Lecture 3

The Anatomy of Recursion

Structure before Complexity

HEAD

```
int function(int n)
```

BODY

```
{
    if (n == 0)
        return 1;

    return ... function(n-1) ...;
}
```

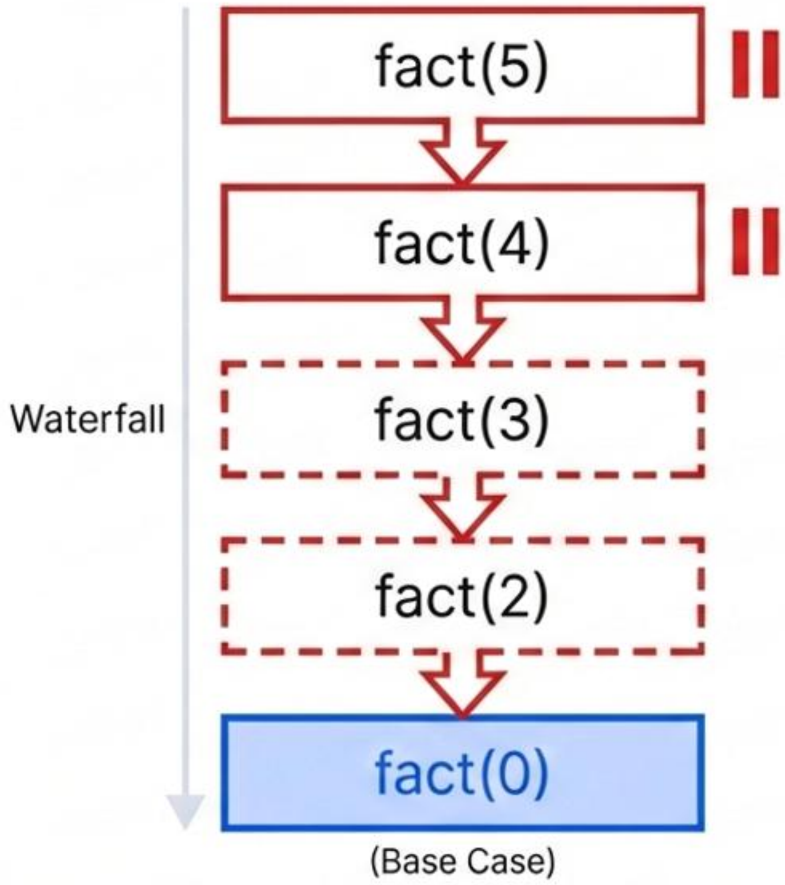
The Exit Strategy (Base Case)
Prevents infinite loops.

The Trap (Recursive Step)
Calls the Head again. This is where T(n) lives.

To calculate Time Complexity $T(n)$, we must quantify the cost of the Body.
It is a sum of the exit cost and the trap cost.

Visualizing the Execution Flow

Recursion is a paused state.




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

$$T(5) = \text{Current Step} + \text{Time(Children)}$$

When a function calls itself, it does not finish immediately. It pauses execution and opens a new instance. The total time is the sum of the work done now plus the work done by all future children.

Deriving the General Formula

The Code-to-Math Rosetta Stone

`if (n==0)`
(Comparison)

$$T(n) = \text{Non-Recursive} + \text{Recursive}$$

`if (n==0) (Comparison)

`return 1` (Arithmetic)

Constant Time ($) or Linear Time ($n$)

`func(new_size)`

Unknown Time (T(new_size)$)

We cannot calculate the final number yet. We can only express the relationship between the current problem size (n) and the next problem size.

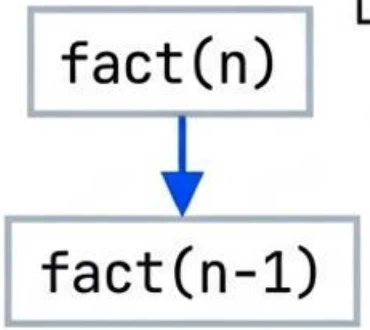
The Rules of Multiplicity

Math Operations vs. Function Calls

The Math Trap

`return 4 * fact(n-1);`

$1 \times T(n-1)$

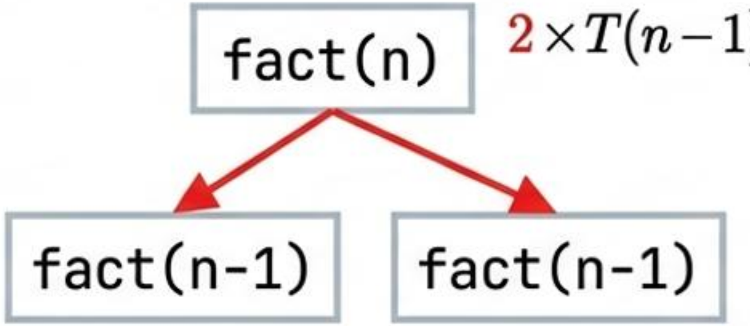


Multiplication is a constant math operation. It creates only one recursive branch.

The Execution Reality

`return fact(n-1) + fact(n-1);`

$2 \times T(n-1)$



Calling the function twice doubles the work. The compiler runs the first, then the second.

Code Snippet

```c
int fact(int n) {
  if (n == 0)
    return 1;
  return 4 * fact(n - 1);
}

// Code Trap

int fact_multi_call(int n) {
  if (n == 0)
    return 1;
  return
    fact_multi_call(n - 1) +
    fact_multi_call(n - 1);
}

// Execution Reality
```

Case 1: The Standard Decrement

Linear Reduction

Code Space

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

Time Space

Constant overhead

$$T(n) = T(n-1) + 1$$

Recursive Work

Case 2: The Constant Multiplier

Ignore the Coefficients

Code Space

```
int fact(int n) {
  if(n==0) return 1;
  return 4 * fact(n-1);
}
```

This is just math. It costs **constant time** (+1). It does **NOT** multiply the recursive term T.

Time Space

```
T(n) = T(n-1) + 1
```

Case 3: The Constant Addition

Arithmetic is Cheap

Code Space

```
int fact(int n) {
  if(n==0) return 1;
  return fact(n-1) + 1;
}

// Base case omitted
return fact(n-1) + 1;
```

Time Space

```
T(n) = T(n-1) + 1
```

Adding 1 to the result is a single operation. The complexity lies in the call, not the arithmetic.

Case 4: Division (Binary Search Logic)

Input Size Transformation

Code Space

```
int fact(int n) {
  if (n <= 1) return 1;
  return fact(n/2) + 1;
}
```

Time Space

$$T(n) = T(n/2) + 1$$

The **problem size** is cut in half.
The recursive term reflects the n
ew size passing through the logic.

Case 5: Multiple Calls (Same Size)

Branching Factor

Code Space

```
int fact(int n) {
  // Base case omitted
  return fact(n/2)
           +
        fact(n/2)
}
```

Time Space

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

Two calls, same size. The compiler executes the first, THEN the se second. Work is doubled.

Case 6: Multiple Calls (Different Sizes)

Asymmetric Recursion

Complete Function

```
int fact(int n) {
  if (n <= 1)
    return 1;
  return fact(n/2)+fact(3*n/2);
}
```

Code Space

```
// Base case omitted
return fact(n/2) + fact(3*n/2);
```

Time Space

$$T(n) = T(n/2) + T(3n/2) + 1$$

Distinct calls with distinct sizes cannot be combined. We must sum the time for each unique path.

Case 7: Loop + Recursion

Linear Overhead

Code Space

```
int recursiveWithLoop(int n) {
  // Base case omitted for brevity
  for(int i=0; i<n; i++) { ... }
```

Iterates n times

```
  return fact(n-1) + fact(n-2);
}
```

Time Space

$$T(n) = T(n-1) + T(n-2) + n$$

The non-recursive preparation work involves a loop. It is no longer constant (+1), it is linear (+n).

The Asymptotic Question

We have the Equation. We need the Complexity.

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

$$T(n) = T(n-1) + T(n-2) + n$$

$$T(n) = 4T(n/2) + O(1)$$

$$T(n) = 2T(n/2) + O(n^2)$$

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

$O(1)$

$O(\log n)$

$O(n)$

$O(n \log n)$

$O(n^2)$

$O(2^n)$

Recurrence relations ($T(n)$) are just the description of the problem. To find the Big O classification, we must apply advanced solver methods to these equations.

Solving Recurrence methods

The Template: Standardizing Recurrence

The Master Method requires fitting the problem to a specific algebraic form.

Branching Factor (Subproblems)

$a \geq 1$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Division Factor (Input Shrinkage)

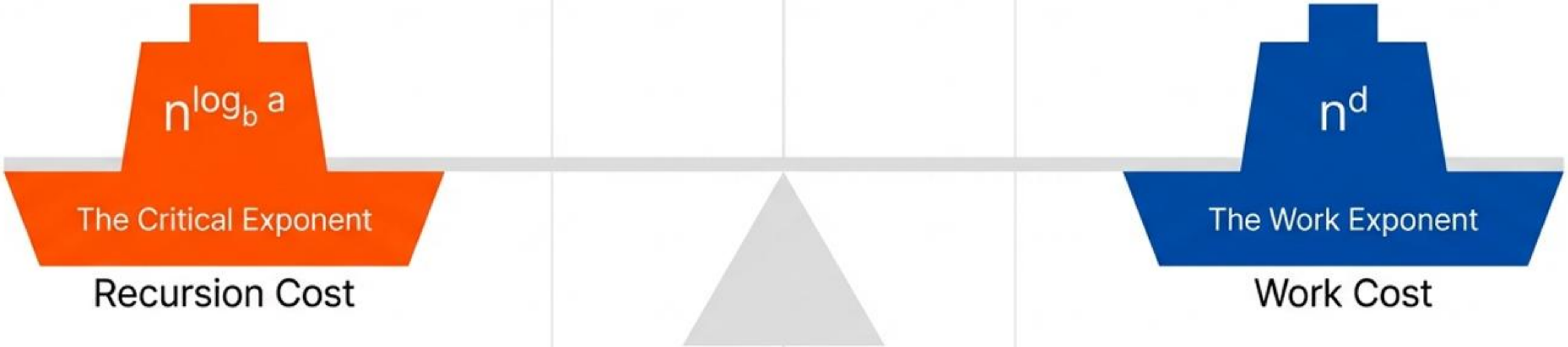
$b > 1$

Work Cost (Merge/Divide Steps)

Crucial Step: Express work as a power of n

$$f(n) \approx n^d$$

The Balance Beam: Recursion vs. Work



Case 1: Recursion Heavy

$\log_b a > d$

Result: $\Theta(n^{\log_b a})$

Case 2: Balanced

$\log_b a = d$

Result: $\Theta(n^d \log n)$

Extended Case:
If balanced but f(n) has $\log^k n$,
Result is $\Theta(n^d \log^{k+1} n)$

Case 3: Work Heavy

$\log_b a < d$

Result: $\Theta(n^d)$

Case 1: Recursion Dominates

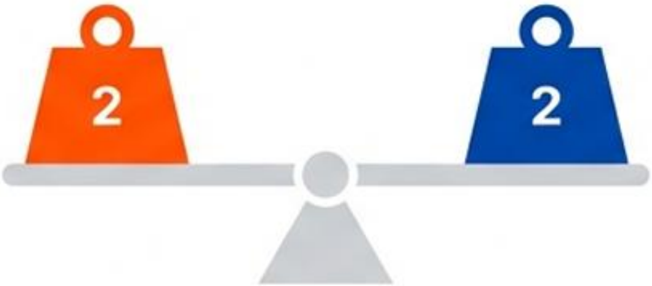
$T(n) = 4T(n/2) + n$

Step 1 (Identify)	Step 2 (Calculate Critical)	Step 3 (Compare)
• $a = 4$ • $b = 2$ • $d = 1$ (from n^1)	$\log_2 4 = 2$	 $2 > 1$
		Result: $\Theta(n^2)$

Since the recursion exponent is larger, the complexity is driven by the depth of the tree.

Case 2: The Balanced State

$$T(n) = 4T(n/2) + n^2$$

Step 1 (Identify)	Step 2 (Calculate Critical)	Step 3 (Compare)
$a = 4$ $b = 2$ $d = 2$ (from n^2)	$\log_2 4 = 2$	 $2 = 2$

Result: $\Theta(n^2 \log n)$

When weights are equal, we multiply the work term by log n.

Case 3: Work Dominates

$$T(n) = 4T(n/2) + n^3$$

Step 1 (Identify)	Step 2 (Calculate Critical)	Step 3 (Compare)
• $a = 4$ • $b = 2$ • $d = 3$ (from n^3)	$\log_2 4 = 2$	 $2 < 3$

Result: $\Theta(n^3)$

The work done at the root node is so heavy it overshadows the recursion cost.

Handling Roots & Fractions

$$T(n) = 2T(n/8) + \sqrt[3]{n}$$

$\sqrt[3]{n} \longrightarrow n^{(1/3)}$

Standard Form Conversion

Step 1 (Identify)	Step 2 (Calculate Critical)	Step 3 (Compare)
• $a = 2$ • $b = 8$ • $d = 1/3$	$\log_8 2 = 1/3$ $(2^3 = 8)$	 $1/3 = 1/3$

Result: $\Theta(n^{1/3} \log n)$

When weights are equal, we multiply the work term by log n.

Composite Exponents

$T(n) = 4T(n/2) + n^2 \sqrt{n}$

$n^2 \cdot n^{0.5} = n^{2.5}$ **$d = 2.5$**

Step 1 (Identify)	Step 2 (Calculate Critical)	Step 3 (Compare)
• $a = 4$ • $b = 2$	$\log_2 4 = 2$	 $2 < 2.5$

Result: $\Theta(n^{2.5})$ or $\Theta(n^2 \sqrt{n})$

Hidden Coefficients & The Zero Power

$T(n) = T(2n/3) + 1$

Step 1 (Identify)

b is the reciprocal = **3/2**

$T(n) = T(2n/3) + 1$

a = 1

Constant 1 = **n^0**
Hence **d = 0**

Step 2 (Calculate Critical)

$\log_{(1.5)} 1 = 0$

Step 3 (Compare)

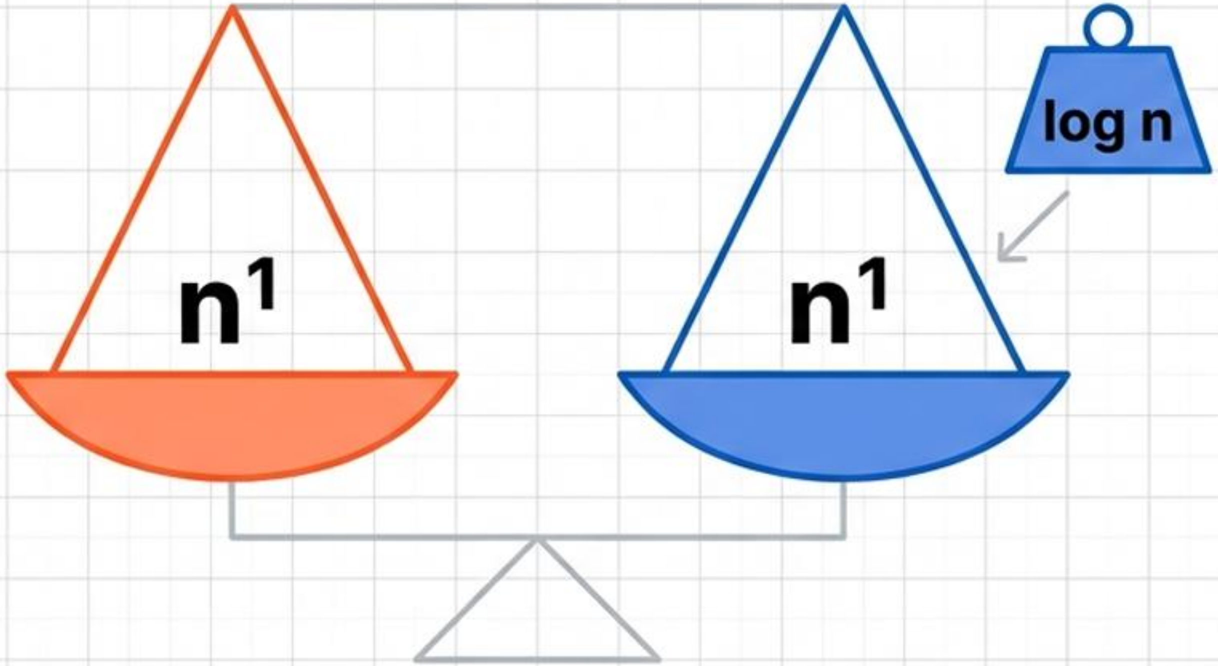
$0 = 0$

Result: $\Theta(n^0 \log n) = \Theta(\log n)$

This represents Binary Search complexity.

The Extended Master Theorem

$T(n) = 2T(n/2) + n \log n$



Base powers match (1=1), but work has extra log factor.

The Rule

If Balanced AND f(n) contains $\log^k n$:

Add 1 to the log power.

Execution

Current log power: $k = 1$

New log power: $k + 1 = 2$

Result: $\Theta(n \log^2 n)$

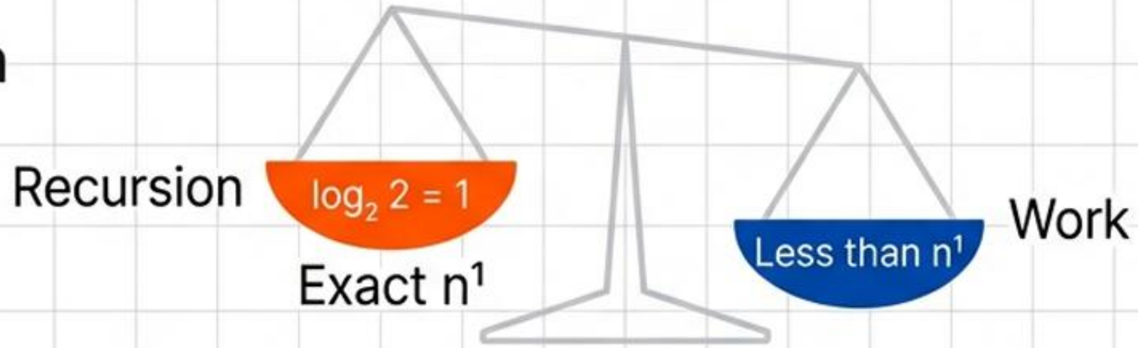
Approximations & Bounding

$$T(n) = 2T(n/2) + n / \log n$$

Problem Statement

n / log n is not a polynomial n^d. The standard method fails.

The Comparison

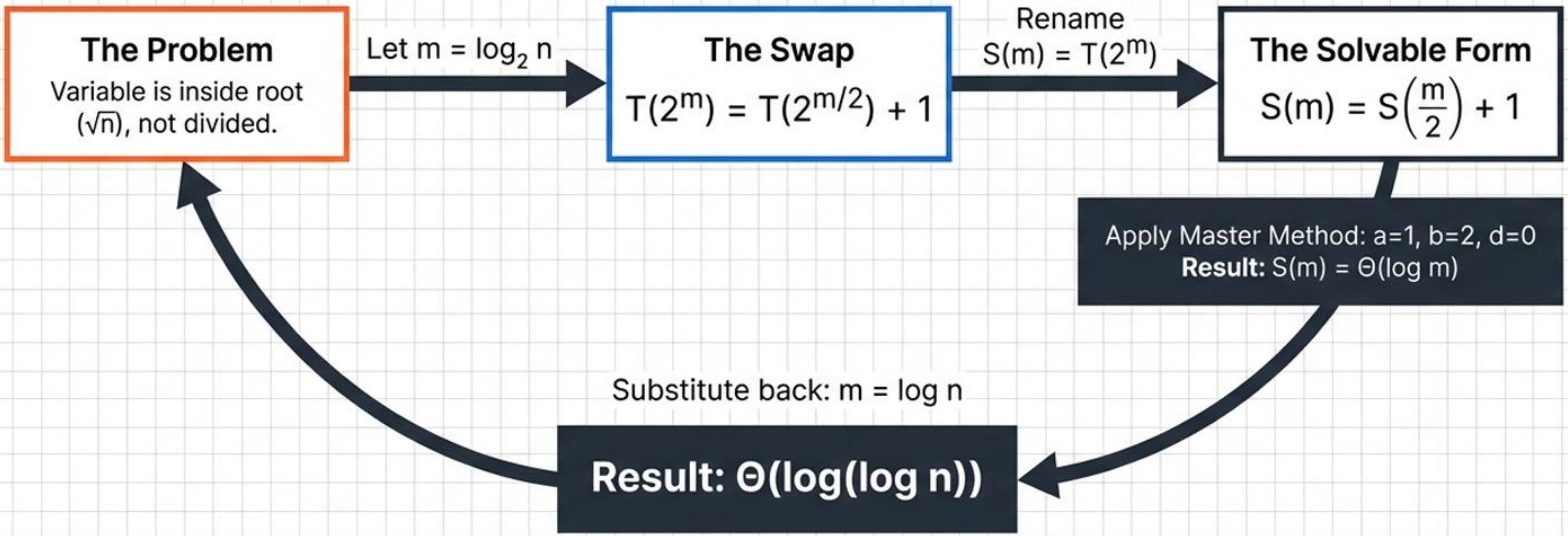


Verdict: Left Side (Orange) is heavier because 1 is strictly greater than the bounded work.

Result: $\Theta(n)$

Advanced Technique: Variable Substitution

$$T(n) = T(\sqrt{n}) + 1$$



Questions

$$T(n) = 2\,T\left(\frac{n}{2}\right) + \log n$$

$$T(n) = T\left(\frac{n}{3}\right) + n \log n$$

$$T(n) = 4\,T(n/2) + n^2/\lg n$$