Chapter 2: Algorithm Complexity

Algorithms

Time and space | Best / worst / average | Big O, Ω, Θ | Growth classes

Loop analysis | Recurrences | Master Theorem | Examples f1–f6

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/algorithms/slides/Chapter_02_Algorithm_Complexity.html

Learning Outcomes

Concepts

Distinguish time and space complexity, and best-, worst- and average-case analysis.

Notation

State the definitions of Big O, Ω and Θ; prove a bound by finding the constants c and n_0.

Loop analysis

Count the steps of single, nested and multiplicative loops with summation formulas (f1, f2, f6).

Recursion

Set up a recurrence for recursive code and solve it by expansion, a recursion tree or the Master Theorem (f3, f4, f5).

Throughout the chapter we work with six short functions (`f1` – `f6`), each representing a different growth class (n, n^3, 2^n, n, n log n, log* n).

Algorithm Complexity

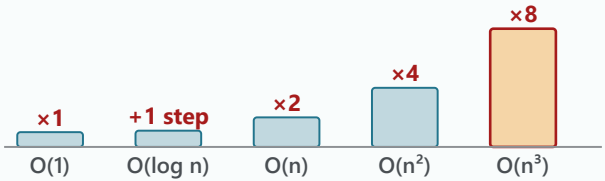
- **Complexity theory** measures an algorithm's **use of resources**: time and memory.
- Resource use is written as a function of the **input size n**: T(n), S(n).
- The goal is not to measure seconds but to see **how** the resource needs **grow** as the input grows.

Time complexity

The number of **basic operations** the algorithm performs: assignments, comparisons, arithmetic, array accesses.

Space complexity

Memory used beyond the input: variables, auxiliary arrays, the recursion stack.

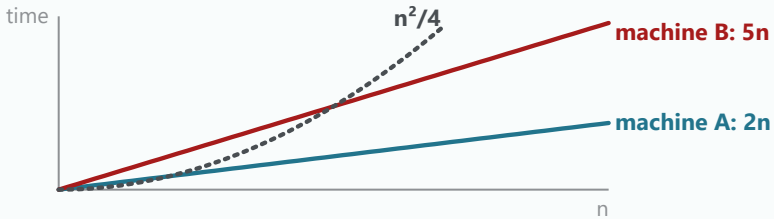


The key question: what happens to the running time when the input size doubles? Does it stay the same, double, quadruple, or get squared?

Why Timing Is Not Enough

```
long start = System.currentTimeMillis();
linearSearch(arr, 10001);  // 10^8 elements
long end = System.currentTimeMillis();
System.out.println("Elapsed: "
    + (end - start) + " ms");
```

- This code times a linear search over an array of 10^8 elements.
- The result depends on the **machine**, the processor, the JIT compiler, the cache and the current load.
- A single measurement **does not tell** us what happens as the input grows.



Experimental measurement

Gives real times but depends on the machine; the code must already be written.

Asymptotic analysis

Gives the step count as a function of n; machine-independent and possible before writing code.

Counting Steps: `linearSearch`

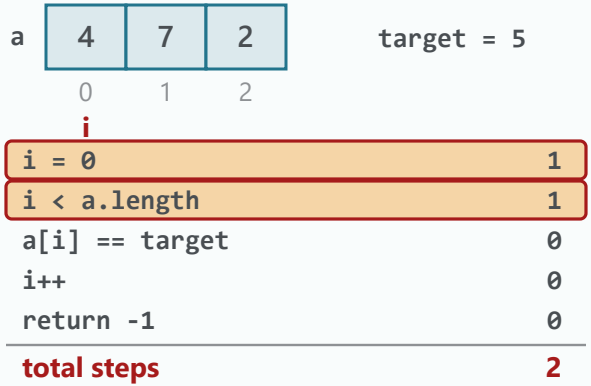
```
for (int i = 0; i < a.length; i++) {
  if (a[i] == target)
    return i;
}
return -1;
```

Operation	Worst-case count
`i = 0`	1
`i < a.length`	n + 1
`a[i] == target`	n
`i++`	n
`return -1`	1
Total T(n)	**3n + 3**

- Each basic operation counts as **1 unit** (the RAM model). Worst case: the target is not in the array.
- The coefficient **3** and the **+3** in 3n + 3 depend on the machine; what does not change is the **linear growth in n** → T(n) = Θ(n).

linearSearch: Tracing the Step Count

```
1 for (int i = 0; i < a.length; i++) {
2   if (a[i] == target)
3     return i;
4 }
5 return -1;
```



a = [4, 7, 2], target = 5 (not in the array: worst case).

`i = 0` is assigned and `0 < 3` is checked: 2 steps.

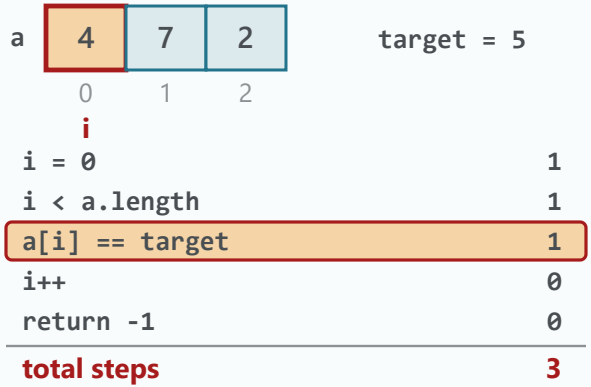
Step 1 / 8

VARIABLES

`i = 0` `target = 5` `steps = 2`

linearSearch: Tracing the Step Count

```
1 for (int i = 0; i < a.length; i++) {
▶   if (a[i] == target)
3     return i;
4 }
5 return -1;
```



a[0] == target? 4 ≠ 5. One comparison so far.

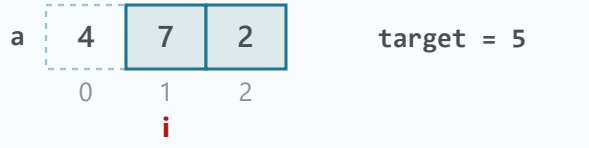
Step 2 / 8

VARIABLES

`i = 0` `target = 5` `steps = 3`

linearSearch: Tracing the Step Count

```
1 for (int i = 0; i < a.length; i++) {
2   if (a[i] == target)
3     return i;
4 }
5 return -1;
```



`i = 0`	1
`i < a.length`	2
`a[i] == target`	1
`i++`	1
`return -1`	0
total steps	**5**

`i++` → i = 1, then `1 < 3` is checked: the loop continues.

Step 3 / 8

VARIABLES

`i = 1` `target = 5` `steps = 5`

linearSearch: Tracing the Step Count

```
1 for (int i = 0; i < a.length; i++) {
2   if (a[i] == target)
3     return i;
4 }
5 return -1;
```



i = 0	1
i < a.length	2
a[i] == target	2
i++	1
return -1	0
total steps	**6**

a[1] == target? 7 ≠ 5. Two comparisons so far.

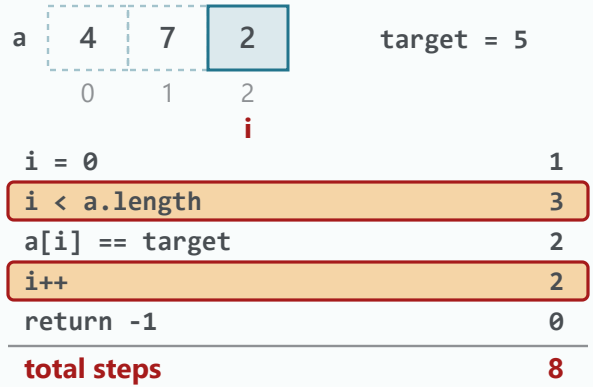
Step 4 / 8

VARIABLES

`i = 1` `target = 5` `steps = 6`

linearSearch: Tracing the Step Count

```
1 for (int i = 0; i < a.length; i++) {
2   if (a[i] == target)
3     return i;
4 }
5 return -1;
```



`i++` → i = 2, then `2 < 3` is checked: the loop continues.

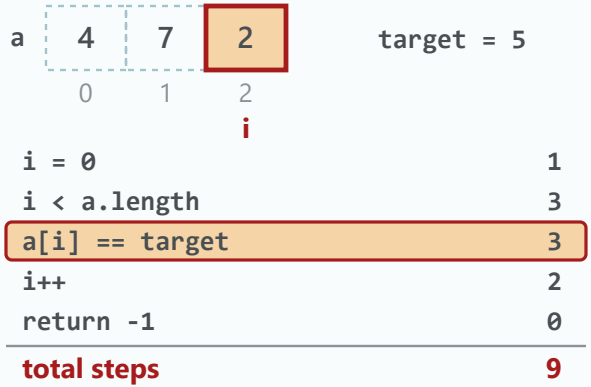
Step 5 / 8

VARIABLES

`i = 2` `target = 5` `steps = 8`

linearSearch: Tracing the Step Count

```
1 for (int i = 0; i < a.length; i++) {
▶   if (a[i] == target)
3     return i;
4 }
5 return -1;
```



a[2] == target? 2 ≠ 5. Three comparisons so far.

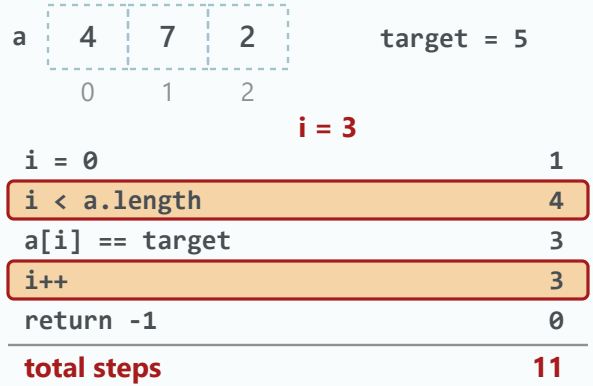
Step 6 / 8

VARIABLES

`i = 2` `target = 5` `steps = 9`

linearSearch: Tracing the Step Count

```
1 ▶ for (int i = 0; i < a.length; i++) {
2     if (a[i] == target)
3         return i;
4 }
5 return -1;
```



`i++` → i = 3. `3 < 3` is false: the condition was checked n + 1 = **4 times** and the loop ends.

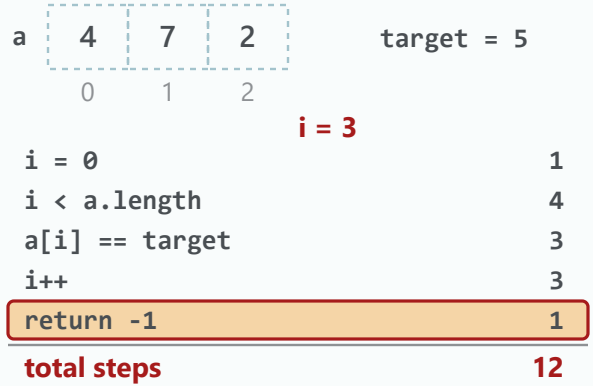
Step 7 / 8

VARIABLES

`i = 3` `target = 5` `steps = 11`

linearSearch: Tracing the Step Count

```
1 for (int i = 0; i < a.length; i++) {
2   if (a[i] == target)
3     return i;
4 }
▶ return -1;
```



`return -1`. **12** steps in total = 3n + 3 (n = 3). A constant number (3) of operations per element: Θ(n).

Step 8 / 8

VARIABLES

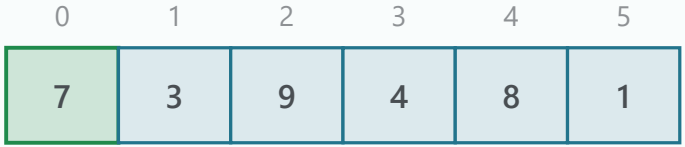
`i = 3` `target = 5` `steps = 12`

Best, Worst and Average Case

Best case
The fewest steps, on the most favorable input. A lower bound; on its own it says little.

Worst case
A **guarantee** that holds for every input: the algorithm never takes longer.

Average case
The expected number of steps under a probability distribution over inputs.



target 7 → 1 comparison

Best case: the target is the first element; one comparison suffices → Θ(1).

Best, Worst and Average Case

Best case
The fewest steps, on the most favorable input. A lower bound; on its own it says little.

Worst case
A **guarantee** that holds for every input: the algorithm never takes longer.

Average case
The expected number of steps under a probability distribution over inputs.



target 5 → 6 comparisons, not found

Worst case: the target is absent (or last); n = 6 comparisons → Θ(n).

Best, Worst and Average Case

Best case

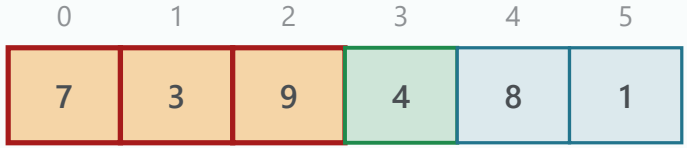
The fewest steps, on the most favorable input. A lower bound; on its own it says little.

Worst case

A **guarantee** that holds for every input: the algorithm never takes longer.

Average case

The expected number of steps under a probability distribution over inputs.



target 4 → 4 comparisons

Average case: if the target is equally likely to be in any position, the average is (1 + 2 + … + n)/n = (n + 1)/2 = 3.5 comparisons → still Θ(n).

Space Complexity

Space complexity usually refers to **auxiliary space**: the input itself is not counted. In recursion every active call keeps a **stack frame**.

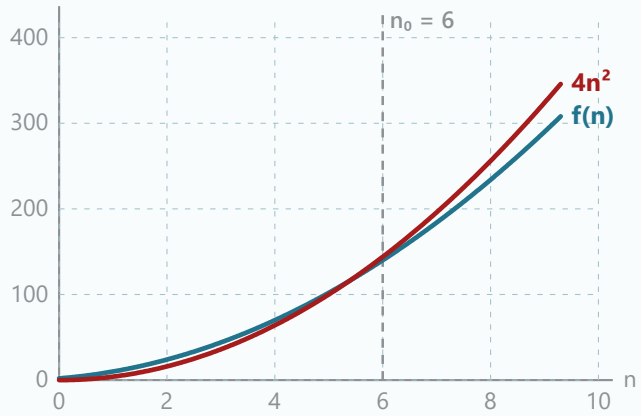
Code	Extra space	Space	Time
`f1` : single loop	only `x` and `i`	$\Theta(1)$	$\Theta(n)$
creating `new int[n]`	auxiliary array of n elements	$\Theta(n)$	$\Theta(n)$
`f3` : f3(n−1) + f3(n−1)	stack depth n (at most n frames at once)	$\Theta(n)$	$\Theta(2^n)$
`f4` : f4(n/2) + f4(n/2)	stack depth $\approx \log_2 n$	$\Theta(\log n)$	$\Theta(n)$
Merge sort	auxiliary array for merging	$\Theta(n)$	$\Theta(n \log n)$

Time and space can grow differently. f3 makes an exponential number of calls, but since the calls finish one after another, at most n frames are on the stack at any time.

Big-O Notation

$$f(n) = O(g(n)) \Leftrightarrow \exists\ c > 0, n_0 : 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

- For large n, f **never exceeds** a constant multiple of g: g is an **upper bound**.
- Small n ($n < n_0$) do not matter; constant factors are absorbed into c.
- Example: $3n^2 + 5n + 2 \leq 4n^2$ for all $n \geq 6 \rightarrow$ **$O(n^2)$**.

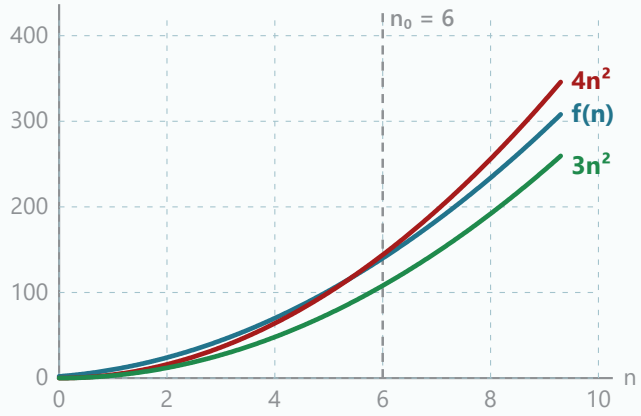


"Big O means the worst case" is not quite right. Big O is a kind of **upper bound**; it can be applied to the worst, best or average case alike. The two get confused because it is usually quoted for worst-case time.

Big Omega (Ω) and Big Theta (Θ)

Notation	Meaning	Definition ($\exists$ c, $n_0 > 0$, for all $n \geq n_0$)	Analogy
f = O(g)	upper bound	$f(n) \leq c \cdot g(n)$	f ≤ g
f = Ω(g)	lower bound	$f(n) \geq c \cdot g(n)$	f ≥ g
f = Θ(g)	tight bound	$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$	f = g

- $f = \Theta(g) \Leftrightarrow f = O(g)$ **and** $f = \Omega(g)$.
- $3n^2 + 5n + 2$ is both $O(n^2)$ and $O(n^3)$, but it is **Θ** only of n^2.
- Linear search: worst case Θ(n), best case Θ(1); O(n) and Ω(1) over all inputs.
- The **case** (which input) and the **bound** (O, Ω, Θ) are separate ideas. Insertion sort is $\Theta(n^2)$ in the worst case and Θ(n) in the best, so no single Θ covers all inputs. Merge sort, by contrast, is Θ(n log n) on every input.

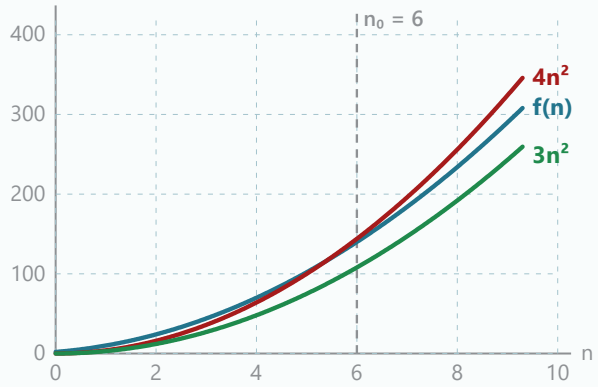


Example: $3n^2 + 5n + 2 = \Theta(n^2)$

Upper bound: $3n^2 + 5n + 2 \leq c_2\, n^2$

Try $c_2 = 4$: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0 \Leftrightarrow n \geq 5.37$. So $\mathbf{n_0 = 6}$.

Step 1 / 4



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

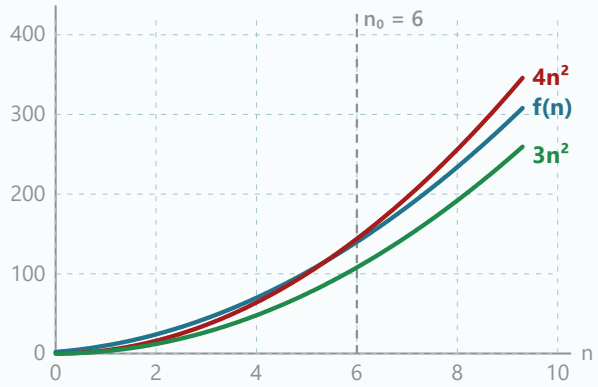
Upper bound: $3n^2 + 5n + 2 \leq c_2\, n^2$

Try $c_2 = 4$: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0 \Leftrightarrow n \geq 5.37$. So $\mathbf{n_0 = 6}$.

Check: $n = 5 \rightarrow 102 > 100$ ✗

$n = 6 \rightarrow 140 \leq 144$ ✓

The inequality fails at n = 5 and holds from n = 6 on (the left side grows by 6n + 8, the right side by 8n + 4).



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

$$\text{Upper bound: } 3n^2 + 5n + 2 \leq c_2\, n^2$$

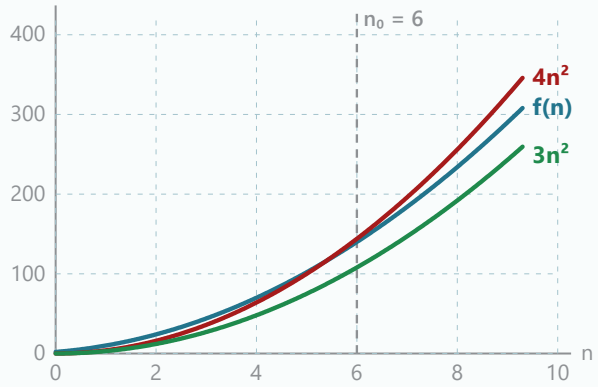
Try $c_2 = 4$: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0 \Leftrightarrow n \geq 5.37$. So $\mathbf{n_0 = 6}$.

$$\text{Check: } n = 5 \to 102 > 100 \;\text{✗}$$
$$n = 6 \to 140 \leq 144 \;\text{✓}$$

The inequality fails at n = 5 and holds from n = 6 on (the left side grows by 6n + 8, the right side by 8n + 4).

$$\text{Lower bound: } 3n^2 + 5n + 2 \geq 3n^2$$

With $c_1 = 3$ the inequality holds for every $n \geq 1$, since $5n + 2 > 0$.



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

$$\text{Upper bound: } 3n^2 + 5n + 2 \leq c_2\, n^2$$

Try $c_2 = 4$: $3n^2 + 5n + 2 \leq 4n^2 \Leftrightarrow n^2 - 5n - 2 \geq 0 \Leftrightarrow n \geq 5.37$. So $\mathbf{n_0 = 6}$.

$$\text{Check: } n = 5 \rightarrow 102 > 100 \ \text{✗}$$
$$n = 6 \rightarrow 140 \leq 144 \ \text{✓}$$

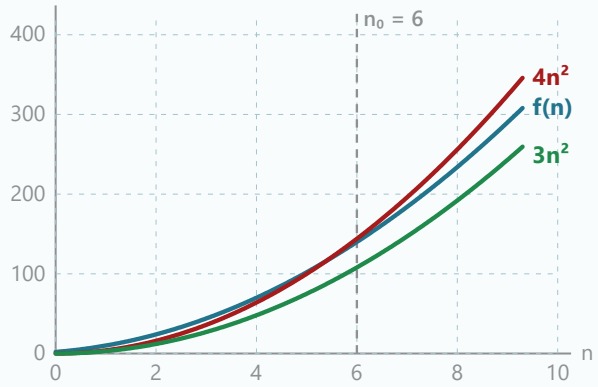
The inequality fails at $n = 5$ and holds from $n = 6$ on (the left side grows by $6n + 8$, the right side by $8n + 4$).

$$\text{Lower bound: } 3n^2 + 5n + 2 \geq 3n^2$$

With $c_1 = 3$ the inequality holds for every $n \geq 1$, since $5n + 2 > 0$.

$$3n^2 \leq f(n) \leq 4n^2,\ n \geq 6$$

$c_1 = 3$, $c_2 = 4$, $n_0 = 6$ satisfy the definition: **$f(n) = \Theta(n^2)$**. The constants are not unique; $c_2 = 10$, $n_0 = 1$ also works.



Rules of Asymptotic Arithmetic

Drop constant factors

$c \cdot f(n) = \Theta(f(n))$: $5n = \Theta(n)$, $n^2/3 = \Theta(n^2)$.

Drop lower-order terms

A polynomial keeps its highest degree: $4n^3 + 100n^2 + 7 = \Theta(n^3)$.

Sum rule (sequential code)

$O(f) + O(g) = O(\max(f, g))$: a $\Theta(n)$ loop + a $\Theta(n^2)$ loop = $\Theta(n^2)$.

Product rule (nested code)

$O(g)$ work inside a loop that runs n times: $O(n \cdot g)$. Check whether the inner work is the same on every iteration.

The log base does not matter

$\log_a n = \log_b n / \log_b a$, a constant factor: $O(\log_2 n) = O(\log_{10} n) = O(\log n)$.

The exponent base does matter

$3^n = O(2^n)$ is **false**, since $3^n/2^n = 1.5^n \to \infty$. Unlike logarithms, changing the base of an exponential is not a constant factor.

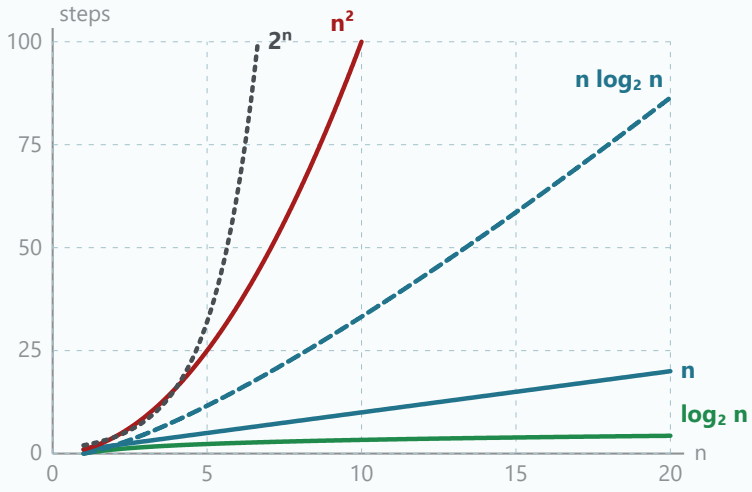
Order of growth: $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < n!$

Common Growth Classes

Class	Name	Typical example	When n doubles	In this chapter
$O(1)$	Constant	printing the first element, array access	unchanged	—
$O(\log n)$	Logarithmic	binary search, fast exponentiation, Euclid	+1 step	f6 (upper bound)
$O(\sqrt{n})$	Square root	primality test (trial division)	≈ 1,41×	—
$O(n)$	Linear	linear search, maximum, array sum	2×	f1, f4
$O(n \log n)$	Linearithmic	merge sort, heapsort, quicksort (average)	a bit more than 2×	f5
$O(n^2)$	Quadratic	bubble, selection, insertion sort	4×	—
$O(n^3)$	Cubic	standard matrix multiplication	8×	f2
$O(2^n)$	Exponential	all subsets, naive recursive Fibonacci	squared	f3
$O(n!)$	Factorial	all permutations	blows up even faster	—

The classes get slower from top to bottom. Polynomial classes (n^k) are usually considered "efficient", exponential ones "inefficient". The true class of f6 is $\log^* n$, which grows even more slowly than $\log n$.

Growth Curves



- For small n the curves are close; at n = 4, n^2 and 2^n are equal (16).
- **2^n** passes 100 steps at n ≈ 6.6, **n^2** after n = 10.
- **n log n** stays close to linear; **log n** is almost flat.
- The extremes drift further apart: at n = 10, n! reaches 3.6 million, while √n is only 10 at n = 100.
- Asymptotic analysis cares about the order of the curves for **large n**.

Growth in Numbers: Calculator

n = 1000 | 10 | 100 | 10^4 | 10^6 | n × 2

f(n)	operations	time at 10^9 ops/s	relative
1	1	1 ns	
$\log_2 n$	10	10 ns	
$\sqrt{n}$	31.6	31.6 ns	
n	1,000	1 µs	
$n \log_2 n$	9,965.8	10 µs	
n^2	1,000,000	1 ms	
n^3	$1 \cdot 10^9$	1 s	
2^n	$1.1 \cdot 10^{301}$	≈ 10^{284} years	
n!	$4 \cdot 10^{2567}$	≈ 10^{2551} years	

n = **1,000** · bar: digits in the operation count (log scale) · red: over 1 hour

With "n × 2", the n column doubles, n^2 grows 4×, n^3 8×; $\log_2 n$ grows by only 1, while 2^n is squared. A faster computer buys a constant factor; a better algorithm changes the growth class.

LOOP ANALYSIS

f1: Single Loop → O(n)

```java
public int f1(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    x++;
  }
  return x;
}
```

- **Initialization:** `int x = 0;` sets x to 0; constant time, O(1).
- **Loop:** `for (int i = 0; i < n; i++)` runs i from 0 to n − 1: **n times**.
- **Increment:** `x++;` runs once per iteration; O(1).

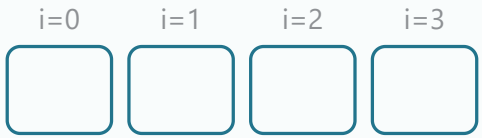
$$T(n) = O(1) + n \cdot O(1) = O(n)$$

The loop runs n times and its body takes constant time, so the total time complexity is **Θ(n)**. The function returns `x = n`, so the return value is the step count itself: f1(16) = 16, f1(32) = 32.

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
▶   int x = 0;
3   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ count: 0

`x` is set to zero. The loop has not started yet.

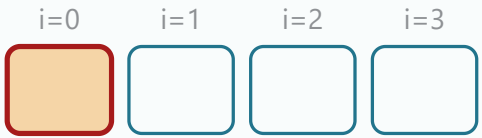
Step 1 / 11

VARIABLES

`n = 4` `x = 0` `i = –`

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ count: 0

i = 0: condition 0 < 4 is true → enter the body.

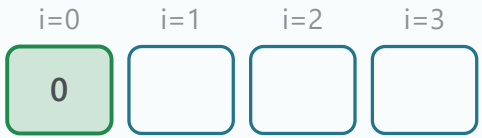
VARIABLES

n = 4 | x = 0 | i = 0

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ count: 1

`x++` → x = 1. This is iteration 1.

Step 3 / 11

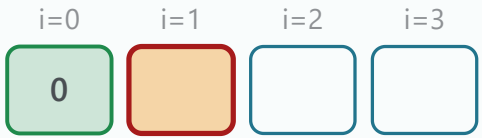
VARIABLES

`n = 4` `x = 1` `i = 0`

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ count: 1

i = 1: condition 1 < 4 is true → enter the body.

Step 4 / 11

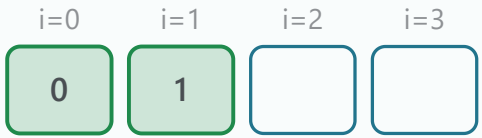
VARIABLES

n = 4 | x = 1 | i = 1

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ count: 2

`x++` → x = 2. This is iteration 2.

Step 5 / 11

VARIABLES

`n = 4` `x = 2` `i = 1`

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ count: 2

i = 2: condition 2 < 4 is true → enter the body.

Step 6 / 11

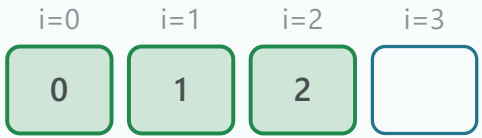
VARIABLES

n = 4 | x = 2 | i = 2

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ count: 3

`x++` → x = 3. This is iteration 3.

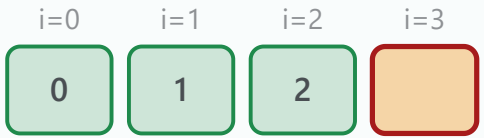
Step 7 / 11

VARIABLES

`n = 4` `x = 3` `i = 2`

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ count: 3

i = 3: condition 3 < 4 is true → enter the body.

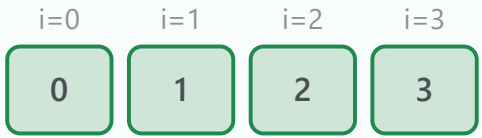
VARIABLES

n = 4 | x = 3 | i = 3

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     x++;
5   }
6   return x;
7 }
```



x++ count: 4

`x++` → x = 4. This is iteration 4.

Step 9 / 11

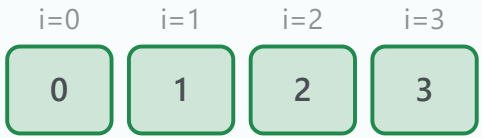
VARIABLES

`n = 4` `x = 4` `i = 3`

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
4     x++;
5   }
6   return x;
7 }
```



x++ count: 4

i = 4: condition 4 < 4 is false → the loop ends. The condition was checked n + 1 = 5 times.

Step 10 / 11

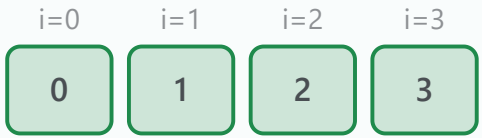
VARIABLES

n = 4 | x = 4 | i = 4

LOOP ANALYSIS

f1(4): Tracing the Code

```
1 public int f1(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
4     x++;
5   }
▶   return x;
7 }
```



x++ count: 4

`return x` → 4. The body ran exactly n = 4 times: Θ(n).

Step 11 / 11

VARIABLES

n = 4 | x = 4 | i = –

LOOP ANALYSIS

f2: Nested Loop → $O(n^3)$

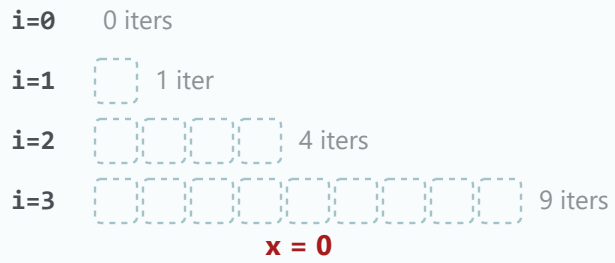
```
public int f2(int n) {
  int x = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i * i; j++) {
      x++;
    }
  }
  return x;
}
```

- **Initialization:** `int x = 0;` constant time, O(1).
- **Outer loop:** i from 0 to n − 1: **n times**.
- **Inner loop:** j from 0 to i · i − 1: **i^2 times**.
- The number of inner iterations **depends on the current value of i**, so we cannot just say "n · n"; we have to **add up** the iteration counts.

Pitfall: seeing two nested loops and automatically saying $O(n^2)$ is wrong. Here the inner bound is i^2, so the result is of order **n^3**.

f2(4): Tracing the Code

```
1 public int f2(int n) {
▶   int x = 0;
3   for (int i = 0; i < n; i++) {
4     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
8   return x;
9 }
```



`f2(4)` is called, `x = 0`. The gray boxes are the inner-loop iterations to come: i^2 boxes in row i.

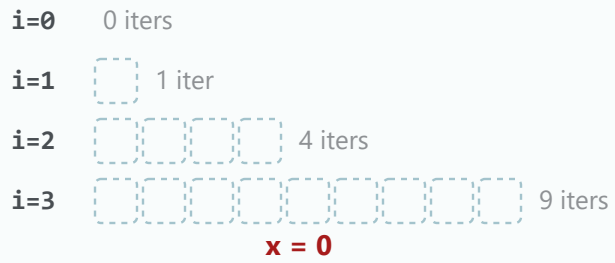
Step 1 / 9

VARIABLES

`n = 4` `i = –` `j = –` `x = 0`

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
▶   for (int i = 0; i < n; i++) {
▶     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
8   return x;
9 }
```



i = 0: the inner condition `0 < 0 · 0` is false at once; the inner loop runs **zero** times.

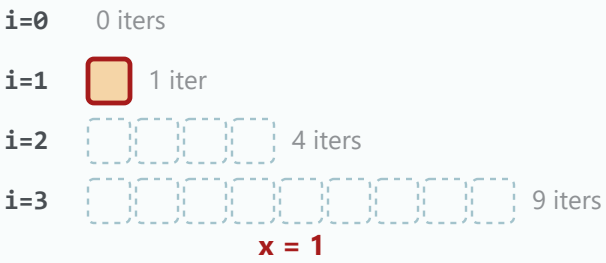
Step 2 / 9

VARIABLES

`n = 4` `i = 0` `j = 0` `x = 0`

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     for (int j = 0; j < i * i; j++) {
▶       x++;
6     }
7   }
8   return x;
9 }
```



i = 1: the inner loop runs while j < 1·1 = 1. First iteration (j = 0): x = 1.

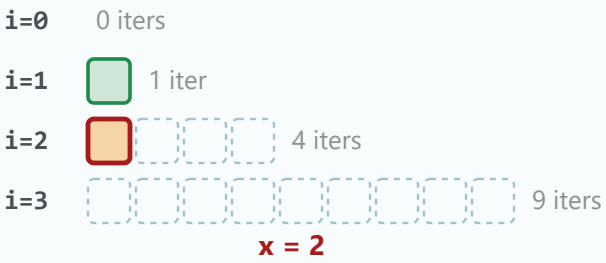
Step 3 / 9

VARIABLES

n = 4 | i = 1 | j = 0 | x = 1

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     for (int j = 0; j < i * i; j++) {
▶       x++;
6     }
7   }
8   return x;
9 }
```



i = 2: the inner loop runs while j < 2·2 = 4. First iteration (j = 0): x = 2.

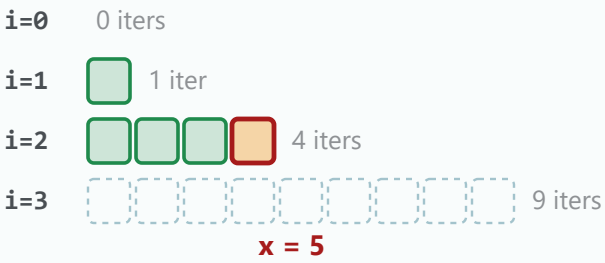
Step 4 / 9

VARIABLES

n = 4 | i = 2 | j = 0 | x = 2

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     for (int j = 0; j < i * i; j++) {
▶       x++;
6     }
7   }
8   return x;
9 }
```



j = 3 is the last iteration: row i = 2 finished in 4 iterations, x = **5**.

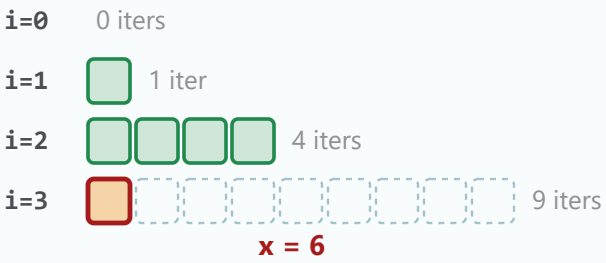
Step 5 / 9

VARIABLES

`n = 4` `i = 2` `j = 3` `x = 5`

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶       for (int j = 0; j < i * i; j++) {
▶         x++;
6       }
7   }
8   return x;
9 }
```



i = 3: the inner loop runs while j < 3·3 = 9. First iteration (j = 0): x = 6.

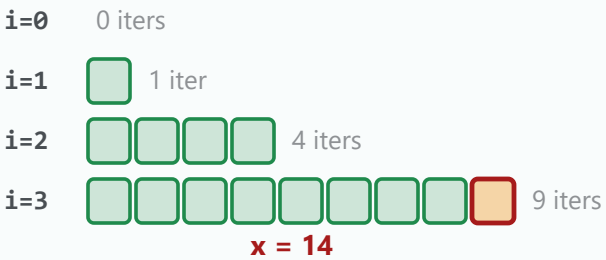
Step 6 / 9

VARIABLES

n = 4 | i = 3 | j = 0 | x = 6

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
▶     for (int j = 0; j < i * i; j++) {
▶       x++;
6     }
7   }
8   return x;
9 }
```



j = 8 is the last iteration: row i = 3 finished in 9 iterations, x = **14**.

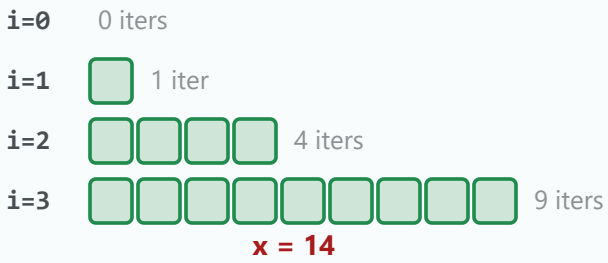
Step 7 / 9

VARIABLES

`n = 4` `i = 3` `j = 8` `x = 14`

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
▶ 3   for (int i = 0; i < n; i++) {
4     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
8   return x;
9 }
```



i = 4: `4 < 4` is false and the outer loop ends. Iterations: 0 + 1 + 4 + 9.

Step 8 / 9

VARIABLES

n = 4 | i = 4 | j = – | x = 14

f2(4): Tracing the Code

```
1 public int f2(int n) {
2   int x = 0;
3   for (int i = 0; i < n; i++) {
4     for (int j = 0; j < i * i; j++) {
5       x++;
6     }
7   }
▶   return x;
9 }
```



`return 14` = $0^2 + 1^2 + 2^2 + 3^2 = \Sigma\, i^2$. Since the rows grow as i^2, the total grows like $n^3/3$: **$\Theta(n^3)$**.

Step 9 / 9

VARIABLES

`n = 4` `i = –` `j = –` `x = 14`

LOOP ANALYSIS

f2(5): How Often Does the Inner Loop Run?

Each bar shows how many times the inner loop runs in one outer iteration (fixed i). The total area of the bars is the number of `x++` operations f2 performs.



total x++ = 0

For i = 0 the inner loop runs **0** times (0 × 0 = 0).

Step 1 / 6

f2(5): How Often Does the Inner Loop Run?

Each bar shows how many times the inner loop runs in one outer iteration (fixed i). The total area of the bars is the number of `x++` operations f2 performs.



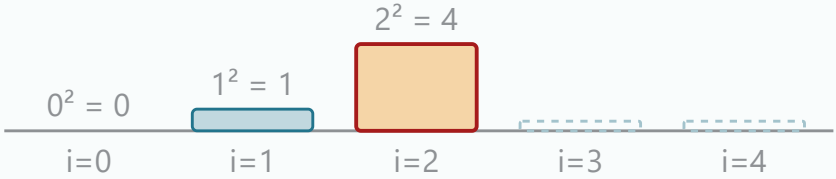
total x++ = 1

For i = 1 the inner loop runs **1** time (1 × 1 = 1).

LOOP ANALYSIS

f2(5): How Often Does the Inner Loop Run?

Each bar shows how many times the inner loop runs in one outer iteration (fixed i). The total area of the bars is the number of `x++` operations f2 performs.



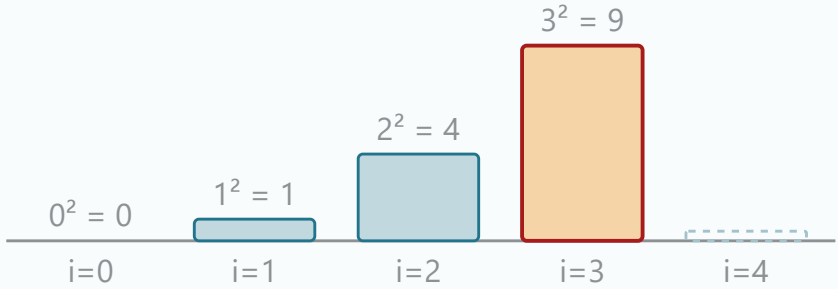
total x++ = 5

For i = 2 the inner loop runs **4** times (2 × 2 = 4).

LOOP ANALYSIS

f2(5): How Often Does the Inner Loop Run?

Each bar shows how many times the inner loop runs in one outer iteration (fixed i). The total area of the bars is the number of `x++` operations f2 performs.



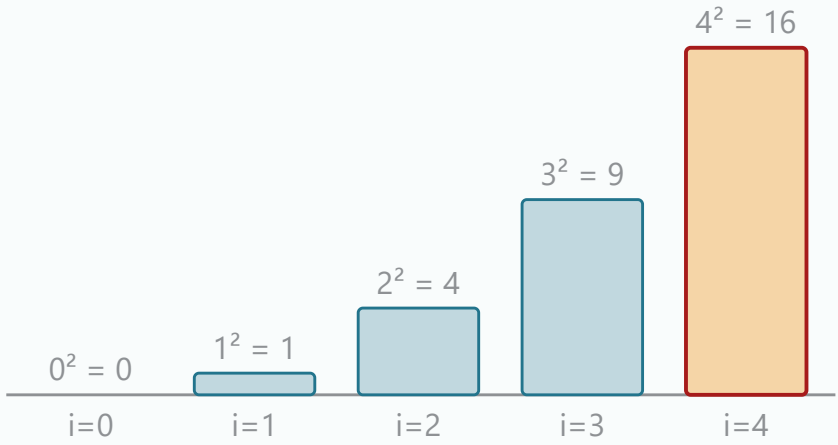
total x++ = 14

For i = 3 the inner loop runs **9** times (3 × 3 = 9).

LOOP ANALYSIS

f2(5): How Often Does the Inner Loop Run?

Each bar shows how many times the inner loop runs in one outer iteration (fixed i). The total area of the bars is the number of `x++` operations f2 performs.



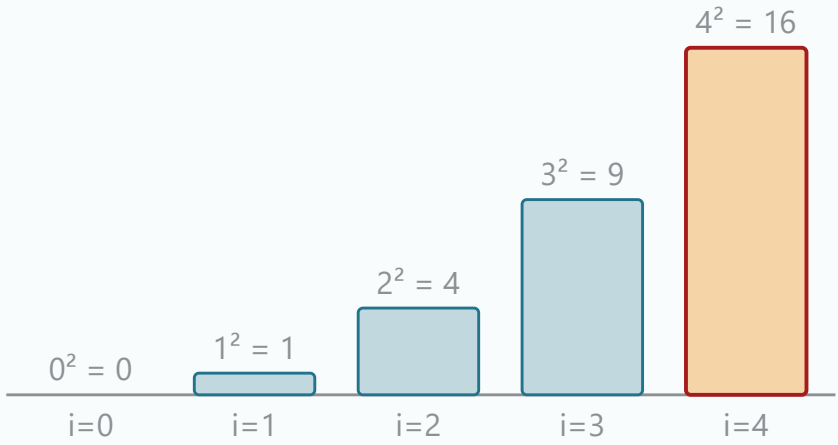
total x++ = 30

For i = 4 the inner loop runs **16** times. In general it runs **i^2** times for each i.

LOOP ANALYSIS

f2(5): How Often Does the Inner Loop Run?

Each bar shows how many times the inner loop runs in one outer iteration (fixed i). The total area of the bars is the number of `x++` operations f2 performs.



total x++ = 30

Total: 0 + 1 + 4 + 9 + 16 = **30** = (n − 1) · n · (2n − 1) / 6 = 4 · 5 · 9 / 6. f2(5) returns 30.

LOOP ANALYSIS

f2: Evaluating the Sum

Total number of inner iterations (i = 0 to n − 1):

$$\Sigma_{i=0}^{n-1} i^2 = (n-1) \cdot n \cdot (2n-1) / 6$$

$$= (2n^3 - 3n^2 + n) / 6 \approx n^3 / 3$$

The dominant term is $n^3/3$; dropping the constant 1/3 and the lower-order terms gives **$T(n) = \Theta(n^3)$**.

n	f2(n)	$n^3/3$
4	14	21
5	30	42
16	1,240	1,365
32	10,416	10,923

Check: when n doubles (16 → 32), the step count grows by 10,416 / 1,240 ≈ **8.4**, close to 2^3 = 8. That is the signature of cubic growth.

LOOP ANALYSIS

Loop Patterns and Summation Formulas

Pattern	Iterations	Result
`for (i = 0; i < n; i++)`	$\Sigma\ 1 = n$	$\Theta(n)$
`for (j = 0; j < n; j++)` inside `for i < n`	$n \cdot n$	$\Theta(n^2)$
`for (j = 0; j < i; j++)` inside `for i < n`	$\Sigma\ i = n(n-1)/2$	$\Theta(n^2)$
`for (j = 0; j < i * i; j++)` inside `for i < n`	$\Sigma\ i^2 = (n-1)n(2n-1)/6$	$\Theta(n^3)$
`for (i = 1; i < n; i *= 2)`	$\lfloor \log_2 n \rfloor$	$\Theta(\log n)$
`for j < n` inside `for (i = n; i > 0; i /= 2)`	$n \cdot (\lfloor \log_2 n \rfloor + 1)$	$\Theta(n \log n)$
`for j < i` inside `for (i = 1; i < n; i *= 2)`	$1 + 2 + 4 + \ldots < 2n$	$\Theta(n)$
`for (i = 0; i < n; i = 1 << i)` (f6)	about log* n + 1	$\Theta(\log^* n)$*

Method: write down the inner iteration count for each outer iteration and add them up; the sum reduces to a known series. * Mathematical model; in Java $n \leq 65536$.

LOOP ANALYSIS

f6: A Shifting Loop

```java
public static int f6(int n) {
  int x = 0;
  // 1 << i equals 2^i (overflow ignored)
  for (int i = 0; i < n; i = 1 << i) {
    x++;
  }
  return x;
}
```

- **Initialization:** `int i = 0;` sets i to 0; constant time, O(1).
- **Condition:** `i < n` checks on every iteration that i is less than n.
- **Update:** `i = 1 << i` shifts 1 left by i bits; the result is $\mathbf{2^i}$, computed in constant time.
- The update is not `i = 2 * i` but `i = 2^i`. i itself **becomes the exponent**, so i grows much faster.

f6(20): Tracing the Code

```
1 public static int f6(int n) {
▶   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
4   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



`x` is set to zero.

Step 1 / 13

VARIABLES

`n = 20` `x = 0` `i = –`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 0 < 20 → true

Start with i = 0; the condition 0 < 20 is true.

Step 2 / 13

VARIABLES

`n = 20` `x = 0` `i = 0`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
4   for (int i = 0; i < n; i = 1 << i) {
▶     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 0 < 20 → true

`x++` → x = 1.

Step 3 / 13

VARIABLES

`n = 20` `x = 1` `i = 0`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
  ▶ for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 1 < 20 → true

i = 1 << 0 = 1; the condition 1 < 20 is true.

Step 4 / 13

VARIABLES

n = 20 | x = 1 | i = 1

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
4   for (int i = 0; i < n; i = 1 << i) {
▶       x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 1 < 20 → true

`x++` → x = 2.

Step 5 / 13

VARIABLES

`n = 20` `x = 2` `i = 1`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 2 < 20 → true

i = 1 << 1 = 2; the condition 2 < 20 is true.

Step 6 / 13

VARIABLES

`n = 20` `x = 2` `i = 2`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
4   for (int i = 0; i < n; i = 1 << i) {
▶       x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 2 < 20 → true

`x++` → x = 3.

Step 7 / 13

VARIABLES

`n = 20` `x = 3` `i = 2`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
  for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 4 < 20 → true

i = 1 << 2 = 4; the condition 4 < 20 is true.

Step 8 / 13

VARIABLES

`n = 20` `x = 3` `i = 4`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
4   for (int i = 0; i < n; i = 1 << i) {
      x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 4 < 20 → true

`x++` → x = 4.

Step 9 / 13

VARIABLES

`n = 20` `x = 4` `i = 4`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 16 < 20 → true

i = 1 << 4 = 16; the condition 16 < 20 is true.

Step 10 / 13

VARIABLES

`n = 20` `x = 4` `i = 16`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
4   for (int i = 0; i < n; i = 1 << i) {
      x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 16 < 20 → true

`x++` → x = 5.

Step 11 / 13

VARIABLES

`n = 20` `x = 5` `i = 16`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
▶   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
7   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 65536 < 20 → false, loop ends

i = 1 << 16 = 65536; the condition 65536 < 20 is false → the loop ends.

VARIABLES

`n = 20` `x = 5` `i = 65536`

f6(20): Tracing the Code

```
1 public static int f6(int n) {
2   int x = 0;
3   // 1 << i equals 2^i (overflow
  ignored)
4   for (int i = 0; i < n; i = 1 << i) {
5     x++;
6   }
▶   return x;
8 }
```

values of i (n = 20); i ← 1 << i



condition: 65536 < 20 → false, loop ends

`return x` → 5. The values of i are 0, 1, 2, 4, 16: only 5 iterations.

Step 13 / 13

VARIABLES

`n = 20` `x = 5` `i = –`

LOOP ANALYSIS

f6: How Many Iterations Really?

Values of i: $0 \to 2^0 = 1 \to 2^1 = 2 \to 2^2 = 4 \to 2^4 = 16 \to 2^{16} = 65536$. Each iteration turns i into an **exponent** (a tower of powers of 2).

A common line of reasoning: "after k steps $2^k \geq n$, so $k \geq \log_2 n$" — true for an `i = 2 * i` loop. In f6, after k iterations i is a tower of height $k - 1$; the number of iterations is about **log* n** (the iterated logarithm).

O(log n) is correct as an **upper bound** but not tight: **Θ(log* n)**.

n	f6(n)	$\lceil \log_2 n \rceil$
4	3	2
16	4	4
32	5	5
1000	5	10
65536	5	16

Java pitfall: the shift distance for an `int` is taken mod 32: `1 << 65536` = 1. So for n > 65536, i cycles through 1, 2, 4, 16, 65536 and f6 **never terminates**. The "ignore overflow" assumption matters here.

Recurrence Relations

The running time of a recursive function is written in terms of **its own running time on smaller inputs**:

$$T(n) = (\text{time of the recursive calls}) + (\text{work outside the calls}), \quad T(\text{base}) = \Theta(1)$$

Expansion (substitution)

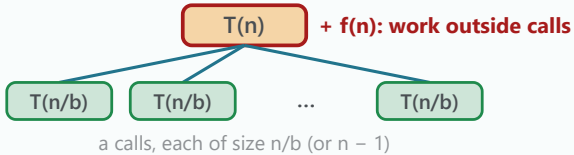
Plug the recurrence into itself repeatedly to find a pattern; sum up once the base case is reached. (f3)

Recursion tree

Draw each call as a node and add up the work on each level. (f3, f4, f5)

Master Theorem

Solve divide-and-conquer recurrences of the form T(n) = aT(n/b) + f(n) directly. (f4, f5)



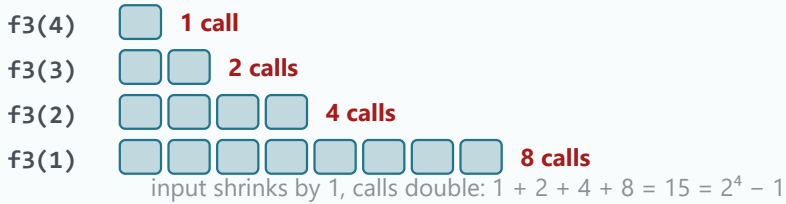
Questions to ask when setting up a recurrence: How many recursive calls are there? How much smaller is each call's input (n − 1 or n/2)? How much work is done outside the calls?

f3: Two Calls on n − 1 → O(2^n)

```java
public long f3(int n) {
  if (n <= 1) {
    return 1;
  }
  return f3(n - 1) + f3(n - 1);
}
```

- **Base case:** for n ≤ 1 it returns 1; constant time, O(1).
- **Recursive case:** for n > 1 it calls `f3(n - 1)` twice and adds the results (constant work c).

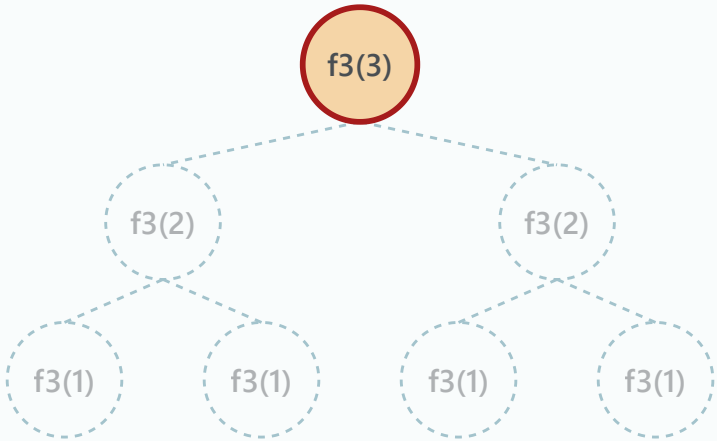
$$T(n) = 2T(n-1) + c, \quad n > 1 \qquad T(1) = c$$



The input shrinks by **only 1** per call, but the number of calls **doubles** at every level. This recurrence is not of the form T(n/b), so the Master Theorem does not apply; we solve it by expansion.

f3(3): Tracing the Calls

```
1 public long f3(int n) {
▶   if (n <= 1) {
3     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(3)` is called: 3 > 1, not the base case.

Step 1 / 10

VARIABLES

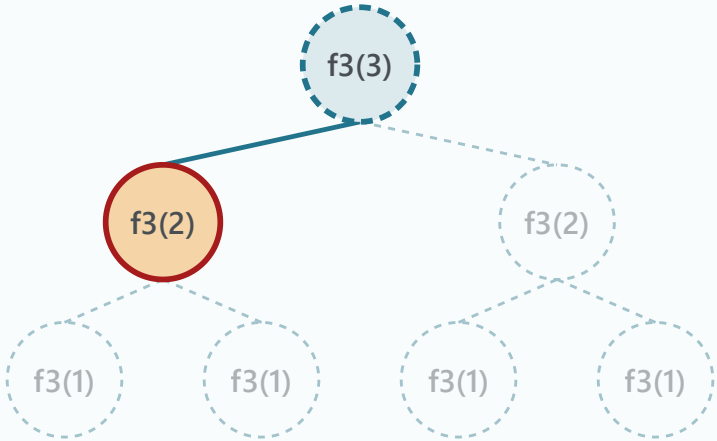
`n = 3` `result = –` `calls = 1`

CALL STACK

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



Line 5: the first `f3(2)` call. f3(3) waits on the stack.

Step 2 / 10

VARIABLES

`n = 2` `result = –` `calls = 2`

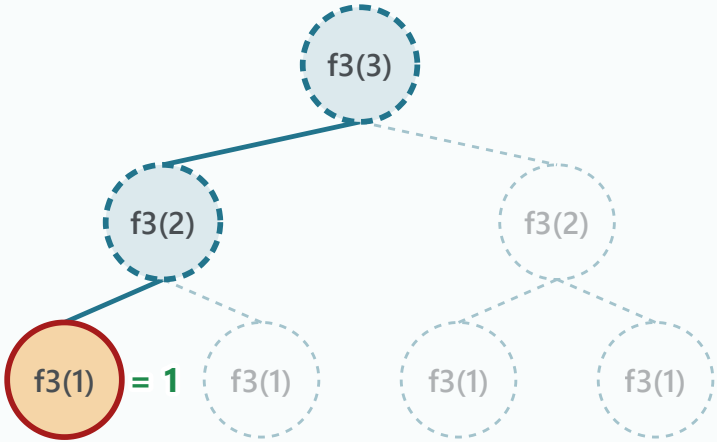
CALL STACK

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(1)` : n ≤ 1 → base case, returns 1.

Step 3 / 10

VARIABLES

`n = 1` `result = 1` `calls = 3`

CALL STACK

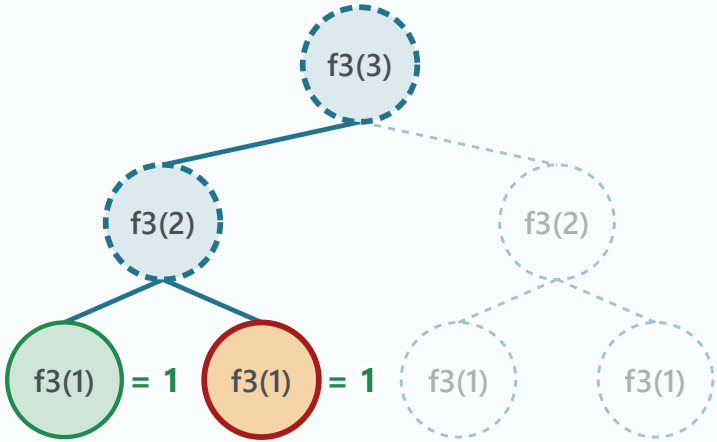
`f3(1)`

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



f3(2) makes its second call: again `f3(1)` = 1.

Step 4 / 10

VARIABLES

`n = 1` `result = 1`

`calls = 4`

CALL STACK

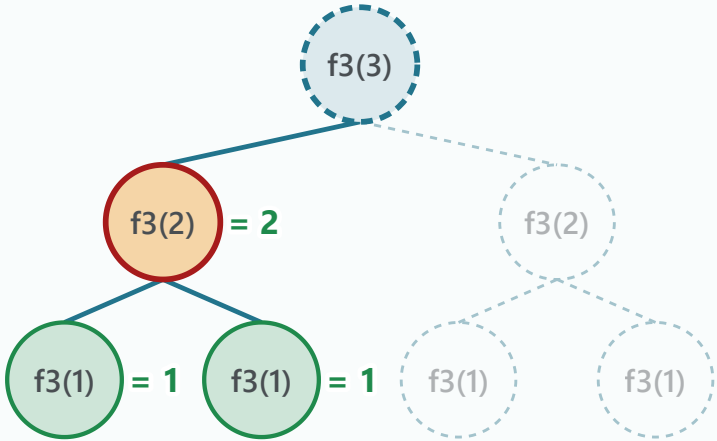
`f3(1)`

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



f3(2) returns 1 + 1 = 2. The left subtree is done.

Step 5 / 10

VARIABLES

`n = 2` `result = 2` `calls = 4`

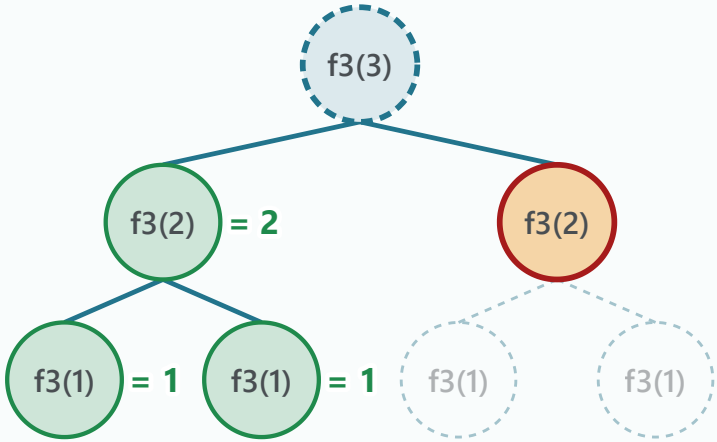
CALL STACK

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



f3(3) calls the second `f3(2)`: the same work is done **all over again**.

Step 6 / 10

VARIABLES

`n = 2` `result = –` `calls = 5`

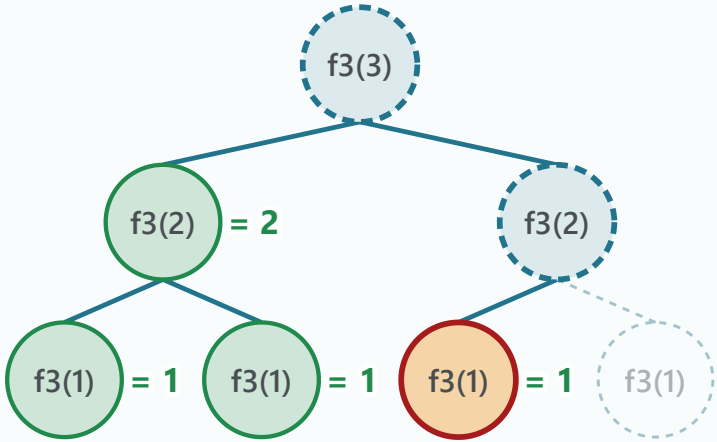
CALL STACK

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(1)` = 1.

Step 7 / 10

VARIABLES

`n = 1` `result = 1`

`calls = 6`

CALL STACK

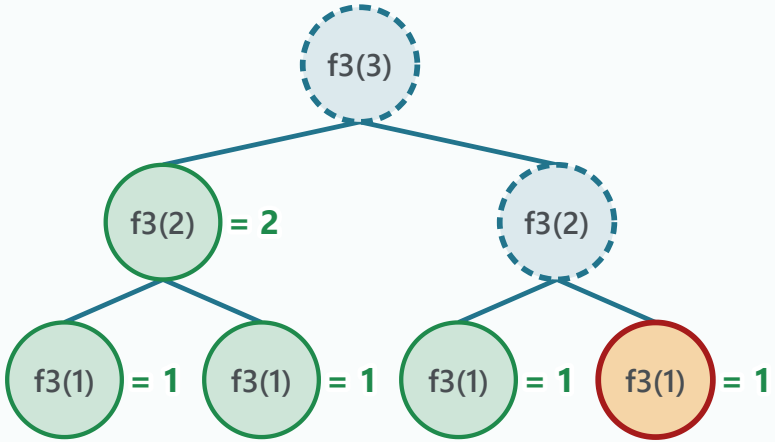
`f3(1)`

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f3(n - 1) + f3(n - 1);
6 }
```



`f3(1)` = 1.

Step 8 / 10

VARIABLES

`n = 1` `result = 1`

`calls = 7`

CALL STACK

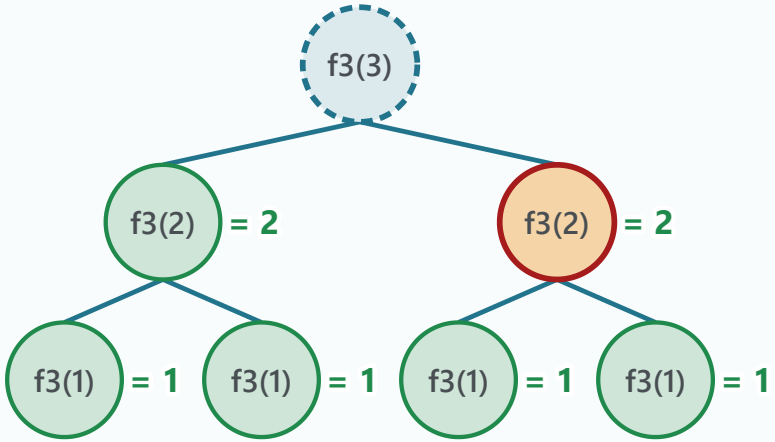
`f3(1)`

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



The second f3(2) returns 2.

Step 9 / 10

VARIABLES

`n = 2` `result = 2`

`calls = 7`

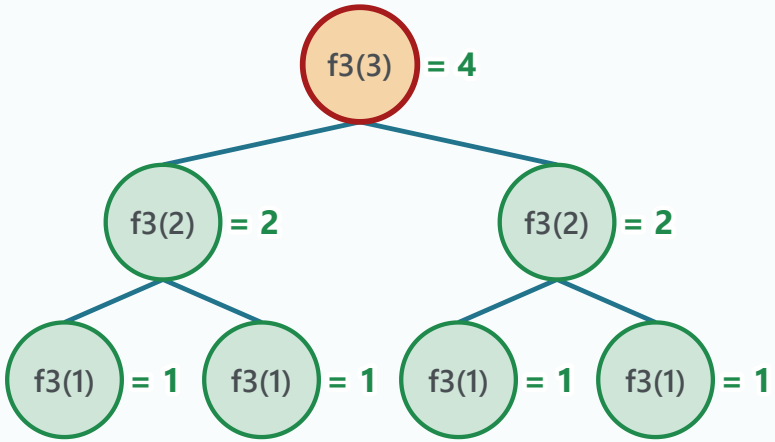
CALL STACK

`f3(2)`

`f3(3)`

f3(3): Tracing the Calls

```
1 public long f3(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f3(n - 1) + f3(n - 1);
6 }
```



f3(3) returns 2 + 2 = 4. **7 = 2^3 − 1** calls in total; the stack depth is at most 3.

Step 10 / 10

VARIABLES

`n = 3` `result = 4` `calls = 7`

CALL STACK

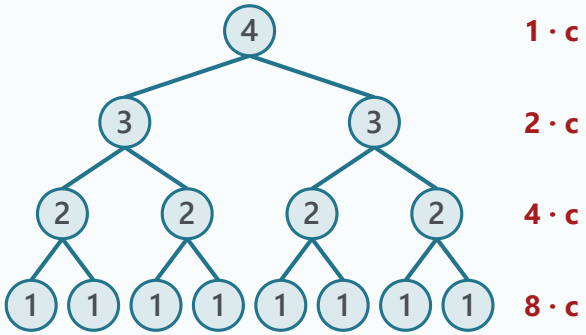
`f3(3)`

f3: Expanding the Recurrence

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

The recurrence itself: two subcalls, constant extra work.

Step 1 / 5



Counted in Java: f3(16) makes 65,535 calls and f3(32) 4,294,967,295 ($2^n - 1$).

f3: Expanding the Recurrence

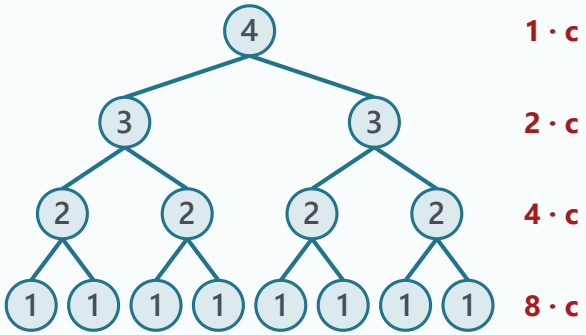
$$T(n) = 2T(n-1) + c$$

The recurrence itself: two subcalls, constant extra work.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

Substitute the definition of T(n − 1).

Step 2 / 5



Counted in Java: f3(16) makes 65,535 calls and f3(32) 4,294,967,295 ($2^n - 1$).

f3: Expanding the Recurrence

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

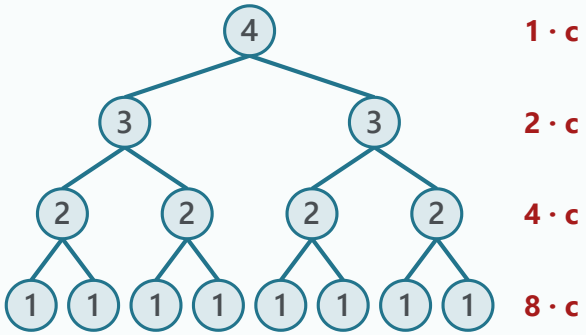
The recurrence itself: two subcalls, constant extra work.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

Substitute the definition of T(n − 1).

$$= 8T(n-3) + 7c$$

Expanding once more reveals the pattern: coefficients 2, 4, 8 …



Counted in Java: f3(16) makes 65,535 calls and f3(32) 4,294,967,295 ($2^n - 1$).

f3: Expanding the Recurrence

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

The recurrence itself: two subcalls, constant extra work.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

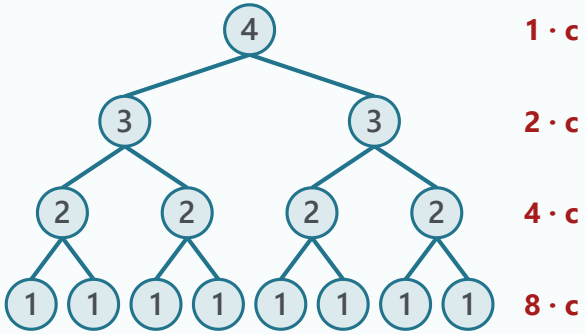
Substitute the definition of T(n − 1).

$$= 8T(n-3) + 7c$$

Expanding once more reveals the pattern: coefficients 2, 4, 8 …

$$= 2^k\, T(n-k) + (2^k - 1)c$$

The general form after k steps. Ignoring the extra work c gives $T(n) = 2^k\, T(n-k)$; the growth class is the same.



Counted in Java: f3(16) makes 65,535 calls and f3(32) 4,294,967,295 ($2^n - 1$).

f3: Expanding the Recurrence

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

The recurrence itself: two subcalls, constant extra work.

$$= 2[2T(n-2) + c] + c = 4T(n-2) + 3c$$

Substitute the definition of T(n − 1).

$$= 8T(n-3) + 7c$$

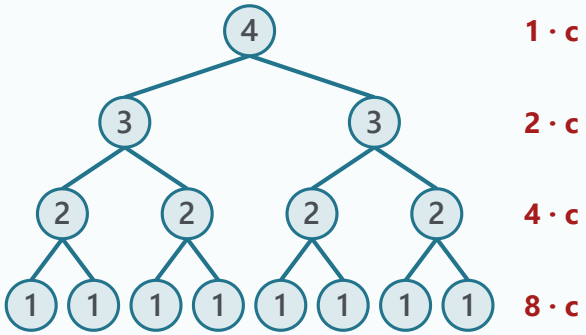
Expanding once more reveals the pattern: coefficients 2, 4, 8 …

$$= 2^k\, T(n-k) + (2^k - 1)c$$

The general form after k steps. Ignoring the extra work c gives $T(n) = 2^k\, T(n-k)$; the growth class is the same.

$$k = n-1 : T(n) = 2^{n-1}c + (2^{n-1} - 1)c = (2^n - 1)c$$

The base case T(1) = c is reached when n − k = 1. Result: **$T(n) = \Theta(2^n)$**.



Step 5 / 5

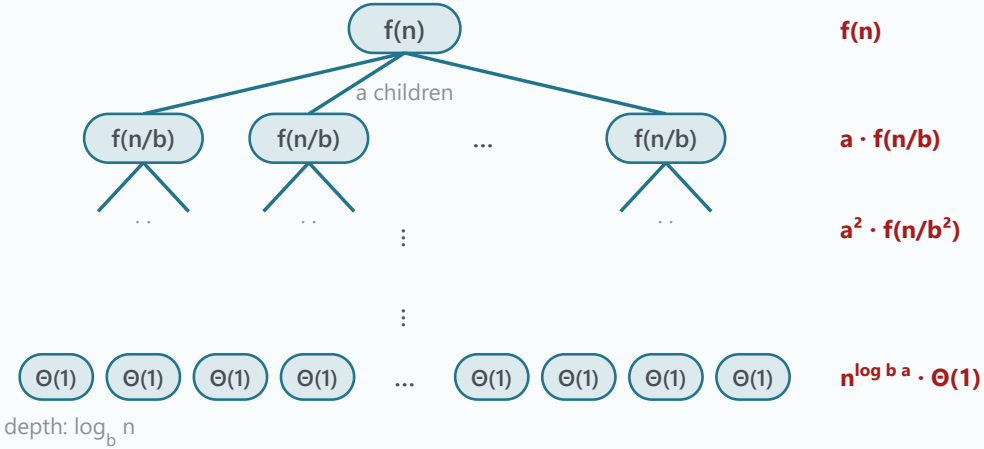
Counted in Java: f3(16) makes 65,535 calls and f3(32) 4,294,967,295 ($2^n - 1$).

Master Theorem: General Form

Solves divide-and-conquer recurrences directly:

$$T(n) = a \cdot T(n/b) + f(n), \quad a \geq 1, b > 1$$

a: number of subproblems (calls) · **b:** shrink factor, subproblem size **n/b** · **f(n):** cost of dividing and combining. Depth $\log_b n$, number of leaves $a^{\log_b n} = n^{\log_b a}$.



Master Theorem: Three Cases

Compare f(n) with $\mathbf{n^{\log_b a}}$, the total work at the leaves. Let c be the polynomial degree of f(n):

Case	Condition	Result	Interpretation
1	$f(n) = O(n^c)$, $c < \log_b a$	$T(n) = \Theta(n^{\log_b a})$	leaves dominate
2	$f(n) = \Theta(n^c)$, $c = \log_b a$	$T(n) = \Theta(n^{\log_b a} \cdot \log n)$	levels balanced
3	$f(n) = \Omega(n^c)$, $c > \log_b a$ *and* $a \cdot f(n/b) \le k \cdot f(n)$, $k < 1$	$T(n) = \Theta(f(n))$	root dominates

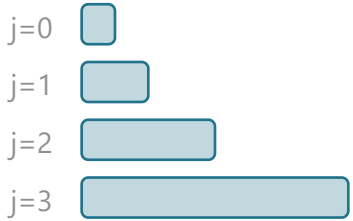
In Cases 2 and 3, writing $f(n) = O(n^c)$ is not enough. For instance $f(n) = 1$ is also $O(n^2)$, yet $T(n) = T(n/2) + 1$ cannot be $\Theta(n^2)$. So Case 2 requires $f = \mathbf{\Theta}(n^c)$, and Case 3 requires $f = \mathbf{\Omega}(n^c)$ plus the **regularity condition**; the results are tight (Θ).

The regularity condition holds automatically for polynomials $f(n) = n^c$: $a(n/b)^c = (a/b^c)\, n^c$ and $a/b^c < 1$.

Why Three Cases? Level Sums

If $f(n) = n^c$, level j of the tree has a^j nodes, each doing $(n/b^j)^c$ work: **level sum = $n^c \cdot (a/b^c)^j$**. The total is a geometric series with ratio $r = a/b^c$.

$4T(n/2) + n$



$r = 2 > 1$: grows

Case 1: last level (leaves) dominates → $\Theta(n^2)$

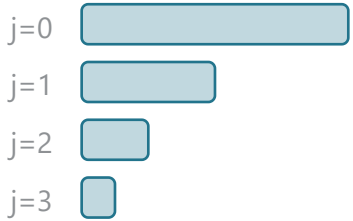
$2T(n/2) + n$



$r = 1$: constant

Case 2: $\log n$ levels $\times$ n work → $\Theta(n \log n)$

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



$r = 1/2 < 1$: shrinks

Case 3: root dominates → $\Theta(n^2)$

$r > 1 \Leftrightarrow b^c < a \Leftrightarrow c < \log_b a$. So the three cases of the Master Theorem correspond to a geometric series that is **increasing**, **constant** or **decreasing**.

Example: T(n) = 2T(n/2) + Θ(n)

$$a = 2, \quad b = 2, \quad f(n) = \Theta(n)$$

Read off the parameters: two subproblems, each of size n/2, linear combining work.

Step 1 / 5

Where does this recurrence arise?

- **Merge sort**
  : split the array in half, sort both halves, merge in Θ(n).
- The chapter's `f5` : `f1(n)` plus two calls to `f5(n/2)` .
- Quicksort with balanced partitions.

Example: T(n) = 2T(n/2) + Θ(n)

$$a = 2, \quad b = 2, \quad f(n) = \Theta(n)$$

Read off the parameters: two subproblems, each of size n/2, linear combining work.

$$\log_b a = \log_2 2 = 1$$

The leaf-work exponent is 1: $n^{\log_2 2} = n$.

Step 2 / 5

Where does this recurrence arise?

- **Merge sort**
  : split the array in half, sort both halves, merge in Θ(n).
- The chapter's `f5` : `f1(n)` plus two calls to `f5(n/2)` .
- Quicksort with balanced partitions.

Example: T(n) = 2T(n/2) + Θ(n)

$$a = 2,\quad b = 2,\quad f(n) = \Theta(n)$$

Read off the parameters: two subproblems, each of size n/2, linear combining work.

$$\log_b a = \log_2 2 = 1$$

The leaf-work exponent is 1: $n^{\log_2 2} = n$.

$$f(n) = \Theta(n^1) \rightarrow c = 1$$

The degree of f(n) is c = 1.

Step 3 / 5

Where does this recurrence arise?

- **Merge sort**
  : split the array in half, sort both halves, merge in Θ(n).
- The chapter's `f5` : `f1(n)` plus two calls to `f5(n/2)` .
- Quicksort with balanced partitions.

Example: T(n) = 2T(n/2) + Θ(n)

$$a = 2, \quad b = 2, \quad f(n) = \Theta(n)$$

Read off the parameters: two subproblems, each of size n/2, linear combining work.

$$\log_b a = \log_2 2 = 1$$

The leaf-work exponent is 1: $n^{\log_2 2} = n$.

$$f(n) = \Theta(n^1) \to c = 1$$

The degree of f(n) is c = 1.

$$c = \log_b a \to \text{Case } 2$$

Equal exponents: every level does the same (n) work.

Where does this recurrence arise?

- **Merge sort**
  : split the array in half, sort both halves, merge in Θ(n).
- The chapter's `f5` : `f1(n)` plus two calls to `f5(n/2)` .
- Quicksort with balanced partitions.

Example: T(n) = 2T(n/2) + Θ(n)

$$a = 2, \quad b = 2, \quad f(n) = \Theta(n)$$

Read off the parameters: two subproblems, each of size n/2, linear combining work.

$$\log_b a = \log_2 2 = 1$$

The leaf-work exponent is 1: $n^{\log_2 2} = n$.

$$f(n) = \Theta(n^1) \rightarrow c = 1$$

The degree of f(n) is c = 1.

$$c = \log_b a \rightarrow \text{Case 2}$$

Equal exponents: every level does the same (n) work.

$$T(n) = \Theta(n \log n)$$

Case 2: $T(n) = \Theta(n^{\log_b a} \cdot \log n) = \mathbf{\Theta(n \log n)}$.

Where does this recurrence arise?

- **Merge sort**
  : split the array in half, sort both halves, merge in Θ(n).
- The chapter's `f5` : `f1(n)` plus two calls to `f5(n/2)` .
- Quicksort with balanced partitions.

Master Theorem: Examples

Recurrence	a, b	$\log_b a$	c	Case	Result	Algorithm
$T(n/2) + 1$	1, 2	0	0	2	$\Theta(\log n)$	binary search
$2T(n/2) + 1$	2, 2	1	0	1	$\Theta(n)$	f4, tree traversal
$2T(n/2) + n$	2, 2	1	1	2	$\Theta(n \log n)$	f5, merge sort
$3T(n/2) + n$	3, 2	≈ 1.585	1	1	$\Theta(n^{1.585})$	Karatsuba multiplication
$4T(n/2) + n$	4, 2	2	1	1	$\Theta(n^2)$	—
$7T(n/2) + n^2$	7, 2	≈ 2.807	2	1	$\Theta(n^{2.807})$	Strassen
$2T(n/2) + n^2$	2, 2	1	2	3	$\Theta(n^2)$	—
$T(n/2) + n$	1, 2	0	1	3	$\Theta(n)$	quickselect (average)

Regularity in the Case 3 rows: $2(n/2)^2 = n^2/2 \le \frac{1}{2} n^2$ and $1 \cdot (n/2) \le \frac{1}{2} n$ hold with $k = \frac{1}{2}$.

When Does the Master Theorem Not Apply?

Subtractive recurrences

$T(n) = 2T(n - 1) + c$ (f3) or $T(n) = T(n - 1) + n$: the subproblem is $n - 1$, not n/b. Use expansion.

Non-polynomial gap

$T(n) = 2T(n/2) + n / \log n$: $f(n)$ is smaller than n by only a log factor; no case fits.

Unequal splits

$T(n) = T(n/3) + T(2n/3) + n$: subproblems of different sizes. A recursion tree (or Akra–Bazzi) gives $\Theta(n \log n)$.

Non-constant a, or a < 1

$T(n) = 2^n\, T(n/2) + n$ or $T(n) = ½\, T(n/2) + n$: the theorem's assumption that $a \geq 1$ is a constant breaks.

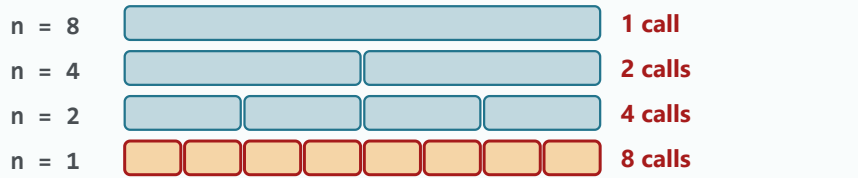
An extension of the basic form: if $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$. For example $T(n) = 2T(n/2) + n \log n = \Theta(n \log^2 n)$.

f4: Two Calls on n/2 → O(n)

```java
public int f4(int n) {
  if (n <= 1) {
    return 1;
  }
  return f4(n / 2) + f4(n / 2);
}
```

- **Base case:** for n ≤ 1 it returns 1; O(1).
- **Recursive case:** for n > 1 it calls `f4(n/2)` twice and adds the results (constant work).

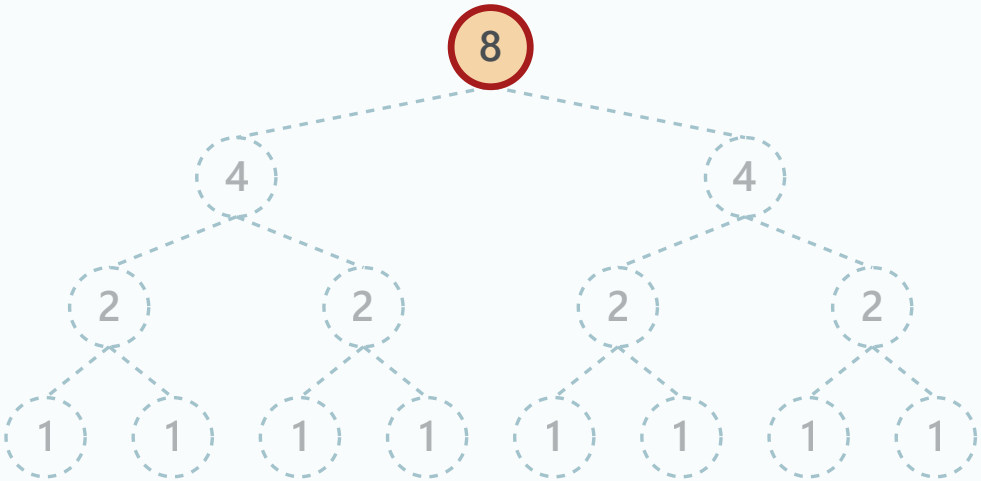
$$T(n) = 2T(n/2) + O(1), \quad T(1) = O(1)$$



input halves: only $\log_2 8 + 1 = 4$ levels, $1 + 2 + 4 + 8 = 15 = 2n - 1$ calls

The only difference from f3 is that the input is **n/2** instead of **n − 1**, yet that brings the result down from $\Theta(2^n)$ to **$\Theta(n)$**. Two calls do not always mean exponential time: how fast the input shrinks decides.

f4(8): Recursion Tree



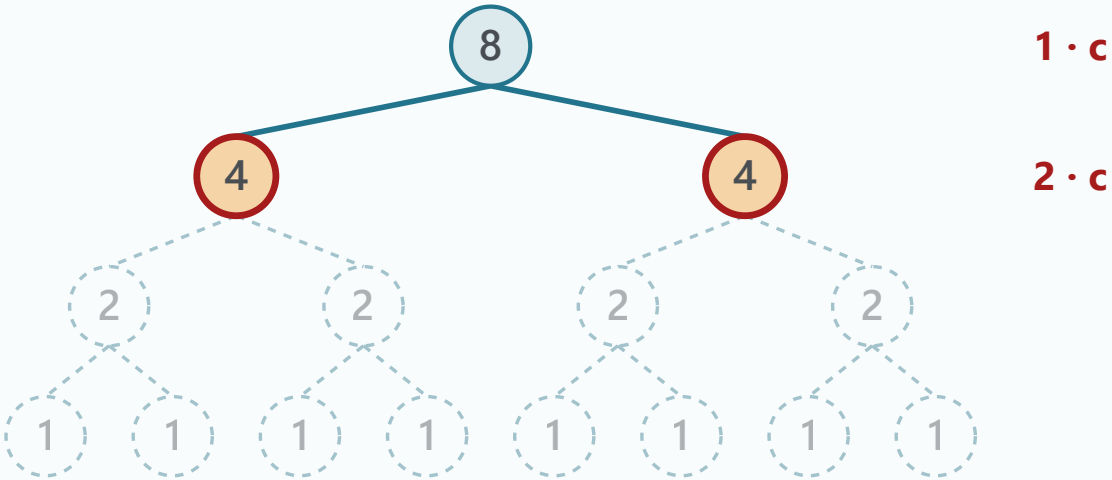
1 · c

Root: `f4(8)` does only constant work (c) and starts two calls.

Step 1 / 4

The level sums grow geometrically, so the last term (the leaves) dominates. In Java: f4(16) makes 31 calls, f4(32) 63 (2n − 1).

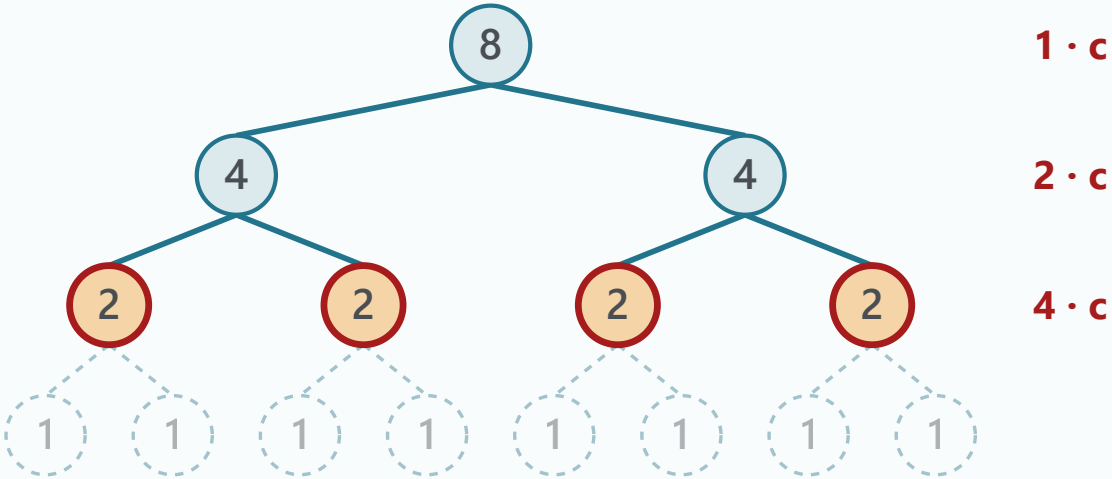
f4(8): Recursion Tree



Level 1: two `f4(4)` calls, 2c in total.

The level sums grow geometrically, so the last term (the leaves) dominates. In Java: f4(16) makes 31 calls, f4(32) 63 (2n − 1).

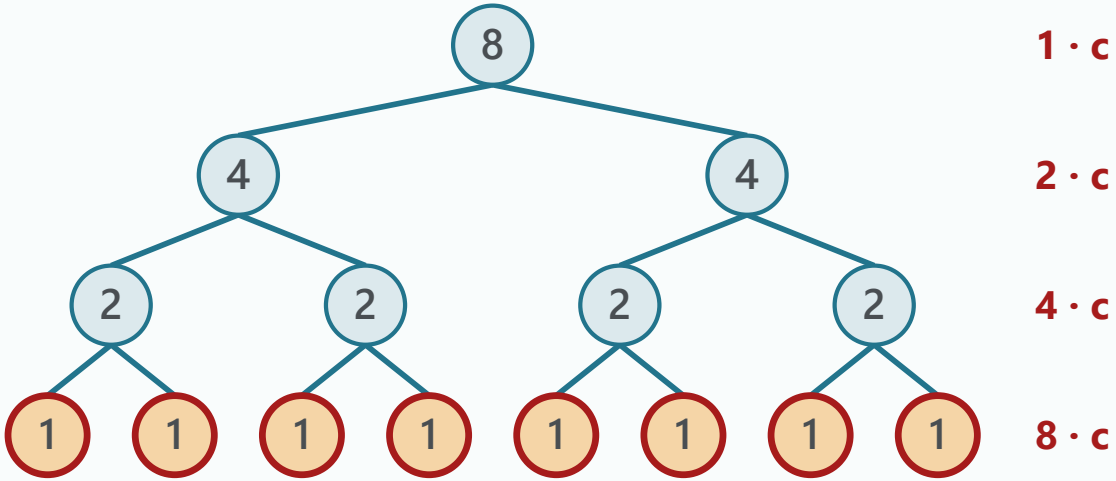
f4(8): Recursion Tree



Level 2: four `f4(2)` calls, 4c in total. The work per level **doubles** at each step.

The level sums grow geometrically, so the last term (the leaves) dominates. In Java: f4(16) makes 31 calls, f4(32) 63 (2n − 1).

f4(8): Recursion Tree



Leaves: eight `f4(1)`, 8c = n · c in total. Overall c(1 + 2 + 4 + 8) = 15c = (2n − 1)c → **Θ(n)**.

The level sums grow geometrically, so the last term (the leaves) dominates. In Java: f4(16) makes 31 calls, f4(32) 63 (2n − 1).

f4: Solving with the Master Theorem

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

Match it against the general form T(n) = aT(n/b) + f(n).

Step 1 / 5

Same result as the tree

The recursion tree had 2n − 1 nodes. The Master Theorem gives the same answer without computing the sum: the leaves dominate.

Note: calling f4 "O(log n)" is a common mistake; the right answer is Θ(n). log n is only the **stack depth** (memory).

f4: Solving with the Master Theorem

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

Match it against the general form T(n) = aT(n/b) + f(n).

$$a = 2, b = 2, f(n) = O(1)$$

Here a = 2, b = 2, f(n) = O(1).

Same result as the tree

The recursion tree had 2n − 1 nodes. The Master Theorem gives the same answer without computing the sum: the leaves dominate.

Note: calling f4 "O(log n)" is a common mistake; the right answer is Θ(n). log n is only the **stack depth** (memory).

f4: Solving with the Master Theorem

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

Match it against the general form T(n) = aT(n/b) + f(n).

$$a = 2, b = 2, f(n) = O(1)$$

Here a = 2, b = 2, f(n) = O(1).

$$\log_b a = \log_2 2 = 1$$

Leaf work: $n^{\log_2 2} = n^1 = n$.

Same result as the tree

The recursion tree had 2n − 1 nodes. The Master Theorem gives the same answer without computing the sum: the leaves dominate.

Note: calling f4 "O(log n)" is a common mistake; the right answer is Θ(n). log n is only the **stack depth** (memory).

f4: Solving with the Master Theorem

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

Match it against the general form T(n) = aT(n/b) + f(n).

$$a = 2, b = 2, f(n) = O(1)$$

Here a = 2, b = 2, f(n) = O(1).

$$\log_b a = \log_2 2 = 1$$

Leaf work: $n^{\log_2 2} = n^1 = n$.

$$f(n) = O(n^0) \to c = 0 < 1$$

f(n) = O(1) corresponds to c = 0, which is less than $\log_b a = 1$.

Same result as the tree

The recursion tree had 2n − 1 nodes. The Master Theorem gives the same answer without computing the sum: the leaves dominate.

Note: calling f4 "O(log n)" is a common mistake; the right answer is Θ(n). log n is only the **stack depth** (memory).

f4: Solving with the Master Theorem

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

Match it against the general form T(n) = aT(n/b) + f(n).

$$a = 2, b = 2, f(n) = O(1)$$

Here a = 2, b = 2, f(n) = O(1).

$$\log_b a = \log_2 2 = 1$$

Leaf work: $n^{\log_2 2} = n^1 = n$.

$$f(n) = O(n^0) \to c = 0 < 1$$

f(n) = O(1) corresponds to c = 0, which is less than $\log_b a = 1$.

$$\text{Case 1: } T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

By the Master Theorem, if f(n) = $O(n^c)$ with c < $\log_b a$ then T(n) = $\Theta(n^{\log_b a})$. Result: **Θ(n)**.

Same result as the tree

The recursion tree had 2n − 1 nodes. The Master Theorem gives the same answer without computing the sum: the leaves dominate.

Note: calling f4 "O(log n)" is a common mistake; the right answer is Θ(n). log n is only the **stack depth** (memory).

f5: Linear Work + Two Calls → O(n log n)

```java
public int f5(int n) {
  if (n <= 1) {
    return 1;
  }
  return f1(n) + f5(n / 2) + f5(n / 2);
}
```

- **Base case:** for n ≤ 1 it returns 1; O(1).
- **Recursive case:** for n > 1 it first calls `f1(n)` (Θ(n) work), then `f5(n/2)` twice.

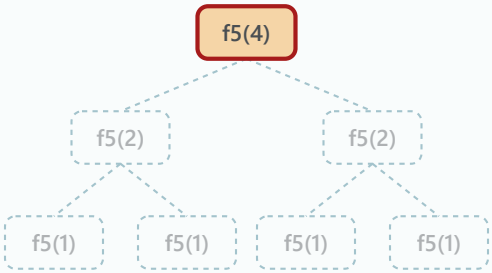
$$T(n) = 2T(n/2) + f1(n) = 2T(n/2) + \Theta(n)$$

The only difference from f4 is the **linear work** in each call. This is exactly the merge sort recurrence. The answer is not "O(n + log n)" but **Θ(n log n)**.

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```

level work



`f5(4)` is called: 4 > 1, not the base case.

Step 1 / 13

VARIABLES

`n = 4`

`f1(n) = –`

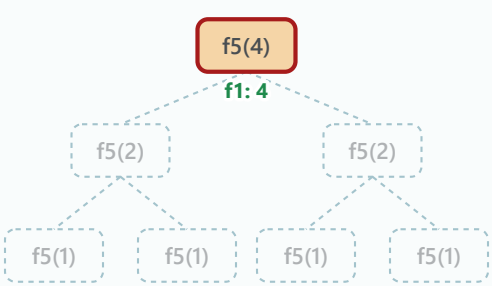
`result = –`

CALL STACK

`f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



level work
4

Line 5 first computes `f1(4)`: a 4-iteration loop, **4** units of work. Then two `f5(2)` calls.

Step 2 / 13

VARIABLES

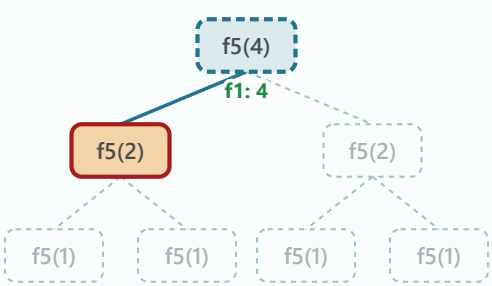
- `n = 4`
- `f1(n) = 4`
- `result = –`

CALL STACK

- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



level work
4

`f5(2)` is called: 2 > 1, not the base case.

Step 3 / 13

VARIABLES

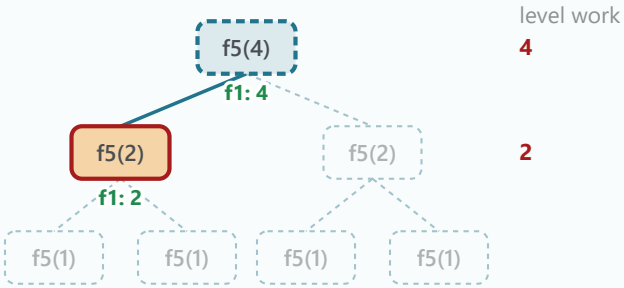
- `n = 2`
- `f1(n) = –`
- `result = –`

CALL STACK

- `f5(2)`
- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



Line 5 first computes `f1(2)`: a 2-iteration loop, **2** units of work. Then two `f5(1)` calls.

Step 4 / 13

VARIABLES

`n = 2`

`f1(n) = 2`

`result = –`

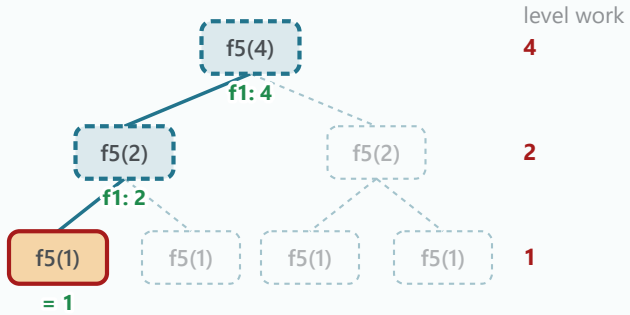
CALL STACK

`f5(2)`

`f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → base case, returns 1.

Step 5 / 13

VARIABLES

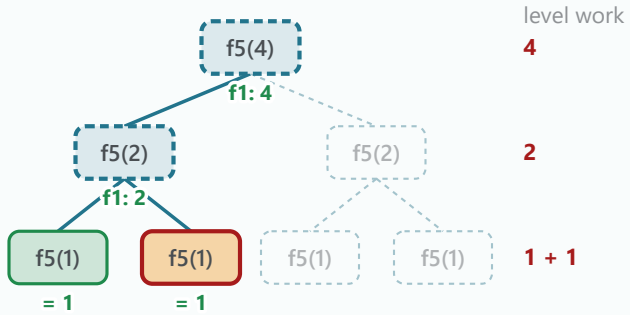
- `n = 1`
- `f1(n) = –`
- `result = 1`

CALL STACK

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → base case, returns 1.

Step 6 / 13

VARIABLES

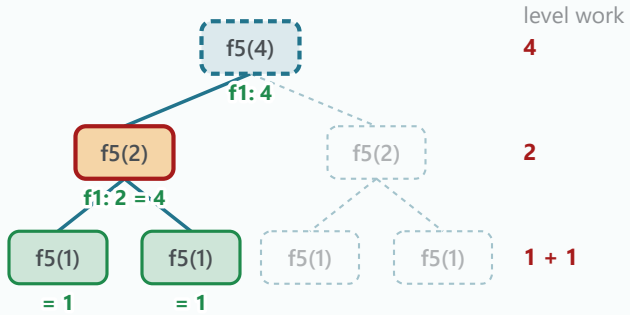
- `n = 1`
- `f1(n) = –`
- `result = 1`

CALL STACK

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



f5(2) = f1(2) + f5(1) + f5(1) = 2 + 1 + 1 = **4** is returned.

Step 7 / 13

VARIABLES

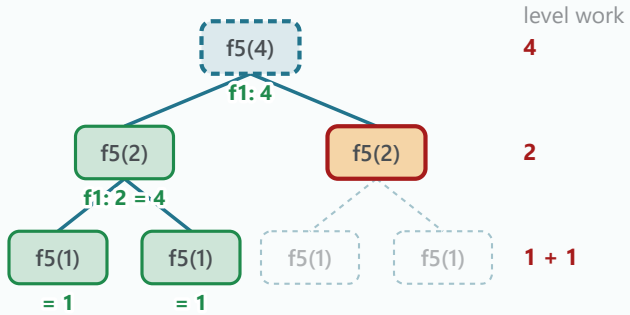
- `n = 2`
- `f1(n) = 2`
- `result = 4`

CALL STACK

- `f5(2)`
- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



The second `f5(2)` call repeats the same work as the first.

Step 8 / 13

VARIABLES

`n = 2`

`f1(n) = –`

`result = –`

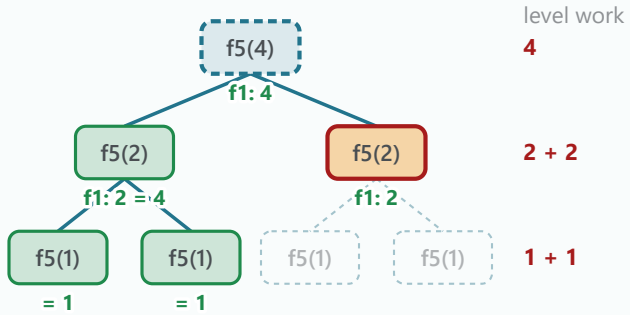
CALL STACK

`f5(2)`

`f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



Line 5 first computes `f1(2)`: a 2-iteration loop, **2** units of work. Then two `f5(1)` calls.

Step 9 / 13

VARIABLES

`n = 2`

`f1(n) = 2`

`result = –`

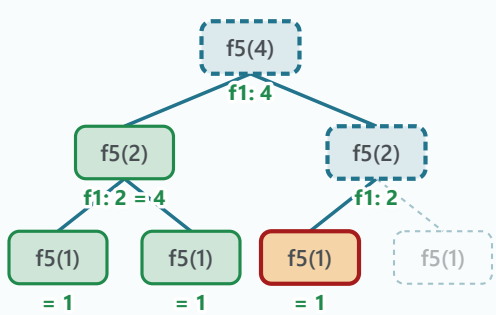
CALL STACK

`f5(2)`

`f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → base case, returns 1.

Step 10 / 13

VARIABLES

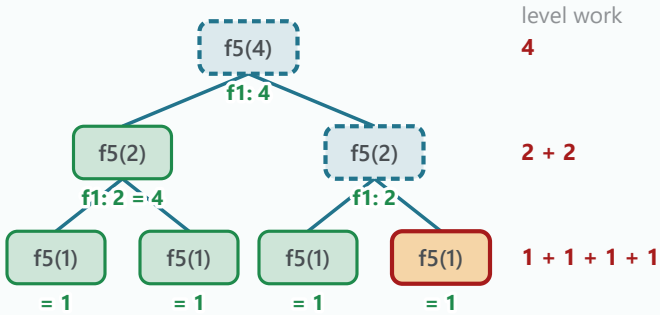
- `n = 1`
- `f1(n) = –`
- `result = 1`

CALL STACK

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
▶   if (n <= 1) {
▶     return 1;
4   }
5   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



`f5(1)` : n ≤ 1 → base case, returns 1.

Step 11 / 13

VARIABLES

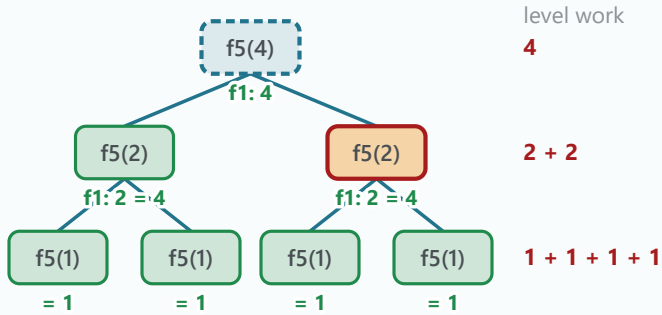
- `n = 1`
- `f1(n) = –`
- `result = 1`

CALL STACK

- `f5(1)`
- `f5(2)`
- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



f5(2) = f1(2) + f5(1) + f5(1) = 2 + 1 + 1 = **4** is returned.

Step 12 / 13

VARIABLES

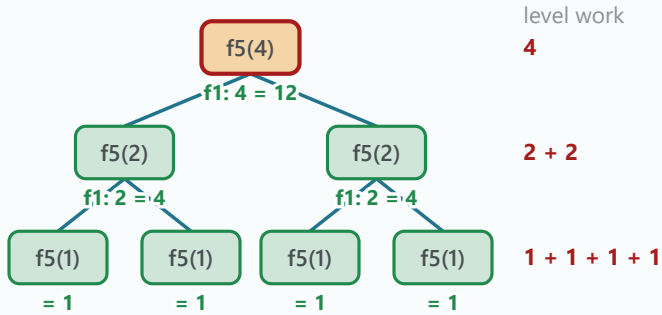
- `n = 2`
- `f1(n) = 2`
- `result = 4`

CALL STACK

- `f5(2)`
- `f5(4)`

f5(4): Tracing the Calls

```
1 public int f5(int n) {
2   if (n <= 1) {
3     return 1;
4   }
▶   return f1(n) + f5(n / 2) + f5(n / 2);
6 }
```



f5(4) = 4 + 4 + 4 = **12**. Each level does n = 4 work: $\log_2 4 = 2$ levels · 4 + 4 leaves → n $\log_2$ n + n. The recurrence is 2T(n/2) + n: **Θ(n log n)**.

Step 13 / 13

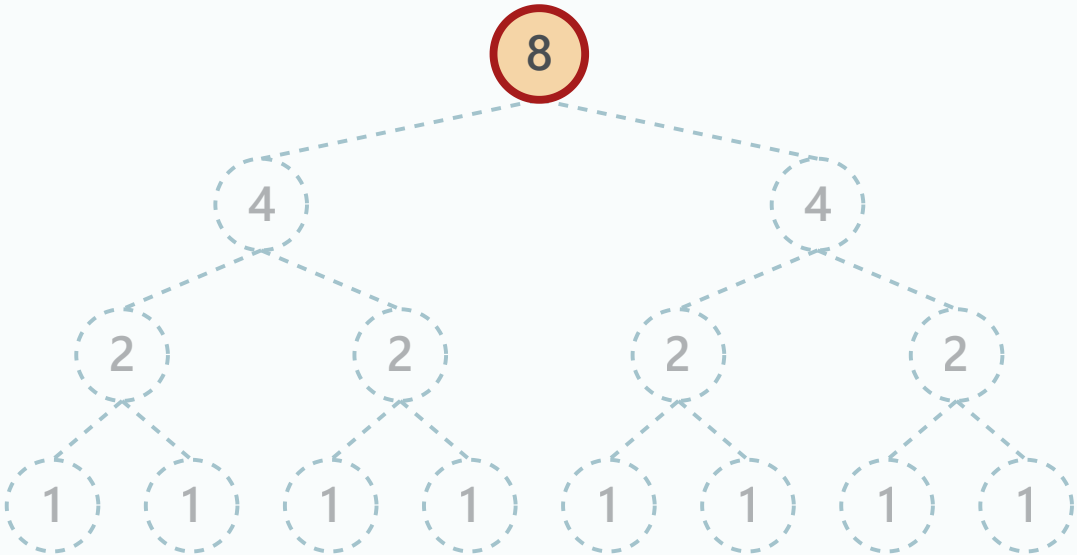
VARIABLES

- `n = 4`
- `f1(n) = 4`
- `result = 12`

CALL STACK

- `f5(4)`

f5(8): Recursion Tree



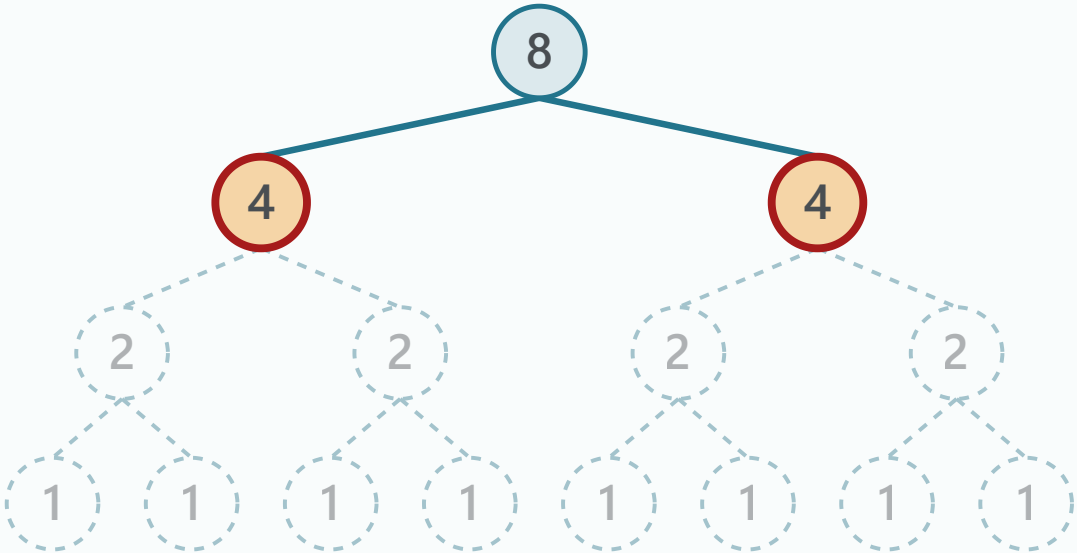
8

Root: inside `f5(8)`, `f1(8)` takes 8 steps.

Step 1 / 4

Each of the $\log_2 n$ levels does n work. In Java: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

f5(8): Recursion Tree



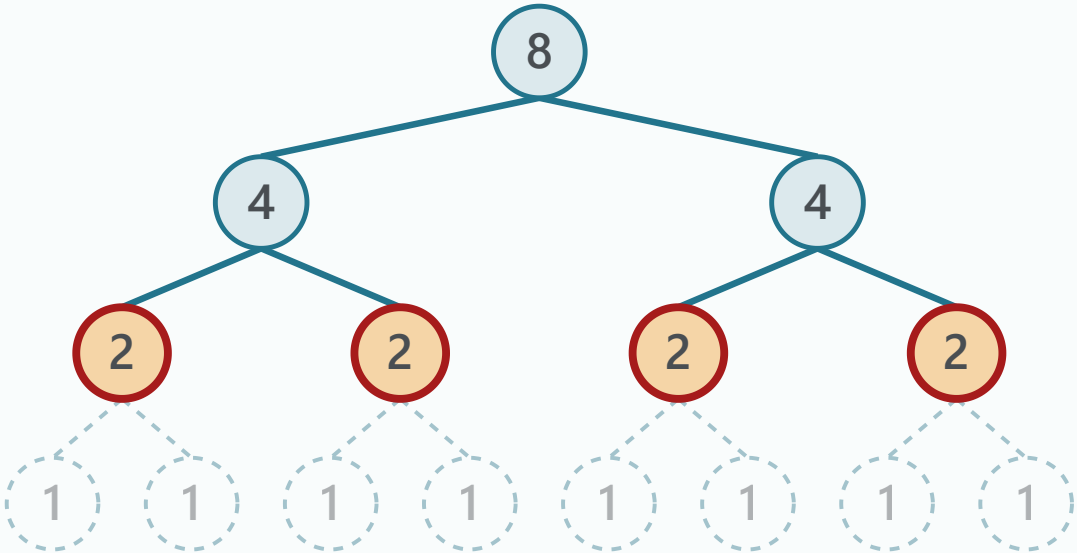
8

4 + 4 = 8

Level 1: two `f5(4)`, each with f1(4) = 4 steps → 8 in total.

Each of the $\log_2 n$ levels does n work. In Java: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

f5(8): Recursion Tree



8

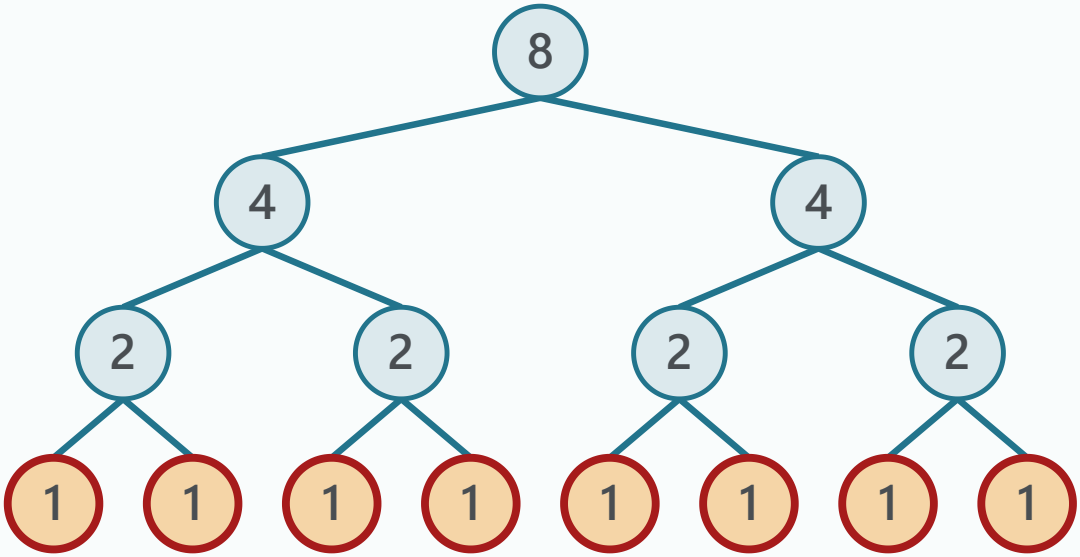
4 + 4 = 8

4 · 2 = 8

Level 2: four `f5(2)`, 2 steps each → again 8. Every level does **n** work.

Each of the $\log_2 n$ levels does n work. In Java: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

f5(8): Recursion Tree



8

4 + 4 = 8

4 · 2 = 8

8 · 1 = 8

Leaves: 8 × 1. Total $3 \cdot 8 + 8 = 32 = n \log_2 n + n \rightarrow$ **$\Theta(n \log n)$**. Java: f5(8) = 32.

Step 4 / 4

Each of the $\log_2 n$ levels does n work. In Java: f5(16) = 80 = 16 · 4 + 16, f5(32) = 192 = 32 · 5 + 32.

f5: Solving with the Master Theorem

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

Of the form T(n) = aT(n/b) + f(n).

Step 1 / 4

	f4	f5
Calls	2 × n/2	2 × n/2
Extra work f(n)	O(1)	Θ(n)
c vs $\log_b a$	0 < 1	1 = 1
Case	1	2
Result	Θ(n)	Θ(n log n)

f5: Solving with the Master Theorem

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

Of the form T(n) = aT(n/b) + f(n).

$$a = 2, b = 2, f(n) = \Theta(n)$$

Same parameters as f4; only f(n) has changed.

Step 2 / 4

	f4	f5
Calls	2 × n/2	2 × n/2
Extra work f(n)	O(1)	Θ(n)
c vs $\log_b a$	0 < 1	1 = 1
Case	1	2
Result	Θ(n)	Θ(n log n)

f5: Solving with the Master Theorem

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

Of the form T(n) = aT(n/b) + f(n).

$$a = 2, b = 2, f(n) = \Theta(n)$$

Same parameters as f4; only f(n) has changed.

$$\log_b a = \log_2 2 = 1, \quad c = 1$$

f(n) = Θ(n) → c = 1 = $\log_b$ a.

Step 3 / 4

	f4	f5
Calls	2 × n/2	2 × n/2
Extra work f(n)	O(1)	Θ(n)
c vs $\log_b$ a	0 < 1	1 = 1
Case	1	2
Result	Θ(n)	Θ(n log n)

f5: Solving with the Master Theorem

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

Of the form T(n) = aT(n/b) + f(n).

$$a = 2, b = 2, f(n) = \Theta(n)$$

Same parameters as f4; only f(n) has changed.

$$\log_b a = \log_2 2 = 1, \quad c = 1$$

f(n) = Θ(n) → c = 1 = $\log_b$ a.

$$\text{Case 2: } T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

Equal exponents: **Θ(n log n)**. The same recurrence as the example earlier in the chapter.

Step 4 / 4

	f4	f5
Calls	2 × n/2	2 × n/2
Extra work f(n)	O(1)	Θ(n)
c vs $\log_b$ a	0 < 1	1 = 1
Case	1	2
Result	Θ(n)	Θ(n log n)

COMPARISON

f1–f6: Step Counts Measured in Java

Function	Recurrence / sum	n = 16	n = 32	Ratio	Result
`f1` return	$\Sigma\ 1 = n$	16	32	× 2	$\Theta(n)$
`f2` return	$\Sigma\ i^2$	1,240	10,416	× 8.4	$\Theta(n^3)$
`f3` calls	$2T(n-1) + c$	65,535	4,294,967,295	× 65,537	$\Theta(2^n)$
`f4` calls	$2T(n/2) + c$	31	63	× 2	$\Theta(n)$
`f5` return	$2T(n/2) + n$	80	192	× 2.4	$\Theta(n \log n)$
`f6` return	$i \leftarrow 2^i$	4	5	+ 1	$\Theta(\log^* n)$ (mathematical model; in Java $n \le 65536$)

(long counter; f3(32) ≈ $4.3 \cdot 10^9$ calls)

The ratio when n doubles gives away the growth class: $\Theta(n)$ → ×2, $\Theta(n \log n)$ → a bit more than ×2, $\Theta(n^3)$ → ×8, $\Theta(2^n)$ → $\times 2^n$.

COMPARISON

Interactive f1–f6 Step Counter

n = 16 n / 2 n × 2

Function	Step count	Relative size (log scale)
`f1` Θ(n)	16	
`f2` Θ(n^3)	1,240	
`f3` Θ(2^n)	65,535	
`f4` Θ(n)	31	
`f5` Θ(n log n)	95	
`f6` Θ(log* n)	4	

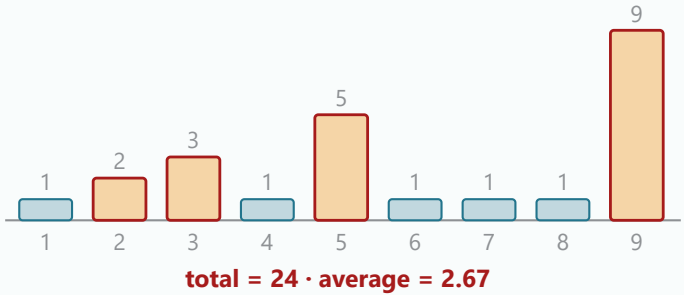
n = **16**: f3 alone makes **65,535** calls.

Steps: number of `x++` for f1, f2, f6; number of calls for f3, f4; calls + f1 loop steps for f5. Values follow the logic of the code (BigInt for f3).

COMPARISON

A Quick Look: Amortized Analysis

- **Amortized analysis** divides the total cost of a **sequence of operations** by the number of operations.
- Dynamic array: when full, its capacity **doubles** and every element is copied (expensive); other insertions cost 1.
- Over n insertions the copies total 1 + 2 + 4 + … < 2n → total < 3n → **amortized Θ(1)** per insertion.
- Unlike the average case, there is no probability: the guarantee holds for **every** sequence of operations.



9 insertions starting from capacity 1; orange bars include a resize (copy).

A single insertion can take Θ(n) in the worst case, yet the whole sequence costs 24 < 3 · 9 = 27 units. More detail: the aggregate, accounting and potential methods.

Summary

Topic	Key points
Complexity	time = number of basic operations T(n); space = extra memory (including the stack); best / worst / average case
Notation	O: upper, Ω: lower, Θ: tight bound; proved with c and n_0; drop constants and lower-order terms
Loops	add up iteration counts: f1 $\Sigma 1 = n \to \Theta(n)$; f2 $\Sigma i^2 \to \Theta(n^3)$; f6 $i \leftarrow 2^i \to \Theta(\log^* n)$ (mathematical model; in Java $n \le 65536$; O(log n) is not tight)
Recursion	set up a recurrence: number of calls, how the input shrinks, extra work; solve by expansion or a recursion tree
f3	$2T(n-1) + c = (2^n - 1)c \to \Theta(2^n)$; the Master Theorem does not apply
Master Theorem	$aT(n/b) + f(n)$: compare f(n) with $n^{\log_b a}$; leaves / balanced / root dominates
f4, f5	$2T(n/2) + 1 \to$ Case 1, $\Theta(n)$; $2T(n/2) + n \to$ Case 2, $\Theta(n \log n)$

SUMMARY

Common Mistakes

Equating Big O with the worst case

O is an upper bound; the worst case is a kind of input. "The best case is O(1)" is a perfectly valid statement.

Keeping constants and small terms

Don't write $O(3n + 3)$ or $O(n^2/3 + n)$: $O(n)$ and $O(n^2)$ are enough.

"Nested loops = n^2"

The inner loop's bound decides: in f2 it is i^2, hence $\Theta(n^3)$; an `i *= 2` loop runs only log n times.

"Two calls = exponential"

f3 $(n - 1)$ is $\Theta(2^n)$, f4 $(n/2)$ is $\Theta(n)$. How the input shrinks changes everything.

Applying the Master Theorem everywhere

It does not hold for recurrences with $T(n - 1)$, unequal splits, or $f(n) = n/\log n$.

Forgetting regularity in Case 3

$c > \log_b a$ is not enough; $a \cdot f(n/b) \leq k \cdot f(n)$ $(k < 1)$ must hold too.

SUMMARY

Sample Problems

1. Show that $2n + 10 = O(n)$ by finding c and n_0.
2. Solve $T(n) = 3T(n/3) + n$.
3. Solve $T(n) = 9T(n/3) + n$.
4. Solve $T(n) = T(n/2) + n$ (check regularity).
5. How many times does `for (i = 1; i < n; i *= 2) for (j = 0; j < i; j++) x++;` execute `x++` ?
6. What is the complexity of `for (i = n; i > 0; i /= 2) for (j = 0; j < n; j++) x++;` ?

▶ **Show answers**

SUMMARY

Exercises

1. If the inner condition in f2 were `j < i`, what would the complexity be? What would f2(5) return?
2. Write and solve the recurrence for `return g(n/2) + g(n/2) + g(n/2) + g(n/2);` (base case $n \le 1$).
3. If the last line of f3 were `return 2 * f3(n - 1);`, how would its return value and running time change?
4. Solve $T(n) = 8T(n/2) + n^2$ and $T(n) = 7T(n/2) + n^2$. Which algorithms do they describe?
5. Prove by contradiction that $n^2 \neq O(n)$.
6. If f6's loop were `for (i = 1; i < n; i *= 2)`, how many iterations would it make for n = 1000? Compare with f6(1000).

▶ **Answers / hints**

References

- Cormen, Leiserson, Rivest and Stein, *Introduction to Algorithms*, 4th ed., MIT Press, 2022 (Chapters 3 and 4).
- Sedgewick and Wayne, *Algorithms*, 4th ed., Addison-Wesley, 2011 (Section 1.4: Analysis of Algorithms).
- Kleinberg and Tardos, *Algorithm Design*, Pearson, 2005 (Chapters 2 and 5).
- D. E. Knuth, "Big Omicron and Big Omega and Big Theta", *SIGACT News* 8(2), 1976.
- J. L. Bentley, D. Haken and J. B. Saxe, "A General Method for Solving Divide-and-Conquer Recurrences", *SIGACT News* 12(3), 1980.

The End

Chapter 2: Algorithm Complexity

Previous: Chapter 1: Introduction · Next: Chapter 3: Sorting Algorithms