Algorithmic Thinking

Notes Complementarias

Autor: Andrés Santiago Ducuara Velásquez

2025

June 13, 2025

Algorithmic Thinking

A Practical Introduction to Programming

First Edition

Contents

Abstract

Algorithmic thinking is a core competency in the education of professionals in Computer Science and Artificial Intelligence. This text, designed as a complementary resource for the course *Algorithmic Thinking*, aims to provide a structured, rigorous, and in-depth guide to the principles and techniques underlying the formulation, analysis, and validation of computational algorithms.

The document adopts a theoretical and practical approach to topics ranging from the fundamental concept of algorithms and mathematical logic to more advanced subjects such as control structures, problem specification, data types, modularity, pointers, and the foundations of object-oriented programming. Additionally, it includes discussions on the boundaries of computation and promotes formal reasoning through exercises, explanatory diagrams using TikZ, and illustrative code examples across various programming paradigms: imperative (C), object-oriented (C++), functional (Haskell), logic-based (Prolog), multiparadigm (Python), concurrent (Erlang), and low-level (Assembly).

Each topic is treated with academic rigor and is supported by both classical and contemporary references, including foundational works such as **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie [**ritchieC**], and **The C++ Programming Language** by Bjarne Stroustrup [**stroustrupC++**], among others. This work aspires not only to serve as a didactic tool but also as a foundation for deeper reflections on the nature of computation, algorithmic efficiency, and the design of robust and elegant solutions.

Although the project was initially conceived and drafted in Spanish, it was later decided to continue its development in English in order to reach a broader audience. As a result, some figures or illustrative content may still appear in Spanish.

This material has been developed to accompany the course throughout the academic year 2025 and was authored by Professor Andrés Santiago Ducuara Velásquez, with the goal of fostering in students a profound, critical, and applied understanding of algorithmic thinking.

Part I

Fundamentals of Algorithmic Thinking

Chapter 1

A History of Logic: From Aristotle to Machine Learning

From the first Greek philosophers to the algorithms that power artificial intelligence, **logic** has been the hidden foundation of human thought and technology. More than a set of abstract rules, logic is the structure that allows us to distinguish between a valid argument and a fallacy, between a solid conclusion and speculation. Its evolution—from Aristotelian syllogisms to programming languages—has shaped not only philosophy and mathematics but also made the digital age and AI possible.

This historical journey is not merely a chronicle of ideas, but an exploration of how humans have sought to **systematize reason** in order to understand the world—and eventually replicate it in machines. How did we go from debates in the Athenian Agora to chatbots generating coherent text? What do Aristotle and Alan Turing have in common? The answer lies in logic: a universal language that transcends cultures and eras, and now underlies every circuit, every algorithm, and every "intelligent" system.

We will now unravel this journey, connecting the dots between philosophy, computing, and the ethical challenges of AI. Understanding logic is not just an academic exercise—it is key to decoding the present and anticipating the future of technology.

Historia de la Lógica

Antigüedad (siglos VI a.C. - V d.C.)

- Presocráticos (Parménides, Heráclito)
- Sócrates (mayéutica)
- Platón (dialéctica)
- Aristóteles (silogismo, Organon)
- Estoicos (lógica proposicional)

Edad Media (siglos V-XV)

- Filósofos árabes (Avicena, Averroes)
- Escolástica (Ockham)

Revolución Científica (siglos XVII-XIX)

- Descartes (método deductivo)
- Leibniz (lógica simbólica, binario)
- Boole (álgebra booleana)
- Frege (lógica de primer orden)

Siglo XX: Lógica y Computación

- Russell/Whitehead (Principia Mathematica)
- Gödel (teoremas incompletitud)
- Turing (máquina de Turing)
- Shannon (lógica + circuitos)

IA (siglo XX-XXI)

- Conferencia Dartmouth
- Prolog (lógica programable)
- IA moderna (LLMs, redes neuronales)

1.1 The Origins of Logic in Antiquity

The study of logic, understood as the discipline that examines the principles of valid reasoning, found its first major manifestation in Ancient Greece. While logical thinking existed in various civilizations in the form of pragmatic inferences or empirical rules, it was in the Greek world where the first systematic attempts to analyze the structure of rational thought were established.

1.1.1 The Presocratics: Early Explorations of Reasoning

During the 6th century BCE, the Presocratic philosophers began to question the nature of reality and knowledge. Although they did not develop formal logic, their reflections laid the groundwork for the analysis of reasoning.

One of the first to address the issue of truth and coherence was **Parmenides of Elea**, who argued that being is unique, immutable, and eternal. His philosophy relied heavily on the idea that contradiction is impossible, anticipating what Aristotle would later formalize as the *principle of non-contradiction*: a proposition cannot be both true and false at the same time.

On the other hand, **Heraclitus of Ephesus** adopted a diametrically opposite stance, emphasizing constant change as the essence of reality. His famous statement that "no one ever steps in the same river twice" illustrates his view of a world in continuous transformation, which led him to reflect on the tensions and contradictions inherent in thought and nature.

Despite their differences, both philosophers contributed to the discussion on the validity of reasoning, paving the way for more rigorous analysis in the following centuries.

1.1.2 Socrates and Dialectical Logic

In the 5th century BCE, **Socrates** introduced a revolutionary approach to reasoning: dialectic as a method of seeking truth. His technique, known as *maieutics*, consisted of a series of questions and answers intended to dismantle erroneous beliefs and reach more precise definitions.

Unlike the sophists, who used language to persuade without necessarily seeking truth, Socrates insisted that knowledge should be obtained through critical analysis. His method emphasized the importance of definitions and universal concepts, laying the foundations for the deductive thinking later developed by his student Plato.

1.1.3 Plato: The Distinction Between Opinion and Knowledge

Plato, influenced by his teacher Socrates, brought logical analysis to a new level by distinguishing between *doxa* (opinion) and *episteme* (true knowledge). According to his Theory of Ideas or Forms, the perceptible world is merely a shadow of an ideal realm where absolute truths reside.

For Plato, logical thought must aim at contemplating these truths, and in his dialogue *The Republic*, he introduces the *Platonic dialectic* as the supreme method of knowledge.

This process, based on pure reason, sought to ascend progressively from the sensible to the intelligible, using principles of logical coherence.
Although Plato did not develop a formal system of logic, his emphasis on the structure of reasoning had a significant influence on his most famous disciple: Aristotle.

1.1.4 Aristotle and the Systematization of Logic

It was **Aristotle**, in the 4th century BCE, who transformed logic into a structured discipline. In his work *Organon*, he compiled and formalized principles that remain fundamental to this day.
One of his greatest contributions was the theory of the *syllogism*, a form of deductive reasoning in which a conclusion is derived from two premises. A classic example is:

> All men are mortal.
>
> Socrates is a man.
>
> Therefore, Socrates is mortal.

Furthermore, Aristotle formulated fundamental principles of logic, such as:

- **The Principle of Identity**: a proposition is identical to itself ($A \equiv A$).
- **The Principle of Non-Contradiction**: a proposition cannot be both true and false at the same time ($\neg(A \wedge \neg A)$).
- **The Law of the Excluded Middle**: a proposition is either true or false, with no intermediate state ($A \vee \neg A$).

These concepts formed the core of Western logical thought for centuries and served as the foundation of Aristotelian logic, which prevailed until the Middle Ages.

1.1.5 The Stoics and Propositional Logic

While Aristotle had developed logic based on terms and categories, it was the **Stoics**, along with the Megarian school, who introduced a new form of logical analysis: *propositional logic*. Whereas Aristotelian logic focused on terms like "man" or "mortal," propositional logic analyzed the relationships between complete statements, using structures such as:

- "If A, then B" (*modus ponens*)
- "If not B, then not A" (*modus tollens*)

For example:

> If it rains, the street gets wet.
>
> It is raining.
>
> Therefore, the street is wet.

This approach anticipated fundamental developments in modern logic, such as the work of Gottlob Frege in the 19th century and contemporary mathematical logic.

1.2 The Aristotelian Legacy and the Evolution of Logic in the Middle Ages

After the fall of the Roman Empire, much of the philosophical and scientific knowledge of Antiquity was relegated in Western Europe. However, in the Islamic and Byzantine worlds, the texts of Aristotle and other Greek philosophers were preserved, commented upon, and expanded. During the **Middle Ages**, logic underwent a significant transformation thanks to the work of Arab philosophers and Christian scholastics, who not only reinterpreted Aristotelian teachings but also integrated them with new perspectives on reasoning and language.

1.2.1 Islamic Philosophy: Avicenna and Averroes

In the Islamic world, Arab philosophers played a crucial role in the transmission and development of Aristotelian logic. **Avicenna (Ibn Sina, 980–1037)**, one of the most influential, introduced innovations in *modal logic*, exploring the concepts of *necessity, possibility, and impossibility* in reasoning. His approach expanded logic beyond simple syllogistic deduction, incorporating notions of contingency and certainty that would influence Christian scholasticism and later logical developments.

On the other hand, **Averroes (Ibn Rushd, 1126–1198)**, known as *The Commentator* due to his exhaustive analyses of Aristotle, defended the autonomy of reason in relation to faith and promoted a strictly logical view of knowledge. His commentaries on Aristotle's *Organon* were instrumental in the reintroduction of logical thought in Europe, where his work would be intensely debated by scholastic philosophers.

1.2.2 Scholasticism and the Refinement of Logic in Europe

With the expansion of Christianity and the consolidation of universities in Europe during the 12th and 13th centuries, Aristotelian logic experienced a resurgence, thanks to the translation of Arabic and Greek philosophical texts. This period, known as **Scholasticism**, was characterized by the application of logical reasoning to theological and philosophical analysis.

One of the great exponents of this tradition was **Thomas Aquinas (1225–1274)**, who employed Aristotelian logic to argue for the existence of God and to establish a synthesis between faith and reason. His work, *Summa Theologica*, is a paradigmatic example of the scholastic method, based on the formulation of questions, objections, and structured responses with logical rigor.

1.2.3 William of Ockham and the Economy of Thought

In late scholasticism, **William of Ockham (1287–1347)** stood out for his critical approach to the overuse of abstractions in medieval philosophy. He is best known for the principle of

logical economy, popularly known as *Ockham's Razor*, which states that *"all other things being equal, the simplest explanation is usually the correct one."*
This principle had implications not only for logic and philosophy but also for the development of modern scientific method, as it encouraged the elimination of unnecessary assumptions in the formulation of theories.

1.3 The Scientific Revolution and Modern Logic (17th–19th Centuries)

With the arrival of the Scientific Revolution, logical thought underwent a radical transformation. Aristotelian logic, which had dominated Western thinking for centuries, began to be questioned and reformulated. New philosophical and mathematical movements led to the development of more abstract and formal systems, laying the foundations of modern logic and its connection to computation.

1.3.1 René Descartes (1596–1650): The Rationalist Method

René Descartes inaugurated the modern era of logical thinking by proposing a method based on systematic doubt and deductive reasoning. In his work *Discourse on the Method* (1637), he established four fundamental rules for rational thought:

1. **Evidence**: Accept only ideas that are clear and distinct.
2. **Analysis**: Divide problems into simpler parts.
3. **Synthesis**: Reconstruct knowledge from the simplest to the most complex.
4. **Enumeration**: Review each step to avoid omissions.

His famous principle *Cogito, ergo sum* ("I think, therefore I am") represents the starting point of modern rationalism. Although he did not develop a formal logical system, his emphasis on clarity and deduction influenced later thought.

1.3.2 Gottfried Wilhelm Leibniz (1646–1716): Forerunner of Symbolic Logic and Computation

Leibniz sought a more formal and abstract logic that would allow thought to be represented and operated upon as if it were a mathematical calculation. His main contributions include:

- **Characteristica Universalis**: An "alphabet of thought" to express all ideas through precise symbols and rules, anticipating symbolic logic.
- **Calculus Ratiocinator**: An idea of computational logic where reasoning could be carried out mechanically.
- **Binary System**: Introduced binary numeration (0 and 1), the foundation of modern computing.

Although his work was not fully realized in his time, his ideas greatly influenced mathematical logic and the development of computer science.

1.3.3 Immanuel Kant (1724–1804): Transcendental Logic

For **Kant**, Aristotelian logic was a "complete" system needing no expansion. However, in his *Critique of Pure Reason*, he introduced **transcendental logic**, distinguishing between:

- **General logic**: Rules of thought independent of experience (based on Aristotle).
- **Transcendental logic**: Conditions under which knowledge is possible, linking logic with the structure of the human mind.

Although Kant did not innovate in formal logic, his distinction between pure and applied logic influenced later developments in the philosophy of language and epistemology.

1.3.4 George Boole (1815–1864): The Algebraic Revolution in Logic

George Boole was the first mathematician to formalize logic using algebraic structures. In his work *The Laws of Thought* (1854), he developed **Boolean algebra**, a system based on fundamental logical operators:

- **AND** ($\wedge$): Conjunction ($A \wedge B$ is true only if both A and B are true).
- **OR** ($\vee$): Disjunction ($A \vee B$ is true if at least one is true).
- **NOT** ($\neg$): Negation ($\neg A$ inverts the truth value of A).

This system became essential for the development of digital circuits and computing, as it enables the representation of logical operations through electrical switches.

1.3.5 Gottlob Frege (1848–1925): First-Order Logic and Quantifiers

Frege revolutionized logic by introducing a new system of notation and structure for mathematical reasoning. His main contributions include:

- **First-order logic**: Introduced quantifiers such as $\forall$ (for all) and $\exists$ (there exists), allowing precise expression of mathematical relationships.
- **Sense and reference distinction**: Laid the groundwork for the philosophy of language by distinguishing the meaning of an expression from its referent.
- **Formalization of arithmetic**: Attempted to show that mathematics could be reduced to logic, anticipating modern mathematical logic.

His work laid the foundation for formal logic and set theory, influencing Bertrand Russell and computational logic.

1.4 Mathematical Logic and Computation (20th Century)

The 20th century marked a turning point in the history of logic, as it evolved from a philosophical discipline into the fundamental basis of computing and artificial intelligence. Efforts to formalize mathematics led to the discovery of the limits of logical systems and the development of computational models that are essential in modern computer science.

1.4.1 Bertrand Russell and Alfred N. Whitehead (1910–1913): The Logicist Program

At the beginning of the 20th century, **Bertrand Russell** and **Alfred North Whitehead** undertook the ambitious task of proving that all mathematics could be reduced to logic. Their work *Principia Mathematica* (published in three volumes between 1910 and 1913) aimed to build a formal system in which all mathematical truths could be derived from a set of logical axioms using precise rules.
Among their main achievements was the introduction of **type theory**, a system designed to avoid paradoxes in set theory, such as the famous **Russell's Paradox**:

> "If we consider the set of all sets that do not contain themselves, does it contain itself?"

Although *Principia Mathematica* represented a monumental advance in formal logic, the logicist program faced insurmountable difficulties after the later discoveries of Kurt Gödel.

1.4.2 Kurt Gödel (1931): Incompleteness Theorems

In 1931, **Kurt Gödel** revolutionized mathematical logic by proving his **incompleteness theorems**, which established fundamental limits for formal systems:

1. **First Incompleteness Theorem**: In any sufficiently expressive formal system (such as that of *Principia Mathematica*), there exist true propositions that cannot be proven within the system.
2. **Second Incompleteness Theorem**: No consistent system can prove its own consistency from within its own axiomatic framework.

These results showed that the dream of reducing all mathematics to logic was impossible and placed the logicist program of Russell and Whitehead into crisis. Gödel's discoveries had a profound impact on computation and artificial intelligence, demonstrating that there are inherent limits to what machines can compute.

1.4.3 Alan Turing (1912–1954): The Turing Machine and Computation

Alan Turing, considered one of the fathers of computer science, addressed the problem of formalizing the concept of algorithm. In his influential 1936 paper, *On Computable Numbers,*

with an Application to the Entscheidungsproblem, he introduced the **Turing Machine**, a theoretical model of computation that simulates the functioning of any algorithmic process. His key contributions include:

- **Formal definition of algorithm**: The Turing Machine enabled a precise formulation of the concept of computability.
- **Halting problem**: Turing proved that there is no universal algorithm capable of determining whether any given algorithm will halt or run indefinitely (the problem of undecidability).
- **Cryptography and practical computation**: During World War II, Turing led the Bletchley Park team in decoding the Enigma machine, contributing to the development of the first electronic computers.

His work laid the foundations of modern computation and artificial intelligence, establishing a direct link between mathematical logic and computer science.

1.4.4 Claude Shannon (1937): Boolean Logic and Electronic Circuits

While Turing developed a theoretical model of computation, **Claude Shannon** applied mathematical logic to physical systems. In his 1937 master's thesis, *A Symbolic Analysis of Relay and Switching Circuits*, he demonstrated that **Boolean algebra** could be used to design electrical circuits.
His discoveries were fundamental to digital electronics:

- Introduced the concept of **logic gates**, enabling logical operations through electrical circuits.
- Established the connection between mathematical logic and computer design.
- Pioneered **information theory**, defining how data can be efficiently represented and transmitted.

Shannon's work enabled the creation of the first electronic computers, cementing the union of logic, mathematics, and technology.

1.5 Artificial Intelligence (20th–21st Centuries)

The development of Artificial Intelligence (AI) has been closely linked to mathematical logic and computation. From its origins in the 1950s to the modern era of neural networks and generative models, AI has evolved through multiple approaches—some based on symbolic logic, others on statistical and connectionist models.

1.5.1 1950s–60s: The Birth of Artificial Intelligence

The term **Artificial Intelligence (AI)** was coined by **John McCarthy** at the historic *Dartmouth Conference* in 1956, where the idea was proposed that machines could simulate

human thinking through rules and logical reasoning. Participants such as **Marvin Minsky**, **Herbert Simon**, and **Allen Newell** laid the foundation for symbolic AI.
This early phase was dominated by logic-based symbolic approaches:

- **Rule-based systems**: Models that represented knowledge using logical facts and inference rules.
- **Early expert systems**: Programs designed to emulate human expert reasoning in specific domains. Examples:
  - *ELIZA* (1966): A rule-based chatbot developed by **Joseph Weizenbaum**, simulating conversations with a psychotherapist through pattern matching.
  - *DENDRAL* (1965): An expert system for identifying chemical structures, developed by **Edward Feigenbaum** and **Joshua Lederberg**.

These initial efforts demonstrated that AI could solve domain-specific problems using logical reasoning, but also revealed limitations in handling complex, real-world environments.

1.5.2 Logic in AI: Formal Languages and Programming

To endow machines with formal reasoning capabilities, logic-based programming languages were developed:

- **John McCarthy and LISP (1958)**:
  - McCarthy created *LISP*, a functional programming language that became the standard for AI research.
  - He also explored *modal logic*, enabling reasoning about uncertainty and future possibilities.
- **Prolog (1972): Logic Programming for AI**
  - Developed by **Alain Colmerauer** and **Robert Kowalski**, Prolog was one of the first declarative programming languages.
  - Based on *first-order logic*, it allowed the definition of facts and rules for automatic inference.
  - Widely used in AI applications such as natural language processing and expert systems.

Despite its theoretical elegance, symbolic AI based on logic struggled in real-world environments requiring uncertain, noisy, or incomplete knowledge.

1.5.3 Limits of Symbolic AI and New Directions (1980–2000)

During the 1980s, it became evident that rule-based systems had serious limitations in dynamic environments and with large data volumes. This led to the emergence of alternative approaches, such as **Machine Learning (ML)**:

- **Artificial Neural Networks (ANNs)**: Inspired by the human brain, neural networks learned patterns from data without relying on explicit rules.
- **Statistical learning**: Models like decision trees and support vector machines (SVMs) replaced symbolic logic in many tasks.
- **Connectionist vs. Symbolic Paradigm**:
  - *Symbolic AI*: Rule-based reasoning.
  - *Connectionist AI*: Data-driven learning using neural networks.

The connectionist approach gained momentum with the explosion of data and computing power, shifting AI away from traditional logic toward probabilistic and learning-based methods.

1.5.4 Contemporary AI: Advanced Models and Logic in the Deep Learning Era

In the 21st century, AI has evolved through advanced models combining logic with statistical and probabilistic methods:

- **Deep Learning Models**:
  - Deep neural networks have proven highly effective in computer vision, natural language processing, and robotics.
  - Examples include *GPT-4*, *DALL · E*, and *AlphaGo*.
- **Fuzzy Logic**:
  - Introduced by **Lotfi Zadeh**, fuzzy logic enables reasoning with imprecision and uncertainty.
  - Applied in automatic control, decision-making, and embedded systems.
- **Logic and Explainability in AI**:
  - A current challenge is the *interpretability* of complex models such as deep neural networks.
  - *Explainable AI (XAI)* techniques aim to make AI decisions understandable.
  - Tools like *formal logic* and *causal models* help explain why AI systems make certain choices.
- **Hybrid Systems: Combining Logic and Machine Learning**:
  - Models like *Explainable Boosting Machines (EBMs)* integrate logic and ML to enhance interpretability.
  - Researchers are exploring methods to combine logical rules with neural networks to leverage the strengths of both paradigms.

Chapter 2

Introduction to Computer Science and Algorithmic Thinking

2.1 Algorithms

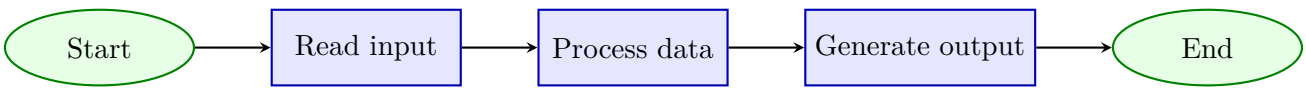
The study of algorithms is fundamental for any student of Computer Science, as they represent the essence of computational thinking and efficient problem-solving. Understanding algorithms not only enables programming but also helps grasp the fundamental principles behind optimization, efficiency, and computational logic.

2.1.1 What is an Algorithm?

According to *Introduction to the Design and Analysis of Algorithms* by Anany Levitin, an algorithm is a finite set of well-defined instructions that, when executed, solve a specific problem. Formally, an algorithm must satisfy the following properties:

1. **Correctness**: It must produce the expected output for all valid inputs.
2. **Efficiency**: It must run within a reasonable time and consume optimal resources.
3. **Finiteness**: It must complete in a finite number of steps.
4. **Clarity**: It must be understandable and easy to implement.

From a more practical perspective, as described in *Algorithm Design* by Jon Kleinberg and Éva Tardos, an algorithm is the strategy that transforms an input into a desired output, with an emphasis on the structure of the problem and techniques to approach it.



2.1.2 Why Study Algorithms?

In *How to Solve It* by George Pólya, the importance of developing problem-solving skills through strategies such as decomposition, induction, and generalization is emphasized. Learning

algorithms involves thinking in structured and efficient ways, which enables tackling complex problems methodically.

2.1.3 What is Algorithmic Problem Solving?

Solving an algorithmic problem involves finding a precise sequence of steps that transforms an input into a desired output. According to Levitin, the problem-solving process follows several stages:

1. **Understanding the problem** – Clearly define what needs to be solved.
2. **Mathematical formulation** – Express the problem in an abstract way.
3. **Algorithm design** – Develop a structured set of steps.
4. **Efficiency analysis** – Evaluate the algorithm's performance.
5. **Implementation and testing** – Convert the algorithm into code and verify its correctness.

2.1.4 Problem Solving

When we talk about solving problems, there are initially two main approaches. One is the traditional manual problem solving, which relies on human intuition and experience. It often involves trial and error to reach a solution. A classic example of this method is:

> Imagine you have an unordered list of numbers and want to find the largest one. If done manually, you simply check each number and remember the largest encountered so far.

On the other hand, there is algorithmic problem solving, which uses the structure of algorithms to define a well-ordered, repeatable sequence to reach a solution. This can be applied to multiple cases without changing the underlying logic. Applying the previous example in an algorithmic form:

1. Define a variable `max` with the first number in the list.
2. Traverse each number in the list:
    - If the current number is greater than `max`, update `max`.
3. At the end, `max` holds the largest number.

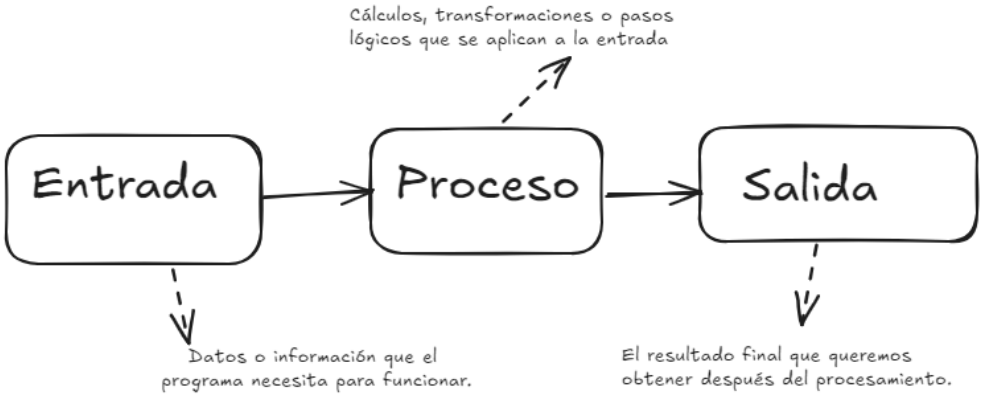
One of the keys of algorithmic thinking is **abstraction**, the process of simplifying a problem by focusing on the essential aspects and ignoring unnecessary details. While abstraction is central to algorithms, other tools are also required, such as:

- **Decomposition** – Breaking down a problem into smaller, more manageable sub-problems.
- **Pattern recognition** – Identifying similar behaviors or structures in different problems.

These cognitive tools have been inherently present in humans since ancient times and have been crucial for the survival and evolution of our species. The combination of these three—abstraction, decomposition, and pattern recognition—with algorithms, is what we define as **algorithmic thinking**.

2.2 Input-Process-Output (IPO) Model

The Input-Process-Output (IPO) model is a structured way to analyze problems before solving them with an algorithm or program. It helps to understand what data is needed, what operations must be performed, and what the expected outcome will be.



2.3 Types of Problems

Before applying a solution strategy, it is important to identify the type of problem to be solved. There are different categories, each with specific approaches and techniques for solving them. The main types of algorithmic problems are:

Type	**Concept**
Search	Focuses on efficiently finding data.
Sorting	Focuses on reorganizing data for efficient access.
Optimization	Seeks the best possible solution.

2.4 Problem-Solving Strategies

Algorithmic strategies provide systematic approaches to solving problems efficiently. These strategies can be classified according to their complexity and applicability.

Name	**Goal**
Brute Force	Try all possible solutions until the correct one is found.
Iterative Improvement	Repeatedly refine a solution until the best option is reached.
Greedy Technique	Make optimal decisions at each step hoping to reach a global optimum.
Recursion	Solve a problem by dividing it into smaller instances of the same problem.
Divide and Conquer	Break the problem into smaller subproblems and solve them recursively.
Memoization	Store intermediate results to avoid unnecessary repetition.
Dynamic Programming	Solve problems by breaking them into overlapping subproblems and storing results.

Chapter 3

Flowcharts: Graphical Representation of Algorithms

A **flowchart** is a graphical representation of an algorithm or process using a set of standardized symbols that illustrate the sequence of steps required to solve a problem. They are used to describe, analyze, and improve processes, making them easier to understand and communicate.

3.0.1 Uses of Flowcharts

Flowcharts are used across various fields, especially in programming and process analysis. Their main uses include:

- **Algorithm design:** Before writing code in a programming language, a flowchart is used to plan the program's logic.
- **Process documentation:** They document business, administrative, or industrial processes.
- **Error detection:** Flowcharts help identify logical errors in an algorithm before implementation.
- **Process improvement:** They assist in optimizing procedures by removing unnecessary steps.

3.0.2 Why Use Flowcharts?

Using flowcharts is essential for:

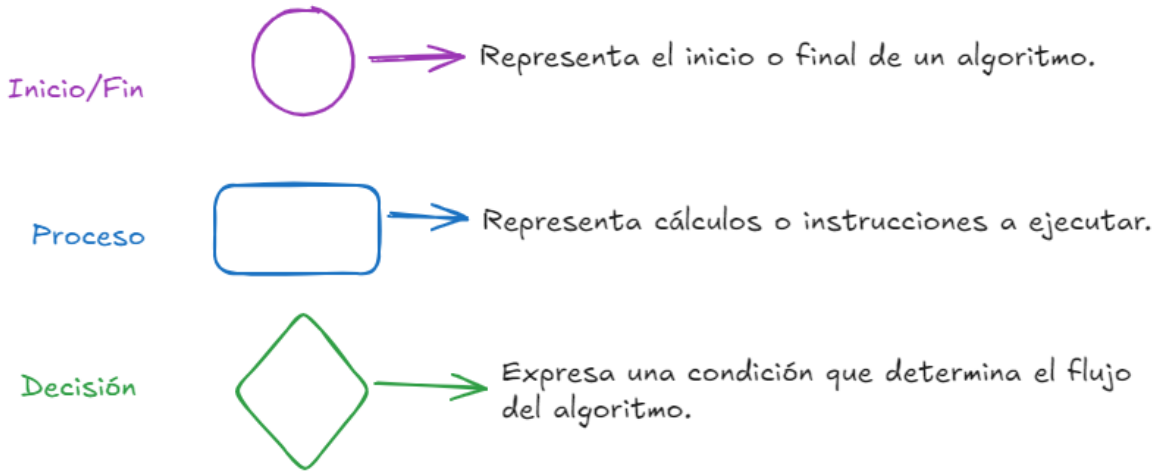
- **Visualizing the structure of an algorithm** in a clear and organized way.
- **Improving communication** between programmers, designers, and stakeholders.
- **Reducing errors** before implementing a program.
- **Facilitating software maintenance**, allowing for more understandable modifications.

3.0.3 Benefits of Flowcharts

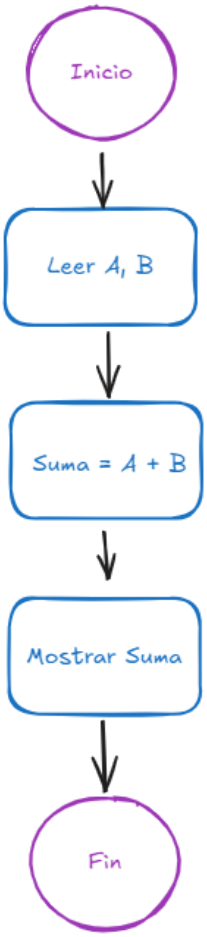
Flowcharts offer many advantages, including:

- **Clarity and simplicity:** They represent processes visually, making them easier to interpret.
- **Standardization:** They use universal symbols that anyone familiar with flowcharts can understand.
- **Debugging support:** They help identify logical errors before code is written.
- **Process optimization:** They help detect inefficient steps in a procedure.
- **Improved documentation:** They are valuable tools for documenting systems and processes in companies or software projects.

3.0.4 Basic Flowchart Symbols



Ejemplo:



3.1 Example: Compute the sum of two numbers

Figure 3.1: flowchart

```
section .text
    global _start
_start:
    mov eax, 5
    add eax, 3
    ; eax now contains 8
```

Listing 3.1: Assembly

```
#include <stdio.h>
int main() {
    int a = 5, b = 3;
    printf("Sum: %d\n", a + b);
    return 0;
}
```

Listing 3.2: C

```
#include <iostream>
int main() {
    int a = 5, b = 3;
    std::cout << "Sum: " << a + b << std::endl;
}
```

Listing 3.3: C++

```
main = print (5 + 3)
```

Listing 3.4: Haskell

```
sum(A, B, R) :- R is A + B.
```

Listing 3.5: Prolog

```
a = 5
b = 3
print("Sum:", a + b)
```

Listing 3.6: Python

```
-module(sum).
-export([add/2]).
add(A, B) -> A + B.
```

Listing 3.7: Erlang

Chapter 4

Programming languages

Programming languages are fundamental tools in computer science, allowing developers to write instructions that computers can understand. Over time, these languages have evolved significantly, adopting various approaches and paradigms to suit different kinds of problems.

4.1 Classification of Programming Languages

Programming languages have evolved to meet different needs and development styles. Depending on their features and usage, they can be classified in several ways.
A common classification is based on the **level of abstraction**, which indicates how close the language is to hardware or how easy it is for humans to read and write. Another classification depends on the **execution model**—compiled, interpreted, or hybrid—which affects performance and portability.
Languages can also be grouped by **paradigm**, or the structural and logical approach they use: imperative, functional, object-oriented, or logical programming.

4.1.1 By Level of Abstraction

1. Low-Level Languages

Low-level languages are closely related to hardware, providing fine control over system resources such as memory and the CPU. While efficient, they are harder to learn and hardware-dependent.

- Machine Language (binary code)
- Assembly (x86, ARM, MIPS)
- C/C++
- Rust
- Pascal (low-level variants)
- Embedded C (for microcontrollers)

```
section .data
    msg db "Hello, world!", 0Ah

section .text
    global _start

_start:
    mov eax, 4
    mov ebx, 1
    mov ecx, msg
    mov edx, 13
    int 0x80

    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Listing 4.1: Assembly

2. High-Level Languages

High-level languages are designed to be human-readable and easier to use. They are generally portable across systems but less efficient due to abstraction overhead.

- Python
- Java
- JavaScript
- Go
- Ruby
- PHP
- Swift
- Kotlin
- R
- MATLAB
- SQL
- HTML/CSS

```
print("Hello, world!")
```

Listing 4.2: Python

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Listing 4.3: Java

4.1.2 By Execution Model

1. Compiled Languages

Compiled languages require translation before execution. A compiler converts source code into machine code, which leads to higher performance.

```c
#include <stdio.h>
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Listing 4.4: C

Compilation stages:

1. Preprocessing
2. Compilation to assembly
3. Assembly to object code
4. Linking to create the executable

Linux Compilation Example:

```
gcc hello.c -o hello
./hello
```

Generated Assembly Example:

```
.LC0:
    .string "Hello, world!"
main:
    push    rbp
    mov     edi, OFFSET FLAT:.LC0
    call    puts
    mov     eax, 0
    pop     rbp
    ret
```

Listing 4.5: Assembly

2. Interpreted Languages

Interpreted languages are executed line by line using an interpreter, providing flexibility and ease of debugging at the cost of performance.

```python
print("Hello, world!")
```

Listing 4.6: Python

Execution steps:

1. Python reads the code
2. Converts it to bytecode
3. Executes via the Python Virtual Machine (PVM)

Example Bytecode:

```
LOAD_CONST  'Hello, world!'
PRINT_ITEM
PRINT_NEWLINE
RETURN_VALUE
```

3. Hybrid Languages

Hybrid languages combine compilation and interpretation. For example, Java compiles to bytecode, which is then interpreted by the JVM.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

Listing 4.7: Java

Execution Process:

```
javac Main.java
java Main
```

Java Bytecode:

```
0: getstatic     #2
3: ldc           #3
5: invokevirtual #4
8: return
```

4.1.3 Execution Models Comparison

Type	Process	Example	Performance	Portability
Compiled	Source → Machine Code	C	High	Low
Interpreted	Source → Runtime Interpretation	Python	Low	High
Hybrid	Source → Bytecode → VM	Java	Medium	High

4.2 Programming Paradigms

Paradigms define the style and structure of programs. Each offers principles and strategies suited to different problems.

4.2.1 1. Imperative Paradigm

Based on step-by-step instructions and mutable state.
Languages: C, Pascal, Fortran, Assembly
Example in C:

```c
#include <stdio.h>

int main() {
    int x = 5;
    x = x + 3;
    printf("%d", x);
    return 0;
}
```

Listing 4.8: C

4.2.2 2. Functional Paradigm

Based on pure functions and immutability.
Languages: Haskell, Lisp, Clojure, F#
Example in Haskell:

```haskell
suma x y = x + y
main = print (suma 5 3)
```

Listing 4.9: Haskell

4.2.3 3. Object-Oriented Paradigm

Organizes code into objects with attributes and methods.
Languages: Java, Python, C++, C#
Example in Python:

```cpp
#include <iostream>
#include <string>

class Person {
private:
    std::string name;
public:
    Person(std::string name) {
        this->name = name;
    }
    void greet() {
        std::cout << "Hello, I'm " << name << std::endl;
        }
};

int main() {
    Person p("Ana");
    p.greet();
    return 0;
}
```

Listing 4.10: C++

4.2.4 4. Logical Paradigm

Uses rules and facts to deduce new information.

Languages: Prolog, Datalog

Example in Prolog:

```
father(juan, maria).
father(juan, jose).
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Listing 4.11: Prolog

4.3 A brief history of Programming Languages

4.3.1 1. First Generation (1940s–1950s): Machine Language

Binary instructions specific to hardware; hard to write, but fast.

Milestones:

- ENIAC (1945): First general-purpose electronic computer
- Von Neumann (1945): Stored-program architecture

4.3.2 2. Second Generation (1950s–1960s): Assembly Language

Mnemonics like `MOV`, `ADD`; easier to program than binary.
Milestones:

- IBM 701 (1952): First commercial IBM computer
- First assemblers (1955)

4.3.3 3. Third Generation (1960s–1980s): High-Level Languages

More readable and portable than previous generations.
Milestones:

- Fortran (1957): Scientific computing
- COBOL (1959): Business applications
- C (1972): UNIX systems
- Structured programming (1968): Dijkstra

4.3.4 4. Fourth Generation (1980s–2000s): OOP and Databases

Focused on code reuse and data management.
Milestones:

- Smalltalk (1980): First fully OOP language
- C++ (1983): OOP extension of C
- SQL (1974/1980s): Databases
- Java (1995): Cross-platform development

4.3.5 5. Fifth Generation (2000–Present): Declarative and Specialized

Languages for AI, big data, and concurrency.
Milestones:

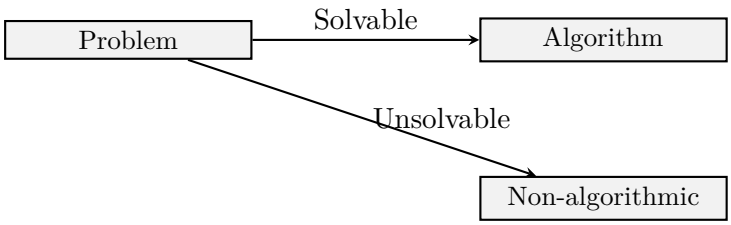
- Python (1991, popular in 2000s): AI and web
- R (1993): Statistics
- Julia (2012): High-performance science
- Go (2009): Concurrency and cloud systems

Chapter 5

Algorithmic vs. Non-Algorithmic Problems

5.1 Definition of a Problem

In computer science, a problem is a well-defined question with a set of possible inputs and expected outputs. A problem is said to be **algorithmic** if there exists a finite procedure (algorithm) that always provides a correct answer in a finite amount of time for every valid input.



5.2 Computable Algorithms

A computable algorithm is one that can be executed by a theoretical machine, such as a Turing machine, which halts after a finite number of steps.

- All arithmetic operations ($+$, $-$, $\times$, $\div$) are computable.
- Searching and sorting can be expressed algorithmically.
- Data transformation tasks are often computable.

5.3 Limits of Algorithmics

Not every problem can be solved by an algorithm. Examples include:

- **The Halting Problem:** No algorithm can decide for every possible program whether it halts.

- **Subjective Problems:** “Is this artwork beautiful?” cannot be formalized universally.
- **Randomness and Chaos:** Some systems are inherently unpredictable.

The Halting Problem

Alan Turing proved that there is no general algorithm that decides whether an arbitrary computer program will eventually halt or continue to run forever.

Chapter 6

Problem Specification

Understanding how to clearly and formally specify a problem is a crucial step in algorithmic thinking. A well-posed problem includes its input data, expected output, and constraints or conditions that must be satisfied.

6.1 Input

The input is the data provided to the algorithm. It must be defined in terms of its structure, format, and acceptable ranges or constraints.

6.2 Output

The output defines the result the algorithm must return. It should be specified in terms of type, structure, and constraints.

6.3 Conditions

Conditions may include mathematical constraints, boundaries on input size, or time/memory limits.

Chapter 7

Basic Concepts

7.1 Data Types and Variables

Data types are the fundamental building blocks of any programming language. They define the kind of data that variables can store, the operations that can be performed on them, and how memory is allocated. Understanding data types is essential for writing robust and efficient algorithms.
Variables act as named storage locations in a program's memory, and each variable must be associated with a specific data type. This association determines the kind of data the variable can hold, how much memory it consumes, and what operations can be applied to it. Declaring variables with the correct data type is crucial for both program correctness and performance.

7.1.1 Primitive Data Types

Primitive data types represent the most basic forms of data:

- **Integer**: Whole numbers, e.g., `int`
- **Floating-point**: Decimal numbers, e.g., `float`, `double`
- **Character**: Single symbols, e.g., `char`
- **Boolean**: Logical values, `true` or `false`

7.1.2 Composite Data Types

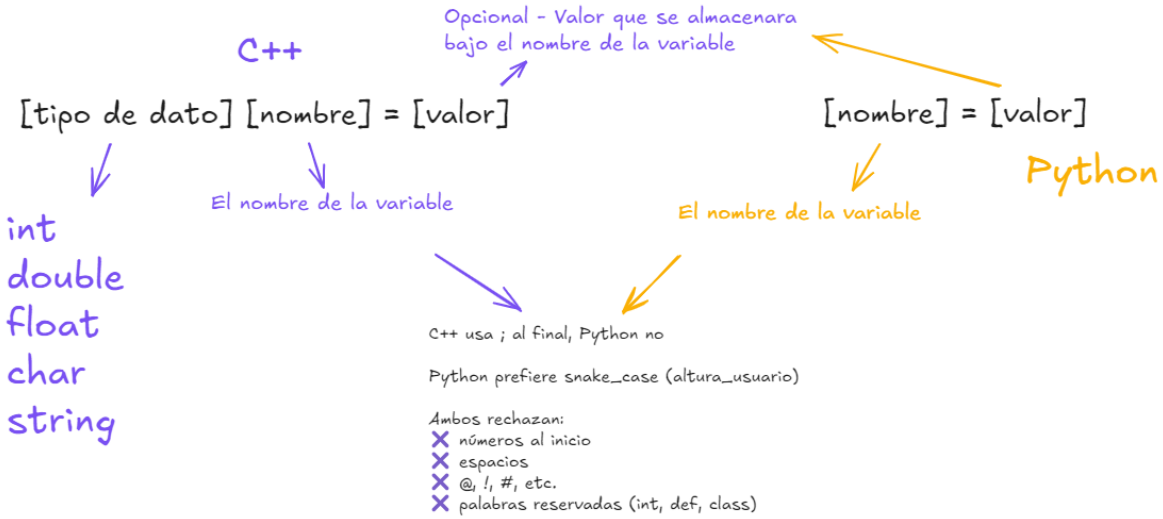
Composite data types are composed of other data types:

- **Arrays**: Ordered collections of elements of the same type
- **Structures**: Grouped fields of various types
- **Lists, Tuples, Maps**: High-level abstractions in functional or dynamic languages

Note

In some programming languages like **Python**, primitive data types such as integers, floats, and booleans are not implemented as raw low-level types but rather as full-fledged *objects* or *classes*. This allows for greater flexibility and abstraction. In contrast, languages like **C** use true primitive types that map closely to memory representations and are not objects.

7.1.3 Example - Declarate varable C++ and Python




```assembly
section .data
    ; Integer types
    int8_val     db 127              ; 8-bit signed integer
    uint8_val    db 255              ; 8-bit unsigned integer
    int16_val    dw 32767            ; 16-bit signed integer
    uint16_val   dw 65535            ; 16-bit unsigned integer
    int32_val    dd 2147483647       ; 32-bit signed integer
    uint32_val   dd 0xFFFFFFFF       ; 32-bit unsigned integer

    ; Floating-point types
    float_val    dd 3.14             ; 32-bit float
    double_val   dq 3.14159265358979 ; 64-bit double

    ; Character
    char_val     db 'A'              ; ASCII character

    ; Boolean (represented as byte: 0 = false, non-zero = true)
    bool_true    db 1
    bool_false   db 0
```

Listing 7.1: Assembly

```c
int main() {
    signed char int8_val = 127;          // 8-bit signed
    unsigned char uint8_val = 255;       // 8-bit unsigned

    short int16_val = 32767;             // 16-bit signed
    unsigned short uint16_val = 65535;   // 16-bit unsigned

    int int32_val = 2147483647;          // 32-bit signed
    unsigned int uint32_val = 4294967295U; // 32-bit unsigned

    float float_val = 3.14f;             // 32-bit float
    double double_val = 3.14159265358979; // 64-bit double

    char char_val = 'A';                 // ASCII character

    bool bool_true = true;
    bool bool_false = false;

    return 0;
}
```

Listing 7.2: C

```cpp
int main() {
    int8_t int8_val = 127;
    uint8_t uint8_val = 255;

    int16_t int16_val = 32767;
    uint16_t uint16_val = 65535;

    int32_t int32_val = 2147483647;
    uint32_t uint32_val = 4294967295U;

    float float_val = 3.14f;
    double double_val = 3.14159265358979;

    char char_val = 'A';

    bool bool_true = true;
    bool bool_false = false;

    std::string str_val = "Hello, world!";
    return 0;
}
```

Listing 7.3: C++

```haskell
entero :: Int
entero = 42

grande :: Integer
grande = 12345678901234567890

decimal :: Float
decimal = 3.14

doblePrecision :: Double
doblePrecision = 2.718281828459045

caracter :: Char
caracter = 'H'

cadena :: String
cadena = "Hola Haskell"

booleano :: Bool
booleano = True

-- Maybe (optional)
valorOpcional :: Maybe Int
valorOpcional = Just 10

valorNada :: Maybe Int
valorNada = Nothing

-- Either (left or right)
resultado :: Either String Int
resultado = Right 200

errorResultado :: Either String Int
errorResultado = Left "Error"
```

Listing 7.4: Haskell

```prolog
color(red).
status(ok).
greeting(hello).

age(25).
temperature(36.6).

welcome("Hello, Prolog!").

is_true(true).
is_false(false).
```

```
12
13 maybe_just(just(42)).
14 maybe_nothing(nothing).
15
16 either_right(right("Success")).
17 either_left(left("Error")).
```

Listing 7.5: Prolog

```
1
2 integer_value = 42
3
4 float_value = 3.1415
5
6 complex_value = 2 + 3j
7
8 string_value = "Hello, Python!"
9
10 bool_true = True
11 bool_false = False
12
13 none_value = None
```

Listing 7.6: Python

```
1 integer_value() -> 42.
2
3 float_value() -> 3.1415.
4
5 atom_value() -> hello.
6
7 bool_true() -> true.
8 bool_false() -> false.
9
10 string_value() -> "Hello, Erlang!".
11
12 char_value() -> $A.  % ASCII code of 'A'
13
14 maybe_value(true) -> {just, 42};
15 maybe_value(false) -> nothing.
16
17 either_value(ok) -> {right, "Success"};
18 either_value(error) -> {left, "Failure"}.
```

Listing 7.7: Erlang

7.2 Operators

Operators allow the manipulation of data stored in variables. They are fundamental components in any programming language, used to perform calculations, comparisons, logical decisions, and bit-level manipulations.
Operators are typically divided into the following main categories:

- Arithmetic operators
- Relational (comparison) operators
- Logical (boolean) operators
- Bitwise operators
- Assignment operators
- Increment/decrement operators (in some languages)
- Special operators (e.g., ternary, pointer, etc.)

7.2.1 Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations between numeric values (integers and floats). These are common across nearly all programming languages.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulo (remainder)	a % b

7.2.2 Bitwise Operators

Bitwise operators perform operations directly on the binary representations of integers. These are especially useful in low-level programming, cryptography, compression algorithms, and hardware control.

Operator	Description	Example
&	Bitwise AND	a & b
|	Bitwise OR	a | b
^	Bitwise XOR	a ^ b
~	Bitwise NOT (1's complement)	~a
«	Left shift	a « 2
»	Right shift	a » 2

Note

Bitwise operations are only valid for integer data types. They are extremely efficient and commonly used for performance-critical code, setting or clearing specific bits, and implementing flags or permissions using binary masks.

```
_start:

    ; result = a + b
    mov eax, [a]
    add eax, [b]
    mov [result], eax

    ; result = a - b
    mov eax, [a]
    sub eax, [b]
    mov [result], eax

    ; result = a * b
    mov eax, [a]
    mov ebx, [b]
    imul ebx
    mov [result], eax

    ; result = a / b
    mov eax, [a]
    cdq                 ; Sign-extend EAX into EDX:EAX
    mov ebx, [b]
    idiv ebx
    mov [result], eax   ; Quotient
    ; Remainder is in EDX if needed

    ; result = a & b (AND)
    mov eax, [a]
    and eax, [b]
    mov [result], eax

    ; result = a | b (OR)
    mov eax, [a]
    or eax, [b]
    mov [result], eax

    ; result = a ^ b (XOR)
    mov eax, [a]
    xor eax, [b]
    mov [result], eax

```

```
    ; result = ~a (NOT)
    mov eax, [a]
    not eax
    mov [result], eax

    ; result = a << 1 (Left Shift)
    mov eax, [a]
    shl eax, 1
    mov [result], eax

    ; result = a >> 1 (Right Shift)
    mov eax, [a]
    shr eax, 1
    mov [result], eax
```

Listing 7.8: Assembly

```
int main() {

    result = a + b;
    result = a - b;
    result = a * b;
    result = a / b;
    result = a % b;

    result = a & b;
    result = a | b;
    result = a ^ b;
    result = ~a;
    result = a << 1;
    result = a >> 1;

    return 0;
}
```

Listing 7.9: C

```
module Main where

import Data.Bits -- Required for bitwise operations

main :: IO ()
main = do
    print $ "a + b = " ++ show (a + b)
    print $ "a - b = " ++ show (a - b)
    print $ "a * b = " ++ show (a * b)
    print $ "a `div` b = " ++ show (a `div` b) -- Integer division
    print $ "a `mod` b = " ++ show (a `mod` b)
```

```haskell

    print $ "a .&. b = " ++ show (a .&. b)  -- Bitwise AND
    print $ "a .|. b = " ++ show (a .|. b)  -- Bitwise OR
    print $ "a `xor` b = " ++ show (a `xor` b)  -- Bitwise XOR
    print $ "complement a = " ++ show (complement a)  -- Bitwise NOT
    print $ "a `shiftL` 1 = " ++ show (a `shiftL` 1)  -- Left shift
    print $ "a `shiftR` 1 = " ++ show (a `shiftR` 1)  -- Right shift
```

Listing 7.10: Haskell

```prolog
arithmetic :-
    Sum is A + B,
    Diff is A - B,
    Prod is A * B,
    Div is A / B,
    IntDiv is A // B,
    Mod is A mod B,

bitwise :-
    A is 6,  % 110
    B is 3,  % 011
    And is A /\ B,
    Or is A \/ B,
    Xor is A xor B,
    NotA is \(A),
    Shl is A << 1,
    Shr is A >> 1,
```

Listing 7.11: Prolog

```python
# operators_demo.py
print(f"a + b = {a + b}")
print(f"a - b = {a - b}")
print(f"a * b = {a * b}")
print(f"a / b = {a / b}")      # Float division
print(f"a // b = {a // b}")    # Integer (floor) division
print(f"a % b = {a % b}")
print(f"a ** b = {a ** b}")    # Exponentiation

print(f"a & b = {a & b}")      # Bitwise AND
print(f"a | b = {a | b}")      # Bitwise OR
print(f"a ^ b = {a ^ b}")      # Bitwise XOR
print(f"~a = {~a}")            # Bitwise NOT (2's complement)
print(f"a << 1 = {a << 1}")    # Left shift
print(f"a >> 1 = {a >> 1}")    # Right shift

c = a
c += b
```

```
c -= b
c *= b
c //= b
c %= b
```

Listing 7.12: Python

```
main() ->

    io:format("A + B = ~p~n", [A + B]),
    io:format("A - B = ~p~n", [A - B]),
    io:format("A * B = ~p~n", [A * B]),
    io:format("A div B = ~p~n", [A div B]),  % Integer division
    io:format("A rem B = ~p~n", [A rem B]),  % Remainder

    io:format("A band B = ~p~n", [A band B]),
    io:format("A bor B = ~p~n", [A bor B]),
    io:format("A bxor B = ~p~n", [A bxor B]),
    io:format("bnot A = ~p~n", [bnot A]),
    io:format("A bsl 1 = ~p~n", [A bsl 1]),  % Bit shift left
    io:format("A bsr 1 = ~p~n", [A bsr 1]).  % Bit shift right
```

Listing 7.13: Erlang

7.2.3 Relational Operators

Relational operators are used to compare values and return boolean results. These results are used to control program flow in conditional statements and loops.

Op	Description	Assembly	C	C++	Prolog	Python	Haskell	Erlang
==	Equality	`CMP AX, BXJE label`	`a == b`	`a == b`	`A =:= B`	`a == b`	`a == b`	`A == B`
!=	Not equal	`CMP AX, BXJNE label`	`a != b`	`a != b`	`A =B̄`	`a != b`	`a /= b`	`A =/= B`
>	Greater than	`CMP AX, BXJG label`	`a > b`	`a > b`	`A > B`	`a > b`	`a > b`	`A > B`
<	Less than	`CMP AX, BXJL label`	`a < b`	`a < b`	`A < B`	`a < b`	`a < b`	`A < B`
>=	Greater or equal	`CMP AX, BXJGE label`	`a >= b`	`a >= b`	`A >= B`	`a >= b`	`a >= b`	`A >= B`
<=	Less or equal	`CMP AX, BXJLE label`	`a <= b`	`a <= b`	`A =< B`	`a <= b`	`a <= b`	`A =< B`

Chapter 8

Control Structures: Conditional Statements

Conditional statements are essential in algorithm design because they allow programs to take different paths depending on specific conditions. This decision-making ability enables algorithms to respond dynamically to varying inputs and states.

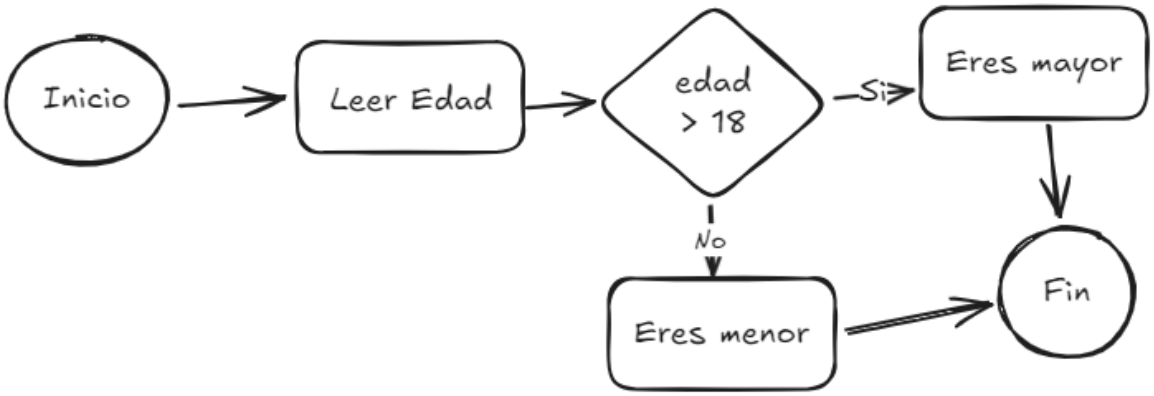
8.1 Basic Conditional Logic

Conditional structures evaluate a Boolean expression and execute code blocks based on the result. The most common constructs are:

- `if` – Executes code if the condition is true
- `else` – Executes code if the condition is false
- `else if` or `elif` – Handles multiple conditions
- `switch` – Simplifies handling multiple exact-match cases

8.2 Simple conditional

A simple conditional allows a program to execute certain code only if a given condition is true. It is typically implemented using `if`, `if-else`, or their equivalents in various languages.




```assembly
section .data
    msg_adult db "Adult", 0xA
    msg_minor db "Minor", 0xA
section .bss
    age resb 1
section .text
    global _start
_start:
    mov eax, 17         ; age = 17
    cmp eax, 18
    jl minor_label

    ; if age >= 18
    mov edx, 6          ; length of "Adult\n"
    mov ecx, msg_adult
    jmp print

minor_label:
    mov edx, 6          ; length of "Minor\n"
    mov ecx, msg_minor

print:
    mov ebx, 1          ; stdout
    mov eax, 4          ; sys_write
    int 0x80

    mov eax, 1          ; sys_exit
    xor ebx, ebx
    int 0x80
```

Listing 8.1: Assembly

```c
#include <stdio.h>
int main() {
    int age = 17;
    if (age >= 18) {
```

```c
        printf("Adult\n");
    } else {
        printf("Minor\n");
    }
    return 0;
}
```

Listing 8.2: C

```cpp
#include <iostream>
int main() {
    int age = 17;
    if (age >= 18) {
        std::cout << "Adult" << std::endl;
    } else {
        std::cout << "Minor" << std::endl;
    }
}
```

Listing 8.3: C++

```haskell
main = do
    let age = 17
    if age >= 18
        then putStrLn "Adult"
        else putStrLn "Minor"
```

Listing 8.4: Haskell

```prolog
check_age(Age) :-
    ( Age >= 18 ->
        writeln('Adult')
    ;
        writeln('Minor')
    ).
% Example call:
% ?- check_age(17).
```

Listing 8.5: Prolog

```python
age = 17
if age >= 18:
    print("Adult")
else:
    print("Minor")
```

Listing 8.6: Python

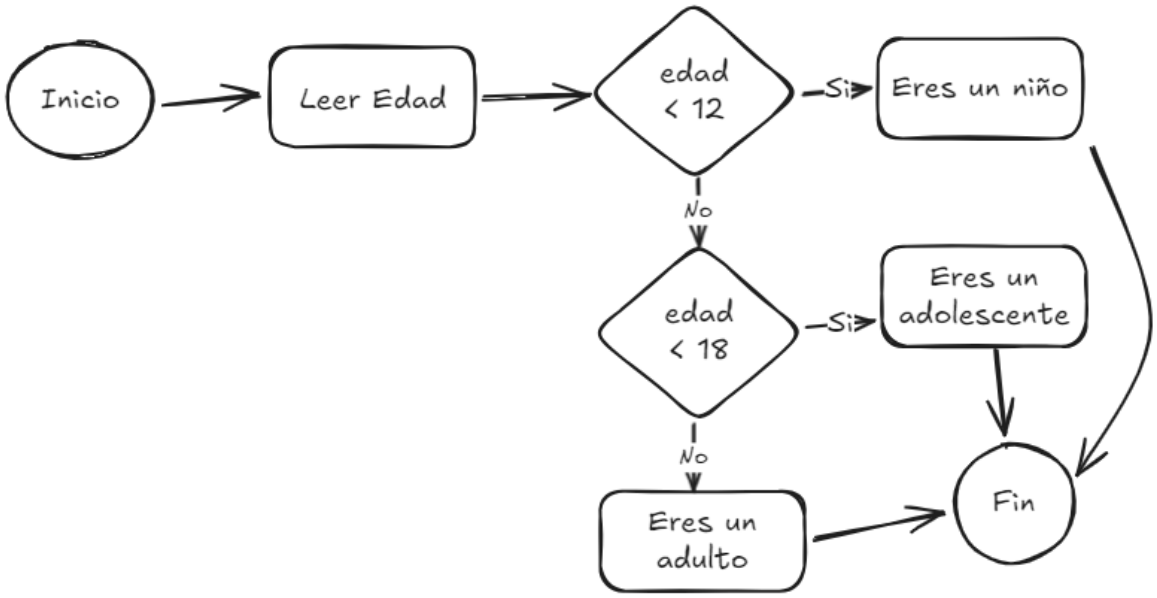
```erlang
-module(age_check).
-export([main/0]).

main() ->
    Age = 17,
    if
        Age >= 18 -> io:format("Adult~n");
        true -> io:format("Minor~n")
    end.
```

Listing 8.7: Erlang

8.3 Multiple Conditional

A multiple conditional allows a program to evaluate more than two mutually exclusive conditions. This structure is commonly implemented using `if - else if - else` or equivalent constructs, depending on the language.
In this example, we evaluate a person's age and categorize them as:

- **Child**: under 12
- **Teenager**: from 12 to 17
- **Adult**: 18 and older




```erlang
section .data
    msg_child db "Child", 0xA
    msg_teen db "Teenager", 0xA
    msg_adult db "Adult", 0xA

```

```assembly
section .text
    global _start
_start:
    mov eax, 11       ; age = 11
    cmp eax, 12
    jl print_child
    cmp eax, 18
    jl print_teen

print_adult:
    mov edx, 6
    mov ecx, msg_adult
    jmp print

print_teen:
    mov edx, 9
    mov ecx, msg_teen
    jmp print

print_child:
    mov edx, 6
    mov ecx, msg_child

print:
    mov ebx, 1
    mov eax, 4
    int 0x80
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Listing 8.8: Assembly

```c
#include <stdio.h>
int main() {
    int age = 11;
    if (age < 12) {
        printf("Child\n");
    } else if (age < 18) {
        printf("Teenager\n");
    } else {
        printf("Adult\n");
    }
    return 0;
}
```

Listing 8.9: C

```cpp
#include <iostream>
int main() {
    int age = 11;
    if (age < 12) {
        std::cout << "Child" << std::endl;
    } else if (age < 18) {
        std::cout << "Teenager" << std::endl;
    } else {
        std::cout << "Adult" << std::endl;
    }
}
```

Listing 8.10: C++

```python
age = 11
if age < 12:
    print("Child")
elif age < 18:
    print("Teenager")
else:
    print("Adult")
```

Listing 8.11: Python

```haskell
main = do
    let age = 11
    putStrLn (
        if age < 12 then "Child"
        else if age < 18 then "Teenager"
        else "Adult"
    )
```

Listing 8.12: Haskell

```prolog
check_age(Age) :-
    ( Age < 12 -> writeln('Child')
    ; Age < 18 -> writeln('Teenager')
    ; writeln('Adult')
    ).
% Example call:
% ?- check_age(11).
```

Listing 8.13: Prolog

```erlang
-module(age_check).
-export([main/0]).

main() ->
```

```
    Age = 11,
    if
        Age < 12 -> io:format("Child~n");
        Age < 18 -> io:format("Teenager~n");
        true -> io:format("Adult~n")
    end.
```

Listing 8.14: Erlang

8.4 Switch or Pattern Matching

Some languages offer a `switch` or `case` statement for cleaner handling of multiple discrete values. Others use pattern matching or function clauses.
In this example, we check an integer and print:

- **One** if the value is 1
- **Two** if the value is 2
- **Three** if the value is 3
- **Other** for all other values

```
section .data
    msg_one db "One", 0xA
    msg_two db "Two", 0xA
    msg_three db "Three", 0xA
    msg_other db "Other", 0xA

section .text
    global _start
_start:
    mov eax, 2

    cmp eax, 1
    je print_one
    cmp eax, 2
    je print_two
    cmp eax, 3
    je print_three

print_other:
    mov edx, 6
    mov ecx, msg_other
    jmp print

print_one:
    mov edx, 4
```

```
    mov ecx, msg_one
    jmp print

print_two:
    mov edx, 4
    mov ecx, msg_two
    jmp print

print_three:
    mov edx, 6
    mov ecx, msg_three

print:
    mov ebx, 1
    mov eax, 4
    int 0x80
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Listing 8.15: Assembly

```c
#include <stdio.h>
int main() {
    int val = 2;
    switch (val) {
        case 1: printf("One\n"); break;
        case 2: printf("Two\n"); break;
        case 3: printf("Three\n"); break;
        default: printf("Other\n");
    }
    return 0;
}
```

Listing 8.16: C

```cpp
#include <iostream>
int main() {
    int val = 2;
    switch (val) {
        case 1: std::cout << "One" << std::endl; break;
        case 2: std::cout << "Two" << std::endl; break;
        case 3: std::cout << "Three" << std::endl; break;
        default: std::cout << "Other" << std::endl;
    }
}
```

Listing 8.17: C++

```python
val = 2
match val:
    case 1:
        print("One")
    case 2:
        print("Two")
    case 3:
        print("Three")
    case _:
        print("Other")
```

Listing 8.18: Python

```haskell
main = do
    let val = 2
    putStrLn $ case val of
        1 -> "One"
        2 -> "Two"
        3 -> "Three"
        _ -> "Other"
```

Listing 8.19: Haskell

```prolog
print_val(1) :- writeln('One').
print_val(2) :- writeln('Two').
print_val(3) :- writeln('Three').
print_val(_) :- writeln('Other').

% Example call:
% ?- print_val(2).
```

Listing 8.20: Prolog

```erlang
-module(switch_case).
-export([main/0]).

main() ->
    Val = 2,
    case Val of
        1 -> io:format("One~n");
        2 -> io:format("Two~n");
        3 -> io:format("Three~n");
        _ -> io:format("Other~n")
    end.
```

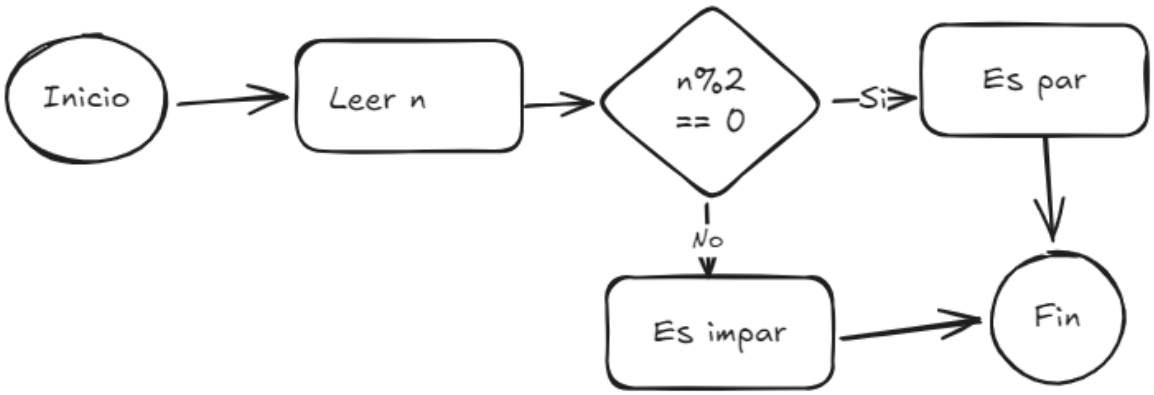
Listing 8.21: Erlang

8.5 Ternary Conditional (If Expression)

The ternary conditional (or inline `if`) is a concise way to choose between two values based on a condition. It has the form:

```
condition ? value_if_true : value_if_false
```

This is supported in many programming languages, either as a built-in ternary operator or via expression syntax.




```
section .data
    even_msg db "Even", 0xA
    odd_msg db "Odd", 0xA

section .text
    global _start
_start:
    mov eax, 4          ; number = 4
    and eax, 1          ; check if LSB is 0
    jz is_even

    ; odd case
    mov ecx, odd_msg
    mov edx, 4
    jmp print

is_even:
    mov ecx, even_msg
    mov edx, 5

print:
    mov ebx, 1
    mov eax, 4
    int 0x80
    mov eax, 1
```

```
    xor ebx, ebx
    int 0x80
```

Listing 8.22: Assembly (Simulated Ternary)

```c
#include <stdio.h>
int main() {
    int num = 4;
    const char* result = (num % 2 == 0) ? "Even" : "Odd";
    printf("%s\n", result);
    return 0;
}
```

Listing 8.23: C

```cpp
#include <iostream>
int main() {
    int num = 4;
    std::cout << (num % 2 == 0 ? "Even" : "Odd") << std::endl;
}
```

Listing 8.24: C++

```python
num = 4
print("Even" if num % 2 == 0 else "Odd")
```

Listing 8.25: Python

```haskell
main = do
    let num = 4
    putStrLn (if even num then "Even" else "Odd")
```

Listing 8.26: Haskell

```prolog
check_even(Number) :-
    (Number mod 2 =:= 0 -> Result = 'Even' ; Result = 'Odd'),
    writeln(Result).

% Example call:
% ?- check_even(4).
```

Listing 8.27: Prolog

```erlang
-module(even_odd).
-export([main/0]).
main() ->
    Num = 4,
    Result = if Num rem 2 =:= 0 -> "Even"; true -> "Odd" end,
    io:format("~s~n", [Result]).
```

Listing 8.28: Erlang

8.6 Best Practices for Conditionals (Multi-language)

Writing conditionals clearly and efficiently helps keep code readable, maintainable, and bug-free. Below are key principles illustrated across multiple programming languages.

8.6.1 Clarity and Readability

- Use descriptive variable names.
- Avoid complex or nested negations.
- Use conditionals as expressions if the language allows.

Examples:

```c
int is_adult = age >= 18;
if (is_adult) {
    printf("Access granted.\n");
}
```

Listing 8.29: C

```python
is_adult = age >= 18
if is_adult:
    print("Access granted.")
```

Listing 8.30: Python

```haskell
let isAdult = age >= 18
in if isAdult then putStrLn "Access granted." else return ()
```

Listing 8.31: Haskell

```prolog
check_age(Age) :-
    (Age >= 18 -> writeln('Access granted'); writeln('Denied')).
```

Listing 8.32: Prolog

```erlang
check_age(Age) ->
    case Age >= 18 of
        true -> io:format("Access granted~n");
        false -> io:format("Denied~n")
    end.
```

Listing 8.33: Erlang

8.6.2 Multiple Conditions

- Combine conditions using logical operators (`&&`/`and`, `||`/`or`, `!`/`not`).
- Use parentheses to clarify grouping.

Examples:

```cpp
if ((age >= 18 && hasID) || hasParentalPermission) {
    std::cout << "Entry allowed.\n";
}
```

Listing 8.34: C++

```python
if (age >= 18 and has_id) or has_parental_permission:
    print("Entry allowed.")
```

Listing 8.35: Python

```haskell
if (age >= 18 && hasID) || hasPermission then putStrLn "Entry allowed.
    "
```

Listing 8.36: Haskell

```erlang
case (Age >= 18 andalso HasID) orelse HasPermission of
    true -> io:format("Entry allowed~n");
    false -> io:format("Denied~n")
end.
```

Listing 8.37: Erlang

8.6.3 Avoid Deep Nesting

- Refactor overly nested conditionals.
- Use guard clauses or helper functions.

Examples:

```python
if user:
    if user.is_active:
        if user.has_permission("admin"):
            print("Access")
```

Listing 8.38: Too Nested

```python
if not user or not user.is_active:
    return
if user.has_permission("admin"):
    print("Access")
```

Listing 8.39: Refactored

8.6.4 Performance and Structure

- Order conditions from most likely to least.
- Use `switch` (or case matching) for constant-value comparisons.

Examples:

```cpp
switch (input) {
    case 1: std::cout << "Start\n"; break;
    case 2: std::cout << "Settings\n"; break;
    default: std::cout << "Exit\n"; break;
}
```

Listing 8.40: C++ Switch

```haskell
case input of
    1 -> putStrLn "Start"
    2 -> putStrLn "Settings"
    _ -> putStrLn "Exit"
```

Listing 8.41: Haskell Case

```erlang
case Input of
    1 -> io:format("Start~n");
    2 -> io:format("Settings~n");
    _ -> io:format("Exit~n")
end.
```

Listing 8.42: Erlang Case

Chapter 9

Repetition Structures: Loops

Repetition structures, also known as loops, are used in programming to repeat a block of instructions as long as a specified condition remains true. Loops are fundamental for tasks such as iterating over data, performing calculations, or automating repetitive tasks.

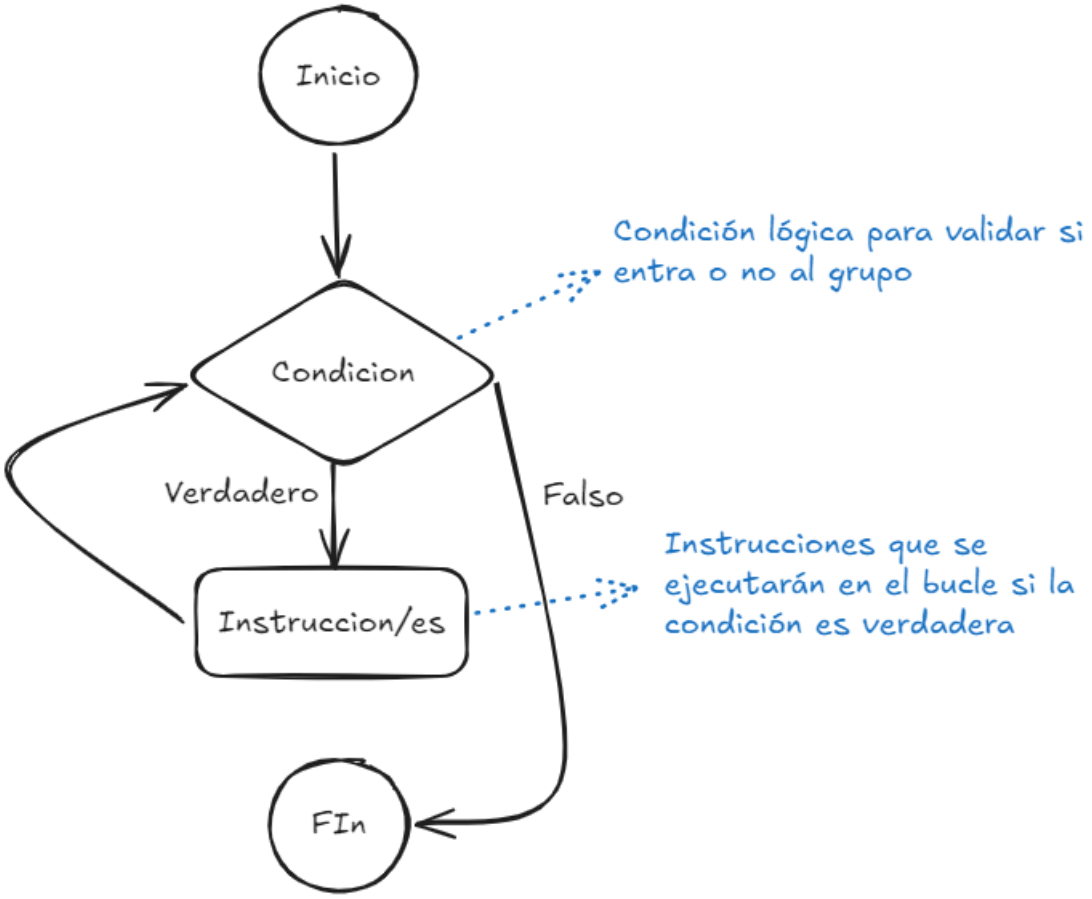
9.1 Types of Loops

Common types of loops include:

- **While loops**: Repeats while the condition is true.
- **Do-while loops**: Executes the block at least once, then checks the condition.
- **For loops**: Ideal for counting and iterating over ranges or collections.

9.2 while Loop (Multi-language)

The `while` loop executes a block of code repeatedly **as long as a given condition is true**. The condition is checked **before each iteration**, meaning the loop may not run at all if the condition is initially false.




```c
#include <stdio.h>
int main() {
    int i = 1;
    while (i <= 5) {
        printf("%d\n", i);
        i++;
    }
    return 0;
}
```

Listing 9.1: C

```cpp
#include <iostream>
int main() {
    int i = 1;
    while (i <= 5) {
        std::cout << i << std::endl;
        i++;
    }
}
```

Listing 9.2: C++

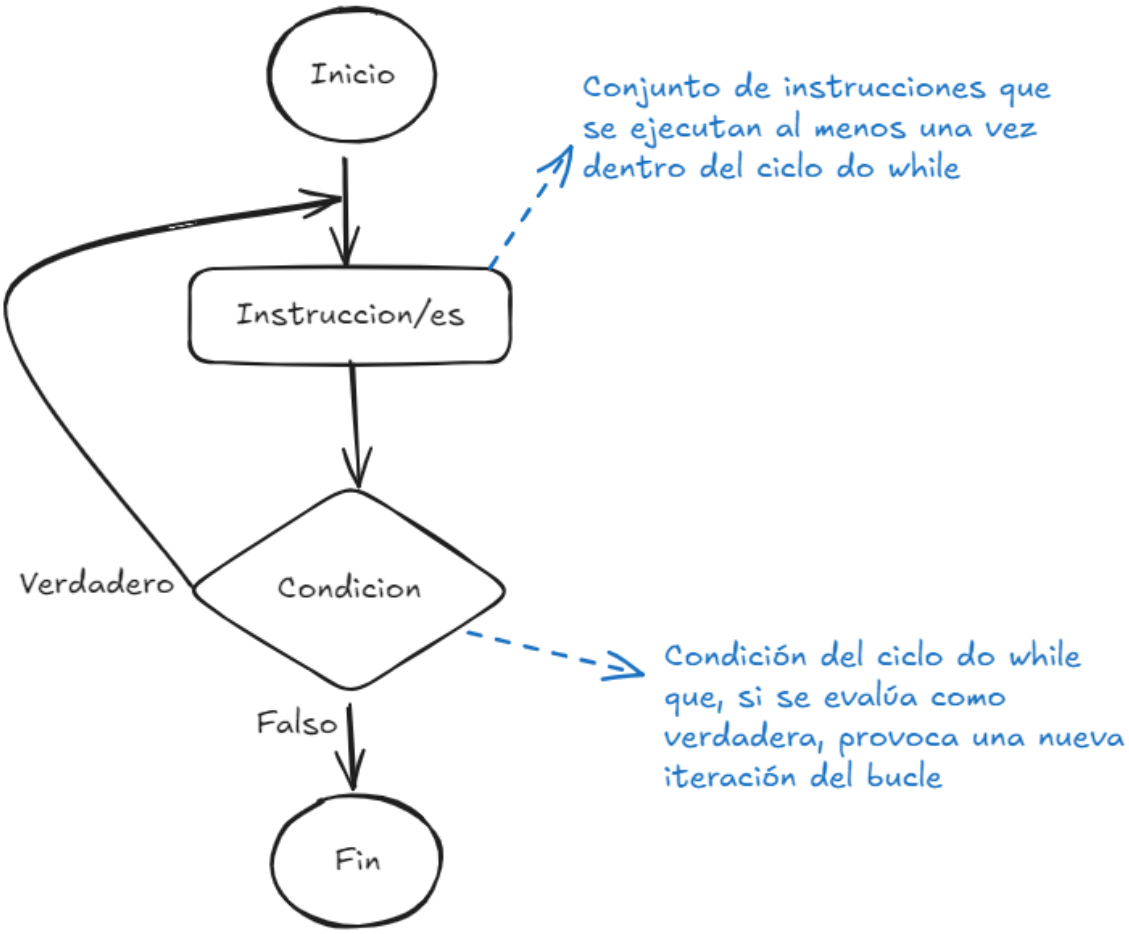
```python
i = 1
while i <= 5:
    print(i)
    i += 1
```

Listing 9.3: Python

> **Note**
>
> In some programming languages, the `while` loop does not exist or does not fit well within the language's paradigm. Instead, other control structures are used to create loops.

9.3 do-while Loop (Multi-language)

The `do-while` loop executes a block of code **at least once**, and then continues executing it as long as a specified condition remains true. The condition is evaluated **after** each iteration.




```c
#include <stdio.h>
int main() {
```

```c
    int i = 1;
    do {
        printf("%d\n", i);
        i++;
    } while (i <= 5);
    return 0;
}
```

Listing 9.4: C

```cpp
#include <iostream>
int main() {
    int i = 1;
    do {
        std::cout << i << std::endl;
        i++;
    } while (i <= 5);
}
```

Listing 9.5: C++

```python
i = 1
while True:
    print(i)
    i += 1
    if i > 5:
        break
```

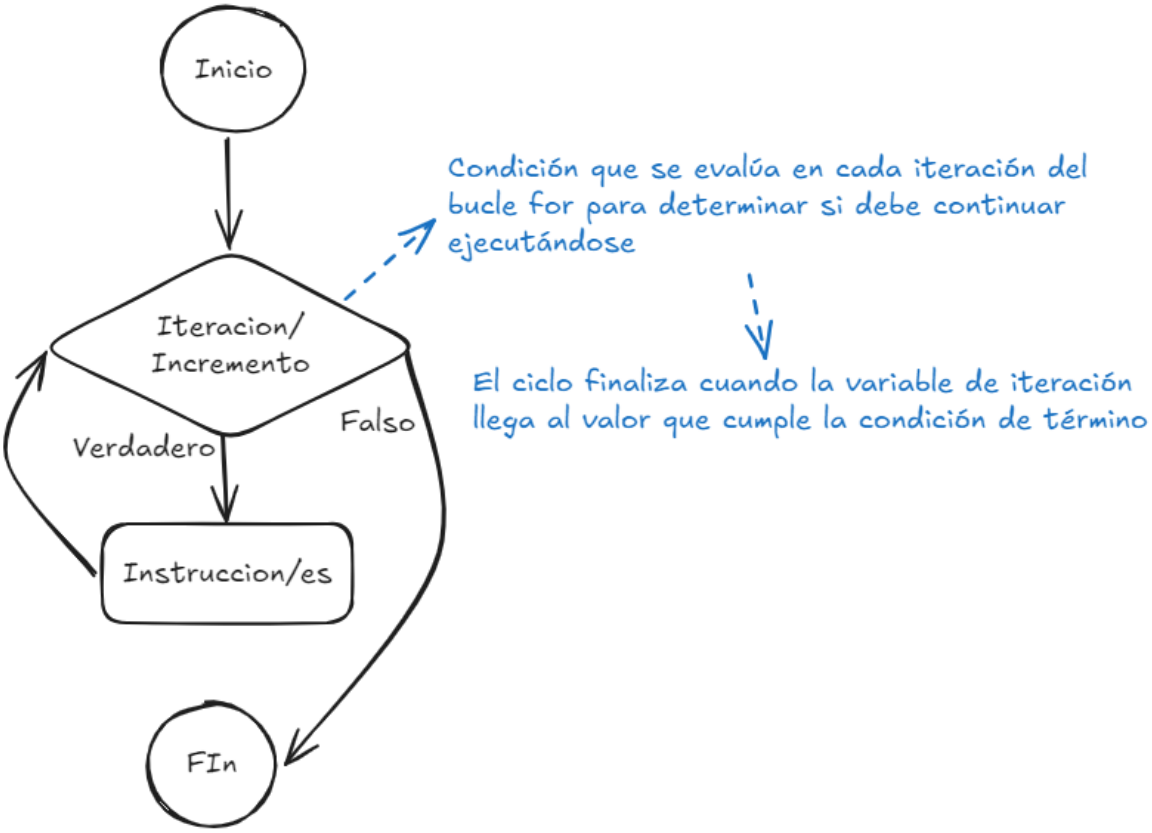
Listing 9.6: Python (emulated do-while)

Note

In some programming languages, the `do-while` loop does not exist or does not fit well within the language's paradigm. Instead, other control structures are used to create loops.

9.4 for Loop (Multi-language)

The `for` loop is a control structure that allows a concise way to iterate over a range or collection. It typically includes an initializer, a condition, and an increment/decrement expression.




```c
#include <stdio.h>
int main() {
    for (int i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Listing 9.7: C

```cpp
#include <iostream>
int main() {
    for (int i = 1; i <= 5; i++) {
        std::cout << i << std::endl;
    }
}
```

Listing 9.8: C++

```python
for i in range(1, 6):
    print(i)
```

Listing 9.9: Python

Note

In some programming languages like Prolog, Haskell, and Erlang, traditional `for` loops do not exist. Instead, recursion or functional constructs such as list comprehensions are used to achieve similar iteration.

9.5 foreach Loop (Multi-language)

The `foreach` loop is used to iterate directly over elements of a collection (such as arrays, lists, or maps). It simplifies code by abstracting away index management.

```cpp
#include <iostream>
#include <vector>
int main() {
    std::vector<int> nums = {10, 20, 30, 40, 50};
    for (int num : nums) {
        std::cout << num << std::endl;
    }
}
```

Listing 9.10: C++

```python
nums = [10, 20, 30, 40, 50]
for num in nums:
    print(num)
```

Listing 9.11: Python

```haskell
main = mapM_ print [10, 20, 30, 40, 50]
```

Listing 9.12: Haskell

```erlang
-module(foreach).
-export([start/0]).

start() -> lists:foreach(fun(X) -> io:format("~p~n", [X]) end, [10,
    20, 30, 40, 50]).
```

Listing 9.13: Erlang

Note

Languages like C, Assembly, and Prolog do not natively support `foreach`. In these cases, iteration is performed using manual indexing (C, Assembly) or recursion (Prolog).

9.6 Control Flow Statements (Multi-language)

Control flow statements are used to alter the standard sequence of instructions. While some, like `break` or `continue`, are common in imperative languages, others like `goto` or `jmp` are lower-level or discouraged in modern programming.

Statement	Purpose	Supported In
`break`	Exit the nearest loop	C, C++, Python, (simulated in others)
`continue`	Skip rest of loop iteration	C, C++, Python
`goto`	Jump to labeled line	C, C++
`jmp`	Unconditional jump (Assembly)	Assembly
`return`	Exit a function	All languages

```c
#include <stdio.h>
int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 2) continue;  // Skip 2
        if (i == 4) break;     // Stop loop
        printf("%d\n", i);
    }

    goto skip;
    printf("Skipped by goto\n");
skip:
    printf("Reached label\n");
    return 0;
}
```

Listing 9.14: C

```cpp
#include <iostream>
int main() {
    for (int i = 0; i < 5; i++) {
        if (i == 2) continue;
        if (i == 4) break;
        std::cout << i << std::endl;
    }

    goto label;
    std::cout << "This won't print\n";
label:
    std::cout << "Label reached\n";
}
```

Listing 9.15: C++

```python
for i in range(5):
    if i == 2:
        continue
    if i == 4:
        break
    print(i)
```

Listing 9.16: Python

```assembly
section .text
    global _start
_start:
    mov eax, 1
    cmp eax, 1
    je skip
    mov eax, 2
skip:
    ; now eax = 1
    ; no structured break/continue
```

Listing 9.17: Assembly (jmp)

Note

Languages like Haskell, Prolog, and Erlang do not use `break`, `continue`, or `goto`. Instead, they rely on pattern matching, recursion, and guards for control flow. Assembly lacks structured control flow, relying solely on conditional or unconditional jumps like `jmp`, `je`, `jne`, etc.

9.7 Comprehensions (Comprensiones)

Comprehensions are concise syntactic constructs for generating, filtering, and transforming collections. They are common in functional and high-level languages.

9.7.1 What are comprehensions?

A comprehension creates a new list (or collection) based on an existing one, applying optional filters or transformations.

9.7.2 Common Use Cases

- Filtering elements
- Applying transformations
- Nested iteration (e.g., matrix flattening)

```python
# List comprehension
evens_squared = [x**2 for x in range(1, 11) if x % 2 == 0]
print(evens_squared)
```

Listing 9.18: Python

```haskell
-- List comprehension
evensSquared = [x^2 | x <- [1..10], even x]
main = print evensSquared
```

Listing 9.19: Haskell

```erlang
-module(comp).
-export([start/0]).

start() ->
    Result = [X*X || X <- lists:seq(1,10), X rem 2 == 0],
    io:format("~p~n", [Result]).
```

Listing 9.20: Erlang (list comprehension)

```cpp
// Requires C++20 or higher
#include <iostream>
#include <vector>
#include <ranges>
using namespace std;

int main() {
    vector<int> nums = {1,2,3,4,5,6,7,8,9,10};
    for (int x : nums | views::filter([](int i){ return i % 2 == 0; })
                      | views::transform([](int i){ return i * i; }))
    {
        cout << x << " ";
    }
}
```

Listing 9.21: C++ (with ranges)

Note

Languages like Python, Haskell, and Erlang support true list comprehensions. In C/C++ and Prolog, similar behavior is achieved via loops or constructs like `findall`. In Assembly, comprehension must be built manually through loops and conditionals.

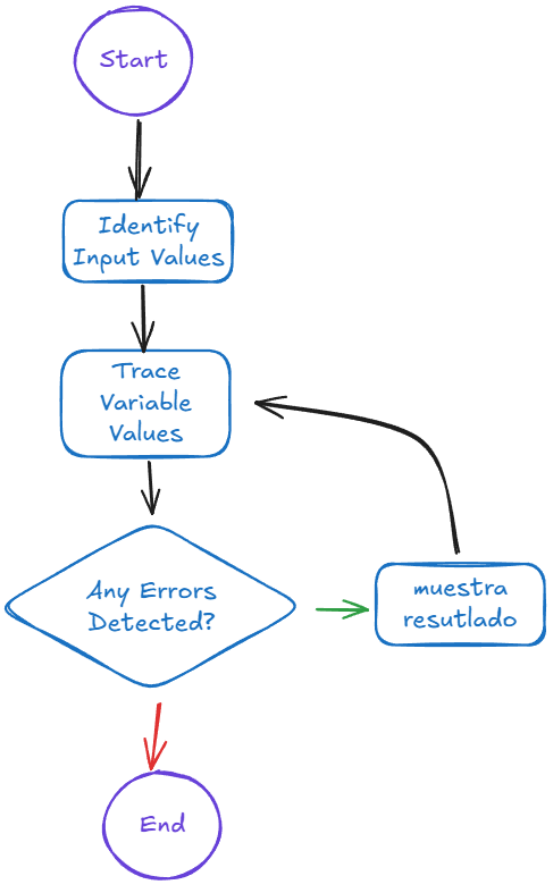
Chapter 10

Desk Checking: Manual Trace of Algorithms

Desk checking is the process of manually simulating the execution of an algorithm to verify its correctness. This method is especially useful during the design phase, as it allows early detection of logic errors before implementation.

10.1 What is Desk Checking?

Desk checking involves tracing the values of variables and the flow of control step by step on paper or a worksheet. It helps programmers understand the behavior of the algorithm for specific input cases.



Let us consider the following example to manually trace a simple loop that calculates the factorial of a number.

10.1.1 Algorithm: Factorial of a Number (Iterative)

```
int n = 5;
int fact = 1;
for (int i = 1; i <= n; i++) {
    fact *= i;
}
```

Iteration	i	fact
Initial	–	1
1	1	1
2	2	2
3	3	6
4	4	24
5	5	120

Table 10.1: Desk Checking Trace Table for Factorial Algorithm

10.1.2 Other examples

```
n = 5
fact = 1
for i in range(1, n+1):
    fact *= i
print(fact)
```

```
int fact = 1;
for (int i = 1; i <= 5; ++i) {
    fact *= i;
}
std::cout << fact;
```

```
factorial n = product [1..n]
```

```
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, F1),
    F is N * F1.
```

```
factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

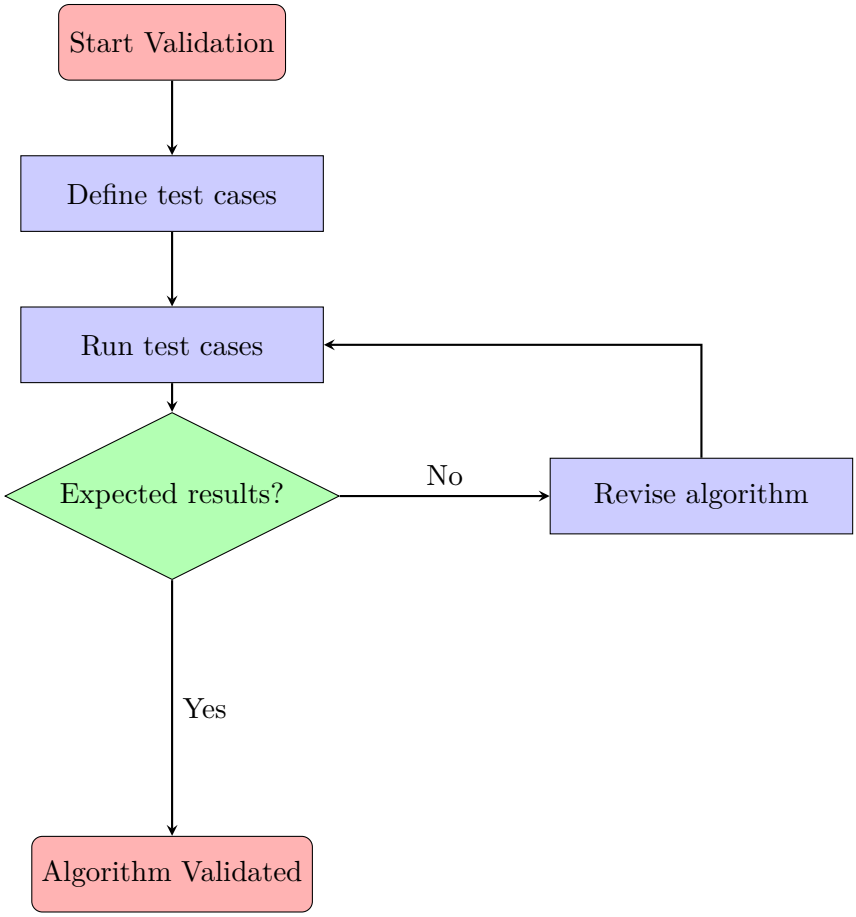
Chapter 11

Algorithm Validation

Algorithm validation is the process of ensuring that an algorithm not only works, but also solves the intended problem correctly, efficiently, and within defined constraints.

11.1 What is Algorithm Validation?

Validation verifies whether the algorithm meets the problem's requirements and behaves as expected with different input cases. It is distinct from verification, which focuses on whether the algorithm was implemented correctly.



11.2 Validation Techniques

- **Test Case Design:** Include typical, boundary, and incorrect cases.
- **Trace Tables:** Useful for step-by-step analysis.
- **Assertions:** Conditions embedded within the algorithm.
- **Formal Methods:** Mathematical proofs of correctness.

11.3 Example: Validate a Palindrome Algorithm

A palindrome is a word or number that reads the same forward and backward.

11.3.1 Algorithm Description (C Language)

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

bool is_palindrome(const char* str) {
    int len = strlen(str);
    for (int i = 0; i < len / 2; i++) {
        if (str[i] != str[len - i - 1])
            return false;
    }
    return true;
}
```

11.3.2 Validation Cases

- Input: "racecar" → Expected: `true`
- Input: "hello" → Expected: `false`
- Input: "" (empty string) → Expected: `true`

11.3.3 Multilingual Implementation Examples

```python
def is_palindrome(s):
    return s == s[::-1]
```

Listing 11.1: Python

```cpp
bool isPalindrome(const std::string& s) {
    return std::equal(s.begin(), s.begin() + s.size()/2, s.rbegin());
}
```

Listing 11.2: C++

```
isPalindrome s = s == reverse s
```

Listing 11.3: Haskell

```
palindrome(X) :- reverse(X, X).
```

Listing 11.4: Prolog

```
is_palindrome(S) -> S == lists:reverse(S).
```

Listing 11.5: Erlang

Chapter 12

One-Dimensional Arrays

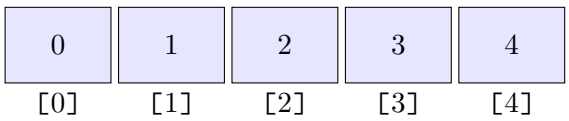
One-dimensional arrays are fundamental data structures that store a sequence of elements of the same type. They are used for storing lists, sequences, and for implementing more complex structures.

12.0.1 Main Characteristics

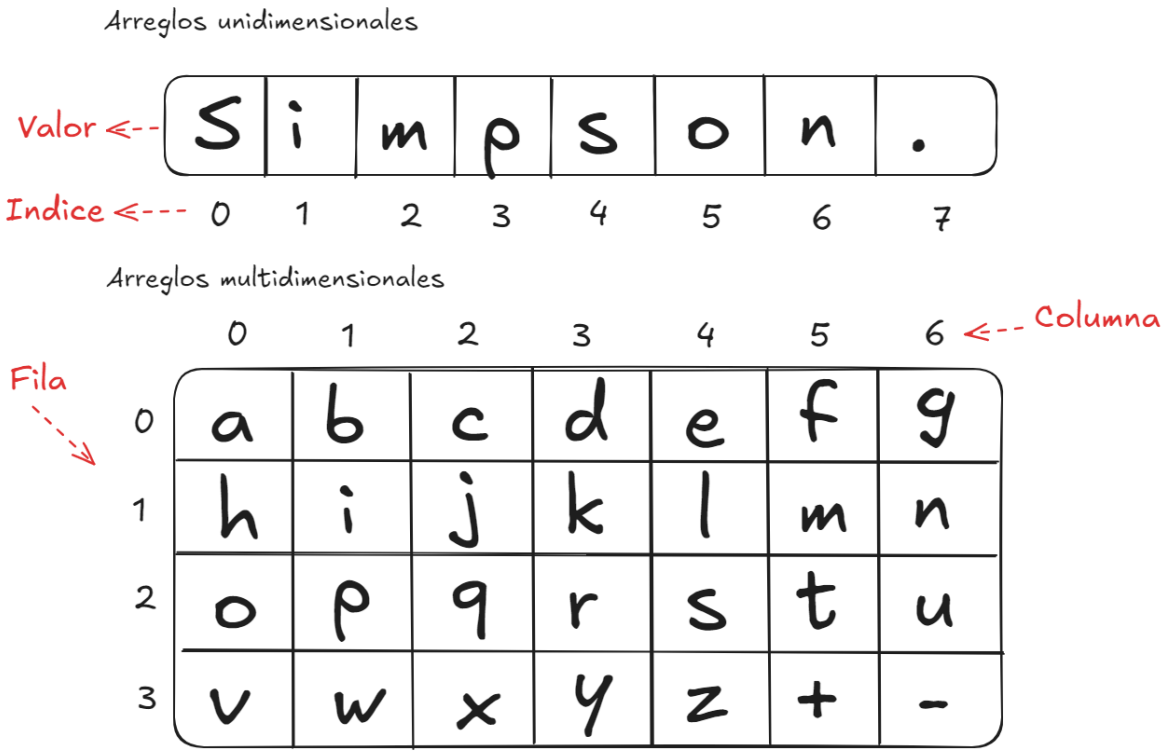
- **Random access**: Allows direct access to any element by its index in constant time $O(1)$.
- **Fixed size**: In many low-level languages, the size must be defined at compile time.
- **Contiguous memory**: Elements are stored in adjacent memory blocks, allowing efficient access.
- **Homogeneity**: All elements must be of the same data type.
- **Indexing**: Indexing typically starts at 0 (as in C, C++, Java, Python) or at 1 (as in MATLAB, R, Fortran).

12.0.2 Memory Representation Diagram

The following table shows a visual representation of a one-dimensional array with 5 integer elements:



Each cell represents a contiguous memory location storing an integer value. The index is used to quickly access any element in the array.



12.0.3 Basic Operations

- **Initialization**
- **Traversal**
- **Searching**
- **Sorting**
- **Updating values**

12.1 One-Dimensional Arrays (Vectors)

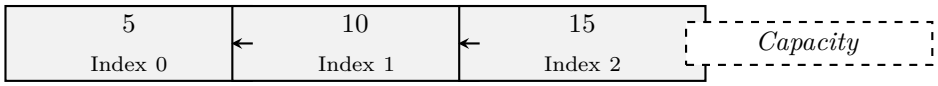
Vectors are fixed-size arrays. Each element is accessed by a zero-based index.

Main Characteristics:

- Random access
- Fixed size
- Contiguous memory
- Homogeneous elements

12.2 Dynamic Lists (Dynamic Arrays)

Dynamic arrays (e.g., ‘ArrayList‘ in Java or ‘vector‘ in C++) allow resizing during runtime.

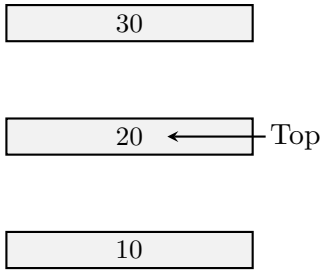


Main Characteristics:

- Random access
- Dynamic resizing
- Contiguous memory
- Homogeneous elements
- Amortized insertion at end

12.3 Stack (LIFO)

A **stack** is a linear data structure that follows the *Last In, First Out* (LIFO) principle. Elements are added and removed from the top.

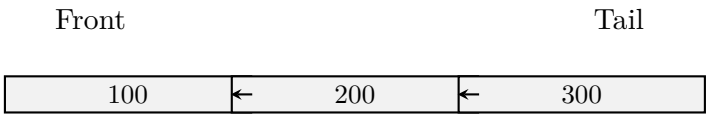


Main Characteristics:

- Access to top element
- LIFO behavior
- Constant time push/pop
- Used for recursion, parsing, undo operations

12.4 Queue (FIFO)

A **queue** is a linear data structure that follows the *First In, First Out* (FIFO) principle. Elements are added at the rear and removed from the front.



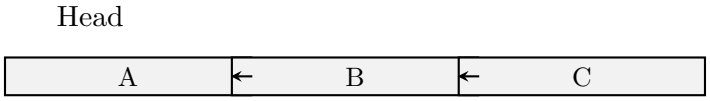
Main Characteristics:

- FIFO behavior
- Enqueue at rear, dequeue from front
- Constant time enqueue/dequeue
- Used in scheduling, buffering, simulations

12.5 Linked Lists

A **linked list** is a collection of nodes where each node contains data and a reference to the next (and possibly previous) node.

12.5.1 Singly Linked List

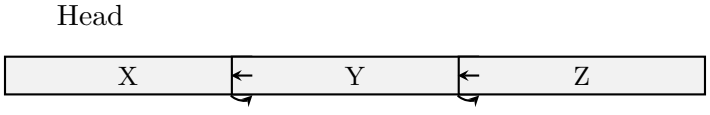


Main Characteristics:

- Dynamic size
- Non-contiguous memory
- Insertion/deletion at head
- Memory overhead (pointers)

12.5.2 Doubly Linked List

Each node has two pointers: one to the next node and one to the previous.



Additional Features:

- Bidirectional traversal
- Easier deletion of arbitrary nodes
- More memory usage due to extra pointer

12.6 Examples

```
section .data
my_array dd 10, 20, 30, 40, 50  ; 5 integers (DWORDs)
```

Listing 12.1: Assembly

```
#include <stdio.h>

int main() {
    int my_array[5] = {10, 20, 30, 40, 50};
    return 0;
}
```

Listing 12.2: C

```
#include <iostream>
#include <vector>

int main() {
    int my_array[5] = {10, 20, 30, 40, 50}; // Static array

    std::vector<int> my_vector = {10, 20, 30, 40, 50}; // Dynamic
    array
    return 0;
}
```

Listing 12.3: C++

```
import Data.Array

myArray :: Array Int Int
myArray = array (0, 4) [(0,10), (1,20), (2,30), (3,40), (4,50)]
```

Listing 12.4: Haskell

```
my_array([10, 20, 30, 40, 50]).

element_at(Index, List, Element) :- nth0(Index, List, Element).
```

Listing 12.5: Prolog

```
my_array = [10, 20, 30, 40, 50]
```

Listing 12.6: Python

```
MyArray = [10, 20, 30, 40, 50].
MyArray = {10, 20, 30, 40, 50}.
Array = array:from_list([10, 20, 30, 40, 50]).
```

Listing 12.7: Erlang

12.7 Common Errors

- Accessing out-of-bounds indices
- Forgetting zero-based indexing
- Not initializing the array

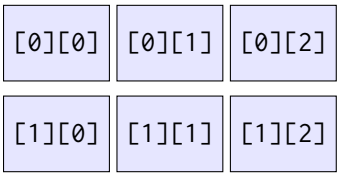
Chapter 13

Multidimensional Arrays

Multidimensional arrays are data structures that allow storing elements in multiple dimensions, extending the concept of one-dimensional arrays. They are widely used in scientific computing, artificial intelligence, image processing, and linear algebra.

13.1 Definition and Declaration

A two-dimensional array is an array of arrays. Each element is accessed using two indices: one for the row and one for the column.



13.2 Matrices (2D Arrays)

A **matrix** is a two-dimensional array that organizes data into rows and columns. Each element is accessed using two indices: one for the row and one for the column.

Visual Example:

$$M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Main Characteristics:

- **Random access** Constant time $O(1)$
- **Contiguous memory layout** (e.g., row-major in C/C++, column-major in Fortran)
- **Homogeneous data types**
- **Straightforward mathematical operations**

13.3 Sparse Matrices

A **sparse matrix** is one in which most of the elements are zero. To optimize memory usage and computational performance, only the non-zero elements and their positions are stored.

13.3.1 Advantages:

- **Memory-efficient**
- **Faster operations in specific cases**
- **Ideal for graphs, sparse systems, etc.**

13.3.2 Common Representations:

- **COO (Coordinate list)**: list of tuples (row, column, value)
- **CSR (Compressed Sparse Row)**: arrays for values, column indices, and row offsets
- **CSC (Compressed Sparse Column)**: similar to CSR, but column-oriented

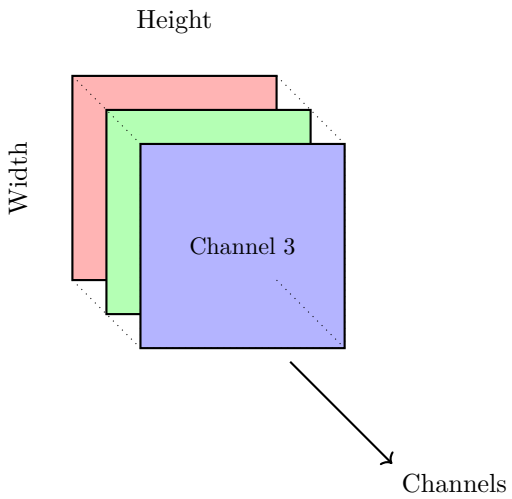
13.3.3 Example (COO format):

```
Row:    [0, 1, 3]
Column: [2, 0, 1]
Value:  [5, 8, 3]
```

This represents a 4x4 matrix with only three non-zero elements.

13.4 Tensors

A **tensor** is a generalization of matrices to n dimensions. While a matrix is 2D, tensors can be 3D, 4D, or higher, depending on the data and application.



13.4.1 Examples of Use:

- **RGB Image**: 3D tensor (height × width × channels)
- **Video**: 4D tensor (frames × height × width × channels)
- **Deep learning models**: 4D to 5D+ tensors

13.4.2 Characteristics:

- **Supports complex, high-dimensional data**
- **Requires specialized libraries**(e.g., NumPy, TensorFlow, PyTorch)
- **Widely used in ML and scientific computing**

13.5 Examples

```
section .data
; 3x3 matrix (row-major)
matrix:     dd 1,2,3, 4,5,6, 7,8,9

; Simulated tensor 2x2x2
tensor:     dd 1,2,3,4, 5,6,7,8

; Sparse matrix: represent with coordinates and values
sparse_rows:    dd 0,1,2
sparse_cols:    dd 2,0,1
sparse_vals:    dd 5,8,3
```

Listing 13.1: Assembly

Note

Assembly does not support multidimensional arrays or sparse matrices natively. All structures must be manually emulated by laying out data in memory and using calculated offsets.

```
#include <stdio.h>

int matrix[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

int tensor[2][2][2] = {
    {{1, 2}, {3, 4}},
    {{5, 6}, {7, 8}}
```

```
};

int sparse_rows[] = {0, 1, 3};
int sparse_cols[] = {2, 0, 1};
int sparse_vals[] = {5, 8, 3};
```

Listing 13.2: C

Note

C supports static multidimensional arrays and allows manual representation of sparse matrices using parallel arrays or structs. However, it lacks built-in abstractions for tensors or sparse matrices—these must be manually implemented or handled using libraries like CSparse, Eigen, or BLAS.

```
#include <vector>

int main() {
    std::vector<std::vector<int>> matrix = {
        {1,2,3}, {4,5,6}, {7,8,9}
    };

    std::vector<std::vector<std::vector<int>>> tensor = {
        { {1,2}, {3,4} },
        { {5,6}, {7,8} }
    };

    std::vector<int> rows = {0,1,3};
    std::vector<int> cols = {2,0,1};
    std::vector<int> values = {5,8,3};

    return 0;
}
```

Listing 13.3: C++

Note

C++ allows nested std::vector or std::array for dynamic multidimensional arrays. Sparse matrices and tensors are often handled through external libraries such as Eigen, Boost uBLAS, or TensorFlow C++ API. Native syntax becomes verbose for higher dimensions.

```
matrix :: [[Int]]
matrix = [[1,2,3], [4,5,6], [7,8,9]]

tensor :: [[[Int]]]
```

```haskell
tensor = [[[1,2],[3,4]], [[5,6],[7,8]]]

sparse :: [(Int, Int, Int)]
sparse = [(0,2,5), (1,0,8), (3,1,3)]
```

Listing 13.4: Haskell

Note

Haskell supports nested lists and abstract data types to model matrices and tensors. However, for performance and large-scale data, libraries like array, vector, or repa are needed. Sparse matrices are typically represented using lists of (row, col, value) triples. Functional paradigms make manipulation elegant but require additional care for efficiency.

```prolog
matrix([
    [1,2,3],
    [4,5,6],
    [7,8,9]
]).

tensor([
    [[1,2],[3,4]],
    [[5,6],[7,8]]
]).

sparse([
    (0, 2, 5),
    (1, 0, 8),
    (3, 1, 3)
]).
```

Listing 13.5: Prolog

Note

Prolog does not natively support arrays. Multidimensional data must be represented using nested lists or compound terms. Sparse matrices are usually modeled as lists of tuples (Row, Column, Value). No inherent support for high-dimensional tensors.

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

tensor = [
```

```
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]]
]

sparse = {
    'rows': [0, 1, 3],
    'cols': [2, 0, 1],
    'data': [5, 8, 3]
}
```

Listing 13.6: Python

Note

Python natively supports multidimensional arrays using nested lists, and more efficiently via libraries like NumPy (for dense arrays and tensors) and SciPy (for sparse matrices). It is the most flexible among general-purpose languages for handling multidimensional data structures.

```
-module(arrays).
-export([matrix/0, tensor/0, sparse/0]).

matrix() ->
    [[1,2,3], [4,5,6], [7,8,9]].

tensor() ->
    [[[1,2],[3,4]], [[5,6],[7,8]]].

sparse() ->
    [{0,2,5}, {1,0,8}, {3,1,3}].
```

Listing 13.7: Erlang

Note

Erlang does not have built-in arrays. Multidimensional arrays must be simulated using nested lists or tuples. It's common to represent sparse data as lists of coordinate-value triples. Erlang is not optimized for matrix-heavy or tensor-heavy computation.

13.6 Dictionaries and Hash Tables

A **dictionary** (or **hash table**) is a data structure that stores *key-value pairs* and allows fast data retrieval based on unique keys. It is commonly implemented using a hash function to map keys to indices in an internal array.

13.6.1 Characteristics

- **Key Uniqueness**: Each key maps to exactly one value.
- **Average Time Complexity**: Access, insertion, and deletion in $O(1)$ time.
- **Hash Function**: Converts keys into indices.
- **Collision Resolution**: Typically via chaining or open addressing.

13.6.2 Language-Specific Implementations

Note Assembly (x86 NASM)

- No native hash table support.
- Must simulate hashing, indexing, and collision handling manually using memory and registers.

```c
typedef struct {
    char* key;
    int value;
} Entry;

Entry hashtable[10];

int hash(const char* key) {
    int h = 0;
    while (*key) h += *key++;
    return h % 10;
}
```

Listing 13.8: C

Note

Manual implementation required. Libraries such as `uthash` are often used. sing memory and registers.

```cpp
#include <unordered_map>

std::unordered_map<std::string, int> dict;
dict["apple"] = 5;
dict["banana"] = 2;
```

Listing 13.9: C++

Note

Use `std::unordered_map` for average $O(1)$ performance.

```python
dict = {"apple": 5, "banana": 2}
```

Listing 13.10: Python

Note

Built-in support with efficient memory layout and fast operations.

```prolog
value(apple, 5).
value(banana, 2).
```

Listing 13.11: Prolog

Note

Emulated via facts or association lists. SWI-Prolog provides `dict` and `assoc` libraries.

```erlang
Dict = #{apple => 5, banana => 2}.
```

Listing 13.12: Erlang

Note

Maps introduced in Erlang 17+. Immutable, functional.

```haskell
import qualified Data.Map as M
dict = M.fromList [("apple", 5), ("banana", 2)]
```

Listing 13.13: Haskell

Note

Use `Data.Map` or `Data.HashMap.Strict` from `unordered-containers` for hash tables.

Chapter 14

Functions and Parameters

In mathematics, a **function** is a relation between two sets: a *domain* and a *codomain*, where each element of the domain is associated with exactly one element of the codomain:

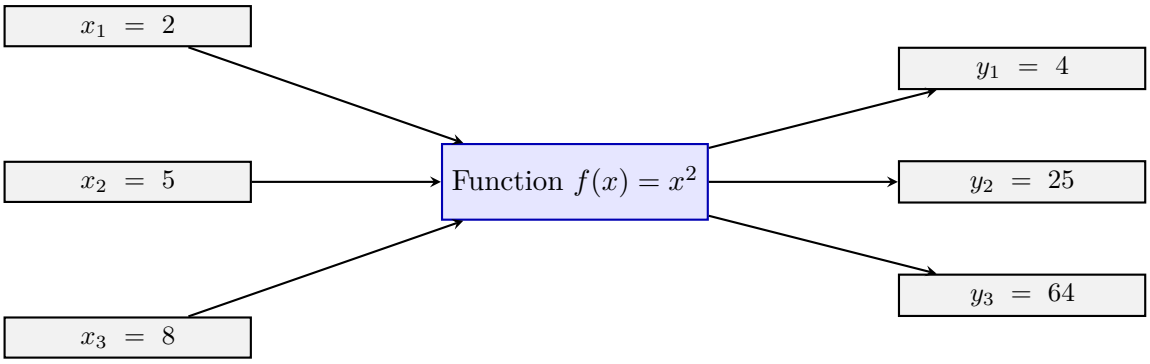
$$f : X \to Y \quad \text{such that} \quad \forall x \in X, \exists! y \in Y : f(x) = y$$

In programming, this concept translates to: fundamental components in programming that allow modular design, code reuse, and abstraction. A function encapsulates a sequence of instructions that can be executed by calling the function's name and providing the required parameters.

- A reusable block of code.
- It maps input parameters (domain) to output results (codomain).
- It encapsulates logic, supports modularity, and may have side effects (in imperative languages).

14.1 Function Classification in Programming

- **Pure functions**: Have no side effects. Always return the same result for the same parameters.
- **Impure functions**: May alter global state or perform input/output operations.
- **Recursive functions**: Call themselves to solve subproblems.
- **Higher-order functions**: Take other functions as parameters or return functions.
- **Lambda/anonymous functions**: Functions without explicit names, common in functional programming.



14.2 Parameters and Return Values

- **Parameters**: Inputs passed to a function.
- **Return value**: The output computed by the function.
- ***Void functions***: Do not return a value.

14.3 Function Definitions in Various Programming Languages

14.3.1 C++ Example

```
add_numbers:
    mov eax, [esp+4]
    add eax, [esp+8]
    ret
```

Listing 14.1: Assembly

```
int sum(int a, int b) {
    return a + b;
}
```

Listing 14.2: C

```
auto sum(int a, int b) -> int {
    return a + b;
}
```

Listing 14.3: C++

```
sum a b = a + b
```

Listing 14.4: Haskell

```
sum(A, B, R) :- R is A + B.
```

Listing 14.5: Prolog

```python
def sum(a, b):
    return a + b
```

Listing 14.6: Python

```erlang
sum(A, B) -> A + B.
```

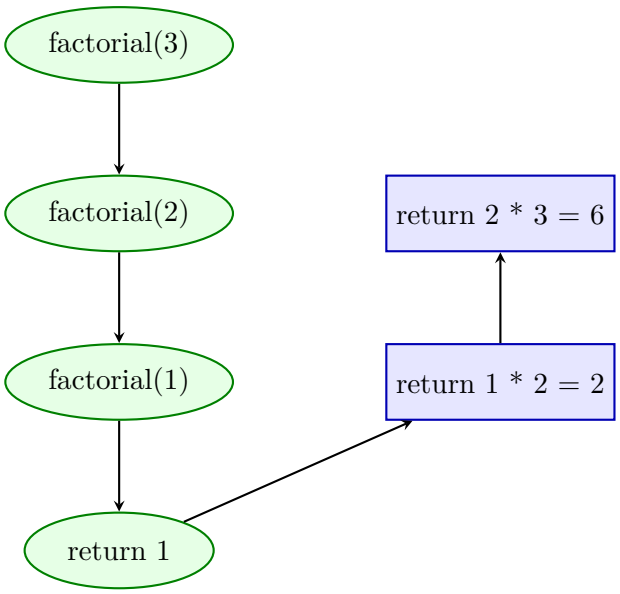
Listing 14.7: Erlang

Note

- **Assembly**: does not support functions directly; labels and registers simulate function behavior.
- **Prolog**: uses logical rules; functions are defined as predicates.
- **Haskell and Erlang**: functions are pure and have no side effects by default.

14.4 Recursive Functions

A recursive function is one that calls itself in order to divide the problem into subproblems.




```c
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Listing 14.8: C

```python
def factorial(n):
    return 1 if n <= 1 else n * factorial(n - 1)
```

Listing 14.9: Python

```haskell
factorial n = if n <= 1 then 1 else n * factorial (n - 1)
```

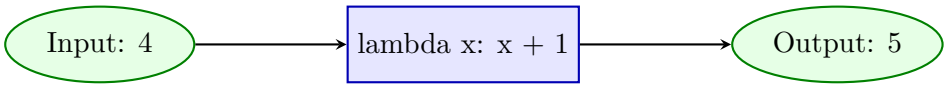
Listing 14.10: Haskell

```prolog
factorial(0, 1).
factorial(N, F) :-
    N > 0, N1 is N - 1,
    factorial(N1, F1),
    F is N * F1.
```

Listing 14.11: Prolog

14.5 Lambda (Anonymous) Functions

Lambda functions are small unnamed functions used primarily for short, functional expressions.




```cpp
auto square = [](int x) { return x * x; };
```

Listing 14.12: C++

```python
square = lambda x: x * x
```

Listing 14.13: Python

```haskell
square = \x -> x * x
```

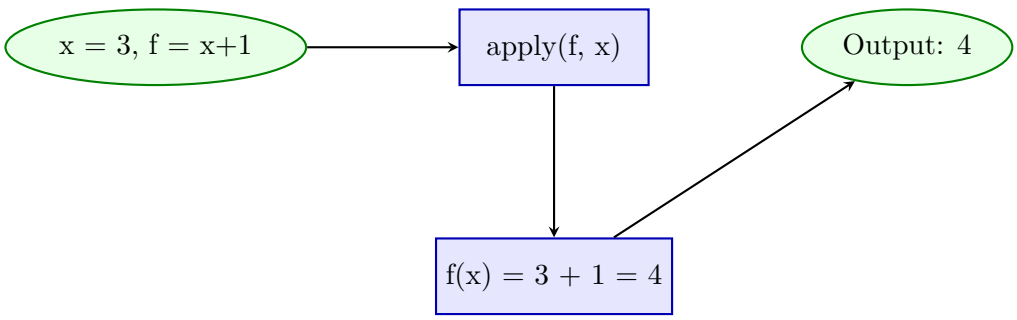
Listing 14.14: Haskell

```erlang
Square = fun(X) -> X * X end.
```

Listing 14.15: Erlang

14.6 Higher-Order Functions

A higher-order function is one that takes another function as input or returns a function.



```cpp
#include <functional>

int apply(std::function<int(int)> f, int x) {
    return f(x);
}
```

Listing 14.16: C++

```python
def apply(f, x):
    return f(x)

print(apply(lambda x: x + 1, 5))
```

Listing 14.17: Python

```haskell
apply f x = f x
```

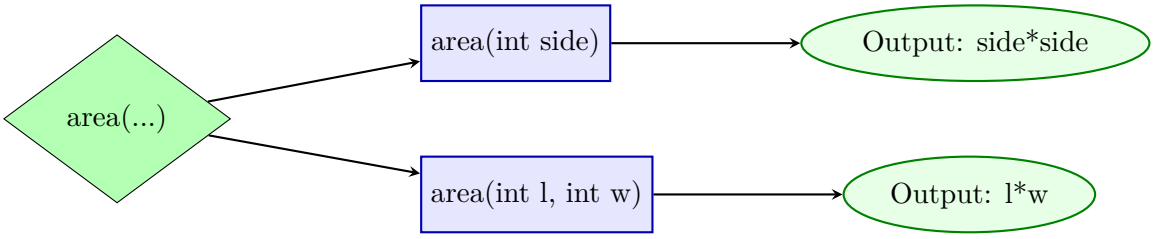
Listing 14.18: Haskell

```erlang
apply(Fun, X) -> Fun(X).
```

Listing 14.19: Erlang

14.7 Function Overloading

Function overloading allows multiple functions with the same name but different parameter types.




```cpp
int area(int side) {
    return side * side;
}

int area(int l, int w) {
    return l * w;
}
```

Listing 14.20: C++

Notes by Language

- **Assembly**: recursion is possible but manually managed via the stack.
- **Prolog**: recursion is common; no lambda or overloading.
- **Python**: supports lambdas, higher-order functions, and decorators to simulate overloading.
- **Haskell**: heavily based on higher-order and lambda functions.
- **C++**: full support for all features, especially with modern C++11+.
- **C**: lacks direct support for lambdas and overloading, but recursion is supported.
- **Erlang**: supports anonymous functions and higher-order functions.

14.8 Pointer Concepts

Pointers are a powerful but complex feature in programming. Beyond basic referencing and dereferencing, they enable advanced operations such as dynamic memory handling, multi-level referencing, and functional indirection.

14.8.1 Pointer Arithmetic

In languages like C/C++, pointers can be incremented or decremented to traverse arrays or memory regions.

- ‘p + 1‘ advances the pointer by the size of the pointed type.
- This is useful in iterating arrays via pointers.

```
int arr[] = {10, 20, 30};
int *p = arr;
printf("%d", *(p + 1)); // Outputs 20
```

14.8.2 Pointers to Pointers

A pointer can hold the address of another pointer, enabling multi-level indirection.

```
int val = 42;
int *p = &val;
int **pp = &p;
printf("%d", **pp); // Outputs 42
```

14.8.3 Dynamic Memory Allocation

In low-level languages, memory can be allocated manually.

- ‘malloc‘, ‘calloc‘, ‘realloc‘ in C

- ‘new‘, ‘delete‘ in C++

```c
int *arr = malloc(5 * sizeof(int));
arr[0] = 1;
free(arr);
```

14.8.4 Function Pointers

Functions can be referenced via pointers, allowing callbacks or flexible APIs.

```c
void greet() {
    printf("Hello\n");
}

void (*func_ptr)() = &greet;
func_ptr(); // Calls greet
```

14.8.5 Smart Pointers (C++)

C++ provides smart pointers to manage dynamic memory safely:

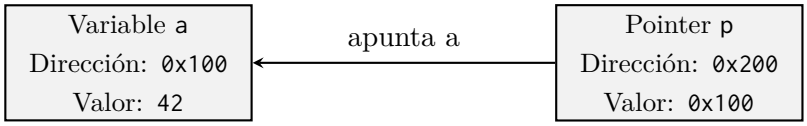
- `std::unique_ptr` for exclusive ownership
- `std::shared_ptr` for shared ownership
- `std::weak_ptr` to break cyclic references

```cpp
#include <memory>
std::unique_ptr<int> p = std::make_unique<int>(5);
```

14.8.6 Language-Specific Notes

- **Python:** All variables are references; pointer-like behavior via object mutability.
- **Prolog:** No direct memory access; term manipulation through logical variables.
- **Erlang:** Immutable variables; uses process messaging instead of memory manipulation.
- **Haskell:** Mutable references via `IORef`, `STRef` in monads.

14.9 Pointer Visualization



14.10 Examples

```
section .data
val dd 42

section .text
global _start

_start:
    mov eax, [val]    ; Load value at address of 'val'
```

Listing 14.21: Assembly

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int value = 42;
    int *p = &value;
    printf("Value: %d\n", *p);

    int *arr = malloc(3 * sizeof(int));
    arr[0] = 1; arr[1] = 2; arr[2] = 3;
    free(arr);
    return 0;
}
```

Listing 14.22: C

```cpp
#include <iostream>
#include <memory>

int main() {
    int value = 42;
    int* ptr = &value;
    std::cout << "Value via pointer: " << *ptr << std::endl;

    std::unique_ptr<int> smartPtr = std::make_unique<int>(99);
    std::cout << "Smart pointer value: " << *smartPtr << std::endl;
}
```

Listing 14.23: C++

```haskell
import Data.IORef

main :: IO ()
main = do
    ref <- newIORef 42
```

```
    writeIORef ref 99
    val <- readIORef ref
    print val
```

Listing 14.24: Haskell

```
% Simulated reference via logic variables
value(42).

get_value(X) :- value(X).
% ?- get_value(X). -> X = 42.
```

Listing 14.25: Prolog

```
# Python uses references for all objects
a = [42]
b = a
b[0] = 99
print(a[0])  # Outputs 99, since both refer to the same object
```

Listing 14.26: Python

```
% Erlang variables are immutable
-module(pointer_sim).
-export([start/0]).

start() ->
    A = 42,
    B = A,
    io:format("Value via B: ~p~n", [B]).
```

Listing 14.27: Erlang

14.11 Best Practices

- Always initialize pointers before use.
- Use ‘NULL‘ or ‘nullptr‘ for clarity.
- Free dynamically allocated memory when no longer needed.
- Avoid pointer arithmetic unless absolutely necessary.
- Prefer smart pointers in C++ to avoid memory leaks.

Chapter 15

Modularity

Modularity is a software design principle that emphasizes dividing a program into distinct components or modules. Each module encapsulates a specific functionality and interacts with others through well-defined interfaces. This approach improves maintainability, readability, testability, and reusability.

15.1 Concept and Importance

- **Reduces complexity**: Large problems are broken down into manageable units.
- **Encourages separation of concerns**: Each module focuses on a single responsibility.
- **Improves testing**: Modules can be tested independently.
- **Enhances team collaboration**: Multiple developers can work on different modules simultaneously.
- **Promotes reusability**: Well-designed modules can be reused across projects.

15.2 Key Concepts

15.2.1 Abstraction

Modules expose only necessary information via interfaces, hiding internal details.

15.2.2 Cohesion

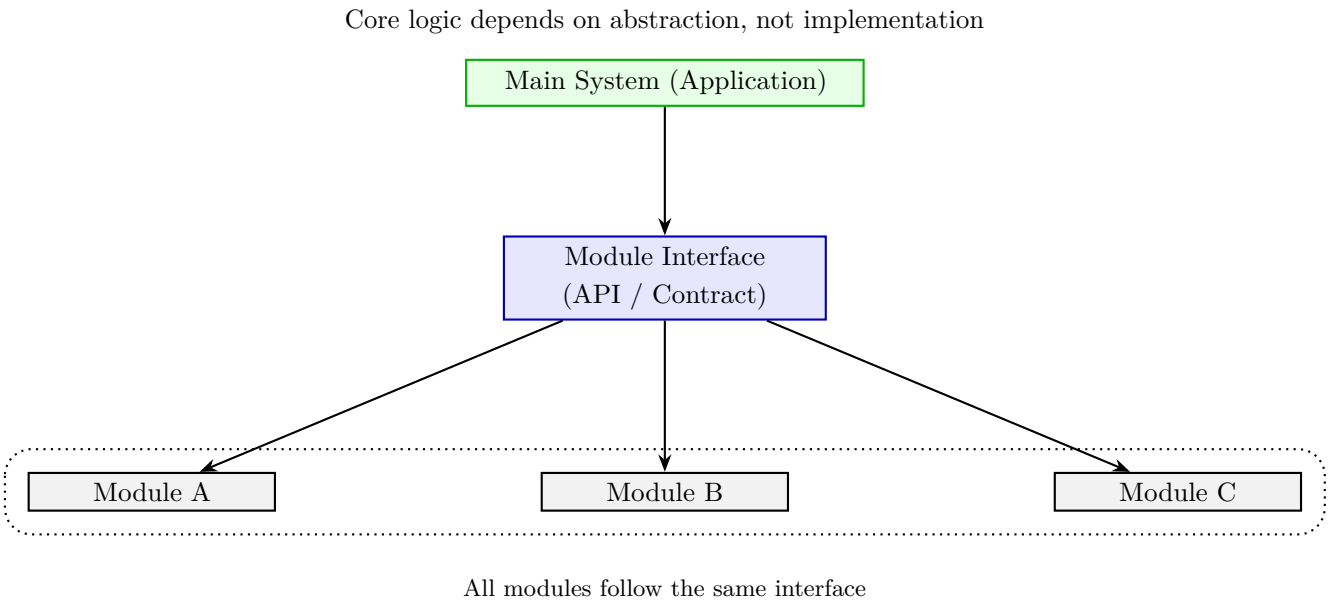
Each module should be highly cohesive — its components should work together to fulfill a single, well-defined purpose.

15.2.3 Coupling

Coupling refers to the degree of dependency between modules. Aim for **low coupling** to reduce the impact of changes.

15.2.4 Dependency Injection

Dependency injection decouples modules by providing their dependencies from the outside, improving modularity and testability.



15.3 Examples in Multiple Paradigms

15.3.1 C (Modular Programming with Headers)

```c
// module.h
int add(int, int);

// module.c
int add(int a, int b) {
    return a + b;
}

// main.c
#include <stdio.h>
#include "module.h"

int main() {
    printf("%d\n", add(3, 4));
    return 0;
}
```

15.3.2 C++ (Classes and Namespaces)

```cpp
#include <iostream>
namespace Math {
    int add(int a, int b) {
        return a + b;
    }
}

int main() {
    std::cout << Math::add(2, 3);
}
```

15.3.3 Python (Modules and Imports)

```python
# math_module.py
def add(a, b):
    return a + b

# main.py
import math_module
print(math_module.add(5, 7))
```

15.3.4 Haskell (Modules and Pure Functions)

```haskell
-- MathModule.hs
module MathModule (add) where
add :: Int -> Int -> Int
add a b = a + b

-- Main.hs
import MathModule
main = print (add 2 3)
```

15.3.5 Prolog (Modular Predicate Definitions)

```prolog
% math.pl
:- module(math, [add/3]).
add(A, B, Result) :- Result is A + B.

% main.pl
:- use_module(math).
:- math:add(2, 3, X), writeln(X).
```

15.3.6 Erlang (Modules and Processes)

```
% math.erl
-module(math).
-export([add/2]).

add(A, B) -> A + B.

% shell
c(math).
math:add(3, 4).
```

15.4 Advanced Modular Constructs

- **Interfaces/Abstract Types**: Provide contracts that multiple modules can implement (e.g., Java interfaces, Haskell typeclasses).
- **Plugins/Dependency Injection**: External modules can be swapped at runtime.
- **Microservices**: System-wide modularity using process-level separation and communication.

15.5 Best Practices

- Keep module APIs minimal and stable.
- Follow the Single Responsibility Principle.
- Favor composition over inheritance in OO design.
- Test modules independently using mocks or stubs.
- Use namespaces/packages to prevent naming collisions.

Chapter 16

Code Modularity and Best Programming Practices

In this chapter, we explore foundational practices for writing clean, modular, and maintainable code. Good programming practices ensure that code is not only correct but also easy to understand, test, modify, and scale over time.

16.1 Clean Code Principles

Writing clean code involves adopting habits and styles that improve software quality. Clean code should be:

- **Readable:** Easy to understand by humans.
- **Maintainable:** Easy to modify and extend.
- **Reusable:** Usable across multiple projects or modules.
- **Testable:** Easy to verify correctness through automated or manual testing.
- **Consistent:** Adheres to a coherent style and naming convention.

16.1.1 Key Guidelines

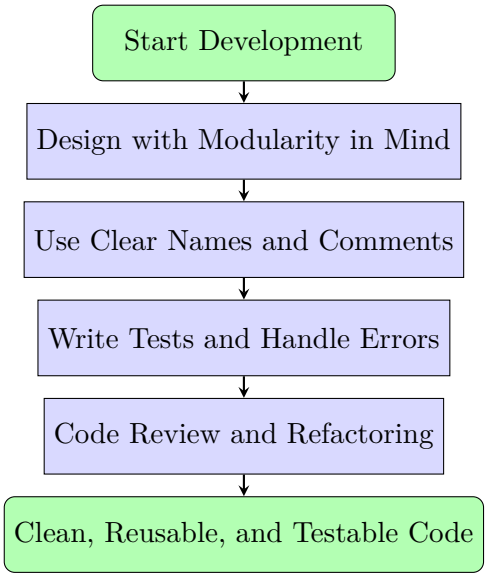
- Write code as if the next person to read it is a junior developer.
- Prefer clarity over cleverness.
- Avoid long functions and deeply nested logic.

16.2 Best Practices Checklist

- Follow consistent naming and formatting conventions.
- Separate logic into well-defined functions or modules.
- Avoid magic numbers and hardcoded constants.

- Apply the **DRY** (Don't Repeat Yourself) principle.
- Use source control (e.g., Git) for collaboration and history tracking.
- Include unit, integration, and edge case tests.
- Add docstrings and comments to clarify intention, not obvious behavior.
- Document interfaces and side effects of functions.
- Handle errors and exceptions gracefully.

16.3 Flowchart: Clean Code Workflow



16.4 Principles of Modular Design

Modular design promotes separation of concerns by dividing a program into independent components. Each module should:

- Have a single, well-defined responsibility.
- Encapsulate its data and logic.
- Expose a minimal and clear interface.
- Be loosely coupled and highly cohesive.

16.4.1 SOLID Principles

- **S** - Single Responsibility Principle
- **O** - Open/Closed Principle
- **L** - Liskov Substitution Principle

- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

16.5 Examples in Various Paradigms

16.5.1 C++ (Encapsulation and Classes)

```cpp
class Calculator {
public:
    int add(int a, int b);
};

int Calculator::add(int a, int b) {
    return a + b;
}

int main() {
    Calculator calc;
    std::cout << calc.add(1, 2) << std::endl;
}
```

Listing 16.1: C++ Example: Modular Class

16.5.2 Python (Type Hints and Docstrings)

```python
def add(a: int, b: int) -> int:
    """
    Adds two integers and returns the result.
    """
    return a + b
```

Listing 16.2: Python Example with Documentation

16.5.3 Haskell (Function Decomposition)

```haskell
main = print (sumOfSquares [1,2,3])

sumOfSquares :: [Int] -> Int
sumOfSquares xs = sum (map (^2) xs)
```

Listing 16.3: Haskell Example with Functional Style

16.5.4 Prolog (Separation of Predicates)

```
% Adds two numbers
add(X, Y, Result) :- Result is X + Y.
```

Listing 16.4: Prolog Example with Predicate Abstraction

16.6 Before and After: Refactoring Example in C

Before Refactoring:

```
int main() {
    int x = 5, y = 10;
    int result = x + y;
    printf("%d\n", result);
}
```

Listing 16.5: C Code - Before Refactor

After Refactoring:

```
int add(int a, int b) {
    return a + b;
}

int main() {
    printf("%d\n", add(5, 10));
}
```

Listing 16.6: C Code - After Refactor

16.7 Code Reviews and Pair Programming

Code should be regularly reviewed by peers to catch bugs, improve design, and ensure adherence to standards. Pair programming helps with knowledge sharing and early defect detection.

Chapter 17

Introduction to Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data, or *objects*, rather than functions and logic. It enables developers to model real-world entities by combining state and behavior into encapsulated units.

17.1 Basic Concepts

- **Class:** A template or blueprint that defines the structure and behavior (data and methods) of objects.
- **Object:** A concrete instance of a class containing real values for the defined attributes and methods.
- **Encapsulation:** Hiding the internal state of an object and requiring interaction through methods.
- **Abstraction:** Simplifying complex reality by exposing only the relevant parts of an object.
- **Inheritance:** A mechanism for a class to acquire properties and behavior of another class.
- **Polymorphism:** The ability to use a single interface to represent different types or behaviors.

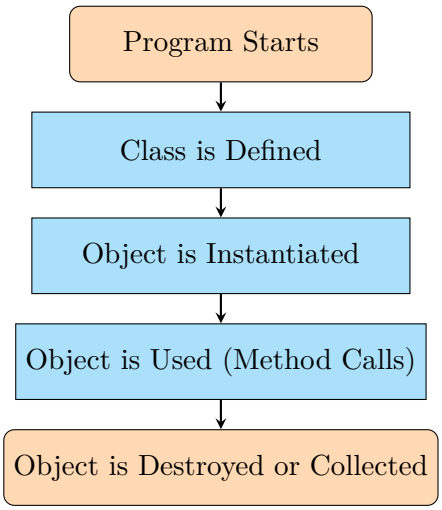
17.1.1 Design Principles

OOP also supports important design principles:

- **Responsibility-Driven Design:** Each class should represent a single responsibility.
- **Message Passing:** Objects communicate by sending and receiving messages (method calls).

- **Composition over Inheritance:** Prefer assembling behavior from smaller parts rather than deep inheritance trees.

17.2 Flowchart: Object Lifecycle



17.3 Example in C++ (Statically Typed OOP)

```cpp
#include <iostream>
using namespace std;

class Person {
public:
    string name;

    Person(string n) : name(n) {}

    void greet() {
        cout << "Hello, " << name << "!" << endl;
    }
};

int main() {
    Person p("Alice");
    p.greet();
    return 0;
}
```

Listing 17.1: OOP Example in C++

17.4 Example in Python (Dynamically Typed OOP)

```python
class Person:
    def __init__(self, name):
        self.name = name

    def greet(self):
        print(f"Hello, {self.name}!")

p = Person("Alice")
p.greet()
```

Listing 17.2: OOP Example in Python

17.5 Comparison with Procedural Programming

- **Procedural:** Focuses on procedures or routines operating on data.
- **OOP:** Organizes code around objects which combine state and behavior.
- **Modularity:** OOP offers stronger modularity via encapsulation and abstraction.
- **Scalability:** OOP scales better in large codebases due to inheritance and polymorphism.

17.6 Advantages and Disadvantages of OOP

17.6.1 Advantages

- Encourages modular and reusable code.
- Makes maintenance and debugging easier.
- Models real-world entities more naturally.
- Promotes team development with clearer interface contracts.

17.6.2 Disadvantages

- Adds complexity for small, simple scripts.
- May introduce performance overhead.
- Overuse of inheritance can lead to fragile designs.

17.7 When to Use OOP

Object-Oriented Programming is ideal when:

- Modeling complex systems with many interacting components.
- Creating large software with reusable and maintainable architecture.

- Building GUI applications, simulations, games, or frameworks.
- Working in collaborative teams that benefit from abstraction and modularity.

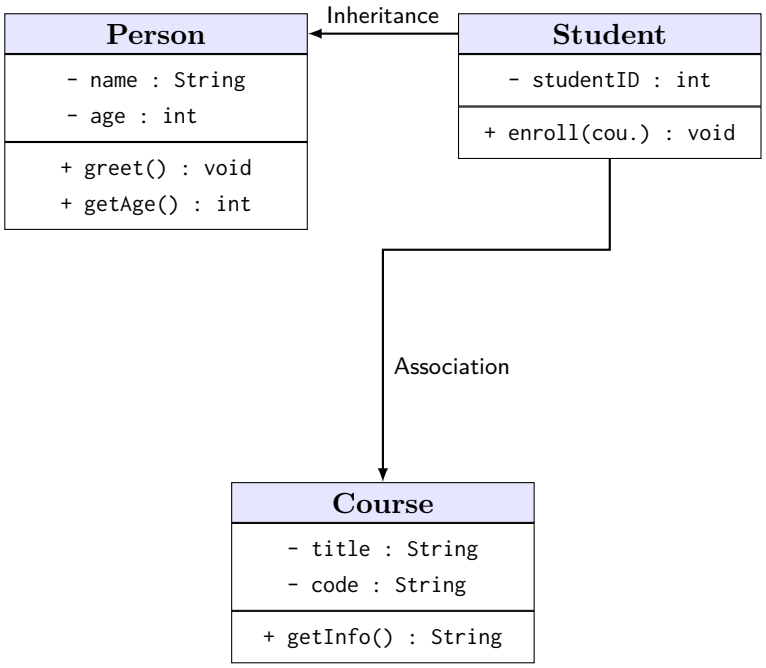
17.8 UML Class Diagrams

The Unified Modeling Language (UML) is a standardized modeling language used to visualize the design of software systems. One of the most common UML diagrams is the **class diagram**, which shows the structure of a system by illustrating classes, their attributes, methods, and relationships.

17.8.1 Key Elements of UML Class Diagrams

- **Classes:** Represented as boxes divided into three parts (name, attributes, methods).
- **Attributes:** Properties or data held by the class.
- **Methods:** Operations or functions that the class can perform.
- **Relationships:**
  - **Inheritance (Generalization):** A subclass inherits from a superclass.
  - **Association:** A "has-a" relationship.
  - **Aggregation/Composition:** A special form of association (whole-part).

17.8.2 UML Class Diagram Example



17.8.3 Explanation of the Diagram

- The `Person` class contains private attributes `name` and `age`, and public methods like `greet()`.
- `Student` inherits from `Person`, and adds a new attribute `studentID` and a method `enroll()`.
- There is an association between `Student` and `Course`, implying a student can enroll in courses.

17.8.4 Usage of UML in Development

UML diagrams are especially useful during the design phase of a project to:

- Clarify object relationships and responsibilities.
- Facilitate communication among developers, analysts, and stakeholders.
- Serve as documentation for long-term maintenance and onboarding.

Part II

Advanced Algorithmic Strategies

Chapter 18

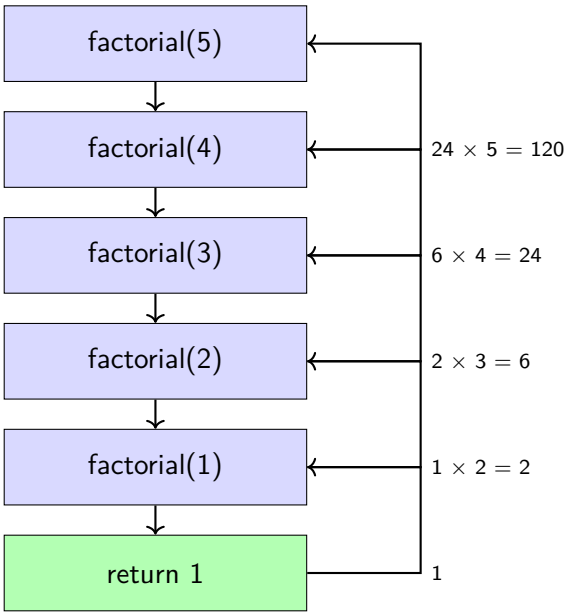
Recursive Thinking and Base Case Design

Recursion is a fundamental concept in computer science and algorithmic design. It involves solving a problem by breaking it down into smaller instances of the same problem. A recursive function calls itself with a modified input, and this process continues until a base case is reached.

18.1 Structure of a Recursive Algorithm

A recursive algorithm typically consists of:

- A **base case**, which terminates the recursion.
- A **recursive step**, where the function calls itself.



18.2 Examples in Multiple Languages

```
factorial:
    cmp rdi, 1
    jbe .base
    push rdi
    dec rdi
    call factorial
    pop rbx
    imul rax, rbx
    ret
.base:
    mov rax, 1
    ret
```

Listing 18.1: Recursive factorial in NASM

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

Listing 18.2: Recursive factorial in C

```
class Math {
public:
    static int factorial(int n) {
        if (n == 0) return 1;
        return n * factorial(n - 1);
    }
};
```

Listing 18.3: Recursive factorial in C++ class

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Listing 18.4: Recursive factorial in Haskell

```
factorial(0, 1).
factorial(N, Result) :-
    N > 0,
    N1 is N - 1,
    factorial(N1, R1),
    Result is N * R1.
```

Listing 18.5: Recursive factorial in Prolog

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Listing 18.6: Recursive factorial in Python

```erlang
factorial(0) -> 1;
factorial(N) when N > 0 -> N * factorial(N - 1).
```

Listing 18.7: Recursive factorial in Erlang

18.3 Common Pitfalls

- **Missing base case:** May lead to infinite recursion and stack overflow.
- **Incorrect base case:** May produce wrong results.
- **Stack depth:** Recursion is limited by the call stack size.

Chapter 19

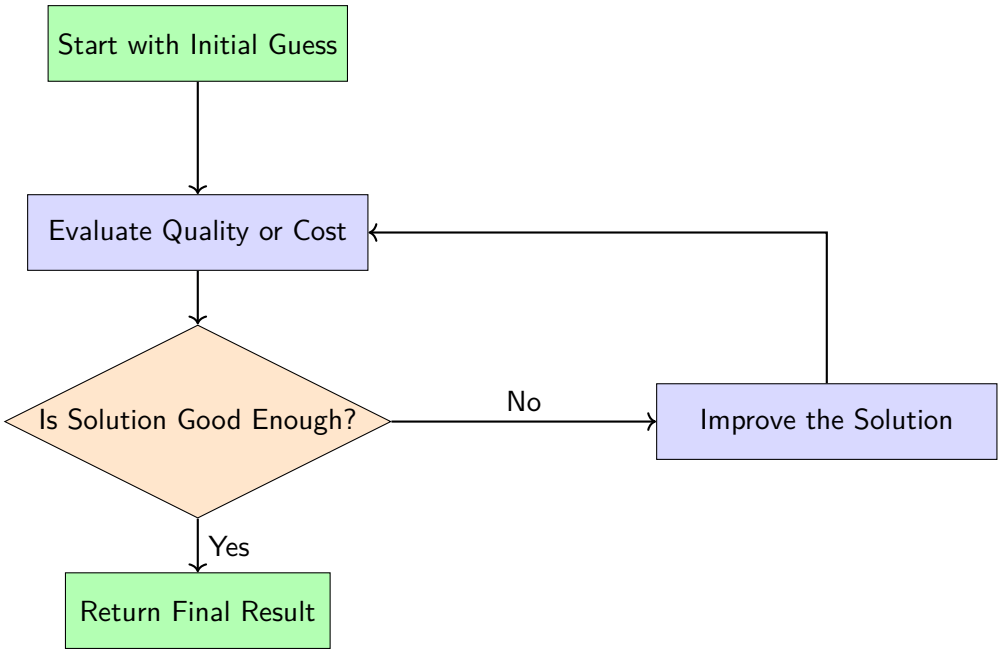
Iterative Improvement Techniques

Iterative improvement is a method used to refine the performance or correctness of an algorithm gradually. Instead of finding an optimal solution in a single step, the algorithm begins with an initial solution and improves it through successive iterations based on certain criteria.

19.1 General Structure

An iterative improvement algorithm typically follows this structure:

1. Initialize a feasible (possibly naive) solution.
2. Evaluate the solution according to some metric.
3. While not optimal:
    - Modify the solution.
    - Re-evaluate.



19.2 Examples in Multiple Languages

```
; Using Newton-Raphson method (conceptual implementation)
; This is simplified pseudo-NASM due to complexity of full
    implementation
```

Listing 19.1: Iterative square root approximation in NASM (conceptual)

```c
double sqrt_approx(double x) {
    double guess = x / 2.0;
    for (int i = 0; i < 10; i++) {
        guess = (guess + x / guess) / 2.0;
    }
    return guess;
}
```

Listing 19.2: Square root approximation using Newton-Raphson in C

```cpp
class Approximator {
public:
    static double sqrt(double x, int iterations = 10) {
        double g = x / 2.0;
        for (int i = 0; i < iterations; ++i)
            g = (g + x / g) / 2.0;
        return g;
    }
};
```

Listing 19.3: Iterative improvement class in C++

```haskell
sqrtApprox x = iterate (\g -> (g + x / g) / 2) (x / 2)
```

Listing 19.4: Iterative improvement using lazy evaluation

```
sqrt_iter(X, G, _, G) :- abs(X - G * G) < 0.0001.
sqrt_iter(X, G, I, R) :-
    I > 0,
    G1 is (G + X / G) / 2,
    I1 is I - 1,
    sqrt_iter(X, G1, I1, R).
```

Listing 19.5: Iterative loop in Prolog with recursion

```python
def sqrt_approx(x, iterations=10):
    guess = x / 2
    for _ in range(iterations):
        guess = (guess + x / guess) / 2
    return guess
```

Listing 19.6: Newton-Raphson square root in Python

```
sqrt_approx(X) -> sqrt_approx(X, X / 2, 10).
sqrt_approx(_, Guess, 0) -> Guess;
sqrt_approx(X, Guess, N) ->
    NewGuess = (Guess + X / Guess) / 2,
    sqrt_approx(X, NewGuess, N - 1).
```

Listing 19.7: Iterative approximation using recursion

19.3 Applications

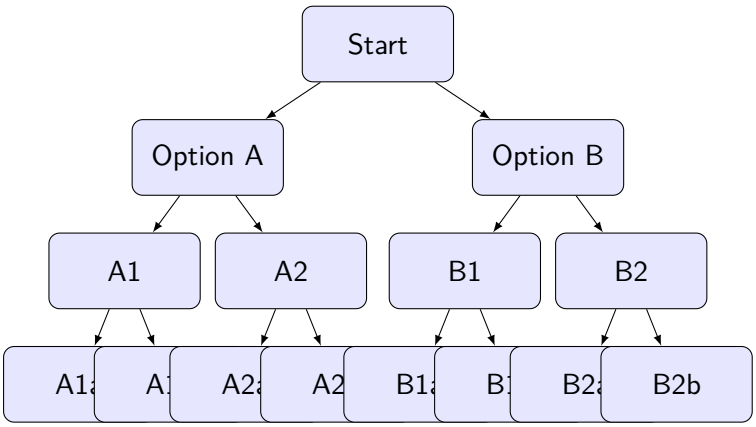
- Square root and root-finding (Newton-Raphson).
- Optimization and tuning (e.g., hyperparameter search).
- Game AI (e.g., Minimax with iterative deepening).

Chapter 20

Brute Force Algorithms

20.1 Introduction

Brute force is a fundamental algorithmic strategy that involves systematically enumerating all possible solutions to a problem and checking which ones meet the required conditions. Although often inefficient, brute force algorithms are simple to implement and can be effective for problems with small input sizes.



20.2 Examples in Multiple Languages

20.2.1 Problem: Find all permutations of a string

```
; Permutations in assembly are non-trivial. Not implemented in full.
; Would require stack manipulation, recursion, and swap logic.
```

Listing 20.1: Conceptual brute force in Assembly

```
void swap(char *x, char *y) {
    char temp = *x; *x = *y; *y = temp;
}

void permute(char *str, int l, int r) {
```

```
    if (l == r) printf("%s\n", str);
    else {
        for (int i = l; i <= r; i++) {
            swap(&str[l], &str[i]);
            permute(str, l + 1, r);
            swap(&str[l], &str[i]);
        }
    }
}
```

Listing 20.2: Brute-force permutation generator in C

```
#include <iostream>
#include <algorithm>
using namespace std;

void brute_permutations(string s) {
    sort(s.begin(), s.end());
    do {
        cout << s << endl;
    } while (next_permutation(s.begin(), s.end()));
}
```

Listing 20.3: Brute-force with std::next permutation

```
import Data.List (permutations)

main = mapM_ putStrLn (permutations "abc")
```

Listing 20.4: Using built-in permutations

```
permute([], []).
permute(L, [H|T]) :-
    select(H, L, R),
    permute(R, T).
```

Listing 20.5: Generate permutations

```
from itertools import permutations

for p in permutations("abc"):
    print("".join(p))
```

Listing 20.6: Using itertools.permutations

```
permute([]) -> [[]];
permute(L) ->
    [[H|T] || H <- L, T <- permute(lists:delete(H, L))].
```

Listing 20.7: Generate permutations

20.3 Applications

- Solving puzzles (e.g., Sudoku, N-Queens).
- Password cracking (with caution in ethical/legal settings).
- Generating all combinations/permutations for exhaustive analysis.

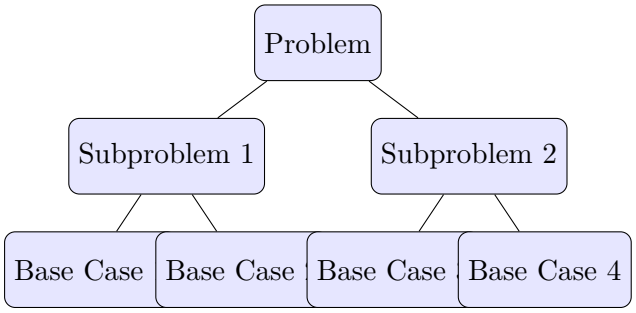
20.4 Limitations

- Poor scalability with large inputs (factorial growth).
- Often replaced with more efficient strategies when possible.

Chapter 21

Divide and Conquer

Divide and Conquer is a classical algorithmic paradigm that solves a problem by dividing it into smaller subproblems, solving each recursively, and then combining their solutions. It is particularly effective for problems that can be broken down naturally into independent subcomponents.



21.1 Examples in Multiple Languages

21.1.1 Problem: Merge Sort Implementation

```
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    int L[n1], R[n2];

    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];

    i = j = 0; k = l;
    while (i < n1 && j < n2)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
```

```c
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Listing 21.1: Merge Sort in C

```cpp
void mergeSort(vector<int>& arr) {
    if (arr.size() <= 1) return;

    int mid = arr.size() / 2;
    vector<int> left(arr.begin(), arr.begin() + mid);
    vector<int> right(arr.begin() + mid, arr.end());

    mergeSort(left);
    mergeSort(right);

    merge(arr, left, right);
}
```

Listing 21.2: Merge Sort in C++

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                arr[k] = left[i]
                i += 1
            else:
                arr[k] = right[j]
                j += 1
            k += 1

        arr[k:] = left[i:] + right[j:]
```

Listing 21.3: Merge Sort in Python

```haskell
mergeSort :: (Ord a) => [a] -> [a]
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs  = merge (mergeSort left) (mergeSort right)
  where (left, right) = splitAt (length xs `div` 2) xs

merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
    | x <= y    = x : merge xs (y:ys)
    | otherwise = y : merge (x:xs) ys
```

Listing 21.4: Merge Sort in Haskell

```prolog
merge_sort([], []).
merge_sort([X], [X]).
merge_sort(List, Sorted) :-
    split(List, L1, L2),
    merge_sort(L1, S1),
    merge_sort(L2, S2),
    merge(S1, S2, Sorted).
```

Listing 21.5: Merge Sort in Prolog

```erlang
merge_sort([]) -> [];
merge_sort([X]) -> [X];
merge_sort(List) ->
    {L1, L2} = lists:split(length(List) div 2, List),
    merge(merge_sort(L1), merge_sort(L2)).

merge([], Y) -> Y;
merge(X, []) -> X;
merge([H1|T1], [H2|T2]) when H1 < H2 -> [H1 | merge(T1, [H2|T2])];
merge([H1|T1], [H2|T2])              -> [H2 | merge([H1|T1], T2)].
```

Listing 21.6: Merge Sort in Erlang

21.2 Applications

- Sorting algorithms (Merge Sort, Quick Sort).
- Matrix multiplication (Strassen's algorithm).
- Fast Fourier Transform (FFT).

- Closest pair of points in computational geometry.

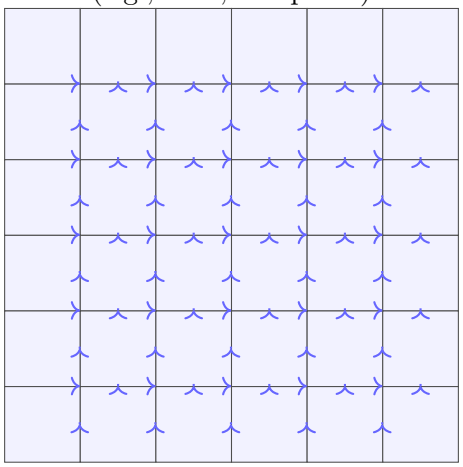
Chapter 22

Dynamic Programming

Dynamic Programming (DP) is an algorithmic technique used to solve problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations. It is especially effective for optimization problems where the solution can be constructed from solutions to overlapping subproblems.

DP Table

(e.g., LCS, Knapsack)



Key Ideas:

- Store solutions to subproblems
- Build solution from smaller results
- Avoid recomputation

22.1 Examples in Multiple Languages

Problem: Longest Common Subsequence (LCS)

```
int lcs(char* X, char* Y, int m, int n) {
    int L[m+1][n+1];
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i-1] == Y[j-1])
                L[i][j] = 1 + L[i-1][j-1];
            else
```

```
                L[i][j] = (L[i-1][j] > L[i][j-1]) ? L[i-1][j] : L[i][j
    -1];
            }
        }
        return L[m][n];
}
```

Listing 22.1: LCS in C

```
int lcs(string X, string Y) {
    int m = X.size(), n = Y.size();
    vector<vector<int>> L(m+1, vector<int>(n+1, 0));
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
            if (X[i-1] == Y[j-1])
                L[i][j] = L[i-1][j-1] + 1;
            else
                L[i][j] = max(L[i-1][j], L[i][j-1]);
    return L[m][n];
}
```

Listing 22.2: LCS in C++

```
def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(m):
        for j in range(n):
            if X[i] == Y[j]:
                dp[i+1][j+1] = dp[i][j] + 1
            else:
                dp[i+1][j+1] = max(dp[i][j+1], dp[i+1][j])

    return dp[m][n]
```

Listing 22.3: LCS in Python

```
lcs :: Eq a => [a] -> [a] -> Int
lcs [] _ = 0
lcs _ [] = 0
lcs (x:xs) (y:ys)
    | x == y    = 1 + lcs xs ys
    | otherwise = max (lcs xs (y:ys)) (lcs (x:xs) ys)
```

Listing 22.4: LCS in Haskell

```
lcs([], _) -> 0;
lcs(_, []) -> 0;
```

```
lcs([H1|T1], [H1|T2]) ->
    1 + lcs(T1, T2);
lcs([_|T1], [_|T2]) ->
    max(lcs([H1|T1], T2), lcs(T1, [H2|T2])).
```

Listing 22.5: LCS in Erlang

22.2 Applications

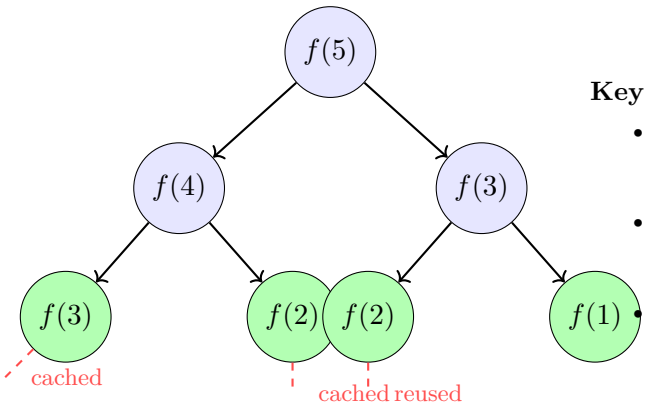
- Longest Common Subsequence
- 0/1 Knapsack Problem
- Matrix Chain Multiplication
- Fibonacci Sequence (optimized)
- Edit Distance (Levenshtein)

Chapter 23

Memoization

Memoization is an optimization technique used primarily to speed up recursive algorithms by storing previously computed results and reusing them when needed. It prevents the repeated evaluation of the same subproblems, improving efficiency especially in exponential-time algorithms.

Recursive Call Tree with Memoization



Key Points:

- Repeated calls to $f(n)$ are avoided.
- Cached values (green) are reused.
- Saves exponential time.

23.1 Example: Fibonacci Numbers

The naive recursive Fibonacci algorithm recalculates the same subproblems repeatedly. Memoization avoids this inefficiency.

```
fib_cache = {}

def fib(n):
    if n in fib_cache:
        return fib_cache[n]
    if n <= 1:
        value = n
    else:
        value = fib(n-1) + fib(n-2)
    fib_cache[n] = value
```

```
    return value
```

Listing 23.1: Memoized Fibonacci in Python

```cpp
#include <unordered_map>
std::unordered_map<int, int> fib_cache;

int fib(int n) {
    if (fib_cache.count(n)) return fib_cache[n];
    if (n <= 1) return n;
    return fib_cache[n] = fib(n-1) + fib(n-2);
}
```

Listing 23.2: Memoized Fibonacci in C++

```haskell
import Data.Function (fix)
import qualified Data.Map as Map

fib = fix (memo Map.empty)
  where
    memo cache f n = Map.findWithDefault v n cache'
      where
        v = if n <= 1 then n else f (n-1) + f (n-2)
        cache' = Map.insert n v cache
```

Listing 23.3: Memoized Fibonacci in Haskell

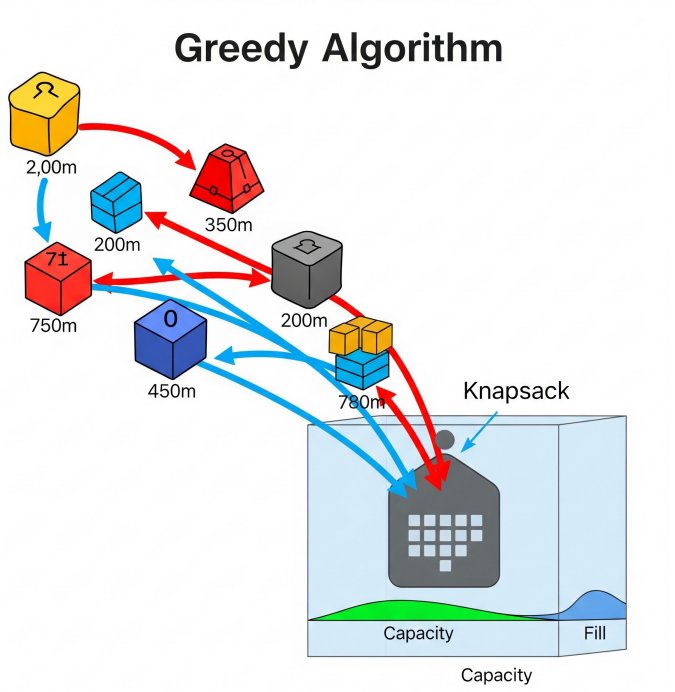
23.2 Applications

- Recursive algorithms with overlapping subproblems
- Dynamic programming with top-down approach
- Optimization of exponential algorithms like:
  - Fibonacci sequence
  - Knapsack
  - Edit Distance

Chapter 24

Greedy Algorithms

Greedy algorithms build up a solution piece by piece, always choosing the option that seems best at the moment. They do not reconsider their choices, which makes them efficient but not always optimal for every problem.



24.1 Example: Activity Selection Problem

Given a set of activities with start and end times, select the maximum number of activities that don't overlap. The greedy choice is to always pick the next activity that ends earliest.

```
#include <stdio.h>
#include <stdlib.h>

int compare(const void *a, const void *b) {
    return ((int*)a)[1] - ((int*)b)[1];
```

```c
}

void activitySelection(int activities[][2], int n) {
    qsort(activities, n, sizeof(activities[0]), compare);
    int lastEnd = activities[0][1];
    printf("Selected: [0] ");
    for (int i = 1; i < n; i++) {
        if (activities[i][0] >= lastEnd) {
            printf("[%d] ", i);
            lastEnd = activities[i][1];
        }
    }
}
```

Listing 24.1: Greedy Activity Selection in C

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

bool comp(pair<int,int> a, pair<int,int> b) {
    return a.second < b.second;
}

void activitySelection(vector<pair<int,int>> &activities) {
    sort(activities.begin(), activities.end(), comp);
    int lastEnd = activities[0].second;
    cout << "Selected: [0] ";
    for (size_t i = 1; i < activities.size(); i++) {
        if (activities[i].first >= lastEnd) {
            cout << "[" << i << "] ";
            lastEnd = activities[i].second;
        }
    }
}
```

Listing 24.2: Greedy Activity Selection in C++

```python
def activity_selection(activities):
    activities.sort(key=lambda x: x[1])
    selected = [activities[0]]
    last_end = activities[0][1]
    for start, end in activities[1:]:
        if start >= last_end:
            selected.append((start, end))
            last_end = end
    return selected
```

Listing 24.3: Greedy Activity Selection in Python

```
select_activities([], [], _).
select_activities([[S,E]|Rest], [[S,E]|Selected], LastEnd) :-
    S >= LastEnd,
    select_activities(Rest, Selected, E).
select_activities([_|Rest], Selected, LastEnd) :-
    select_activities(Rest, Selected, LastEnd).
```

Listing 24.4: Greedy Selection in Prolog

24.2 Use Cases and Limitations

Greedy algorithms work well for:

- Activity selection
- Huffman coding
- Fractional knapsack
- Minimum spanning tree (e.g., Prim's, Kruskal's)

They do not always give optimal results for problems like:

- 0/1 Knapsack
- Traveling Salesman Problem
- Certain dynamic programming problems

Chapter 25

Complexity - Big-o notation

25.1 Mathematical Foundations

25.1.1 Asymptotic Notation: Rigorous Definitions

Definition 25.1 (Big O Notation)**.** Let $f, g : \mathbb{N} \to \mathbb{R}^+$ be functions. We say that $f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that:

$$0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0 \tag{25.1}$$

Definition 25.2 (Omega Notation)**.** $f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that:

$$0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq n_0 \tag{25.2}$$

Definition 25.3 (Theta Notation)**.** $f(n) = \Theta(g(n))$ 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:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0 \tag{25.3}$$

25.1.2 Limit-Based Characterization

Theorem 25.1 (Limit Test for Big O)**.** *If* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = L$ *where* L *is a non-negative real number, then:*

- *If* $L = 0$*, then* $f(n) = o(g(n))$ *(little-o)*
- *If* $0 < L < \infty$*, then* $f(n) = \Theta(g(n))$
- *If* $L = \infty$*, then* $f(n) = \omega(g(n))$ *(little-omega)*

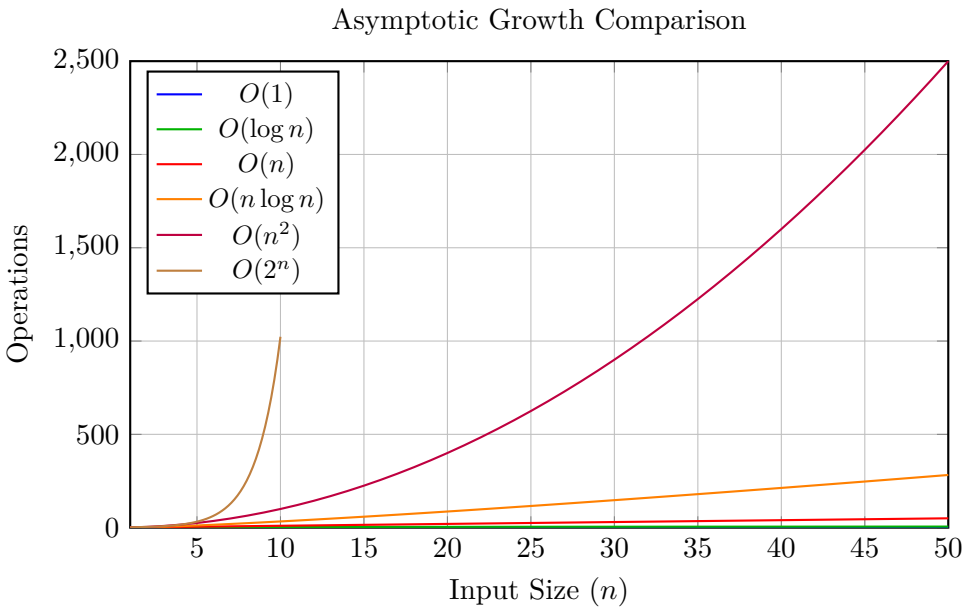


Figure 25.1: Growth rates of common complexity functions

25.2 Detailed Complexity Analysis of Algorithms

25.2.1 Binary Search: Mathematical Proof of O(log n) Complexity

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1
```

Listing 25.1: Binary Search Implementation

Theorem 25.2 (Binary Search Complexity)**.** *The time complexity of binary search on a sorted array of size n is $\Theta(\log n)$.*

Proof. Let $T(n)$ be the number of operations required to search in an array of size n.
In each iteration, we:

1. Calculate the middle index: $O(1)$
2. Compare the middle element with target: $O(1)$

3. Reduce the search space by half: $n \to \lceil n/2 \rceil$

The recurrence relation is:

$$T(n) = T(\lceil n/2 \rceil) + c \tag{25.4}$$

where c is a constant representing the work done in each iteration.
For simplicity, assume $n = 2^k$ for some integer k. Then:

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + c && (25.5)\\ &= T(2^{k-2}) + 2c && (25.6)\\ &= \dots && (25.7)\\ &= T(2^0) + kc && (25.8)\\ &= T(1) + kc && (25.9) \end{aligned}$$

Since $k = \log_2 n$, we have:

$$T(n) = T(1) + c \log_2 n = O(\log n) \tag{25.10}$$

For the lower bound, in the worst case, we must eliminate all but one element, which requires at least $\lfloor \log_2 n \rfloor$ iterations. Therefore, $T(n) = \Omega(\log n)$.
Combining both bounds: $T(n) = \Theta(\log n)$. □

25.2.2 Merge Sort: Mathematical Analysis

```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

```

```
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Listing 25.2: Merge Sort Implementation

Theorem 25.3 (Merge Sort Complexity)**.** *The time complexity of merge sort is* $\Theta(n \log n)$ *and space complexity is* $\Theta(n)$*.*

Proof. Let $T(n)$ be the time required to sort an array of size n.
The merge sort algorithm:

1. Divides the array into two halves: $O(1)$
2. Recursively sorts each half: $2T(n/2)$
3. Merges the sorted halves: $\Theta(n)$

The recurrence relation is:

$$T(n) = 2T(n/2) + \Theta(n) \tag{25.11}$$

Using the Master Theorem with $a = 2$, $b = 2$, and $f(n) = \Theta(n)$:

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n \tag{25.12}$$

$$f(n) = \Theta(n) = \Theta(n^{\log_b a}) \tag{25.13}$$

Since $f(n) = \Theta(n^{\log_b a})$, we apply Case 2 of the Master Theorem:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n) \tag{25.14}$$

For space complexity, the recursion depth is $\log n$, and at each level we store $O(n)$ elements in temporary arrays during merging. Therefore, space complexity is $\Theta(n)$. □

25.2.3 Master Theorem: A Powerful Tool for Recurrence Relations

Theorem 25.4 (Master Theorem)**.** *Let* $a \geq 1$ *and* $b > 1$ *be constants, and let* $f(n)$ *be a function. Consider the recurrence:*

$$T(n) = aT(n/b) + f(n) \tag{25.15}$$

Then $T(n)$ *has the following asymptotic bounds:*

1. *If* $f(n) = O(n^{\log_b a - \epsilon})$ *for some constant* $\epsilon > 0$*, then* $T(n) = \Theta(n^{\log_b a})$
2. *If* $f(n) = \Theta(n^{\log_b a})$*, then* $T(n) = \Theta(n^{\log_b a} \log n)$
3. *If* $f(n) = \Omega(n^{\log_b a + \epsilon})$ *for some constant* $\epsilon > 0$*, and if* $af(n/b) \leq cf(n)$ *for some constant* $c < 1$ *and sufficiently large* n*, then* $T(n) = \Theta(f(n))$

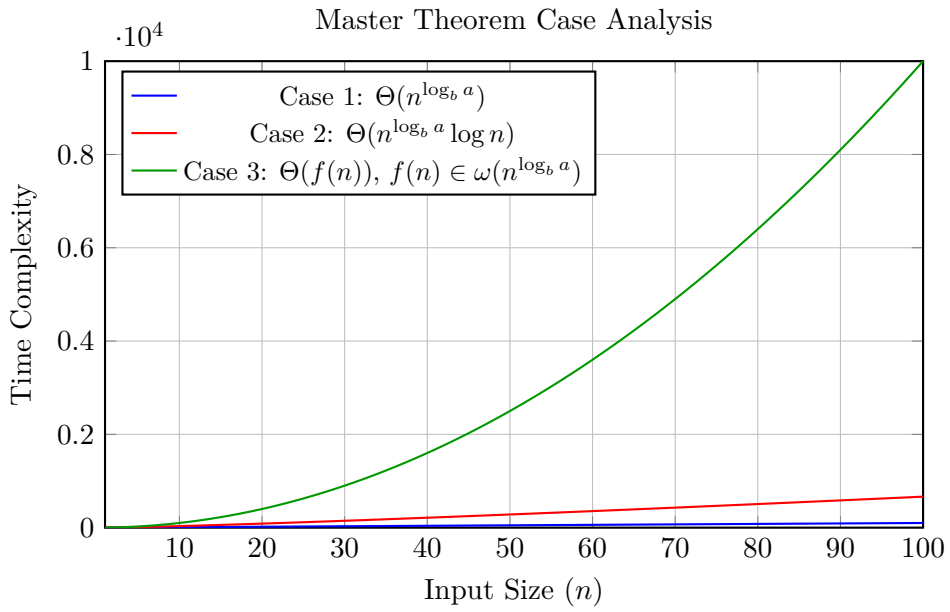


Figure 25.2: Visual representation of Master Theorem cases

25.2.4 Quick Sort: Probabilistic Analysis

```cpp
#include <vector>
#include <algorithm>

class QuickSort {
public:
    static void quickSort(std::vector<int>& arr, int low, int high) {
        if (low < high) {
            int pi = partition(arr, low, high);
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

private:
    static int partition(std::vector<int>& arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                std::swap(arr[i], arr[j]);
            }
        }
        std::swap(arr[i + 1], arr[high]);
        return i + 1;
```

```
27      }
28  };
```

Listing 25.3: Quick Sort Implementation

Theorem 25.5 (Quick Sort Expected Complexity)**.** *The expected time complexity of randomized quick sort is* $\Theta(n \log n)$*.*

Proof. Let X_{ij} be an indicator random variable that is 1 if elements z_i and z_j are compared during the execution of the algorithm, and 0 otherwise, where $z_1, z_2, \ldots, z_n$ are the elements in sorted order.
The total number of comparisons is:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \tag{25.16}$$

By linearity of expectation:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1] \tag{25.17}$$

Elements z_i and z_j are compared if and only if one of them is chosen as a pivot before any element z_k where $i < k < j$ is chosen as a pivot.
The probability that z_i is compared with z_j is:

$$\Pr[X_{ij} = 1] = \frac{2}{j - i + 1} \tag{25.18}$$

Therefore:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \tag{25.19}$$

$$= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \quad \text{(substituting } k = j - i + 1\text{)} \tag{25.20}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \tag{25.21}$$

$$= 2(n-1) \sum_{k=1}^{n} \frac{1}{k} \tag{25.22}$$

$$= 2(n-1) htbp_n \tag{25.23}$$

where $htbp_n$ is the n-th harmonic number. Since $htbp_n = \Theta(\log n)$:

$$E[X] = O(n \log n) \tag{25.24}$$

□

25.2.5 Fibonacci Sequence: Dynamic Programming vs. Naive Recursion

```python
# Recursive implementation - O(2^n)
def fibonacci_naive(n):
    if n <= 2:
        return 1
    return fibonacci_naive(n-1) + fibonacci_naive(n-2)

# Dynamic programming implementation - O(n)
def fibonacci_dp(n):
    if n <= 2:
        return 1

    dp = [0] * (n + 1)
    dp[1] = dp[2] = 1

    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2]

    return dp[n]

# Memoized recursive implementation - O(n)
def fibonacci_memo(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 2:
        return 1

    memo[n] = fibonacci_memo(n-1, memo) + fibonacci_memo(n-2, memo)
    return memo[n]
```

Listing 25.4: Fibonacci Implementations Comparison

Theorem 25.6 (Fibonacci Complexity Analysis)**.** *1. Naive recursive Fibonacci has time complexity* $\Theta(\phi^n)$ *where* $\phi = \frac{1+\sqrt{5}}{2}$ *is the golden ratio.*

2. Dynamic programming and memoized versions have time complexity $\Theta(n)$ *and space complexity* $\Theta(n)$*.*

Proof. For the naive recursive version, let $T(n)$ be the time to compute $F(n)$.
The recurrence relation is:

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \tag{25.25}$$

This is similar to the Fibonacci recurrence itself. The characteristic equation is:

$$x^2 = x + 1 \Rightarrow x^2 - x - 1 = 0 \tag{25.26}$$

The roots are:

$$x = \frac{1 \pm \sqrt{5}}{2} \tag{25.27}$$

The golden ratio $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the larger root.
Therefore, $T(n) = \Theta(\phi^n)$.
For the dynamic programming version, we compute each Fibonacci number exactly once, performing constant work for each. Therefore, $T(n) = \Theta(n)$. □

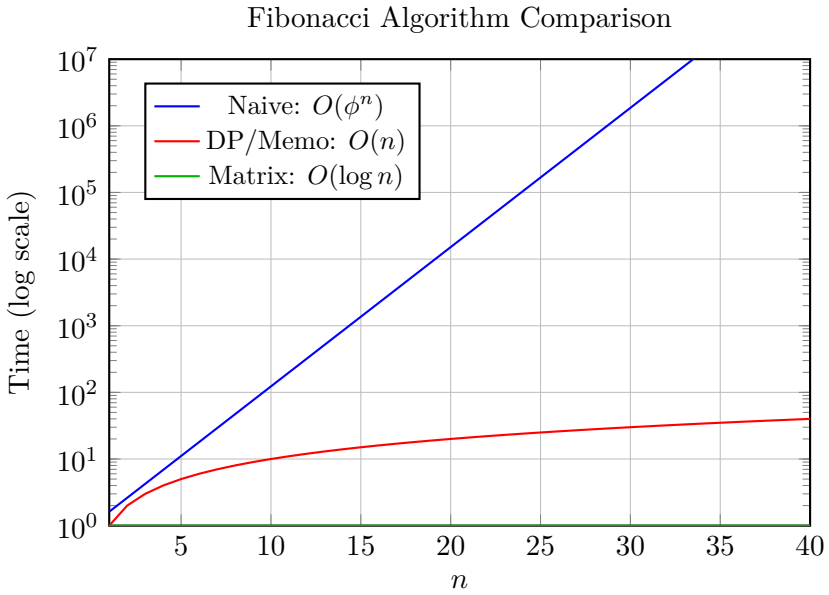


Figure 25.3: Comparison of Fibonacci algorithm complexities

25.2.6 Haskell Functional Programming: Lazy Evaluation Impact

```haskell
-- Merge Sort in Haskell
mergeSort :: Ord a => [a] -> [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs = merge (mergeSort left) (mergeSort right)
  where
    (left, right) = splitAt (length xs `div` 2) xs

merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys

-- Infinite Fibonacci sequence using lazy evaluation
fibonacci :: [Integer]
```

```haskell
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)

-- Get nth Fibonacci number in O(n) time, O(1) space
fib :: Int -> Integer
fib n = fibonacci !! n
```

Listing 25.5: Haskell Merge Sort with Lazy Evaluation

Theorem 25.7 (Lazy Evaluation Complexity)**.** *In Haskell's lazy evaluation model, the infinite Fibonacci sequence can be computed with:*

- *Time complexity:* $O(n)$ *to compute the first* n *terms*
- *Space complexity:* $O(1)$ *amortized per element due to garbage collection*

Proof. The infinite list `fibonacci` is defined as:

$$\text{fibonacci} = [F_0, F_1, F_2, F_3, \ldots] \tag{25.28}$$

where each F_i is computed as $F_{i-1} + F_{i-2}$ using `zipWith`.
When we request `fibonacci !! n`, Haskell's lazy evaluation computes only the necessary elements. Each F_i is computed exactly once and takes constant time (assuming constant-time addition).
The space complexity is amortized $O(1)$ because once an element is computed and used, it can be garbage collected if no longer referenced. □

25.2.7 Matrix Exponentiation: Logarithmic Fibonacci

Theorem 25.8 (Matrix Fibonacci Formula)**.** *The* n*-th Fibonacci number can be computed using matrix exponentiation:*

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{25.29}$$

Proof. Let $M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. We prove by induction that:

$$M^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \tag{25.30}$$

Base case ($n = 1$):

$$M^1 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix} \tag{25.31}$$

Inductive step: Assume the formula holds for n. Then:

$$M^{n+1} = M^n \cdot M \tag{25.32}$$

$$= \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \tag{25.33}$$

$$= \begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} \tag{25.34}$$

$$= \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix} \tag{25.35}$$

Using fast matrix exponentiation, we can compute M^n in $O(\log n)$ time. □

25.3 Advanced Complexity Analysis

25.3.1 Amortized Analysis: Dynamic Arrays

Definition 25.4 (Amortized Complexity)**.** The amortized cost of an operation is the average cost per operation over a sequence of operations, even if individual operations may have different costs.

Theorem 25.9 (Dynamic Array Amortized Analysis)**.** *In a dynamic array with doubling strategy, the amortized cost of insertion is* $O(1)$.

Proof. Consider a sequence of n insertions starting with an empty array. Let c_i be the cost of the i-th insertion.
For most insertions, $c_i = 1$ (simple append). htbpowever, when the array is full and needs to be doubled, $c_i = i$ (copy all existing elements plus insert new one).
Array doubling occurs when the array size is a power of 2. The total cost is:

$$\sum_{i=1}^{n} c_i = n + \sum_{j=0}^{\lfloor \log_2 n \rfloor} 2^j \tag{25.36}$$

$$= n + (2^{\lfloor \log_2 n \rfloor + 1} - 1) \tag{25.37}$$

$$< n + 2n = 3n \tag{25.38}$$

Therefore, the amortized cost per insertion is $\frac{3n}{n} = 3 = O(1)$. □

25.3.2 Probabilistic Analysis: Skip Lists

Definition 25.5 (Skip List)**.** A skip list is a probabilistic data structure that maintains a sorted list with $O(\log n)$ expected search time.

Theorem 25.10 (Skip List Complexity)**.** *In a skip list with* n *elements:*

- *Expected search time:* $O(\log n)$
- *Expected space:* $O(n)$

- *Expected height:* $O(\log n)$

Proof. Let X_i be the indicator random variable for whether element i appears at level k. The probability that an element appears at level k is p^k where $p = 1/2$.
The expected number of elements at level k is:

$$E[\text{elements at level } k] = n \cdot p^k = n \cdot 2^{-k} \tag{25.39}$$

The expected height of the skip list is $O(\log n)$ because the probability that any element reaches height $c \log n$ is:

$$\Pr[\text{height} \geq c \log n] = n \cdot 2^{-c \log n} = n \cdot n^{-c} = n^{1-c} \tag{25.40}$$

For $c > 1$, this probability approaches 0 as $n \to \infty$.
The expected search time is $O(\log n)$ because we expect to traverse $O(\log n)$ levels and $O(1)$ nodes per level. □

25.3.3 Parallel Complexity: Work and Span

Definition 25.6 (Work and Span)**.** For a parallel algorithm:

- **Work** T_1: Total number of operations
- **Span** T_∞: Length of the critical path
- **Parallelism** $P = T_1/T_\infty$: Maximum theoretical speedup

Theorem 25.11 (Parallel Merge Sort)**.** *Parallel merge sort has:*

- *Work:* $T_1(n) = O(n \log n)$
- *Span:* $T_\infty(n) = O(\log^2 n)$
- *Parallelism:* $O(n/\log n)$

Proof. The work analysis is identical to sequential merge sort: $T_1(n) = O(n \log n)$.
For span analysis, the recurrence is:

$$T_\infty(n) = T_\infty(n/2) + O(\log n) \tag{25.41}$$

The $O(\log n)$ term comes from parallel merging. Solving this recurrence:

$$T_\infty(n) = O(\log^2 n) \tag{25.42}$$

The parallelism is:

$$P = \frac{T_1(n)}{T_\infty(n)} = \frac{O(n \log n)}{O(\log^2 n)} = O\left(\frac{n}{\log n}\right) \tag{25.43}$$

□

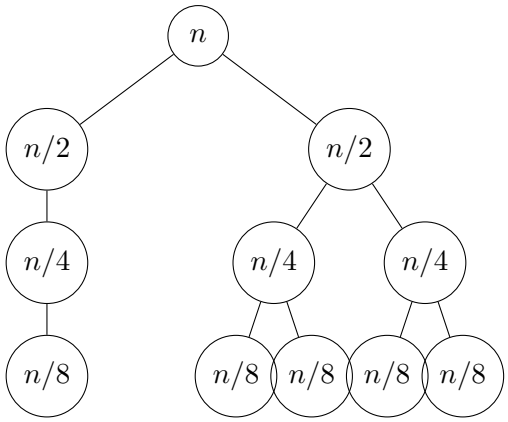


Figure 25.4: Parallel merge sort recursion tree showing work distribution

25.4 Space Complexity Analysis

25.4.1 Recursive Space Complexity

Theorem 25.12 (Recursive Call Stack Space)**.** *For a recursive algorithm with maximum recursion depth d and space usage s per recursive call, the total space complexity is $O(d \cdot s)$.*

```python
# Tail-recursive factorial - O(n) time, O(n) space due to call stack
def factorial_recursive(n):
    if n <= 1:
        return 1
    return n * factorial_recursive(n - 1)

# Iterative factorial - O(n) time, O(1) space
def factorial_iterative(n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Tree traversal - O(n) time, O(h) space where h is tree height
def inorder_traversal(root, result=[]):
    if root:
        inorder_traversal(root.left, result)
        result.append(root.val)
        inorder_traversal(root.right, result)
    return result

# Matrix multiplication - O(n^3) time, O(n^2) space
def matrix_multiply(A, B):
    n = len(A)
    C = [[0] * n for _ in range(n)]

    for i in range(n):
```

```
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]

    return C
```

Listing 25.6: Space Complexity Examples

25.4.2 Memory htbpierarchy and Cache Complexity

Definition 25.7 (Cache-Oblivious Algorithm)**.** An algorithm whose performance is optimal across all levels of memory hierarchy without explicit knowledge of cache parameters.

Theorem 25.13 (Cache-Oblivious Matrix Multiplication)**.** *The recursive matrix multiplication algorithm achieves optimal cache performance with:*

- *Work:* $O(n^3)$
- *Cache misses:* $O(n^3/B + n^2/\sqrt{M})$

where B *is block size and* M *is cache size.*

```
template<typename T>
void matrix_multiply_recursive(
    const T* A, const T* B, T* C,
    int n, int rowA, int colA, int rowB, int colB, int rowC, int colC)
    {

    if (n <= ThtbpREShtbpOLD) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    C[(rowC + i) * n + (colC + j)] +=
                        A[(rowA + i) * n + (colA + k)] *
                        B[(rowB + k) * n + (colB + j)];
                }
            }
        }
        return;
    }

    int half = n / 2;

    matrix_multiply_recursive(A, B, C, half, rowA, colA, rowB, colB,
    rowC, colC);
    matrix_multiply_recursive(A, B, C, half, rowA, colA + half, rowB +
    half, colB, rowC, colC);

```

```
    matrix_multiply_recursive(A, B, C, half, rowA, colA, rowB, colB + half, rowC, colC + half);
    matrix_multiply_recursive(A, B, C, half, rowA, colA + half, rowB + half, colB + half, rowC, colC + half);

    matrix_multiply_recursive(A, B, C, half, rowA + half, colA, rowB, colB, rowC + half, colC);
    matrix_multiply_recursive(A, B, C, half, rowA + half, colA + half, rowB + half, colB, rowC + half, colC);

    matrix_multiply_recursive(A, B, C, half, rowA + half, colA, rowB, colB + half, rowC + half, colC + half);
    matrix_multiply_recursive(A, B, C, half, rowA + half, colA + half, rowB + half, colB + half, rowC + half, colC + half);
}
```

Listing 25.7: Cache-Oblivious Matrix Multiplication

25.5 Graph Algorithms Complexity Analysis

25.5.1 Depth-First Search (DFS)

```python
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=' ')

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs_recursive(graph, neighbor, visited)

    return visited

def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex, end=' ')

            # Add neighbors to stack
            for neighbor in graph[vertex]:
```

```
                if neighbor not in visited:
                    stack.append(neighbor)

    return visited
```

Listing 25.8: DFS Implementation and Analysis

Theorem 25.14 (DFS Complexity)**.** *Depth-First Search on a graph $G = (V, E)$ has:*

- *Time complexity: $O(V + E)$*
- *Space complexity: $O(V)$ for visited set plus $O(V)$ for recursion stack*

Proof. Each vertex is visited exactly once, requiring $O(V)$ time. For each vertex v, we examine all edges incident to v. Since each edge is examined twice (once from each endpoint), the total time for edge examination is $O(E)$.
Therefore, total time complexity is $O(V + E)$.
Space complexity includes:

- Visited set: $O(V)$ space
- Recursion stack: $O(V)$ in worst case (linear graph)

Total space: $O(V)$. □

25.5.2 Dijkstra's Algorithm

```
import heapq
from collections import defaultdict

def dijkstra(graph, start):
    distances = {node: float('infinity') for node in graph}
    distances[start] = 0
    pq = [(0, start)]
    visited = set()

    while pq:
        current_distance, current_node = heapq.heappop(pq)

        if current_node in visited:
            continue

        visited.add(current_node)

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))
```

```
24
25      return distances
```

Listing 25.9: Dijkstra's Shortest Path Algorithm

Theorem 25.15 (Dijkstra's Algorithm Complexity)**.** *Dijkstra's algorithm using a binary heap has:*

- *Time complexity:* $O((V + E) \log V)$
- *Space complexity:* $O(V)$

Proof. The algorithm maintains a priority queue of vertices. Each vertex is extracted from the queue exactly once, requiring $O(V \log V)$ time total.
Each edge is relaxed at most once. Edge relaxation may involve decreasing a key in the priority queue, which takes $O(\log V)$ time. Since there are E edges, this contributes $O(E \log V)$ time.
Total time complexity: $O(V \log V + E \log V) = O((V + E) \log V)$.
Space complexity is $O(V)$ for the distance array and priority queue. □

25.5.3 Floyd-Warshall Algorithm

```python
def floyd_warshall(graph):
    V = len(graph)
    dist = [[float('inf')] * V for _ in range(V)]

    for i in range(V):
        dist[i][i] = 0

    for i in range(V):
        for j in range(V):
            if graph[i][j] != 0:
                dist[i][j] = graph[i][j]

    for k in range(V):
        for i in range(V):
            for j in range(V):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist

def floyd_warshall_with_path_reconstruction(graph):
    V = len(graph)
    dist = [[float('inf')] * V for _ in range(V)]
    next_vertex = [[None] * V for _ in range(V)]

    for i in range(V):
        dist[i][i] = 0
```

```
28
29      for i in range(V):
30          for j in range(V):
31              if graph[i][j] != 0:
32                  dist[i][j] = graph[i][j]
33                  next_vertex[i][j] = j
34
35      for k in range(V):
36          for i in range(V):
37              for j in range(V):
38                  if dist[i][k] + dist[k][j] < dist[i][j]:
39                      dist[i][j] = dist[i][k] + dist[k][j]
40                      next_vertex[i][j] = next_vertex[i][k]
41
42      return dist, next_vertex
```

Listing 25.10: Floyd-Warshall All-Pairs Shortest Path

Theorem 25.16 (Floyd-Warshall Correctness and Complexity)**.** *The Floyd-Warshall algorithm correctly computes all-pairs shortest paths with:*

- *Time complexity:* $\Theta(V^3)$
- *Space complexity:* $\Theta(V^2)$

Correctness. Let $d_{ij}^{(k)}$ be the shortest path distance from i to j using only vertices $\{1, 2, \ldots, k\}$ as intermediate vertices.
The recurrence relation is:

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \tag{25.44}$$

Base case: $d_{ij}^{(0)} = w_{ij}$ (direct edge weight or ∞).
The algorithm implements this recurrence, computing $d_{ij}^{(V)}$ which represents the shortest path using all vertices as potential intermediates.
Time complexity: Three nested loops, each running V times, gives $\Theta(V^3)$.
Space complexity: $\Theta(V^2)$ for the distance matrix. □

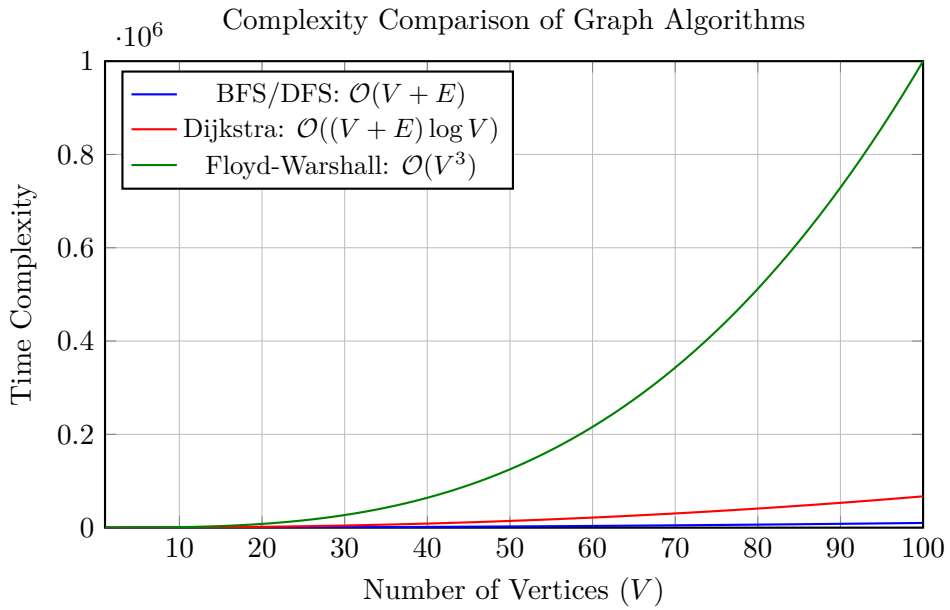


Figure 25.5: Asymptotic time complexities of common graph algorithms under dense graph assumptions ($E = \mathcal{O}(V^2)$).

25.6 Advanced Mathematical Techniques

25.6.1 Generating Functions for Recurrence Relations

Definition 25.8 (Generating Function)**.** For a sequence $a_0, a_1, a_2, \ldots$, the generating function is:

$$G(x) = \sum_{n=0}^{\infty} a_n x^n \tag{25.45}$$

Theorem 25.17 (Fibonacci Generating Function)**.** *The generating function for Fibonacci numbers is:*

$$F(x) = \frac{x}{1 - x - x^2} \tag{25.46}$$

Proof. Let $F(x) = \sum_{n=0}^{\infty} F_n x^n$ where $F_0 = 0, F_1 = 1$.

From the Fibonacci recurrence $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$:

$$F(x) = F_0 + F_1 x + \sum_{n=2}^{\infty} F_n x^n \tag{25.47}$$

$$= 0 + x + \sum_{n=2}^{\infty} (F_{n-1} + F_{n-2}) x^n \tag{25.48}$$

$$= x + \sum_{n=2}^{\infty} F_{n-1} x^n + \sum_{n=2}^{\infty} F_{n-2} x^n \tag{25.49}$$

$$= x + x \sum_{n=2}^{\infty} F_{n-1} x^{n-1} + x^2 \sum_{n=2}^{\infty} F_{n-2} x^{n-2} \tag{25.50}$$

$$= x + x \sum_{m=1}^{\infty} F_m x^m + x^2 \sum_{k=0}^{\infty} F_k x^k \tag{25.51}$$

$$= x + x(F(x) - F_0) + x^2 F(x) \tag{25.52}$$

$$= x + xF(x) + x^2 F(x) \tag{25.53}$$

Solving for $F(x)$:

$$F(x) = x + xF(x) + x^2 F(x) = x + F(x)(x + x^2) \tag{25.54}$$

$$F(x)(1 - x - x^2) = x \tag{25.55}$$

$$F(x) = \frac{x}{1 - x - x^2} \tag{25.56}$$

□

25.6.2 Probabilistic Method in Algorithm Analysis

Theorem 25.18 (Randomized QuickSelect Expected Time)**.** *The expected time complexity of randomized QuickSelect to find the k-th smallest element is $O(n)$.*

```python
import random

def quickselect(arr, k):
    if len(arr) == 1:
        return arr[0]

    pivot = random.choice(arr)

    less = [x for x in arr if x < pivot]
    equal = [x for x in arr if x == pivot]
    greater = [x for x in arr if x > pivot]

    if k < len(less):
        return quickselect(less, k)
    elif k < len(less) + len(equal):
        return pivot
    else:
```

```
        return quickselect(greater, k - len(less) - len(equal))

# Deterministic median-of-medians version - O(n) worst case
def median_of_medians(arr, k):
    if len(arr) <= 5:
        return sorted(arr)[k]

    groups = [arr[i:i+5] for i in range(0, len(arr), 5)]
    medians = [sorted(group)[len(group)//2] for group in groups]

    pivot = median_of_medians(medians, len(medians)//2)

    less = [x for x in arr if x < pivot]
    equal = [x for x in arr if x == pivot]
    greater = [x for x in arr if x > pivot]

    if k < len(less):
        return median_of_medians(less, k)
    elif k < len(less) + len(equal):
        return pivot
    else:
        return median_of_medians(greater, k - len(less) - len(equal))
```

Listing 25.11: Randomized QuickSelect Implementation

Expected Time Analysis of QuickSelect. Let $T(n)$ be the expected time to find the k-th element in an array of size n.

In the best case, the pivot is the median, giving us $T(n) = T(n/2) + O(n)$. In the worst case, the pivot is the minimum or maximum, giving us $T(n) = T(n-1) + O(n)$.

For a random pivot, let X be the size of the larger subproblem. The expected value of X is:

$$E[X] = \sum_{i=\lceil n/2 \rceil}^{n-1} \frac{2i}{n} = \frac{2}{n} \sum_{i=\lceil n/2 \rceil}^{n-1} i \tag{25.57}$$

Computing the sum:

$$\sum_{i=\lceil n/2 \rceil}^{n-1} i = \sum_{i=1}^{n-1} i - \sum_{i=1}^{\lceil n/2 \rceil - 1} i \tag{25.58}$$

$$= \frac{(n-1)n}{2} - \frac{(\lceil n/2 \rceil - 1)\lceil n/2 \rceil}{2} \tag{25.59}$$

$$\leq \frac{(n-1)n}{2} - \frac{(n/2-1)(n/2)}{2} \tag{25.60}$$

$$= \frac{3n^2 - 2n}{8} \tag{25.61}$$

Therefore:

$$E[X] \leq \frac{2}{n} \cdot \frac{3n^2 - 2n}{8} = \frac{3n-2}{4} \leq \frac{3n}{4} \tag{25.62}$$

The recurrence becomes:

$$E[T(n)] \leq E[T(3n/4)] + cn \tag{25.63}$$

Solving this recurrence using the substitution method, we can show that $E[T(n)] = O(n)$. □

25.6.3 Linear Programming and Simplex Algorithm Complexity

Theorem 25.19 (Simplex Algorithm Complexity)**.** *The Simplex algorithm for linear programming has:*

- *Worst-case time complexity: Exponential*
- *Average-case time complexity: Polynomial (smoothed analysis)*
- *Practical performance: Often linear in the number of constraints*

```
import numpy as np
from scipy.optimize import linprog

def solve_linear_program():
    c = [-1, -1]  # Coefficients of objective function

    A_ub = [[1, 2], [2, 1]]
    b_ub = [3, 3]

    bounds = [(0, None), (0, None)]

    result = linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

    return result
```

Listing 25.12: Simplex Algorithm Concept

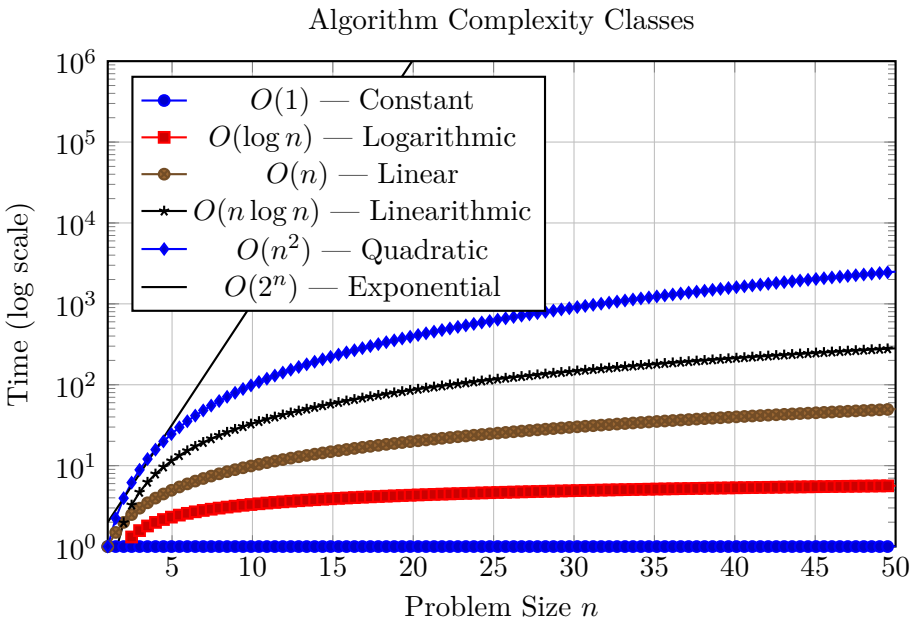


Figure 25.6: Comparison of common algorithmic complexity classes.

25.7 Conclusion and Future Directions

25.7.1 Summary of Key Mathematical Insights

The mathematical analysis of algorithmic complexity reveals several fundamental principles:

1. **Asymptotic Analysis** provides a framework for understanding algorithm scalability independent of implementation details.
2. **Recurrence Relations** and their solution techniques (Master Theorem, generating functions) enable precise complexity characterization.
3. **Probabilistic Analysis** shows that randomization can improve expected performance even when worst-case bounds remain unchanged.
4. **Amortized Analysis** reveals that sequences of operations can have better performance than individual worst-case bounds suggest.
5. **Parallel Complexity** introduces work-span models that capture the essence of parallel algorithm efficiency.

25.7.2 Advanced Topics for Further Study

- **Parametrized Complexity**: Analysis beyond input size
- **Fine-grained Complexity**: Tight bounds within polynomial time
- **Quantum Complexity**: Complexity in quantum computing models
- **Communication Complexity**: Lower bounds on information exchange
- **Streaming Algorithms**: Complexity with memory constraints

25.7.3 Mathematical Tools Summary

Technique	Application	Example
Master Theorem	Divide-and-conquer recurrences	Merge sort analysis
Generating Functions	Combinatorial recurrences	Fibonacci complexity
Probability Theory	Randomized algorithms	QuickSelect expected time
Linear Algebra	Graph algorithms	Matrix-based shortest paths
htbparmonic Analysis	Summation bounds	QuickSort average case
Potential Method	Amortized analysis	Dynamic arrays

Table 25.1: Mathematical techniques in complexity analysis

The rigorous mathematical foundation of algorithmic complexity analysis provides both theoretical insights and practical guidance for algorithm design and selection. Understanding these mathematical principles enables computer scientists to make informed decisions about algorithm efficiency and to develop new algorithms with provable performance guarantees.

Part III

Autonomous Work

Chapter 26

Exercises

This document outlines a collection of optional exercises, challenges, and projects designed to encourage autonomous learning in programming and algorithmic thinking. Students are expected to work through these tasks independently to reinforce their understanding of fundamental concepts and develop problem-solving skills.
The exercises range from basic programming problems to more complex algorithmic implementations, and they are categorized to cover a wide variety of topics. This initiative promotes active learning, creativity, and self-efficacy among students.

Objectives

- Reinforce programming fundamentals.
- Encourage logical and algorithmic thinking.
- Provide a diverse set of practice problems for self-paced learning.
- Foster autonomy and personal responsibility in the learning process.

26.1 Exercises

The exercises are organized from basic to advanced levels and cover various areas such as arithmetic operations, data structures, control flow, recursion, mathematical analysis, simulations, and string manipulation.

26.1.1 Fundamentals and Control Flow

- Add two numbers input by the user.
- Convert Celsius to Fahrenheit.
- Determine if a number is even or odd.
- Calculate the area of a circle with a given radius.
- Swap the values of two variables.

- Determine if a number is positive, negative, or zero.
- Find the maximum of three numbers.
- Display the multiplication table of a number.
- Calculate the factorial of a number.
- Reverse an integer number.
- Count the digits of a number.
- Sum the digits of a number.
- Check if a number is prime.
- Determine if a year is a leap year.
- Sum the first N natural numbers.

26.1.2 Lists and Arrays

- Find the minimum and maximum in a list.
- Calculate the average of a list.
- Count how many even and odd numbers are in a list.
- Sort a list in ascending order.
- Rotate a list to the right by one position.
- Concatenate two lists without using `+`.
- Count how many times a number appears in a list.
- Find the most frequent element in a list.
- Find the second largest number in a list.
- Remove the first element from a list.
- Sum consecutive numbers in a list.
- Find the greatest common divisor (GCD) of a list of numbers.

26.1.3 Strings and Text Processing

- Check if a word is a palindrome.
- Verify if two words are anagrams.
- Count the frequency of each letter in a string.
- Implement Caesar cipher.
- Check if a string is a pangram.

- Compress a string by removing consecutive repeated characters.
- Count characters in a string using pointers.
- Check if a phrase is a palindrome of words.

26.1.4 Mathematics and Algorithms

- Convert a decimal number to binary.
- Generate a random number between 1 and 100.
- Find the GCD and LCM of two or three numbers.
- Calculate the area of a triangle (Heron's formula).
- Solve a quadratic equation.
- Implement the Collatz conjecture.
- Calculate volume of a sphere.
- Euler's number estimation.
- Determine if a number is perfect, abundant, or a triangular number.
- Determine if a number is a Fibonacci, Kaprekar, Narcissistic, or Mersenne number.
- Use the Leibniz series to approximate π.
- Use Newton's method to compute square and cube roots.
- Implement the Binomial Theorem and Stirling's approximation.
- Use Taylor series to approximate logarithms.
- Generate Fibonacci, Lucas, and Padovan sequences.
- Determine arithmetic or geometric sequences.
- Sum of squares of the first N natural or odd numbers.

26.1.5 Simulation and Games

- Simulate a dice roll.
- Simulate Rock-Paper-Scissors.
- Number guessing game.
- Generate Pythagorean triplets.
- Random number generator using uniform or normal distribution.

26.1.6 Graphics and Patterns

- Print a pyramid pattern of asterisks.
- Print right-aligned, inverted, and square patterns.
- Print a hollow rectangle of size $N \times M$.
- Print an asterisk chessboard with `*` and `+`.
- Print patterns in shapes of "X", "Z", Floyd's triangle, Pascal's triangle.
- Print a zig-zag number pattern.

26.1.7 Data Structures and Memory

- Show memory address and size of a variable.
- Swap two numbers using pointers.
- Sum array elements using pointers.
- Find the maximum value in an array using pointers.
- Dynamically allocate an array of integers.

26.1.8 Applications and Systems

- Implement a basic calculator supporting $+, -, *, /$.
- Movie ticket reservation system.
- Student grade management system.
- Currency converter.
- User authentication system.
- Expression evaluator.
- Voting system with automatic counting.
- Game score tracking system.
- Calorie and BMI calculator.

Chapter 27

Introductory Student Projects

Objective: This section includes a set of beginner-friendly programming projects aimed at first-year computer science students. These projects help in developing logical thinking, understanding basic syntax, and practicing real-world problem-solving.

27.1 Beginner Level

- **Student Grade Calculator** – Calculates average and letter grades based on user input.
- **Simple Banking System** – Provides basic banking features like deposit, withdraw, and view balance.
- **Temperature Converter** – Converts between Celsius, Fahrenheit, and Kelvin.
- **Basic Calculator** – Handles basic arithmetic operations through a menu.
- **Age Calculator** – Calculates age from birth date in years and months.

27.2 Intermediate Level

- **Library Book Manager** – Add, remove, and list books with optional file storage.
- **Contact Book Application** – Phonebook program with CRUD operations.
- **To-Do List Application** – Task management with status updates.
- **Simple Voting System** – Voting with candidates and automatic tally.
- **Number Guessing Game** – Feedback-driven guessing game with scoring.

27.3 Advanced Beginner Level

- **Expense Tracker** – Categorizes and tracks expenses over time.
- **Password Generator & Validator** – Secure password generation and strength check.

- **Simple Chatbot** – Keyword-matching chatbot for common queries.
- **Maze Solver** – 2D array maze with DFS or BFS implementation.
- **Calendar Generator** – Displays calendar for given month/year.

Chapter 28

Technical Coding Challenges from Industry Interviews

This section compiles a set of well-known technical problems inspired by real coding interviews at leading technology companies. Each challenge is designed to strengthen your skills in algorithmic design, code implementation, and logic visualization through flowcharts. The problems are categorized by difficulty and cover areas such as arrays, strings, recursion, dynamic programming, and system design.

28.1 Level: Easy

1. **Two Sum Problem**
   Given an array of integers, return indices of the two numbers such that they add up to a specific target.
   Source: Amazon, Google.

2. **Reverse a Linked List**
   Reverse a singly linked list in-place.
   Source: Meta, Microsoft.

3. **Check for Palindrome String**
   Determine whether a string is a palindrome, ignoring non-alphanumeric characters.
   Source: Apple, Amazon.

4. **FizzBuzz Variant**
   Print numbers from 1 to N, but for multiples of 3 print "Fizz", for 5 print "Buzz", and for both print "FizzBuzz".
   Source: Google.

5. **Merge Two Sorted Arrays**
   Merge two sorted arrays into one sorted array without using built-in sort.
   Source: Microsoft.

28.2 Level: Medium

1. **Longest Substring Without Repeating Characters**
   Given a string, find the length of the longest substring without repeating characters.
   Source: Amazon, Google.

2. **Detect Cycle in a Linked List**
   Return true if a cycle is detected using Floyd's algorithm (Tortoise and Hare).
   Source: Facebook.

3. **Valid Parentheses**
   Check if a string containing just the characters '(', ')', '', '', '[' and ']' is valid.
   Source: Microsoft, Adobe.

4. **Matrix Rotation (90 degrees)**
   Rotate a given N x N matrix by 90 degrees clockwise in-place.
   Source: Google, Apple.

5. **Top K Frequent Elements**
   Return the K most frequent elements in an array. Use a heap or bucket sort.
   Source: Amazon, TikTok.

28.3 Level: Hard

1. **Word Ladder**
   Given two words and a dictionary, find the shortest transformation sequence from start to end by changing only one letter at a time.
   Source: Google, Bloomberg.

2. **Trapping Rain Water**
   Compute how much water it can trap after raining given an elevation map.
   Source: Apple, Microsoft.

3. **Serialize and Deserialize Binary Tree**
   Implement methods to convert a binary tree to a string and vice versa.
   Source: Meta, Google.

4. **Median of Two Sorted Arrays**
   Find the median of two sorted arrays in $O(\log(min(n, m)))$ *time.*
   Source: Google, Uber.

5. **Longest Valid Parentheses**
   Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) substring.
   Source: Amazon, Twitter.

28.4 Level: Advanced

1. **Design LRU Cache**
   Design a data structure that follows the constraints of a Least Recently Used (LRU) cache. Implement with O(1) time for get and put.
   Source: Facebook, Netflix.

2. **N-Queens Problem**
   Place N queens on an N x N chessboard such that no two queens threaten each other. Return all possible configurations.
   Source: Google, Microsoft.

3. **Implement Trie (Prefix Tree)**
   Design a trie with insert, search, and startsWith methods.
   Source: Amazon, Meta.

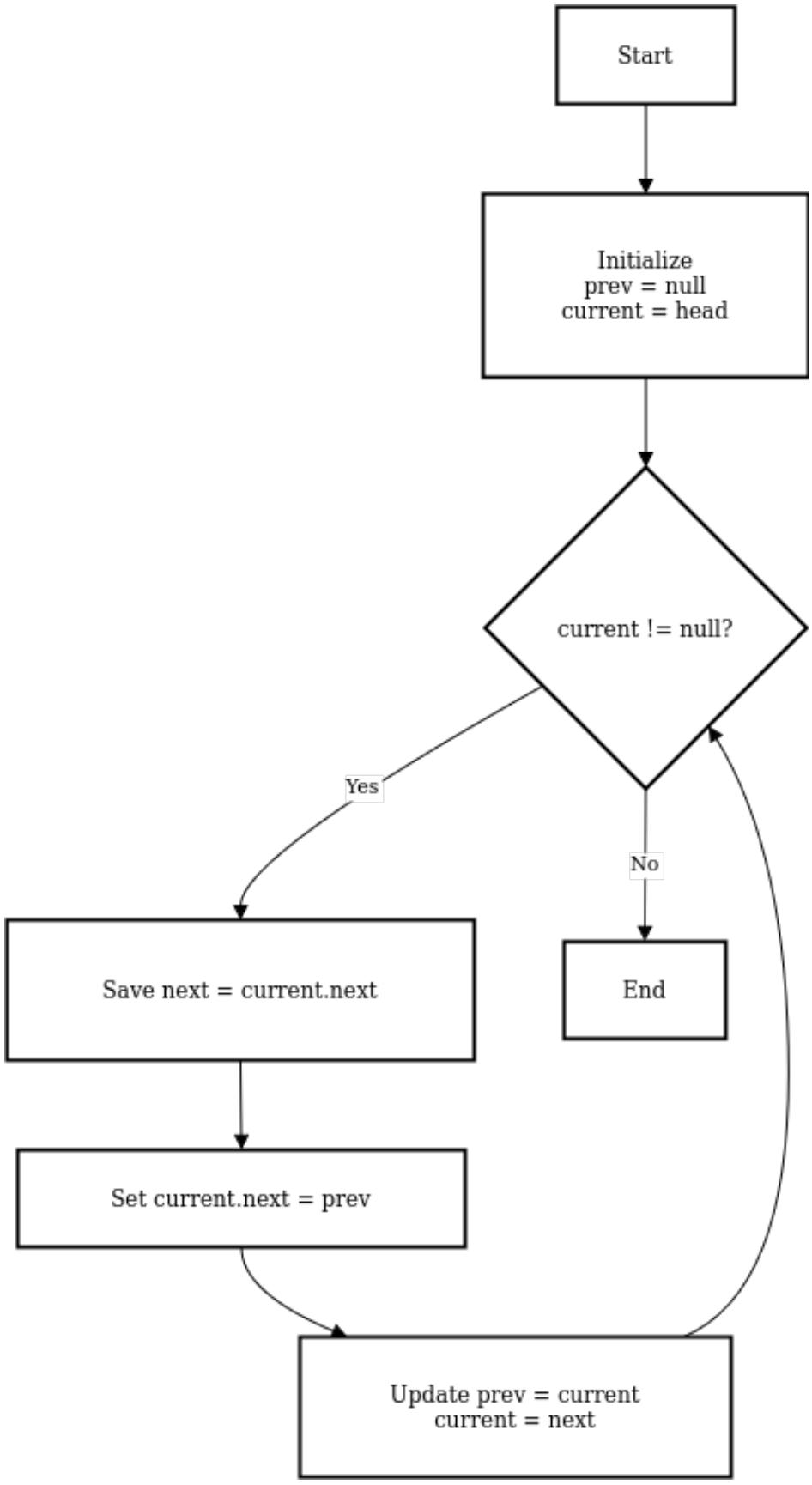
4. **Concurrent File System (Design Question)**
   Design a simplified file system that supports concurrency with locking.
   Source: Google, Dropbox.

5. **Hard Dynamic Programming – Edit Distance**
   Given two strings, find the minimum number of operations required to convert one into the other (insert, delete, replace).
   Source: Google, Adobe.

28.5 Visualization (Flowchart)

Students are encouraged to create flowcharts for each algorithm to better understand the control flow and decision-making process. Tools such as `draw.io`, `Lucidchart`, or LaTeX packages like `TikZ` and `pgfplots` can be used.

Example: Reverse a Linked List




```
mov EBX, EAX     ; current = head
xor ECX, ECX     ; prev = null
```

```
3
4  reverse_loop:
5      test EBX, EBX
6      jz done
7
8      mov EDX, [EBX+4]    ; next = current->next
9      mov [EBX+4], ECX    ; current->next = prev
10     mov ECX, EBX        ; prev = current
11     mov EBX, EDX        ; current = next
12     jmp reverse_loop
```

Listing 28.1: Assembly

```
1  struct Node {
2      int data;
3      struct Node* next;
4  };
5
6  struct Node* reverse(struct Node* head) {
7      struct Node* prev = NULL;
8      struct Node* current = head;
9      struct Node* next = NULL;
10
11     while (current != NULL) {
12         next = current->next;
13         current->next = prev;
14         prev = current;
15         current = next;
16     }
17
18     return prev;
19 }
```

Listing 28.2: C

```
1  struct Node {
2      int data;
3      Node* next;
4  };
5
6  Node* reverse(Node* head) {
7      Node* prev = nullptr;
8      Node* current = head;
9
10     while (current) {
11         Node* next = current->next;
12         current->next = prev;
13         prev = current;
```

```
        current = next;
    }

    return prev;
}
```

Listing 28.3: C++

```
reverseList :: [a] -> [a]
reverseList = foldl (flip (:)) []
```

Listing 28.4: Haskell

```
reverse_list([], []).
reverse_list([H|T], Reversed) :-
    reverse_list(T, RT),
    append(RT, [H], Reversed).
```

Listing 28.5: Prolog

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

def reverse(head):
    prev = None
    current = head

    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node

    return prev
```

Listing 28.6: Python

```
reverse_list(L) -> reverse_list(L, []).

reverse_list([], Acc) -> Acc;
reverse_list([H|T], Acc) ->
    reverse_list(T, [H|Acc]).
```

Listing 28.7: Erlang

```
class Node {
    data: number;
```

```
    next: Node | null;

    constructor(data: number) {
        this.data = data;
        this.next = null;
    }
}

function reverse(head: Node | null): Node | null {
    let prev: Node | null = null;
    let current: Node | null = head;

    while (current !== null) {
        const next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }

    return prev;
}
```

Listing 28.8: Js/Ts

Each problem is an opportunity to explore advanced coding patterns such as recursion, divide-and-conquer, greedy algorithms, and graph traversal. These exercises not only prepare students for technical interviews but also deepen their computational thinking.