Time Complexity Part 1

Contents

[1. Basic Loop Iteration Patterns 2](#_Toc199513571)

[Simple for Loop 2](#_Toc199513572)

[Two Independent Loops 2](#_Toc199513573)

[Nested Loop 2](#_Toc199513574)

[2. Counting Elements in Ranges 2](#_Toc199513575)

[3. Gaussian sum 3](#_Toc199513576)

[4. Arithmetic Progression (AP) 3](#_Toc199513577)

[5. Geometric Progression (GP) 3](#_Toc199513578)

[6. Special Loop Patterns 4](#_Toc199513579)

[Odd Numbers Only 4](#_Toc199513580)

[Logarithmic Growth 4](#_Toc199513581)

[Halving 5](#_Toc199513582)

[Summary 5](#_Toc199513583)

[7. Comparative Analysis 6](#_Toc199513584)

Basics

## Basic Loop Iteration Patterns

### Simple for Loop

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

...

}

* Number of iterations:

### Two Independent Loops

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

for (int j = 1; j <= m; j++) { ... }

* Total iterations:

### Nested Loop

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

for (int j = 1; j <= n; j++) {

...

}

}

* Inner loop runs n times per outer loop.
* Total iterations:

## Counting Elements in Ranges

* Inclusive range
  + This means, we must include both A and B while counting.
  + If , , then number of elements in the range 1 to 10 is:
* Semi-inclusive or :
  + : We must include A in the count but not B.
    - If , , then number of elements in the range 1 to 10 excluding 10 is:
  + : We must not include A but we must include B while counting.
    - If , , then number of elements in the range 1 to 10 excluding 1 is:
* Exclusive (A, B): B - A – 1
  + We must not include both A and B while counting.
  + If , , then number of elements in the range 1 to 10 excluding both 1 and 10 is:

## Gaussian sum

* Often called the **sum of the first n natural numbers.**

## Arithmetic Progression (AP)

* Properties:
  + Sequence:
  + term:
  + Sum of first terms:

## Geometric Progression (GP)

* Properties:
  + Sequence:
  + term:
  + Sum of first terms :
  + If :

## Special Loop Patterns

### Odd Numbers Only

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

* Iterates over odd numbers
* Iterations:

### Logarithmic Growth

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

* Starts with
* In each iteration, is multiplied by
* Loop continues while
* So, the values of will be:
* The loop stops when:
* So, we want to find the maximum such that:
* Apply logarithm (base 2) to both sides:
* So, the number of iterations will be:
* Example:
  + Let’s take n = 32.
  + i = 1 → 2⁰
  + i = 2 → 2¹
  + i = 4 → 2²
  + i = 8 → 2³
  + i = 16 → 2⁴
  + i = 32 → 2⁵
  + i = 64 →
* At this point, would give 64, which is > 32, so the loop exits.
* So, the loop ran **6 times**, and:

### Halving

while (n > 1) {

n /= 2;

}

* Divides by 2 until ≤ 1 ⇒ **log₂(n)** iterations
  + 32 → 16 → 8 → 4 → 2 → 1 → stop
  + 1 → stop
  + , taking log on both sides
  + log₂(n) k
  + k ≈ floor(log₂(n))
* Number of iterations k ≈ floor(log₂(n))

### Summary

* Loop iteration patterns are foundational to understanding time complexity.
* Patterns like linear (n), quadratic (n²), and logarithmic (log n) emerge naturally in code.
* Get comfortable identifying and estimating iteration counts.

## Comparative Analysis

| **Order** | **Complexity** |
| --- | --- |
| 1 | **1** |
| 2 | **log n** |
| 3 | **√n** |
| 4 | **n** |
| 5 | **n log n** |
| 6 | **n√n** |
| 7 | **n²** |
| 8 | **n³** |
| 9 | **2ⁿ** |
| 10 | **n!** |

* From Slowest to Fastest
* Comparison:
  + **vs** **vs**
    - is much smaller than for large
    - Example:
      * n = 10⁹
      * √n ≈ 31,622
      * log₂(n) ≈ 30
    - So,
  + **vs**
    - We know is smaller than .
    - When we multiply both by n,
* Quick Reference Order (Smallest to Largest)