Home

WEEK 02: RUNNING TIMES (LECTURE)

Michael T. Gastner

2026-01-12

Home

INTRO

Home

LEARNING OBJECTIVES

By the end of this lecture, you should be able to:

1. Define key features of an algorithm, including input size and basic steps.
2. Express an algorithm’s running time as a function $T(n)$.
3. Use asymptotic notation (O, Ω, Θ) to describe and compare running times.
4. Derive the asymptotic running times of non-recursive algorithms by counting basic steps.
5. Recognize key properties of divide-and-conquer algorithms.
6. Solve divide-and-conquer recurrences using recursion trees and the Master Theorem.

Home

FROM WEEK 01 TO WEEK 02

Last week, we stated the following objectives for studying data structures and algorithms:

1. Is the computational procedure **correct**?
2. Does it always **terminate**?
3. Given a particular data structure as input, how much
    a. **time** does the procedure take?
    b. **memory** does the procedure use?

We addressed Questions 1 and 2 in Week 01. Today, we focus on Question 3a: **running times**.

WHAT DO WE MEAN BY “RUNNING TIME”?

In everyday language, an algorithm’s running time is the **time on a clock** (e.g., seconds).

In this course, we usually do **not** measure seconds directly, because they depend on:

- the computer and its CPU,
- the programming language and compiler,
- and many implementation details.

Instead, we use a **machine-independent model**:

Definition

The **running time** $\boldsymbol{T(n)}$ of an algorithm is the number of **basic steps** (e.g., comparisons, assignments, and arithmetic operations) it performs on an input of size n (e.g., the number of keys to sort).

Later, we will compare algorithms by how $T(n)$ **grows as $\boldsymbol{n}$ increases**.

WHAT COUNTS AS A "BASIC STEP"?

To make $T(n)$ meaningful, we use a simple cost model:

Definition

A **basic step** is either

- one operation on a value with a **fixed number of bits** (e.g., a 32-bit integer or a 64-bit floating-point number), or
- one access to a **single** array element $A[i]$, given the index i.

We assume that each basic step takes at most a constant amount of time, regardless of the input size n.

This model simplifies reality, but it lets us compare algorithms without depending on a particular CPU, compiler, or programming language.

HOW IS INPUT SIZE MEASURED?

The **input size** n describes how “big” the problem instance is. How we define it depends on the problem.

Examples:

- **Sorting:** n = number of keys in the array.
- **Matrix multiplication:** n = number of rows/columns of an $n \times n$ matrix. The input contains n^2 numbers, not n.
- **Graph algorithms:** Size is usually described by two parameters: the number of vertices and the number of edges.

Home

RUNNING TIMES

RUNNING TIMES OF NON-RECURSIVE ALGORITHMS

For **non-recursive** algorithms, we will derive $T(n)$ as follows:

1. Specify what the input size n measures for this problem.
2. Count how many basic steps are executed, as a function of n.
3. Sum the counts to obtain $T(n)$.

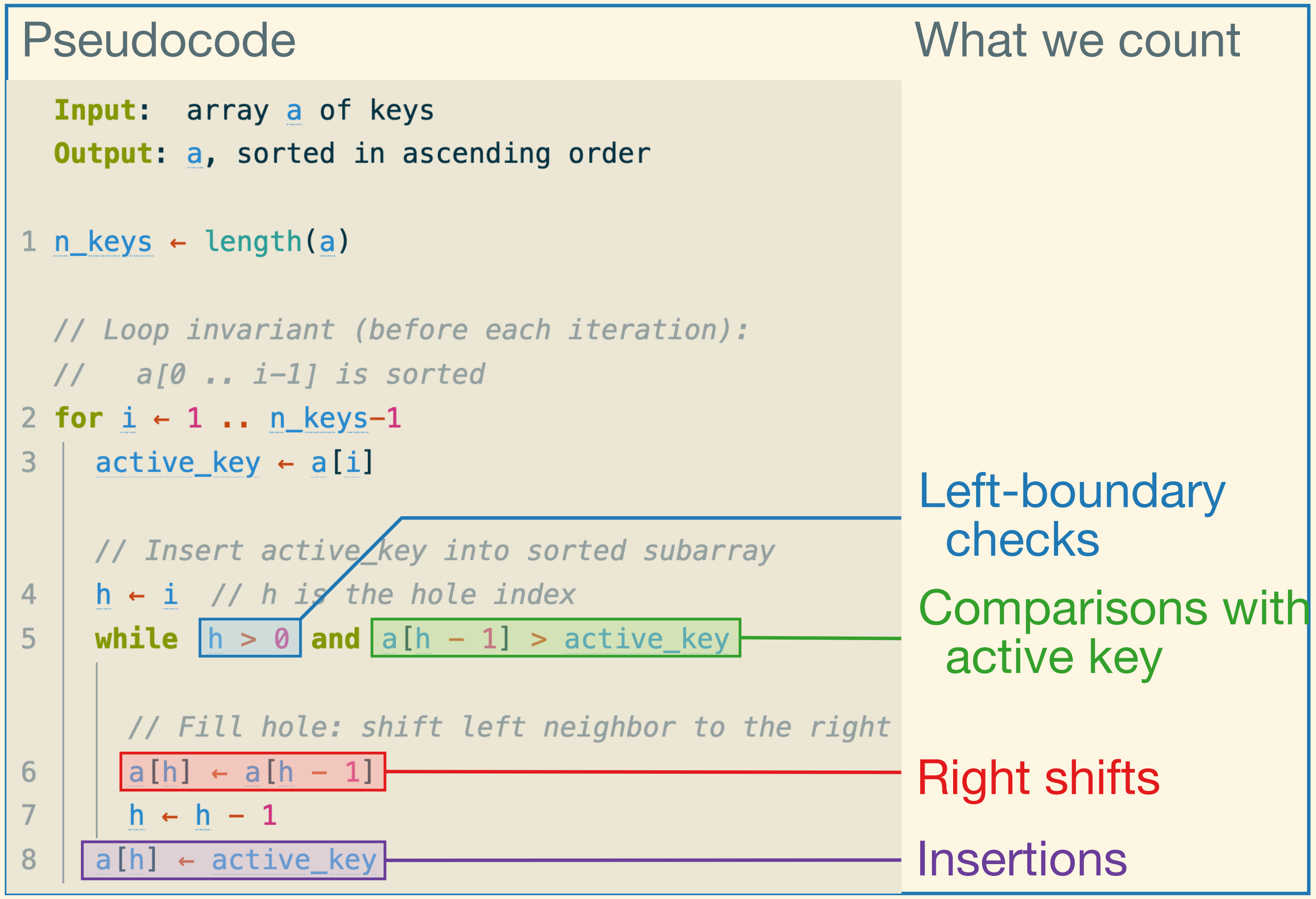
Rule of thumb: Nested loops usually multiply counts.

For example, if an outer loop runs n times and the inner loop runs n times for each outer iteration, then the inner-loop body executes n^2 times:

```
Input: array a with n elements

for i ← 0 .. n - 1
    for j ← 0 .. n - 1
        a[i] ← a[i] + 1
```

EXAMPLE: INSERTION SORT



```
Input:  array a of keys
Output: a, sorted in ascending order

n_keys ← length(a)

// Loop invariant (before each iteration):
//   a[0 .. i−1] is sorted
for i ← 1 .. n_keys−1
    active_key ← a[i]

    // Insert active_key into sorted subarray
    h ← i  // h is the hole index
    while h > 0 and a[h − 1] > active_key

        // Fill hole: shift left neighbor to the right
        a[h] ← a[h − 1]
        h ← h − 1
    a[h] ← active_key
```

The highlighted basic steps are not the only ones, but they are enough to characterize the running time.

Home

INSERTION SORT ON AN ARRAY SORTED IN ASCENDING ORDER

Open https://apps.michael-gastner.com/insertion-sort/ and choose ascending-order input from the example dropdown. You can edit the input by typing in the text box. Increase the input size (try 5, 10, and 20 keys) and observe how the **Final** counts of the following metrics change:

- Left-boundary checks
- Comparisons with active key
- Right shifts
- Insertions

Do the observations match your expectations?

IF THE INPUT IS ALREADY IN ASCENDING ORDER, $T(n)$ IS LINEAR

```
  Input:  array a of keys
  Output: a, sorted in ascending order

1 n_keys ← length(a)

  // Loop invariant (before each iteration):
  //   a[0 .. i−1] is sorted
2 for i ← 1 .. n_keys−1
3   active_key ← a[i]

    // Insert active_key into sorted subarray
4   h ← i  // h is the hole index
5   while  h > 0  and  a[h − 1] > active_key

      // Fill hole: shift left neighbor to the right
6     a[h] ← a[h − 1]
7     h ← h − 1
8   a[h] ← active_key
```

If a is in ascending order, then the `while` condition in line 5 fails immediately because `a[h - 1] > active_key` is false. Thus, **no right shifts** (line 6) are ever performed.

The outer loop (lines 2–8) runs $n-1$ times. Each iteration performs:

- one **left-boundary check** (line 5),
- one **comparison with the active key** (also line 5),
- one **insertion** (line 8).

Therefore, the total number of steps scales roughly in proportion to n, consistent with the app output.

Home

INSERTION SORT ON AN ARRAY SORTED IN DESCENDING ORDER

Open https://apps.michael-gastner.com/insertion-sort/ again. This time, choose descending-order input from the example dropdown. Again, increase the input size (try 5, 10, and 20 keys) and observe how the **Final** counts of the following metrics change:

- Left-boundary checks
- Comparisons with active key
- Right shifts
- Insertions

Which metric grows fastest as n increases?

Home

COUNTING BASIC STEPS FOR INPUT IN DESCENDING ORDER

```
  Input:  array a of keys
  Output: a, sorted in ascending order

1 n_keys ← length(a)

  // Loop invariant (before each iteration):
  //   a[0 .. i−1] is sorted
2 for i ← 1 .. n_keys−1
3 │ active_key ← a[i]
  │
  │ // Insert active_key into sorted subarray
4 │ h ← i  // h is the hole index
5 │ while  h > 0  and  a[h − 1] > active_key
  │ │
  │ │ // Fill hole: shift left neighbor to the right
6 │ │ a[h] ← a[h − 1]
7 │ │ h ← h − 1
8 │ a[h] ← active_key
```

When a is in descending order and the outer iteration i starts, the `active_key` is smaller than all i keys to its left. Thus, the `while` loop shifts each key one position to the right before inserting `active_key` at index 0, resulting in:

- $i+1$ **left-boundary checks** (line 5; $+1$ because we check `h > 0` once more when h becomes 0),
- i **comparisons with active key** (line 5; no comparison when h becomes 0 due to short-circuit evaluation),
- i **right shifts** (line 6),
- one **insertion** (line 8).

Home

IF THE INPUT IS IN DESCENDING ORDER, $T(n)$ IS QUADRATIC

Let us sum the comparisons with the active key over all outer iterations $i = 1, 2, \ldots, n-1$. Similar reasoning applies to the left-boundary checks and right shifts:

$$\text{Number of comparisons} = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2},$$

which grows roughly in proportion to n^2 as n increases. For large n, this will exceed the linear growth of insertions.

Therefore, the total number of steps grows roughly in proportion to n^2, consistent with the app output.

Home

FOCUS ON THE DOMINANT GROWTH TREND

On the previous slides, we counted basic steps for insertion sort and observed two different growth patterns:

- If the input is in **ascending** order, the **step counts** grow roughly in proportion to n.
- If it is in **descending** order, they grow approximately in proportion to n^2.

When we compare algorithms, we care most about what happens as the input gets large. In that regime, the **dominant growth trend** matters more than small details such as the “$+1$” effect in the left-boundary checks.

ASYMPTOTICS

Home

ASYMPTOTIC NOTATIONS: O, Ω, AND Θ

To describe the **dominant growth trend** of a running-time function $T(n)$ for large n, we compare it to a simpler function $g(n)$. We ask whether a constant multiple of $g(n)$ provides an upper bound, a lower bound, or both for $T(n)$ once n is sufficiently large. We will make this notion mathematically precise on the following slides.

Notation	**Pronunciation**	**Bound provided by $g(n)$**
$O(g(n))$	Big O	Upper
$\Omega(g(n))$	Big Omega	Lower
$\Theta(g(n))$	Theta	Tight (upper and lower)

For instance, the upcoming definitions allow us to state the insertion-sort results as:

- Best case (ascending order): $\Theta(n)$
- Worst case (descending order): $\Theta(n^2)$

Home

O-NOTATION: DEFINITION

The O-notation describes the **upper bound** of a function's growth rate.

Definition

Let $g(n)$ be a function such that $g(n) \geq 0$ for all $n \geq n_0$ (for some n_0). Then $O(g(n))$ denotes the set of functions

$$\begin{aligned} O(g(n)) = \{f(n) :\ & \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ & |f(n)| \leq cg(n) \text{ for all } n \geq n_0\}. \end{aligned}$$

O-NOTATION: EXAMPLE

Insertion sort has a worst-case running time of the form $T(n) = qn^2 + rn + s$ for some constants $q \neq 0, r$, and s.

Choose $c = |q| + |r| + |s|$ and $n_0 = 1$. Then, for all $n \geq n_0$,

$$\begin{aligned} |T(n)| &= |qn^2 + rn + s| \\ &\leq |q|n^2 + |r|n + |s| \\ &\leq |q|n^2 + |r|n^2 + |s|n^2 \\ &= (|q| + |r| + |s|)n^2 = cn^2, \end{aligned}$$

where the first inequality follows from the triangle inequality, and the second inequality holds because $n \geq 1$ implies $n \leq n^2$.

Hence, $T(n) \in O(n^2)$.

Home

Ω-NOTATION: DEFINITION

The Ω-notation describes the **lower bound** of a function's growth rate.

Definition

Let $g(n)$ be a function such that $g(n) \geq 0$ for all $n \geq n_0$ (for some n_0). Then $\Omega(g(n))$ denotes the set of functions

$$\begin{aligned}\Omega(g(n)) = \{f(n) :\ & \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ & |f(n)| \geq cg(n) \text{ for all } n \geq n_0\}.\end{aligned}$$

Home

Ω-NOTATION: EXAMPLE

Consider again the insertion sort's worst-case running time $T(n) = qn^2 + rn + s$ with $q \neq 0$. Choose $c = |q|/4$ and $n_0 = \max\left(1,\ 2|r/q|,\ 2\sqrt{|s/q|}\right)$. Then, for all $n \geq n_0$, we have $|r|n \leq \frac{|q|}{2}n^2$ and $|s| \leq \frac{|q|}{4}n^2$. Hence, by the reverse triangle inequality,

$$\begin{aligned} |T(n)| &= |qn^2 + rn + s| \\ &\geq |q|n^2 - |r|n - |s| \\ &\geq |q|n^2 - \frac{|q|}{2}n^2 - \frac{|q|}{4}n^2 \\ &= \frac{|q|}{4}n^2 = cn^2. \end{aligned}$$

Therefore, $T(n) \in \Omega(n^2)$.

Home

Θ-NOTATION: DEFINITION

The Θ-notation describes the **tight bound** of a function's growth rate.

Definition

Let $g(n)$ be a function such that $g(n) \geq 0$ for all $n \geq n_0$ (for some n_0). Then $\Theta(g(n))$ denotes the set of functions

$$\Theta(g(n)) = \{f(n) : \begin{array}{l} \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that} \\ c_1 g(n) \leq |f(n)| \leq c_2 g(n) \text{ for all } n \geq n_0\}. \end{array}$$

Instead of writing $f(n) \in \Theta(g(n))$, it is common to use the shorthand $f(n) = \Theta(g(n))$. The same convention applies to O and Ω notations.

Home

RELATIONSHIP BETWEEN Θ, O, AND Ω

Theorem

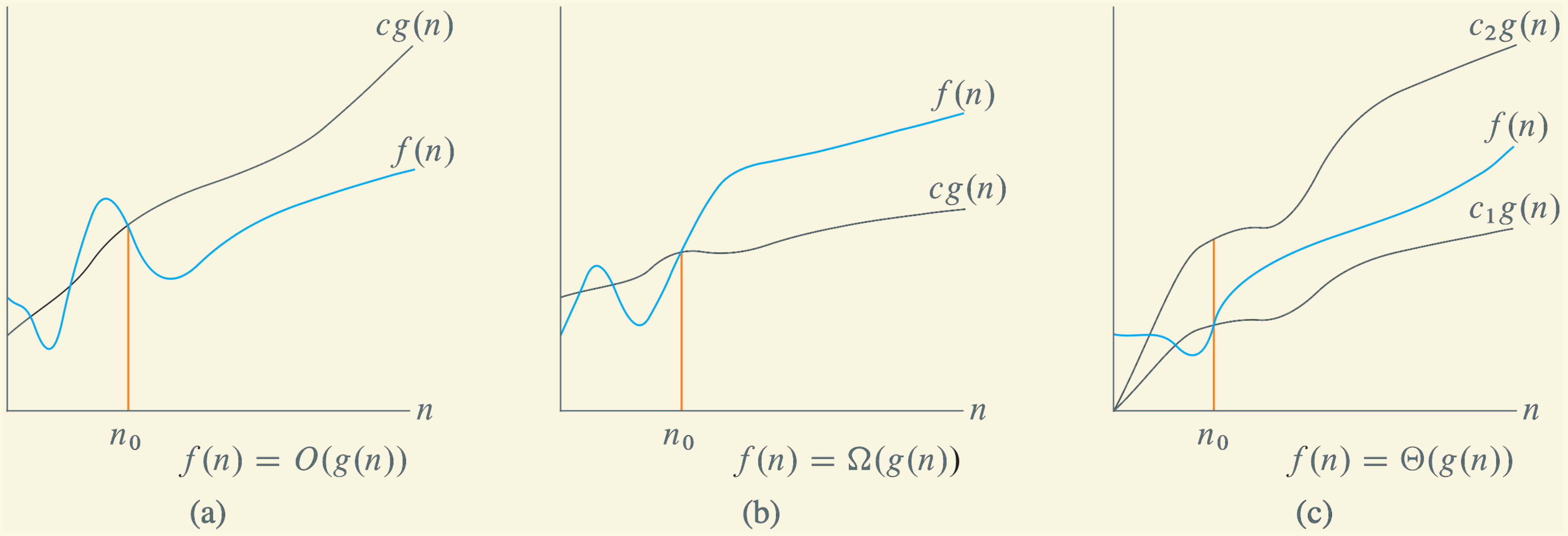
For any two functions $f(n)$ and $g(n)$, the following equivalence holds:

$$f(n) \in \Theta(g(n)) \quad \text{if and only if} \quad f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n)).$$

For instance, we have shown that the worst-case running time of insertion sort satisfies $T(n) \in O(n^2)$ and $T(n) \in \Omega(n^2)$. Therefore, $T(n) \in \Theta(n^2)$.

ILLUSTRATION OF ASYMPTOTIC NOTATIONS

Figure 3.2 from Cormen *et al.* (2022):



STATE THE TIGHTEST AND SIMPLEST KNOWN BOUND

As a matter of style:

- If you can prove that a function is $\Theta(g(n))$, state $\Theta(g(n))$; do not use O or Ω instead.
- Otherwise, state the tightest bounds you can justify. For example, any function that is $O(n^2)$ is also $O(n^3)$, but $O(n^2)$ is more informative because it provides a tighter upper bound.
- Use the simplest functions possible. For instance, if a function is $\Theta(n^2)$, it is also $\Theta(9n^2 - 2n)$, but $\Theta(n^2)$ is clearer and more concise.
- When we write $f(n) = O(g(n))$, the "$=$" sign is shorthand for set membership; do not treat it as algebraic equality.

Home

COMMON GROWTH RATES (FROM SMALLEST TO LARGEST)

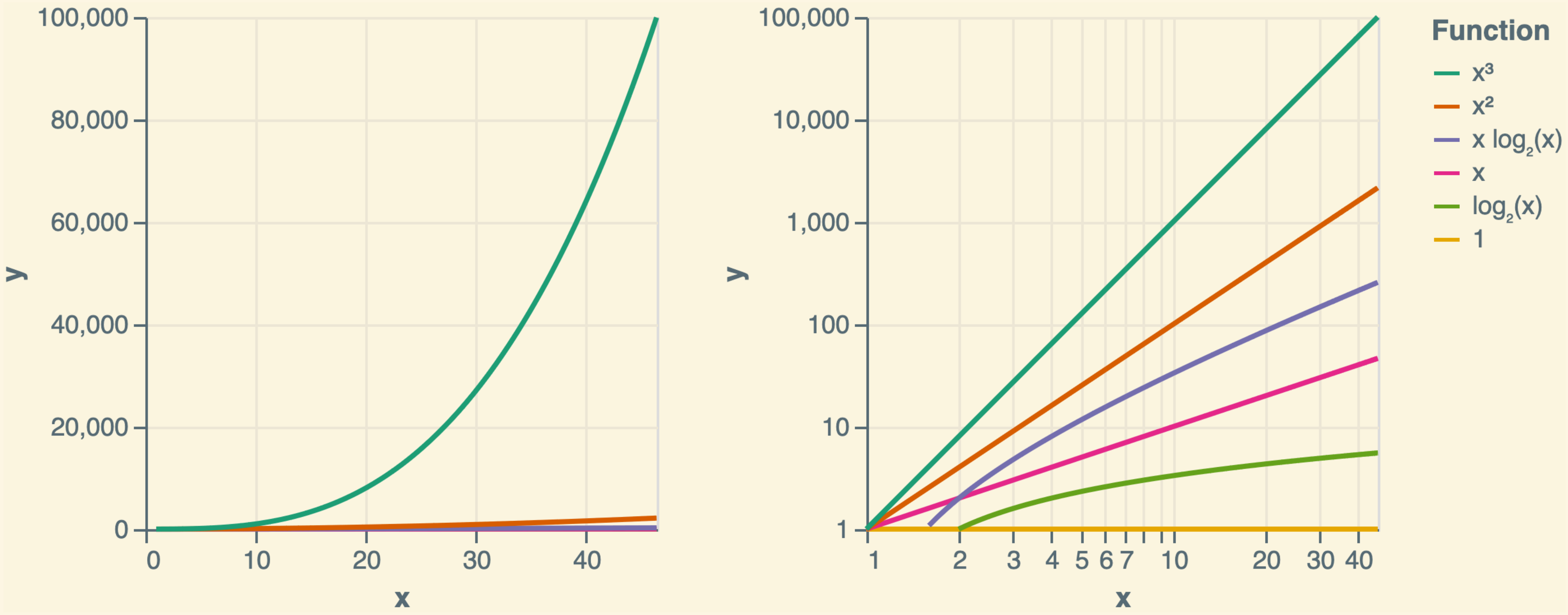
For sufficiently large n, the following functions grow in this order:

- 1 (constant)
- $\log n$ (logarithmic). The base of $\log n$ is irrelevant in asymptotic notation because changing the base only introduces a constant factor.
- $(\log n)^k$ for $k > 0$ (polylogarithmic)
- n^z for $0 < z < 1$ (sublinear power law, e.g., $\sqrt{n}$)
- n (linear)
- $n \log n$ (linearithmic)
- n^k for $k > 1$ (polynomial; includes $n^2, n^3, \ldots$)
- a^n for $a > 1$ (exponential; e.g., 2^n)
- $n!$ (factorial)

Home

PLOTS OF COMMON FUNCTIONS

The plots below illustrate functions commonly encountered in running-time analysis. The left plot uses linear scales for both the x-axis and y-axis, whereas the right plot employs logarithmic scales.



Home

QUIZ: ORDERING BY ASYMPTOTIC GROWTH RATES

Rank the following functions of n by order of growth; that is, find an arrangement (g_1, g_2, g_3, g_4) of the functions from slowest-growing to fastest-growing:

a. $\log n$

b. n^2

c. $n \log n$

d. n

Home

DIVIDE AND CONQUER

DIVIDE-AND-CONQUER ALGORITHMS

In Week 01, we defined a **recursive algorithm** as one that calls itself.

So far, we have derived the running time of a non-recursive algorithm: insertion sort. As we saw in this example, non-recursive algorithms usually require counting how often each operation is executed.

For recursive algorithms, different techniques are required. **Divide-and-conquer algorithms** are a prominent class of recursive algorithms, and several methods exist for analyzing their running times.

A divide-and-conquer algorithm typically consists of three key steps, listed on the following slide.

Home

THREE STEPS OF DIVIDE-AND-CONQUER ALGORITHMS

1. **Divide:** Partition the problem into one or more subproblems, each representing a smaller instance of the same problem.
2. **Conquer:** Solve the subproblems recursively until they reach a base case that can be solved directly.
3. **Combine:** Integrate the solutions of the subproblems to obtain a solution to the original problem.

MERGE SORT: AN EXAMPLE OF DIVIDE-AND-CONQUER

You implemented **merge sort** in Week 01. It is a paradigmatic example of a divide-and-conquer algorithm:

```
Input:  array a of keys
Output: a, sorted in ascending order

n ← length(a)
sort(a, 0, n − 1)

procedure sort(a, l, r)

  | // l and r are the left and right boundary indices
  | // respectively; m is the midpoint
  | // Postcondition: a[l .. r] sorted
  | if l ≥ r
  | |  return
  | m ← floor((l + r) / 2)
  | sort(a, l, m)
  | sort(a, m + 1, r)
  | merge(a, l, m, r)
```

Base case:
subarray of length n=1

Divide the subarray in half

Conquer: 2 recursive calls

Combine: $\theta(n)$ each time

Home

RECURRENCE RELATION FOR MERGE SORT'S RUNNING TIME

Summarizing the time required by the divide, conquer, and combine steps, we obtain the following recurrence relation for merge sort's running time. For simplicity, we assume that the input size n is a power of two:

$$T(n) = 2T(n/2) + \Theta(n). \tag{1}$$

Note that Equation 1 implies that there exist constants $c_1, c_2 > 0$ such that

$$c_1 n \leq |T(n) - 2T(n/2)| \leq c_2 n$$

for all sufficiently large n.

Next, we will represent the recurrence relation as a tree and derive $T(n)$ from it.

Home

RECURSION TREES

Home

RECURSION TREES

Definition

A **recursion tree** visualizes a recurrence relation by expanding the recursive calls into a tree. Each node represents one subproblem, labeled with the work done **outside the recursive calls** at that call (the "combine" cost).

For merge sort, Equation 1 expands as shown on the right.

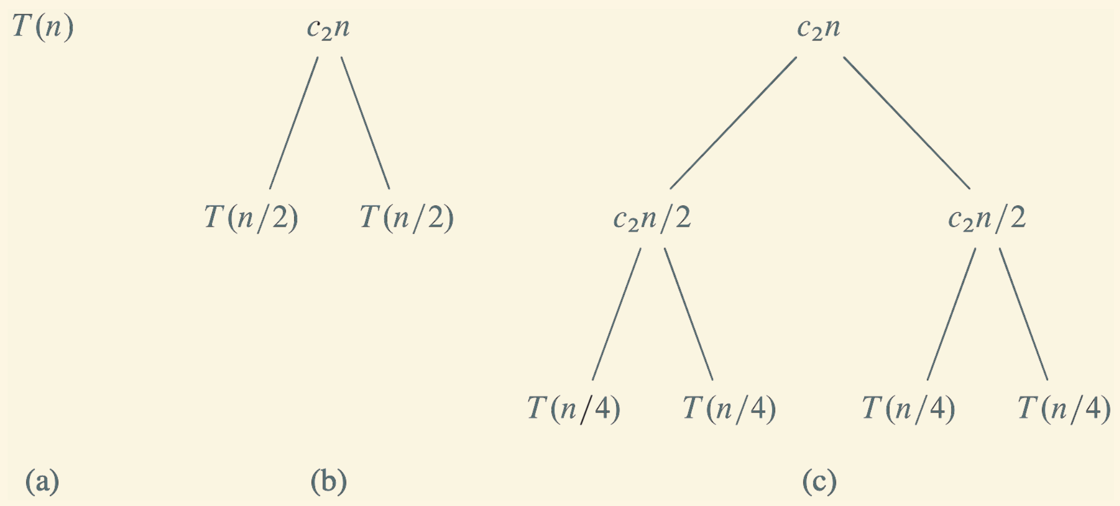
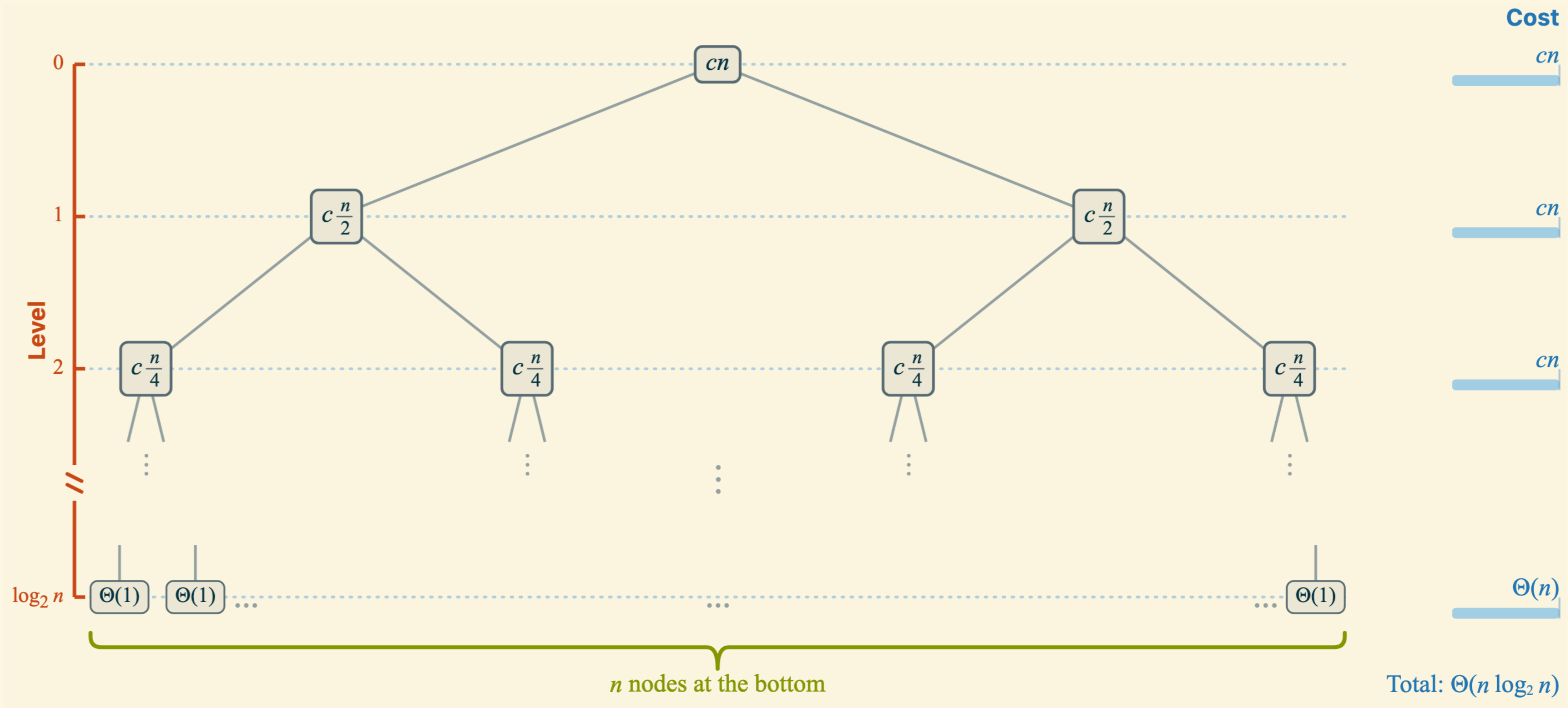


Figure 2.5(a)–(c) from Cormen *et al.* (2022)

Home

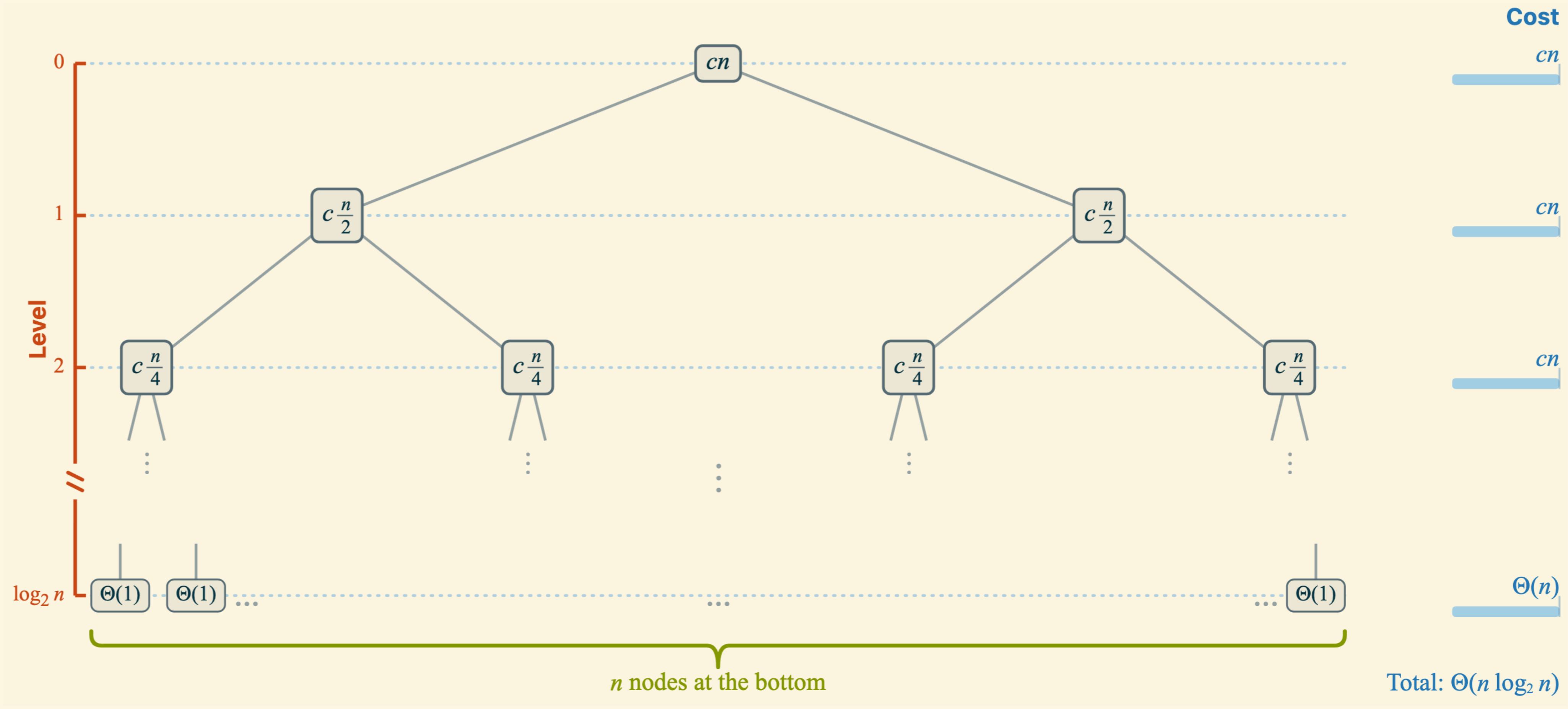
RECURSION TREE FOR MERGE SORT

The tree continues until the base case is reached, where the subproblem size equals 1. Merge sort needs $\log_2 n$ levels to reach the base case because the subproblem size halves at each level.



Home

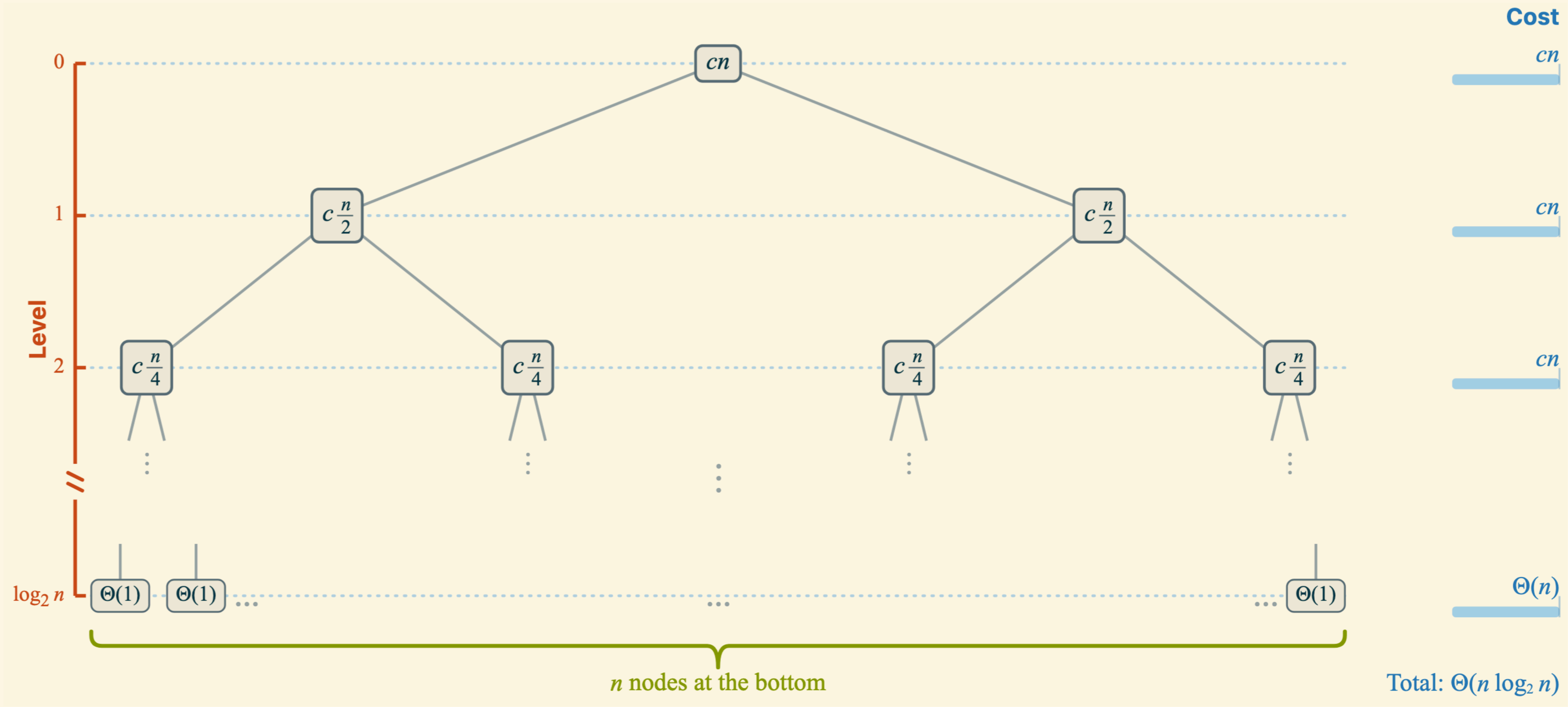
COST PER NON-LEAF LEVEL



At each level i between 0 and $\log_2 n - 1$ of the recursion tree, the total cost is $\Theta(n)$ because there are 2^i subproblems at that level, each requiring $\Theta(n/2^i)$ time to merge. Thus, the total cost across levels 0 to $\log_2 n - 1$ is $\log_2 n \cdot \Theta(n) = \Theta(n \log n)$.

Home

COST OF THE LEAF LEVEL AND CONCLUSION



The cost of the bottom level is $\Theta(1) \cdot n = \Theta(n)$, which is negligible compared to the combined cost of the non-leaf levels. We conclude that **the running time of merge sort is $\boldsymbol{\Theta(n \log n)}$.**

Home

MASTER THEOREM

MASTER THEOREM

Although recursion trees help build intuition, they do not by themselves provide a mathematically rigorous method for solving recurrences. The Master Theorem provides an alternative technique with mathematical guarantees.

Theorem

Let $a \geq 1$ and $b > 1$ be constants, and let $f(n)$ be a function with $f(n) \geq 0$ for all sufficiently large n. Consider recurrences of the form $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$ for values of n for which the recurrence is defined (e.g., $n = b^k$). Let $p = \log_b a$.

1. If $f(n) \in O\big(n^{p-\varepsilon}\big)$ for some $\varepsilon > 0$, then $T(n) \in \Theta\big(n^p\big)$.
2. If there exists $k \geq 0$ such that $f(n) \in \Theta\big(n^p \log^k n\big)$, then $T(n) \in \Theta\big(n^p \log^{k+1} n\big)$.
3. If $f(n) \in \Omega\big(n^{p+\varepsilon}\big)$ for some $\varepsilon > 0$ and there exist constants $c < 1$ and n_0 such that for all $n \geq n_0$ we have $a\,f\left(\frac{n}{b}\right) \leq c\,f(n)$, then $T(n) \in \Theta\big(f(n)\big)$.

Home

APPLYING THE MASTER THEOREM TO MERGE SORT

Recall the merge sort recurrence: $T(n) = 2T(n/2) + \Theta(n)$, so $a = 2$, $b = 2$, and $f(n) = n$.

First, we compute $p = \log_b a = \log_2 2 = 1$. Case 2 of the Master Theorem applies with $k = 0$.

Therefore, the total running time is $T(n) = \Theta\left(n^1 \log^{0+1} n\right) = \Theta(n \log n)$, consistent with our earlier result.

CONCLUSION

Home

SUMMARY OF KEY LEARNING OUTCOMES

1. We defined the running time $T(n)$ of an algorithm as the number of basic steps it performs on an input of size n. ↪
2. We introduced the O, Ω, and Θ notations to describe and compare asymptotic growth rates. ↪
3. We derived asymptotic running times of non-recursive algorithms by counting basic steps and summing them over loop iterations. ↪
4. We compared common growth rates (e.g., n, $n \log n$, n^2) to reason about which terms dominate for large inputs. ↪
5. We derived the recurrence relation for merge sort. ↪
6. We solved the merge-sort recurrence using these techniques:

   a. Recursion trees ↪

   b. Master Theorem ↪

   For merge sort, this analysis yields $T(n) \in \Theta(n \log n)$. ↪

Home

OUTLOOK

In the lab later this week, you will implement several divide-and-conquer algorithms in C++. You will also analyze their running times experimentally and compare the results to the theoretical predictions.

Home

BIBLIOGRAPHY

- Cormen, T.H. *et al.* (2022) *Introduction to algorithms*. 4th ed. MIT Press.