Analyzing Code Systematically

A Methodology for Determining Complexity

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

What We Know So Far

Lesson 1: Empirical timing

- Measured actual running times
- Saw $O(n)$, $O(n \log n)$, and $O(n^2)$ in action

Lesson 2: Formal definitions

- Proved complexity bounds mathematically
- Understood Big-O, Ω, and Θ

Today: Systematic code analysis

- **How to look at any code and determine its complexity**
- A step-by-step methodology
- Common patterns to recognize

Recap: Why Do We Drop Constants and Lower-Order Terms?

The core idea behind Big-O

> **Key Idea:** Big-O describes how an algorithm scales as n gets large. Constants and lower-order terms become irrelevant at that scale.

Think about it this way: say one computer does an operation in 1 nanosecond and another does it in 100 nanoseconds. That's a constant factor of 100x — but if one algorithm is $O(n)$ and the other is $O(n^2)$, the $O(n)$ algorithm **will always win** eventually, no matter how slow the hardware is. Constants depend on the machine.

The growth rate depends on the **algorithm**. Big-O is about the algorithm.

Recap: Why Only the Highest-Order Term?

The dominant term swallows everything else

Consider $f(n) = 5n^3 + 200n^2 + 10000n + 999999$. It looks complicated, but let's see what happens as n grows:

n	$5n^3$	$200n^2$	$10000n$	999999	Total
10	5,000	20,000	100,000	999,999	1,124,999
1,000	5,000,000,000	200,000,000	10,000,000	999,999	$\approx$ 5,211,000,000
1,000,000	5×10^{18}	2×10^{14}	10^{10}	10^6	$\approx 5 \times 10^{18}$

By $n = 1{,}000{,}000$, the $5n^3$ term is **trillions of times** larger than everything else combined. The other terms don't matter. So we just say $f(n) = O(n^3)$.

The Analysis Methodology

Four Steps to Analyze Any Code

1. **Identify the input size** – What is n?
2. **Count operations** – How many times does each line execute?
3. **Express as a function** – Write $T(n)$
4. **Find the dominant term** – Drop constants and lower-order terms

We'll practice this on many examples today

Foundation: The RAM Model of Computation

What Counts as "Constant Time"?

Random Access Machine (RAM) Model:

- Standard model for analyzing algorithms
- Each "simple operation" takes 1 time unit

Operations we consider $O(1)$:

- **Arithmetic:** $+$, -, *, /, % on integers/floats
- **Comparison:** $<$, $>$, $==$, $!=$, $\leq$, $\geq$
- **Memory access:** array indexing (a[i]), pointer dereference
- **Assignment:** $x = y$
- **Logical ops:** &&, ||, !
- **Bitwise ops:** &, |, ^, $<<$, $>>$

Caveat: Assumes numbers fit in a word (typically 32 or 64 bits)

What's NOT O(1)?

These operations are NOT constant time:

- **String operations:** concatenation, comparison (depends on string length)
- **Array operations:** copying entire arrays, shifting elements
- **I/O operations:** printing a number takes $O(\log n)$ time
  - Printing 5 $\rightarrow$ 1 digit $\rightarrow$ O(1)
  - Printing 12345 $\rightarrow$ 5 digits $\rightarrow$ O(5)
  - Printing $n \rightarrow O(\log n)$ digits
- **Exponentiation:** a^b is NOT O(1) if b is large
  - Naïve approach: $O(b)$ multiplications
  - Fast exponentiation: $O(\log b)$
- **Big integer arithmetic:** when numbers exceed word size (e.g., 1000-digit numbers)

For this course: We assume operations on integers/floats that fit in 64 bits, and we avoid I/O in our complexity analysis.

Why Does Printing Take $O(\log n)$ Time?

Key insight: The number of digits grows logarithmically with the value.

Examples:

Number n	Digits
5	1
50	2
500	3
5,000	4
50,000	5
1,000,000	7

Mathematical fact:

A number n has exactly

$$\lfloor \log_{10} n \rfloor + 1$$

digits in base 10.

Why?

- 1-digit: $1 \leq n < 10 = 10^1$
- 2-digit: $10 \leq n < 100 = 10^2$
- 3-digit: $100 \leq n < 1000 = 10^3$
- d-digit: $10^{d-1} \leq n < 10^d$

Printing implications:

- Printing requires outputting each digit
- Must process $\Theta(\log n)$ digits
- Therefore: printing n takes $O(\log n)$ time

Pattern 1: Simple Loop

The Most Common Pattern

```
int sum = 0;
for (int i = 0; i < n; i++) {
    sum += i;  // O(1) arithmetic
}
```

Analysis:

1. Input size: n (size of the range)
2. Loop executes: n times (i = 0, 1, 2, ..., n-1)
3. Each iteration: $O(1)$ work
   - Comparison `i < n`: $O(1)$ (RAM model)
   - Increment `i++`: $O(1)$ (arithmetic)
   - Addition `sum += i`: $O(1)$ (arithmetic)
4. Total: $T(n) = n \times O(1) = O(n)$

Variation: Different Loop Bounds

```
// Loop to n/2
int sum = 0;
for (int i = 0; i < n/2; i++)
    {
     sum += i;
}
```

Analysis:

- Executes $n/2$ times
- $T(n) = n/2 = O(n)$

Still $\Theta(n)$!

```
// Loop to 100
int sum = 0;
for (int i = 0; i < 100; i++)
    {
     sum += i;
}
```

Analysis:

- Executes 100 times
- $T(n) = 100 = O(1)$

$\Theta(1)$ - constant!

Key insight: Constants don't affect Big-O. Loop to $n/2$, $3n$, or $n - 100$ all give $O(n)$.

Pattern 2: Independent Nested Loops

When inner loop doesn't depend on outer

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

Analysis:

1. Outer loop: n iterations
2. Inner loop: n iterations (for each outer iteration)
3. Total iterations: $n \times n = n^2$
4. Each iteration: $O(1)$ work

Result: $\Theta(n^2)$

Variation: Different Sized Loops

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

Analysis:

- Outer loop: n iterations
- Inner loop: m iterations (for each outer)
- Total: $n \times m$ iterations

Result: $\Theta(n \cdot m)$

Special case: If $m = n$, then $\Theta(n \cdot n) = \Theta(n^2)$

Pattern 3: Dependent Nested Loops

Inner loop bound depends on outer variable

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {  // j starts at i
        sum += i + j;
    }
}
```

Analysis:

- When $i = 0$: inner runs n times
- When $i = 1$: inner runs $n - 1$ times
- When $i = 2$: inner runs $n - 2$ times

Pattern 3: Dependent Nested Loops

Inner loop bound depends on outer variable

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {  // j starts at i
        sum += i + j;
    }
}
```

Analysis:

- When $i = 0$: inner runs n times
- When $i = 1$: inner runs $n - 1$ times
- When $i = 2$: inner runs $n - 2$ times
- Total: $n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$

Pattern 3: Dependent Nested Loops

Inner loop bound depends on outer variable

```
int sum = 0;
for (int i = 0; i < n; i++) {
    for (int j = i; j < n; j++) {  // j starts at i
        sum += i + j;
    }
}
```

Analysis:

- When $i = 0$: inner runs n times
- When $i = 1$: inner runs $n - 1$ times
- When $i = 2$: inner runs $n - 2$ times
- Total: $n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$

Result: $\frac{n^2+n}{2} = \Theta(n^2)$

Common Dependent Loop Patterns

Triangular (upper):

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

Sum: $n + (n-1) + \cdots + 1$

Result: $\Theta(n^2)$

Triangular (lower):

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

Sum: $1 + 2 + \cdots + n$

Result: $\Theta(n^2)$

Key insight: Both triangular patterns give $\Theta(n^2)$, just with different constants.

What About Triple Nesting?

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

Analysis:

- Outer: n iterations
- Middle: n iterations per outer $= n \times n$ total
- Inner: n iterations per middle $= n \times n \times n$ total

Result: $\Theta(n^3)$ **Pattern:** k independent nested loops $= \Theta(n^k)$

Pattern 4: Logarithmic - Halving

The signature of binary search

```
int i = n;
while (i > 1) {
    // do some O(1) work
    i = i / 2;  // halve each time
}
```

Analysis:

- Start: $i = n$
- After 1 iteration: $i = n/2$
- After 2 iterations: $i = n/4$
- After k iterations: $i = n/2^k$
- Stop when $i \leq 1$, so $n/2^k = 1 \Rightarrow k = \log_2 n$

Result: $\Theta(\log n)$

Variation: Doubling Instead of Halving

```
for (int i = 1; i < n; i = i * 2) {
    // do some O(1) work
}
```

Analysis:

- Iteration 0: $i = 1$
- Iteration 1: $i = 2$
- Iteration 2: $i = 4$
- Iteration 3: $i = 8$
- Iteration k: $i = 2^k$
- Stop when $2^k \geq n$, so $k = \log_2 n$

Result: $\Theta(\log n)$

Logarithm Bases in Big-O

They all look the same asymptotically

Mathematical fact: $\log_a n = \frac{\log_b n}{\log_b a}$

This means:

$$\log_2 n = \frac{\log_{10} n}{\log_{10} 2} = \frac{1}{0.301} \log_{10} n \approx 3.32 \log_{10} n$$

But 3.32 is just a constant!

Therefore: $\log_2 n = O(\log_{10} n)$ and $\log_{10} n = O(\log_2 n)$

In Big-O: We just write $O(\log n)$ without specifying the base

The base doesn't matter asymptotically!

Real Example: Binary Search

```
int binarySearch(int arr[], int n, int target) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;  // not found
}
```

Analysis:

- Each iteration halves the search space
- $n \to n/2 \to n/4 \to \cdots \to 1$
- Takes $\log_2 n$ iterations $\Rightarrow$ **Result:** $\Theta(\log n)$

Pattern 5: Sequential Code Sections

When code has multiple independent parts

```
// Section 1: O(n)
int sum1 = 0;
for (int i = 0; i < n; i++) {
    sum1 += i;
}
// Section 2: O(n^2)
int sum2 = 0;
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        sum2 += i + j;
    }
}
```

Analysis: $T(n) = O(n) + O(n^2)$

Rule: When adding complexities, **keep only the largest term**

Why Keep Only the Largest Term?

Example: $T(n) = 5n + 3n^2$

For large n:

n	$5n$	$3n^2$	Ratio
10	50	300	6:1
100	500	30,000	60:1
1,000	5,000	3,000,000	600:1
10,000	50,000	300,000,000	6,000:1

The $5n$ term becomes **insignificant** compared to $3n^2$

General rule: $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$

Common Mistake: Don't Add Big-O Terms!

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

✗ **WRONG:** "Two loops, so $O(n) + O(n) = O(2n) = O(n)$"

✓ **CORRECT:** "Two sequential $O(n)$ sections $= O(n)$"

Why wrong thinking doesn't matter here: You got the right answer, but for the wrong reason. The correct reasoning is: $\max(O(n), O(n)) = O(n)$.

Be careful with: $O(n) + O(n^2) \neq O(n^2 + n)$ ... it's just $O(n^2)$!

Critical Distinction: Nested vs Sequential

NESTED (Multiplicative):

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

Inner runs n times
FOR EACH outer iteration
$= n \times n = O(n^2)$

SEQUENTIAL (Maximum):

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

Each runs n times
INDEPENDENTLY
$= \max(n, n) = O(n)$

Nested = multiply, Sequential = take maximum

Practice: Analyze This Code

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j = j * 2) {
        sum += i + j;
    }
}
```

Questions:

1. How many times does the outer loop run?
2. How many times does the inner loop run?
3. What is the total complexity?

Practice: Analyze This Code

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j = j * 2) {
        sum += i + j;
    }
}
```

Questions:

1. How many times does the outer loop run?
2. How many times does the inner loop run?
3. What is the total complexity?

Answer:

- Outer: n times
- Inner: $\log n$ times (doubling pattern)
- Total: $n \times \log n = \Theta(n \log n)$

Practice: Analyze This Code

```
int i = 1;
while (i < n) {
    for (int j = 0; j < i; j++) {
        sum += i + j;
    }
    i = i * 2;
}
```

Hint: The outer loop is logarithmic, but inner loop size changes!

Practice: Analyze This Code

```
int i = 1;
while (i < n) {
    for (int j = 0; j < i; j++) {
        sum += i + j;
    }
    i = i * 2;
}
```

Hint: The outer loop is logarithmic, but inner loop size changes!

Answer:

- Outer iterations: $i = 1, 2, 4, 8, \ldots, n$ ($\log n$ values)
- Inner work per iteration: $1 + 2 + 4 + 8 + \cdots + n = 2n - 1$
- Total: $\Theta(n)$

(Geometric series: $1 + 2 + 4 + \cdots + 2^k = 2^{k+1} - 1$*)*

Geometric Series: The Derivation

A clever algebraic trick

Goal: Find a closed form for $S = 1 + r + r^2 + r^3 + \cdots + r^k$

The trick: Multiply both sides by r and subtract!

$$\begin{aligned} S &= 1 + r + r^2 + r^3 + \cdots + r^k \\ rS &= \quad r + r^2 + r^3 + \cdots + r^k + r^{k+1} \end{aligned}$$

Geometric Series: The Derivation

A clever algebraic trick

Goal: Find a closed form for $S = 1 + r + r^2 + r^3 + \cdots + r^k$

The trick: Multiply both sides by r and subtract!

$$\begin{aligned} S &= 1 + r + r^2 + r^3 + \cdots + r^k \\ rS &= \quad r + r^2 + r^3 + \cdots + r^k + r^{k+1} \end{aligned}$$

Subtracting the second from the first:

$$\begin{aligned} S - rS &= 1 + \cancel{r + r^2 + \cdots + r^k} - \cancel{r + r^2 + \cdots + r^k} - r^{k+1} \\ S - rS &= 1 - r^{k+1} \\ S(1 - r) &= 1 - r^{k+1} \end{aligned}$$

Geometric Series: The Derivation

A clever algebraic trick

Goal: Find a closed form for $S = 1 + r + r^2 + r^3 + \cdots + r^k$

The trick: Multiply both sides by r and subtract!

$$\begin{aligned} S &= 1 + r + r^2 + r^3 + \cdots + r^k \\ rS &= \quad r + r^2 + r^3 + \cdots + r^k + r^{k+1} \end{aligned}$$

Subtracting the second from the first:

$$\begin{aligned} S - rS &= 1 + \cancel{r + r^2 + \cdots + r^k} - \cancel{r + r^2 + \cdots + r^k} - r^{k+1} \\ S - rS &= 1 - r^{k+1} \\ S(1 - r) &= 1 - r^{k+1} \end{aligned}$$

Therefore:

$$S = \frac{1 - r^{k+1}}{1 - r} = \frac{r^{k+1} - 1}{r - 1} \quad \text{for } r \neq 1$$

Geometric Series: Algorithm Analysis Applications

Why we care about this formula

General formula:

$$1 + r + r^2 + r^3 + \cdots + r^k = \frac{r^{k+1} - 1}{r - 1}$$

Most common case in CS (powers of 2): When $r = 2$:

$$1 + 2 + 4 + 8 + \cdots + 2^k = \frac{2^{k+1} - 1}{2 - 1} = 2^{k+1} - 1$$

Concrete examples:

- $1 + 2 + 4 + 8 = 15 = 2^4 - 1$
- $1 + 2 + 4 + 8 + 16 = 31 = 2^5 - 1$
- Sum to n: $1 + 2 + \cdots + n = 2n - 1$

Where you'll see it:

- Binary tree node counts
- Doubling loop analysis
- Divide-and-conquer algorithms
- Merge sort complexity

Best, Worst, and Average Case

Not all inputs are equal

```
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) return i;
    }
    return -1;
}
```

Analysis depends on where target is:

- **Best case:** Target is first element $\Rightarrow \Theta(1)$
- **Worst case:** Target is last or not present $\Rightarrow \Theta(n)$
- **Average case:** Target at random position $\Rightarrow \Theta(n)$

Which do we report?

Usually **worst case**, because:

- Gives a guarantee (never worse than this)
- Often matches average case anyway

When Average Case Analysis Matters

Examples where average case is important:

- **Quicksort:**
  - Worst case: $O(n^2)$ (rare, with bad pivot choices)
  - Average case: $O(n \log n)$ (typical behavior)
  - We use it because average case is good
- **Hash tables:**
  - Worst case: $O(n)$ (all keys hash to same bucket)
  - Average case: $O(1)$ (with good hash function)
  - We use them because average case is excellent

Rule of thumb:

- If worst case is rare and average case is good, use average case
- Otherwise, report worst case for safety

Summary of Common Patterns

Code Pattern	Complexity	Example
Simple loop to n	$O(n)$	`for (i=0; i<n; i++)`
Nested independent loops	$O(n^2)$	Two `for` loops to n
Triple nested	$O(n^3)$	Three `for` loops to n
Dependent nested (triangular)	$O(n^2)$	`for j=i to n`
Halving/doubling	$O(\log n)$	`i = i/2` or `i = i*2`
Binary search	$O(\log n)$	Halve search space
Nested: outer n, inner $\log n$	$O(n \log n)$	Sorting algorithms
Sequential sections	$O(\max)$	Take largest term

Remember: These are patterns, not rules. Always analyze the specific code!

Common Mistakes to Avoid

1. ✗ **"Two loops means $O(n^2)$"**
   - Only if nested! Sequential loops are $O(n)$
2. ✗ **Forgetting what's inside the loop**
   - If loop body is $O(n)$, total is loop count $\times$ $O(n)$
3. ✗ **Adding Big-O terms incorrectly**
   - $O(n) + O(n^2) = O(n^2)$, not $O(n^2 + n)$
4. ✗ **Calling any growth "exponential"**
   - n^2 is quadratic/polynomial, not exponential
   - Exponential means 2^n, e^n, etc.
5. ✗ **Assuming constants don't matter**
   - They don't affect Big-O, but they matter in practice!

Try These Before Next Class

1. What is the complexity of selection sort? (Hint: find minimum, swap, repeat)
2. Analyze this code:

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

3. Why is merge sort $O(n \log n)$? (We'll prove this next class)
4. Find a code snippet with $O(n^{1.5})$ complexity

Hint for #2: Inner loop runs i^2 times. Sum: $0^2 + 1^2 + 2^2 + \cdots + n^2 = ?$

Assignment 3 Solutions: Problems 1–2

Problem 1: Warm-up

```
for (int i = 0; i < 3*n; i++)
```

(a) Outer loop executes $3n$ times

(b) No inner loop

(c) $T(n) = 3n$

(d) $\boxed{O(n)}$

Constant multiples are dropped in Big-O.

Problem 2: Nested Loops

```
for (int i = 0; i < n; i++)
  for (int j = 0; j < 5; j++)
```

(a) Outer loop: n times

(b) Inner loop: 5 times (constant, independent of n)

(c) $T(n) = n \cdot 5 = 5n$

(d) $\boxed{O(n)}$

A constant inner loop does not increase the degree.

Assignment 3 Solutions: Problems 3–4

Problem 3: Triangular Pattern

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

(a) Outer loop: n times

(b) Inner loop: $i + 1$ times on iteration i

(c) $T(n) = \sum_{i=0}^{n-1}(i+1) = \sum_{i=1}^{n} i = \dfrac{n(n+1)}{2}$

(d) $\boxed{O(n^2)}$

Problem 4: Halving Loop

```
int k = n;
while (k > 0)
    k = k / 3;
```

(a) Loop executes $\lfloor \log_3 n \rfloor + 1$ times

(b) No inner loop

(c) $T(n) = \lfloor \log_3 n \rfloor + 1$

(d) $\boxed{O(\log n)}$

Logarithm base does not affect Big-O class: $\log_3 n = \log_3 2 \cdot \log_2 n$, *constant factor drops.*

Assignment 3 Solutions: Problem 5

Problem 5: Sequential Sections

Section 1

```
for (i = 0; i < n; i
    ++)
```

Executes n times.
$T_1(n) = n$

$\Rightarrow O(n)$

Section 2

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

i doubles each step:
$1, 2, 4, \ldots < n$
Executes $\lfloor \log_2 n \rfloor$ times.

$\Rightarrow O(\log n)$

Section 3

```
for (i = 0; i < n; i
    ++)
  for (j = 0; j < n;
       j++)
```

$n \cdot n$ iterations.

$\Rightarrow O(n^2)$

Combining sequential sections — take the dominant term:

$$T(n) = n + \log n + n^2 \Rightarrow \boxed{O(n^2)}$$

Assignment 3 Solutions: Problems 6–7

Problem 6: Nested with Doubling

```
for (int i = 0; i < n; i++)
  for (int j = 1; j < n;
       j = j*2)
```

(a) Outer loop: n times

(b) Inner loop: j doubles each step $\Rightarrow \lfloor \log_2 n \rfloor$ times, *independent of i*

(c) $T(n) = n \cdot \lfloor \log_2 n \rfloor$

(d) $\boxed{O(n \log n)}$

Problem 7: Reverse Triangular

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

(a) Outer loop: n times

(b) Inner loop: $n - i$ times on iteration i

(c) $T(n) = \sum_{i=0}^{n-1} (n - i) =$
$n + (n-1) + \cdots + 1 = \dfrac{n(n+1)}{2}$

(d) $\boxed{O(n^2)}$

Same sum as Problem 3, reversed.

Assignment 3 Solutions: Problems 8–9

Problem 8: Triple Nesting

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
```

(a) Outer loop: n times

(b) Middle loop: n times

(c) Inner loop: n times

(d) $T(n) = n \cdot n \cdot n = n^3$

(e) $\boxed{O(n^3)}$

Problem 9: Different Bounds

```
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    for (k = 0; k < n; k++)
```

(a) Outer loop: n times

(b) Middle loop: m times

(c) Inner loop: n times

(d) $T(n, m) = n \cdot m \cdot n = n^2 m$

(e) $\boxed{O(n^2 m)}$

Cannot simplify further without knowing the relationship between n and m.

Assignment 3 Solutions: Problem 10 (Challenge)

Problem 10: Nested with Doubling

```
for (int i = 1; i < n; i = i*2)    // outer: i = 1, 2, 4, 8,
    ..., 2^(k-1)
    for (int j = 0; j < i; j++)     // inner: runs i times
```

(a, b) Outer loop runs $k = \lfloor \log_2 n \rfloor$ times. On iteration t (where $t = 0, 1, \ldots, k-1$), $i = 2^t$, so inner loop runs 2^t times.

(c) Total operations:

$$T(n) = \sum_{t=0}^{\lfloor \log_2 n \rfloor - 1} 2^t = 1 + 2 + 4 + \cdots + 2^{k-1}$$

This is a geometric series. Applying $\sum_{t=0}^{k-1} 2^t = 2^k - 1$:

$$T(n) = 2^{\lfloor \log_2 n \rfloor} - 1 < n$$

(d) $\boxed{O(n)}$ **Intuition** Despite the nested loop, the total work is bounded by n because the outer loop's iterations are geometrically sparse, each one doubles, so the sum converges.