Time and Space Complexity

Table of Contents

1. Why Time Complexity Matters

The Problem: Speed Matters!

Imagine you write two programs that solve the same problem:

Program A:

```
for(int i = 0; i < n; i++) {
   sum += arr[i];
}
```

Program B:

```
for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
      if(i == j) {
         sum += arr[i];
      }
   }
}
```

Both give the same output!

But when n = 10,000:

- Program A: Finishes instantly
- Program B: Takes several seconds

Why? Because Program A does ~10,000 operations, while Program B does ~100,000,000 operations!

Why Not Measure in Seconds?

Problem 1: Hardware Dependency

- Same code runs at different speeds on different computers
- Your laptop vs. your friend's laptop → different times

Problem 2: We Care About Growth, Not Exact Time

- Question: "If I double the input size, how much slower does it get?"
- NOT: "How many seconds does it take on my specific computer?"

Solution: Time Complexity measures how operations grow with input size!

2. What is Time Complexity?

Definition:

Time complexity is a function that describes how the number of operations an algorithm performs grows as the input size increases.

Simple Definition:

"As input size n increases, how does the work increase?"

Example 1: Linear Growth

cpp

```
void printNumbers(int n) {
   for(int i = 0; i < n; i++) {
      cout << i;
   }
}
```

Analysis:

- n = 10 → 10 operations
- n = 100 → 100 operations
- n = 1000 → 1000 operations

Pattern: Operations = n

Time Complexity Function: f(n) = n

Example 2: Quadratic Growth

cpp

```
void printPairs(int n) {
   for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         cout << i << "," << j;
      }
   }
}
```

Analysis:

- n = 10 → 100 operations
- n = 100 → 10,000 operations
- n = 1000 → 1,000,000 operations

Pattern: Operations = n × n = n^2

Time Complexity Function: f(n) = n^2

3. Counting Operations

What Counts as an Operation?

Basic operations:

- Assignment: x = 5
- Comparison: if(x > 5)
- Arithmetic: x + y, x * y
- Array access: arr[i]
- Print statement: cout << x

Example: Counting Step by Step

cpp

```
int sum = 0;            // 1 operation
```

```
for(int i = 0; i < n; i++) {  // n+1 comparisons, n assignments, n increments
    sum += arr[i];       // n operations
}
```

Exact count: 1 + (n+1) + n + n + n = 3n + 2 operations

But we simplify this! (explained later)

4. Growth Patterns

The Growth Hierarchy (Slowest to Fastest)

Pattern	Formula	n=10	n=100	n=1000	Name
Constant	1	1	1	1	Constant
Logarithmic	log n	~3	~7	~10	Logarithmic
Linear	n	10	100	1,000	Linear
Linearithmic	n log n	~30	~700	~10,000	Linearithmic
Quadratic	n^2	100	10,000	1,000,000	Quadratic
Cubic	n^3	1,000	1,000,000	1,000,000,000	Cubic
Exponential	2^n	1,024	10^{30}	10^{300}	Exponential

Visual Comparison

When n = 1,000:

- **Constant (1):** 1 operation → Instant ✓
- **Linear (n):** 1,000 operations → Instant ✓
- **Quadratic (n^2):** 1,000,000 operations → ~1 second
- **Cubic (n^3):** 1,000,000,000 operations → ~16 minutes ✗

Key Insight: As n grows, the growth pattern matters MORE than constants!

5. Big-O, Omega, and Theta Notations

These are **mathematical tools** to compare functions. They help us express how functions grow relative to each other.

Big-O (O) - Upper Bound

Symbol: O (pronounced "Big-Oh")

Formal Definition:

f(n) = O(g(n)) means there exist constants C > 0 and n_0 such that:

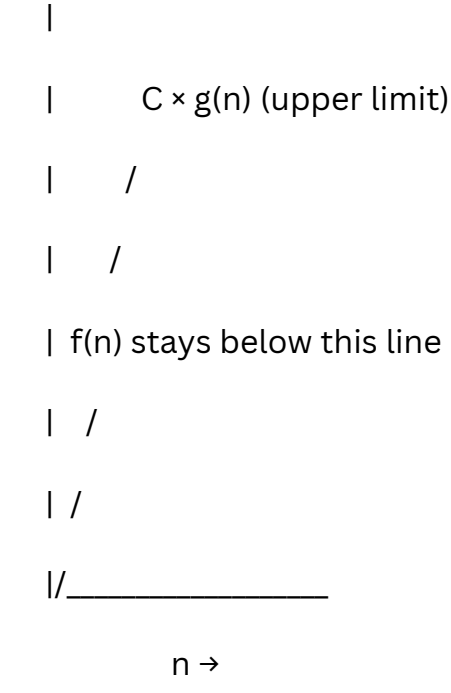
$f(n) \leq C \times g(n)$ for all $n \geq n_0$

Plain English:

"My function f(n) grows **at most** as fast as g(n)"

"f(n) is **bounded above** by g(n)"

Visual Understanding:



Example:

$f(n) = 3n^2 + 100n + 50$

Claim: $f(n) = O(n^2)$

Proof: We need to find C and n_0 such that:

$3n^2 + 100n + 50 \leq C \times n^2$

For $n \geq 1$:

- $3n^2 + 100n + 50 \leq 3n^2 + 100n^2 + 50n^2 = 153n^2$

So C = 153, $n_0 = 1$ works!

Therefore: $f(n) = O(n^2)$ ✓

Multiple Valid Big-O:

For $f(n) = 3n^2 + 100n + 50$, ALL of these are TRUE:

- $f(n) = O(n^2)$ ✓ (tight bound - best)
- $f(n) = O(n^3)$ ✓ (loose bound - valid but not useful)
- $f(n) = O(n^{10})$ ✓ (very loose - technically correct)

We prefer the tightest bound!

Big-Omega (Ω) - Lower Bound

Symbol: Ω (pronounced "Big-Omega")

Formal Definition:

$f(n) = \Omega(g(n))$ means there exist constants $C > 0$ and n_0 such that:

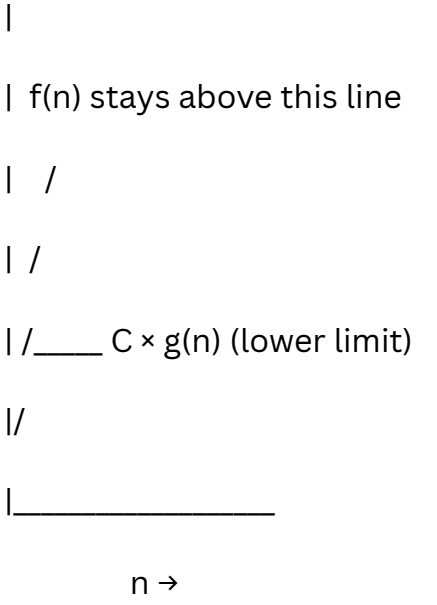
$f(n) \geq C \times g(n)$ for all $n \geq n_0$

Plain English:

"My function f(n) grows **at least** as fast as g(n)"

"f(n) is **bounded below** by g(n)"

Visual Understanding:



Example:

$f(n) = 3n^2 + 100n + 50$

Claim: $f(n) = \Omega(n^2)$

Proof: We need to find C and n_0 such that:

$3n^2 + 100n + 50 \geq C \times n^2$

For $n \geq 1$:

- $3n^2 + 100n + 50 \geq 3n^2$

So C = 3, $n_0 = 1$ works!

Therefore: $f(n) = \Omega(n^2)$ ✓

Multiple Valid Big-Omega:

For $f(n) = 3n^2 + 100n + 50$, ALL of these are TRUE:

- $f(n) = \Omega(n^2)$ ✓ (tight bound - best)
- $f(n) = \Omega(n)$ ✓ (loose bound - valid but not useful)
- $f(n) = \Omega(1)$ ✓ (very loose - technically correct)

Again, we prefer the tightest bound!

Big-Theta (Θ) - Tight Bound

Symbol: Θ (pronounced "Big-Theta")

Formal Definition:

$f(n) = \Theta(g(n))$ means BOTH:

- $f(n) = O(g(n))$ **AND**
- $f(n) = \Omega(g(n))$

Equivalently: There exist constants $C_1, C_2 > 0$ and n_0 such that:

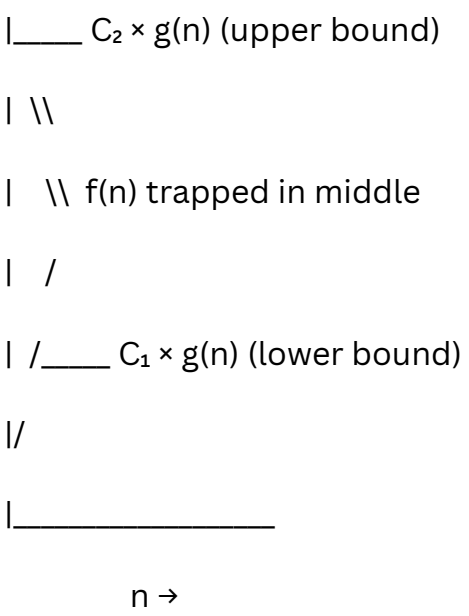
$C_1 \times g(n) \leq f(n) \leq C_2 \times g(n)$ for all $n \geq n_0$

Plain English:

"My function f(n) grows **exactly** at the same rate as g(n)"

"f(n) is **sandwiched** between two multiples of g(n)"

Visual Understanding:



Example:

$f(n) = 3n^2 + 100n + 50$

Claim: $f(n) = \Theta(n^2)$

Proof Part 1 (Upper Bound):

- $3n^2 + 100n + 50 \leq 153n^2$ (we proved this earlier)
- So $f(n) = O(n^2)$ ✓

Proof Part 2 (Lower Bound):

- $3n^2 + 100n + 50 \geq 3n^2$ (we proved this earlier)
- So $f(n) = \Omega(n^2)$ ✓

Since BOTH are true:

$3n^2 \leq 3n^2 + 100n + 50 \leq 153n^2$

Therefore: $f(n) = \Theta(n^2)$ ✓

Important Property of Theta:

For $f(n) = 3n^2 + 100n + 50$:

- $f(n) = \Theta(n^2)$ ✓ (ONLY THIS ONE!)
- $f(n) \neq \Theta(n^3)$ ✗ (too loose)
- $f(n) \neq \Theta(n)$ ✗ (too tight)

Theta gives you the EXACT growth rate!

Comparison: Big-O vs Omega vs Theta

Notation	Meaning	Symbol	Visual
Big-O (O)	Upper bound	≤	Function stays BELOW
Omega (Ω)	Lower bound	≥	Function stays ABOVE
Theta (Θ)	Tight bound	=	Function SANDWICHED

Key Insight:

For $f(n) = 3n^2 + 100n + 50$:

Big-O (Upper Bounds - all true):

$O(n^{100})$ ← very loose

$O(n^3)$ ← loose

$O(n^2)$ ← tight ✓ (best)

Omega (Lower Bounds - all true):

$\Omega(n^2)$ ← tight ✓ (best)

$\Omega(n)$ ← loose

$\Omega(1)$ ← very loose

Theta (Exact - only one):

$\Theta(n^2)$ ← ONLY THIS! ✓

When to Use Each Notation?

In Practice:

1. **Big-O (O):**Most commonly used
   - "Worst-case time complexity is $O(n^2)$"
   - Gives an upper bound guarantee

2. **Theta (Θ):**Most precise
    - "Time complexity is Θ(n²)"
    - States exact growth rate
3. **Omega (Ω):**Rarely used alone
    - "Best-case time complexity is Ω(1)"
    - Gives a lower bound

Honest Truth: For most programming, you can skip the notation and just say:

- "Runs in n² time"
- "Best case: 1, Worst case: n"

The notation exists for mathematical rigor in academic papers!

6. Common Time Complexities

O(1) - Constant Time

Definition: Operation takes the same time regardless of input size

Examples:

cpp

// Example 1: Array access

int x = arr[5];

// Example 2: Simple arithmetic

int sum = a + b;

// Example 3: Swap

int temp = a;

a = b;

b = temp;

Characteristics:

- No loops
- Direct access/calculation
- Always takes same number of steps

O(n) - Linear Time

Definition: Operations grow directly with input size

Examples:

cpp

```
// Example 1: Print all elements
for(int i = 0; i < n; i++) {
   cout << arr[i];
}

// Example 2: Sum array
int sum = 0;
for(int i = 0; i < n; i++) {
   sum += arr[i];
}

// Example 3: Find maximum
int max = arr[0];
for(int i = 1; i < n; i++) {
   if(arr[i] > max) {
      max = arr[i];
   }
}
```

Pattern Recognition:

- Single loop from 0 to n
- Loop executes n times
- Constant work inside loop

O(n^2) - Quadratic Time

Definition: Operations grow as square of input size

Examples:

cpp

```
// Example 1: Nested loops (square)
for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
      cout << "*";
   }
}

// Example 2: Bubble Sort
for(int i = 0; i < n-1; i++) {
   for(int j = 0; j < n-i-1; j++) {
      if(arr[j] > arr[j+1]) {
         swap(arr[j], arr[j+1]);
      }
   }
}

// Example 3: Check duplicates
for(int i = 0; i < n; i++) {
   for(int j = i+1; j < n; j++) {
      if(arr[i] == arr[j]) {
         return true;
      }
   }
```

}

Pattern Recognition:

- Two nested loops
- Outer loop: n times
- Inner loop: n times (or depends on outer)
- Total: n × n = n²

O(n³) - Cubic Time

Definition: Operations grow as cube of input size

Examples:

cpp

```
// Example 1: Three nested loops
for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
      for(int k = 0; k < n; k++) {
         cout << i << j << k;
      }
   }
}


// Example 2: Matrix multiplication
for(int i = 0; i < n; i++) {
   for(int j = 0; j < n; j++) {
      C[i][j] = 0;
      for(int k = 0; k < n; k++) {
         C[i][j] += A[i][k] * B[k][j];
      }
   }
```

}

Pattern Recognition:

- Three nested loops
- Each loop runs n times
- Total: n × n × n = n^3

O(log n) - Logarithmic Time

Definition: Operations grow logarithmically (very slow growth)

Key Concept: Algorithm divides problem in half each step

Example Pattern:

cpp

// Counting powers of 2

int count = 0;

int i = 1;

while(i < n) {

  i = i * 2; // Doubles each time

  count++;

}

// Runs $\log_2(n)$ times

Characteristics:

- Problem size halves/doubles each step
- Very efficient even for large n
- $\log_2(1{,}000{,}000) \approx 20$ steps only!

Note: We'll cover this in detail with Binary Search later

O(n log n) - Linearithmic Time

Definition: Combination of linear and logarithmic

Examples:

- Merge Sort
- Quick Sort (average case)
- Heap Sort

Characteristics:

- Better than $O(n^2)$
- Worse than $O(n)$
- Common in efficient sorting algorithms

Time Complexity Comparison Table

Complexity	n=10	n=100	n=1,000	n=10,000	Speed
O(1)	1	1	1	1	Excellent ✓✓✓
O(log n)	3	7	10	13	Excellent ✓✓✓
O(n)	10	100	1,000	10,000	Good ✓✓
O(n log n)	30	700	10,000	130,000	Good ✓✓
$O(n^2)$	100	10,000	1,000,000	100,000,000	Bad ✗
$O(n^3)$	1,000	1,000,000	10^9	10^{12}	Very Bad ✗✗
$O(2^n)$	1,024	10^{30}	10^{300}	∞	Terrible ✗✗✗

7. Practice Programs

Program 1: Single Loop

cpp

```
void printArray(int arr[], int n) {
   for(int i = 0; i < n; i++) {
      cout << arr[i] << " ";
   }
}
```

Your Turn:

- How many times does the loop run?
- Time Complexity?

<details> <summary>Answer</summary>

- Loop runs: **n times**
- Time Complexity: **Θ(n)**
- Best case: n
- Worst case: n
- Average case: n

</details>

Program 2: Nested Loop (Square)

cpp

```
void printPairs(int n) {
   for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         cout << i << "," << j << endl;
      }
   }
}
```

Your Turn:

- Outer loop runs? Inner loop runs?
- Total operations?
- Time Complexity?

<details> <summary>Answer</summary>

- Outer loop: **n times**
- Inner loop: **n times** (for each outer iteration)
- Total: **n × n = n²**
- Time Complexity: **Θ(n²)**

</details>

Program 3: Triangle Pattern

cpp

```
void printTriangle(int n) {
   for(int i = 1; i <= n; i++) {
      for(int j = 1; j <= i; j++) {
         cout << "* ";
      }
      cout << endl;
   }
}
```

Your Turn:

- Inner loop depends on outer loop!
- When i=1, inner runs 1 time
- When i=2, inner runs 2 times
- When i=n, inner runs n times
- Total operations?

<details> <summary>Answer</summary>

Total operations: 1 + 2 + 3 + ... + n = **n(n+1)/2**

Simplified: **$n^2/2 + n/2$**

Drop constants and lower terms: **n^2**

Time Complexity: **$\Theta(n^2)$**

Even though it's fewer operations than a square pattern, it's still quadratic growth!

</details>

Program 4: Constant Time

cpp

```
int getMiddle(int arr[], int n) {
   return arr[n/2];
}
```

Your Turn:

- How many operations regardless of n?

- Time Complexity?

<details> <summary>Answer</summary>

- Operations: **Always 1** (just array access)
- Time Complexity: **Θ(1)**

No loops, direct access!

</details>

Program 5: Linear Search

cpp

```
int search(int arr[], int n, int target) {
   for(int i = 0; i < n; i++) {
      if(arr[i] == target) {
         return i;  // Found!
      }
   }
   return -1;  // Not found
}
```

Your Turn:

- Best case?
- Worst case?
- Average case?

<details> <summary>Answer</summary>

Best case: Target at index 0

- Operations: **1**
- Time: **Θ(1)**

Worst case: Target at end or not found

- Operations: **n**
- Time: **Θ(n)**

Average case: Target in middle (on average)

- Operations: **n/2**
- Simplified: **n**
- Time: **Θ(n)**

</details>

Program 6: Triple Nested Loop

cpp

```
void print3D(int n) {
   for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         for(int k = 0; k < n; k++) {
            cout << "*";
         }
      }
   }
}
```

Your Turn:

- Total operations?
- Time Complexity?

<details> <summary>Answer</summary>

- Total: **n × n × n = n^3**
- Time Complexity: **Θ(n^3)**

Very slow for large n!

</details>

Program 7: Multiple Sequential Loops

cpp

```
void multipleLoops(int arr[], int n) {
   // Loop 1
   for(int i = 0; i < n; i++) {
```

```
    cout << arr[i];
  }

  // Loop 2
  for(int i = 0; i < n; i++) {
    cout << arr[i] * 2;
  }

  // Loop 3
  for(int i = 0; i < n; i++) {
    cout << arr[i] * 3;
  }
}
```

Your Turn:

- Total operations?
- Time Complexity?

<details> <summary>Answer</summary>

- Loop 1: **n operations**
- Loop 2: **n operations**
- Loop 3: **n operations**
- Total: **n + n + n = 3n**

Drop constants: **n**

Time Complexity: **Θ(n)**

Key Rule: Multiple sequential loops → Add them up, then drop constants!

</details>

Program 8: Loop with Division

cpp

```
int countDivisions(int n) {
```

```
    int count = 0;

    while(n > 1) {

        n = n / 2;

        count++;

    }

    return count;

}
```

Your Turn:

- How many times does loop run?
- Pattern?

<details> <summary>Answer</summary>

Pattern: n keeps halving

- Start: n
- After 1 iteration: n/2
- After 2 iterations: n/4
- After 3 iterations: n/8
- ...
- After k iterations: n/(2^k) = 1

Solving: $2^k = n \rightarrow k = \log_2(n)$

Time Complexity: **Θ(log n)**

This is logarithmic time!

</details>

8. Important Rules for Time Complexity

Rule 1: Drop Constants

Bad: $T(n) = 5n$ **Good:** $T(n) = n$

Bad: $T(n) = 3n^2 + 2$ **Good:** $T(n) = n^2$

Why? Constants don't affect growth rate for large n!

Rule 2: Drop Lower Order Terms

Bad: $T(n) = n^2 + n + 1$ **Good:** $T(n) = n^2$

Bad: $T(n) = n^3 + n^2 + n$ **Good:** $T(n) = n^3$

Why? Higher order term dominates!

When n = 1,000:

- n^3 = 1,000,000,000
- n^2 = 1,000,000 (1000× smaller, negligible!)
- n = 1,000 (1,000,000× smaller, negligible!)

Rule 3: Sequential Loops → Add

cpp

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

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


```

Total: n + n = 2n → O(n)

Rule 4: Nested Loops → Multiply

cpp

```
for(int i = 0; i < n; i++) {      // n times
   for(int j = 0; j < n; j++) {  // n times each
   }
}


```

Total: $n \times n = n^2 \rightarrow O(n^2)$

Rule 5: Different Variables → Keep Both

cpp

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

for(int j = 0; j < m; j++) { }


Time: O(n + m)  // Keep both!
```

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

Time: O(n × m) // Keep both!

Rule 6: Dependent Inner Loop

```
// Triangle pattern
for(int i = 0; i < n; i++) {
   for(int j = 0; j < i; j++) {  // Depends on i!
   }
}
```

Total: 1 + 2 + 3 + ... + n = n(n+1)/2 = **$O(n^2)$**

Still quadratic!

Quick Reference Card

Time Complexity Cheat Sheet

Code Pattern	Time Complexity	Name
Single statement	O(1)	Constant
One loop to n	O(n)	Linear
Two nested loops to n	$O(n^2)$	Quadratic
Three nested loops to n	$O(n^3)$	Cubic
Loop that divides/doubles	O(log n)	Logarithmic
Loop + divide/conquer	O(n log n)	Linearithmic

Best Practices

1. **Always analyze worst case first** (most common in interviews)
2. **Look for the dominant term** (highest power of n)
3. **Count nested loops** (multiplication)
4. **Identify early exits** (affects best case)
5. **Consider input size** (what is n?)

Common Mistakes to Avoid

❌ **Mistake 1:** Saying O(2n) instead of O(n) ✓ **Correct:** Drop the constant → O(n)

❌ **Mistake 2:** Saying $O(n^2 + n)$ instead of $O(n^2)$ ✓ **Correct:** Drop lower terms → $O(n^2)$

❌ **Mistake 3:** Forgetting to multiply nested loops ✓ **Correct:** Nested loops multiply

❌ **Mistake 4:** Thinking $O(n^2)$ is "better than" O(n) ✓ **Correct:** Smaller complexity = faster (O(n) is better)

9. Space Complexity

What is Space Complexity?

Definition:

Space Complexity is a function that describes how much memory (RAM) an algorithm uses as the input size increases.

Simple Definition:

"As input size n increases, how much extra memory does my program need?"

Time vs Space - The Key Difference

Aspect	Time Complexity	Space Complexity
Measures	Number of operations	Amount of memory
Resource	CPU time	RAM
Question	"How long to run?"	"How much memory needed?"
Cost	Processor cycles	Memory bytes

What Takes Memory?

Things that consume space:

1. **Variables**

```
cpp

int x = 5;          // Takes 4 bytes

double y = 3.14;    // Takes 8 bytes
```

1. **Arrays**

```
int arr[100];       // Takes 100 × 4 = 400 bytes

int matrix[n][n];   // Takes n² × 4 bytes
```

1. **Dynamic Data Structures**

```
cpp

vector<int> v(n);   // Takes n × 4 bytes

map<int, int> m;    // Depends on size
```

1. **Recursion Call Stack**

```
cpp

void recursion(int n) {
   if(n == 0) return;
   recursion(n-1);  // Each call uses stack space!
}
```

Two Types of Space

1. Input Space (Fixed Space)

Space taken by the **input** itself - **we DON'T count this!**

```
cpp

void processArray(int arr[], int n) {
   // arr[n] is INPUT
```

```
    // We don't count input in space complexity!
}
```

Why not? Because input size is given - we can't control it!

2. Auxiliary Space (Extra Space)

Extra space used **during** algorithm execution - **THIS is what we count!**

```
cpp
void processArray(int arr[], int n) {
    int sum = 0;      // Extra variable - COUNT this
    int temp[n];      // Extra array - COUNT this

    // Space Complexity = O(n)
}
```

Important Rule

Space Complexity = Auxiliary Space + Input Space

But typically when we say "Space Complexity", we mean **Auxiliary Space** only!

Space Complexity Examples

Example 1: O(1) - Constant Space

```
cpp
int sumArray(int arr[], int n) {
    int sum = 0;      // 1 variable
    for(int i = 0; i < n; i++) {  // 1 variable
        sum += arr[i];
    }
    return sum;
}
```

Space Analysis:

- sum: 1 integer (4 bytes)
- i: 1 integer (4 bytes)
- Total extra space: **8 bytes** (constant!)

Space Complexity: O(1) ✓

Key Point: No matter how large n is, we only use 2 variables!

Example 2: O(n) - Linear Space

cpp

```
void reverseArray(int arr[], int n) {
   int temp[n];     // NEW array of size n

   for(int i = 0; i < n; i++) {
      temp[i] = arr[n-1-i];
   }

   for(int i = 0; i < n; i++) {
      arr[i] = temp[i];
   }
}
```

Space Analysis:

- temp[n]: n integers (n × 4 bytes)
- i: 1 integer (4 bytes)
- Total: **n + 1** → Drop constant → **n**

Space Complexity: O(n) ✓

Example 3: O(n²) - Quadratic Space

cpp

```
void createMatrix(int n) {
   int matrix[n][n];  // 2D array
```

```
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            matrix[i][j] = i * j;
        }
    }
}
```

Space Analysis:

- matrix[n][n]: n × n integers ($n^2 \times 4$ bytes)

Space Complexity: $O(n^2)$ ✓

Example 4: O(1) - In-Place Algorithm

cpp

```
void bubbleSort(int arr[], int n) {
    for(int i = 0; i < n-1; i++) {
        for(int j = 0; j < n-i-1; j++) {
            if(arr[j] > arr[j+1]) {
                // Swap in place
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

Space Analysis:

- i, j, temp: 3 variables (constant)

- No extra arrays!

Space Complexity: O(1) ✓

Note: Time is O(n²), but space is O(1)!

This is called an **"in-place"** algorithm.

Example 5: Recursion Stack Space

cpp

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

What happens in memory?

Call Stack for factorial(5):

```
factorial(5)
  → factorial(4)
    → factorial(3)
      → factorial(2)
        → factorial(1)
          → factorial(0) → returns 1
        → returns 1
      → returns 2
    → returns 6
  → returns 24
→ returns 120
```

Maximum Stack Depth: n levels

Each level stores:

- Parameter n
- Return address
- Local variables

Space Complexity: O(n) (for recursion stack) ✓

Example 6: Multiple Arrays

```
void processData(int arr[], int n) {
    int copy1[n];    // O(n) space
    int copy2[n];    // O(n) space
    int result[n];   // O(n) space

    // Process...
}
```

Space Analysis:

- Total: n + n + n = **3n**
- Drop constant: **n**

Space Complexity: O(n) ✓

Time vs Space Trade-off

Often you can **trade time for space** or **space for time**!

Problem: Check if Array has Duplicates

Approach 1: Less Space, More Time ⏰📦

cpp

```
bool hasDuplicate(int arr[], int n) {
    for(int i = 0; i < n; i++) {
        for(int j = i+1; j < n; j++) {
            if(arr[i] == arr[j]) {
                return true;
            }
```

```
    }
  }
  return false;
}
```

Analysis:

- **Time:** O(n²) - nested loops
- **Space:** O(1) - only i, j variables

Trade-off: Slow but memory-efficient!

Approach 2: More Space, Less Time 📦⏰

cpp

```
bool hasDuplicate(int arr[], int n) {
  bool seen[10001] = {false};  // Extra array

  for(int i = 0; i < n; i++) {
    if(seen[arr[i]]) {
      return true;
    }
    seen[arr[i]] = true;
  }
  return false;
}
```

Analysis:

- **Time:** O(n) - single loop (much faster!)
- **Space:** O(10001) = O(1) - fixed array size

Trade-off: Fast but uses more memory!

When to Choose What?

Use less space:

- Mobile devices (limited RAM)
- Embedded systems
- Processing huge datasets

Use more space:

- Speed is critical
- Have plenty of RAM
- Real-time applications

Space Complexity Patterns

Pattern 1: No Extra Space (In-place)

Characteristics:

- Modifies input array directly
- Uses only a few variables
- Space: O(1)

cpp

```
void reverseInPlace(int arr[], int n) {
   for(int i = 0; i < n/2; i++) {
      int temp = arr[i];
      arr[i] = arr[n-1-i];
      arr[n-1-i] = temp;
   }
}
// Space: O(1) ✓
```

Pattern 2: Single Extra Array

Characteristics:

- Creates one array of size n
- Space: O(n)

cpp

```
void mergeSorted(int a[], int b[], int n) {
   int result[2*n];  // Extra array
```

```
    // Merge logic...
}

// Space: O(n) ✓
```

Pattern 3: 2D Arrays

Characteristics:

- Matrix or table
- Space: $O(n^2)$

```
cpp

void printPascal(int n) {
    int triangle[n][n];
    // Fill triangle...
}

// Space: O(n²) ✓
```

Pattern 4: Recursion Depth

Characteristics:

- Call stack grows with recursion
- Space: O(depth)

```
cpp

void printN(int n) {
    if(n == 0) return;
    cout << n;
    printN(n-1);  // Stack depth = n
}

// Space: O(n) for stack ✓
```

Common Space Complexities

Complexity	Description	Example
O(1)	Constant space	Few variables, in-place sorting
O(log n)	Logarithmic space	Binary search recursion stack
O(n)	Linear space	Extra array, linear recursion
O(n log n)	Linearithmic space	Merge sort (with optimization)
O(n^2)	Quadratic space	2D matrix, adjacency matrix
O(2^n)	Exponential space	Storing all subsets

What We Count vs What We Don't

✅ DO Count (Auxiliary Space):

1. **Extra variables** you create
2. **Extra arrays/data structures**
3. **Recursion call stack depth**
4. **Dynamic memory allocations**

❌ DON'T Count:

1. **Input array** itself (given to us)
2. **Output** (if required by problem)
3. **Code/instruction space**
4. **Compiler overhead**

Practice: Calculate Space Complexity

Problem 1

```
int findMax(int arr[], int n) {

    int max = arr[0];
```

```
    for(int i = 1; i < n; i++) {
        if(arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

<details> <summary>Answer</summary>

Variables: max, i (2 variables)

Space Complexity: O(1) ✓

No extra arrays, just constant variables!

</details>

Problem 2

cpp

```
void printTriangle(int n) {
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= i; j++) {
            cout << "* ";
        }
        cout << endl;
    }
}
```

Variables: i, j (2 variables)

Space Complexity: O(1) ✓

Just loops, no arrays created!

Note: Time is $O(n^2)$, but Space is O(1)!

</details>

Problem 3

cpp

```
int fibonacci(int n) {
   if(n <= 1) return n;
   return fibonacci(n-1) + fibonacci(n-2);
}
```

Recursion tree depth: Maximum n levels deep

Each call stores: parameters and return address

Space Complexity: O(n) ✓ (for call stack)

Note: Time is $O(2^n)$, Space is O(n)!

</details>

Problem 4

cpp

```
void create2DArray(int n) {
   int arr[n][n];
   for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
         arr[i][j] = i + j;
      }
   }
}
```

Array: arr[n][n] = n^2 elements

Space Complexity: $O(n^2)$ ✓

</details>

Problem 5

cpp

```
void processArray(int arr[], int n) {
   int temp1[n];
   int temp2[n];
   int temp3[n];

   // Some processing...
}
```

Arrays: 3 arrays of size n each

Total: n + n + n = 3n

Drop constant: n

Space Complexity: O(n) ✓

</details>

Time and Space Analysis Together

When analyzing algorithms, consider **both**!

Example: Bubble Sort

cpp

```
void bubbleSort(int arr[], int n) {
   for(int i = 0; i < n-1; i++) {
      for(int j = 0; j < n-i-1; j++) {
         if(arr[j] > arr[j+1]) {
            int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
         }
      }
```

```
    }
}
```

Complete Analysis:

- **Time Complexity:** O(n²) - nested loops
- **Space Complexity:** O(1) - in-place, constant variables

Trade-off: Slow time, but excellent space efficiency!

Example: Merge Sort (Simplified)

cpp

```
void mergeSort(int arr[], int n) {
    if(n <= 1) return;

    int mid = n/2;
    int left[mid];      // Extra space
    int right[n-mid];   // Extra space

    // Copy, sort recursively, merge...
}
```

Complete Analysis:

- **Time Complexity:** O(n log n) - much faster!
- **Space Complexity:** O(n) - needs extra arrays

Trade-off: Fast time, but uses more space!

Key Takeaways - Space Complexity

1. **Space complexity** measures extra memory used
2. **Count auxiliary space** (not input)
3. **Same notations apply:** O, Ω, Θ
4. **Recursion uses stack space** (often O(depth))
5. **In-place algorithms** use O(1) space
6. **Trade-offs exist:** Fast time vs less space

Summary

Key Takeaways

1. **Time Complexity** measures how operations grow with input size
2. **Space Complexity** measures how memory usage grows with input size
3. **We drop constants and lower terms** to focus on growth rate
4. **Three notations:**
   - **O (Big-O):** Upper bound (≤)
   - **Ω (Omega):** Lower bound (≥)
   - **Θ (Theta):** Exact bound (=)
5. **Common complexities (best to worst):**
   - $O(1) \rightarrow O(\log n) \rightarrow O(n) \rightarrow O(n \log n) \rightarrow O(n^2) \rightarrow O(n^3) \rightarrow O(2^n)$
6. **For practical use:**You can skip notation and just state complexity directly!
   - "Runs in n^2 time" is clear
   - "Uses O(n) extra space" is clear
   - "Best case: 1, Worst case: n" is clear
7. **Always consider both:** Time AND Space complexity!

Practice Exercises

Calculate **both time AND space complexity** for these programs:

1. Print all elements in array
2. Find sum of all pairs in array
3. Check if array is sorted
4. Count occurrences of each element
5. Reverse array using extra array
6. Reverse array in-place
7. Create multiplication table (2D array)
8. Calculate factorial recursively
9. Bubble sort
10. Copy array to new array

Solutions will be discussed in next lecture!

Algorithm Comparison Table

Algorithm	Time Complexity	Space Complexity	Notes
Linear Search	O(n)	O(1)	In-place
Bubble Sort	$O(n^2)$	O(1)	In-place
Print Array	O(n)	O(1)	Read-only
Copy Array	O(n)	O(n)	Extra array
Matrix Creation	$O(n^2)$	$O(n^2)$	2D space
Factorial (recursion)	O(n)	O(n)	Stack space

Additional Resources

- Practice more problems on arrays
- Analyze sorting algorithms next
- Study recursive time complexity later

Remember: Understanding WHY matters more than memorizing formulas!