Algorithms and Data Structures

3rd Semester Software Technology Engineering

Eduard Fekete

November 8, 2025

Contents

1 Introduction

Algorithms are formal, deterministic procedures transforming input to output through a finite sequence of well-defined steps. They encode computation as logic, not as syntax. What differentiates a good algorithm is not correctness alone but asymptotic efficiency under resource constraints: time, space, and often communication cost. In modern software systems, the majority of practical engineering failures originate not from correctness errors but from asymptotic ignorance. Understanding runtime growth is predictive power — it allows engineering before scaling breaks.

Data structures are engineered spatial encodings that give certain classes of algorithms structural leverage. Their entire purpose is to reduce entropy in access patterns: accelerating lookup, minimizing recomputation, avoiding redundancy, reducing cache misses, exploiting sparsity, and transforming unstructured data into shape. Lists, trees, heaps, graphs, hash tables, tries — these are not vocabulary items but complexity tradeoff mechanisms. Every structural choice rewrites the computational geometry of a problem.

The field is therefore not a set of memorized templates but a combinatorial design discipline. First principles create the ability to compute systematically: modeling problems as state machines, reducing problems to known primitives, selecting structure to match access semantics, and then proving bounds analytically.

2 Basics

2.1 Pseudocode conventions

- **Indentation** indicates block structure
- **Loops:** counter typically start at **1**
- `A[i : j]` contains `A[i]`, `A[i + 1]`, ..., `A[j]`
- Pass parameters to a procedure **by value**: the procedure receives its own copy of the parameters
- Return multiple values at once without bundling them into an object

2.1.1 Example:

```
SUM-ARRAY(A, n)
// this is a comment
1. sum = 0
2. for i = 1 to n
3.     sum = sum + A[i]
4. return sum
```

3 Time Complexity

3.1 Notations

Notation	Name	Meaning
O	Big O	Upper bound on growth rate
Ω	Big Omega	Lower bound on growth rate
Θ	Big Theta	Tight bound on growth rate
o	Little o	Strict upper bound on growth rate
ω	Little omega	Strict lower bound on growth rate

3.2 Definitions

$f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

$f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0$

$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

$f(n) = o(g(n))$ if for any positive constant $c > 0$, there exists a constant n_0 such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0$

$f(n) = \omega(g(n))$ if for any positive constant $c > 0$, there exists a constant n_0 such that $0 \leq c \cdot g(n) < f(n)$ for all $n \geq n_0$

4 Binary Search

4.1 Pseudocode

```
BINARY-SEARCH(A, n, target)
low = 1
high = n
while low <= high
    mid = (low + high) / 2
    if A[mid] == target
        return mid
    else if A[mid] < target
        low = mid + 1
    else
        high = mid - 1
return NOT-FOUND
```

4.2 Time complexity analysis

4.2.1 Average Case

- Each iteration halves the search space
- Number of iterations: $\log_2 n$
- Each iteration takes constant time $O(1)$
- Total time complexity: $O(\log n)$

4.2.2 Worst Case

- Target not found
- Number of iterations: $\log_2 n$
- Each iteration takes constant time $O(1)$
- Total time complexity: $O(\log n)$

4.2.3 Best Case

- Target found at the middle index
- Number of iterations: 1
- Each iteration takes constant time $O(1)$
- Total time complexity: $O(1)$

5 Recursion

5.1 Example: Factorial

```
FACTORIAL(n)
if n == 0
    return 1
else
    return n * FACTORIAL(n - 1)
```

5.2 Time complexity analysis

- Each call to `FACTORIAL` makes one recursive call with `n - 1`
- Number of calls: $n + 1$ (from `n` down to `0`)
- Each call takes constant time $O(1)$
- Total time complexity: $O(n)$

5.3 Space complexity analysis*

- Each call to `FACTORIAL` adds a new frame to the call stack
- Maximum depth of recursion: $n + 1$
- Each frame takes constant space $O(1)$
- Total space complexity: $O(n)$

6 Linked Lists

7 Stacks and Queues

8 Sets and Maps

9 Hashing and Hash Tables

10 Trees

10.1 Binary Trees

10.2 Binary Search Trees

10.3 Red-Black Trees

10.4 AVL Trees*

10.5 Huffman Encoding

11 Sorting Algorithms

11.1 Selection Sort

11.2 Insertion Sort

11.3 Merge Sort

11.4 Quick Sort*

11.5 Bubble Sort*

12 Divide and Conquer Algorithms

13 Master Theorem

13.1 Recurrence equation

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

- $a > 0$
- $b > 1$

13.2 Benchmark table

Benchmark	Name	Function
Non-recursive work	Driving function	$f(n) = n^d(log(n))^k$
Recursive work	Watershed function	$n^{log_b(a)}, w = log_b(a)$

13.3 Master theorem idea

- Which term dominates time complexity of the recurrence?
- Is there a polynomial gap between:

Type	Meaning
Non-recursive work	divide, combine
Recursive work	conquer

13.4 Definitions

Driving function: $f(n) = n^d(log(n))^k$

Watershed exponent: $w = log_b(a)$

13.5 Master theorem cases

Case no	Condition	Who dominates	Result for T(n)
1	$w > d$	Recursive - Driving function is smaller than watershed function by polynomial gap	$\Theta(n^w)$
2	$w = d$	Tie - Driving function and watershed function are asymptotically equal	$\Theta\big(n^d \log^{k+1} n\big)$
3	$w < d$	Non-recursive - Watershed function is smaller than driving function by polynomial gap	$\Theta\big(f(n)\big)$

13.6 Procedure

1. Identify a, b, $f(n)$
2. Compute watershed exponent w
3. Rewrite driving function $f(n)$
4. Compare w and d
5. Conclude $\rightarrow$ time complexity of recurrence equation