AP Computer Science Principles Summary

Robin Wiethüchter (AI Assisted)

October 24, 2025

DISCLAIMER: WORK IN PROGRESS

This document is currently under development. Sections may be incomplete, contain errors, or be subject to significant changes.

Contents

1 Big Idea 1: Creative Development

The process of developing computational artifacts (like programs, digital images, audio, video, presentations, or web pages) is iterative and often collaborative. This Big Idea explores how computing innovations are created and their potential impacts.

1.1 Collaboration

Developing complex computational artifacts often requires collaboration, where individuals bring diverse perspectives and skills. Effective collaboration includes:

- Communication: Clearly explaining ideas, progress, and problems.
- Shared Goals: Working towards a common objective.
- Contribution: Each member actively participates.
- Conflict Resolution: Addressing disagreements constructively.
- Using Tools: Employing version control (like Git) or shared document platforms to manage contributions.

1.2 Program Function and Purpose

Every program or computational artifact is designed with a specific purpose and function.

- **Purpose**: The problem the program aims to solve or the creative expression it intends to convey (e.g., calculating grades, simulating planetary motion, creating interactive art).
- **Function**: What the program actually does; the specific tasks it performs (e.g., takes user input, performs calculations, displays output, responds to events).

Understanding the intended purpose helps guide the development process and evaluate the final product.

1.3 Program Design and Development

This is an iterative process involving cycles of designing, implementing, testing, and refining. Key phases include:

1. **Investigation/Understanding the Problem**: Defining the requirements, purpose, and target audience.
2. **Design**: Planning the program's structure, algorithms, user interface, and data representation. This might involve pseudocode, flowcharts, or diagrams.
3. **Implementation (Coding)**: Writing the program code in a chosen programming language.
4. **Testing**: Identifying and fixing errors (debugging). This involves running the program with various inputs and scenarios.

5. **Refinement**: Improving the program based on testing, feedback, or new requirements. This could involve adding features, optimizing performance, or enhancing usability.
6. **Documentation**: Explaining how the program works, how to use it, and the design choices made. This can include comments in the code and external documents.

This process is rarely linear; developers often loop back to earlier phases.

1.4 Identifying and Correcting Errors (Debugging)

Errors (bugs) are inevitable in programming. Debugging strategies include:

- **Testing**: Systematically checking program behavior with different inputs.
- **Reading Error Messages**: Understanding compiler/interpreter messages.
- **Print Statements/Logging**: Inserting temporary output commands to track program flow and variable values.
- **Using a Debugger Tool**: Step-by-step execution, inspecting variables, setting breakpoints.
- **Simplifying the Problem**: Isolating the error by commenting out code sections or testing smaller parts.

Different types of errors exist:

- **Syntax Errors**: Violations of the programming language's grammar rules (caught by the compiler/interpreter).
- **Runtime Errors**: Errors occurring during program execution (e.g., division by zero, accessing invalid memory).
- **Logic Errors**: The program runs but produces incorrect results because the algorithm or its implementation is flawed. These are often the hardest to find.

2 Big Idea 2: Data

Computers store, process, and transmit information digitally using binary data. This Big Idea focuses on how data is represented, manipulated, and used.

2.1 Bits and Bytes

The fundamental unit of digital information is the **bit** (binary digit), which can represent one of two states: 0 or 1. Bits are grouped together to represent more complex data.

- **Byte**: A group of 8 bits. A common unit for measuring data size.
- **Number of States**: With n bits, you can represent 2^n distinct states or values. For example, 8 bits (1 byte) can represent $2^8 = 256$ different values (often 0-255). Adding one more bit doubles the number of possible values.

2.2 Number Systems

Computers use binary internally, but humans often use other systems for convenience.

- **Binary (Base-2)**: Uses only digits 0 and 1. Each position represents a power of 2 (e.g., $1101_2 = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 8 + 4 + 0 + 1 = 13_{10}$).
- **Decimal (Base-10)**: The system we use daily, with digits 0-9. Each position represents a power of 10.

Overflow Error: Computers use a fixed number of bits (e.g., 8 bits) to represent numbers. This limits the largest value they can store (for 8 bits, often 255). An overflow error happens when a calculation's result is too large for the available bits. When this occurs, the result often "wraps around." For example, adding 1 to the maximum value might result in 0, or even a negative number if signed numbers are being used. This unexpected wrap-around leads to incorrect results.

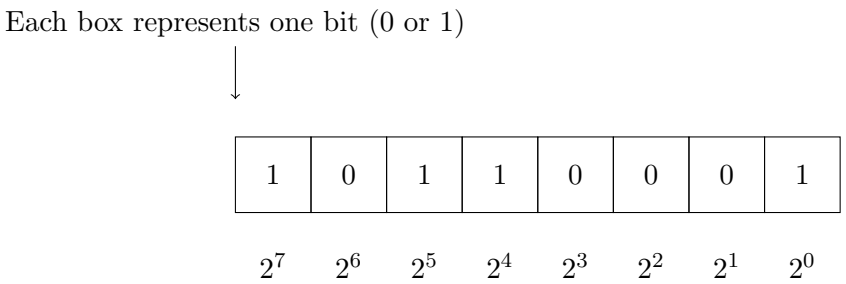
Round-off Error: Similarly, computers use a finite number of bits to represent real numbers (like decimals or fractions), which can have infinite precision. This means computers often store an *approximation* rather than the exact value. This small difference is called a round-off error. While often tiny, these errors can accumulate during calculations, potentially leading to noticeable inaccuracies in the final result (e.g., 0.1 + 0.2 might be stored as something like 0.30000000000000004 instead of exactly 0.3).

2.3 Sampling and Representing Analog Data

To represent continuous, real-world phenomena (like sound waves or visual scenes) digitally, we need to convert them into discrete values. This involves:

- **Sampling**: Measuring the phenomenon's value at regular intervals in either time (for sound) or space (for images). The **sampling rate** (or resolution) determines how frequently these measurements are taken. More samples generally lead to a more accurate representation but require more data.

This process inherently involves approximation, as the continuous reality is represented by discrete digital data. The number of bits used to store each sample (bit depth) determines the level of detail or number of distinct values that can be represented.



$$1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$
$$= 128 + 0 + 32 + 16 + 0 + 0 + 0 + 1$$
$$= 177_{10}$$

Figure 2.1: Converting binary to decimal: Each bit's value is multiplied by its position value (powers of 2)

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ & 1 & 1 & & 1 \\ + & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 \end{array} \quad \begin{array}{l} = 11_{10} \\ \\ +13_{10} \\ \\ = 24_{10} \end{array}$$

Figure 2.2: Binary addition example (1011 + 1101 = 11000) showing carry bits in red

Representing Images

Images are represented by sampling visual information across a 2D space. The image is divided into a grid of **pixels** (picture elements), each representing a sample point. For each pixel, color information is captured, often using RGB (Red, Green, Blue) values, each stored with a certain number of bits (bit depth) to represent different color intensities. Image file formats (JPEG, PNG, GIF) use different encoding and compression techniques.

Representing Sound

Sound is represented by sampling an analog sound wave's amplitude (height) at regular time intervals (the sampling rate). Each sample's amplitude is then stored using a specific number of bits (bit depth). Higher sampling rates and bit depths capture the original sound more accurately, resulting in higher fidelity but larger file sizes.

2.4 Data Compression

Compression reduces the number of bits needed to store or transmit data.

- **Lossless Compression**: Allows perfect reconstruction of the original data (e.g., ZIP, PNG). Algorithms look for redundancies (like repeated patterns) and encode them more efficiently.
- **Lossy Compression**: Discards some information deemed less important to achieve higher compression ratios (e.g., JPEG, MP3). This is acceptable for images and audio where humans may not notice the slight loss of quality. The original data cannot be perfectly recovered.

The choice depends on the data type and requirements (e.g., text needs lossless, video often uses lossy).

2.5 Extracting Information from Data

Large datasets can reveal patterns, trends, and insights not obvious from individual data points. Techniques include:

- **Filtering**: Selecting a subset of data based on criteria (e.g., show only sales from the last month).
- **Sorting**: Arranging data in a specific order (e.g., alphabetically, numerically).
- **Aggregation**: Summarizing data (e.g., calculating averages, sums, counts).
- **Visualization**: Creating charts and graphs (bar charts, scatter plots, heat maps) to make patterns easier to understand.
- **Data Mining**: Using algorithms to discover complex patterns and relationships in large datasets.
- **Machine Learning**: Using algorithms that can learn from data to make predictions or classify information automatically.

Computers are essential for processing large datasets efficiently. These extraction techniques are often implemented using programming algorithms that iterate through data structures like lists (see Section 3.6) to filter, aggregate, or transform information based on specific criteria.

```
// Example: Find all scores above 90 in a list
highScores <- [] // Create an empty list to store results
allScores <- [75, 92, 88, 95, 60, 100]

FOR EACH score IN allScores
{
    IF (score > 90)
    {
        APPEND(highScores, score) // Add score to the highScores list
    }
}

DISPLAY("High scores found:")
DISPLAY(highScores) // Output: [92, 95, 100]
```

Listing 2.1: AP Pseudocode: Filtering Data in a List

2.6 Digital Data Security and Privacy

Collecting and storing digital data raises significant concerns:

- **Security**: Protecting data from unauthorized access, modification, or destruction (e.g., using passwords, encryption, firewalls).
- **Privacy**: Controlling how personal information is collected, used, and shared. This includes issues like data breaches, surveillance, and PII (Personally Identifiable Information).

- **Anonymization**: Removing PII (Personal Identifiable Information, i.e., name, address, phone number, etc.) from datasets to protect privacy while still allowing analysis. However, re-identification can sometimes be possible by combining anonymized data with other sources.
- **Metadata Concerns**: Even metadata (e.g., file creation times, locations in photos) can reveal sensitive information.

3 Big Idea 3: Algorithms and Programming

Programming enables us to implement algorithms that solve problems. This involves using programming languages to express instructions a computer can execute.

3.1 Variables and Assignments

Variables are named storage locations for data values that can change during program execution. You can think of a variable as a box with a label on it. We can look at what value is in the box or put a new value in the box.

- **Declaration/Assignment**: In AP Pseudocode, variables are often declared implicitly when they are first assigned a value using the assignment operator (¡-). In other languages, you might need to declare the type first.

```
x <- 5          // Assigns the value 5 to variable x
y <- 10
z <- x + y      // Assigns the value of x + y (15) to z
message <- "Hello" // Assigns a string value
isValid <- true   // Assigns a boolean value

// Updating a variable's value
x <- x + 1      // Reads the current value of x (5), adds 1, assigns the
    result (6) back to x
```

Listing 3.1: AP Pseudocode: Assignment Examples

- **Naming**: Meaningful variable names (e.g., `totalScore` instead of just `s`) significantly improve code readability and understanding.

Common Data Types in Programming

Variables hold different kinds of data, known as data types. Common types include:

- **Numbers**: Used for mathematical values. Can be integers (whole numbers, like ‘5‘, ‘-10‘) or real numbers (with decimals, like ‘3.14‘). Internally represented using binary (see Section 2.2). Be aware of potential **overflow** and **round-off errors** (Section 2.2).
- **Text (Strings)**: Sequences of characters (letters, numbers, symbols) enclosed in quotes, like ‘”Hello”‘ or ‘”CSP2024”‘. Characters are represented internally using standards like ASCII or Unicode.
- **Booleans**: Represent logical values, either ‘true‘ or ‘false‘. Essential for making decisions in programs (see Section 3.3).
- **Lists (Arrays)**: Ordered collections of items, which can be of any data type (including other lists). Covered in detail in Section 3.6.

The type of data determines the operations that can be performed on it (e.g., you can do arithmetic on numbers, but not directly on strings).

3.2 Mathematical and Logical Expressions

Programs use expressions, combining values, variables, and operators, to compute new values. Categories include:

- Numbers, Strings, Variables
- Logical Operators
- Relational Operators
- Parentheses

Mathematical Expressions

Use arithmetic operators to perform calculations.

- Operators: `+`, `-`, `*`, `/`, `MOD` (modulus/remainder).
- Order of Operations: Follows standard mathematical precedence (PEMDAS/BODMAS). Parentheses `( )` can be used to force a specific evaluation order.

```
totalCost <- price * quantity
remainder <- 17 MOD 5 // remainder will be 2
average <- (a + b + c) / 3 // Parentheses ensure addition happens before division
```

Listing 3.2: AP Pseudocode: Math Expressions

Relational Operators

Relational operators compare two values and evaluate to a Boolean result (`true` or `false`).

Code Symbol	AP Symbol	Meaning
`==`	`=`	Equal to
`!=`	$\neq$	Not equal to
`>`	`>`	Greater than
`<`	`<`	Less than
`>=`	$\geq$	Greater than or equal to
`<=`	$\leq$	Less than or equal to

Logical Operators

Logical operators combine two or more Boolean expressions.

Code Symbol	AP Symbol	Meaning
`&&`	`AND`	Logical AND (true only if *both* operands are true)
`||`	`OR`	Logical OR (true if *at least one* operand is true)
`!`	`NOT`	Logical NOT (evaluates to true if the operand is false, and vice versa)

Boolean Expression Examples

- *Example 1:*

```
(x > 10) AND (y < 20)
```

  Evaluates to `true` if `x` is greater than 10 *and* `y` is less than 20. (e.g., true for `x = 11, y = 19`; false for `x = 11, y = 21`)

- *Example 2:*

```
NOT (a = b)
```

  Evaluates to `true` if `a` is *not* equal to `b`. (e.g., true for `a = 5, b = 2`; false for `a = 11, b = 11`)

- *Example 3 (Combined):*

```
(score >= 70 AND score < 100) OR (isBonusEligible = true)
```

  Evaluates to `true` if the score is between 70 (inclusive) and 100 (exclusive), *or* if the student is eligible for a bonus.

Operator Comparison (JavaScript vs AP Pseudocode)

Different programming languages may use different symbols for the same operation.

Operation	`Common Code (e.g., JS)`	`AP Pseudocode`
Assignment	`x = 5`	`x <- 5`
Equal to	`a == b`	`a = b`
Not Equal to	`a != b`	`a ≠ b`
Logical OR	`a || b`	`a OR b`
Logical AND	`a && b`	`a AND b`
Logical NOT	`!a`	`NOT a`

3.3 Control Structures

Control structures determine the order in which instructions are executed.

- **Sequence**: Instructions are executed one after another in the order they appear.
- **Selection (Conditionals)**: Executes different blocks of code based on a Boolean condition. Uses `IF`, `ELSE IF`, `ELSE`. (See details below)
- **Iteration (Loops)**: Repeats a block of code multiple times. (See details below)

Selection / Conditionals (Detailed)

Basic IF Statement

Executes a block of code only if a condition is true.

```
IF (condition)
{
    // Statements executed if condition is true
    statements
}
// Code here executes regardless
```

Listing 3.3: AP Pseudocode: Basic IF

Example:

```
score <- 100
IF (score = 100) // Note: AP uses = for comparison
{
    DISPLAY("Perfect score!")
}
```

IF-ELSE Statement

Executes one block of code if the condition is true, and a different block if the condition is false.

```
IF (condition)
{
    // Statements executed if condition is true
    statements_if_true
}
ELSE
{
    // Statements executed if condition is false
    statements_if_false
}
```

Listing 3.4: AP Pseudocode: IF-ELSE

Example:

```
temperature <- 15
IF (temperature > 20)
{
    DISPLAY("It's warm.")
}
ELSE
{
    DISPLAY("It's cool.") // This will be displayed
}
```

IF - ELSE IF - ELSE Statement

Checks multiple conditions in order. The first condition that evaluates to true has its corresponding block executed. If none are true, the optional `ELSE` block is executed.

```
IF (condition1)
{
    // Statements executed if condition1 is true
    statements1
}
ELSE IF (condition2)
{
    // Statements executed if condition1 is false AND condition2 is true
```

```
    statements2
}
ELSE // Optional
{
    // Statements executed if all preceding conditions are false
    statements_else
}
```

Listing 3.5: AP Pseudocode: IF - ELSE IF - ELSE

Example (Grading):

```
score <- 85
grade <- ""
IF (score >= 90)
{
    grade <- "A"
}
ELSE IF (score >= 80)
{
    grade <- "B" // This block executes, grade becomes "B"
}
ELSE IF (score >= 70)
{
    grade <- "C"
}
ELSE
{
    grade <- "D"
}
DISPLAY(grade)
```

Important: Only one block (the first one whose condition is met) in an IF-ELSE IF-...-ELSE structure will execute.

Multiple Independent IF Statements

Unlike ELSE IF, separate IF statements are evaluated independently.

```
score <- 100
count <- 0
IF (score >= 80) // Condition is true
{
    count <- count + 1 // count becomes 1
}
// This IF is checked regardless of the previous one
IF (score >= 90) // Condition is true
{
    count <- count + 1 // count becomes 2
}
count <- count + 1 // count becomes 3 (unconditional)
// This IF is also checked independently
IF (score >= 100) // Condition is true
{
    count <- count + 1 // count becomes 4
}
DISPLAY(count) // Displays 4
```

Listing 3.6: AP Pseudocode: Multiple Independent IFs

Here, multiple IF blocks can execute if their respective conditions are true.

Nested Conditionals

An IF statement can be placed inside another IF or ELSE block.

```
homeworkComplete <- true
score <- 95
grade <- ""

IF (homeworkComplete = true) // Outer condition
{
    // This block executes only if homeworkComplete is true
    DISPLAY("Checking score because homework is complete...")
    IF (score >= 90) // Inner condition
    {
        grade <- "A" // This executes
    }
    ELSE
    {
        grade <- "B"
    }
}
ELSE
{
    grade <- "Incomplete"
}
DISPLAY(grade) // Displays "A"
```

Listing 3.7: AP Pseudocode: Nested IFs

Indentation helps visualize the structure of nested blocks.

Iteration / Loops (Detailed)

Loops repeat blocks of code.

- `REPEAT n TIMES`: Executes the block a fixed number of times.

```
count <- 0
REPEAT 5 TIMES
{
    count <- count + 1
    DISPLAY(count) // Displays 1, 2, 3, 4, 5
}
```

Listing 3.8: AP Pseudocode: REPEAT n TIMES

- `REPEAT UNTIL (condition)`: Executes the block *first*, then checks the condition. Repeats as long as the condition is `false`. The block always executes at least once.

```
input <- ""
REPEAT UNTIL (input = "quit")
{
    DISPLAY("Enter command (or 'quit'):")
    input <- INPUT()
    // Process input...
    DISPLAY("You entered: " + input)
```

```
}
DISPLAY("Exiting loop.")
```

Listing 3.9: AP Pseudocode: REPEAT UNTIL

- `FOR EACH item IN list` (Covered more in Lists): Iterates through each element in a list sequentially.

```
names <- ["Alice", "Bob", "Charlie"]
FOR EACH name IN names
{
    DISPLAY("Hello, " + name)
}
```

Listing 3.10: AP Pseudocode: FOR EACH

These three structures (Sequence, Selection, Iteration) are the fundamental building blocks for constructing any algorithm.

3.4 Algorithms

An algorithm is a finite sequence of well-defined, computer-implementable instructions to solve a class of problems or perform a computation. Algorithms can be expressed in natural language, flowcharts, or pseudocode before being implemented in a programming language.

3.5 Procedures (Functions/Methods)

Procedures (often called functions or methods) are named blocks of code that perform a specific task. They help organize code, reduce redundancy, and improve reusability.

- **Definition**: Specifies the procedure name, parameters (inputs), and the code it executes.
- **Parameters**: Variables listed in the definition that receive values when the procedure is called.
- **Arguments**: The actual values passed to the parameters when the procedure is called.
- **Return Values**: Procedures can optionally return a value back to the calling code using the `RETURN` statement.
- **Modularity**: Breaking down a complex problem into smaller, manageable procedures.
- **Abstraction**: Hiding the complex details of implementation behind a simple interface (the procedure call).

```
// Procedure Definition
PROCEDURE calculateArea(length, width)
{
    area <- length * width
    RETURN(area)
}

```

```
// Procedure Call
roomLength <- 10
roomWidth <- 8
roomArea <- calculateArea(roomLength, roomWidth)
DISPLAY("The area is: ")
DISPLAY(roomArea)
```

Listing 3.11: AP Pseudocode: Procedure Definition and Call

When ‘calculateArea‘ is called, the value of ‘roomLength‘ (10) is passed as the argument for the ‘length‘ parameter, and ‘roomWidth‘ (8) is passed as the argument for the ‘width‘ parameter. Inside the procedure, these values are used to calculate ‘area‘. The ‘RETURN(area)‘ statement sends the calculated value (80) back to where the procedure was called. This returned value is then assigned to the ‘roomArea‘ variable.

3.6 Lists (Arrays)

Lists (or arrays in some languages) are ordered collections of items (elements). AP Pseudocode uses 1-based indexing.

- **Creating Lists**: `myList <- [item1, item2, item3]`
- **Accessing Elements**: By index. `firstItem <- myList[1]`, `thirdItem <- myList[3]`
- **Modifying Elements**: `myList[2] <- newItem`
- **Length**: Finding the number of elements. `len <- LENGTH(myList)`
- **Common Operations**: Adding items (`APPEND`), removing items (`REMOVE`), inserting items (`INSERT`).
- **Iteration with Lists**: Using `FOR EACH`.

```
scores <- [85, 92, 78, 95]
sum <- 0
FOR EACH score IN scores
{
    sum <- sum + score
}
average <- sum / LENGTH(scores)
DISPLAY(average)
```

Listing 3.12: AP Pseudocode: List Iteration

3.7 Algorithm Analysis

Comparing algorithms that solve the same problem.

- **Correctness**: Does the algorithm always produce the correct result?
- **Efficiency**: How many resources (primarily time, measured often by the number of steps relative to the input size) does the algorithm use, especially as the input size grows?
- **Readability/Simplicity**: Is the algorithm easy to understand and implement?

Efficiency is crucial when dealing with large inputs. We categorize algorithms based on how their runtime grows with the input size:

- **Reasonable Time**: Algorithms whose runtime grows polynomially (or slower) with the input size. These algorithms remain practical even for relatively large inputs. Examples include logarithmic, linear, and polynomial growth rates.
- **Unreasonable Time**: Algorithms whose runtime grows extremely fast (e.g., exponentially or factorially) with the input size. While correct, these algorithms quickly become impractical for all but the smallest inputs.

Common Algorithm Examples and Efficiency

Different algorithms solving the same problem can have vastly different efficiencies.

- **Linear Search**: Checks each element in a list sequentially until the target is found or the list ends.
  - **Efficiency**: Reasonable. In the worst case, it takes a number of steps roughly proportional to the size of the list (approximately n steps if the list has n items). Suitable for small or unsorted lists.
- **Binary Search**: Efficiently finds items in a *sorted* list by repeatedly dividing the search interval in half. It compares the target value to the middle element; if they don't match, it eliminates half the list where the item cannot be and repeats the process on the remaining half.
  - **Efficiency**: Reasonable. Much faster than linear search for large lists. The number of steps grows very slowly compared to the list size (proportional to $\log n$ steps for a list of size n). Requires the list to be sorted first.
- **Traveling Salesman Problem (TSP) - Brute Force**: The problem is to find the shortest possible route that visits each city in a given list exactly once and returns to the starting city. A brute-force algorithm checks every possible route.
  - **Efficiency**: Unreasonable. The number of possible routes grows factorially ($n!$) with the number of cities (n). This becomes computationally infeasible extremely quickly, even for a moderate number of cities (e.g., 20 cities have $20! \approx 2.4 \times 10^{18}$ possible routes).

Runtime Classification Table

Classification	Example Growth Rate	Description
Reasonable		
Logarithmic	$\log n$, $5 \times \log n$	Very efficient; runtime increases very slowly as input size grows (e.g., Binary Search).
Linear	n, $2n + 5$	Runtime grows directly proportional to input size (e.g., Linear Search).
Polynomial	n^2, $n^3 + n$, $10n^2$	Runtime grows faster but still manageable for moderately large inputs (e.g., many basic sorting algorithms).
Unreasonable		
Exponential	2^n, $1.5^n + n^2$	Runtime grows extremely rapidly; impractical for large inputs.
Factorial	$n!$	Runtime grows even faster than exponential; only feasible for very small inputs (e.g., Brute-force TSP).

Heuristics: Sometimes, finding an exact, optimal solution using a reasonable time algorithm is not known or practical (especially for problems where the best known exact solutions run in unreasonable time, like TSP). A **heuristic** is an approach that finds an *approximate* solution that is "good enough" for practical purposes, often much faster than finding the perfect solution. It's a shortcut or rule of thumb used when an exact solution is too hard or slow to find.

Undecidable Problem: A problem for which no algorithm can ever be constructed that always leads to a correct yes-or-no answer for every possible input (e.g., the Halting Problem - determining whether an arbitrary program will eventually stop running or continue forever).

3.8 Simulations

Simulations use computer models to mimic real-world phenomena or systems. They allow us to:

- Test hypotheses
- Explore complex systems where real-world experiments are impossible, dangerous, or expensive
- Make predictions

Simulations involve designing models with specific rules, variables, and interactions, often incorporating randomness (using random number generators) to reflect real-world variability.

4 Big Idea 4: Computer Systems and Networks

Computing requires hardware and software working together. Networks, especially the Internet, connect computers globally, enabling communication and information sharing.

4.1 The Internet

The Internet is a global network of interconnected computer networks. It's a system for communication and information exchange.

- **Decentralized**: No single point of control or failure.
- **Interoperability**: Uses standardized, open protocols (rules for communication) allowing different types of devices and networks to connect.
- **World Wide Web (WWW)**: A system of linked hypertext documents accessed via the Internet (using protocols like HTTP/HTTPS). The Web is *part* of the Internet, not the same thing. **Crucially, the Internet is the underlying network infrastructure (hardware and protocols), while the WWW is an information system built on top of the Internet.**

4.2 How the Internet Works

Data travels across the Internet using standardized procedures (protocols). Key concepts:

- **Protocols**: Standardized sets of rules governing communication. Open protocols (like those used on the Internet) allow different types of devices and networks to connect and interoperate smoothly.
- **IP Addresses**: Every device connected to the Internet needs a unique numerical label, its IP address (Internet Protocol address), to be identified and receive data.
- **Domain Name System (DNS)**: Since remembering numerical IP addresses is difficult, the DNS acts like the Internet's phonebook. It translates human-readable domain names (e.g., `www.google.com`) into the corresponding IP addresses needed for routing.
- **Packets**: Data sent over the Internet is broken down into small units called packets. Each packet contains a portion of the data plus metadata (like source/destination IP addresses) needed for routing and reassembly.
- **Routing**: Devices called routers examine packet destination addresses and forward them across networks towards their destination. The Internet is designed to be dynamic; packets from the same message might take different routes depending on network conditions and efficiency.
- **Packet Reliability**: Protocols like TCP (Transmission Control Protocol) ensure reliable data transfer by managing packet sequencing, checking for errors, and requesting retransmission of lost packets. Other protocols like UDP prioritize speed over guaranteed delivery for

applications like streaming. (You don't need to know the deep specifics of TCP/UDP, just that mechanisms exist for reliability vs. speed trade-offs).

- **World Wide Web (WWW) and HTTP/HTTPS**: The WWW is a system of linked information (web pages, files, etc.) accessed *via* the Internet. It uses the HTTP (Hypertext Transfer Protocol) or HTTPS (secure version) to request and display web content. The WWW is one of many services that run on the Internet, but it is not the Internet itself.
- **Scalability**: The Internet's design using open protocols, packet switching, and decentralized routing allows it to expand to handle ever-increasing numbers of users and devices effectively.

4.3 Fault Tolerance

Systems that can continue operating even if some components fail are fault-tolerant. The Internet achieves fault tolerance through:

- **Redundancy**: Multiple paths exist between networks. If one path or router fails, traffic can often be automatically redirected through alternative routes.
- **Packet Switching**: Because data is broken into independent packets, if some packets are lost or take a faulty route, they can often be re-sent or rerouted without losing the entire message (especially when using reliable protocols like TCP).

This design makes the Internet robust against many common failures, ensuring communication can continue.

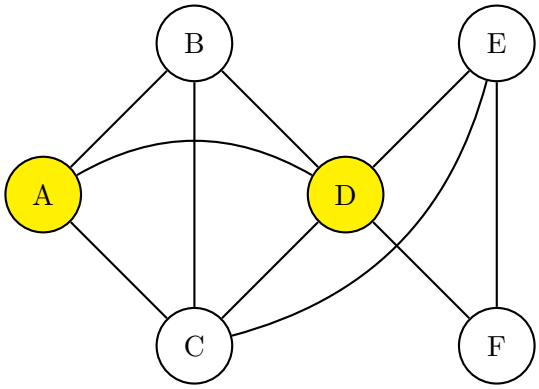


Figure 4.1: Network diagram illustrating redundancy and fault tolerance. Multiple paths exist between nodes (e.g., A to D directly, via B, or via C). If one path (e.g., A-B-D) fails, data can be rerouted through another (e.g., A-C-D or the direct A-D path), allowing the network to continue functioning.

4.4 Parallel and Distributed Computing

These approaches use multiple computers or processors to solve large problems faster.

- **Parallel Computing**: Uses multiple processors working simultaneously on parts of the same task, often within a single computer system.

- **Distributed Computing**: Spreads tasks across multiple, independent computers connected by a network (like the Internet). Examples include large-scale scientific simulations (e.g., protein folding) or big data processing.
- **Speedup**: The performance gain achieved by using parallel/distributed systems compared to a single processor. Ideally, N processors could provide N times speedup, but communication overhead and task dependencies often limit this.

4.5 Cybersecurity

Protecting computer systems, networks, and data from unauthorized access, attacks, damage, or theft.

- **Common Threats**:
  - **Malware**: Malicious software (viruses, worms, ransomware, spyware).
  - **Phishing**: Tricking users into revealing sensitive information (passwords, credit card numbers) often via deceptive emails, messages, or websites designed to look legitimate.
  - **Rogue Access Point**: An unauthorized wireless access point installed on a network, potentially allowing attackers to intercept traffic or gain unauthorized access.
  - **Keylogger**: Malware or hardware that records keystrokes entered on a keyboard, often used to steal passwords or other sensitive information.
- **Countermeasures**:
  - **Authentication**: Verifying identity (passwords, biometrics).
    - **Multifactor Authentication (MFA)**: Enhances security by requiring two or more distinct pieces of evidence (factors) to verify identity (e.g., something you know like a password + something you have like a code from your phone).
  - **Encryption**: Scrambling data (plaintext) into an unreadable format (ciphertext) using an algorithm and a key, so it's unintelligible without the key. Used for secure communication (HTTPS) and data storage.
    - **Symmetric Key Encryption**: Uses the **same secret key** for both encryption and decryption. Faster, but requires a secure way to share the secret key beforehand between the sender and receiver.
    - **Public Key (Asymmetric) Encryption**: Uses **two keys**: a *public key* (shared freely) for encryption and a corresponding *private key* (kept secret by the owner) for decryption. Anyone can encrypt a message using the recipient's public key, but only the recipient with the matching private key can decrypt it. This solves the key sharing problem of symmetric encryption but is computationally slower.
      - **Public Key**: Can be shared with anyone without compromising security. Used to encrypt messages intended for the owner of the corresponding private key.
      - **Private Key**: Must be kept secret by the owner. Used to decrypt messages encrypted with the corresponding public key.

Cybersecurity is an ongoing challenge due to evolving threats and system complexity.

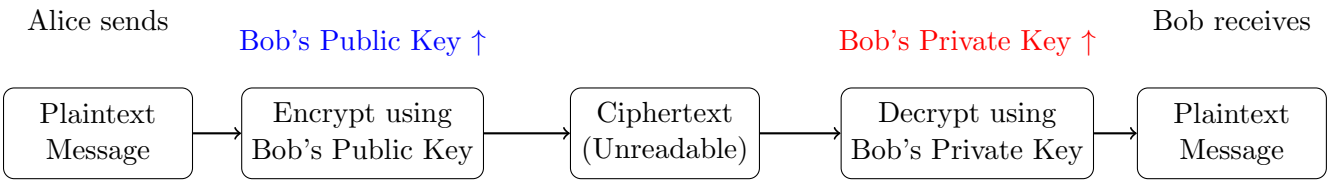


Figure 4.2: Public Key Encryption Example: Alice encrypts a message using Bob's public key. Only Bob, with his corresponding private key, can decrypt it.

5 Big Idea 5: Impact of Computing

Computing technologies have profoundly changed society, bringing numerous benefits but also raising significant ethical and societal challenges.

5.1 Beneficial Effects

Computing innovations have driven progress in many areas:

- **Communication**: Facilitating instant global communication (email, social media, video conferencing).
- **Access to Information**: Providing vast resources for learning and research (World Wide Web, online libraries, search engines).
- **Collaboration**: Enabling collaboration on projects regardless of geographic location.
- **Creativity and Expression**: Offering new tools for art, music, writing, and design.
- **Automation**: Performing repetitive or dangerous tasks, increasing efficiency in various industries.
- **Economic Growth**: Creating new industries, jobs, and markets.
- **Scientific Advancement**: Enabling complex simulations, data analysis, and discovery in fields like medicine, climate science, and physics.
- **Crowdsourcing / Citizen Science**: Leveraging the skills, knowledge, or computing power of large numbers of people, often via the internet, to solve complex problems, gather data, or contribute to research (e.g., protein folding projects, identifying galaxies, transcribing historical documents).
- **Accessibility**: Providing assistive technologies for people with disabilities.

5.2 Harmful Effects and Challenges

Alongside benefits, computing raises concerns:

- **Bias in Algorithms**: Algorithms, often trained on historical data, can reflect and amplify existing societal biases (e.g., in facial recognition, loan applications, hiring tools).
- **Privacy Concerns**: Widespread collection and analysis of personal data raise issues of surveillance, data breaches, and misuse of information.
- **Security Risks**: Increased reliance on digital systems makes individuals and infrastructure vulnerable to cyberattacks (as discussed in Big Idea 4).
- **Digital Divide**: Unequal access to computing devices, the internet, and digital literacy skills creates disparities in opportunities (economic, educational, social).
- **Workforce Impact**: Automation can displace workers in certain sectors, requiring workforce adaptation and retraining.

- **Information Reliability**: The ease of spreading information online makes it challenging to distinguish credible sources from misinformation and disinformation.
- **Intellectual Property**: Digital content (music, movies, software) is easily copied, leading to challenges in protecting copyright and preventing piracy.
- **Social Impacts**: Effects on social interaction, mental health (e.g., social media addiction), and the nature of community.

5.3 Bias in Computing

Bias can be introduced into computing systems at various stages:

- **Data Bias**: If the data used to train an algorithm is not representative of the population it will be used on, the algorithm may perform poorly or unfairly for certain groups.
- **Algorithm Bias**: The design choices made by programmers can inadvertently introduce bias.
- **Human Bias**: The people designing, implementing, and using technology can bring their own conscious or unconscious biases.

It's crucial to be aware of potential biases and strive to create and use technology equitably.

5.4 Legal and Ethical Concerns

Computing raises complex legal and ethical questions:

- **Copyright and Intellectual Property**: How to balance creators' rights with public access to information and innovation.
  - **Copyright**: The legal right granted to the creator of original works (literature, music, software, etc.), usually for a limited time. It gives the creator exclusive rights to copy, distribute, and adapt the work.
  - **Open Source Software (OSS)**: Software for which the original source code (the human-readable instructions written by programmers) is made freely available and may be redistributed and modified. Users can see how the software works, fix bugs, or adapt it. This contrasts with proprietary software, where the source code is kept secret. Examples include the Linux operating system and the Firefox web browser. OSS licenses still have rules, but they prioritize sharing and modification.
  - **Creative Commons (CC)**: A set of public copyright licenses that enable the free distribution of an otherwise copyrighted work. A creator uses a CC license to give others permission to share, use, and build upon their work, under certain conditions specified by the license (e.g., requiring attribution, prohibiting commercial use). This provides a more flexible alternative to the traditional "all rights reserved" copyright, facilitating sharing and collaboration, especially for digital content like images, music, and text.
  - **Fair Use**: A legal doctrine that permits limited use of copyrighted material without permission from the rights holders for purposes such as criticism, comment, news reporting, teaching, scholarship, or research.

- **Data Privacy Laws**: Regulations like GDPR (Europe) or CCPA (California) attempt to give individuals more control over their personal data.
- **Responsibility**: Who is responsible when an autonomous system (like a self-driving car) causes harm?
- **Ethical Use**: Considering the potential consequences of technology and using it responsibly.