LECTURE 1 OF 12

Introduction, Algorithm Analysis, and Big-O Notation

Data Structures & Algorithms

Based on Robert Sedgewick & Kevin Wayne
Algorithms, 4th Edition (Addison-Wesley, 2011)

Chapters 1.1 & 1.4

Duration: 4 hours (with breaks)

Learning Objectives

By the end of this lecture, students will be able to:

1. **Articulate** why algorithms and their analysis are central to computer science and software engineering.
2. **Apply** the scientific method to measure and predict algorithm performance through empirical observation and mathematical modeling.
3. **Define and distinguish** Big-O, Big-Omega, and Big-Theta notation and use them to classify growth rates.
4. **Identify** the common order-of-growth families—constant, logarithmic, linear, linearithmic, quadratic, cubic, and exponential—and recognize which family a given algorithm belongs to.
5. **Analyze** code fragments (loops, nested loops, recursive calls) by counting operations and expressing the result in asymptotic notation.
6. **Evaluate** practical considerations including amortized cost, memory consumption, and situations where constant factors matter.

1. Why Study Algorithms?

Estimated time: 30 minutes — Sedgewick Ch. 1.1

1.1 What Is an Algorithm?

> **DEFINITION 1.1 — ALGORITHM**
>
> An **algorithm** is a finite, deterministic, and effective step-by-step procedure for solving a computational problem. It takes input, performs a sequence of well-defined operations, and produces output.

This definition, though straightforward, conceals enormous depth. The concept dates back to the 9th-century Persian mathematician Muhammad ibn Musa al-Khwarizmi, whose name gives us the word "algorithm." But the modern formalization came in the 1930s through the

work of Alan Turing, Alonzo Church, and Kurt Gödel, who demonstrated that some problems are fundamentally unsolvable by any algorithm (the *Halting Problem*), while others are solvable but may require impractically large amounts of time or space.

Sedgewick emphasizes a pragmatic viewpoint: an algorithm is inseparable from the **data structures** it operates on. Throughout this course, we study algorithms and data structures together because choosing the right data structure is often what makes an efficient algorithm possible.

1.2 Real-World Applications

Algorithms are not abstract academic exercises. They underpin virtually every piece of technology in modern life:

Domain	Algorithm / Technique	Impact
Internet routing	Dijkstra's shortest path	Billions of packets routed per second across the global Internet
Web search	PageRank, inverted indices	Organizes and retrieves information from billions of web pages
Genomics	String matching, dynamic programming	Sequence alignment for the Human Genome Project
Social networks	Graph algorithms (BFS, DFS)	Friend recommendations, community detection
Cryptography	RSA, number-theoretic algorithms	Secure e-commerce, banking, and communication
Logistics	Linear programming, network flow	FedEx routes millions of packages daily
Compression	Huffman coding, LZW	JPEG, MP3, ZIP reduce data sizes dramatically

1.3 Historical Context

The study of algorithm efficiency has a rich lineage that directly shapes how we write software today:

- **Euclid (~300 BCE)** — The Euclidean algorithm for computing the greatest common divisor is one of the oldest known algorithms still in use.
- **Babbage (1830s)** — Designed mechanical computing engines, raising the first questions about how many "steps" a computation requires.
- **Turing (1936)** — Formalized the notion of computation and proved limits of what can be computed.
- **Von Neumann (1945)** — Invented merge sort and established the first rigorous analysis of a sorting algorithm.
- **Knuth (1968–)** — Published *The Art of Computer Programming*, establishing algorithm analysis as a mathematical discipline.
- **Sedgewick (1983–)** — Extended Knuth's work with an emphasis on practical, implementable algorithms and empirical validation.

1.4 The Role of Algorithms in Computing

> **KEY CONCEPT**
>
> A fast algorithm on a slow computer will invariably outperform a slow algorithm on a fast computer, once the input is large enough. Hardware improvements give constant-factor speedups; algorithmic improvements can give polynomial or exponential speedups.

Consider a concrete scenario. Suppose Computer A executes 10 billion instructions per second and runs an insertion sort (proportional to n^2 operations). Computer B executes only 10 million instructions per second but runs merge sort (proportional to n log n operations). For n = 10 million items:

- **Computer A (insertion sort):** roughly $(10^7)^2 / 10^{10}$ = 10,000 seconds ≈ 2.8 hours
- **Computer B (merge sort):** roughly $10^7 \times 23 / 10^7 \approx 23$ seconds

The thousand-fold slower machine finishes 400 times faster because of the better algorithm. This is why algorithm analysis matters: it tells us what is fundamentally possible, independent of hardware trends.

Sedgewick frames the study of algorithms around three key questions:

1. **Correctness:** Does the algorithm solve the stated problem for all valid inputs?
2. **Efficiency:** How do the time and space requirements grow as the input size increases?

3. **Optimality:** Is there a fundamentally better approach, or is this algorithm the best we can do?

2. The Scientific Method for Algorithm Analysis

Estimated time: 45 minutes — Sedgewick Ch. 1.4

2.1 Overview of the Approach

Sedgewick advocates a **scientific approach** to understanding algorithm performance. Just as physicists build models of the natural world, computer scientists build models of computation:

1. **Observe** — Run the program on real data and measure its running time.
2. **Hypothesize** — Propose a mathematical model consistent with the observations.
3. **Predict** — Use the model to predict running time for new inputs.
4. **Verify** — Run the program on new data and compare with predictions.
5. **Validate** — Repeat until the model is satisfactory.

This cycle mirrors the scientific method. A model that consistently makes accurate predictions is a useful model, even if it does not capture every detail of the computation.

2.2 Empirical Analysis

2.2.1 The Doubling Experiment

One of Sedgewick's most powerful techniques is the **doubling experiment**. The procedure is simple:

1. Run the algorithm on an input of size N and record the time $T(N)$.
2. Double the input size and run again to get $T(2N)$.
3. Compute the ratio $T(2N) / T(N)$.
4. Repeat for increasing values of N.

EXAMPLE 2.1 — DOUBLING EXPERIMENT FOR THREE-SUM

The **Three-Sum problem** asks: given N integers, how many triples sum to zero? A brute-force solution uses three nested loops, giving a running time proportional to N^3.

```
function ThreeSum(a[0..n-1]):
    count ← 0
    for i ← 0 to n-3:
        for j ← i+1 to n-2:
            for k ← j+1 to n-1:
                if a[i] + a[j] + a[k] = 0:
                    count ← count + 1
    return count
```

Doubling experiment results (hypothetical):

N	Time (sec)	Ratio $T(2N)/T(N)$
250	0.0	—
500	0.1	—
1,000	0.8	8.0
2,000	6.4	8.0
4,000	51.1	8.0
8,000	410.8	8.0

KEY CONCEPT — INTERPRETING THE RATIO

If $T(N) \approx a\, N^b$, then $T(2N)/T(N) \approx 2^b$. A ratio of approximately $8 = 2^3$ implies $b \approx 3$, confirming the cubic running time. A ratio of 4 would suggest quadratic, 2 would suggest linear, and so on.

2.2.2 Log-Log Plots

Taking logarithms of both sides of $T(N) = a\, N^b$ gives:

$$\log T(N) = b \log N + \log a$$

This is a straight line on a log-log plot with slope b. By plotting empirical data on log-log axes, we can visually estimate the exponent. This technique is standard in the natural sciences and is equally powerful in algorithm analysis.

2.3 Mathematical Models

2.3.1 Counting Operations

Knuth's insight was that we can derive a **mathematical expression** for the running time by identifying the most frequently executed operations and counting them precisely.

For the Three-Sum example, the number of times the innermost comparison is executed is exactly:

$$C(N) = N \text{ choose } 3 = N(N-1)(N-2) / 6$$

For large N, this is approximately $N^3/6$.

2.3.2 Cost Model

> **DEFINITION 2.1 — COST MODEL**
>
> A **cost model** specifies the basic operations that we count to measure an algorithm's running time. For sorting algorithms, the cost model typically counts *comparisons* and *exchanges*. For search algorithms, it counts *comparisons* or *array accesses*.

The cost model abstracts away machine-dependent details (clock speed, cache effects, instruction pipelining) and focuses on the operations that dominate the running time. This abstraction is what makes algorithm analysis portable across different hardware.

2.4 Tilde Approximations

> **DEFINITION 2.2 — TILDE NOTATION**
>
> We write $f(N) \sim g(N)$ to indicate that $f(N) / g(N) \to 1$ as $N \to \infty$. In other words, $g(N)$ approximates $f(N)$ with vanishing relative error.

Sedgewick's tilde notation captures the *leading term* of a function, discarding lower-order terms. For instance:

- $N^3/6 - N^2/2 + N/3 \sim N^3/6$
- $3N^2 + 7N + 42 \sim 3N^2$
- $N \log N + N \sim N \log N$

Note that tilde notation **preserves the constant coefficient** of the leading term. This is more precise than Big-O notation, which discards constants. The tilde approach is useful when we want to make specific quantitative predictions, as in the doubling experiment.

2.5 Order-of-Growth Classifications

Despite the diversity of algorithms, the vast majority fall into a small number of growth-rate families. Sedgewick identifies the following standard classifications:

Order of Growth	Name	Typical Code Pattern	Example	$T(2N)/T(N)$
1	Constant	Single statement	Array index lookup	1
$\log N$	Logarithmic	Divide in half	Binary search	~1
N	Linear	Single loop	Find the maximum	2
$N \log N$	Linearithmic	Divide and conquer	Merge sort	~2
N^2	Quadratic	Double loop	Check all pairs	4
N^3	Cubic	Triple loop	Check all triples	8
2^N	Exponential	Exhaustive search	All subsets	$T(N)$

Table 2.1: Standard order-of-growth classifications (Sedgewick, Table 1.4.7)

IMPORTANT DISTINCTION

Algorithms with exponential growth rates are considered **intractable** for large inputs. A problem of size $N = 40$ with a 2^N algorithm requires over 1 trillion operations. Even at a billion operations per second, this takes over 18 minutes. At $N = 100$, it would take longer than the age of the universe.

To build intuition for how dramatically these growth rates differ, consider the time required for each, assuming one operation takes 1 nanosecond:

Growth Rate	$N = 10$	$N = 100$	$N = 1{,}000$	$N = 1{,}000{,}000$
1	1 ns	1 ns	1 ns	1 ns
$\log N$	3.3 ns	6.6 ns	10 ns	20 ns
N	10 ns	100 ns	1 µs	1 ms
$N \log N$	33 ns	664 ns	10 µs	20 ms
N^2	100 ns	10 µs	1 ms	16.7 min
N^3	1 µs	1 ms	1 sec	31.7 years
2^N	1 µs	4×10^{13} years	unfathomable	

Table 2.2: Concrete running times at 1 nanosecond per operation

3. Big-O, Big-Omega, and Big-Theta Notation

Estimated time: 60 minutes — Supplementary to Sedgewick Ch. 1.4

3.1 Motivation

Tilde notation (Section 2.4) gives us precise leading-term approximations. But in many contexts, we want to make *looser* statements—for instance, "the running time grows no faster than quadratically." This is the role of **asymptotic notation**, the language of algorithm analysis.

The three principal notations—O, Ω, and Θ—correspond to upper bounds, lower bounds, and tight bounds, respectively. Together they form a complete vocabulary for describing how functions grow.

3.2 Big-O Notation (Upper Bound)

DEFINITION 3.1 — BIG-O

We write $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that:

$$0 \le f(n) \le c \cdot g(n) \quad \text{for all } n \ge n_0$$

Informally: f grows **no faster than** g, up to a constant factor, for sufficiently large n.

EXAMPLE 3.1 — PROVING $f(N) = 3N^2 + 7N + 4$ IS $O(N^2)$

We need constants c and n_0 such that $3n^2 + 7n + 4 \le c \cdot n^2$ for all $n \ge n_0$.

For $n \ge 1$: $7n \le 7n^2$ and $4 \le 4n^2$

So: $3n^2 + 7n + 4 \le 3n^2 + 7n^2 + 4n^2 = 14n^2$

Choose $c = 14$ and $n_0 = 1$. □

3.2.1 Common Misunderstandings

> **WARNING**
>
> Big-O provides an **upper bound**, not a tight bound. Saying $f(n) = O(n^3)$ for a quadratic function is technically correct but imprecise. In practice, always aim for the **tightest** Big-O bound you can prove.

A common error among students is to treat Big-O as meaning "equals" or "is approximately." It is an *inequality*: it says the function is bounded above. The statement $5n = O(n^2)$ is true, but it wastes information.

3.3 Big-Omega Notation (Lower Bound)

> **DEFINITION 3.2 — BIG-OMEGA**
>
> We write $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that:
>
> $$0 \leq c \cdot g(n) \leq f(n) \quad \text{for all } n \geq n_0$$
>
> Informally: f grows **at least as fast as** g, up to a constant factor.

Big-Omega is the mirror image of Big-O. While Big-O sets a ceiling, Big-Omega sets a floor. Lower bounds are particularly important when proving that no algorithm for a given problem can be faster than a certain growth rate (e.g., comparison-based sorting requires $\Omega(n \log n)$ comparisons).

3.4 Big-Theta Notation (Tight Bound)

> **DEFINITION 3.3 — BIG-THETA**
>
> We write $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ **and** $f(n) = \Omega(g(n))$. Equivalently, there exist positive constants c_1, c_2, and n_0 such that:
>
> $$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0$$
>
> Informally: f grows **at the same rate as** g, up to constant factors.

Big-Theta is the strongest of the three notations. When we say an algorithm runs in $\Theta(n \log n)$, we are saying it is both $O(n \log n)$ and $\Omega(n \log n)$—we have pinned down the growth rate precisely.

3.5 Summary of Asymptotic Notations

Notation	Intuitive Meaning	Analogy	Usage
$f = O(g)$	f grows no faster than g	$f \leq g$ (asymptotically)	Upper bound on worst case
$f = \Omega(g)$	f grows at least as fast as g	$f \geq g$ (asymptotically)	Lower bound / best case guarantee
$f = \Theta(g)$	f grows at the same rate as g	$f = g$ (asymptotically)	Tight / exact characterization

3.6 Common Growth Rates in Detail

3.6.1 O(1) — Constant Time

The running time does not depend on the input size. Examples: accessing an array element by index, pushing onto a stack (non-resizing), computing a hash function with fixed-length keys.

3.6.2 O(log n) — Logarithmic Time

The running time grows proportionally to the number of times the input can be halved. This is the hallmark of **divide-and-conquer** strategies that discard half the input at each step. Binary search is the canonical example. Note: in algorithm analysis, the base of the logarithm does not matter because $\log_a n = \log_b n \,/\, \log_b a$, and the divisor is a constant absorbed by the O notation. By convention, we write $\log n$ and assume base 2 unless stated otherwise.

3.6.3 O(n) — Linear Time

The running time is proportional to the input size. A single pass through the data falls in this category. Examples: finding the maximum, computing a sum, linear search.

3.6.4 O(n log n) — Linearithmic Time

Slightly slower than linear, this is the growth rate of the best comparison-based sorting algorithms (merge sort, heapsort) and many divide-and-conquer algorithms. It arises when we split a problem in half and then combine the results in linear time, recurring to depth log n.

3.6.5 O(n^2) — Quadratic Time

Typical of algorithms that examine all pairs of input elements. Examples: insertion sort, selection sort, brute-force closest pair.

3.6.6 O(n^3) — Cubic Time

Typical of algorithms that examine all triples. Example: the brute-force Three-Sum (Section 2.2) and naive matrix multiplication.

3.6.7 O(2^n) — Exponential Time

Arises when an algorithm must consider all subsets of the input. This is characteristic of brute-force solutions to NP-hard problems like the Traveling Salesman Problem or the subset-sum problem. Exponential algorithms are infeasible for all but the smallest inputs.

3.7 How to Determine the Order of Growth

Given a function $f(n)$, follow these steps to determine its asymptotic growth rate:

1. **Identify the dominant term.** This is the term that grows fastest as $n \to \infty$. In a polynomial, it is the term with the highest exponent.
2. **Drop constant coefficients.** The constant factor does not affect the growth rate classification.
3. **Drop lower-order terms.** They become negligible relative to the dominant term.

EXAMPLE 3.2 — SIMPLIFYING TO BIG-THETA

Function	Dominant Term	Growth Rate
$5n^3 + 2n^2 + 100n + 17$	$5n^3$	$\Theta(n^3)$
$n \log n + 1000n$	$n \log n$	$\Theta(n \log n)$
$3 \cdot 2^n + n^5$	$3 \cdot 2^n$	$\Theta(2^n)$
$\log(n^2) + \log n$	$2 \log n$	$\Theta(\log n)$
42	42	$\Theta(1)$

3.8 Useful Asymptotic Properties

THEOREM 3.1 — TRANSITIVITY

If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

The same holds for Ω and Θ.

THEOREM 3.2 — SUM RULE

If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then:

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$$

In words: the growth rate of a sum is determined by the faster-growing term.

THEOREM 3.3 — PRODUCT RULE

If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then:

$$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

This is why nested loops multiply: an $O(n)$ outer loop containing an $O(n)$ inner loop gives $O(n^2)$.

4. Analyzing Code

Estimated time: 60 minutes — Sedgewick Ch. 1.4

4.1 General Methodology

To analyze the running time of an algorithm expressed in code (or pseudocode), follow this systematic procedure:

1. **Choose a cost model:** Identify the basic operation that dominates the running time (comparison, addition, array access, etc.).
2. **Identify the input size parameter:** This is typically called n and represents the number of elements, characters, nodes, etc.
3. **Count the number of times the basic operation is executed** as a function of n.
4. **Simplify** using tilde notation or asymptotic notation.

4.2 Simple Loops

EXAMPLE 4.1 — SINGLE LOOP (LINEAR)

```
1  function Sum(a[0..n-1]):
2      total ← 0                  // executed 1 time
3      for i ← 0 to n-1:          // loop runs n times
4          total ← total + a[i]   // executed n times
5      return total                 // executed 1 time
```

Analysis: The addition on line 4 (our cost model operation) executes exactly n times. The running time is $\Theta(n)$.

EXAMPLE 4.2 — LOOP WITH HALVING (LOGARITHMIC)

```
function CountHalves(n):
    count ← 0
    while n > 1:
        n ← floor(n / 2)
        count ← count + 1
    return count
```

Analysis: The value of n is halved in each iteration. Starting from n, we can halve $\lfloor \log_2 n \rfloor$ times before reaching 1. The loop body executes $\Theta(\log n)$ times.

EXAMPLE 4.3 — LOOP WITH DOUBLING INDEX (LOGARITHMIC)

```
function CountDoubles(n):
    count ← 0
    i ← 1
    while i < n:
        count ← count + 1
        i ← i * 2
    return count
```

Analysis: The variable i takes on values 1, 2, 4, 8, ..., 2^k where $2^k < n$. The loop runs $\lceil \log_2 n \rceil$ times. Running time: $\Theta(\log n)$.

4.3 Nested Loops

EXAMPLE 4.4 — INDEPENDENT NESTED LOOPS (QUADRATIC)

```
function AllPairs(n):
    count ← 0
    for i ← 0 to n-1:            // n iterations
        for j ← 0 to n-1:        // n iterations (each time)
            count ← count + 1    // executed n × n times
    return count
```

Analysis: The inner loop runs n times for each of the n iterations of the outer loop. By the Product Rule: total operations = $n \times n = n^2$. Running time: $\Theta(n^2)$.

EXAMPLE 4.5 — DEPENDENT (TRIANGULAR) NESTED LOOPS (QUADRATIC)

```
function UpperTriangle(n):
    count ← 0
    for i ← 0 to n-1:
        for j ← i+1 to n-1:
            count ← count + 1
    return count
```

Analysis: The inner loop runs $(n-1)$, $(n-2)$, ..., 1, 0 times as i goes from 0 to $n-1$. The total is:

$$\sum_{i=0}^{n-1} (n-1-i) = (n-1) + (n-2) + \ldots + 1 + 0 = n(n-1)/2 \sim n^2/2$$

Running time: $\Theta(n^2)$. Note the tilde approximation is $n^2/2$, which is more precise than the Θ bound.

EXAMPLE 4.6 — TRIPLE NESTED LOOPS (CUBIC)

```
1  function AllTriples(a[0..n-1]):
2      count ← 0
3      for i ← 0 to n-3:
4          for j ← i+1 to n-2:
5              for k ← j+1 to n-1:
6                  if a[i] + a[j] + a[k] = 0:
7                      count ← count + 1
8      return count
```

Analysis: The number of times line 6 executes is the number of ways to choose 3 items from n:

$$C(n, 3) = n! \,/\, (3!(n-3)!) = n(n-1)(n-2) \,/\, 6 \sim n^3/6$$

Running time: $\Theta(n^3)$.

4.4 Sequential Composition

EXAMPLE 4.7 — SEQUENTIAL CODE BLOCKS

```
function ProcessData(a[0..n-1]):
    // Phase 1: Linear scan
    for i ← 0 to n-1:              // O(n)
        process(a[i])

    // Phase 2: All-pairs comparison
    for i ← 0 to n-1:              // O(n^2)
        for j ← i+1 to n-1:
            compare(a[i], a[j])

    // Phase 3: Constant work
    return result                      // O(1)
```

Analysis: By the Sum Rule, the total running time is determined by the dominant phase:

$$O(n) + O(n^2) + O(1) = O(n^2)$$

4.5 Analyzing Recursive Algorithms

Recursive algorithms are analyzed by writing a **recurrence relation** that expresses the running time of the algorithm on input of size n in terms of its running time on smaller inputs.

EXAMPLE 4.8 — BINARY SEARCH (LOGARITHMIC)

```
function BinarySearch(a[lo..hi], key):
    if lo > hi:
        return -1                    // not found
    mid ← lo + (hi - lo) / 2
    if key < a[mid]:
        return BinarySearch(a[lo..mid-1], key)
    else if key > a[mid]:
        return BinarySearch(a[mid+1..hi], key)
    else:
        return mid                   // found
```

Recurrence:

$$T(n) = T(n/2) + O(1), \quad T(1) = O(1)$$

Solution: Each recursive call halves the problem. After k calls, we have $n/2^k = 1$, so $k = \log_2 n$. Each call does O(1) work. Total: $T(n) = O(\log n)$.

EXAMPLE 4.9 — MERGE SORT (LINEARITHMIC)

```
function MergeSort(a[0..n-1]):
    if n ≤ 1:
        return a
    mid ← n / 2
    left  ← MergeSort(a[0..mid-1])
    right ← MergeSort(a[mid..n-1])
    return Merge(left, right)          // Merge is O(n)
```

Recurrence:

$$T(n) = 2\,T(n/2) + O(n), \quad T(1) = O(1)$$

Solution (by expansion):

- Level 0: 1 problem of size n → $O(n)$ merge work
- Level 1: 2 problems of size $n/2$ → $O(n)$ merge work
- Level 2: 4 problems of size $n/4$ → $O(n)$ merge work
- ... (log n levels total)
- Level log n: n problems of size 1 → $O(n)$ base-case work

Total: $O(n)$ work per level × log n levels = $O(n \log n)$.

More precisely, the number of comparisons is $\sim n \log_2 n$.

EXAMPLE 4.10 — NAIVE FIBONACCI (EXPONENTIAL)

```
function Fib(n):
    if n ≤ 1:
        return n
    return Fib(n-1) + Fib(n-2)
```

Recurrence:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Analysis: This is itself the Fibonacci recurrence. $T(n)$ grows as $\Theta(\varphi^n)$ where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$ is the golden ratio. This is exponential growth: $T(50) \approx 2 \times 10^{10}$ calls. This example powerfully illustrates how a "simple" recursive algorithm can be catastrophically slow, motivating the dynamic programming techniques we will study in later lectures.

4.6 Best Case, Worst Case, and Average Case

For most algorithms, the running time depends not just on the *size* of the input but also on the *particular input*:

DEFINITION 4.1 — BEST, WORST, AND AVERAGE CASE

- **Best case:** The minimum running time over all possible inputs of size n.
- **Worst case:** The maximum running time over all possible inputs of size n.
- **Average case:** The expected running time, averaged over all inputs of size n under a specified probability distribution (often the uniform distribution over all permutations).

EXAMPLE 4.11 — LINEAR SEARCH

```
function LinearSearch(a[0..n-1], key):
    for i ← 0 to n-1:
        if a[i] = key:
            return i
    return -1
```

Best case: The key is the first element. $\Theta(1)$ comparisons.

Worst case: The key is the last element or not present. $\Theta(n)$ comparisons.

Average case (assuming key is present and equally likely to be at any position): Expected number of comparisons = $(1 + 2 + \ldots + n) / n = (n+1)/2 = \Theta(n)$.

KEY CONCEPT

In practice, **worst-case analysis** is the most commonly used because it provides a guarantee: the algorithm will *never* be slower than the worst-case bound, regardless of input. Average-case analysis is useful but requires assumptions about input distributions that may not hold in practice.

EXAMPLE 4.12 — INSERTION SORT (COMPREHENSIVE ANALYSIS)

```
1  function InsertionSort(a[0..n-1]):
2      for i ← 1 to n-1:
3          key ← a[i]
4          j ← i - 1
5          while j ≥ 0 and a[j] > key:
6              a[j+1] ← a[j]
7              j ← j - 1
8          a[j+1] ← key
```

Best case (already sorted): The `while` condition on line 5 fails immediately for every i. Inner loop does 0 swaps. Comparisons: $n-1$. Running time: $\Theta(n)$.

Worst case (reverse sorted): For each i, the inner loop runs i times. Total comparisons: $1 + 2 + \ldots + (n-1) = n(n-1)/2 \sim n^2/2$. Running time: $\Theta(n^2)$.

Average case (random permutation): Each element is expected to move half the distance to its correct position. Total comparisons $\sim n^2/4$. Running time: $\Theta(n^2)$.

4.7 Summary of Analysis Patterns

Code Pattern	Running Time	Reasoning
Single statement / fixed operations	$O(1)$	No dependence on n
Loop: `i = 1; while i < n: i = i*2`	$O(\log n)$	Index doubles each iteration
Loop: `for i = 0 to n-1`	$O(n)$	Linear scan
Two sequential $O(n)$ loops	$O(n)$	Sum Rule: $O(n) + O(n) = O(n)$
Nested: outer $O(n)$, inner $O(n)$	$O(n^2)$	Product Rule
Nested: outer $O(n)$, inner $O(\log n)$	$O(n \log n)$	Product Rule
Three nested $O(n)$ loops	$O(n^3)$	Product Rule applied twice
$T(n) = T(n/2) + O(1)$	$O(\log n)$	Binary search pattern
$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$	Merge sort pattern
$T(n) = 2T(n/2) + O(1)$	$O(n)$	Binary tree traversal pattern
$T(n) = T(n-1) + O(n)$	$O(n^2)$	Selection sort pattern
$T(n) = T(n-1) + T(n-2) + O(1)$	$O(2^n)$	Naive Fibonacci pattern

Table 4.1: Common code patterns and their growth rates

5. Practical Considerations

Estimated time: 30 minutes — Sedgewick Ch. 1.4

5.1 Amortized Analysis

DEFINITION 5.1 — AMORTIZED COST

Amortized analysis determines the average cost per operation over a worst-case sequence of operations, even though individual operations may occasionally be expensive. The key insight is that expensive operations are infrequent enough to be "paid for" by the cheap operations.

EXAMPLE 5.1 — DYNAMIC ARRAY (RESIZING ARRAY)

Consider a dynamic array that doubles its capacity when full. Most insertions (appends) take O(1). But when the array is full, a resize operation copies all n elements to a new array of size $2n$, costing O(n).

Analysis: Starting with an array of size 1, after n insertions we have performed resize operations at sizes 1, 2, 4, 8, ..., n. The total cost of all resizes is:

$$1 + 2 + 4 + 8 + \ldots + n = 2n - 1$$

Adding the n constant-time insertions, the total cost for n operations is at most $3n$. Therefore:

$$\text{Amortized cost per insertion} = 3n / n = 3 = O(1)$$

Each insertion costs O(1) *amortized*, even though a single insertion can cost O(n) in the worst case. This is the strategy used by virtually all dynamic array implementations (ArrayList, Python list, std::vector).

KEY CONCEPT

Amortized O(1) is **not** the same as average-case O(1). Amortized analysis applies to the *worst case* over a sequence of operations, with no probabilistic assumptions. Average-case analysis requires a probability distribution over inputs.

5.2 Memory Usage Analysis

Time is not the only resource. **Memory** (space) consumption is equally important, especially for large-scale applications and embedded systems. Sedgewick's approach applies the same order-of-growth analysis to space.

5.2.1 Typical Memory Costs

Data Type	Typical Memory (bytes)
boolean	1
byte / char	1–2
int (32-bit)	4
float (32-bit)	4
long / double (64-bit)	8
Object reference / pointer	4–8
Array of n ints	$\sim 4n$ + overhead
Array of n objects	$\sim 8n$ (references) + object sizes
Linked list node	~16–40 (data + pointer + overhead)

5.2.2 In-Place vs. Not-In-Place Algorithms

DEFINITION 5.2 — IN-PLACE ALGORITHM

An algorithm is **in-place** if it uses at most O(1) extra memory beyond the input itself (or, in some definitions, O(log n) for recursion stack).

Algorithm	Time	Extra Space	In-Place?
Insertion sort	$O(n^2)$	O(1)	Yes
Merge sort	$O(n \log n)$	$O(n)$	No
Heapsort	$O(n \log n)$	O(1)	Yes
Binary search	$O(\log n)$	O(1) iterative / $O(\log n)$ recursive	Yes

5.3 When Constant Factors Matter

Asymptotic analysis is invaluable for understanding scalability, but it discards constant factors and lower-order terms. In practice, these details sometimes matter:

1. **Small inputs:** For small n (say, $n < 50$), an $O(n^2)$ algorithm with a small constant can beat an $O(n \log n)$ algorithm with a large constant. This is why hybrid sorting algorithms (like Timsort) switch to insertion sort for small subarrays.
2. **Cache effects:** An algorithm that accesses memory sequentially (good *spatial locality*) can be many times faster than one that jumps around, even if both have the same asymptotic complexity. Array-based structures are often faster than linked lists in practice, despite similar Big-O bounds, due to cache behavior.
3. **System-level overhead:** Function call overhead, memory allocation, garbage collection, and virtual method dispatch all contribute constant-factor costs that can dominate for small inputs or tight inner loops.
4. **Equal asymptotic classes:** When comparing two algorithms in the same complexity class (e.g., two $O(n \log n)$ sorts), constants and lower-order terms determine which is faster in practice. Quicksort (average $\sim 1.39 n \log n$) tends to outperform merge sort (always $\sim n \log n$) in practice due to lower constant factors and better cache behavior, despite quicksort's $O(n^2)$ worst case.

KEY CONCEPT — THE PRACTICAL ALGORITHM DESIGNER'S MOTTO

Use asymptotic analysis to choose the right *algorithm class*, then use empirical measurement and profiling to tune constants, optimize cache behavior, and validate predictions on real hardware.

6. Summary and Key Takeaways

6.1 Core Ideas

1. **Algorithms are fundamental.** They determine what is computationally feasible, independent of hardware. A better algorithm beats a faster computer.
2. **The scientific method applies.** Observe, hypothesize, predict, verify. The doubling experiment is a powerful empirical tool for discovering growth rates.
3. **Tilde notation** (~) captures the leading term with its constant, giving quantitative predictions. **Asymptotic notation** (O, Ω, Θ) classifies growth rates, discarding constants.
4. **Big-O is an upper bound** ("at most this fast-growing"), **Big-Omega is a lower bound** ("at least this fast-growing"), and **Big-Theta is a tight bound** ("grows at exactly this rate").
5. **Most algorithms fall into a few growth-rate families:** constant, logarithmic, linear, linearithmic, quadratic, cubic, or exponential.
6. **Analyzing code** involves identifying the basic operation, counting executions, and simplifying. Nested loops multiply; sequential blocks take the max; recursion requires solving a recurrence.
7. **Worst-case analysis** provides guarantees. **Amortized analysis** gives per-operation costs averaged over sequences. Both are worst-case techniques (no probability assumptions).
8. **Memory matters too.** Space complexity uses the same asymptotic framework as time complexity.
9. **Constants matter in practice.** Asymptotic analysis guides algorithm selection; empirical testing validates real-world performance.

6.2 Looking Ahead

In Lecture 2, we will begin studying **fundamental data structures**—stacks, queues, and linked lists—using the analytical framework established today. We will see how the choice of data structure directly affects the efficiency of algorithms, and we will analyze the performance of each operation on each structure.

The analytical skills from this lecture will be used in every subsequent lecture. Practice them until they become second nature.

7. Practice Problems

Problem 1: Asymptotic Classification

Classify each function using Θ notation (give the tightest bound):

a. $7n^2 + 3n + 12$

b. $4n \log_2 n + 100n$

c. 2^{n+3}

d. $\log(n^3)$

e. $n(n-1)(n-2) / 6$

Solution

a. $\Theta(n^2)$ — The $7n^2$ term dominates.

b. $\Theta(n \log n)$ — $n \log n$ grows faster than n, so $4n \log n$ dominates.

c. $\Theta(2^n)$ — $2^{n+3} = 8 \cdot 2^n$, and the constant 8 is absorbed.

d. $\Theta(\log n)$ — $\log(n^3) = 3 \log n$, and the constant 3 is absorbed.

e. $\Theta(n^3)$ — Expanding gives $n^3/6 - n^2/2 + n/3$. The $n^3/6$ term dominates.

Problem 2: Proving Big-O

Prove formally that $f(n) = 5n^2 + 3n$ is $O(n^2)$ by finding explicit constants c and n_0.

Solution

We need: $5n^2 + 3n \leq c \cdot n^2$ for all $n \geq n_0$.

For $n \geq 1$: $3n \leq 3n^2$.

Therefore: $5n^2 + 3n \leq 5n^2 + 3n^2 = 8n^2$.

Choose $c = 8$ and $n_0 = 1$. For all $n \geq 1$, $f(n) \leq 8n^2$. $\square$

Problem 3: Code Analysis

Determine the running time of the following code fragment in terms of n:

```
count ← 0
for i ← 1 to n:
    j ← 1
    while j < n:
        count ← count + 1
        j ← j * 2
```

Solution

The outer `for` loop runs n times. For each iteration of the outer loop, the inner `while` loop starts with $j = 1$ and doubles j until $j \geq n$. The inner loop runs $\lceil \log_2 n \rceil$ times per outer iteration.

By the Product Rule: total = $n \times \lceil \log_2 n \rceil = \Theta(n \log n)$.

Problem 4: Code Analysis with Dependent Loops

Determine the running time:

```
count ← 0
for i ← 0 to n-1:
    for j ← 0 to i:
        for k ← 0 to j:
            count ← count + 1
```

Solution

The number of times line 5 executes is:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i} (j+1) = \sum_{i=0}^{n-1} (i+1)(i+2)/2$$

The inner double sum for fixed i is $\sum_{j=0}^{i}(j+1) = 1 + 2 + \ldots + (i+1) = (i+1)(i+2)/2$.

Summing over i: $\sum_{i=0}^{n-1} (i+1)(i+2)/2 \sim n^3/6$ as $n \to \infty$.

Running time: $\Theta(n^3)$.

Problem 5: Recursive Analysis

What is the running time of the following recursive function?

```
function Mystery(n):
    if n ≤ 1:
        return 1
    return Mystery(n/3) + Mystery(n/3) + Mystery(n/3)
```

Solution

Recurrence: $T(n) = 3\,T(n/3) + O(1)$, with $T(1) = O(1)$.

Solving by expansion:

- Level 0: 1 call, O(1) non-recursive work
- Level 1: 3 calls, each O(1) → O(3) total
- Level 2: 9 calls, each O(1) → O(9) total
- Level k: 3^k calls
- There are $\log_3 n$ levels

Total calls = $1 + 3 + 9 + \ldots + 3^{\log_3 n} = (3^{\log_3 n + 1} - 1) / 2 = (3n - 1) / 2$.

Since $3^{\log_3 n} = n$, the total work is $\Theta(n)$.

Running time: $\Theta(n)$. (The work is dominated by the leaves of the recursion tree.)

Problem 6: Doubling Experiment

You run a doubling experiment and observe the following data:

N	Time (sec)	Ratio
1,000	0.05	—
2,000	0.21	4.2
4,000	0.82	3.9
8,000	3.30	4.0
16,000	13.1	4.0

a. What is the likely order of growth?

b. Estimate the running time for $N = 32{,}000$.

c. Write the tilde approximation.

Solution

a. The ratio stabilizes at approximately $4 = 2^2$, so $b \approx 2$. The order of growth is $\Theta(n^2)$.

b. $T(32{,}000) \approx 4 \times T(16{,}000) = 4 \times 13.1 = \mathbf{52.4\ seconds}$.

c. Using $T(N) = a\,N^2$ and the data point $T(16{,}000) = 13.1$:
$a = 13.1 / (16{,}000)^2 \approx 5.12 \times 10^{-8}$.
So $T(N) \sim 5.12 \times 10^{-8} \cdot N^2$.

Problem 7: True or False

State whether each claim is true or false, and briefly justify.

a. $n^2 = O(n^3)$

b. $n^3 = O(n^2)$

c. $2n + 5 = \Theta(n)$

d. $n \log n = O(n^2)$

e. If $f(n) = O(n^2)$, then $3f(n) = O(n^2)$

Solution

a. **True.** $n^2 \leq n^3$ for $n \geq 1$. Choose $c = 1$, $n_0 = 1$. (But this is a loose upper bound.)

b. **False.** n^3 grows faster than n^2. No constant c can make $n^3 \leq c \cdot n^2$ for all large n, since $n^3/n^2 = n \to \infty$.

c. **True.** For the O direction: $2n + 5 \leq 7n$ for $n \geq 1$. For the Ω direction: $2n + 5 \geq 2n$ for all n. So $2n + 5 = \Theta(n)$.

d. **True.** $n \log n \leq n \cdot n = n^2$ for $n \geq 1$ (since $\log n \leq n$). Choose $c = 1$, $n_0 = 1$.

e. **True.** If $f(n) \leq c \cdot n^2$ for $n \geq n_0$, then $3f(n) \leq 3c \cdot n^2$ for $n \geq n_0$. Choose $c' = 3c$. Multiplying by a constant does not change the asymptotic class.

Problem 8: Memory Analysis

Consider an algorithm that creates an $n \times n$ matrix of integers, then creates a one-dimensional array of size n to store row sums. What is the space complexity?

Solution

The $n \times n$ matrix uses $\Theta(n^2)$ space (for n^2 integers). The auxiliary array uses $\Theta(n)$ space. By the Sum Rule:

$$\text{Total space} = \Theta(n^2) + \Theta(n) = \Theta(n^2)$$

The matrix dominates the space complexity.

References

1. Sedgewick, R. & Wayne, K. (2011). *Algorithms*, 4th Edition. Addison-Wesley. Chapters 1.1, 1.4.
2. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*, 3rd Edition. MIT Press. Chapter 3 (Growth of Functions).
3. Knuth, D. E. (1997). *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, 3rd Edition. Addison-Wesley.
4. Sedgewick, R. & Wayne, K. Companion website: https://algs4.cs.princeton.edu