1. Introduction of Programming

Syeda Jannatul Naim
Lecturer | Dept. of CSE
World University of Bangladesh (WUB)

January, 2025

Content

1. History of Programming

Programming is the process of instructing computers to perform specific tasks by writing sets of instructions called programs or code. It's a form of problem-solving where we translate human logic into a language computers understand.

1. Machine Language (1st Generation)

- Written in **binary (0s and 1s)**
- Very fast but **extremely difficult** for humans

Example:

```
10101010 00010101
```

2. Assembly Language (2nd Generation)

- Uses mnemonics like `ADD`, `MOV`
- Still machine-dependent

Example:

```css
MOV A, B
ADD A, C
```

3. High-Level Languages (3rd Generation)

- Easy to read and write
- Portable across machines

Examples:

- **C** (1972)
- FORTRAN
- COBOL
- Pascal

4. Object-Oriented Languages (4th Generation)

- Uses objects and classes
- Better for large software

Examples:

- C++
- Java
- Python

2. Procedure Oriented Programming and Object Oriented Programming

Structured/Procedural Programming

Philosophy: "Divide and conquer" - break problems into functions/procedures

Characteristics:

- Top-down approach
- Functions operate on data
- Global and local variables
- Examples: C, Pascal, FORTRAN

Object-Oriented Programming (OOP)

Philosophy: Model real-world objects with attributes and behaviors

Four Pillars:

1. **Encapsulation:** Bundling data and methods
2. **Inheritance:** Creating new classes from existing ones
3. **Polymorphism:** Same interface, different implementations
4. **Abstraction:** Hiding complex implementation

2. Procedure Oriented Programming and Object Oriented Programming

Aspect	Procedural	Object-Oriented
Approach	Top-down	Bottom-up
Focus	Functions/Procedures	Objects/Classes
Data Security	Less secure (global data)	More secure (encapsulation)
Reusability	Function libraries	Class inheritance
Best For	Small projects, system programming	Large projects, GUI applications
Example	Operating systems, utilities	Enterprise software, games

3. Program Development Process

a. **Stage-1: program design**
   i. Problem analysis
   ii. Outline the program structure
   iii. Algorithm development
   iv. Selection of control structures
b. **Stage-2: program coding**
c. **Stage-3: program testing and debugging**
   i. Types of errors
   ii. Program testing: unit and integration testing
   iii. Program debugging
d. **Stage-4: Program Efficiency (during algorithm development)**
   i. Execution time (time complexity)
   ii. Memory requirements (space complexity)

Stage-1: Program Design

1. Problem Analysis

Understand:

- What is input?
- What is output?
- Constraints?

Example

> Find the sum of N numbers
> Input: N numbers
> Output: Sum

2. Outline the Program Structure

Break problem into parts (functions)

Example:

- Input numbers
- Calculate sum
- Display result

3. Algorithm Development

Step-by-step solution

Algorithm: Sum of N numbers

1. Start
2. Read N
3. Initialize sum = 0
4. Repeat N times:
   - Read number
   - sum = sum + number
5. Print sum
6. End

4. Selection of Control Structures

- **Sequence**: step by step
- **Selection**: `if`, `switch`
- **Iteration**: `for`, `while`

Stage-2: Program Coding

Convert algorithm into code:

```c
#include <stdio.h>

int main() {
    int n, sum = 0, x;
    scanf("%d", &n);

    for(int i = 0; i < n; i++) {
        scanf("%d", &x);
        sum += x;
    }
    printf("Sum = %d", sum);
    return 0;
}
```

Stage-3: Program Testing and Debugging

1. Types of Errors

(a) Syntax Errors

- Grammar mistakes

```c
printf("Hello")   // missing semicolon
```

(b) Runtime Errors

- Occur during execution

```c
int x = 5 / 0;    // divide by zero
```

(c) Logical Errors

- Wrong logic, wrong output

```c
avg = sum * n;    // should be sum / n
```

2. Program Testing

Unit Testing

- Test each function separately

Integration Testing

- Test combined modules

3. Program Debugging

Finding and fixing errors.

Methods:

- Print variable values
- Step-by-step execution (debugger)
- Remove errors one by one

Stage-4: Program Efficiency (during algorithm development)

- After designing an algorithm, we must check **how efficient it is**. Efficient programs:
  - Run **faster,** Use **less memory,** Scale well for **large inputs**
- Program efficiency mainly depends on:
  1. Execution Time (Time Complexity)
  2. Memory Requirements (Space Complexity)

What is Time Complexity?

Time complexity measures **how the running time of an algorithm grows** as the input size increases.

- It does **NOT** measure actual time in seconds.
- It measures **number of operations** performed.
- We use **Big-O notation** to represent time complexity.

What is Space Complexity?

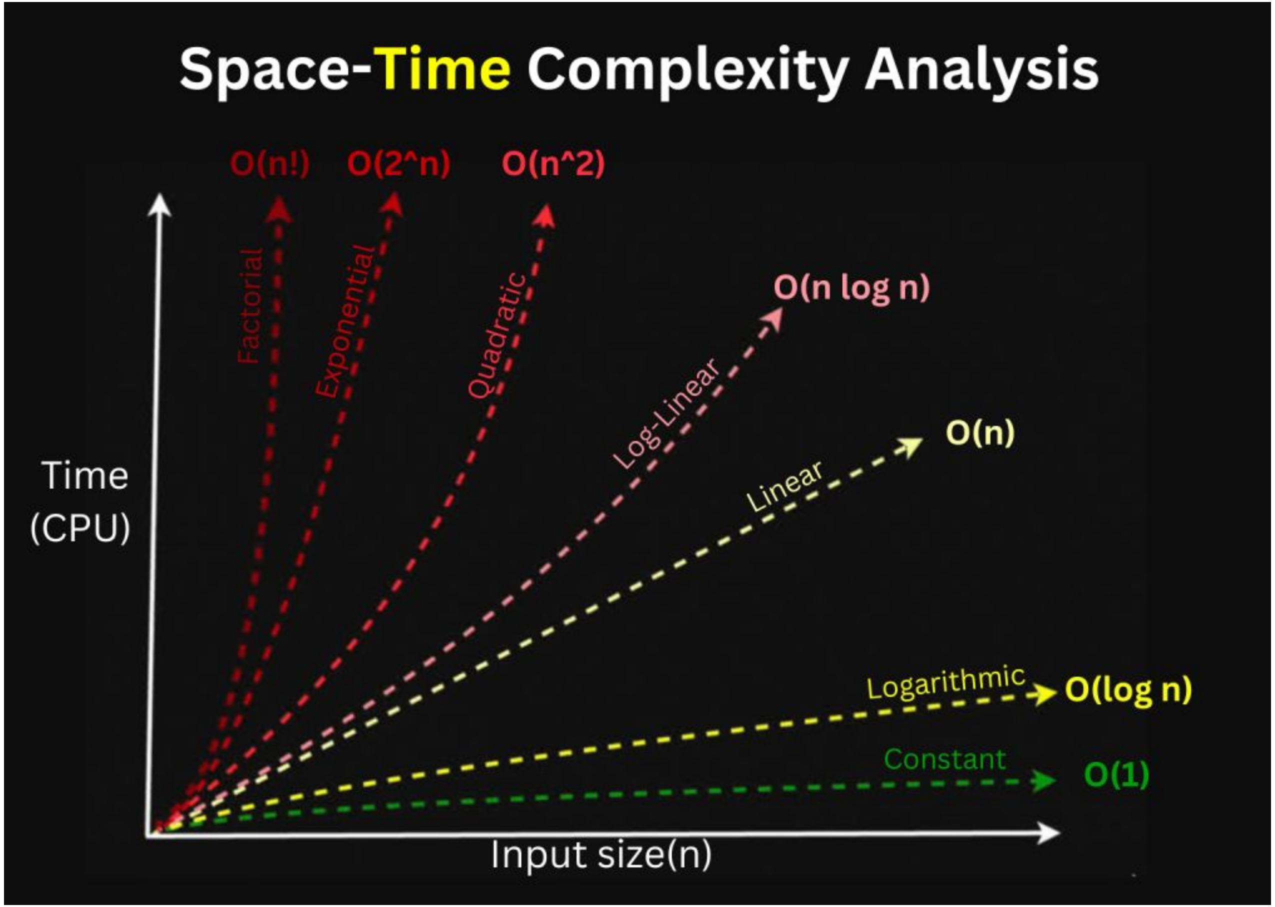
Space complexity measures the **amount of extra memory** used by an algorithm. Includes:

- Variables
- Arrays
- Dynamic memory
- Recursive stack

Stage-4: Program Efficiency (during algorithm development)

Complexity	Name	Operations for n=1000	Growth Rate
O(1)	Constant	1	Excellent
O(log n)	Logarithmic	~10	Excellent
O(n)	Linear	1,000	Good
O(n log n)	Linearithmic	~10,000	Fair
$O(n^2)$	Quadratic	1,000,000	Poor
$O(n^3)$	Cubic	1,000,000,000	Very Poor
O(2^n)	Exponential	1.07×10^{301}	Terrible
O(n!)	Factorial	4×10^{2567}	Catastrophic

Stage-4: Program Efficiency (during algorithm development)



Stage-4: Program Efficiency (during algorithm development)

- **Time Limit Exceeded (TLE)**

TLE (Time Limit Exceeded) means our program **did not finish within the allowed time** set by the system (online judge, compiler, or runtime environment). The logic may be correct, but the algorithm is **too slow**.

- **Why TLE Happens:**

1. Inefficient time complexity
   Using $O(n^2)$ or $O(2^n)$ when n is large.
2. Unnecessary nested loops
3. Repeated calculations
4. Slow input/output
5. Infinite or very long loops
6. Using recursion without optimization
7. Brute-force approach instead of optimized algorithms

Stage-4: Program Efficiency (during algorithm development)

- **Operations Per Second (Approximate):**
  - **Modern CPU:** 1 to 5 × 10^8 operations/second
  - **Contest/Online Judge:** Usually 10^7-10^8 operations/second
    - 1 second limit: ~10^8 operations
    - 2 second limit: ~$2×10^8$ operations
    - 5 second limit: ~$5×10^8$ operations
  - **TLE on Languages: C/C++** < **Java** < **Python** = **1** : **2** : **4**
  - **How to Avoid TLE:**
    - Analyze **time complexity before coding**
    - Avoid nested loops where possible
    - Use **hashing**, **binary search**, **two pointers**
    - Replace recursion with DP or memoization
    - Use **fast I/O** (scanf/printf in C/C++, buffered I/O)
    - Break early when condition is met
    - Precompute results if reused

4. History of C language

1969-1971: Predecessors

- **BCPL** (Basic Combined Programming Language) by Martin Richards
- **B Language** by Ken Thompson (simplified BCPL)
- Used for early UNIX development

1972: Birth of C

- **Dennis Ritchie** at Bell Labs creates C
- Combined features of BCPL and B
- Added data types and structures
- Used to rewrite UNIX kernel

1978: K&R C

- "The C Programming Language" book by Brian Kernighan and Dennis Ritchie
- Became the de facto standard

1989: ANSI C (C89)

- First standardized version by ANSI
- Added function prototypes, void pointers

1990: ISO C (C90)

- International standardization
- Same as ANSI C89

1999: C99 Standard

- Added `//` comments, `inline` functions
- Variable-length arrays
- `long long` data type
- Designated initializers

2011: C11 Standard

- Multi-threading support
- Anonymous structures/unions
- Bounds-checking functions

2017: C17 Standard

- Bug fixes and clarifications
- No major new features

4. History of C language

2023: C23 Standard (Latest)

- Enhanced Unicode support
- `#embed` preprocessor directive
- `constexpr`-like functionality

Key Contributors:

- **Dennis Ritchie** - Creator of C
- **Ken Thompson** - Creator of B language and UNIX
- **Brian Kernighan** - Co-author of K&R book
- **ANSI/ISO Committees** - Standardization

5. Characteristics of C language

1. Mid-Level Language
   - High-level features (functions, control structures)
   - Low-level capabilities (pointer arithmetic, memory access)
2. Procedural Language
   - Programs are collections of functions
   - Top-down execution
3. Structured Language
   - Code organized in logical blocks
   - Supports loops and decision-making
4. Portable
   - Write once, compile anywhere
   - Standard library ensures consistency
5. Fast and Efficient
   - Compiles to efficient machine code
   - Minimal runtime overhead
6. Rich Operator Set
   - Arithmetic, logical, bitwise, relational operators
   - Pointer operators (`*`, `&`)
7. Memory Management
   - Direct memory access via pointers
   - Manual memory allocation/deallocation
8. Extensible
   - Can create your own libraries
   - Interface with assembly code

5. Importance of C language

1. Foundation of Modern Computing
   - UNIX/Linux written in C
   - Windows kernel has C components
   - Database systems (Oracle, MySQL)
2. System Programming
   - Operating systems
   - Device drivers
   - Compilers and interpreters
3. Embedded Systems
   - Microcontrollers
   - IoT devices
   - Automotive systems
4. Educational Value
   - Teaches fundamental concepts
   - Understanding of memory management
   - Basis for learning C++, Java, C#
5. Performance-Critical Applications
   - Game engines
   - Graphics programming
   - Scientific computing
6. Influence on Other Languages
   - C++: "C with classes"
   - Java: Similar syntax
   - Python: Implemented in C
   - JavaScript: First engines in C

6. WHERE TO USE C

1. **System Programming**
2. **Operating Systems**

- Linux kernel: ~15 million lines of C
- Windows NT kernel components
- Real-time operating systems (RTOS)

3. **Embedded Systems**

4. **Compilers & Interpreters**

- GCC (C compiler written in C)
- Python interpreter (CPython)
- Lua interpreter

5. **Performance-Critical Applications**

- Game physics engines
- Financial trading systems
- Image/video processing

6. WHERE NOT TO USE C

1. **Rapid application development**
2. **Web application**
3. **GUI application**
4. **Large enterprise applications**
   a. Better alternatives: Java, C#, Python
   b. Built-in frameworks for:
      i. Database connectivity
      ii. Web services
      iii. Security
      iv. Distributed computing
5. **Data Science and AI based applications**

END

of

Chapter 1

Reference: E. Balaguruswamy; Programming in ANSI C; *chapter-1, chapter-15;*