COMSC 2043
Counting

Student Workbook

Jeremy Evert
Southwestern Oklahoma State University

November 4, 2025

Contents

1 Counting: Combinatorial Foundations of Computer Science **5**

2 Section 6.1 — The Basics of Counting **7**

2.1 Introduction 7

2.2 Basic Counting Principles 7

2.3 Worked Examples 7

2.4 Python Demonstrations 8

3 Section 6.1 — Extended Product Rule **11**

3.1 The Extended Product Rule 11

3.2 Worked Examples 11

3.3 Apply It Yourself 12

3.4 Python Demonstrations 17

4 Section 6.1 — Product Rule in Action **19**

4.1 Real-World Applications of the Product Rule 19

4.2 Python Demonstrations 24

Chapter 1

Counting: Combinatorial Foundations of Computer Science

How Chapter 6 Fits into the Big Picture

In the grand journey of discrete mathematics, **Chapter 6: Counting** marks a turning point. Up to this point, we have focused on the language of logic (Chapter 1), the structures that give shape to mathematical reasoning such as sets, functions, and sequences (Chapter 2), and the design of step-by-step procedures through algorithms (Chapter 3). We have also examined how numbers behave and how to prove properties about them, both through number theory and by building proofs inductively and recursively (Chapters 4 and 5).

Now, the focus shifts from *why* things are true to *how many* ways they can be true. Counting gives us the ability to measure the size of possibilities—whether it's the number of valid passwords, the number of paths through a network, or the number of ways to schedule tasks on a processor. In computer science, these aren't abstract curiosities: they are the foundation of data analysis, probability, algorithmic efficiency, and cryptographic security.

Overview of Chapter 6

This chapter introduces the fundamental principles of **combinatorics**—the art and science of counting. Each section develops a new tool in the combinatorial toolkit:

6.1 The Basics of Counting introduces the rule of sum and the rule of product, the two fundamental ideas that allow us to count complex structures by breaking them into simpler cases.

6.2 The Pigeonhole Principle formalizes an intuitive idea: if you have more objects than containers, at least one container must hold more than one object. This principle turns up everywhere—from hashing algorithms to error detection and compression schemes.

6.3 Permutations and Combinations explores ordered and unordered selections—how to count arrangements of items with or without repetition. These are the cornerstones of combinatorial reasoning and underpin probability theory and algorithmic enumeration.

6.4 Binomial Coefficients and Identities connects counting to algebra through Pascal's triangle and the binomial theorem, showing how algebraic expressions encode combinatorial ideas.

6.5 Generalized Permutations and Combinations extends our tools to more complex scenarios: multisets, repeated elements, and the counting of indistinguishable objects.

6.6 Generating Permutations and Combinations turns theory into practice. Here we explore how to systematically produce combinations and permutations using algorithmic logic—a perfect bridge between mathematics and computer science.

Each topic builds naturally on the last, leading toward a robust framework for reasoning about *how many* ways something can happen—a question that lies at the heart of computational problem solving.

Counting in the Broader Context of Computer Science

Counting is more than arithmetic—it is **computation in miniature**. Every time a program loops through possibilities, a scheduler distributes resources, or a cryptographer designs a key system, counting quietly determines what's possible and what's practical. The efficiency of algorithms often depends on how many steps or arrangements exist; probability models rely on counting possible outcomes; and even machine learning models depend on combinatorial structures when exploring feature combinations or optimization paths.

In short, the study of counting provides a bridge between abstract reasoning and algorithmic design. It transforms intuition into strategy, helping computer scientists predict complexity, measure growth, and understand limits.

By mastering this chapter, you are learning to see the invisible arithmetic beneath every decision tree, database index, and combinational circuit—a skill as essential to the digital age as logic and code themselves.

Chapter 2

Section 6.1 — The Basics of Counting

2.1 Introduction

Suppose that a password on a computer system consists of six, seven, or eight characters. Each of these characters must be a digit or a letter of the alphabet, and each password must contain at least one digit. How many such passwords are there?

Questions like this form the beating heart of discrete mathematics: *How many ways can something happen?* Counting allows us to measure complexity, probability, and possibility. In computer science, every algorithm that loops, branches, or searches is secretly counting something.

2.2 Basic Counting Principles

Two fundamental rules guide our reasoning:

The Sum Rule If a task can be done in n_1 ways *or* n_2 ways (but not both), then it can be done in $n_1 + n_2$ ways.

The Product Rule If a task can be broken into two independent subtasks—first done in n_1 ways, then in n_2 ways—then the whole procedure can be done in $n_1 \times n_2$ ways.

2.3 Worked Examples

Example 2.1 (Assigning Offices). *A new company with two employees, Sanchez and Patel, rents a floor with 12 offices. How many ways are there to assign different offices to these two employees?*

Solution. *12 choices for Sanchez, then 11 remaining for Patel. By the product rule:* $12 \times 11 = 132$ *possibilities.*

Example 2.2 (Labeling Auditorium Chairs). *Each seat is labeled with one uppercase English letter followed by a positive integer not exceeding 100. How many unique chair labels exist?*

Solution. 26 *possible letters* $\times$ 100 *integers* $= 2600$ *labels.*

Example 2.3 (Counting Ports in a Data Center). *There are 32 computers, each with 24 ports. How many total ports exist?*

Solution. $32 \times 24 = 768$ *ports.*

Example 2.4 (Challenging A: License Plate Combinations). *A region issues license plates with 3 uppercase letters followed by 3 digits (e.g., ABC123). How many distinct plates are possible if repetition is allowed?*

Solution. $26^3 \times 10^3 = 175{,}760{,}000$ *possible plates.*

If letters and digits cannot repeat, the count becomes $26 \times 25 \times 24 \times 10 \times 9 \times 8 = 112{,}320{,}000$.

Example 2.5 (Challenging B: Passwords with a Digit Requirement). *A password must be 6, 7, or 8 characters long, each either a letter or digit, and must contain at least one digit. How many possible passwords exist?*

Solution. *Let A be all possible strings of the given length (letters + digits), and B be those with only letters.*

Then valid passwords $= A - B$.

$$\begin{aligned}
|A_6| &= 36^6, & |B_6| &= 26^6 \\
|A_7| &= 36^7, & |B_7| &= 26^7 \\
|A_8| &= 36^8, & |B_8| &= 26^8 \\
\textit{Total} &= (36^6 - 26^6) + (36^7 - 26^7) + (36^8 - 26^8)
\end{aligned}$$

2.4 Python Demonstrations

Below is Python code that mirrors the reasoning in these examples. Each section prints both the symbolic reasoning and the computed value. Students are encouraged to modify the numbers and observe how the results change.

Listing 2.1: Demonstrating the Product and Sum Rules in Python

```
"""
```

```
COMSC␣2043␣-␣Counting␣Demonstrations
Author:␣Jeremy␣Evert
This␣script␣demonstrates␣the␣basic␣principles␣of␣counting␣using␣Python.
"""

from math import comb, perm

def example_offices():
    n1, n2 = 12, 11
    total = n1 * n2
    print(f"Example␣1:␣Assigning␣offices\n12␣*␣11␣=␣{total}\n")

def example_chairs():
    total = 26 * 100
    print(f"Example␣2:␣Chair␣labeling\n26␣*␣100␣=␣{total}\n")

def example_ports():
    total = 32 * 24
    print(f"Example␣3:␣Data␣center␣ports\n32␣*␣24␣=␣{total}\n")

def example_license_plates():
    allow_repeat = 26**3 * 10**3
    no_repeat = (26*25*24) * (10*9*8)
    print("Example␣4:␣License␣plates")
    print(f"␣␣With␣repetition:␣{allow_repeat:,}")
    print(f"␣␣Without␣repetition:␣{no_repeat:,}\n")

def example_passwords():
    # Helper function: A^n - B^n
    def count_valid(length):
        return 36**length - 26**length

    total = sum(count_valid(n) for n in (6, 7, 8))
    print("Example␣5:␣Passwords␣with␣at␣least␣one␣digit")
    print(f"␣␣Total␣valid␣passwords:␣{total:,}\n")

def main():
    print("===␣Counting␣Demonstrations␣===\n")
    example_offices()
    example_chairs()
    example_ports()
    example_license_plates()
    example_passwords()
    print("Done!")

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

Discussion and Reflection

Each of these problems could be solved by intuition, but writing code forces precision. Python doesn't "believe" in the product rule—it performs it. This helps students see that the abstract logic of counting translates directly into computation.

Encourage students to:

- Change numbers and verify the rules still hold.
- Add print statements to trace intermediate steps.
- Reflect on where independence between tasks exists—and where it doesn't.

Next Steps

In the next section we will apply these rules to cases where choices overlap or restrict one another— leading to the powerful and deceptively simple **Pigeonhole Principle**.

Chapter 3

Section 6.1 — Extended Product Rule

3.1 The Extended Product Rule

The basic product rule can be generalized to any number of sequential tasks. Suppose a procedure consists of tasks $T_1, T_2, \ldots, T_m$, where each task T_i can be completed in n_i ways, independent of how the earlier tasks were done. Then the entire procedure can be performed in

$$n_1 \times n_2 \times \cdots \times n_m.$$

This can be proven formally by mathematical induction on the number of tasks—an idea that connects back beautifully to Chapter 5 on induction and recursion.

3.2 Worked Examples

Example 3.1 (Bit Strings of Length Seven). *How many different bit strings of length seven are there?*

Solution. *Each of the seven bits can be chosen in two ways—either 0 or 1. By the product rule, the total number of bit strings is:*

$$2^7 = 128.$$

◀

Example 3.2 (License Plates). *How many different license plates can be made if each contains three uppercase English letters followed by three digits?*

Solution. *There are* 26 *choices for each of the three letters and* 10 *choices for each of the three digits:*

$$26^3 \times 10^3 = 17{,}576{,}000.$$

So there are 17,576,000 *possible license plates.* ◀

Example 3.3 (Counting Functions)**.** *How many functions exist from a set with m elements to a set with n elements?*

Solution. *Each element of the domain can be mapped to any of the n elements in the codomain. Thus, by the product rule:*

$$n^m \textit{ functions.}$$

For instance, there are $5^3 = 125$ functions from a three-element set to a five-element set. ◀

Example 3.4 (Counting One-to-One Functions)**.** *How many one-to-one (injective) functions exist from a set with m elements to a set with n elements?*

Solution. *If $m > n$, none exist. When $m \leq n$, the first element can map to any of n values, the next to $(n-1)$ remaining values, and so on, giving:*

$$n(n-1)(n-2)\cdots(n-m+1).$$

For example, from a three-element set to a five-element set:

$$5 \times 4 \times 3 = 60.$$

◀

3.3 Apply It Yourself

Warm-Up Problem

How many different bit strings of length eight are there?

Solution: Each bit can be either 0 or 1, so there are $2^8 = 256$ possible bit strings.

Practice Problem 1 — Easier

How many license plates can be made if each plate has two uppercase English letters followed by two digits?

Solution: Each of the two letters has 26 possibilities, and each digit has 10:

$$26^2 \times 10^2 = 67{,}600.$$

Thus, there are 67,600 unique plates.

Practice Problem 2 — Moderate

How many one-to-one functions are there from a set with four elements to a set with six elements?

Solution:

$$6 \times 5 \times 4 \times 3 = 360.$$

There are 360 one-to-one functions.

Practice Problem 3 — Challenging

How many functions exist from a set with $m = 10$ elements to a set with $n = 3$ elements? How many of these are one-to-one?

Solution:

- Total functions: $3^{10} = 59{,}049$.
- One-to-one functions: none, since $m > n$.

Practice Problem 4 — Stretch Challenge

A company issues employee IDs consisting of: two uppercase letters, followed by one of five department codes (A–E), followed by a 3-digit number (000–999). How many possible IDs can be created?

Solution:

$$26^2 \times 5 \times 1000 = 3{,}380{,}000.$$

So there are 3,380,000 unique ID numbers possible.

3.4 Python Demonstrations

Below is Python code that illustrates these calculations. It emphasizes how the product rule scales as more tasks are added.

Listing 3.1: Extended Product Rule Examples

```python
"""
COMSC 2043 - Extended Product Rule Demonstrations
Author: Jeremy Evert
"""

from math import prod

def bit_strings(n):
    return 2 ** n

def license_plates(letters, digits):
    return 26 ** letters * 10 ** digits

def functions(m, n):
    return n ** m

def one_to_one_functions(m, n):
    if m > n:
        return 0
    choices = [n - i for i in range(m)]
    return prod(choices)

def main():
    print("=== Extended Product Rule ===\n")
    print(f"Bit strings of length 7: {bit_strings(7)}")
    print(f"License plates (3 letters, 3 digits): {license_plates(3, 3):,}")
    print(f"Functions from 3->5: {functions(3,5)}")
    print(f"One-to-one from 3->5: {one_to_one_functions(3,5)}")
    print(f"Warm-up (8-bit strings): {bit_strings(8)}")
    print(f"Practice 1 (2 letters, 2 digits): {license_plates(2,2):,}")
    print(f"Practice 2 (one-to-one 4->6): {one_to_one_functions(4,6)}")
    print(f"Practice 3 (functions 10->3): {functions(10,3)} | one-to-one: {one_to_one_functions(10,3)}")
    print(f"Practice 4 (employee IDs): {26**2 * 5 * 1000:,}")
    print("\nDone!")

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

Reflection

The extended product rule shows how independence and sequence multiply possibilities. The pattern $n_1 n_2 \cdots n_m$ appears everywhere—from network addressing and password generation to database schema design and combinatorial search. Every time we build loops or nested conditionals in code, we are implicitly applying the same principle.

Chapter 4

Section 6.1 — Product Rule in Action

4.1 Real-World Applications of the Product Rule

Counting problems appear everywhere—from numbering systems to programming loops and data sets. The following examples show how the product rule powers many of these real-world situations.

—

Example 4.1 (The North American Telephone Numbering Plan). *The North American Numbering Plan (NANP) defines 10-digit phone numbers, split into a three-digit area code, a three-digit office code, and a four-digit station code.*

Let:

- N*: any digit from 2–9*
- Y*: either 0 or 1*
- X*: any digit 0–9*

Old Plan (NYX–NNX–XXXX):

$$\begin{aligned} \textit{Area codes: } & 8 \times 2 \times 10 = 160, \\ \textit{Office codes: } & 8 \times 8 \times 10 = 640, \\ \textit{Station codes: } & 10^4 = 10{,}000. \end{aligned}$$

Total: $160 \times 640 \times 10{,}000 = 1.024 \times 10^9$ *phone numbers.*

New Plan (NXX–NXX–XXXX):

$$\begin{aligned}\textit{Area codes: } & 8 \times 10 \times 10 = 800,\\ \textit{Office codes: } & 8 \times 10 \times 10 = 800,\\ \textit{Station codes: } & 10^4 = 10{,}000.\end{aligned}$$

Total: $800 \times 800 \times 10{,}000 = 6.4 \times 10^9$ *phone numbers.*

Solution. The new plan increases capacity by a factor of about six. A small change in one rule leads to billions more combinations—combinatorics in action. ◀

—

Practice Problems

1. Easier: A small town uses 7-digit numbers in the format NXX–XXX, where N is from 2–9 and X is from 0–9. How many phone numbers can be issued?

Solution:

$$8 \times 10^5 = 8{,}000{,}000.$$

2. Moderate: If the first two digits must be odd (1, 3, 5, 7, 9), how many numbers exist?

Solution:

$$5 \times 5 \times 10^5 = 2{,}500{,}000.$$

3. Stretch: If each number begins with one of 26 region letters (A–Z), how many distinct identifiers exist?

Solution:

$$26 \times 8{,}000{,}000 = 208{,}000{,}000.$$

—

Example 4.2 (Counting Nested Loops)**.** *Consider the pseudocode:*

```
k = 0
for i1 in range(1, n1+1):
    for i2 in range(1, n2+1):
        ...
        for im in range(1, nm+1):
            k += 1
```

Each loop represents one "task" T_i with n_i iterations. By the product rule, the innermost statement executes

$$n_1 \times n_2 \times \cdots \times n_m$$

times, so the final value of k is exactly that product.

—

Practice Problems

1. Easier: If the outer loop runs 5 times and the inner loop 4 times, how many increments occur?

$$5 \times 4 = 20.$$

2. Moderate: For three nested loops running 3, 4, and 2 times respectively:

$$3 \times 4 \times 2 = 24.$$

3. Stretch: Write a Python function that computes k automatically for any $(n_1, n_2, \ldots, n_m)$.

```
from math import prod
def nested_loops(*sizes):
    return prod(sizes)
print(nested_loops(3, 4, 2)) # Output: 24
```

—

Example 4.3 (Counting Subsets of a Finite Set)**.** *Use the product rule to show that a set S with $|S| = n$ has 2^n subsets.*

Solution. *For each element of S, decide whether to:*

1. *include it in the subset, or*
2. *exclude it.*

Two independent choices per element yield 2^n total subsets. Each subset corresponds to a unique bit string of length n.

—

Practice Problems

1. **Easier:** How many subsets does $\{a, b, c\}$ have?

$$2^3 = 8.$$

2. **Moderate:** If $|S| = 10$, how many subsets contain at least one element?

$$2^{10} - 1 = 1023.$$

3. **Stretch:** How many subsets of a set with n elements have exactly k members?

$$\binom{n}{k}.$$

—

4.2 Python Demonstrations

Listing 4.1: Counting with Loops and Sets

```python
from itertools import product, chain, combinations
from math import prod

# Telephone numbering comparison
def phone_numbers(area, office, station):
    return area * office * station

print("Old␣plan:", phone_numbers(160, 640, 10_000))
print("New␣plan:", phone_numbers(800, 800, 10_000))

# Nested loop counter
```

```
def nested_count(*sizes):
    return prod(sizes)

print("Nested␣loop␣3x4x2:", nested_count(3,4,2))

# Subsets demonstration
S = {'a', 'b', 'c'}
power_set = list(chain.from_iterable(combinations(S, r) for r in range(len(S)+1)))
print("Subsets␣of␣{a,b,c}:", power_set)
print("Count:", len(power_set))
```

—

Reflection

These three themes—telephone numbers, nested loops, and subsets—share one idea:

$$\text{Total outcomes} = \prod_i (\text{ways to complete task } i).$$

Whenever choices multiply, the product rule is at work. Computer scientists live inside this principle every day: loops, data encodings, and branching logic all trace their roots to simple multiplication of possibilities.