Lecture 1

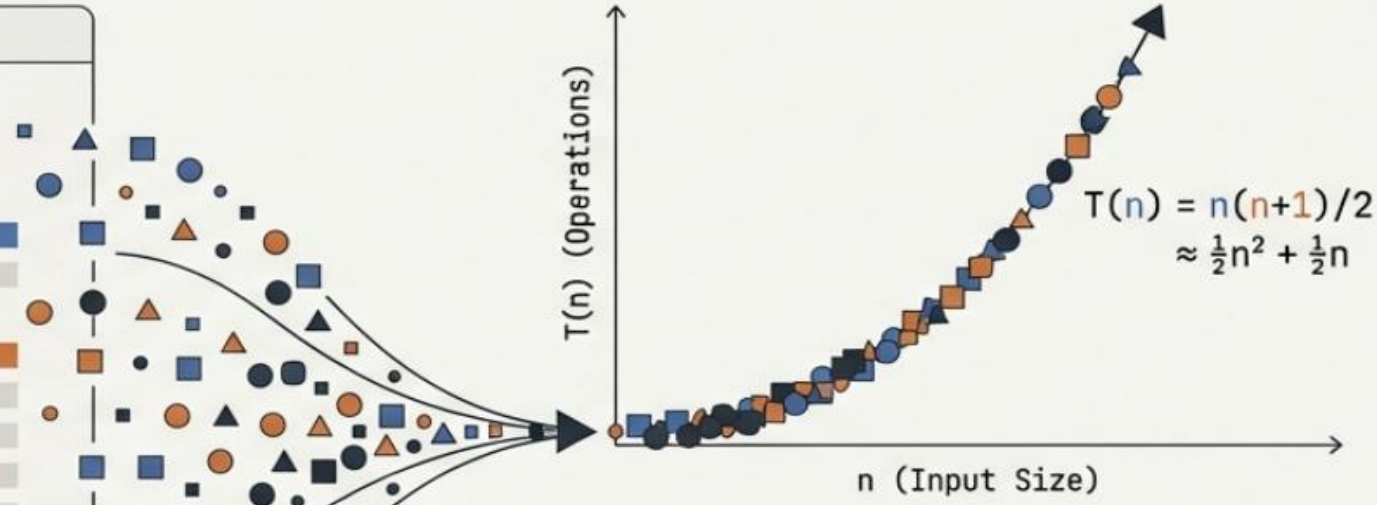
ALGORITHM ANALYSIS: FROM CONCEPT TO CALCULATION

A comprehensive guide to T(n), Time Complexity, and Summation Equations.

Code Implementation

```
// Outer loop (Slate Blue)
for (i = 1; i <= n; i++) { // #4B6EAF
    // Inner loop (Burnt Orange)
    for (j = 1; j <= i; j++) { // #D97D3E
        // Core operation
        executeOperation();
    }

    /* #4B8B3B
     * Comment: This represents the nested structure
     * contributing to the total time complexity.
     * */
}
```



$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n$$

Visualization of the transformation from nested loops to summation equations and the resulting polynomial time complexity function.

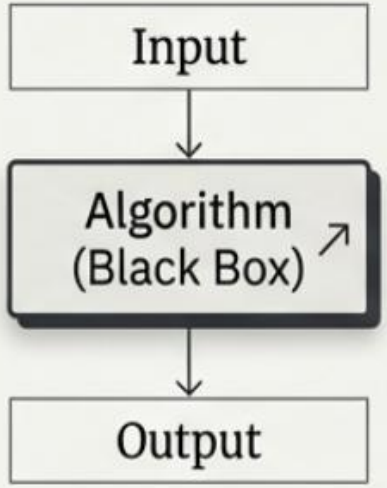
The Core Conflict: Algorithm vs. Program

The Workflow

The Problem (Task): Defined strictly by Inputs and Outputs.

The Algorithm (The "Thought"): The logical steps to transform Input to Output. This is hidden from the client.

The Program (The Product): An instance of the algorithm written in a specific language (C++, Java) running on specific hardware.



The Scenario

A **Client** requests a solution (e.g., "Sum two numbers").

A **Programmer** can devise multiple Algorithms (**Algo 1, Algo 2, Algo 3**) to solve the same problem.

The Dilemma: The client wants the "Best" version.

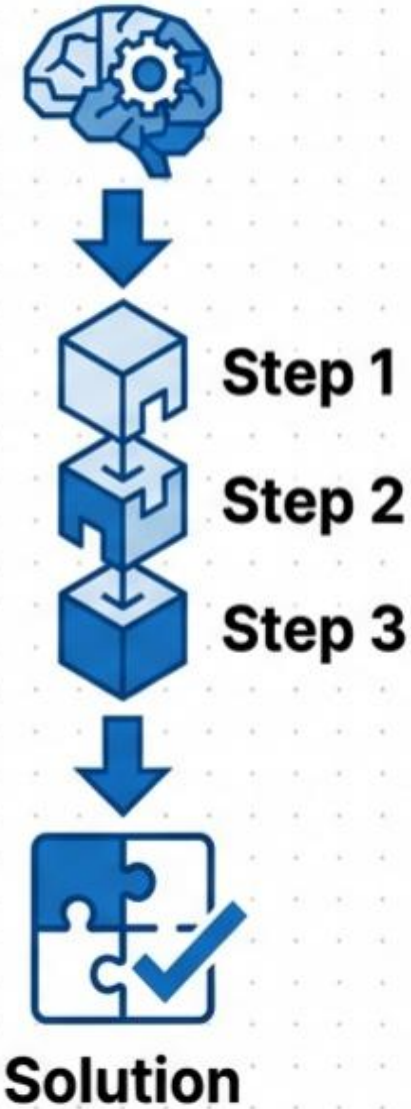
Selection Criteria: High Efficiency (Speed) and Low Resource Usage (Space).

Key Takeaway: We analyze the Algorithm to predict the performance of the Program.

An Algorithm is a Method of Thinking

Core Definition:
An algorithm is not code; it is the process, operation, or logical path used to solve a problem.

- A set of logical, ordered steps.
- A universal recipe independent of programming language.
- The **mental blueprint** created before coding.



Defining the Problem in Computer Science

In CS, a ‘problem’ is a Task to be executed.

INPUT

The Givens

THE ALGORITHM / PROCESS

The Gap: Logic required to transform data

OUTPUT

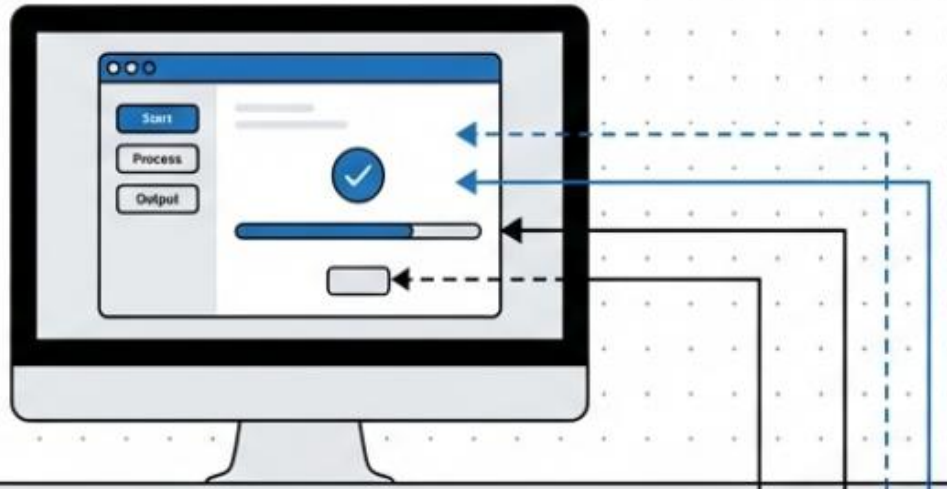
The Solution

Concept vs. Implementation

Understanding the difference between the abstract logic and the final product.

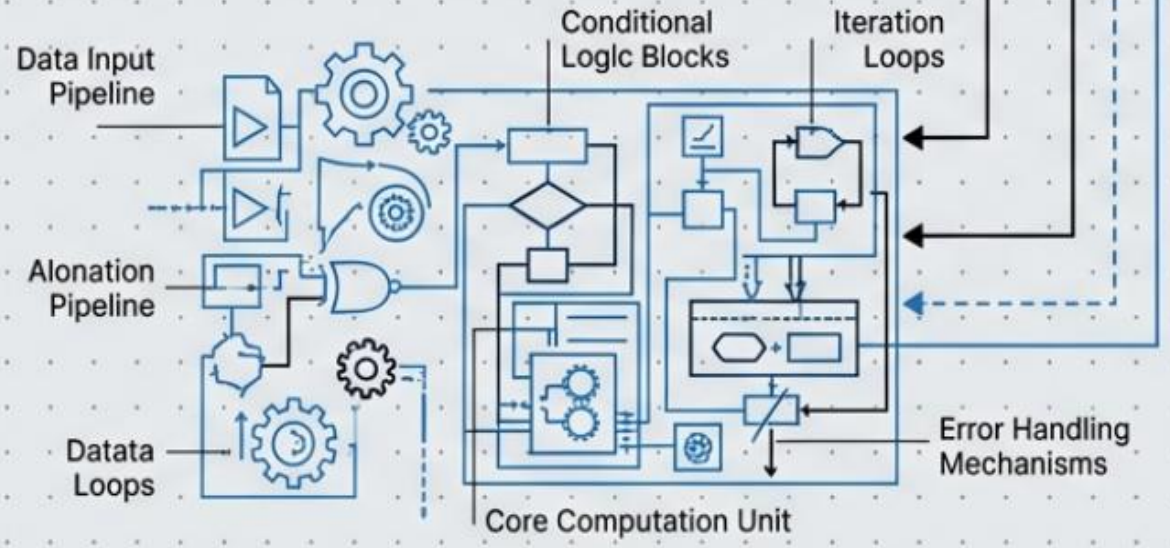
The Client Sees:
THE PROGRAM

An instance of the algorithm written in a specific language (C++, Java, Python).

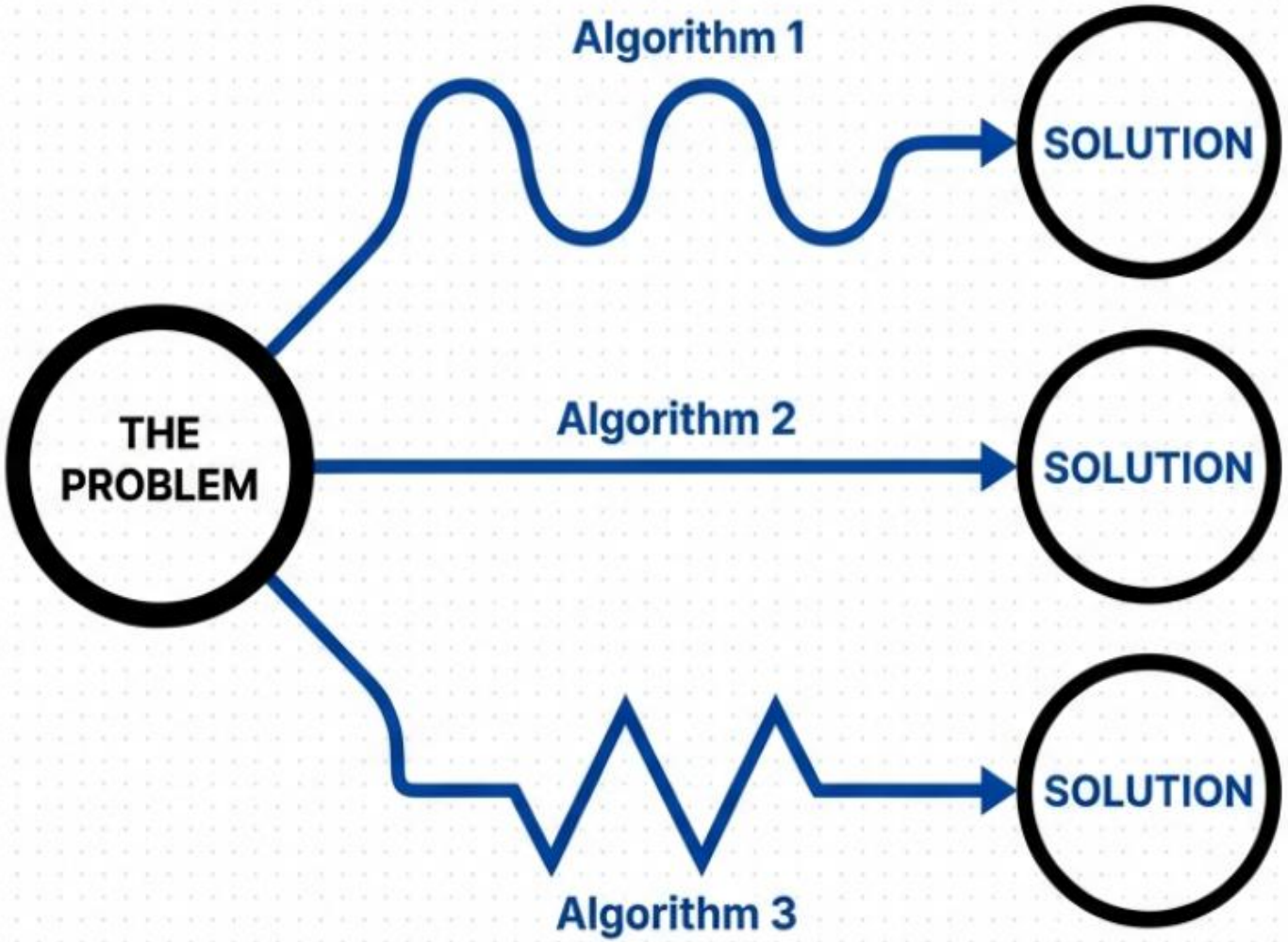


The Programmer Sees:
THE ALGORITHM

The hidden logic and thought process. Universal and hardware-independent.



The Reality of Multiple Solutions

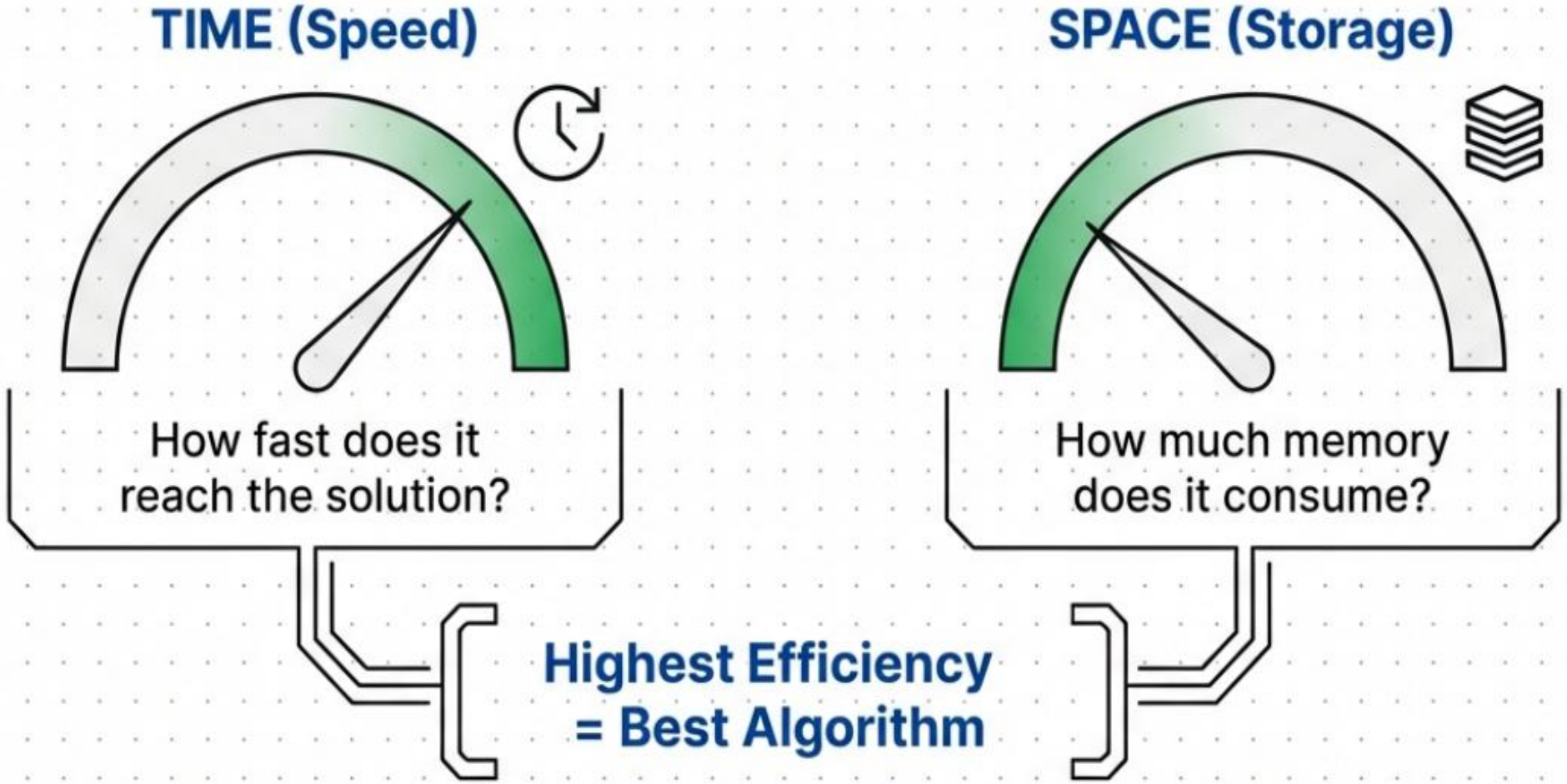


The Conflict:

Since logic is subjective, multiple valid algorithms exist for any problem. If Alg 1, Alg 2, and Alg 3 all produce the correct output, how do we choose?

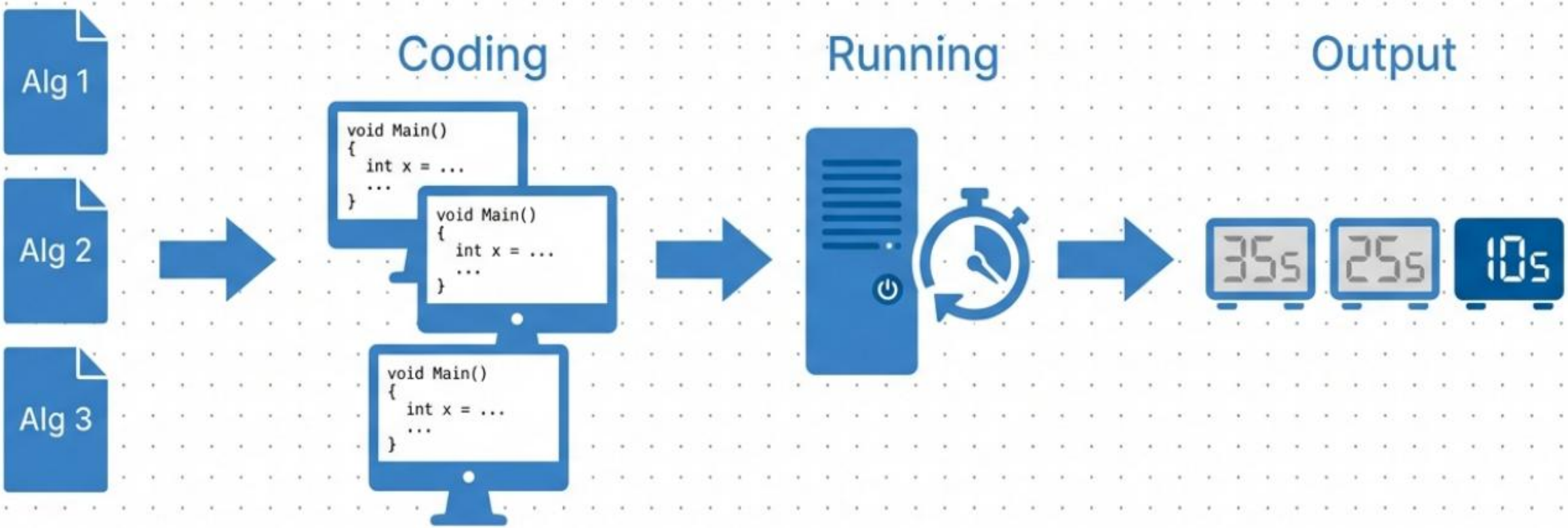
Criteria for Selection: Efficiency

We seek the candidate that delivers the fastest speed with the lowest memory usage.



The Old Method: Empirical Testing

"Try them all and see who wins."

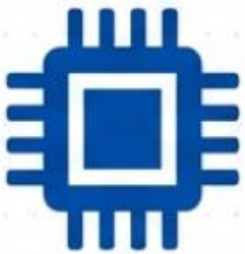


The Constraints of Empirical Testing

To **compare fairly**, variables must be rigorously controlled.

Same Programming Language

Cannot compare C++ to Python. Syntax affects speed.



Same Hardware

Cannot compare a supercomputer to a laptop. Hardware specs affect speed.

Exact Measurements

Results are specific to that exact moment and machine (e.g., Alg 1 = 35s).

The Solution: Algorithm Analysis

A theoretical method to evaluate the Time and Space complexity of an algorithm.

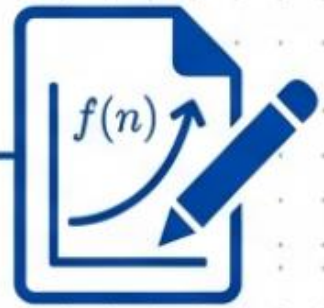
Physical Measurement

Physical Measurement

Results are tied to specific machines. An algorithm might perform differently on different architectures.

VS

Theoretical Estimation

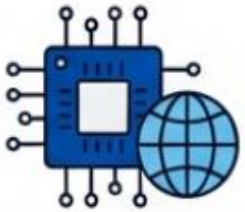
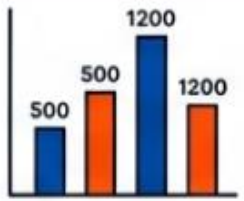
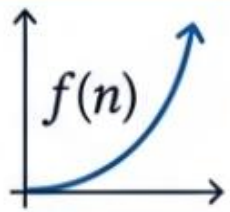


Theoretical Estimation

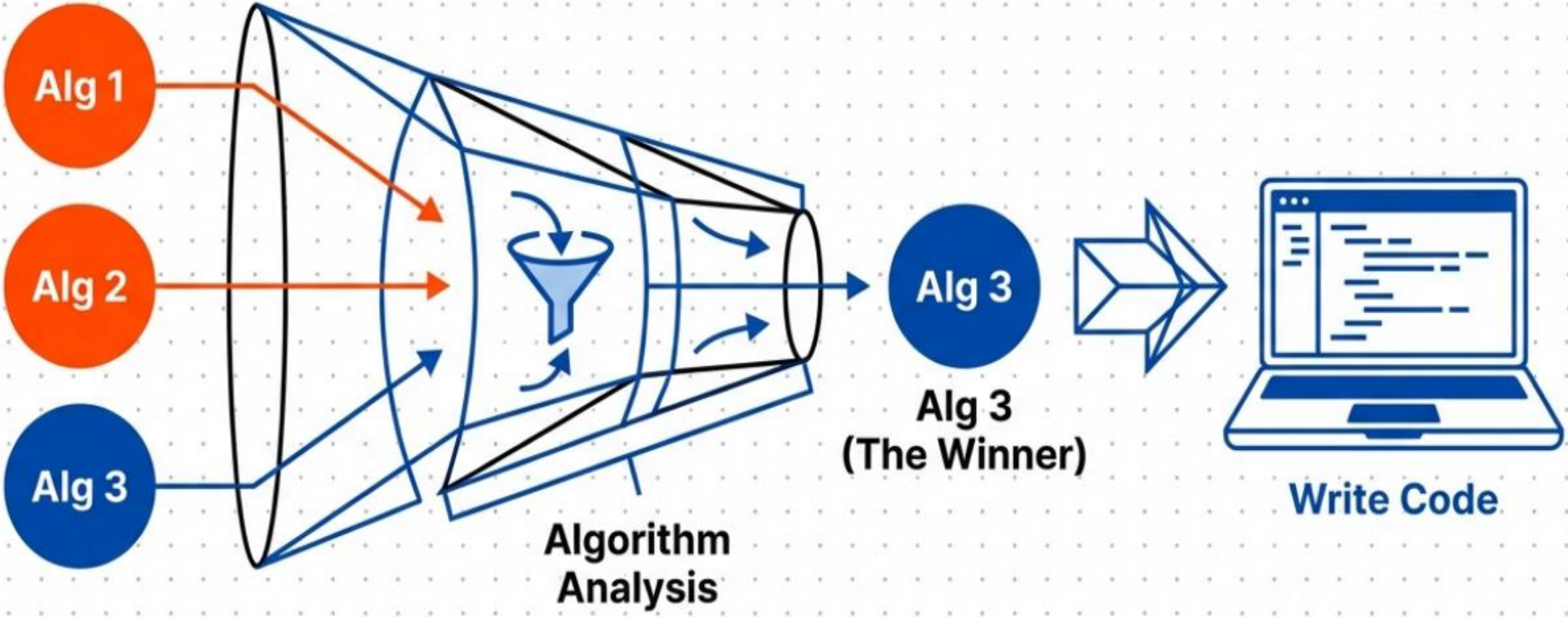
A way to judge algorithms BEFORE writing code.

No Coding Required. No Hardware Required.

Empirical Testing vs. Algorithm Analysis

Empirical (Old Way)		Algorithm Analysis (New Way)	
	Requires full coding of all solutions.		No coding required.
	Hardware Dependent results.		Hardware Independent.
	Wastes Development Time.		Saves Development Time.
	Results are isolated/exact numbers.		Results are predictive functions.

“Optimizing the Development Lifecycle”



Result: We only code the best solution.

The Next Step

We know we need to measure "**Time**" theoretically. But how do we count Time without a **stopwatch?**

Coming Up:
Mathematical Techniques for Time Complexity

The Detailed Method: Counting Units of Time

Since hardware speed varies, we count Operations, not seconds.

The Unit of Time Rule

A **single statement** executed once = **1 Unit.**

- Assignment (x=1)
- Arithmetic (i+j)
- Print (print x)

```
int i = 1;   ⟶ 1 Unit
int j = 2;   ⟶ 1 Unit
x = i + j;   ⟶ 1 Unit
print x;     ⟶ 1 Unit
```

Total = 1 + 1 + 1 + 1 = 5 Units

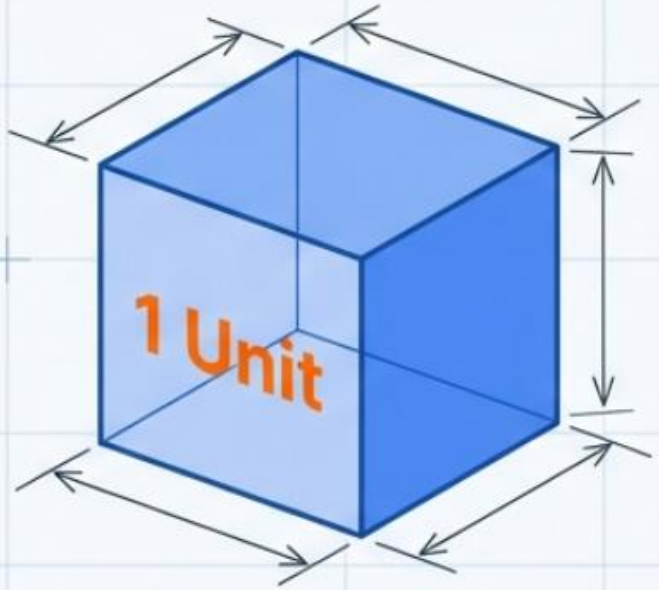
T(n) is an equation representing total runtime relative to input size (n).

Physical time is an unreliable metric.

The Problem: Hardware Dependency

- Supercomputer: 0.0001 seconds
- Laptop: 0.05 seconds
- Result: Inconsistent Data

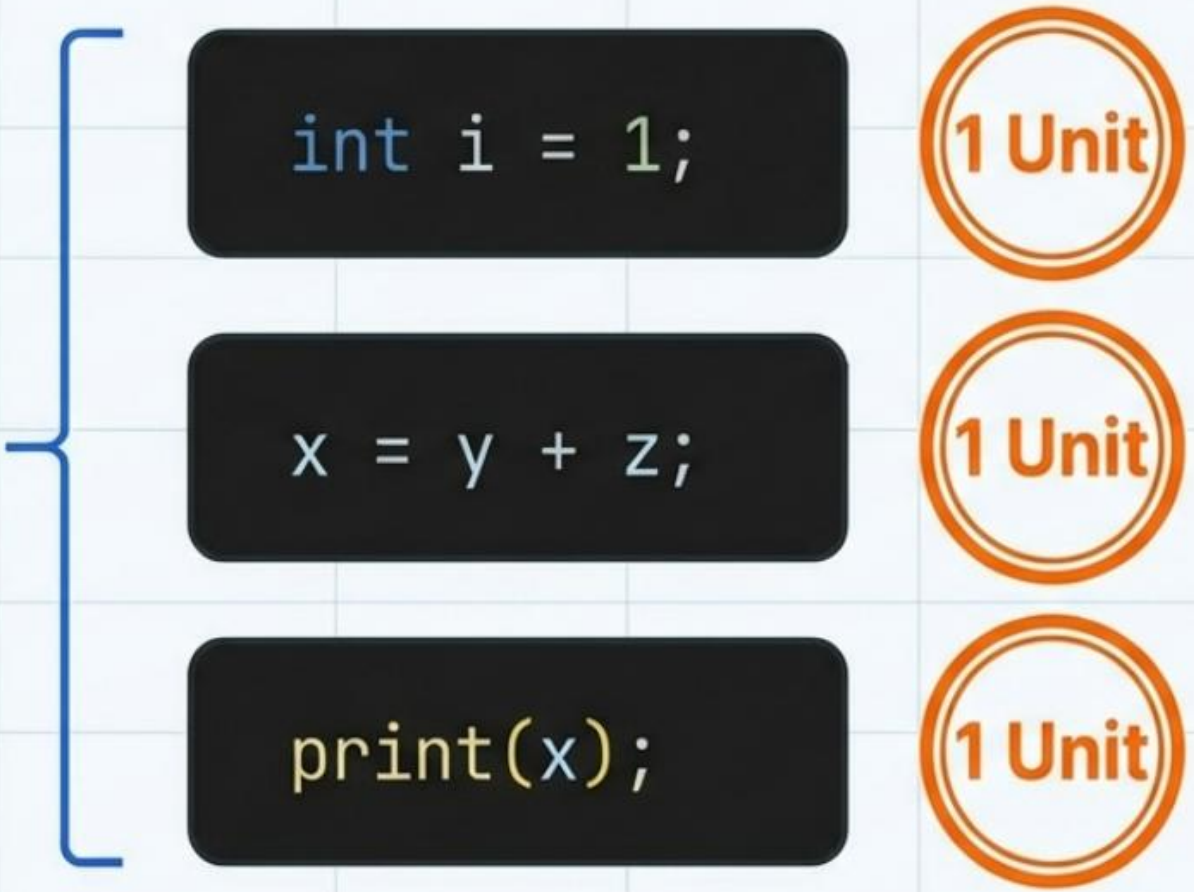
The Solution: Theoretical Units



- Independent of hardware speed
- Measures steps, not seconds
- Definition: 1 Unit = 1 Executed Statement

The Fundamental Rule: One Statement = One Unit

If executed once, the cost is **constant.**



Assumption: An algorithm must have at least one step. 0 steps = No Algorithm.

Case Study A: Calculating Linear Execution

```
1. int i = 1;
2. int j = 2;
3. int x = i + j;
4. print(x);
```

Line 1 → 1 Unit

Line 2 → 1 Unit

Line 3 → 1 Unit

Line 4 → 1 Unit

Total Time = 4 Units

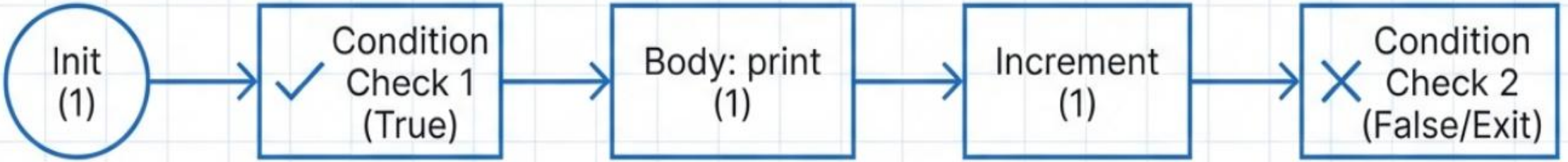
Sequential code implies Constant Time.

Case Study B: The Fixed Loop

Tracing a loop that runs exactly 1 time.

```
for(i=1; i<=1; i++) {
  print(i);
}
```

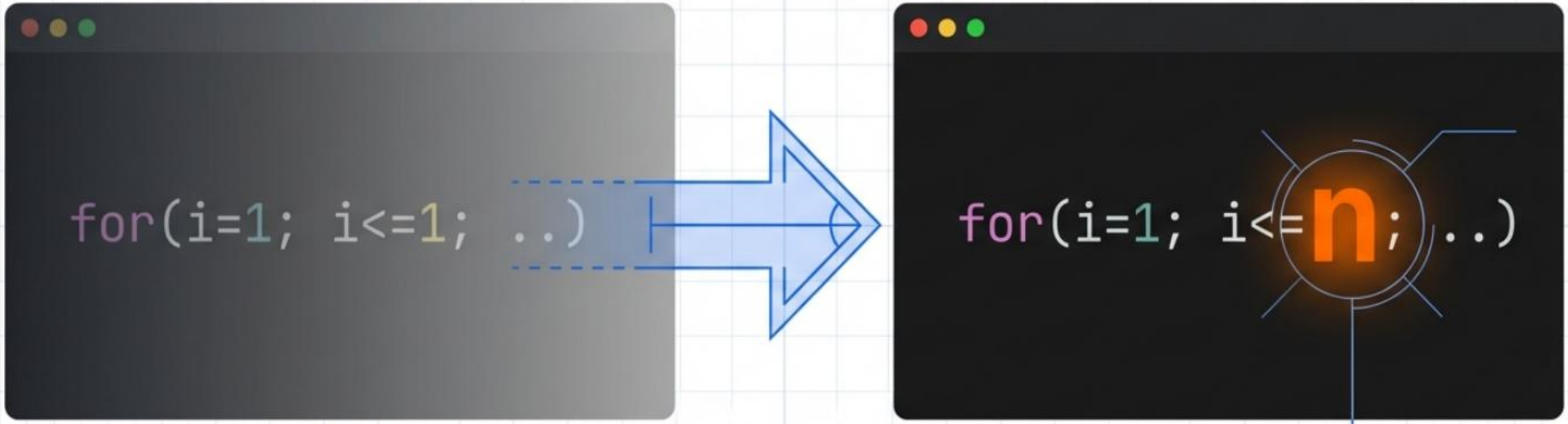
RUN 1 TIME FOR TRUE CHECK + 1 EXTRA CHECK THAT FAILS TO EXIT



1 + 1 + 1 + 1 = 5 Units

(Note: Condition accounts for 2 of these units).

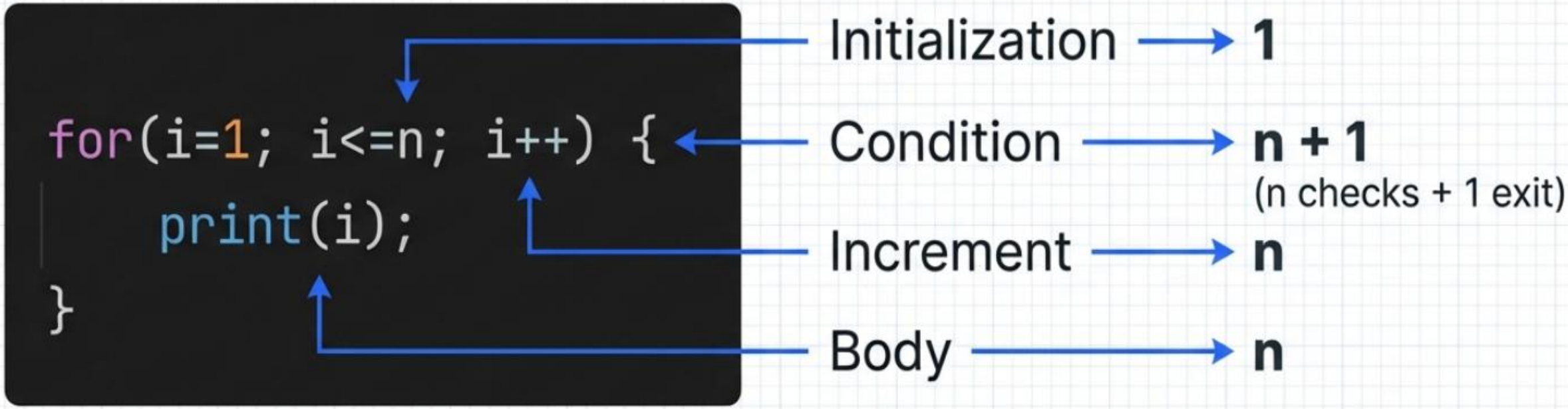
Enter the Unknown: The Variable N



N = Size of the Problem

n → Input determined at runtime (e.g., 1, 100, 1,000,000).

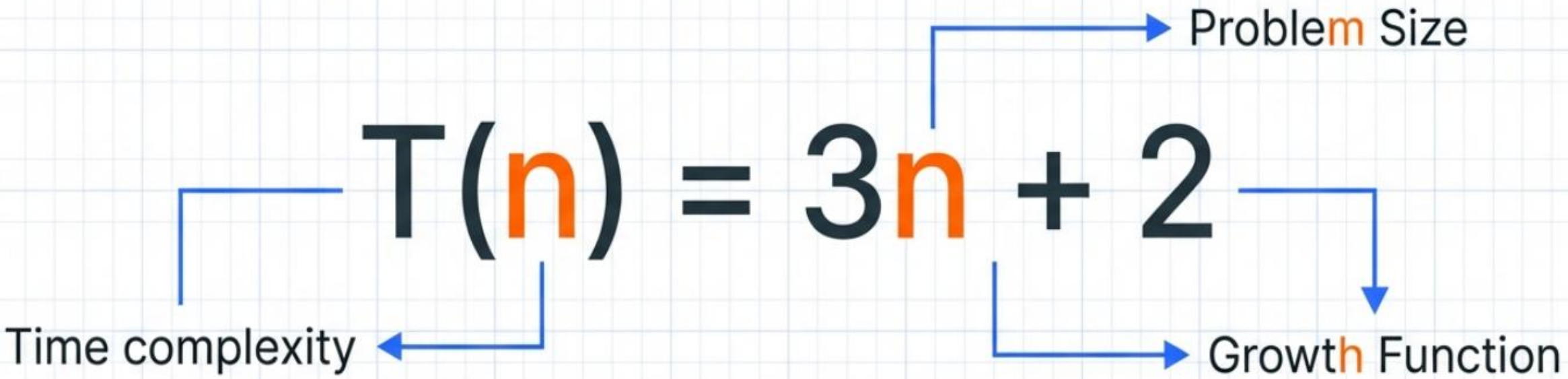
Deriving the Linear Equation



Sum = 1 + (**n** + 1) + **n** + **n**

Total = 3n + 2

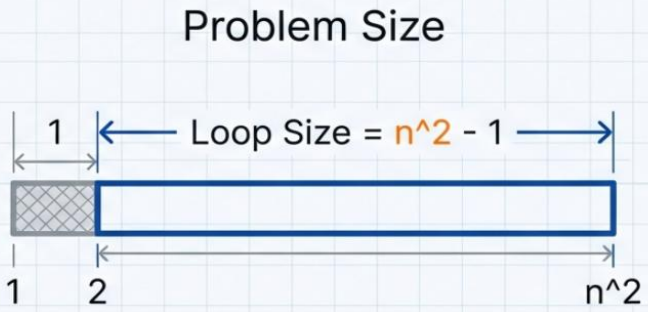
Defining the Function T(n)



We can now predict performance for **ANY** input size without running the code.

Case Study C: Complex Bounds

```
for(i=2; i <= n^2; i++) {
    print(i);
    print(i);
}
```



Bound Limit: **n^2**
Start Offset: **-1** (Starts at 2)

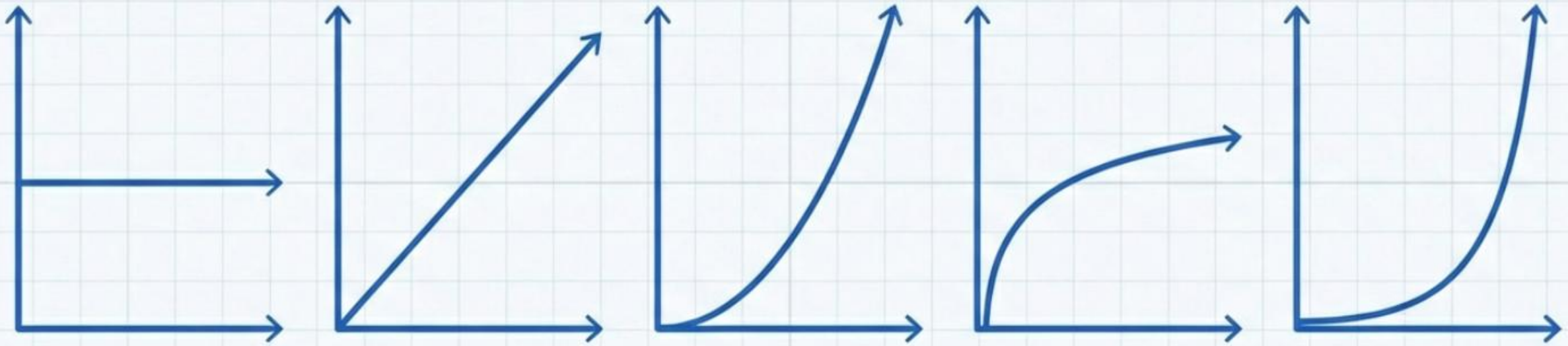
Solving the Quadratic Equation

Init: 1
Condition: (n^2 - 1) + 1 = n^2
Increment: n^2 - 1
Body (2 lines): 2 * (n^2 - 1)

Sum: 1 + n^2 + (n2 - 1) + (2n2 - 2)
Simplify: 1 + n^2 + n^2 - 1 + 2n^2 - 2

Final Result: T(n) = 4n^2 - 2

The Spectrum of Complexity



Constant
$T(n) = 1$

Linear
$T(n) = an + b$

Quadratic
$T(n) = an^2 + bn + c$

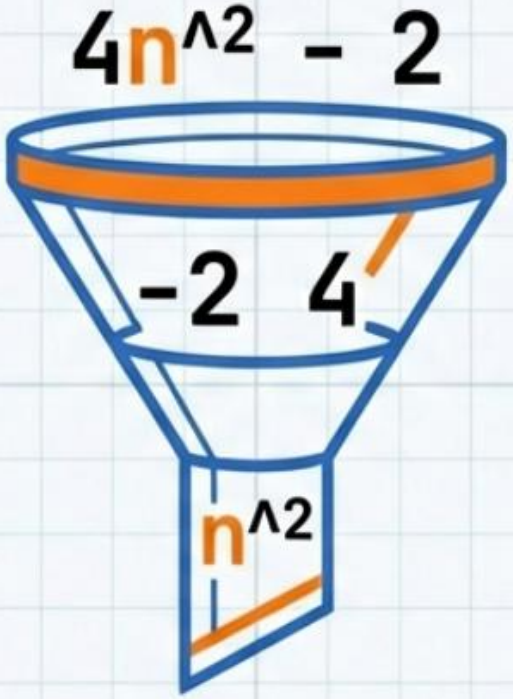
Logarithmic
$T(n) = a \log n + b$

Exponential
$T(n) = 2^n$

Precision vs. Significance

$4n^2 - 2$

Precise Answer



$>$

n^2

Dominant Term

As N approaches infinity, constants (-2) and multipliers (4) become irrelevant to the growth curve. We calculate precisely to learn mechanics. We analyze generally to understand scale.

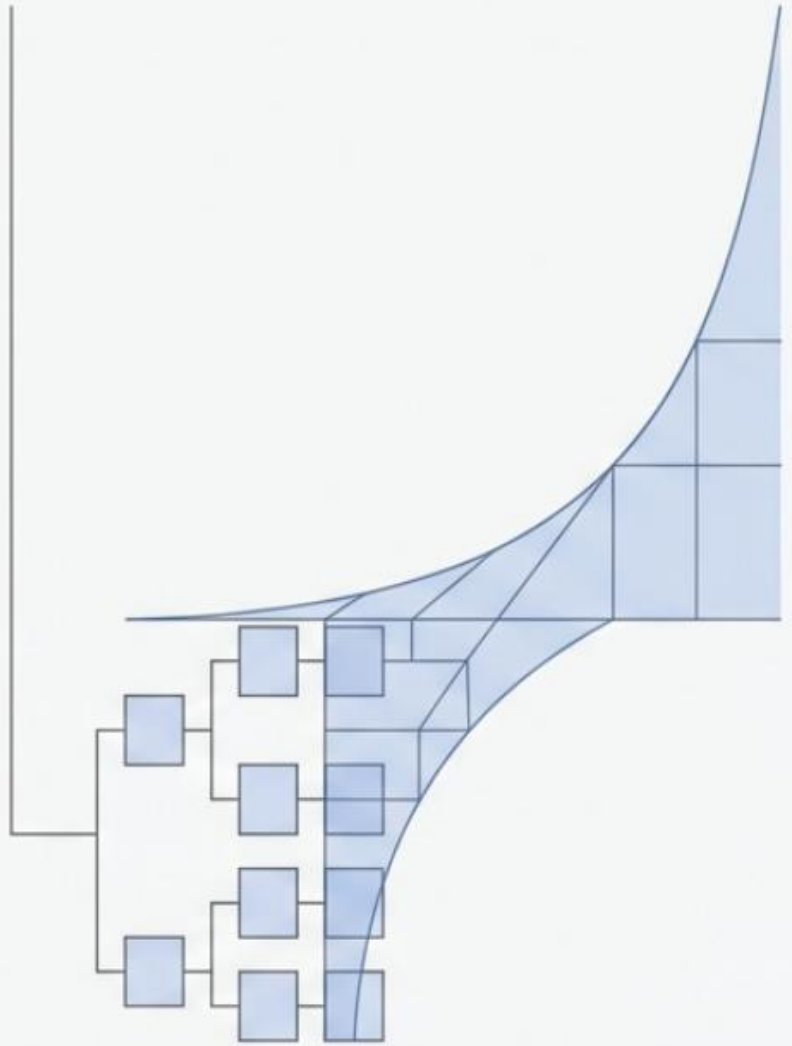
The Algorithm Analysis Cheatsheet

Component	Logic	Cost (Units)
Simple Statement	Executed once	1
Loop Initialization	Runs once at start	1
Loop Condition	Runs for **size** + exit	**Size** + 1
Loop Increment	Runs for **size**	**Size**
Loop Body	Runs for **size** * lines	**Size** * **(statements)**

T(n) = Sum of all steps

Algorithm Analysis: Calculating T(n) Using the Simple Method

From Precise Counting to Asymptotic Estimation



Escalating Complexity Series | Part 1

The Shift in Perspective

The Precise Method

```
1 for (int i = 0; i < n; i++) {
2     sum += i;
3     printf("Value: %d\n", sum);
4 }
```

$$T(n) = 3n + 2$$

Counting every statement.
Exact but tedious.

The Simple Method

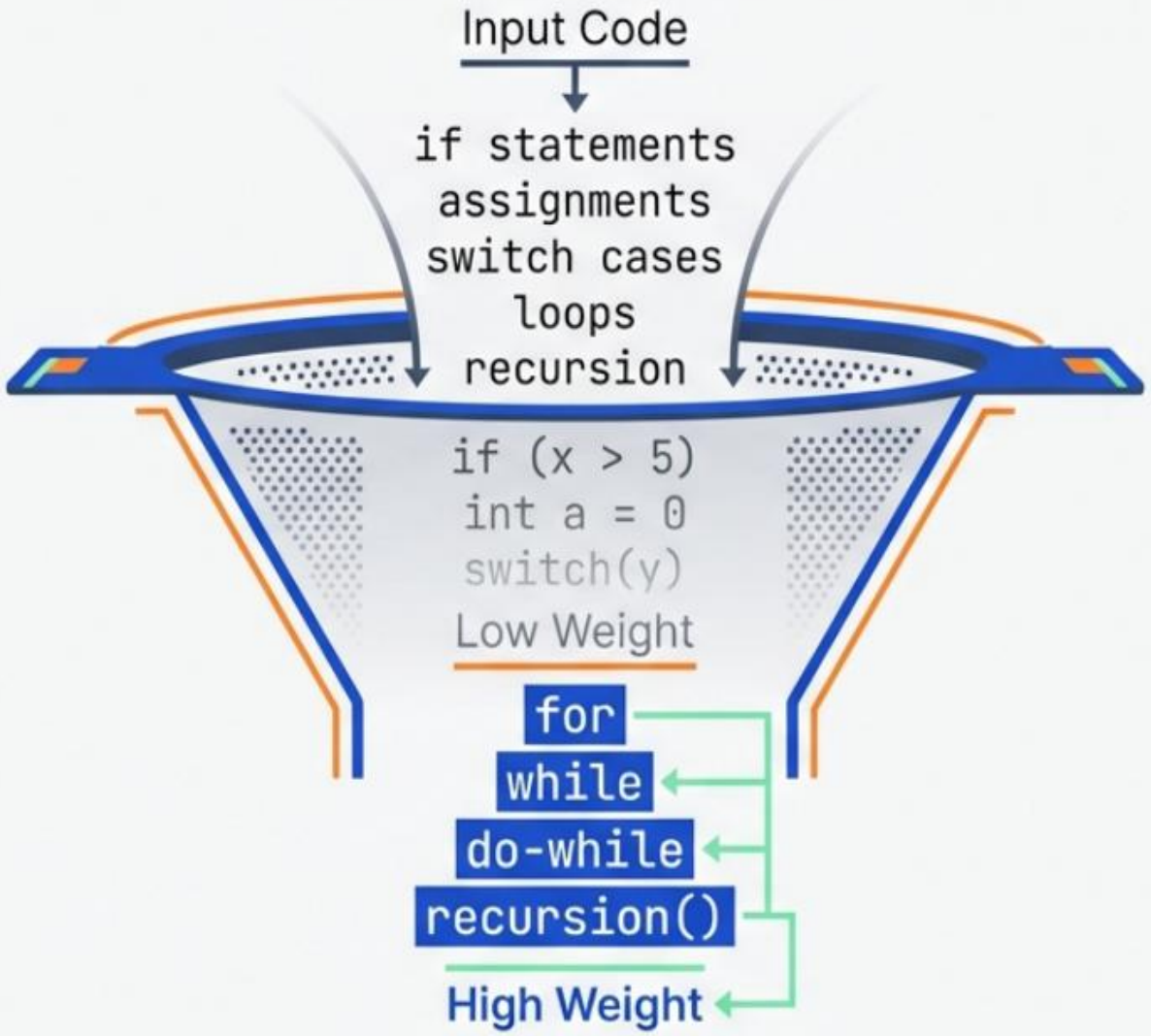
```
for (int i = 0; i < n; i++) {
    sum += i;
    printf("Value: %d\n", sum);
}
```

$$T(n) = O(n)$$

Identifying growth patterns.
Approximate and fast.

Goal: Determine the Order of Growth, not the exact constants.

The Filter: Identifying High-Weight Structures



Ignore basic operations. Isolate repetition. We only measure what repeats.

The Block Visualization Technique

```
for (int i = 1; i <= n; i++) {
    // statement
}
```

The Scope

1. Start: “1”
2. End: “N”
3. Step: “+1”

→ **Executes N Times**

Pattern A: Linear Growth O(n)

The Code

```
for (int i = 1; i <= n; i++) {
   ...
}

for (int i = n; i >= 1; i--) {
   ...
}
```

The Rule

Addition (+) or Subtraction (-)

If the iterator increments or decrements by a constant amount, the complexity is Linear.

$$T(n) \propto n$$

Constants are ignored. i += 2 is still considered Linear (n).

Pattern B: Logarithmic Growth O(log n)

The Code

```
for (int i = 1; i<=n; i *= 2) {
   ...
}

for (int i = n; i >= 1; i /= 3) {
   ...
}
```

The Rule

Multiplication (*) or Division (/)

If the iterator multiplies or divides by a constant factor, the complexity is Logarithmic.

$$T(n) \propto \log n$$

The base of the log (2, 3, 10) is a constant factor and is generalized as log n.

Composition Rules: Sequential vs. Nested

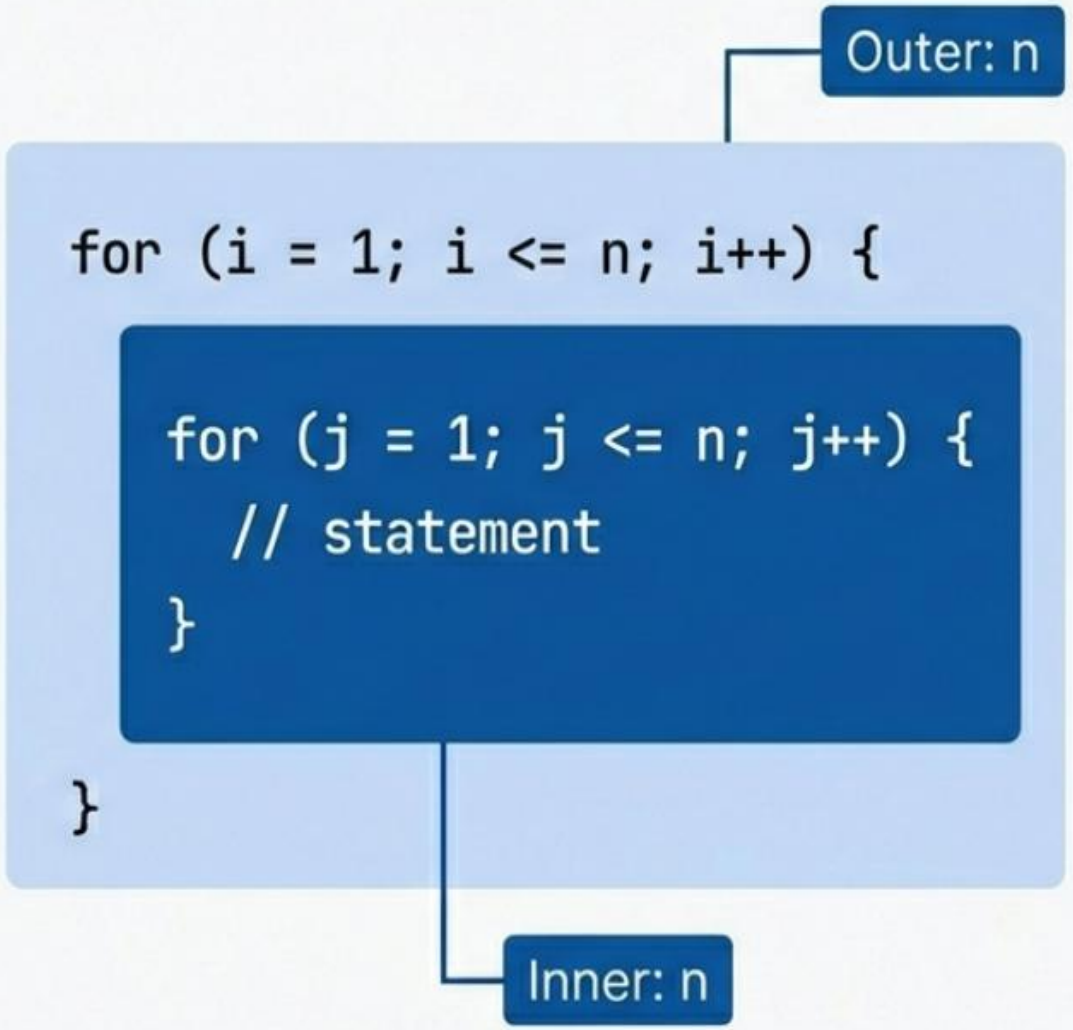
Sequential

Add the complexities.

Nested

Multiply the complexities.

Applied Analysis: The Quadratic Case



Relationship: Nested

Math: Outer × Inner

Calculation: $n \times n = n^2$

Complexity: Quadratic O(n²)

Handling Mixed Structures

```
for (i...n) {
  for (j...n) { ... }
}
```

→ Block 1
Cost = n^2

```
for (k=1; k<=n; k*=3) {
  ...
}
```

→ Block 2
Cost = $\log n$

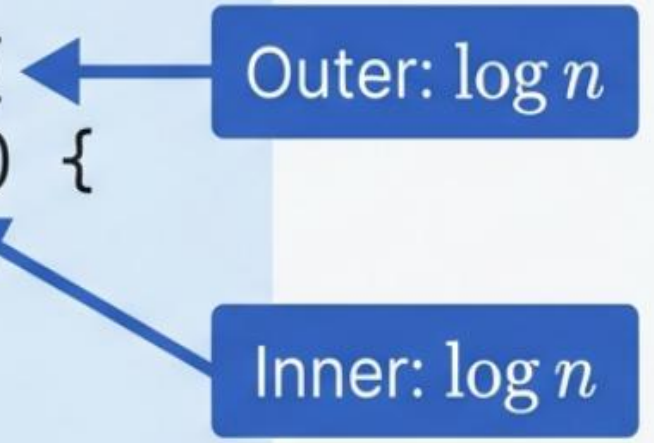
Total T(n) = $n^2 + \log n$

Dominance Rule: The largest term defines the complexity.

$O(n^2)$

Nested Logarithmic Loops

```
for (i = 1; i <= n; i *= 3) {
    for (j = 1; j <= n; j *= 3) {
        // statement
    }
}
```



Calculation: $\log n \times \log n$

Result: $(\log n)^2$

Note: This is $\log^2 n$. It is NOT $\log(n^2)$.

The Simple Method Cheat Sheet

	Increment/Decrement	n (Linear)
	Multiply/Divide	log n (Logarithmic)
	Sequential	Add (+)
	Nested	Multiply (×)

Use these rules for standard loops with fixed boundaries (1 to N).

The Challenge: Dependent Loops

```
for (i = 1; i <= n; i++) {
    for (j = 1; j <= i; j++) {
        // statement
    }
}
```

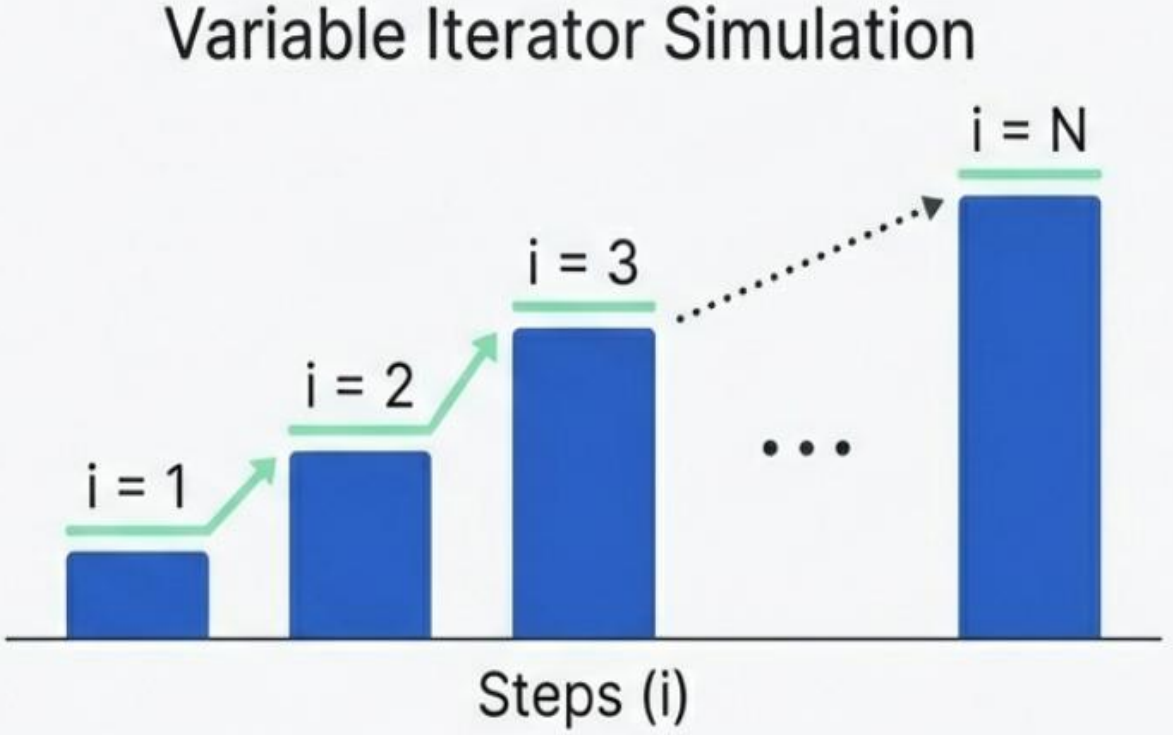
Wait. The limit isn't N. It's i.

Is the complexity n × i

Why the Simple Method Fails Here

T(n) must be a function of N (Input Size).

‘i’ is a variable iterator, not a constant. It changes value every step.



The inner loop grows. It is not fixed width.

Simple multiplication assumes a fixed width box. This box is a triangle.

The Verdict on the Simple Method

Valid Use Cases

- ✓ Standard loops.
- ✓ Fixed boundaries (1 to N).
- ✓ Quick estimation/comparison.
- ✓ Identifying Linear vs. Quadratic.

Invalid Use Cases

- ✗ Dependent loops (Inner loop depends on Outer).
- ✗ Variable boundaries (1 to i).
- ✗ Precise instruction counting.

The Simple Method is an estimation heuristic, not a universal law.

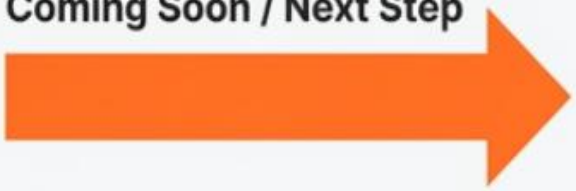
Beyond Estimation

Summary

We have mastered the structural estimation of:

- Linear O(n)
- Logarithmic O(log n)
- Quadratic $O(n^2)$

Coming Soon / Next Step



For dependent loops (1 to i), we require Mathematical **Summation**.

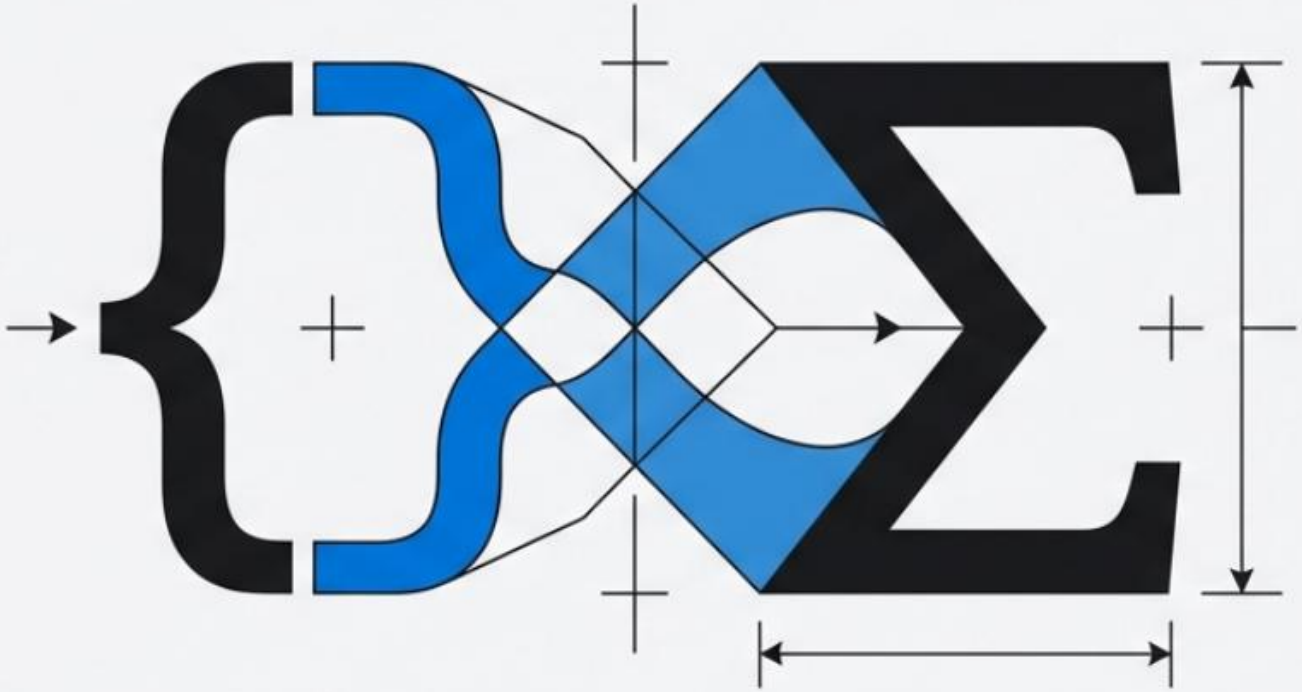
$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Algorithm Analysis Part 1 Complete.

Algorithm Analysis: Calculating T(n) Using Summation

A Definitive Guide to Converting Code Loops into Mathematical Equations

```
for (int i = 0; i < n; i++) {
  ...
}
```

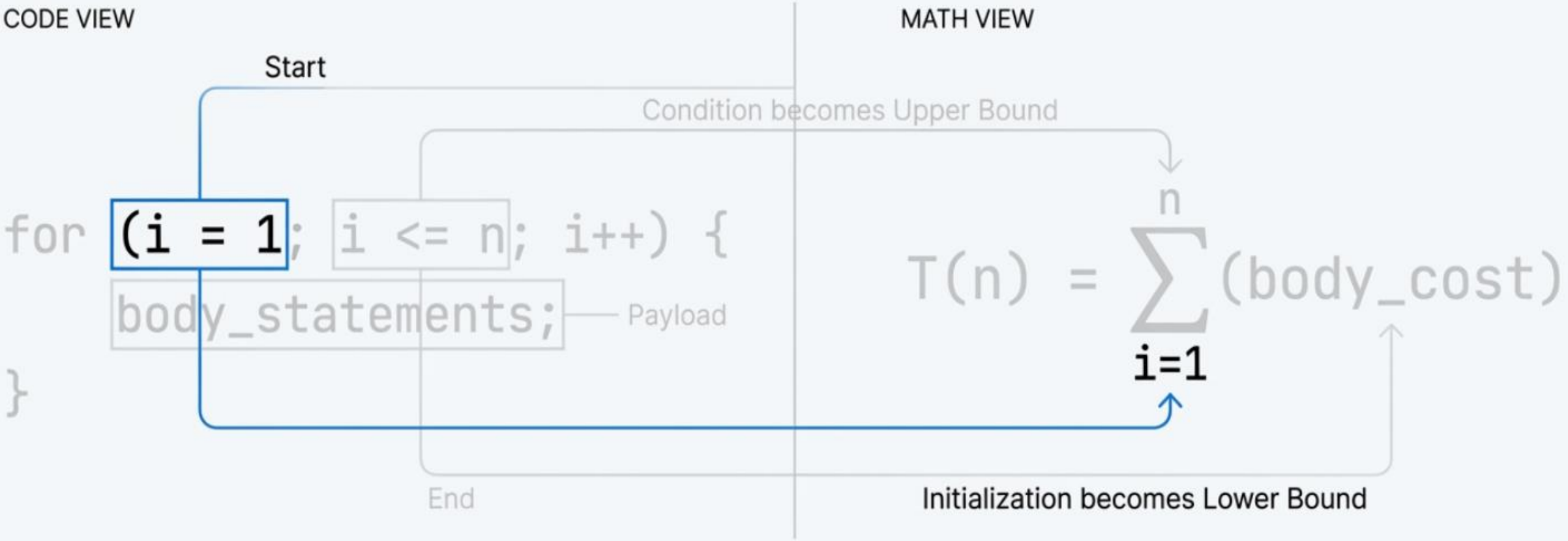


$$T(n) = \sum_{i=0}^{n-1} (c) = c \cdot n$$

Stop guessing. Start deriving. Every loop structure in programming has a direct equivalent in algebraic summation. This guide bridges the gap between code logic and mathematical proof.

Mapping the Initialization

Start → Lower Bound



The loop's starting variable assignment defines the Lower Bound of the summation. It tells the math where to begin counting.

Mapping the Condition

Condition → Upper Bound

CODE VIEW

Condition becomes Upper Bound

Start

End

```
for (i = 1; i <= n; i++) {
    body_statements;
}
```

Payload

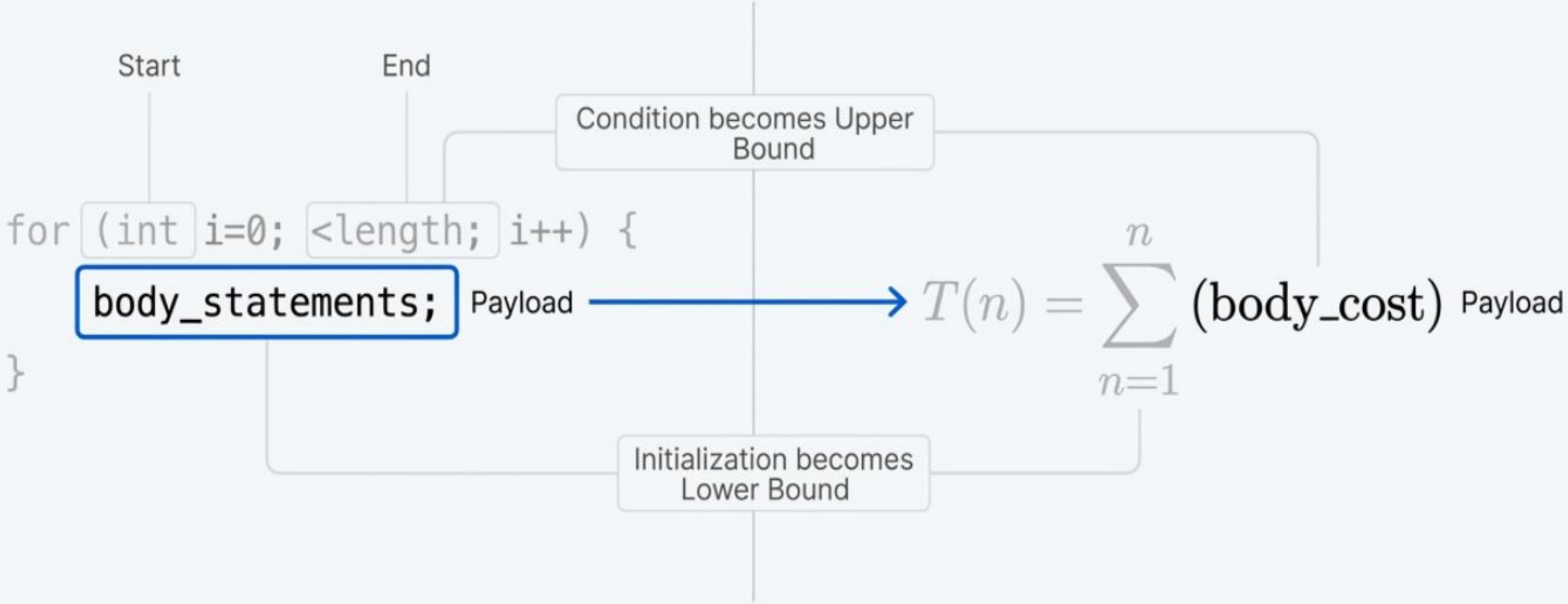
MATH VIEW

$$T(n) = \sum_{i=1}^{n} (body\_cost)$$

The loop's exit condition defines the Upper Bound of the summation. It tells the math the maximum value 'i' will reach.

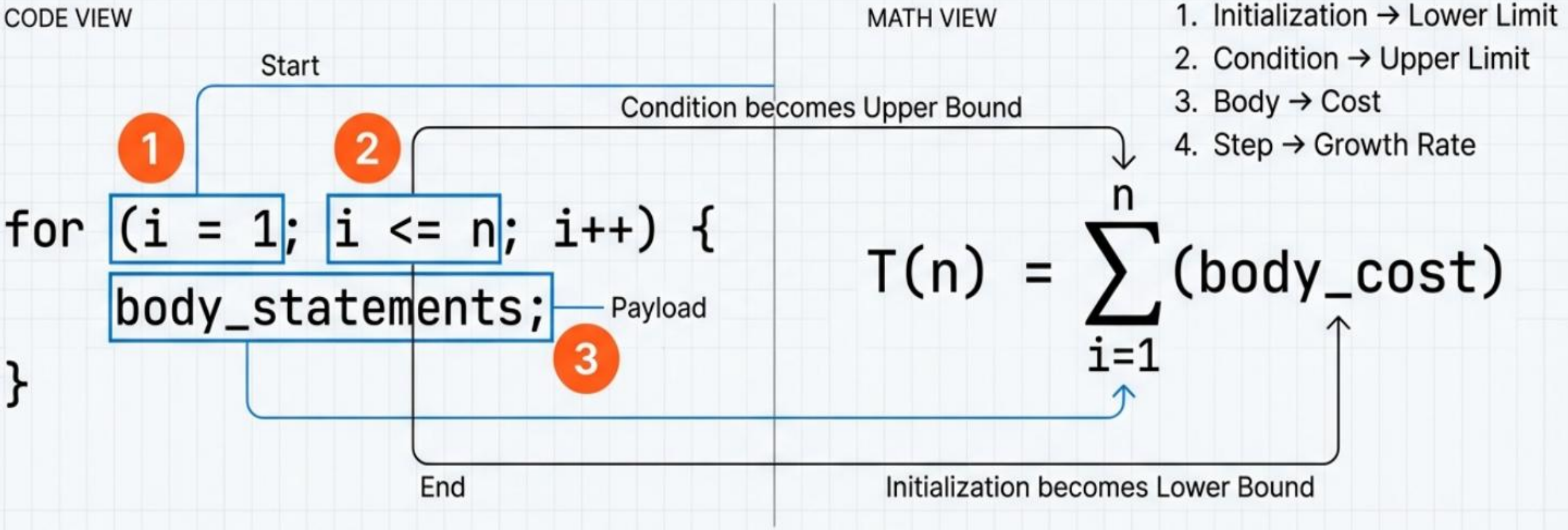
Mapping the Payload

Body → Summation Term



The statements executed inside the curly braces represent the work done in each iteration. This becomes the term inside the parenthesis—the cost we are adding up repeatedly.

The Complete Translation Map



4

STEP LOGIC RULE:

If (++ or --) → Bound is n (Linear). If (* or /) → Bound is log n (Logarithmic).

The Mathematical Toolkit

Fundamental laws required for algebraic derivation.

The Constant Rule

$$\sum_{i=1}^{n} c = c \cdot n$$

Summing a constant 'c', 'n' times.

The Arithmetic Series

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Used when the inner loop depends on the outer variable 'i'.

The Logarithmic Rule

$$\sum_{i=1}^{\log n} c = c \cdot \log n$$

Applies when loop variables multiply or divide (e.g., i=i*2).

The Dependent Variable Rule

$$\sum_{j=\text{lower}}^{\text{upper}} 1 = (\text{upper} - \text{lower} + 1)$$

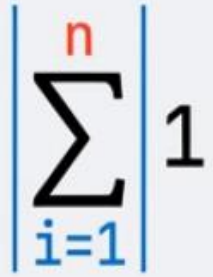
Calculates iteration count when bounds are variables.

Basic Linear Loops: Forward and Reverse

Forward Loop

```
for(i=1; i<=n; i++) {
  stmt;
}
```

Maps to



$$\sum_{i=1}^{n} 1$$

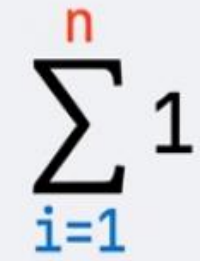
Apply Constant Rule: $1 \cdot n$

$$T(n) = n$$

Reverse Loop

```
for(i=n; i>=1; i--) {
  stmt;
}
```

Normalize Bounds



$$\sum_{i=1}^{n} 1$$

Counting down n→1 is equivalent to counting up 1→n.

Apply Constant Rule: $1 \cdot n$

$$T(n) = n$$

The Logarithmic Loop (Multiplication)

```
for (i=1; i<=n; i=i*2) {
   stmt;
}
```

Step is multiplication (*2).

The loop does not run ‘n’ times.
It runs based on powers of 2.

Translation

Upper Bound = $\log_2 n$

$$\sum_{k=1}^{\log_2 n} 1$$

↓ Apply Constant Rule to limit

$$1 \cdot (\log_2 n)$$

$$\mathbf{T(n) = \log_2 n}$$

If step was *3, result would be $\log_3 n$.

Independent Nested Loops

Outer

```
for (i=1; i<=n; i++) {      // Outer
    for (j=1; j<=n; j++) { // Inner
        stmt;
    }
}
```

Inner

$$\sum_{i=1}^{n} \left[\sum_{j=1}^{n} 1 \right] \xrightarrow{\text{Solve Inner First (Constant Rule)}} \sum_{i=1}^{n} [n] \xrightarrow{\text{Solve Outer (Constant relative to i)}} n \cdot n \longrightarrow \boxed{T(n) = n^2}$$

(Translation) (Substitution) (Calculation) (Result)

Dependent Nested Loops (The Triangle)

```
for (i=1; i<=n; i++) {
    for (j=1; j<=i; j++) {  // Limit depends on i
        stmt;
    }
}
```

$$\sum_{i=1}^{n} \left[\sum_{j=1}^{i} 1 \right]$$

↓ Inner loop runs “i” times.

$$\sum_{i=1}^{n} i$$

↓ Apply Arithmetic Series Law.

$$T(n) = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \rightarrow O(n^2)$$

Logarithmic Nested Loops

```
for (i=1; i<=n; i=i*3) {
    for (j=1; j<=n; j=j*3) {
        stmt;
    }
}
```

Step 1: Map Bounds

Outer Limit: $\log_3 n$

Inner Limit: $\log_3 n$

Step 2: Write Equation

$$\sum_{i=1}^{\log_3 n} \left[\sum_{j=1}^{\log_3 n} 1 \right]$$

Step 3: Solve Inner

$$\sum_{i=1}^{\log_3 n} \left[\sum_{j=1}^{\log_3 n} \log_3 n \right] \longrightarrow (\log_3 n) \cdot (\log_3 n)$$

Step 4: Solve Outer

Final Result

$$T(n) = (\log_3 n)^2$$

Advanced Derivation: Complex Dependencies (Part 1)

Setup for nested loop with offsets.

The Problem

Outer Bound

```
for (i=0; i < n-2; i++) {
  /\ JetBrains Mono font  \
  for (j=i+1; j < n; j++) {
    stmt;
  }
}
```

Inner Start

Inner End

The Translation

Requires "Upper - Lower + 1" Rule

$$\sum_{i=0}^{n-3} \left[\sum_{j=i+1}^{n-1} 1 \right]$$

Note: Bounds are adjusted to be inclusive (n-2 becomes n-3, n becomes n-1).

Advanced Derivation: The Solution (Part 2)

Step 1: Inner Loop Count

$$\text{Count} = (\text{Upper} - \text{Lower} + 1)$$

$$\text{Count} = (n - 1) - (i + 1) + 1$$

$$\text{Count} = n - 1 - i$$

Step 2: Update Outer Summation

$$\sum_{i=0}^{n-3} (n - 1 - i)$$

Step 3: Distribute

$$\sum_{i=0}^{n-3} n \; - \; \sum_{i=0}^{n-3} 1 \; - \; \sum_{i=0}^{n-3} i$$

Step 4: Solve

$$n(n-2) - (n-2) - (\text{Arithmetic Series})$$

Dominant term is n × n. → Complexity: $O(n^2)$

Special Case Laws & Formulas

`x^i`

$$\sum_{i=1}^{n} x^i = \frac{x^{n+1} - x}{x-1}$$

Powers

`i²`

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

Squares

`1/i`

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$$

Reciprocals

Advanced Summation Rules

Rule 5: Geometric Series

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1} - 1}{x - 1}$$

Rule 6: Harmonic Series

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln n$$

Rule 7: The 'Magic' Log Rule

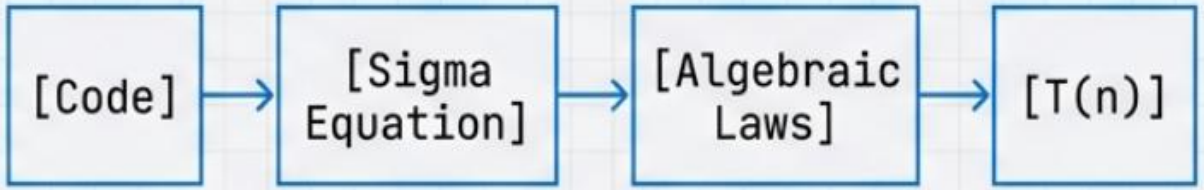
Summation bounds go to $\log_2 n$ and body is 2^i

$$\sum_{i=1}^{\log_2 n} 2^i = 2(n - 1)$$

Appears when doubling iterator (i*2) inside a loop.

The Analysis Workflow

1. **Define Bounds:**
   Map **`init`/`condition`** to **Lower/Upper** Sigma limits.
2. **Identify Step:**
   `+` is **Linear**. **`*`** is **Logarithmic.**
3. **Check Dependency:**
   Does inner loop use outer variable? (e.g. **`j < i`**)
4. **Simplify:**
   Solve algebra to find highest order term.



Algorithmic Translation

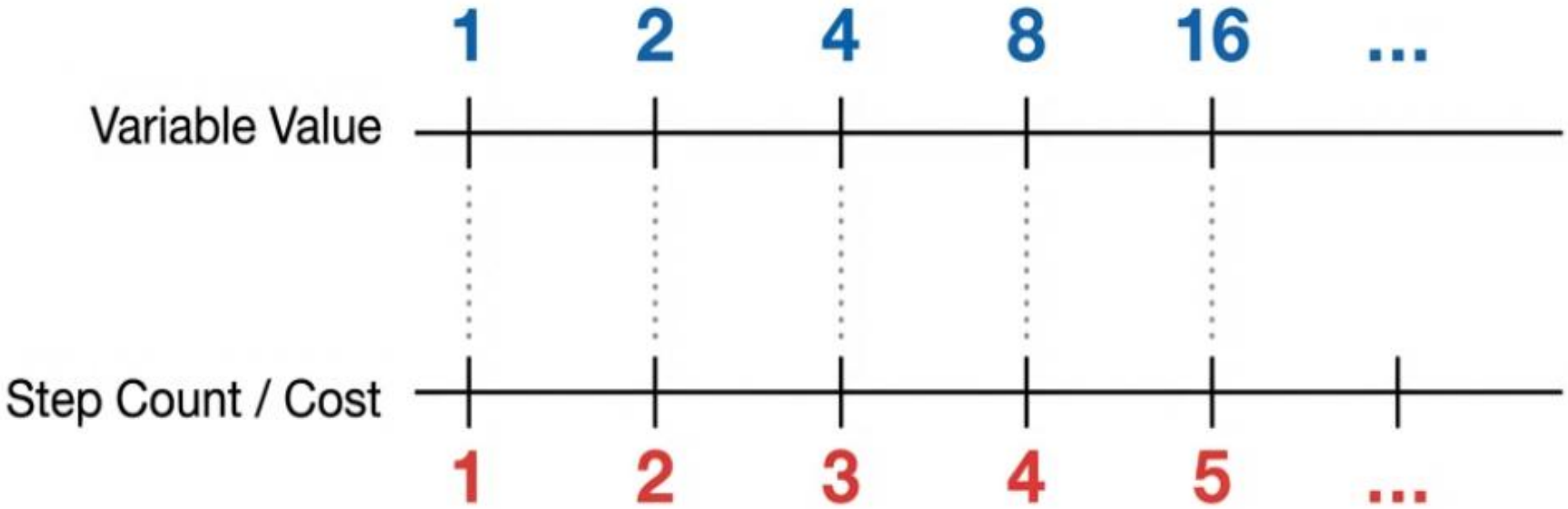
From Loops to Summations: A Guide to Modeling Iteration Complexity

```
for (int i = 1; i <= n; i *= 2)
```

$$\sum_{k=0}^{\log_2 n}$$

To understand efficiency, we must stop reading code line-by-line and start modeling it mathematically.

Don’t Simulate the Execution. Count the Steps.



The Insight:

- The Sigma symbol ($\sum$) counts the *iterations*.
- The body of the Sigma counts the *work* (1 unit) done per iteration.

The Baseline: Lininear Translation

CODE SNIPPET

```
for (int i = 1; i <= n; i++)
```

Linear Increment (+1)

MATHEMATICAL MODEL

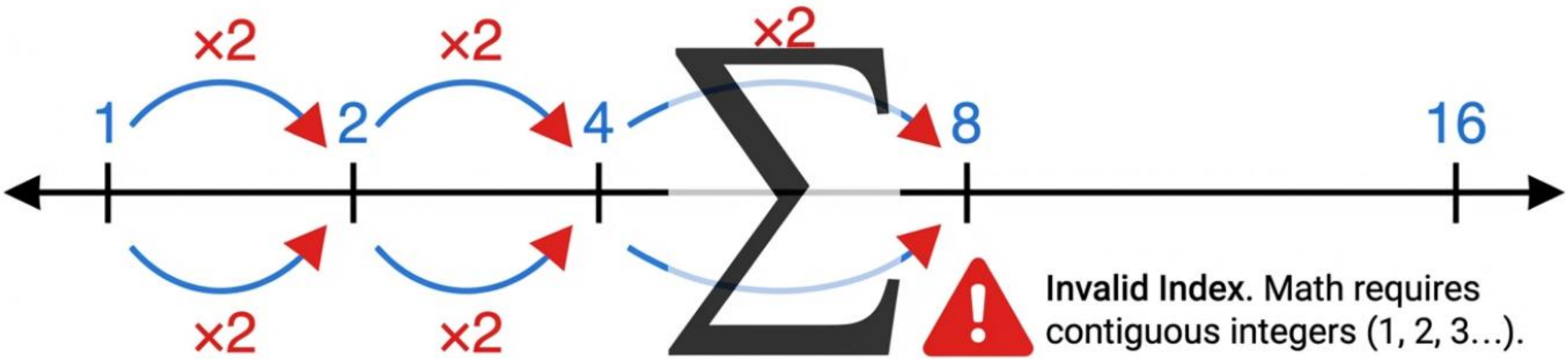
$$\sum_{i=1}^{n} 1 = n$$

$i=1$ ← Index matches Variable

— When the loop step is linear (+1), the code variable and the summation index are identical.

Academic Editorial

The Friction: When ‘i’ Skips Values



We cannot write $\sum_{i=1,\,\text{step=doubling}}^{n}$. We need a bridge between the geometric code and the linear math.

The Substitution Rule: Introducing k

k represents the Step Number.

Step Count (k)	0	1	2	3	...
Variable Value (i)	1	2	4	8	...

$$i = 2^k$$

We substitute the code variable i with the exponential expression 2^k. This allows us to sum over k, which grows linearly $(0, 1, 2, 3\ldots)$, satisfying the rules of mathematics.

Transforming the Bounds

Original Condition

$$i \leq n$$

Substitution ($i = 2^k$)

$$2^k \leq n$$

Solve for k

$$k \leq \log_2 n$$

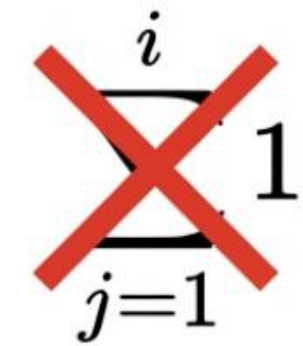
$$\sum_{k=0}^{\log_2 n} \cdots$$

The loop doesn't run n times. It runs $\log n$ times.
We no longer sum over i; we sum over k.

Handling Inner Loop Dependencies

When the inner loop depends on the outer loop (e.g., `j <= i`), we must not use **i** in the summation bound. Substitute it immediately.

`j <= i` →



Outer Loop Step →

Case Study I: Linear / Independent

Code Snippet

```
for(int i = 1; i <= n; i++)
  for(int j = 1; j <= n; j++)


for(int i = 1; i <= n; i++)
  for(int j = 1; j <= n; j++)
```

Analysis

Outer: Linear $(1 \dots n)$

Inner: Linear $(1 \dots n)$

Dependency: None

Time Complexity

Derivation

$$\sum_{i=1}^{n} \sum_{j=1}^{n} 1$$

$$= \sum_{i=1}^{n} n$$

$$= n \cdot n$$

$O(n^2)$

Case Study II: Linear / Dependent

Code Snippet

```
for(int i = 1; i <= n; i++)
  for(int j = 1; j <= i; j++)


for(int i = 1; i <= n; i++)
  for(int j = 1; j <= i; j++)
```

Analysis

Outer: Linear $(1 \dots n)$

Inner: Dependent $(1 \dots i)$

Type: Triangular Series

Time Complexity

Derivation

$$\sum_{i=1}^{n} \sum_{j=1}^{i} 1$$

$$= \sum_{i=1}^{n} i$$

$$= \frac{n(n+1)}{2}$$

$O(n^2)$

Case Study III: Logarithmic / Logarithmic

Code Snippet

```
for(int i = 1; i <= n; i *= 2)
  for(int j = 1; j <= i; j *= 2)
    print "hello";
```

Analysis

- Outer: Geometric ($i = 2^k$)
- Inner: Geometric ($j = 2^m$)
- Bounds: $k \leq \log n, m \leq k$

Derivation

$$\sum_{k=0}^{\log n} \sum_{m=0}^{k} 1$$

$$= \sum_{k=0}^{\log n} (k+1)$$

$$\approx \frac{1}{2}(\log n)^2$$

$O(\log^2 n)$

Case Study IV: Logarithmic / Linear (The Trap)

Code Snippet

```
for(int i = 1; i <= n; i *= 2)
  for(int j = 1; j <= i; j++)
    print "hello";
```

Analysis

- Outer: Geometric ($i=2^k$)
- Inner: Linear dependent ($j \leq i$)
- Note: Inner runs 2^k times.

Derivation

$$\sum_{k=0}^{\log n} \sum_{j=1}^{2^k} 1$$

$$= \sum_{k=0}^{\log n} 2^k$$

$$= 2^{\log n+1} - 1 = 2n - 1$$

$O(n)$

Questions
Find Time Equation T(n)

```
int x = 1 + 3;
for(int i = 1; i ≤ n; i++)
    print i;
int y = x * 120;
print y;
```

```
for(int i = 1; i ≤ n; i *= 2)
    print i;
```

```
for(int i = n; i ≥ 1; i--)
    print i;
```

```
for(int i = n; i ≥ 1; i/= 2)
    print i;
```

```
for(int i = 1; i ≤ n; i++)
    for(int j = 1; j ≤ n; j++)
        print "hello";
```

```
for(int i = 1; i ≤ n; i *= 3)
    for(int j = 1; j ≤ n; j++)
        print "hello";
```

```
for(int i = 1; i ≤ n; i++)
    for(int j = 1; j ≤ n; j++)
        print "hello";
for(int i = 1; i ≤ n; i *= 3)
    print "hello";
```

```
for(int i = 1; i ≤ n; i *= 3)
    for(int j = 1; j ≤ n; j *= 3)
        print "hello";
```

```
for(int i = 1; i ≤ n; i++)
    for(int j = 1; j ≤ i; j++)
        print "hello";
```

```
for(int i = 1; i ≤ n; i *= 2)
    print i;
```

```
for(int i = 1; i ≤ n; i++)
    print i;
```

```
for(int i = n; i ≥ 1; i /= 2)
    print i;
```

```
for(int i = n − 1; i ≥ 1; i − −)
    print i;
```

```
int F(A,n)
S=0;
for i = 0 to n-2
  for j = i+1 to n-1
        if (Ai > Aj) S++
return S
```

```
for(int i = 1; i ≤ n; i++)
    for(int j = 1; j ≤ n; j++)
        print "hello";
```

```
for(int i = 1; i ≤ n; i *= 2)
    for(int j = 1; j ≤ i; j *= 2)
        print "hello";
```

```
for(int i = 1; i ≤ n; i++)
    for(int j = 1; j ≤ i; j++)
        print "hello";
```

```
for(int i = 1; i ≤ n; i *= 2)
    for(int j = 1; j ≤ i; j++)
        print "hello";
```

```
function mystery(n)
    r := 0
    for i := 1 to n − 1 do
        for j := i + 1 to n do
            for k := 1 to j do
                r := r + 1
    return(r)
```

```
function prestiferous(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                for l := 1 to i + j − k do
                    r := r + 1
    return(r)
```

```
function pesky(n)
    r := 0
    for i := 1 to n do
        for j := 1 to i do
            for k := j to i + j do
                r := r + 1
    return(r)
```

```
function conundrum(n)
    r := 0
    for i := 1 to n do
        for j := i + 1 to n do
            for k := i + j − 1 to n do
                r := r + 1
    return(r)
```