COMSC 2043
Induction and Recursion

Student Workbook

Jeremy Evert
Southwestern Oklahoma State University

October 27, 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?

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."

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.

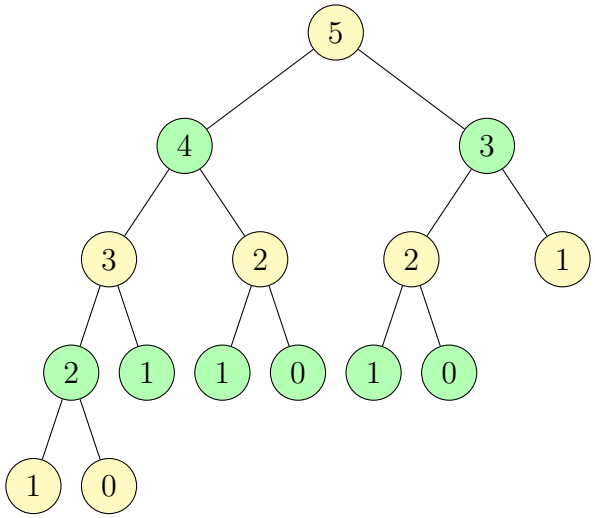
Recursion is nature's favorite loop.

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.

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"], tracke

    # 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"], tracke

    # Write results
    with open("fib_results.csv", "w", newline="") as f:
        writer = csv.writer(f)
        writer.writerow(["method", "n", "value", "time_sec", "calls", "adds",
        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("    Fibonacci analysis complete. Results saved to fib_results.csv
```

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

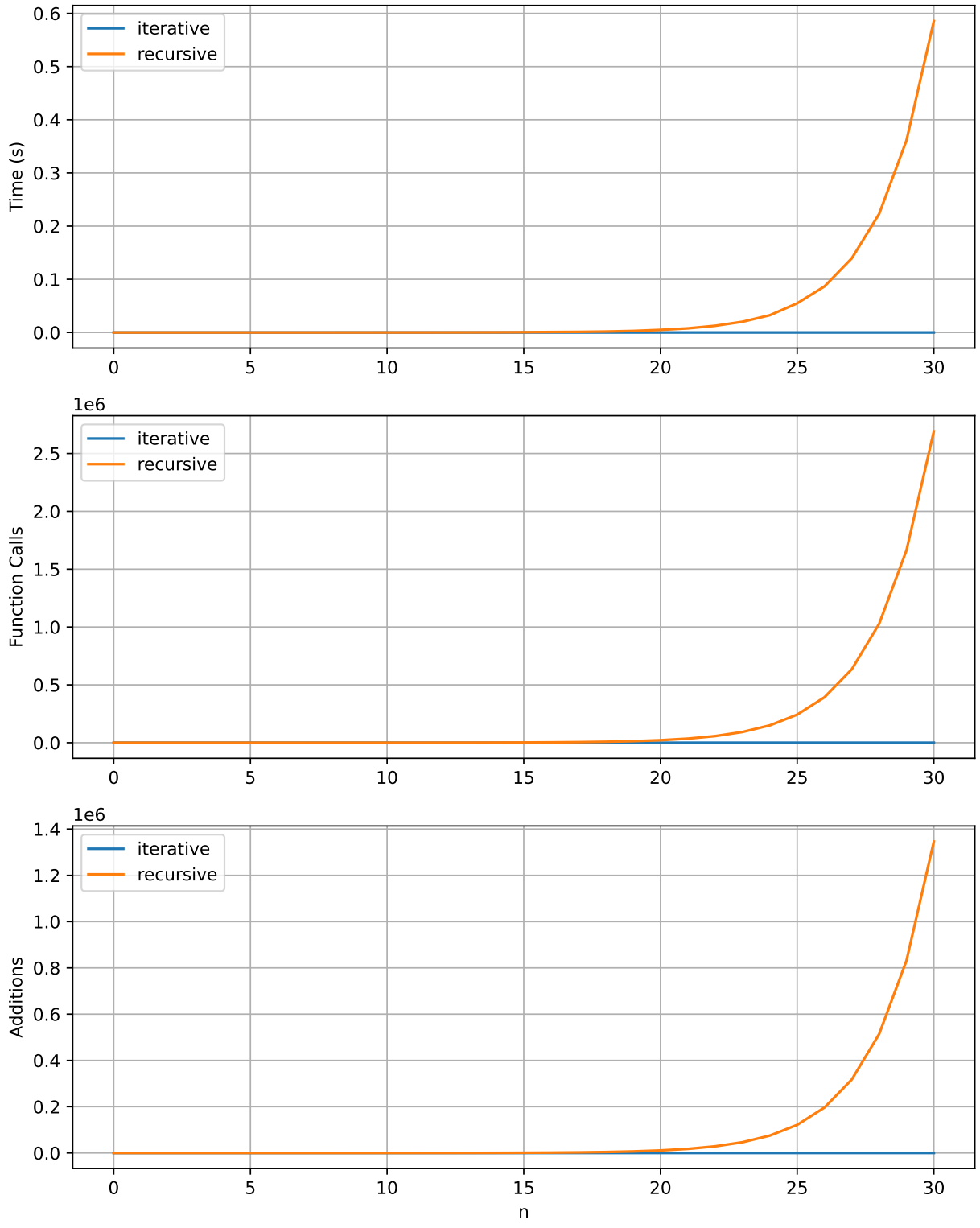


Figure 4.1: Recursive vs. Iterative Fibonacci performance

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.