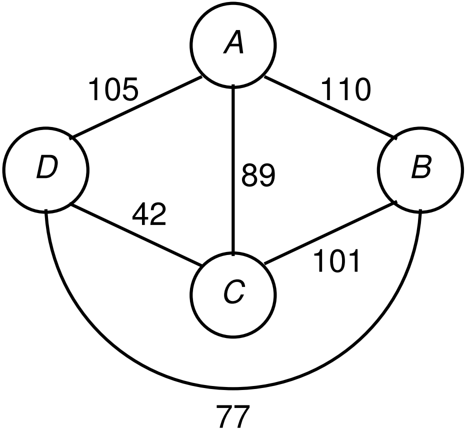
**Input**: Complete undirected graph $G = (V, E)$; weight function $w : E \to \mathbb{Z}$.

**Output**: A permutation $v_1, v_2, \ldots, v_n$ of $V$ such that

$$w(v_n, v_1) + \sum_{i=1}^{n-1} w(v_i, v_{i+1})$$

is minimized.

# Example — TSP Instance



# Algorithm Design for TSP

**Exhaustive search** — an approach to the TRAVELING SALESPERSON PROBLEM

- Given $G = (V, E)$ and $w : E \to \mathbb{Z}$
- Generate every permutation of $V$
- Compute weight of each
- Return permutation of minimum weight

# Pseudocode for TSP-EXHAUSTIVE

TSP-EXHAUSTIVE$(G, w)$
// $G = (V, E)$ is a complete undirected graph.
// $w : E \to \mathbb{Z}$ is an edge weight function.
// Returns a permutation of $V$ of minimum total weight.
$s^* = \infty$ // minimum weight so far
$\pi^* = \text{NIL}$ // permutation of weight $s^*$
**for** $\pi = v_1, v_2, \ldots, v_n$ a permutation of $V$
  $s = w(v_n, v_1) + \sum_{i=1}^{n-1} w(v_i, v_{i+1})$
  **if** $s < s^*$
    $s^* = s$
    $\pi^* = \pi$
**return** $\pi^*$

# Time Complexity

- **for** loop executed $n!$ times
- Line 7: summation takes $c_1 n$ operations for a positive constant $c_1$
- Generate next permutation in constant time $c_2$; see papers by Heap and Sedgewick under Resources
- Total time:

$$T(n) = n!(c_1 n + c_2) + c_3$$

- Asymptotics:

$$T(n) = O(n \cdot n!)$$
$$T(n) = \Omega(n \cdot n!)$$
$$T(n) = \Theta(n \cdot n!)$$

# Algorithms Solve Problems