# COMSC 2043
# Induction and Recursion

Student Workbook

Jeremy Evert
Southwestern Oklahoma State University

October 28, 2025

# Contents

# Chapter 1

# Foundations of Induction and Recursion

This chapter introduces the central ideas of **mathematical induction** and **recursion** as presented in Chapter 5 of Kenneth Rosen's *Discrete Mathematics and Its Applications*.

## 1.1 The Big Picture

Mathematical induction and recursion are two sides of the same elegant coin. Induction is how we *prove* things about a process that repeats. Recursion is how we *define* that process.

We use induction to reason that what works for one step will work for the next. We use recursion to build structures or compute results by defining a problem in terms of smaller instances of itself.

## 1.2 Key Ideas from Rosen's Chapter 5

- **Basis Step:** Prove that a statement is true for an initial value (usually $n = 0$ or $n = 1$).
- **Inductive Step:** Assume it is true for $n = k$ (the *inductive hypothesis*) and prove it for $n = k + 1$.
- **Strong Induction:** Sometimes we assume it's true for *all* previous cases up to $k$ to prove it for $k + 1$.
- **Recursive Definitions:** A way to define sets, sequences, or functions in terms of themselves.

- **Structural Induction:** A generalization used for recursively defined structures like trees or expressions.

## 1.3 Why This Matters

Induction teaches us to trust the domino effect: if one falls and the rule is consistent, they all fall. Recursion lets us *build the dominoes* themselves.

They are the grammar and logic behind everything from factorial functions to sorting algorithms to proofs of algorithmic correctness.

## 1.4 Example: The Factorial Function

The factorial of $n$, written $n!$, is defined recursively:

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n > 0 \end{cases}$$

We can prove by induction that $n! \geq 2^{n-1}$ for all $n \geq 1$.

**Proof (sketch):**

- **Base case:** When $n = 1$, we have $1! = 1 \geq 2^0 = 1$ ✓
- **Inductive step:** Assume $k! \geq 2^{k-1}$ for some integer $k \geq 1$. Then

$$(k+1)! = (k+1)\,k! \geq (k+1)2^{k-1} \geq 2^k.$$

Therefore, the inequality holds for $k+1$, completing the induction.

## 1.5 A Student Challenge

> "Induction is not a leap of faith — it's a method of climbing an infinite ladder, one rung at a time."

**Challenge:** Write your own recursive function in Python that computes $n!$, and then write a proof by induction showing why it works for all $n \geq 0$.

## 1.6 Checkpoint Questions

1. What are the two main steps of a proof by induction?
2. How is recursion related to induction?
3. Give a real-world example of a recursive process.
4. Can every recursive definition be proven correct using induction?

# Let's Talk About It!

**1. What are the two main steps of a proof by induction?**
Induction is a two-act play:

- The *Basis Step* is your opening scene—prove the statement works for the smallest case (often $n = 0$ or $n = 1$).
- The *Inductive Step* is the domino push—assume it works for one case ($n = k$) and show that it must work for the next ($n = k + 1$).

Once both are in place, you've built a logical escalator that carries truth upward forever.

**2. How is recursion related to induction?**
Recursion and induction are best friends—like code and proof, or peanut butter and jelly. Recursion *defines* a process step by step ("to find $n!$, multiply $n$ by the factorial of one less"). Induction *proves* that this step-by-step definition behaves exactly as intended for all values. One builds; the other validates. Together they make math and programming hum in harmony.

**3. Give a real-world example of a recursive process.**
Think of a set of nested Russian dolls, or a friend who keeps telling you to "just do what I did, but one less time." Climbing a staircase, sorting a deck by repeatedly finding the smallest card, baking cookies batch by batch—all are recursive stories. Each new step is built by repeating a smaller, familiar step.

**4. Can every recursive definition be proven correct using induction?**
Almost! If your recursive definition is well-behaved (that is, every call eventually hits a simple base case), induction can usually prove it correct. But if your recursion never bottoms out—say, a function that calls itself forever—then induction has nothing solid to stand on. So the moral: always make sure your recursion has a comfy base case to land on, or your proof (and your program) will fall into an infinite abyss.

**Big Picture Reflection:**
Induction and recursion are more than tricks—they're patterns for thinking. When you write code or tackle life's big problems, remember:

> Start small, trust the process, and let logic do the climbing.

# Chapter 2

# Recursive Algorithms

Recursion is the heartbeat of algorithmic thinking. It allows us to describe a complex process in terms of smaller, self-similar pieces. In this chapter, we'll explore how to turn recursive definitions into recursive algorithms.

## 2.1 From Definition to Algorithm

A recursive algorithm solves a problem by reducing it to smaller instances of the same problem. Every recursive algorithm must include:

1. **Base Case:** a simple situation that can be solved directly.
2. **Recursive Case:** a rule for breaking the problem into smaller subproblems.
3. **Convergence:** assurance that each recursive call moves toward the base case.

## 2.2 Example: Summation

Let us define a function that computes the sum of the first $n$ natural numbers:

$$S(n) = \begin{cases} 0, & n = 0 \\ n + S(n-1), & n > 0 \end{cases}$$

**Recursive Algorithm (Python style):**

```
def sum_to_n(n):
    if n == 0:
        return 0
```

```
    else:
        return n + sum_to_n(n - 1)
```

Each recursive call reduces $n$ by one until it reaches the base case $n = 0$. This process mirrors the inductive definition of summation.

## 2.3 Example: Factorial Function

We can express $n!$ recursively as:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n > 0. \end{cases}$$

**Recursive Algorithm:**

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Each call creates a new stack frame that holds the current value of $n$ until the base case is reached. When the recursion unwinds, the frames multiply their stored $n$ values in reverse order.

## 2.4 Example: Fibonacci Sequence

The Fibonacci sequence is defined by

$$F(n) = \begin{cases} 0, & n = 0, \\ 1, & n = 1, \\ F(n-1) + F(n-2), & n > 1. \end{cases}$$

**Recursive Algorithm:**

```
def fib(n):
    if n <= 1:
        return n
```

```
else:
    return fib(n - 1) + fib(n - 2)
```

This version is beautifully simple but computationally explosive—each call spawns two more. For large $n$, this algorithm grows exponentially in time. We will later optimize it using **memoization** and **iteration**.

## 2.5 Tracing a Recursive Call

Let us trace the computation of `sum_to_n(4)`:

$$S(4) = 4 + S(3) = 4 + (3 + S(2)) = 4 + 3 + 2 + 1 + S(0) = 10$$

During recursion, a *call stack* forms:

$$S(4) \to S(3) \to S(2) \to S(1) \to S(0)$$

and then unwinds back in the reverse order as results return upward.

## 2.6 Analyzing Recursive Cost

The cost of recursion depends on the number of calls and the work per call. For the factorial function, the recurrence relation is:

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

which resolves to $O(n)$. For Fibonacci, the recurrence

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

yields $O(2^n)$ — a dramatic difference.

## 2.7 Challenge Problems

1. Write a recursive algorithm for computing the sum of digits of an integer.
2. Modify the Fibonacci algorithm to use memoization.
3. Trace the recursion tree of `fib(5)` and count the total number of calls.

4. Prove by induction that your optimized Fibonacci algorithm runs in $O(n)$ time.

   "A recursive algorithm is not just a loop—it is a story told backward and forward at the same time."

# Let's Talk About It!

**1. Why does every recursive function need a base case?**
Because without one, it's like telling a story that never ends—our program keeps calling itself until it falls off the call-stack cliff. In `factorial`, the base case $n = 0$ gives recursion a place to stop and return.

**2. How do we know the recursive step gets closer to the base?**
Each call should simplify the problem. When `sum_to_n(n)` calls `sum_to_n(n-1)`, we shrink the task by one. That shrinking is our mathematical version of gravity pulling everything back to the ground truth.

**3. Why is Fibonacci so expensive?**
Because it does the same work again and again—like explaining recursion to five people individually instead of once to the group. Memoization (saving past results) fixes that.

**4. Big Picture:** Recursion is how we think about problems that break down into self-similar parts. In life, it's the same idea—learn one step well, then repeat it with a twist. Computers just make that rhythm literal.

> Base case: start small. Recursive case: keep improving. Return: a better version of yourself.

# Chapter 3

# The Fibonacci Sequence — Nature's Algorithm

## 3.1 The Story of Fibonacci

Long ago, in a quiet Italian village, a mathematician named Leonardo of Pisa—nicknamed **Fibonacci**—noticed patterns in nature that seemed to whisper numbers into every petal, every pinecone, every spiral galaxy.

The Fibonacci sequence begins simply:

$$F(0) = 0, \quad F(1) = 1,$$

and each term afterwards is born from the sum of its two predecessors:

$$F(n) = F(n-1) + F(n-2).$$

So, the sequence grows:

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34, \ldots$$

Each generation in this sequence depends upon the previous two—just as branches split from branches, and ideas sprout from ideas.

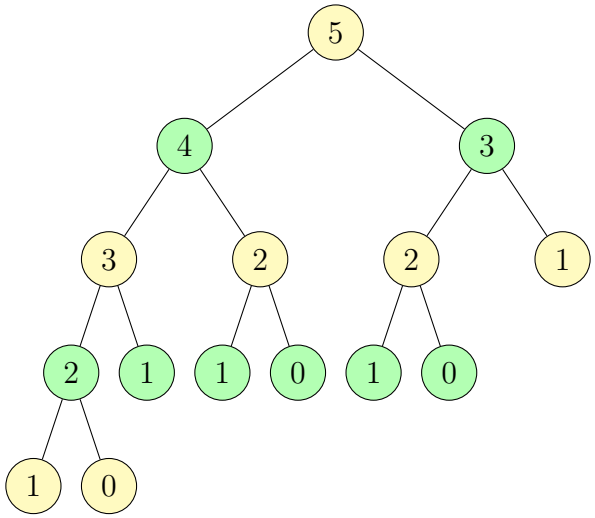
*Recursion is nature's favorite loop.*

## 3.2 Seeing the Pattern

To visualize how recursion unfolds, consider the computation of $F(5)$:

$$\texttt{fib(5)} \to \texttt{fib(4)} + \texttt{fib(3)}$$

Each call branches into two smaller ones, creating a beautiful tree of self-reference:




Notice how each recursive call duplicates work from previous branches. This inefficiency is the price of elegance—but later, we'll learn to make it faster.

## 3.3 A Gentle Python Introduction

```
def fib_recursive(n):
    """Return the nth Fibonacci number using recursion."""
    if n <= 1:
        return n
    return fib_recursive(n - 1) + fib_recursive(n - 2)

for i in range(10):
    print(f"F({i}) = {fib_recursive(i)}")
```

Let's unpack what happens. When you call `fib_recursive(5)`, the function calls itself twice, then each of those calls calls itself twice more, until it reaches the **base cases** where $n = 0$ or $n = 1$.

Each branch blooms, divides, and resolves—just like the petals of a sunflower spiraling toward the sun.

## 3.4 Recursive Beauty

Recursion is powerful because it mirrors the way we define patterns in nature: each step depends upon smaller, simpler versions of itself.

> *To understand recursion, one must first understand recursion.*

This idea is both a mathematical truth and a philosophical koan. In the next chapters, we'll measure just how costly this beauty is, and how algorithmic growth behaves as $n$ increases.

**Coming soon:**

*Chapter 4 — Measuring the Cost of Recursion*

*and*

*Chapter 5 — Understanding Big-O Through Fibonacci.*

# Let's Talk About It!

**1. What makes Fibonacci so famous?**
It's simple enough for a child to grasp but deep enough to hide in pinecones, sunflowers, and galaxies. In programming, it's our first dance with recursion—the moment we realize functions can dream about themselves.

**2. Why does `fib_recursive(5)` grow like a tree?**
Because each call splits in two, like branches reaching for light. That's the visual map of exponential growth. When we later prune those branches with memoization, the forest becomes efficient.

**3. What lesson hides inside this inefficiency?**
Beauty and cost are often linked. Nature spends energy to grow spirals; algorithms spend time to find order. Recognizing that trade-off is part of computational maturity.

**4. How can we generalize this?**
The Fibonacci idea—each term built from the previous two—shows up everywhere: population models, musical rhythm, financial compounding, even in the way you build understanding. Each new concept stands on the two before it.

**Big Picture Reflection:**
Recursion teaches patience. Each function call trusts the smaller ones beneath it to do their job. Learning is the same: stack your understanding, return results upward, and celebrate when the pattern emerges.

> Small truths multiplied by courage create big understanding.

# Chapter 4

# Measuring the Cost of Recursion

## 4.1 Purpose

In this chapter, we investigate how recursive and iterative algorithms differ in their computational cost. Using the Fibonacci sequence as our case study, we'll see how recursion can sometimes explode in complexity.

## 4.2 The Experiment

We designed a Python experiment that measures:

- The number of function calls
- The number of additions
- The number of variable assignments
- The time required to compute $F_n$

### 4.2.1 The Tracker Class

Listing 4.1: DataTracker and Fibonacci comparison

```
#!/usr/bin/env python3
import csv
import time
import matplotlib.pyplot as plt
```

```
class DataTracker:
    def __init__(self):
        self.calls = 0
        self.adds = 0
        self.assigns = 0
        self.times = {}

    def track(self, n, value, t0, t1):
        self.times[n] = {
            "value": value,
            "time": t1 - t0,
            "calls": self.calls,
            "adds": self.adds,
            "assigns": self.assigns
        }

    def reset(self):
        self.calls = 0
        self.adds = 0
        self.assigns = 0

def fib_recursive(n, tracker):
    tracker.calls += 1
    if n <= 1:
        tracker.assigns += 1
        return n
    tracker.adds += 1
    return fib_recursive(n-1, tracker) + fib_recursive(n-2, tracker)

def fib_iterative(n, tracker):
    tracker.calls += 1
    a, b = 0, 1
    tracker.assigns += 2
    for _ in range(2, n+1):
        tracker.adds += 1
        a, b = b, a + b
        tracker.assigns += 2
    return b if n else a
```

```
def measure_fibonacci(max_n=30):
    tracker = DataTracker()
    results = []

    # Recursive measurement
    for n in range(0, max_n+1):
        tracker.reset()
        t0 = time.time()
        val = fib_recursive(n, tracker)
        t1 = time.time()
        tracker.track(n, val, t0, t1)
        results.append(["recursive", n, val, tracker.times[n]["time"],
            tracker.calls, tracker.adds, tracker.assigns])

    # Iterative measurement
    for n in range(0, max_n+1):
        tracker.reset()
        t0 = time.time()
        val = fib_iterative(n, tracker)
        t1 = time.time()
        tracker.track(n, val, t0, t1)
        results.append(["iterative", n, val, tracker.times[n]["time"],
            tracker.calls, tracker.adds, tracker.assigns])

    # Write results
    with open("fib_results.csv", "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["method", "n", "value", "time_sec", "calls", "adds",
            "assigns"])
        writer.writerows(results)

    return results

def plot_results(csv_file="fib_results.csv"):
    import pandas as pd
    df = pd.read_csv(csv_file)
    fig, axs = plt.subplots(3, 1, figsize=(8, 10))
```

```
    for method, group in df.groupby("method"):
        axs[0].plot(group["n"], group["time_sec"], label=method)
        axs[1].plot(group["n"], group["calls"], label=method)
        axs[2].plot(group["n"], group["adds"], label=method)

    axs[0].set_ylabel("Time␣(s)")
    axs[1].set_ylabel("Function␣Calls")
    axs[2].set_ylabel("Additions")
    axs[2].set_xlabel("n")
    for ax in axs:
        ax.legend()
        ax.grid(True)
    plt.tight_layout()
    plt.savefig("fib_results_plot.pdf")

if __name__ == "__main__":
    measure_fibonacci(30)
    plot_results()
    print("[OK]␣Fibonacci␣analysis␣complete.␣Results␣saved␣to␣fib_results.csv␣and␣
        fib_results_plot.pdf")
```

### 4.2.2 Generated Data

After running the Python code, the script produces two files:

- `fib_results.csv` — tabular data of timing and operation counts

- `fib_results_plot.pdf` — visualization of recursive vs iterative performance

## 4.3 Results

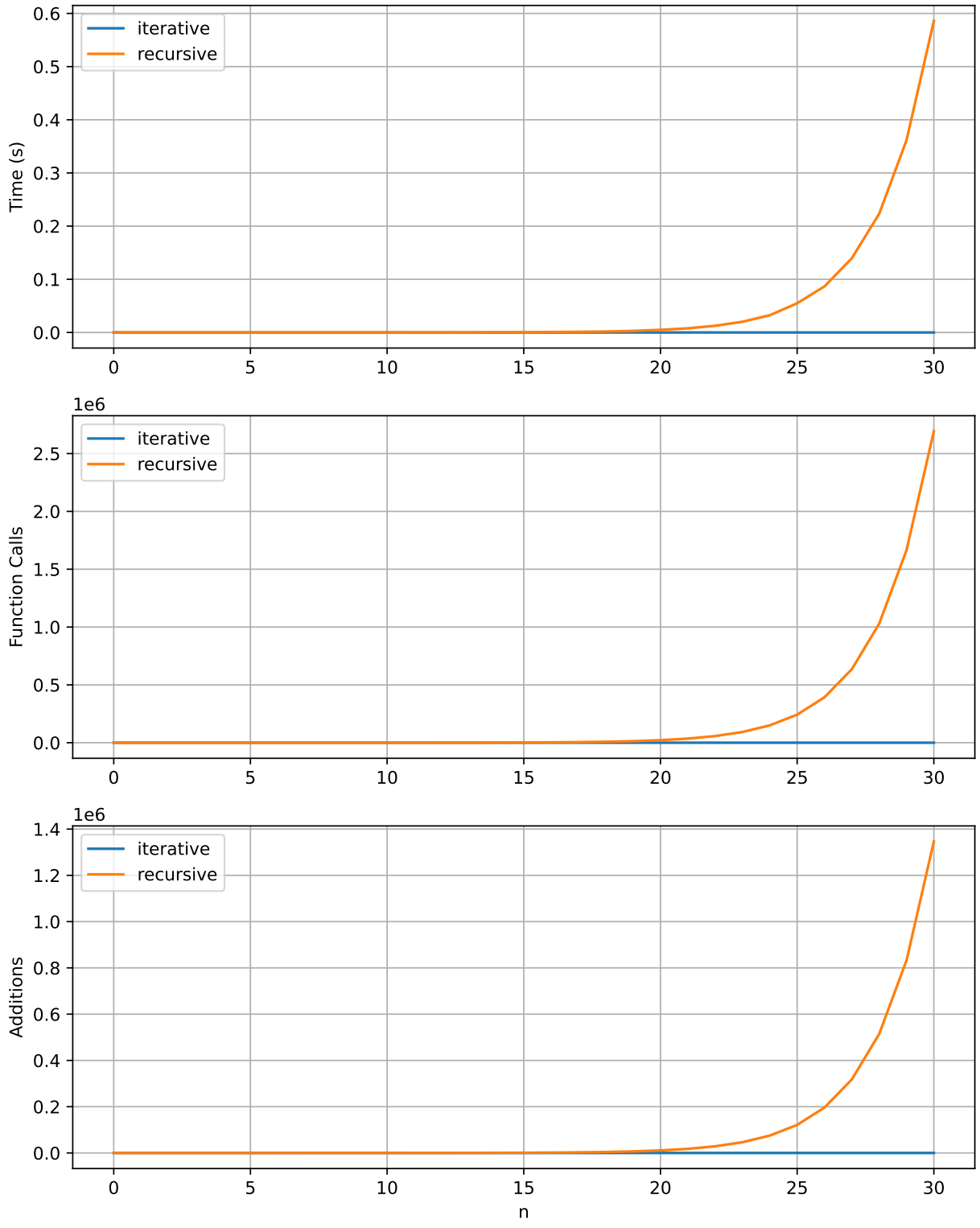



Figure 4.1: Recursive vs. Iterative Fibonacci performance

## 4.4 Reflection

Discuss:

- Why does the recursive version slow down so dramatically?
- How might memoization or dynamic programming fix this?
- What do you notice about the patterns of function calls?

## 4.5 Student Task

Run the experiment on your own system. Then modify `fib_recursive` to include memoization and compare your results.

# Chapter 5

# Code Appendix – Fibonacci Source Files

## 5.1 Overview

This appendix provides a complete snapshot of the Python source code used in this workbook. Each time you rebuild the book, the latest version of the code is automatically included — ensuring this text remains a living record of our experiments.

## 5.2 Fibonacci Analysis Script

```
#!/usr/bin/env python3
import csv
import time
import matplotlib.pyplot as plt

class DataTracker:
    def __init__(self):
        self.calls = 0
        self.adds = 0
        self.assigns = 0
        self.times = {}

    def track(self, n, value, t0, t1):
        self.times[n] = {
            "value": value,
            "time": t1 - t0,
            "calls": self.calls,
            "adds": self.adds,
            "assigns": self.assigns
```

```
        }

    def reset(self):
        self.calls = 0
        self.adds = 0
        self.assigns = 0

def fib_recursive(n, tracker):
    tracker.calls += 1
    if n <= 1:
        tracker.assigns += 1
        return n
    tracker.adds += 1
    return fib_recursive(n-1, tracker) + fib_recursive(n-2, tracker)

def fib_iterative(n, tracker):
    tracker.calls += 1
    a, b = 0, 1
    tracker.assigns += 2
    for _ in range(2, n+1):
        tracker.adds += 1
        a, b = b, a + b
        tracker.assigns += 2
    return b if n else a

def measure_fibonacci(max_n=30):
    tracker = DataTracker()
    results = []

    # Recursive measurement
    for n in range(0, max_n+1):
        tracker.reset()
        t0 = time.time()
        val = fib_recursive(n, tracker)
        t1 = time.time()
        tracker.track(n, val, t0, t1)
        results.append(["recursive", n, val, tracker.times[n]["time"], tracker.calls,
            tracker.adds, tracker.assigns])

    # Iterative measurement
    for n in range(0, max_n+1):
        tracker.reset()
        t0 = time.time()
```

```
        val = fib_iterative(n, tracker)
        t1 = time.time()
        tracker.track(n, val, t0, t1)
        results.append(["iterative", n, val, tracker.times[n]["time"], tracker.calls,
            tracker.adds, tracker.assigns])

    # Write results
    with open("fib_results.csv", "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["method", "n", "value", "time_sec", "calls", "adds", "assigns"])
        writer.writerows(results)

    return results

def plot_results(csv_file="fib_results.csv"):
    import pandas as pd
    df = pd.read_csv(csv_file)
    fig, axs = plt.subplots(3, 1, figsize=(8, 10))

    for method, group in df.groupby("method"):
        axs[0].plot(group["n"], group["time_sec"], label=method)
        axs[1].plot(group["n"], group["calls"], label=method)
        axs[2].plot(group["n"], group["adds"], label=method)

    axs[0].set_ylabel("Time (s)")
    axs[1].set_ylabel("Function Calls")
    axs[2].set_ylabel("Additions")
    axs[2].set_xlabel("n")
    for ax in axs:
        ax.legend()
        ax.grid(True)
    plt.tight_layout()
    plt.savefig("fib_results_plot.pdf")

if __name__ == "__main__":
    measure_fibonacci(30)
    plot_results()
    print("[OK] Fibonacci analysis complete. Results saved to fib_results.csv and
        fib_results_plot.pdf")
```

Listing 5.1: `fib_analysis.py` — A fully instrumented Fibonacci analysis with timing and operation counts.

# Chapter 6

# Understanding Big-O Through Fibonacci

## 6.1 Why Fibonacci is the Perfect Big-O Lens

Fibonacci exposes two very different growth stories:

- The naive recursive version has *exponential* time, roughly $O(2^n)$.
- With memoization (or bottom-up DP), the time drops to *linear*, $O(n)$, with $O(n)$ space.

This makes it a great stage to see how algorithm design transforms cost. *(See Chapters 2–4 for setup and experiments.)* :contentReference[oaicite:0]index=0

## 6.2 Deriving the Cost

### 6.2.1 Naive recursion: $O(2^n)$ time

Define the cost $T(n)$ of

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n > 1 \end{cases}$$

Each call to $F(n)$ triggers two subcalls (except at base cases). A standard bound is

$$T(n) = T(n-1) + T(n-2) + O(1) \quad \Rightarrow \quad T(n) = \Theta(\varphi^n),$$

where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618$ and $\varphi^n = \Theta(2^n)$. Intuition: the recursion tree doubles "enough" times that the total work explodes exponentially.

### 6.2.2 Memoization: $O(n)$ time, $O(n)$ space

Top-down memoization stores previously computed $F(k)$ so each $k \in \{0, \ldots, n\}$ is computed once:

$$T(n) = T(n-1) + O(1), \quad \text{amortized over } n \text{ distinct subproblems} \Rightarrow T(n) = O(n).$$

Space is $O(n)$ for the table/stack (top-down) or just $O(n)$ array (bottom-up). The iterative two-variable version achieves $O(1)$ extra space.

## 6.3 Teaching Edge: Memoization as Memory of the Mind

> Memoization is the art of remembering your past. When a recursive function revisits a subproblem it has already solved, it wastes effort—like a forgetful mathematician re-deriving $2+2$ each time. By storing results in a dictionary (Python's built-in memory palace), each distinct input is computed exactly once:
>
> $$T(n) = T(n-1) + O(1) \quad \Rightarrow \quad T(n) = O(n)$$
>
> The recursion tree collapses from an exponential bush into a single, graceful vine.

This small conceptual leap—*remember instead of repeat*—is a cornerstone of efficient thinking, in code and in life.

## 6.4 Visual: Exponential vs Linear

Figure 6.1 plots $2^n$ against $n$. The lines start friendly, then $2^n$ rockets away like a bottle rocket with a PhD.

## 6.5 Code: Three Flavors of Fibonacci

We include three implementations and an experiment harness that writes CSV and plots.

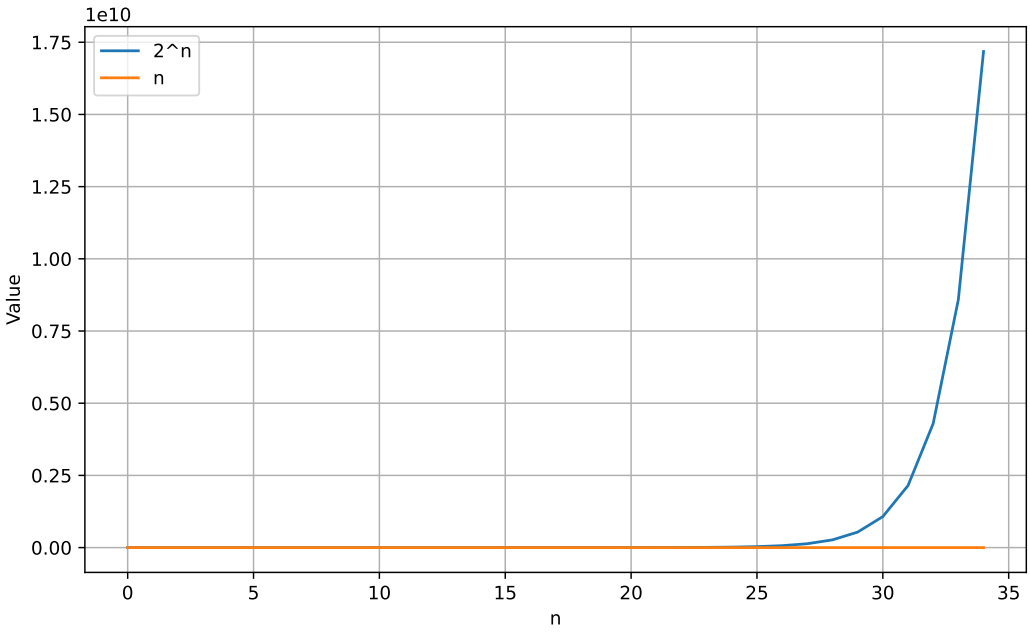



Figure 6.1: $2^n$ vs $n$ on the same axes (log-scale optional).

### 6.5.1 Tracker and Implementations

```python
#!/usr/bin/env python3
# ASCII-only code; safe for LaTeX listings.

class DataTracker:
    def __init__(self):
        self.calls = 0
        self.adds = 0
        self.assigns = 0

    def reset(self):
        self.calls = 0
        self.adds = 0
        self.assigns = 0

def fib_recursive(n, trk: DataTracker):
    trk.calls += 1
    if n <= 1:
        trk.assigns += 1
        return n
    trk.adds += 1
    return fib_recursive(n - 1, trk) + fib_recursive(n - 2, trk)
```

```python

def fib_memo(n, trk: DataTracker, memo=None):
    # Top-down memoization: O(n) time, O(n) space
    if memo is None:
        memo = {}
    trk.calls += 1
    if n in memo:
        return memo[n]
    if n <= 1:
        trk.assigns += 1
        memo[n] = n
        return n
    trk.adds += 1
    val = fib_memo(n - 1, trk, memo) + fib_memo(n - 2, trk, memo)
    memo[n] = val
    return val

def fib_iter(n, trk: DataTracker):
    trk.calls += 1
    if n <= 1:
        trk.assigns += 1
        return n
    a, b = 0, 1
    trk.assigns += 2
    for _ in range(2, n + 1):
        trk.adds += 1
        a, b = b, a + b
        trk.assigns += 2
    return b
```

Listing 6.1: Tracker and Fibonacci variants

### 6.5.2 Experiment: timing, counts, and plots

```python
#!/usr/bin/env python3
# ASCII-only benchmark; produces CSV and a plot PDF.

import csv
import time
import matplotlib.pyplot as plt
from fib_variants import DataTracker, fib_recursive, fib_memo, fib_iter

def run_suite(max_n=35):
```

```
    rows = []
    trk = DataTracker()

    # Recursive
    for n in range(max_n + 1):
        trk.reset()
        t0 = time.time()
        val = fib_recursive(n, trk)
        t1 = time.time()
        rows.append(["recursive", n, val, t1 - t0, trk.calls, trk.adds, trk.assigns])

    # Memoized
    for n in range(max_n + 1):
        trk.reset()
        t0 = time.time()
        val = fib_memo(n, trk, memo={})
        t1 = time.time()
        rows.append(["memoized", n, val, t1 - t0, trk.calls, trk.adds, trk.assigns])

    # Iterative
    for n in range(max_n + 1):
        trk.reset()
        t0 = time.time()
        val = fib_iter(n, trk)
        t1 = time.time()
        rows.append(["iterative", n, val, t1 - t0, trk.calls, trk.adds, trk.assigns])

    with open("fib_results_ext.csv", "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(["method","n","value","time_sec","calls","adds","assigns"])
        w.writerows(rows)

    return rows

def plot(csv_path="fib_results_ext.csv"):
    import pandas as pd
    df = pd.read_csv(csv_path)
    methods = ["recursive", "memoized", "iterative"]

    # Time plot
    fig = plt.figure(figsize=(8, 9))
    ax1 = plt.subplot(3,1,1)
    ax2 = plt.subplot(3,1,2)
```

```
    ax3 = plt.subplot(3,1,3)

    for m in methods:
        g = df[df["method"] == m]
        ax1.plot(g["n"], g["time_sec"], label=m)
        ax2.plot(g["n"], g["calls"], label=m)
        ax3.plot(g["n"], g["adds"], label=m)

    ax1.set_ylabel("Time (s)")
    ax2.set_ylabel("Function Calls")
    ax3.set_ylabel("Additions")
    ax3.set_xlabel("n")
    for ax in (ax1, ax2, ax3):
        ax.grid(True)
        ax.legend()

    plt.tight_layout()
    plt.savefig("chapters/fib_bench_plot.pdf")

if __name__ == "__main__":
    run_suite(35)
    plot()
    print("[OK] Wrote fib_results_ext.csv and chapters/fib_bench_plot.pdf")
```

Listing 6.2: Benchmark and plots for recursive vs memo vs iterative

## 6.6 Practical Implications

- **Naive recursion** demonstrates explosive growth; great teaching tool, terrible production tool beyond modest $n$.
- **Memoization** replaces recomputation with table lookups: a tiny idea with huge consequences.
- **Iterative DP** keeps the wins while minimizing memory; for Fibonacci, two scalars suffice.

## 6.7 Optional: Memory Probe (Laptop)

To show that Big-O is a map (not the terrain), we sample real memory/CPU while the naive version gnaws on the stack. See Listing 6.3.

```python
#!/usr/bin/env python3
# ASCII-only. Samples process memory/CPU while naive recursion runs.
# This will not be as precise as Valgrind, but it is classroom-friendly.

import os, time, threading
try:
    import psutil
except ImportError:
    psutil = None

from fib_variants import DataTracker, fib_recursive

SAMPLE_INTERVAL = 0.05 # seconds

def sampler(stop_flag, pid, out_path):
    if psutil is None:
        return
    proc = psutil.Process(pid)
    with open(out_path, "w") as f:
        f.write("t_sec,rss_bytes,cpu_percent\n")
        t0 = time.time()
        while not stop_flag["stop"]:
            t = time.time() - t0
            try:
                rss = proc.memory_info().rss
                cpu = proc.cpu_percent(interval=None)
            except psutil.Error:
                break
            f.write(f"{t:.3f},{rss},{cpu:.1f}\n")
            time.sleep(SAMPLE_INTERVAL)

def main(n=38):
    # WARNING: pick n that is slow enough to show growth, but not so large it hangs.
    trk = DataTracker()
    stop_flag = {"stop": False}
    out_csv = "fib_memtrace.csv"

    th = None
    if psutil is not None:
        th = threading.Thread(target=sampler, args=(stop_flag, os.getpid(), out_csv))
        th.daemon = True
        th.start()

```

```
    t0 = time.time()
    val = fib_recursive(n, trk)
    t1 = time.time()

    stop_flag["stop"] = True
    if th is not None:
        th.join()

    with open("fib_memprobe_summary.txt", "w") as f:
        f.write("n,value,time_sec,calls,adds,assigns\n")
        f.write(f"{n},{val},{t1-t0:.6f},{trk.calls},{trk.adds},{trk.assigns}\n")

    print("[OK] Wrote fib_memtrace.csv and fib_memprobe_summary.txt")

if __name__ == "__main__":
    main(38)
```

Listing 6.3: Lightweight memory/CPU probe while running recursion

## 6.8 Student Prompts

1. For which $n$ does naive recursion become impractical on your machine?
2. Modify memoization to bottom-up; compare peak memory with `tracemalloc`.
3. Explain why memoization changes the recursion tree's shape and total nodes visited.

# Chapter 7

# Big-O Meets the Real World

## 7.1 When the Laptop Screams

We have theorized, graphed, and philosophized. Now let us observe the beast in the silicon jungle. Using our `fib_memprobe.py` script, we recorded CPU and memory usage as $F(n)$ climbed higher and higher—until the machine begged for mercy.

### 7.1.1 Empirical Memory and CPU Usage

Figure 7.1 plots actual measurements from `fib_memtrace.csv`. Each point represents a recursive call depth sampled over time.

### 7.1.2 Big-O vs. Reality

Big-O describes asymptotic shape, not scale. It ignores constants, compiler optimizations, cache misses, and human sighs. Yet its *curve* predicts the suffering accurately:

$$O(2^n) \text{ feels like doubling pain every increment of } n.$$

Memoization reclaims sanity by converting the chaos to $O(n)$. The plot in Figure 7.2 overlays theoretical curves with empirical data—showing that while constants differ, shapes endure.

## 7.2 Beyond Big-O: When the Stack Runs Out

The naive recursive Fibonacci is limited not just by time but by stack depth and heap exhaustion. Past a certain $n$, Python throws a `RecursionError`. The cost curve is no longer

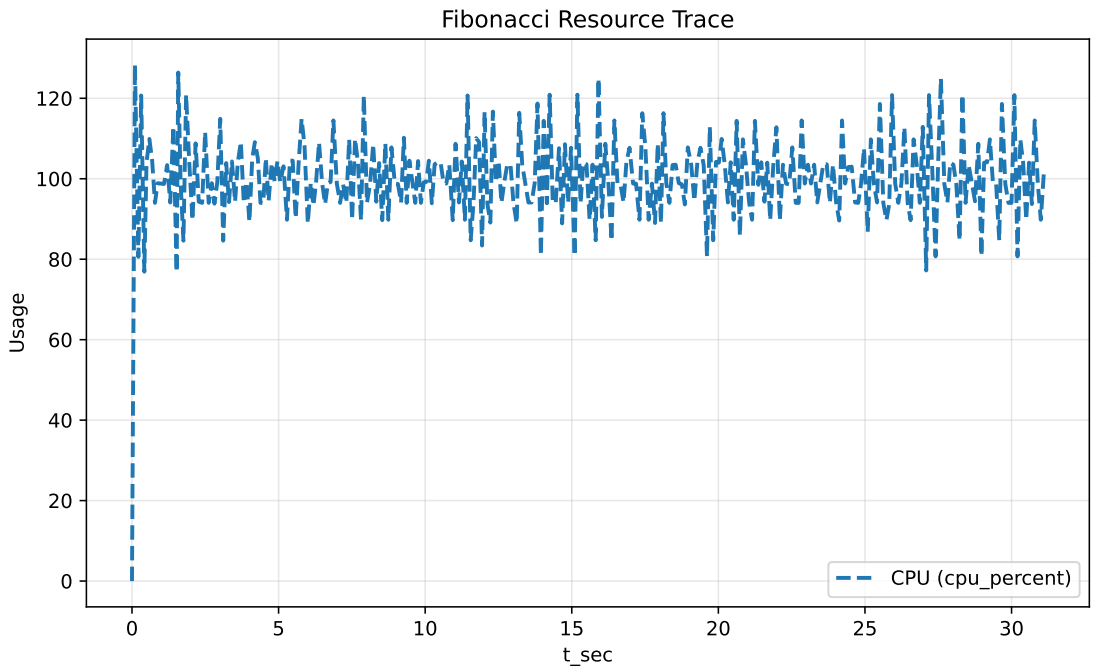



Figure 7.1: Observed CPU and memory footprint for naive recursion. Exponential calls lead to geometric memory growth.

mathematical—it is existential.

> **Observation:** Big-O assumes infinite memory and patience. Your laptop does not.

## 7.3 Reflection Prompts

1. At what $n$ did your system fail or slow dramatically?
2. Compare your CPU temperature trace to your time complexity plot.
3. What constant factors made the real data deviate from $2^n$?
4. How does memoization mimic biological memory in reducing energy waste?

## 7.4 Epilogue: The Shape of Pain

In this chapter, the curve of $2^n$ became audible—the fans whirred, the heat rose, the machine groaned. The math foretold it. And that, dear reader, is the poetry of Big-O.

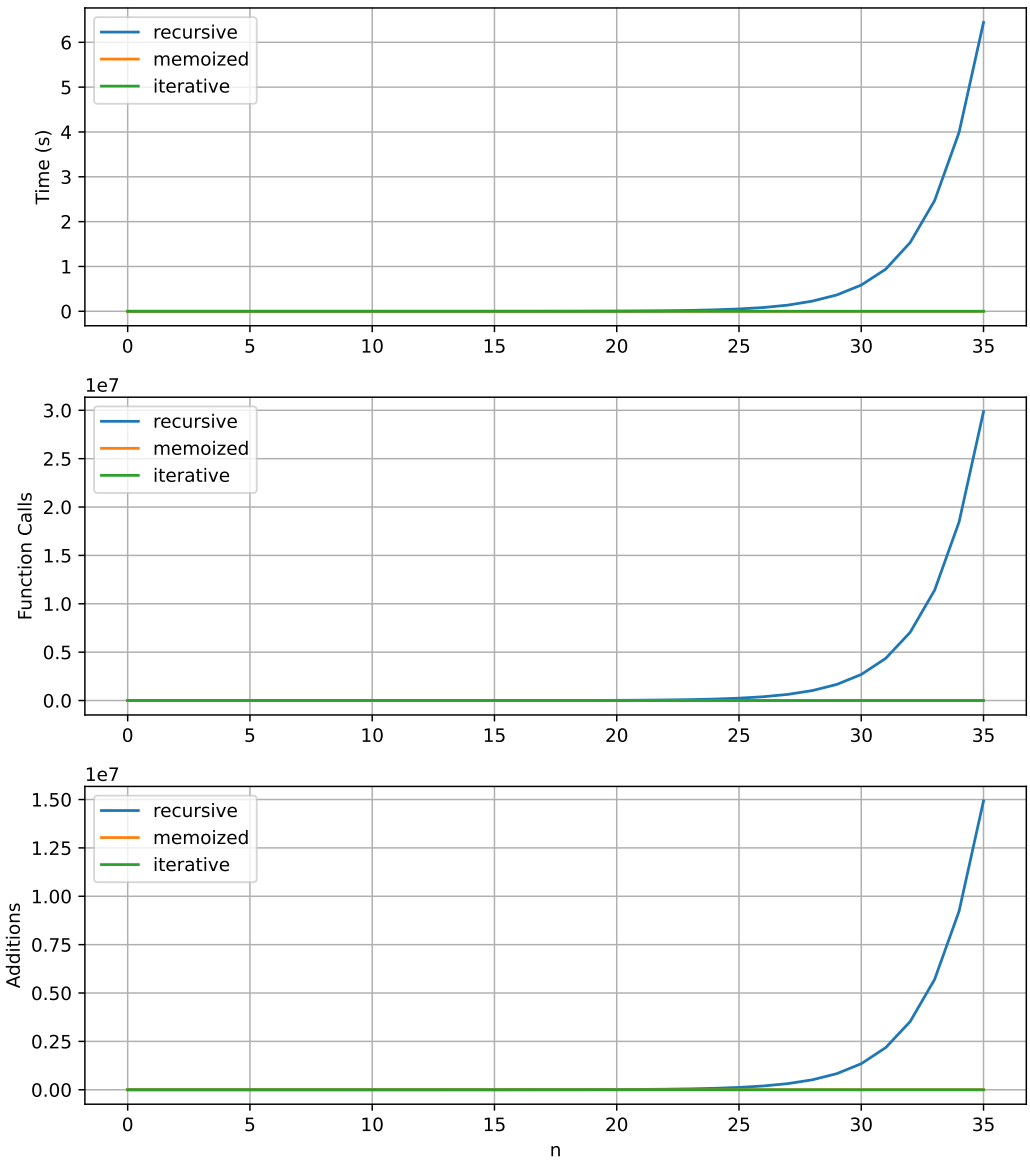



Figure 7.2: Naive recursion (red, $O(2^n)$) vs. memoized (green, $O(n)$). Reality hugs the theory.