

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
5	Section 6.2 — DNA, Genomes, and the Sum Rule	27
5.1	DNA and Combinatorics	27
5.2	Introducing the Sum Rule	31
5.3	Python Demonstrations	35
6	Section 6.3 — Permutations and Combinations	37
6.1	Counting Arrangements and Selections	37
6.2	Combinations — When Order Does Not Matter	38
6.3	Python Demonstrations	41

7	Section 6.1.3 — More Complex Counting Problems	43
7.1	Worked Example A — Short variable names with reserved words	43
7.2	Worked Example B — 6–8 char passwords, at least one digit	44
7.3	Try It — Easier (solution on the next page)	45
7.4	Challenge — Harder (solution on the next page)	47
8	Section 6.1.4 — The Subtraction Rule (Avoiding Double Counting)	49
8.1	Introduction: When Counting Goes Wrong	49
8.2	Example 1: Bit Strings with Conditions (Rosen Example 18)	49
8.3	Python Demonstration	52

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 = $A - B$.

$$|A_6| = 36^6,$$

$$|B_6| = 26^6$$

$$|A_7| = 36^7,$$

$$|B_7| = 26^7$$

$$|A_8| = 36^8,$$

$$|B_8| = 26^8$$

$$\text{Total} = (36^6 - 26^6) + (36^7 - 26^7) + (36^8 - 26^8)$$

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

```
1  """
2  COMSC_2043_-_Counting_Demonstrations
3  Author: _Jeremy_Evert
4  This_script_demonstrates_the_basic_principles_of_counting_using_Python.
5  """
6
7  from math import comb, perm
8
9  def example_offices():
10     n1, n2 = 12, 11
11     total = n1 * n2
12     print(f"Example_1: _Assigning_offices\n12_*_11=_{total}\n")
13
14  def example_chairs():
15     total = 26 * 100
16     print(f"Example_2: _Chair_labeling\n26_*_100=_{total}\n")
17
18  def example_ports():
19     total = 32 * 24
20     print(f"Example_3: _Data_center_ports\n32_*_24=_{total}\n")
21
22  def example_license_plates():
23     allow_repeat = 26**3 * 10**3
24     no_repeat = (26*25*24) * (10*9*8)
25     print("Example_4: _License_plates")
26     print(f"_ _With_repetition: _{allow_repeat:,}")
27     print(f"_ _Without_repetition: _{no_repeat:,}\n")
28
29  def example_passwords():
30     # Helper function:  $A^n - B^n$ 
31     def count_valid(length):
32         return 36**length - 26**length
33
34     total = sum(count_valid(n) for n in (6, 7, 8))
35     print("Example_5: _Passwords_with_at_least_one_digit")
36     print(f"_ _Total_valid_passwords: _{total:,}\n")
37
```

```
38 def main():
39     print("=== Counting Demonstrations ===\n")
40     example_offices()
41     example_chairs()
42     example_ports()
43     example_license_plates()
44     example_passwords()
45     print("Done!")
46
47 if __name__ == "__main__":
48     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, \dots, 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:

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

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

```
1  """
2  COMSC_2043_-_Extended_Product_Rule_Demonstrations
3  Author:_Jeremy_Evert
4  """
5
6  from math import prod
7
8  def bit_strings(n):
9      return 2 ** n
10
11 def license_plates(letters, digits):
12     return 26 ** letters * 10 ** digits
13
14 def functions(m, n):
15     return n ** m
16
17 def one_to_one_functions(m, n):
18     if m > n:
19         return 0
20     choices = [n - i for i in range(m)]
21     return prod(choices)
22
23 def main():
24     print("===_Extended_Product_Rule_===\n")
25     print(f"Bit_strings_of_length_7:_{bit_strings(7)}")
26     print(f"License_plates_(3_letters,_3_digits):_{license_plates(3,3):,}")
27     print(f"Functions_from_3->5:_{functions(3,5)}")
28     print(f"One-to-one_from_3->5:_{one_to_one_functions(3,5)}")
```

```
29     print(f"Warm-up_{8-bit_strings}:{bit_strings(8)}")
30     print(f"Practice_1_{2_letters,2_digits}:{license_plates(2,2):,}")
31     print(f"Practice_2_{one-to-one_4->6}:{one_to_one_functions(4,6)}")
32     print(f"Practice_3_{functions_10->3}:{functions(10,3)}|_one-to-one:_{one_to_one_functions(10,3)}")
33     print(f"Practice_4_{employee_IDs}:{26**2*_5*_1000:,}")
34     print("\nDone!")
35
36 if __name__ == "__main__":
37     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):

Area codes: $8 \times 2 \times 10 = 160$,

Office codes: $8 \times 8 \times 10 = 640$,

Station codes: $10^4 = 10,000$.

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

New Plan (NXX–NXX–XXXX):

Area codes: $8 \times 10 \times 10 = 800$,

Office codes: $8 \times 10 \times 10 = 800$,

Station codes: $10^4 = 10,000$.

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:

```

1 k = 0
2 for i1 in range(1, n1+1):
3     for i2 in range(1, n2+1):
4         ...
5         for im in range(1, nm+1):
6             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, \dots, n_m) .

```

1 from math import prod
2 def nested_loops(*sizes):
3     return prod(sizes)
4 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

```
1 from itertools import product, chain, combinations
2 from math import prod
3
4 # Telephone numbering comparison
5 def phone_numbers(area, office, station):
6     return area * office * station
7
8 print("Old_plan:", phone_numbers(160, 640, 10_000))
9 print("New_plan:", phone_numbers(800, 800, 10_000))
10
11 # Nested loop counter
```



```
12 def nested_count(*sizes):
13     return prod(sizes)
14
15 print("Nested_loop_3x4x2:", nested_count(3,4,2))
16
17 # Subsets demonstration
18 S = {'a', 'b', 'c'}
19 power_set = list(chain.from_iterable(combinations(S, r) for r in range(len(S)+1)))
20 print("Subsets_of_{a,b,c}:", power_set)
21 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.

Chapter 5

Section 6.2 — DNA, Genomes, and the Sum Rule

5.1 DNA and Combinatorics

DNA (deoxyribonucleic acid) and RNA (ribonucleic acid) encode the very instructions of life. Each strand of DNA consists of *nucleotides*, with four possible bases:

A (adenine), C (cytosine), G (guanine), T (thymine).

In RNA, thymine (T) is replaced by uracil (U). These bases form long chains whose order determines genes and, ultimately, the proteins that define an organism.

From a combinatorial perspective, each base position can be filled in one of four ways. Hence, a DNA strand with n bases corresponds to 4^n possible sequences — the **product rule** in biological disguise.

—

Example 5.1 (Encoding Amino Acids). Proteins are built from amino acids. Humans use 22 essential amino acids, and each amino acid is encoded by a *codon* — a short sequence of bases.

Reasoning:

- One base: $4^1 = 4$ possibilities (not enough).
- Two bases: $4^2 = 16 < 22$ (still too few).
- Three bases: $4^3 = 64$ (more than enough).

Therefore, a codon must consist of three bases, providing 64 distinct triplets to encode 22 amino acids. This redundancy helps prevent catastrophic errors from single-base mutations.

Solution. Nature, like a clever coder, includes redundancy and error recovery in its data encoding. The product rule explains why three-base codons are the smallest possible unit that can represent all amino acids.

—

Example 5.2 (Counting DNA Sequences). DNA length varies dramatically among organisms:

- Simple organisms (e.g., bacteria): 10^5 – 10^7 bases.
- Complex organisms (e.g., mammals): 10^8 – 10^{10} bases.

By the product rule, a DNA molecule of length n has 4^n possible base sequences:

$$4^{10^5} \quad \text{for bacteria,} \quad 4^{10^8} \quad \text{for mammals.}$$

These quantities are astronomically large, illustrating why genetic diversity is effectively limitless.

Solution. Even though DNA uses just four symbols, its combinatorial potential is enormous. This exponential explosion of possible sequences ensures endless biological variety — one reason every living thing is unique.

—

Practice Problems — DNA and Counting

1. Easier: How many distinct RNA sequences of length four are possible?

Solution: Each position has four choices (A, C, G, U):

$$4^4 = 256 \text{ possible sequences.}$$

—

2. Moderate: A gene contains 12 bases. How many unique base sequences can it have?

Solution:

$$4^{12} = 16,777,216.$$

Even a short gene can produce over sixteen million variations.

—

3. Stretch: A virus has a genome with 30,000 bases. Estimate $4^{30,000}$ in powers of ten.

Solution:

$$4^{30,000} = (2^2)^{30,000} = 2^{60,000} \approx 10^{18,000}.$$

That's a one followed by eighteen thousand zeros — a vast genetic universe.

—

5.2 Introducing the Sum Rule

So far, our counting used multiplication — combining independent choices. But what if we have *mutually exclusive* options, and we must choose exactly one? That's where the **sum rule** comes in.

Definition 5.1 (Sum Rule). If a task can be done in n_1 ways or in n_2 ways, where the sets of outcomes are disjoint (no overlap), then the total number of ways is:

$$n_1 + n_2.$$

—

Example 5.3 (Faculty or Student Representative). Suppose either a member of the mathematics faculty or a mathematics major is chosen to serve on a university committee.

Given:

$$37 \text{ faculty members,} \quad 83 \text{ students.}$$

Since no one is both faculty and student, the total number of possible representatives is:

$$37 + 83 = 120.$$

Solution. The sum rule models disjoint alternatives — you choose one group or the other, not both. In programming, this parallels an `if/else` decision branch.

—

Example 5.4 (Choosing from Multiple Project Lists). A student can pick one project from any of three lists containing 23, 15, and 19 options, respectively. No project appears on more than one list.

Solution:

$$23 + 15 + 19 = 57$$

possible projects in total.

Solution. Each list represents a distinct pool of choices — separate “doors” leading to different sets of outcomes. Adding their sizes counts all unique possibilities.

—

Practice Problems — Applying the Sum Rule

1. Easier: You can adopt either a cat (6 breeds) or a dog (9 breeds) from the shelter. How many total options are there?

Solution:

$$6 + 9 = 15.$$

—

2. Moderate: A restaurant offers 12 vegetarian entrées and 8 meat dishes. If you can choose only one, how many possible meals are there?

Solution:

$$12 + 8 = 20.$$

—

3. Stretch: A student can take one course chosen from:

- 5 online classes,
- 3 hybrid classes, or
- 4 in-person classes.

No course appears in more than one category. How many total courses are available?

Solution:

$$5 + 3 + 4 = 12.$$

This illustrates the extended sum rule:

$$n_1 + n_2 + \cdots + n_m.$$

5.3 Python Demonstrations

Listing 5.1: DNA and Sum Rule Demonstrations

```

1  """
2  COMSC_2043_DNA_and_Sum_Rule_Examples
3  Author: Jeremy Evert
4  """
5
6  from math import pow
7
8  # DNA combinatorics
9  def dna_sequences(length):
10     """Returns the number of possible DNA (or RNA) sequences of given length."""
11     return int(pow(4, length))
12
13  # Sum rule demonstrations
14  faculty, students = 37, 83
15  project_lists = [23, 15, 19]
16
17  print("=== DNA and Sum Rule Examples ===\n")
18  print(f"DNA sequences of length 4: {dna_sequences(4)}")
19  print(f"DNA sequences of length 12: {dna_sequences(12):,}")
20  print(f"Committee choices (faculty or students): {faculty+students}")
21  print(f"Total projects across lists: {sum(project_lists)}")
22  print("\nDone!")

```

Reflection

The **product rule** multiplies possibilities; the **sum rule** adds alternatives. Together they are the foundation of all combinatorics — and of all computation. Loops represent multiplication (repetition of actions), while conditional branches represent addition (choosing between paths). From DNA sequences to decision trees, counting principles are the quiet mathematics behind every system of order — living or digital.

Chapter 6

Section 6.3 — Permutations and Combinations

6.1 Counting Arrangements and Selections

When counting, we often face two big questions:

1. Does *order* matter?
2. Can we *reuse* elements?

If order matters, we are dealing with **permutations**. If order does not matter, we are counting **combinations**. Let's unpack each idea carefully.

—

Example 6.1 (Permutations — When Order Matters). Suppose you have three different books and you want to know how many ways they can be arranged on a shelf.

Reasoning.

- 3 choices for the first position.
- 2 choices for the second position.
- 1 choice for the last position.

By the product rule:

$$3 \times 2 \times 1 = 6.$$

We call this $3!$ (“three factorial”). In general, $n! = n \times (n - 1) \times \cdots \times 1$.

Solution. If you label the books A, B, and C, the six permutations are:

ABC, ACB, BAC, BCA, CAB, CBA.

Factorials grow explosively— $10! = 3,628,800$ —so a few extra items make a huge difference!

—

Example 6.2 (Permutations of Subsets). How many ways can you arrange 4 of 10 distinct students in a line?

Reasoning. You are selecting 4 *different* people, and the order matters.

$$P(10, 4) = \frac{10!}{(10 - 4)!} = \frac{10!}{6!} = 10 \times 9 \times 8 \times 7 = 5040.$$

Solution. $P(n, r)$ counts the number of ways to arrange r objects from a set of n . Python's `math.perm(n, r)` computes this directly.

—

6.2 Combinations — When Order Does Not Matter

When order no longer matters, we divide out the repeated arrangements that represent the same group.

Example 6.3 (Combinations Formula). From 10 students, how many groups of 4 can you form?

Reasoning.

$$C(10, 4) = \frac{10!}{4! \times (10 - 4)!} = \frac{10!}{4! 6!} = 210.$$

Solution. In Python, use `math.comb(10, 4)` to get the same answer. Combinations appear in probability, lotteries, and team selections.

—

Practice Problems — Permutations and Combinations

1. **Easier:** How many ways can 5 different books be arranged on a shelf?

Solution:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

—

2. Moderate: From a group of 8 students, how many teams of 3 can be formed?

Solution:

$$C(8, 3) = \frac{8!}{3! 5!} = 56.$$

—

3. Stretch: How many 4-digit PINs can be formed using digits 0–9 if:

1. Digits cannot repeat.
2. Digits may repeat.

Solution:

$$(a) \text{ No repeats: } P(10, 4) = \frac{10!}{6!} = 5040.$$

$$(b) \text{ Repeats allowed: } 10^4 = 10,000.$$

This comparison highlights how reuse (repetition) increases the count dramatically.

—

6.3 Python Demonstrations

Listing 6.1: Exploring permutations and combinations in Python

```

1  """
2  COMSC_2043_-_Section_6.3:_Permutations_&_Combinations
3  """
4
5  import math
6
7  # 1. Factorials
8  print("5!=", math.factorial(5))
9
10 # 2. Permutations (order matters)
11 print("Permutations_of_10_choose_4:", math.perm(10, 4))
12
13 # 3. Combinations (order does not matter)
14 print("Combinations_of_10_choose_4:", math.comb(10, 4))
15
16 # 4. Comparing repetition
17 no_repeats = math.perm(10, 4)
18 with_repeats = 10**4
19 print(f"No_repeats:_{no_repeats},_With_repeats:_{with_repeats}")

```

—

Reflection

Permutations and combinations represent the heart of counting theory. They explain why adding order multiplies possibilities—and why ignoring it divides them out. When students

grasp this duality, they start to see connections between probability, coding loops, and even DNA patterns.

Remember:

$$\text{Permutations: } P(n, r) = \frac{n!}{(n-r)!}, \quad \text{Combinations: } C(n, r) = \frac{n!}{r!(n-r)!}.$$

Chapter 7

Section 6.1.3 — More Complex Counting Problems

Why this section matters

Up to now, we’ve used the product rule (choices multiply) and the sum rule (disjoint options add). Many real problems mix both ideas and add constraints (like “must begin with a letter” or “must include at least one digit”). Today’s patterns: (i) break by *length* and *first-character rules*, (ii) count with the product rule, and (iii) subtract forbidden cases (complements and inclusion–exclusion).

7.1 Worked Example A — Short variable names with reserved words

Example 7.1. A toy language allows variable names of length 1 or 2. Characters are alphanumeric (letters/digits) but *the first character must be a letter*. Upper/lowercase are treated the same. Additionally, *five specific two-character strings are reserved* and therefore not allowed as variable names. How many valid variable names exist?

Solution. Let V be the number of valid names. Split by length:

Length 1. First character must be a letter \Rightarrow 26 possibilities.

Length 2. First char: 26 choices (letter). Second char: 36 choices (letter or digit). So

$$26 \cdot 36 = 936 \text{ raw two-character strings.}$$

But five of these two-character strings are reserved. Exclude them:

$$V_2 = 936 - 5 = 931.$$

Total. $V = V_1 + V_2 = 26 + 931 = \boxed{957}$.

Sanity checks. (1) The “first is a letter” constraint is applied to both lengths. (2) The -5 only hits two-character names (not the length-1 pool). (3) We didn’t accidentally double-subtract: reserved items are disjoint from the length-1 set.

Python quick check.

```

1 import string
2 letters = string.ascii_uppercase # treat case-insensitive
3 alnum = string.ascii_uppercase + string.digits
4
5 reserved = {"IF", "DO", "TO", "ON", "GO"} # example 5
6 V1 = len(letters)
7 V2 = sum(1 for a in letters for b in alnum if (a+b) not in reserved)
8 print(V1 + V2) # 957

```

7.2 Worked Example B — 6–8 char passwords, at least one digit

Example 7.2. A password is 6, 7, or 8 characters long. Each character is an uppercase letter or a digit. A password is valid only if it includes *at least one digit*. How many valid passwords are there in total?

Solution. For a fixed length n , count all strings over $\{A \dots Z, 0 \dots 9\}$ and subtract the “all letters” strings:

$$P_n = 36^n - 26^n.$$

Sum over $n \in \{6, 7, 8\}$:

$$P = (36^6 - 26^6) + (36^7 - 26^7) + (36^8 - 26^8).$$

Numerically,

$$\begin{aligned}36^6 - 26^6 &= 2,176,782,336 - 308,915,776 = 1,867,866,560, \\36^7 - 26^7 &= 78,364,164,096 - 8,031,810,176 = 70,332,353,920, \\36^8 - 26^8 &= 2,821,109,907,456 - 208,827,064,576 = 2,612,282,842,880.\end{aligned}$$

Therefore

$$P = 2,684,483,063,360.$$

Common gotchas. (1) The complement must match the exact constraint (“at least one digit” \Rightarrow complement is “no digits”). (2) Don’t multiply by 3 lengths; you *add* across mutually exclusive lengths (sum rule). (3) No double-counting across lengths because 6/7/8 are disjoint cases.

Python spot-check.

```
1 def valid(n): return 36**n - 26**n
2 total = sum(valid(n) for n in (6,7,8))
3 print(f"{total:,}") # 2,684,483,063,360
```

7.3 Try It — Easier (solution on the next page)

A promo code is either *four* or *five* characters long. Each character is a letter or digit, and every promo code must contain *at least one letter*. How many promo codes are possible?

Solution (Easier)

Let Q_n be the count for fixed length n . Use a complement again: “at least one letter” \Rightarrow subtract “no letters” (i.e., *all digits*).

$$Q_n = 36^n - 10^n.$$

Sum for $n \in \{4, 5\}$:

$Q = (36^4 - 10^4) + (36^5 - 10^5) = 1,679,616 - 10,000 + 60,466,176 - 100,000 = 62,035,792.$

7.4 Challenge — Harder (solution on the next page)

Passwords are still 6–8 characters using letters/digits, but now add two constraints:

1. The **first character must be a letter**, and
2. The password must contain **at least two digits** overall.

How many valid passwords exist?

Solution (Harder)

Fix a length $n \in \{6, 7, 8\}$. Force the first character to be a letter (26 ways). The remaining $n - 1$ positions each have 36 choices. We need *at least two digits across those $n - 1$ trailing positions*. Count by complement on the tail:

For the last $n - 1$ positions:

$$\#(\text{at least 2 digits}) = 36^{n-1} - \underbrace{26^{n-1}}_{0 \text{ digits}} - \underbrace{\binom{n-1}{1} \cdot 10 \cdot 26^{n-2}}_{\text{exactly 1 digit}}.$$

Multiply by 26 for the first letter:

$$H_n = 26 \left(36^{n-1} - 26^{n-1} - (n-1) \cdot 10 \cdot 26^{n-2} \right).$$

Compute each n and sum:

$$\begin{aligned} H_6 &= 26(36^5 - 26^5 - 5 \cdot 10 \cdot 26^4) = \boxed{669,136,000}, \\ H_7 &= 26(36^6 - 26^6 - 6 \cdot 10 \cdot 26^5) = \boxed{30,029,584,000}, \\ H_8 &= 26(36^7 - 26^7 - 7 \cdot 10 \cdot 26^6) = \boxed{1,266,414,489,600}. \end{aligned}$$

Therefore,

$$\boxed{H = H_6 + H_7 + H_8 = 1,297,113,209,600.}$$

Why inclusion–exclusion? “At least two digits” means we must remove both the 0-digit and 1-digit cases from the tail. Exactly-one-digit strings on the tail: choose the position ($n - 1$ ways), choose the digit (10), and fill others with letters (26^{n-2}).

Python verification.

```

1 def count_at_least_two_with_first_letter(n):
2     from math import comb
3     tail = 36**(n-1) - 26**(n-1) - comb(n-1,1)*10*26**(n-2)
4     return 26 * tail
5
6 print(sum(count_at_least_two_with_first_letter(n) for n in (6,7,8)))
7 # 1,297,113,209,600

```


Chapter 8

Section 6.1.4 — The Subtraction Rule (Avoiding Double Counting)

8.1 Introduction: When Counting Goes Wrong

By now, we’ve seen how to count outcomes using the **Product Rule** (for “and”) and the **Sum Rule** (for “or”). But what if we count the same thing twice?

That’s where the **Subtraction Rule** comes in. It tells us how to avoid double-counting when two sets overlap.

Definition 8.1 (Subtraction Rule). If a task can be done in n_1 ways or in n_2 ways, but some outcomes are counted twice because they can be done both ways (there are $n_{1,2}$ overlaps), then the correct total is:

$$n_1 + n_2 - n_{1,2}.$$

This simple idea is foundational in combinatorics and probability — and a secret weapon against overcounting errors.

—

8.2 Example 1: Bit Strings with Conditions (Rosen Example 18)

Example 8.1 (Counting Bit Strings of Length 8). How many bit strings of length 8 start with a 1 *or* end with the two bits 00?

Solution Outline:

- Let A = set of 8-bit strings that start with a 1.

- Let B = set of 8-bit strings that end with 00.
- We want $|A \cup B| = |A| + |B| - |A \cap B|$.

Step 1: Count $|A|$. If the first bit is fixed as 1, there are 7 free bits left:

$$|A| = 2^7 = 128.$$

Step 2: Count $|B|$. If the last two bits are fixed as 00, there are 6 free bits:

$$|B| = 2^6 = 64.$$

Step 3: Count the overlap $|A \cap B|$. If a string both starts with 1 *and* ends with 00, that leaves 5 free bits in the middle:

$$|A \cap B| = 2^5 = 32.$$

Step 4: Apply the Subtraction Rule.

$$|A \cup B| = 128 + 64 - 32 = 160.$$

Solution. So, there are 160 bit strings of length 8 that start with 1 or end with 00.

Key Idea: The subtraction corrects for overlap — things counted twice. Without it, you'd have $128 + 64 = 192$, which overcounts by the 32 strings that fit both patterns.

—

Practice Problems — Subtraction Rule in Action

1. Easier Problem: How many 5-bit strings start with 0 or end with 1?

Solution:

$$|A| = 2^4 = 16, \quad |B| = 2^4 = 16, \quad |A \cap B| = 2^3 = 8.$$

$$|A \cup B| = 16 + 16 - 8 = 24.$$

Answer: 24 such bit strings exist.

—

2. Stretch Problem: How many 10-bit strings start with “11” or end with “000”?

Solution:

- $|A|$: Starts with 11 $\Rightarrow 2^8 = 256$.
- $|B|$: Ends with 000 $\Rightarrow 2^7 = 128$.
- $|A \cap B|$: Starts with 11 and ends with 000 $\Rightarrow 2^5 = 32$.

$$|A \cup B| = 256 + 128 - 32 = 352.$$

Answer: 352 such bit strings exist.

—

8.3 Python Demonstration

Listing 8.1: Demonstrating the Subtraction Rule with Bit Strings

```

1  """
2  COMSC_2043_Subtraction_Rule_Examples
3  Author: Jeremy Evert
4  """
5
6  def count_subtraction_rule(n1, n2, overlap):
7      """Apply the subtraction rule."""
8      return n1 + n2 - overlap
9
10 # Example 1: 8-bit strings starting with 1 or ending with 00
11 A = 2**7 # starts with 1
12 B = 2**6 # ends with 00
13 overlap = 2**5 # both conditions
14 print("8-bit strings:", count_subtraction_rule(A, B, overlap))
15
16 # Practice check: 10-bit version
17 A, B, overlap = 2**8, 2**7, 2**5
18 print("10-bit strings:", count_subtraction_rule(A, B, overlap))

```

—

Reflection

The subtraction rule protects us from our own enthusiasm — we tend to count everything, including repeats! Mathematically, it's the bridge from simple counting to true combinatorial reasoning.

It also underlies many powerful ideas in computer science:

- detecting overlaps in sets (databases and queries),
- correcting for double-counted results in logic and probability,
- and later — the **Inclusion–Exclusion Principle**, which generalizes this very rule.

The moral? When two paths overlap, you must subtract the crossroads.