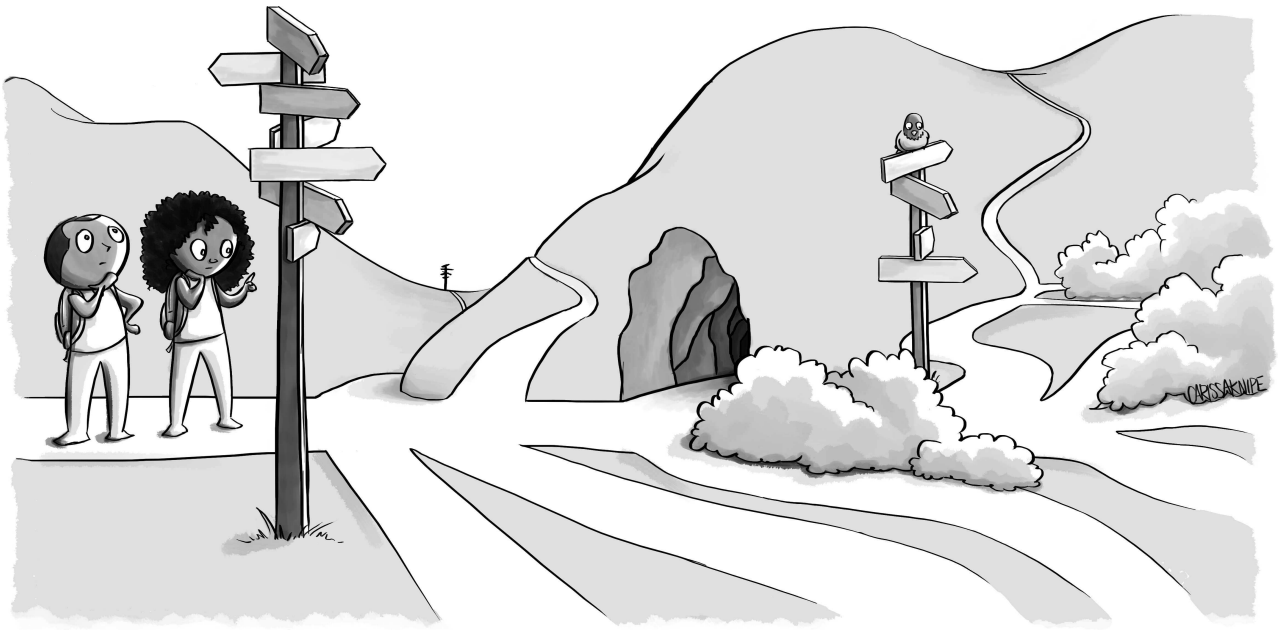
9
Counting



In which our heroes encounter many choices, some of which may lead them to live more happily than others, and a precise count of their number of options is calculated.

9.1 Why You Might Care

> How do I love thee? Let me count the ways.
>
> Elizabeth Barrett Browning (1806–1861)

This chapter is devoted to the apparently trivial task of *counting*. By "counting," we mean the following problem: given a potentially convoluted description of a set S, compute the cardinality of S—that is, compute the number of elements in S. It may seem bizarre that counting could somehow be harder than at the preschool level (just count! *one, two, three*), but it will turn out that we can solve surprisingly subtle problems with some useful and general (and subtle) techniques.

We'll start in Section 9.2 by introducing basic counting techniques—how to compute the cardinality of a union $A \cup B$ of two sets, or sequences from the Cartesian product $A \times B$ of two sets. We then turn in Section 9.3 to one of the best counting strategies: *being lazy!* If we can show that $|A| = |B|$ and we already know the value of $|B|$, then figuring out $|A|$ is easy; we'll often use functions to relate two sets so that we can then lazily compute the size of the apparently harder-to-count set. Finally, in Section 9.4, we will explore *combinations* ("how many ways are there to choose an unordered collection of k items out of a set of n possibilities?") and *permutations* ("how many ways are there to put a set of n items into some order?").

Why does counting matter in computer science? There are, again, surprisingly many applications. Here are a few examples. One common (though very basic) style of algorithm is a *brute-force algorithm*, which finds the best *whatzit* by trying every possible *whatzit* and seeing which one is best. Determining whether a brute-force algorithm is fast enough depends on counting how many possible *whatzit*s there are. A more advanced algorithmic design technique, called *dynamic programming*, can be used to design efficient recursive solutions to problems—as long as there aren't too many *distinct* subproblems. Counting techniques are even powerful enough to establish a mind-bending result about *computability*: we will be able to prove that *there are more problems than computer programs*—which means that there are some problems that cannot be solved by any program!

Probability (see Chapter 10) has a plethora of applications in computer science, ranging from randomized algorithms in sorting (algorithms that process their input by making random decisions about how to act) to models of random noise in speech recognition or random errors in typing (if I'm trying to type the letter `p`, what is the chance that I accidentally type `o` instead?). We can think of the probability of some event X happening, roughly, as two counting problems: the numerator and denominator of the ratio

$$\frac{\text{the number of ways } X \text{ can happen}}{\text{the number of ways } X \text{ can either happen or not happen}}.$$

There are many other applications of counting scattered throughout computer science, and we will discuss a few more along the way: breaking cryptographic systems, compressing audio/image/video files, and changing the addressing scheme on the internet because we've run out of smaller addresses, to name a few.

9.2 Counting Unions and Sequences

> If a man who cannot count finds a four-leaf clover, is he entitled to happiness?
>
> Stanislaw J. Lec (1909–1966)

Suppose that we have two sets A and B from which we must choose an element. There are two different natural scenarios that meet this one-sentence description: we must choose a total of one element from *either* A or B, or we must choose one element from *each* of A and B. For example, consider a restaurant that offers soups $A = \{\text{chicken noodle}, \text{beer cheese}, \text{minestrone}, \ldots\}$ and salads $B = \{\text{caesar}, \text{house}, \text{arugula}, \ldots\}$. A lunch special that includes soup *or* salad involves choosing an $x \in A \cup B$. A dinner special including soup *and* salad involves choosing an $x \in A$ and also choosing a $y \in B$—that is, choosing an element $\langle x, y \rangle \in A \times B$. In Section 9.2.1, we'll start with two basic rules for computing these cardinalities:

- *Sum Rule:* If A and B are disjoint, then $|A \cup B| = |A| + |B|$.
- *Product Rule:* The number of pairs $\langle x, y \rangle$ with $x \in A$ and $y \in B$ is $|A \times B| = |A| \cdot |B|$.

These rules will handle the simple restaurant scenarios above, but there are a pair of extensions that we'll introduce to handle slightly more complex situations. The first (Section 9.2.2) extends the Sum Rule to allow us to calculate the cardinality of a union of two sets *even if* those sets may contain elements in common:

- *Inclusion–Exclusion:* $|A \cup B| = |A| + |B| - |A \cap B|$.

The second extension (Section 9.2.3) generalizes the Product Rule to allow us to calculate the cardinality of a set of pairs $\langle x, y \rangle$ even if the choice of x changes the list (but not the number) of possible choices for y:

- *Generalized Product Rule:* Consider pairs $\langle x, y \rangle$ of the following form: we can choose any $x \in A$, and, for each such x, there are precisely n different choices for y. Then the total number of pairs meeting this description is $|A| \cdot n$.

The remainder of this section will give the details of these four rules, and how to use these rules individually and in combination.

9.2.1 The Basics: The Sum and Product Rules

Sum Rule: counting unions

Our first rule addresses the *union* of two sets: if two sets A and B are disjoint, then the cardinality of their union is simply the sum of their sizes:

Theorem 9.1 (Sum Rule)
Let A and B be sets. If $A \cap B = \varnothing$, then $|A \cup B| = |A| + |B|$.

More generally, consider a collection of $k \geq 1$ sets $A_1, A_2, \ldots, A_k$. If these sets are all disjoint—that is, if $A_i \cap A_j = \varnothing$ whenever $i \neq j$—then the cardinality of their union is the sum of their cardinalities: $|A_1 \cup A_2 \cup \cdots \cup A_k| = |A_1| + |A_2| + \cdots + |A_k|$.

The Sum Rule captures an intuitive fact: if a box contains some red things and some blue things, then the total number of things in the box is the number of red things plus the number of blue things. Here are a few examples that use this rule:

Example 9.1 (Counting disjoint unions)

- Let $A := \{1, 2\}$ and $B := \{3, 4, 5, 6\}$. Thus $|A| = 2$ and $|B| = 4$. Observe that the sets A and B are disjoint. By the sum rule, $|A \cup B| = |A| + |B| = 2 + 4 = 6$. Indeed, we have $A \cup B = \{1, 2, 3, 4, 5, 6\}$, which contains 6 elements.
- There are 11 starters on your school's women's soccer team. Suppose there are 8 nonstarters on the team. The total number of people on the team is $19 = 11 + 8$.
- At a certain school in the midwest, there are currently 30 computer science majors who are studying abroad. There are 89 computer science majors who are studying on campus. Then the total number of computer science majors is $119 = 89 + 30$.
- Consider a computer lab that contains 32 Macs and 14 PCs and 1 PDP-8 (a 1960s-era machine, one of the first computers that was sold commercially). Then the total number of computers in the lab is $47 = 32 + 14 + 1$.

Example 9.2 (Students in classes)

Problem: During this term, there are 19 students taking Data Structures, and 39 students taking Mathematics of Computer Science. Let S denote the set of students taking Data Structures *or* Mathematics of Computer Science this term. What is $|S|$?

Solution: There isn't enough information to answer the question!

- *If* there are no students who are taking both classes (that is, if $DS \cap MOCS = \varnothing$), then $|S| = |DS| + |MOCS| = 19 + 39 = 58$.
- But, for all we know from the problem statement, every student in Data Structures is also taking Mathematics of Computer Science. In this case, we have $DS \subset MOCS$ and thus $S = DS \cup MOCS = MOCS$; therefore $|S| = |MOCS| = 39$.

(The *Inclusion–Exclusion Rule*, in Section 9.2.2, formalizes the calculation of $|A \cup B|$ in terms of $|A|$, $|B|$, and $|A \cap B|$, in the manner that we just considered.)

Taking it further: The logic that we used in Example 9.2 to conclude that there were at most 58 students in the two classes combined is an application of the general fact that $|A \cup B| \leq |A| + |B|$. While this fact is pretty simple, it turns out to be remarkably useful in proving facts about probability. The *Union Bound* states that the probability that *any* of $A_1, A_2, \ldots, A_k$ occurs is at most $p_1 + p_2 + \cdots + p_k$, where p_i denotes the probability that A_i occurs. The Union Bound turns out to be useful when each A_i is a "bad event" that we're worried might happen, and these bad events may have complicated probabilistic dependencies—but if we can show that the probability that every particular one of these bad events is some very small ε, then we can use the Union Bound to conclude that the probability of experiencing *any* bad event is at most $k \cdot \varepsilon$. (See Exercise 10.141, for example.)

Using the Sum Rule in less obvious settings

As a general strategy for solving counting problems, we can try to find a way to apply the Sum Rule—even if it does not superficially seem to be applicable. If we can find a way to partition an apparently complicated set S into simple disjoint sets $S_1, S_2, \ldots, S_k$ such that $\bigcup_{i=1}^{k} S_i = S$, then we can use the Sum Rule to find $|S|$.

In this spirit, here's a somewhat more complex example of using the Sum Rule, where we have to figure out the subsets ourselves: let's determine how many 8-bit strings contain precisely two ones. (The full list of the bitstrings meeting this condition appears in Figure 9.1.)

```
11000000  01100000  00110000  00011000  00001100  00000110  00000011
          10100000  01010000  00101000  00010100  00001010  00000101
                    10010000  01001000  00100100  00010010  00001001
                              10001000  01000100  00100010  00010001
                                        10000100  01000010  00100001
                                                  10000010  01000001
                                                            10000001
```

Figure 9.1: All bitstrings in $\{0,1\}^8$ that contain exactly two ones.

Example 9.3 (8-bit strings with exactly 2 ones)
Problem: How many elements of $\{0,1\}^8$ have precisely two 1s?

Solution: Obviously, we can just count the number of bitstrings in Figure 9.1, which yields the answer: there are 28 such bitstrings. But let's use the Sum Rule instead.

What does a bitstring $x \in \{0,1\}^8$ with two ones look like? There must be two indices i and j—say with $i > j$—such that $x_i = x_j = 1$, and all other components of x must be 0:

$$x = \underbrace{00\cdots0}_{j-1 \text{ zeros}} \overbrace{1}^{\text{one in position } j} \underbrace{00\cdots0}_{i-j-1 \text{ zeros}} \overbrace{1}^{\text{one in position } i} \underbrace{00\cdots0}_{8-i \text{ zeros}}.$$

(For example, the bitstring `01001000` has ones in positions $j = 2$ and $i = 5$, interspersed with an initial block of $j - 1 = 1$ zero, a block of $i - j - 1 = 2$ between-the-ones zeros, and a block of $8 - i = 3$ final zeros.)

We are going to divide the set of 8-bit strings with two 1s *based on the index i*. That is, suppose that $x \in \{0,1\}^8$ contains two ones, and the *second* 1 in x appears in bit position #i. Then there are $i - 1$ positions in which the *first* one could appear—any of the slots $j \in \{1, 2, \ldots, i-1\}$ that come before i. (See Figure 9.1, where the $(i-1)$st column contains all $i - 1$ bitstrings whose second 1 appears in position #i. For example, column #3 contains the 3 bitstrings with $x_{4,5,6,7,8} =$ `10000`: that is, `10010000`, `01010000`, and `00110000`.) Because every x with exactly two ones has an index i of its second 1, we can use the Sum Rule to say that the answer to the given question is

$$\begin{aligned}\sum_{i=1}^{8} \big[\text{number of bitstrings with the second 1 in position } i\big] &= \sum_{i=1}^{8} (i-1) \\ &= 0 + 1 + \cdots + 7 \\ &= 28.\end{aligned}$$

(We'll also see another way to solve this example later, in Example 9.39.)

Problem-solving tip: When you're trying to find the cardinality of a complicated set S, try to find a way to split S into a collection of simpler disjoint sets, and then apply the Sum Rule.

Let's also generalize this example to bitstrings of arbitrary length:

Example 9.4 (k-bit strings with exactly 2 ones)
Consider the set $S := \{x \in \{0,1\}^k : x \text{ has precisely two 1s}\}$. As in Example 9.3, every bitstring $x \in S$ has an index i of its second 1; we'll use the value of i to partition S into sets that can be easily counted, and then use the Sum Rule to find $|S|$. Specifically, for each index i with $1 \leq i \leq k$, define the set

$$\begin{aligned} S_i &= \{x \in S : x_i = 1 \text{ and } x_{i+1} = x_{i+2} = \cdots = x_k = 0\}. \\ &= \left\{x \in \{0,1\}^k : \left[\exists j \leq i-1 : x_i = x_j = 1 \text{ and } x \text{ has no other 1s}\right]\right\}. \end{aligned}$$

Observe that $|S_i| = i - 1$: there are $i - 1$ different possible values of j. Also, observe that $S = \bigcup_{i=1}^{k} S_i$ and that, for any $i \neq i'$, the sets S_i and $S_{i'}$ are disjoint. Thus

$$|S| = \left|\bigcup_{i=1}^{k} S_i\right| = \sum_{i=1}^{k} |S_i| = \sum_{i=1}^{k} (i-1) = \frac{k(k-1)}{2} \qquad (*)$$

by the Sum Rule and the formula for the sum of the first n integers (Example 5.4).

As a check of our formula, let's verify our solution for some small values of k:

- For $k = 2$, $(*)$ says there are $\frac{2(2-1)}{2} = 1$ strings with two 1s. Indeed, there's just one: `11`.
- For $k = 3$, indeed there are $\frac{3(3-1)}{2} = 3$ strings with two 1s: `011`, `101`, and `110`.
- For $k = 4$, there are $\frac{4 \cdot 3}{2} = 6$ such strings: `1100`, `1010`, `0110`, `1001`, `0101`, and `0011`.

Note that $(*)$ matches Example 9.3: for $k = 8$, we have $28 = \frac{8 \cdot 7}{2}$ strings with two 1s.

Problem-solving tip: Check to make sure your formulas are reasonable by testing them for small inputs (as we did in Example 9.4).

Product Rule: counting sequences

Our second basic counting rule addresses the *Cartesian product* of sets. Recall that, for sets A and B, the Cartesian product $A \times B$ consists of all pairs $\langle a, b\rangle$ with $a \in A$ and $b \in B$. (For example, $\{1,2,3\} \times \{\mathtt{x},\mathtt{y}\} = \{\langle 1,\mathtt{x}\rangle, \langle 1,\mathtt{y}\rangle, \langle 2,\mathtt{x}\rangle, \langle 2,\mathtt{y}\rangle, \langle 3,\mathtt{x}\rangle, \langle 3,\mathtt{y}\rangle\}$.) The cardinality of $A \times B$ is the product of the cardinalities of A and B:

Theorem 9.2 (Product Rule)
Let A and B be sets. Then $|A \times B| = |A| \cdot |B|$.

More generally, consider a collection of k arbitrary sets $A_1, A_2, \ldots, A_k$, and consider the set of k-element sequences where, for each i, the ith component is an element of A_i. The number of such sequences is given by the product of the sets' cardinalities:

$$|A_1 \times A_2 \times \cdots \times A_k| = |A_1| \cdot |A_2| \cdot \cdots \cdot |A_k|.$$

Here are a few examples of counting using the Product Rule:

Example 9.5 (Counting sequences)

- Let $A := \{1,2\}$ and $B := \{3,4,5,6\}$. By the product rule, $|A \times B| = |A| \cdot |B| = 2 \cdot 4 = 8$. Indeed, $A \times B = \{\langle 1,3\rangle, \langle 1,4\rangle, \langle 1,5\rangle, \langle 1,6\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 2,5\rangle, \langle 2,6\rangle\}$, which contains 8 elements.
- At a certain school in the midwest, there are currently 56 senior computer science majors and 63 junior computer science majors. Then the number of ways to choose a pair of class representatives, one senior and one junior, is $56 \cdot 63 = 3528$.
- Consider a tablet computer that is sold with three different options: a choice of protective cover, a choice of stylus, and a color. If there are 7 different styles of protective cover, 5 different styles of stylus, and 3 different colors, then there are $7 \cdot 5 \cdot 3 = 105$ different configurations of the computer.

Like the Sum Rule, the Product Rule should be reasonably intuitive: if we are choosing a pair $\langle a, b\rangle$ from $A \times B$, then we have $|A|$ different choices of the first component a—and, for each of those $|A|$ choices, we have $|B|$ choices for the second component b. (Thinking of A as $A = \{a_1, a_2, \ldots, a_{|A|}\}$, we can even view $\{\langle a, b\rangle : a \in A, b \in B\}$ as

$$\{\langle a_1, b\rangle : b \in B\} \cup \{\langle a_2, b\rangle : b \in B\} \cup \quad \cdots \quad \cup \{\langle a_{|A|}, b\rangle : b \in B\}.$$

By the Sum Rule, this set has cardinality $|B| + |B| + \cdots + |B|$, with one term for each element of A—in other words, it has cardinality $|A| \cdot |B|$.) Here are a few more examples:

Example 9.6 (32-bit strings)

Problem: How many different 32-bit strings are there?

Solution: The set of 32-bit strings is $\{0,1\}^{32}$—that is, elements of

$$\underbrace{\{0,1\} \times \{0,1\} \times \{0,1\} \times \cdots \times \{0,1\}}_{32 \text{ times}}.$$

Because $|\{0,1\}| = 2$, the Product Rule lets us conclude that $|\{0,1\}^{32}|$ is

$$\underbrace{2 \cdot 2 \cdot 2 \cdot \quad \cdots \quad \cdot 2}_{32 \text{ times}} = 2^{32}.$$

(We can use the same type of analysis to show that there are $2^4 = 16$ strings of 4 bits; for concreteness, they're all listed in Figure 9.2.)

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

Figure 9.2: The set of all 4-bit strings.

Example 9.7 (Number of possible shortened URLs)

A *URL-shortening service* like `bit.ly` or `snipurl.com` allows a user to compress a long URL into a much shorter sequence of characters. (The shorter URL can then be used in emails or tweets or other contexts in which a long URL is unwieldy.) For example, by entering the URL of Alan Turing's Wikipedia page into `bit.ly`, I got the URL

`http://bit.ly/1o6HPM` as a shortened form of `http://en.wikipedia.org/wiki/Alan_Turing`.

If a shortened URL consists of 6 characters, each of which is a digit, lowercase letter, or uppercase letter, the number of possible shortened URLs is, using the Product Rule,

$$|C \times C \times C \times C \times C \times C| = |C| \cdot |C| \cdot |C| \cdot |C| \cdot |C| \cdot |C| = |C|^6,$$

where $C = \{\texttt{0}, \ldots, \texttt{9}\} \cup \{\texttt{a}, \ldots, \texttt{z}\} \cup \{\texttt{A}, \ldots, \texttt{Z}\}$ is the set of possible characters. Because $|C| = 10 + 26 + 26 = 62$ via the Sum Rule, we know that there are $62^6 = 56{,}800{,}235{,}584$ possible shortened 6-character URLs.

Taking it further: The point of a URL-shortening service is to translate long URLs into short ones, but it's theoretically impossible for *every* URL to be shortened by this service: there are more possible URLs of length k than there are URLs of length strictly less than k. A similar issue arises with *file compression* algorithms, like ZIP, that try to reduce the space required to store a file. See the discussion on p. 938.

Product Rule: counting sequences from a fixed set

This use of the Product Rule—to count the number of sequences of length k with elements all drawn from a fixed set S, rather than having a different set of options for each component—is common enough that we'll note it as a separate rule:

Theorem 9.3 (Product Rule: sequences of elements from a single set S) *For any set S and any $k \in \mathbb{Z}^{\geq 1}$, the number of k-tuples from the set $S^k = \underbrace{S \times S \times \cdots \times S}_{k \text{ times}}$ is $|S^k| = |S|^k$.*

A notational reminder regarding Theorem 9.3: S^k is the set

$$S \times S \times \cdots \times S,$$

that is, the set of k-tuples where each component is an element of S. On the other hand, $|S|^k$ is the number $|S|$ raised to the kth power.

Here's another example using this special case of the Product Rule:

Example 9.8 (MAC addresses)

Problem: A *media access control address*, or *MAC address*, is a unique identifier for a network adapter, like an ethernet card or wireless card. A MAC address consists of a sequence of six groups of pairs of hexadecimal digits. (A *hexadecimal digit* is one of 0123456789ABCDEF.) For example, F7:DE:F1:B6:A4:38 is a MAC address. (The pairs of digits are traditionally separated by colons when written down.) How many different MAC addresses are there?

Solution: There are 16 different hexadecimal digits. Thus, using the Product Rule, there are $16 \cdot 16 = 256$ different pairs of hexadecimal digits, ranging from 00 to *FF*. Using the Product Rule again, as in Example 9.7, we see that there are 256^6 different sequences of six pairs of hexadecimal digits. Thus there are $256^6 = [16^2]^6 = [(2^4)^2]^6 = 2^{48} = 281{,}474{,}976{,}710{,}656$ total different MAC addresses.

Taking it further: In addition to the numerical addresses assigned to particular hardware devices—the MAC addresses from Example 9.8—each device that's connected to the internet is also assigned an address, akin to a mailing address, that's used to identify the destination of a packet of information. But we've had to make a major change to the way that information is transmitted across the internet because of a counting problem: we've run out of addresses! See the discussion on p. 919.

9.2.2 *Inclusion–Exclusion: Unions of Nondisjoint Sets*

The counting techniques that we've introduced so far have some important restrictions. We can only use the Sum Rule to calculate $|A \cup B|$ when A and B are *disjoint*. And we are only able to use the Product Rule to calculate the number of sequences when the set of options for the second component does not depend on the choice that we made in the first component. In the remainder of this section, we will extend our techniques to remove these restrictions so that we can handle more general problems. Let's start with a specific example of the cardinality of the union of nondisjoint sets:

Example 9.9 (Primes and odds)
Consider the set $O = \{1,3,5,7,9\}$ of odd numbers less than 10 and the set $P = \{2,3,5,7\}$ of prime numbers less than 10. What is $|O \cup P|$?

It might be tempting to use the Sum Rule to conclude that $|O \cup P| = |O| + |P| = 5 + 4 = 9$. But this conclusion is incorrect, because $P \cap O = \{3,5,7\} \neq \varnothing$, so the Sum Rule doesn't apply. In particular, $O \cup P = \{1,2,3,5,7,9\}$, so $|O \cup P| = 6$.

The issue with the naïve application of the Sum Rule in Example 9.9 is called *double counting*: in the expression $|O| + |P|$, we counted the elements in the intersection $O \cap P$ twice, which gave us the incorrect total count. The idea underlying the Inclusion–Exclusion Rule is to correct for this error: to compute the size of the union of two sets A and B, we extend the Sum Rule to correct for the double counting by subtracting $|A \cap B|$ from the final result. (See Figure 9.3.) This counting rule is called inclusion–exclusion because we *include* (add) the cardinalities of the two individual sets, and then *exclude* (subtract) the cardinality of the intersection of the pairs:

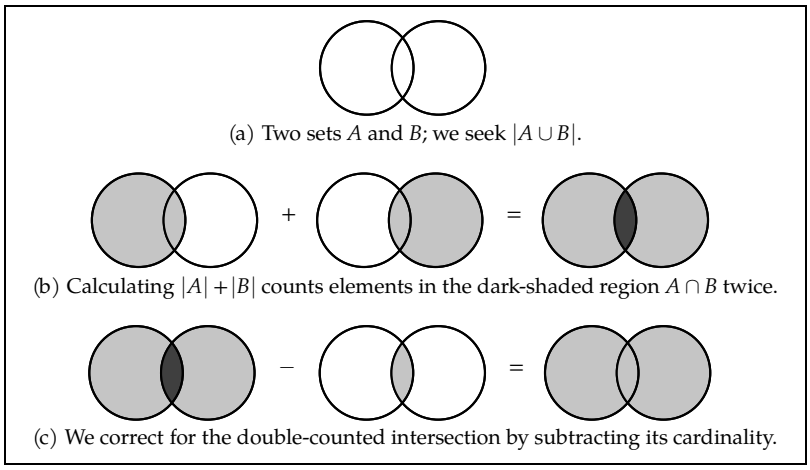


Figure 9.3: The Inclusion–Exclusion Rule.

Problem-solving tip: Sometimes the easiest way to solve a problem—in CS or in life!—is to find an imperfect approximation to the solution, and then correct for whatever inaccuracies result. Inclusion–Exclusion is a good example of this estimate-and-fix strategy.

Theorem 9.4 (Inclusion–Exclusion)
Let A and B be sets. Then $|A \cup B| = |A| + |B| - |A \cap B|$.

Here are a few small examples:

Example 9.10 (Counting not necessarily disjoint unions)

- Let $A := \{1,2,3\}$ and $B := \{3,4,5,6\}$. Thus $A \cap B = \{3\}$, and so $|A| = 3$ and $|B| = 4$ and $|A \cap B| = 1$. By the inclusion–exclusion rule, $|A \cup B| = |A| + |B| - |A \cap B| = 3 + 4 - 1 = 6$. Indeed, we have $A \cup B = \{1,2,3,4,5,6\}$, which contains 6 elements.

- At a certain school in the midwest, there are 119 computer science majors and 65 math majors. There are 7 students double majoring in CS and math. Thus a total of $119+65-7=177$ different students are majoring in either of the two fields.
- There are 21 consonants (`BCDFGHJKLMNPQRSTVWXYZ`) in English. There are 6 vowels in English (`AEIOUY`). There is one letter that's both a vowel and a consonant (`Y`). Thus there are $21+6-1=26$ total letters.
- Let E be the set of even integers between 1 and 100. Let O be the set of odd integers between 1 and 100. Note that $|E| = 50$, $|O| = 50$, and $|E \cap O| = 0$. Thus $|E \cup O| = 50+50-0 = 100$.

Here's an example that uses Inclusion–Exclusion to compute the cardinality of a slightly more complicated set:

Example 9.11 (ATM machine PIN numbers)
Problem: A certain bank's customers can select a 4-digit number (called a *PIN*) to access their accounts, but the bank insists that the PIN may not start with the same digit repeated three times (for example, 7770) or end with the same digit repeated three times (for example, 0111). How many *invalid* PINs are there?

Solution: Let S denote the set of PINs that start with three repeated digits. Let E denote the set of PINs that end with three repeated digits. Then the set of invalid PINs is $S \cup E$.

- Note that $|S| = 100$: we can view a PIN in S as a sequence of two digits $\langle x, y\rangle \in \{0, 1, \ldots, 9\}^2$, with x repeated three times in the PIN. (So $\langle 3, 1\rangle$ corresponds to the PIN 3331.) By the Product Rule, there are $10^2 = 100$ such codes.
- Similarly, $|E| = 100$: we can think of an element of E as a sequence of two digits $\langle x, y\rangle \in \{0, 1, \ldots, 9\}^2$, where y is repeated three times in the PIN.

If $S \cap E$ were empty, then we could apply the Sum Rule to compute $|S \cup E|$. But there *are* PINs that are in both S and E:

- A 4-digit number $\langle x, y, z, w\rangle$ is in $S \cap E$ if and only if $x = y = z$ (because $\langle x, y, z, w\rangle \in S$) *and* $y = z = w$ (because $\langle x, y, z, w\rangle \in E$). That is, any 4-digit number that consists of the same digit repeated four times is in $S \cap E$. Thus

$$S \cap E = \{0000, 1111, 2222, 3333, 4444, 5555, 6666, 7777, 8888, 9999\},$$

and $|S \cap E| = 10$.

(See Figure 9.4 for S, E, and $S \cap E$.) Applying the Inclusion–Exclusion rule, we see that the set $S \cup E$ of invalid PINs has cardinality $|S|+|E|-|S \cap E| = 100+100-10 = 190$. (So $10{,}000 - 190 = 9810$ PINs are valid.)

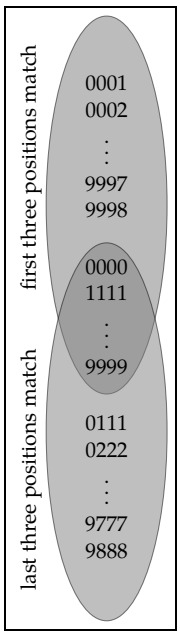


Figure 9.4: Invalid PINs, starting or ending with the same digit repeated three times.

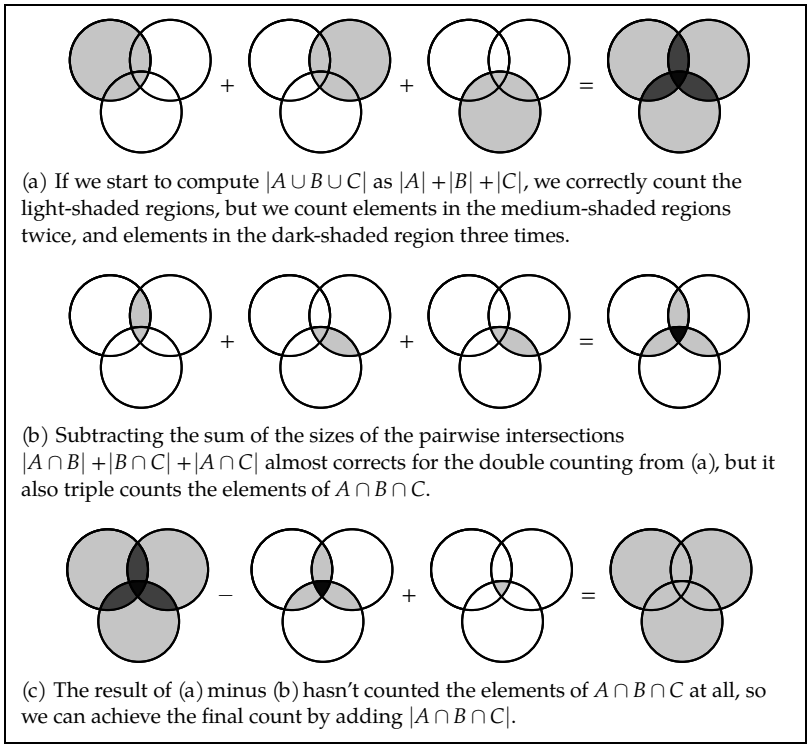
The basic Sum Rule is actually a special case of the Inclusion–Exclusion Rule: if A and B are disjoint, then $|A \cap B| = \varnothing$, so $|A \cup B| = |A|+|B|-|A \cap B| = |A|+|B|-0 = |A|+|B|$.

Inclusion–Exclusion for three sets

Theorem 9.4 describes how to calculate the cardinality of the union of *two* sets, but this idea can be generalized. The basic idea is simple: we will try counting in the easiest way possible, and then we'll correct for any overcounting or undercounting.

For example, we can compute the cardinality of the union of *three* sets $A \cup B \cup C$ using a more complicated version of Inclusion–Exclusion:

- We add (include) the three singleton sets ($|A| + |B| + |C|$), but this sum counts any element contained in more than one of the three sets more than once.
- So we subtract (exclude) the three pairwise intersections ($|A \cap B| + |A \cap C| + |B \cap C|$) from the sum. But we're not done: imagine an element contained in *all three* of A, B, and C; such an element was included three times and then excluded three times, so it hasn't been counted at all.
- So we add (include) the three-way intersection $|A \cap B \cap C|$.



(a) If we start to compute $|A \cup B \cup C|$ as $|A| + |B| + |C|$, we correctly count the light-shaded regions, but we count elements in the medium-shaded regions twice, and elements in the dark-shaded region three times.

(b) Subtracting the sum of the sizes of the pairwise intersections $|A \cap B| + |B \cap C| + |A \cap C|$ almost corrects for the double counting from (a), but it also triple counts the elements of $A \cap B \cap C$.

(c) The result of (a) minus (b) hasn't counted the elements of $A \cap B \cap C$ at all, so we can achieve the final count by adding $|A \cap B \cap C|$.

Figure 9.5: The Inclusion–Exclusion Rule for three sets A, B, and C. See Theorem 9.5.

This calculation yields the following three-set rule for inclusion–exclusion. (Or see Figure 9.5 for a visual illustration of why this calculation is correct.)

Theorem 9.5 (Inclusion–Exclusion for three sets)
Let A, B, and C be sets. Then $|A \cup B \cup C|$ is given by

$$|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|.$$

Here are a couple of small examples of the three-set version of inclusion–exclusion:

Example 9.12 (Counting three-set unions)

- Let $A := \{0, 1, 2, 3, 4\}$ and $B := \{0, 2, 4, 6\}$ and $C := \{0, 3, 6\}$. Then

$$\begin{aligned} &|A \cup B \cup C| \\ &= \underbrace{5}_{|A|} + \underbrace{4}_{|B|} + \underbrace{3}_{|C|} - \underbrace{3}_{|A\cap B|=|\{0,2,4\}|} - \underbrace{2}_{|A\cap C|=|\{0,3\}|} - \underbrace{2}_{|B\cap C|=|\{0,6\}|} + \underbrace{1}_{|A\cap B\cap C|=|\{0\}|} \\ &= 12 - 7 + 1 = 6, \end{aligned}$$

by Inclusion–Exclusion. Indeed, $A \cup B \cup C = \{0, 1, 2, 3, 4, 6\}$. (See Figure 9.6.)

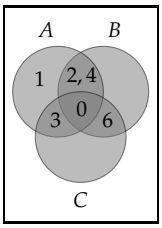


Figure 9.6: Some small sets.

- Consider the words ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, and EIGHT. Let E be the set of these words containing at least one E, let T be the words containing a T, and let R be the words containing an R. Then

$$\begin{aligned}
&|E \cup T \cup R| \\
&= \underbrace{5}_{E=\{\text{ONE, THREE, FIVE, SEVEN, EIGHT}\}} + \underbrace{3}_{T=\{\text{TWO, THREE, EIGHT}\}} + \underbrace{2}_{R=\{\text{THREE, FOUR}\}} - \underbrace{2}_{E\cap T=\{\text{THREE, EIGHT}\}} - \underbrace{1}_{E\cap R=\{\text{THREE}\}} - \underbrace{1}_{T\cap R=\{\text{THREE}\}} + \underbrace{1}_{E\cap T\cap R=\{\text{THREE}\}} \\
&= 7,
\end{aligned}$$

  and, indeed, seven of the eight words are in $E \cup T \cup R$ (the only one missing is SIX).

We'll close with a slightly bigger example, about integers divisible by 2, 3, or 5:

Example 9.13 (Divisibility)
Problem: How many integers between 1 and 1000, inclusive, are evenly divisible by any of 2, 3, or 5?

Solution: Define the following sets:

$$\begin{aligned}
A &= \{n \in \{1, \ldots, 1000\} : 2 \mid n\} \\
B &= \{n \in \{1, \ldots, 1000\} : 3 \mid n\} \\
C &= \{n \in \{1, \ldots, 1000\} : 5 \mid n\}.
\end{aligned}$$

We must compute $|A \cup B \cup C|$.

- It's fairly easy to see that $|A| = 500$, $|B| = 333$, and $|C| = 200$, because $A = \{2n : 1 \le n \le 500\}$, $B = \{3n : 1 \le n \le 333\}$, and $C = \{5n : 1 \le n \le 200\}$.
- Observe that $A \cap B$ is the set of integers between 1 and 1000 that are divisible by both 2 and 3—that is, the set of integers divisible by 6. By the same logic that we used to compute $|A|$, $|B|$, and $|C|$, we see
  - $|A \cap B| = |\{6n : 1 \le n \le 166\}| = 166$,
  - $|A \cap C| = |\{10n : 1 \le n \le 100\}| = 100$, and
  - $|B \cap C| = |\{15n : 1 \le n \le 66\}| = 66$.
- And, using the same approach, we can conclude that $A \cap B \cap C = \{n : 30 \mid n\} = \{30n : 1 \le n \le 33\}$, so $|A \cap B \cap C| = 33$.

Therefore, using the Inclusion–Exclusion Rule, $|A \cup B \cup C|$ is

$$\underbrace{500}_{|A|} + \underbrace{333}_{|B|} + \underbrace{200}_{|C|} - \underbrace{166}_{|A\cap B|} - \underbrace{100}_{|A\cap C|} - \underbrace{66}_{|B\cap C|} + \underbrace{33}_{|A\cap B\cap C|} = 734.$$

Problem-solving tip: To verify a calculation like this one, it's a good idea (and very easy!) to write a short program.

We can further generalize the inclusion–exclusion principle to calculate the cardinality of the union of an arbitrary number of sets. (See Exercises 9.30 and 9.181.)

9.2.3 *The Generalized Product Rule*

The Product Rule (Theorem 9.2) tells us how to compute the number of 2-element sequences where the first element is drawn from the set A and the second from the set B—specifically, it says that $|A \times B|$ is $|A| \cdot |B|$. But there are many types of sequences that do not precisely fit this setting: the Product Rule only describes the set of sequences where each component is selected from a *fixed* set of options. If the set of options for choice #2 depends on choice #1, then we cannot directly apply the Product Rule. However, the basic principle of the Product Rule still applies if the *number* of different choices for the second component is the same regardless of the choice of the first component, even if the *particular set of choices* can differ:

Theorem 9.6 (Generalized Product Rule)
Let S denote a set of sequences, each of length k, where for each index $i \in \{1, \ldots, k\}$ the following condition holds: for each choice of the first $i-1$ components of the sequence, there are exactly n_i choices for the ith component. Then $|S| = \prod_{i=1}^{k} n_i$.

Here are a few examples using the Generalized Product Rule:

Example 9.14 (Gold, silver, and bronze)
Problem: A set S of eight sprinters qualify for the finals of the 100-meter dash in the Olympics. One will win the gold medal, another the silver, and a third the bronze. How many different trios of medalists are possible?

Solution: It "feels" like we can solve this problem using the Product Rule, by choosing a sequence of three elements from S, where we forbid duplication in our choices. But our choice of gold, silver, and bronze medalists would be from

$$S \times \big(S - \{\text{the gold medalist}\}\big) \times \big(S - \{\text{the gold and silver medalists}\}\big)$$

and the Product Rule doesn't permit the set of choices for the second component to depend on the first choice, or the options for the third choice to depend on the first two choices.

Instead, observe that there are 8 choices for the gold medalist. For each of those choices, there are 7 choices for the silver medalist. For each of these pairs of gold and silver medalists, there are 6 choices for the bronze medalist. Thus, by the Generalized Product Rule, the total number of trios of medalists is $8 \cdot 7 \cdot 6 = 336$.

Example 9.15 (Opening moves in a chess game)
In White's very first move in a chess game, there are $n_1 = 10$ pieces that can move: any of White's 8 pawns or 2 knights. Each of these pieces has $n_2 = 2$ legal moves: the pawns can move forward either 1 or 2 squares, and the knights can move either ↰ or ↱. (See Figure 9.7.) Thus there are $n_1 \cdot n_2 = 10 \cdot 2 = 20$ legal first moves.

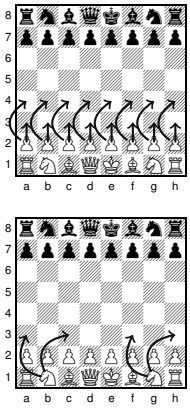


Figure 9.7: The valid first moves in a chess game.

Example 9.16 (Students in classes)
At a certain school in the midwest, each of 2023 students enrolls in exactly 3 classes per term. The set

$$\textit{Enrollments} := \{\langle s, c\rangle : s \text{ is a student enrolled in class } c \text{ during the current term}\}$$

has cardinality $2023 \cdot 3 = 6069$, by the Generalized Product Rule: for each of the $n_1 = 2023$ choices of student, there are $n_2 = 3$ choices of classes. (Note that the original Product Rule does not apply, because the set *Enrollments* is not a Cartesian product: in general, two students are not enrolled in the same *classes*—just the same *number* of classes.)

Although we didn't say we were doing so, we actually used the underlying idea of the Generalized Product Rule in Example 9.11. Let's make its use explicit here:

Example 9.17 (4-digit PINs starting with a triplicated digit)
Let $S \subseteq \{0, 1, \ldots, 9\}^4$ denote the set of 4-digit PINs that start with three repeated digits. We claim that $|S| = 100$, as follows:

- There are $n_1 = 10$ choices for the first digit.
- There is only $n_2 = 1$ choice for the second digit: it must match the first digit.
- There's also only $n_3 = 1$ choice for the third digit: it must match the first two.
- There are $n_4 = 10$ choices for the fourth digit.

Thus there are $n_1 \cdot n_2 \cdot n_3 \cdot n_4 = 10 \cdot 1 \cdot 1 \cdot 10 = 100$ elements of S.

Permutations

The Generalized Product Rule sheds some light on a concept that arises in a wide range of contexts: a *permutation* of a set S, which is any ordering of the elements of S.

Definition 9.1 (Permutation)
A permutation *of a set S is a sequence of elements from S that is of length $|S|$ and contains no repetitions. In other words, a permutation of S is an ordering of the elements of S.*

As a first example, let's list all the permutations of the set $\{1, 2, \ldots, n\}$ for a few small values of n:

- for $n = 1$, there's just one ordering: $\langle 1\rangle$.
- for $n = 2$, there are two orderings: $\langle 1, 2\rangle$ and $\langle 2, 1\rangle$.
- for $n = 3$, there are six: $\langle 1, 2, 3\rangle$, $\langle 1, 3, 2\rangle$, $\langle 2, 1, 3\rangle$, $\langle 2, 3, 1\rangle$, $\langle 3, 1, 2\rangle$, and $\langle 3, 2, 1\rangle$.
- for $n = 4$, there are twenty-four: six with 1 as the first element (which can then be followed by any of the six permutations of $\langle 2, 3, 4\rangle$), six with 2 as the first element, six with 3 first, and six with 4 first, yielding a total of $4 \cdot 6 = 24$ orderings.

How many permutations of an n-element set are there? There are several ways to see the general pattern, including recursively, but it may be easiest to use the Generalized Product Rule to count the number of permutations:

Theorem 9.7 (Number of permutations)
Let S be any set, and write $n := |S|$. The number of different permutations of S is $n!$.

Proof. There are n choices for the first element of a permutation of S. For the second element, there are $n-1$ choices (all but the element chosen first). There are $n-2$ choices for the third slot (all but the elements chosen first and second). In general, for the ith element, there are $n-i+1$ choices. Thus the number of permutations of S is

$$\prod_{i=1}^{n}(n-i+1) = \prod_{j=1}^{n} j = n!$$

by the Generalized Product Rule. □

Here's a small example for a concrete set S:

Example 9.18 (10-digit numbers)
Problem: What fraction of integers between 0 and 9,999,999,999 (all written as 10-digit numbers, including any leading zeros) have no repeated digits?

Solution: We seek a 10-digit sequence with no repetitions—that is, a permutation of $\{0, 1, \ldots, 9\}$. There are $10! = 3{,}628{,}800$ such permutations, by Theorem 9.7. There are a total of 10^{10} integers between 0 and 9,999,999,999, by the Product Rule. Thus the fraction of these integers with no repeated digits is $\frac{10!}{10^{10}} \approx 0.00036\cdots$, about one out of every 2750 integers in this range.

Taking it further: A permutation of a set S is an ordering of that set S—so thinking about permutations is closely related to thinking about *sorting algorithms* that put an out-of-order array into a specified order. By using the counting techniques of this section, we can prove that algorithms *must* take a certain amount of time to sort; see the discussion on p. 920.

We will also return to permutations frequently later in the chapter. For example, in Section 9.4, we will address counting questions like the following: *how many different 13-card hands can be drawn from a standard 52-card deck of playing cards?* (Here's one way to think about it: we can lay out the 52 cards in any order—any permutation of the cards—and then pick the first 13 of them as a hand. We'll have to correct for the fact that any ordering of the first 13 cards—and, for that matter, any ordering of the last 39—will count as the same hand. But permutations will also help us to think about this correction!)

9.2.4 Combining Products and Sums

Suppose that we select a pair $\langle a, b\rangle$ from a set of possible choices. The Product Rule tells us how many ways to make these choices if the particular choice of a does not affect the set of options from which b is chosen. The Generalized Product Rule tells us how many ways to make these choices if the particular choice of a does not affect *the size of* the set of options from which b is chosen. But if the *number* of options for the

choice of b differs based on the choice of a, even the Generalized Product Rule does not apply. In this case, we can use a combination of the Sum Rule and the Generalized Product Rule to calculate the number of results. We'll close this section with a few examples of these somewhat more complex counting questions.

Example 9.19 (Ordering coffee)
A certain coffeeshop sells the following espresso-based drinks:

americano*, cappuccino, espresso*, latte, macchiato, mocha.

The drinks marked with an asterisk do not contain milk; the others do. All drinks can be made with either decaf or regular espresso. All milk-containing drinks can be made with any of {soy, skim, 2%, whole} milk. How many different drinks are sold by this coffeeshop?

We can think of a chosen drink as a sequence of the form

$$\langle \text{drink type}, \text{milk type (or "none")}, \text{espresso type} \rangle.$$

There are $4 \cdot 4 \cdot 2 = 32$ choices of milk-based drinks (4 drink types, 4 milk types, and 2 espresso types). There are $2 \cdot 1 \cdot 2 = 4$ choices of non-milk-based drinks (2 drink types, 1 "milk" type ["none"], and 2 espresso types). Thus the total number of different drinks sold by this coffeeshop is $32 + 4 = 36$.

Example 9.20 (Text numbers)
Problem: In the United States, a text message can be sent either to a regular 10-digit phone number, or to a so-called *short code* which is a 5- or 6-digit number. Neither a phone number nor a short code can start with a 0 or a 1. How many different textable numbers are there in the United States?

Solution: Let $D = \{2, 3, \ldots, 9\}$. Note $|D| = 8$. The set of valid textable numbers is:

$$\underbrace{D \times (D \cup \{0,1\})^9}_{\text{phone numbers}} \cup \underbrace{D \times (D \cup \{0,1\})^4}_{\text{5-digit short codes}} \cup \underbrace{D \times (D \cup \{0,1\})^5}_{\text{6-digit short codes}}.$$

The Product Rule tells us that $|D \times (D \cup \{0,1\})^j| = |D| \cdot |D \cup \{0,1\}|^i = 8 \cdot 10^i$ for any i. (To be totally pedantic: we're using the Sum Rule to conclude that $|D \cup \{0,1\}| = |D| + |\{0,1\}| = 10$, because D and $\{0,1\}$ are disjoint.) Therefore:

$$\begin{aligned}
&\left|D \times (D \cup \{0,1\})^9 \cup D \times (D \cup \{0,1\})^4 \cup D \times (D \cup \{0,1\})^5\right| \\
&= \left|D \times (D \cup \{0,1\})^9\right| + \left|D \times (D \cup \{0,1\})^4\right| + \left|D \times (D \cup \{0,1\})^5\right| && \textit{Sum Rule: the three types of numbers are disjoint because they have different lengths} \\
&= 8 \cdot 10^9 + 8 \cdot 10^4 + 8 \cdot 10^5 && \textit{Product Rule, as described in the previous paragraph} \\
&= 8{,}000{,}880{,}000.
\end{aligned}$$

Problem-solving tip: When you're confronted with a counting problem that appears complicated, try to find a nice way of splitting the problem into several disjoint options. Often a difficult counting problem is actually the sum of two simple counting problems.

Combining sums and products: prefix-free codes

We'll end the section with two somewhat more complicated counting problems, where we're asked to calculate the number of objects meeting some particular condition: sets of bitstrings such that no string is a prefix of another, and results of a best-of-five series of games. In both cases, we can give a solution based entirely on a brute-force approach by simply enumerating all possible sequences, eliminating any that don't meet the stated condition, and counting the uneliminated sequences one by one. But there are also ways to break down the set of objects of interest into subsets that we can count using the Sum and (Generalized) Product Rules.

A *prefix-free code* is a set C of bitstrings with the property that no $x \in C$ is a prefix of any other $y \in C$. (For example, if $010 \in C$, then we must have $0101 \notin C$, because 010 is a prefix of $\underline{010}1$.) Let's compute the number of prefix-free codes where all of the codewords are only 1 or 2 bits long:

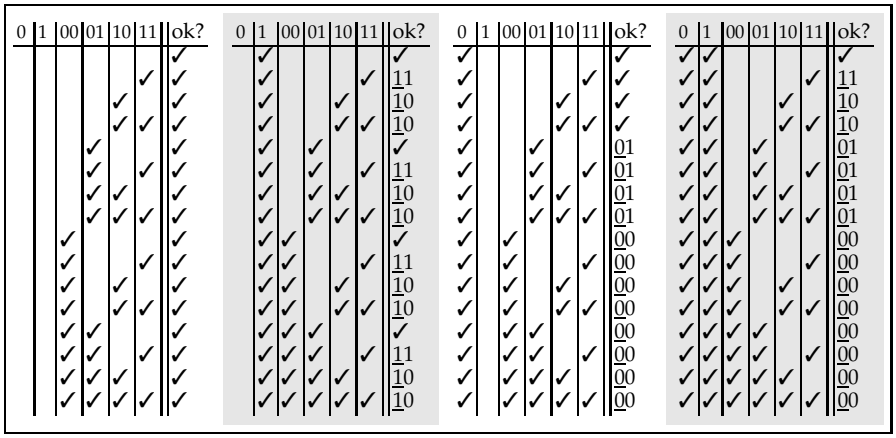


Figure 9.8: All 64 subsets of $\{0, 1, 00, 01, 10, 11\}$, with indication of whether the subset is prefix-free or not. In each row (a subset), if the set is not prefix-free, then one violation found in the set is listed.

Example 9.21 (Prefix-free codes)
One simple way to find the number of prefix-free codes $C \subseteq \{0,1\}^1 \cup \{0,1\}^2$ is to write down all subsets of $S := \{0,1\}^1 \cup \{0,1\}^2$, and then check each subset to eliminate any set that violates the prefix rule. (See Figure 9.8, which was generated by a computer program; there are 25 codes in the table that pass the prefix test.) There are $2^{|S|} = 2^6 = 64$ subsets of S: we can describe each subset of S as an element of $\{\text{yes}, \text{no}\}^{|S|}$ where the ith component tells us whether the ith element of S is in the set. The Product Rule tells that $|\{\text{yes}, \text{no}\}^{|S|}| = 2^6 = 64$. (See Lemma 9.10.)

Here's a different approach, involving more thinking and less brute-force calculation. Let's partition the set of valid codes into four classes based on whether $0 \in C$ and $1 \in C$:

- If $0 \notin C$ and $1 \notin C$, then any subset of $\{00, 01, 10, 11\}$ can be in C.
- If $0 \notin C$ and $1 \in C$, then any subset of $\{00, 01\}$ can also be in C.
- If $0 \in C$ and $1 \notin C$, then any subset of $\{10, 11\}$ can also be in C.
- If $0 \in C$ and $1 \in C$, then no 2-bit strings can be included.

By the Product Rule, there are, respectively, 2^4 and 2^2 and 2^2 and 2^0 choices corresponding to these classes. (The four classes correspond to the four columns of Figure 9.8.) By the Sum Rule, the total number of prefix-free codes using 1- and 2-bit strings is $16 + 4 + 4 + 1 = 25$.

Taking it further: Prefix-free codes are useful in that they can be transmitted unambiguously, without a special marker that separates codewords. For example, consider the prefix-free code $\{0, 10, 11\}$. Then a sequence 0101111100 can only be interpreted as $0\|10\|11\|11\|10\|0$. If a code is not prefix-free—like the English language!—then a sequence of codewords cannot be unambiguously decoded: for example, `THEME` might be one word (*theme*) or it might be two (*the me*).

Huffman coding—named after David Huffman, a 20th-century American computer scientist—is an algorithm for computing a prefix-free code that can be used for data compression for English (for example), by allowing us to translate each letter into a corresponding code word. Huffman coding carefully assigns shorter codewords to more commonly used letters, and thus has a special property: among all prefix-free codes, its codewords have the smallest length, on average. A Huffman code can be constructed using a simple greedy approach; for more, see a good textbook on algorithms.

Combining sums and products: a best-of-five series

Here's one more example of using our counting rules in combination:

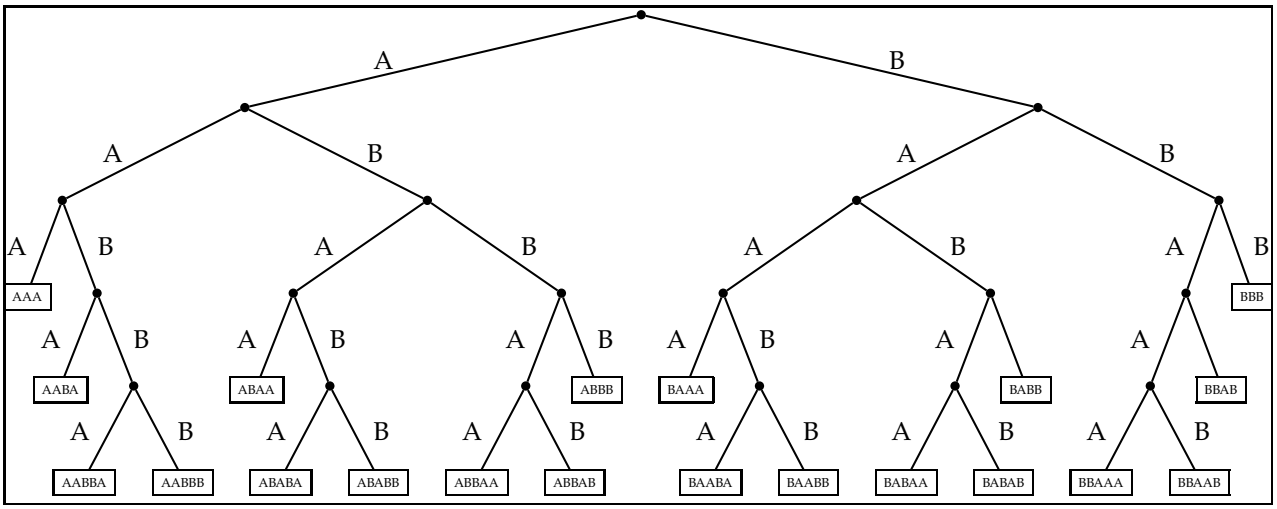


Figure 9.9: A tree representing each best-of-five series of games between two teams, A and B. The branch points correspond to the games, and are labeled by the winner of the game. The 20 different sequences of outcomes are shown at the bottom of the tree.

Example 9.22 (A best-of-five series)

Problem: Suppose that two teams A and B play a best-of-five series of games: the teams play until one team has won three games, at which point the match is over, and that team is the winner. How many different sequences of outcomes are there?

Solution: The simplest approach is to use brute force: simply write out all possible sequences of outcomes, and count them up. This approach is shown in Figure 9.9. However, there's another way to count. Suppose that team A wins the series:

- There's 1 outcome in which A never loses: A wins games 1, 2, and 3.
- There are 3 outcomes in which A loses once: A loses immediately before its first win ($BAAA$), before its second win ($ABAA$), or before its third win ($AABA$).
- If A loses twice, then A must have won the fifth game, and exactly two of the first four. Thinking of the outcomes of the first four games as 4-bit strings with 1s denoting A's wins, Example 9.4 says there are precisely 6 such outcomes.

In sum, there are $1 + 3 + 6 = 10$ ways for A to win the series. There are 10 analogous ways for B to win, so there are 20 outcomes in total.

Computer Science Connections

Running out of IP addresses, and IPv6

A crucial component of the internet is the assignment of an *address* to every machine connected to the network. This address is called an *IP address*, where "IP" stands for *Internet Protocol*—the algorithm by which packets of information are handled while they're being transmitted across the internet. Each packet of information to be transmitted stores a variety of pieces of information, including (1) some basic header information; (2) a *source address* (the sender of the information); (3) a *destination address* (the intended recipient of the information); and (4) the data to be transmitted (the "payload").

The subfield of computer science called *computer networking* is devoted to everything about how the internet (or some smaller network) works: design of the network, physical systems, protocols for routing, and more.[1] Here we are going to concentrate on the *IP address itself*, and a particular issue related to how many—or how few!—addresses there are.

For more, see a good textbook on computer networks, like

[1] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach*. Addison–Wesley, 6th edition, 2013.

Each device on the internet that can send or receive information needs an address by which to do so. For almost the entire history of the internet, an IP address has simply been a 32-bit string. These IP addresses are typically represented as an element of $\{0,\ldots,255\}^4$ instead of as an element of $\{0,1\}^{32}$, by converting 8 bits at a time into base-10 numbers, and then writing each 8-bit chunk separated by periods. For example, the site `cs.carleton.edu` is associated with the IP address

$$\underbrace{10001001}_{137} . \underbrace{00010110}_{22} . \underbrace{00000100}_{4} . \underbrace{00010111}_{23}.$$

You can find the IP address of your favorite site using a tool called `nslookup` on most machines, which checks a so-called *name server* to translate a site's name (like `whitehouse.gov`) into an IP address (like `173.223.132.110`).

As an easy counting problem, we can check that there only $2^{32} = 4{,}294{,}967{,}296$ different possible 32-bit IP addresses—about 4.3 billion addresses. Every machine connected to the internet needs to be addressable to receive data, so that means that we can only support about 4.3 billion connected devices. In the 1990s and 2000s, more and more people began to have machines connected to the internet, and each person also began to have more and more devices that they wanted to connect. It became clear that we were facing a dire shortage of IP addresses! As such, a new version of the Internet Protocol (version six, hence called *IPv6*) has been introduced.

There are some strategies from computer networking for conserving addresses by "translation," so that several computers $c_1, c_2, \ldots$ can be connected via an access point p—where p is the only machine that has a public, visible IP address. All of those computers' traffic is handled by p, but p must be able to reroute the traffic it receives to the correct one of the c_i computers. For more information, see the Kurose–Ross textbook cited previously.

In IPv6, instead of using 32-bit addresses, we now use 128-bit addresses. There are some tricky elements to the transition from 32-bit to 128-bit addresses—your computer better keep working!—but there are now 2^{128} different addresses available. That's $340{,}282{,}366{,}920{,}938{,}463{,}463{,}374{,}607{,}431{,}768{,}211{,}456 \approx 3.4 \times 10^{38}$, which should hold us for a few millennia. For example, `whitehouse.gov` is associated with a 32-bit address `173.223.132.110`, *and* a 128-bit address `2600:1408:0010:019a:0fc4`, represented by 5 blocks of 4 hexadecimal numbers—that is, as an element of

$$\Big[\{\texttt{0},\texttt{1},\texttt{2},\texttt{3},\texttt{4},\texttt{5},\texttt{6},\texttt{7},\texttt{8},\texttt{9},\texttt{a},\texttt{b},\texttt{c},\texttt{d},\texttt{e},\texttt{f}\}^4\Big]^5.$$

Computer Science Connections

A Lower Bound for Comparison-Based Sorting

Most people who encounter the *sorting* problem—given an array $A[1 \ldots n]$, rearrange A so that it's in ascending order—initially devise a quadratic-time algorithm. (For simplicity, suppose that we're sorting *distinct* elements.) The most common examples of $\Theta(n^2)$-time algorithms are Selection Sort, Insertion Sort, and Bubble Sort. Then, after a lot of thought (and, usually, some help), those people often are able to devise a $O(n \log n)$-time sorting algorithm, like Merge Sort, Quick Sort, or Heap Sort. (See Section 6.3.)

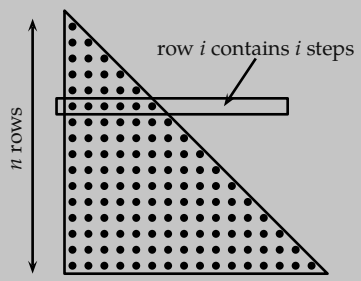
But suppose that you were extra impatient with the speed of your sorting algorithm, and you were extra, extra clever. Could you do asymptotically better than $O(n \log n)$ in the worst case? The answer, we'll show, is no—with a footnote: any "comparison-based" sorting algorithm requires $\Omega(n \log n)$ time. (The footnote is that it depends on what we mean by "sort," as we'll see.)

```
selectionSort(A[1...n]):
 1: for i := 1 to n:
 2:     minIndex := i
 3:     for j := i + 1 to n:
 4:         if A[j] < A[minIndex] then
 5:             minIndex := j
 6:     swap A[i] and A[minIndex]
```

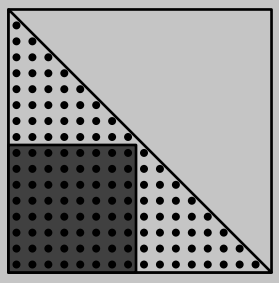
Figure 9.10: Selection Sort.

A Warm-up: Selection Sort

First, recall Selection Sort, shown in Figure 9.10. One way to analyze its running time is as we did in Example 6.7: there are n iterations, and in the $(n - i)$th iteration we require i steps. In other words, the running time of Selection Sort is $\sum_{i=1}^{n} i$. We could repeat the straightforward inductive proof that $\sum_{i=1}^{n} i = n(n+1)/2$, but instead Figure 9.11 gives a more visual way of seeing this result. Figure 9.11(a) shows a shaded triangle that represents the running time of selection sort: $\sum_{i=1}^{n} i$, where row i of the triangle has i steps in it. Figure 9.11(b) shows that this triangle is *contained within an n-by-n square* and also *contains an* $\frac{n}{2}$*-by-*$\frac{n}{2}$ *square*. Thus the area of the triangle is upper bounded by $n \cdot n = n^2$ and lower bounded by $\frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$, and therefore is $\Theta(n^2)$. This picture is a visual representation of a more algebraic proof:



(a) Selection Sort's running time.



(b) The analysis of the running time.

Figure 9.11: A visual representation of the proof that Selection Sort runs in $\Theta(n^2)$ time.

$$\sum_{i=1}^{n} i \leq \sum_{i=1}^{n} n = n^2, \qquad \text{and} \qquad \sum_{i=1}^{n} i \geq \sum_{i=\frac{n}{2}+1}^{n} i \geq \sum_{i=\frac{n}{2}+1}^{n} \tfrac{n}{2} = \frac{n^2}{4}.$$

While the analysis of Selection Sort isn't necessary for our main proof, the style of analysis from Figure 9.11 will be useful in a moment.

There Are No $O(n)$ Comparison-Based Sorting Algorithms

All of the sorting algorithms that we've encountered in the book are *comparison-based sorting algorithms*: they proceed by repeatedly *comparing* the values of two elements x_i and x_j from the input array without considering *the values themselves.* Depending on the result of the comparison, the algorithm may then swap some elements of the array. (Comparison-based sorting algorithms probably include every sorting algorithm that you've ever seen, except counting, radix, and bucket sorts.)

One way to view a comparison-based sorting algorithm is through a *decision tree*, like the one shown in Figure 9.12 for Selection Sort on a 3-element array. The internal nodes encode the comparisons made by the algorithm. The leaves correspond to sorted orders—the output of the sorting algorithm.

Computer Science Connections

Sorting Lower Bounds, continued

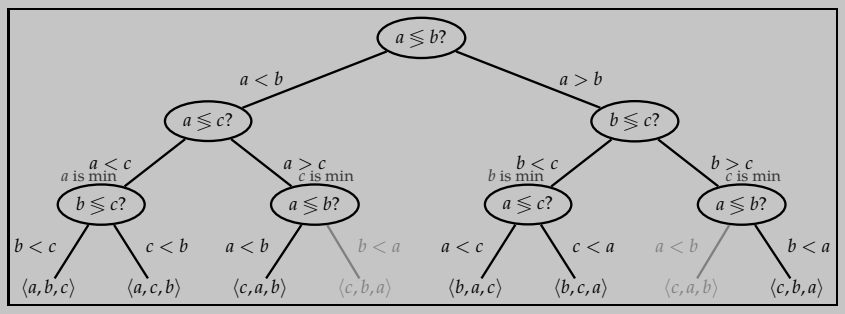


Figure 9.12: The decision tree for Selection Sort on the input array $\langle a, b, c\rangle$. Selection Sort first does two comparisons to find the minimum value of $\{a, b, c\}$, and subsequently compares the remaining two elements to decide on the final order. The two lighter-shaded branches of the tree are logically inconsistent, but Selection Sort reexecutes the a-versus-b comparison in those cases.

The running time of the sorting algorithm whose input corresponds to a particular leaf is Ω(number of comparisons on that root-to-leaf path) because, although the algorithm might do more than compare—in fact, it must (for example, it has to perform swaps)—it must do at least these comparisons.

We will use the decision tree to establish a lower bound on the running time of comparison-based sorting algorithms:

Theorem 9.8
Any comparison-based sorting algorithm requires $\Omega(n \log n)$ time.

Proof. Consider the decision tree T of the sorting algorithm. First, observe that *T must have at least $n!$ leaves.* There are $n!$ different permutations of the input, and a correct algorithm must be capable of producing any of these permutations as output. Second, observe that *T has at most 2^d nodes at depth d.* (It's a binary tree!) Thus the height h of T satisfies $2^h \geq n!$. Taking logarithms of both sides, we have

$$\begin{aligned} h \geq \log_2(n!) &= \log_2\left[n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (\tfrac{n}{2}+1) \cdot (\tfrac{n}{2}) \cdot \cdots \cdot 1\right] \\ &\geq \log_2\left[n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (\tfrac{n}{2}+1)\right] \\ &\geq \log_2\left[(\tfrac{n}{2})^{(n/2)}\right] \\ &= (\tfrac{n}{2}) \cdot \log_2(n/2) \\ &= \Omega(n \log n). \end{aligned}$$

□

The crucial fact here is precisely analogous to the one in Figure 9.11:

$$\prod_{i=1}^{n} i \geq \prod_{i=\frac{n}{2}+1}^{n} i \geq \prod_{i=\frac{n}{2}+1}^{n} \tfrac{n}{2} = (\tfrac{n}{2})^{n/2}.$$

The only difference is that here we're using products instead of summations.

A Linear-Time Sorting Algorithm

While we've now shown that every *comparison-based* sorting algorithm takes $\Omega(n \log n)$ time, there are faster algorithms for special cases. Figure 9.13 shows one, called *counting sort*, which allows us to sort without comparing elements to each other. As long as the elements of the array are *integers from a small range,* then this algorithm is fast: the running time is $\Theta(c+n)$ (the last nested loop requires $\sum_v count[v] = n$ time); as long as c is small, this algorithm runs in linear time.

```
countingSort(A[1...n]):
Input: array (A[1...n]) where each
    A[i] ∈ {1, 2, ..., c}.
 1: for v := 1 to c:
 2:     count[v] := 0
 3: for i := 1 to n:
 4:     count[A[i]] := count[A[i]] + 1
 5: i := 1
 6: for v := 1 to c:
 7:     for t := 1 to count[v]:
 8:         A[i] := v
 9:         i := i + 1
```

Figure 9.13: Counting Sort.

9.2.5 Exercises

9.1 A *tweet* (a message posted on Twitter) is a sequence of at most 140 characters. Assuming there are 256 valid different characters that can appear in each position, how many distinct tweets are possible?

Cars in the United States display license plates containing an alphanumeric code, whose format varies from state to state. Each of the following questions describes the format of currently issued license plates by some state. For each, determine the number of different license plate codes (often misleadingly called license plate numbers *despite the presence of letters) of the listed form. All letters in all codes are upper case.*

9.2 Minnesota: digit-digit-digit-letter-letter-letter (as in `400GPA`).

9.3 Pennsylvania: letter-letter-letter-digit-digit-digit-digit (as in `EEE2718`).

9.4 Connecticut: digit-letter-letter-letter-letter-digit (as in `4FIVE6`).

9.5 You have been named Secretary of Transportation for the State of [Your Favorite State]. Congratulations! You're considering replacing the current license plate format `ABCD-1234` (4 letters followed by 4 digits) with a sequence of any k symbols, each of which can be either a letter or a digit. How large must k be so that your new format has at least as many options as the old format did?

9.6 Until recently, France used license plates that contain codes of any of the following forms:

- digit-digit-digit-letter-digit-digit.
- digit-digit-digit-letter-letter-digit-digit, where the first letter is alphabetically $\leq$ `P`.
- digit-digit-digit-digit-letter-letter-digit-digit, where the first letter is alphabetically $\geq$ `Q`.
- digit-digit-digit-letter-letter-letter-digit-digit.

How many license plates, in total, met the French requirements?

9.7 A particular voicemail system allows numerical passwords of length 3, 4, or 5 digits. How many passwords are possible in this system?

9.8 What about numerical passwords of length 4, 5, or 6?

A contact lens is built with the following parameters: a (spherical) power *(for correcting near- or farsightedness); and, possibly, a* cylindrical power *and an* axis *(for correcting astigmatism). For a particular brand of contacts, the possible parameters for a lens that corrects near- or farsightedness only are*

- *a power between* -6.00 *and* $+6.00$ *inclusive in* 0.25 *steps (excluding* 0.00*); between* 6.50 *and* 8.00 *inclusive in* 0.50 *steps; and between* -6.50 *and* -10.00 *inclusive in* 0.50 *steps,*

and the parameters for a lens that corrects astigmatism are

- *one of the powers listed previously;*
- *a cylindrical power in* $\{-0.75, -1.25, -1.75, -2.25\}$*; and*
- *an axis between* 10° *and* 180° *in steps of* 10°.

9.9 How many different contact lenses are there?

9.10 A patient needing vision correction in both eyes may get different contact lenses for each eye. A *prescription* assigns a lens for the left eye and for the right eye. How many contact prescriptions are there?

9.11 During the West African Ebola crisis that started in 2014, geneticists were working to trace the spread of the disease. To do so, they acquired DNA samples of the viruses from a number of patients, and affixed a unique "tag" to each patient's sample.[2] A *tag* is a sequence of 8 nucleotides—each an element of {`A`, `C`, `G`, `T`}—attached to the end of a virus sample from each patient, so that subsequently it will be easy to identify the patient associated with a particular sample. How many different such tags are there?

[2] Richard Preston. The Ebola wars. *The New Yorker*, 27 October 2014.

9.12 In a computer science class, there are 14 students who have previously written a program in Java, and 12 students who have previously written a program in Python. How many students have previously written a program in at least one of the two languages? (If you can't give a single number as a definitive answer, give as narrow a range of possible values as you can.)

9.13 True story: a relative was given a piece of paper with the password to a wireless access point that was written as follows: a154bc0401011. But she couldn't tell from this handwriting whether each "1" was `1` (one), `l` (ell), or `I` (eye); or whether "0" was `0` (zero) or `O` (oh). How many possible passwords would she have to try before having exhausted all of the possibilities?

A Rubik's cube*—named after the 20th-century Hungarian architect Ernő Rubik—is a 3-by-3-by-3 grid of cells, where any of the six nine-cell faces (top, bottom, left, right, front, back) can be rotated 90° clockwise or counterclockwise in a single move. (See Figure 9.14.) Each face of each cell is colored with one of six colors (blue, red, green, yellow, white, and orange); initially, all nine cell-faces on each cube-face have the same color, but the cube can then be scrambled. The challenge is to use rotations to configure a scrambled cube such that each face of the cube contains nine cells of the same color.*

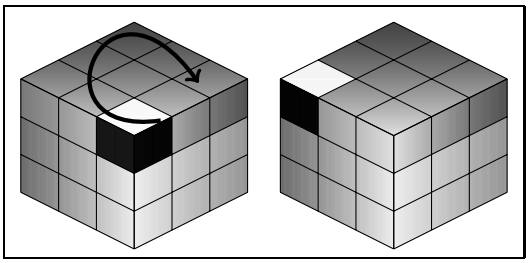


Figure 9.14: A Rubik's cube with the top face (and one cell in particular) highlighted, and the result of a single move—rotating the top face clockwise.

9.14 How many Rubik's cube moves are there?

9.15 It is known that, from any configuration, 26 moves suffice to solve the cube. (Note that we're counting every 90° rotation as a move; if you rotate the same face 180° by using two consecutive 90° moves, it counts as two moves.) How many sequences of 26 moves are possible?

9.16 It's useless to rotate a face clockwise in one move, and rotate the same face counterclockwise in the next move. (You've just undone the previous move.) A counterclockwise move followed by a clockwise move is analogous. How many sequences are there of 26 moves that never undo the previous move?

Emacs *is a widely used software program for—among other things—editing text documents (including this book!). Here's a mildly simplified description of Emacs (to make this problem more manageable). In Emacs, a* command character *is produced by pressing a letter key while holding down either the* Control *key, the* Meta *key, or both. (For example, Control+*Y *or Meta+*B *or Control+Meta+*U *are command characters.)*

9.17 How many command characters are there in Emacs?

*Emacs is complicated enough that it needs more commands than Exercise 9.17 allows. To allow for more commands, Emacs has been extended, as follows. Meta+*X *and Control+*X*—as in e*X*tended—are command* prefixes*, so that neither Meta+*X *nor Control+*X *is a valid command, but, for example, "Control+*X *Control+*U*" is (and it's different from Control+*U*). A valid command can be formed by Control+*X *or Meta+*X *followed by any letter or any command character (including Control+*X *or Meta+*X*). All other command characters from Exercise 9.17 are still valid.*

9.18 How many command characters are there now?

9.19 Argue that, for any sets A and B, $|A \cup B| = |A - B| + |B - A| + |A \cap B|$. (Use the Sum Rule.)

9.20 How many 100-bit strings have *at most* 2 ones? (Use Example 9.4.)

9.21 Determine how many k-bit strings have exactly three 1s using the approach in Example 9.4—that is, by dividing the set of bitstrings based on the position of the third one.

9.22 (*programming required*) Write a program, in a language of your choice, to enumerate all bitstrings in $\{0,1\}^{16}$ and count the number that have 0, 1, 2, and 3 ones. Use this program to verify your answer to the last exercise and your approach to Exercise 9.20.

9.23 The following is a simpler "solution" to Example 9.4, where we computed the number of elements of $\{0,1\}^k$ that have precisely two 1s. What, exactly, is wrong with this argument?

We wish to determine $|S|$, where S is the set of k-bit strings with exactly 2 ones. Define $S_i := \{x \in S : x_i = 1\}$, for each $i \in 1, 2, \ldots, k$. Observe that $S = \bigcup_{i=1}^{k} S_i$ and that $|S_i| = k - 1$. Therefore, by the Sum Rule, $|S| = \sum_{i=1}^{k} |S_i| = \sum_{i=1}^{k} (k-1) = k(k-1)$.

Unicode *is a character set frequently used on the web; it supports hundreds of thousands of characters from many languages—English, Greek, Chinese, Arabic, and all other scripts in current use. A very common encoding scheme for Unicode, called* UTF-8*, uses a variable number of bits to represent different characters (with more commonly used characters using fewer bits). Valid UTF-8 characters can be of any of the following forms, using* 1*,* 2*,* 3*, or* 4 *bytes, and have one of the following forms (where* `x` *represents an arbitrary bit):*

- `0xxxxxxx`
- `110xxxxx 10xxxxxx`
- `1110xxxx 10xxxxxx 10xxxxxx`
- `11110yyy 10yyxxxx 10xxxxxx 10xxxxxx`, *with a further restriction: the first five bits (marked* `yyyyy`*) must be either of the form* `0xxxx` *or* `10000`*.*

The ith character in the Unicode character set is encoded by the ith legal UTF-8 representation, resulting from converting i into binary and filling in the `x` *(and* `y`*) bits from the templates.*

9.24 How many characters can be encoded using UTF-8?

9.25 There's a rule for Unicode that doesn't allow excess zero padding: if a character can be encoded using one byte, then the two-byte encoding is illegal. For example, `01010101` encodes the same character as `11000001 10010101`; thus the latter is illegal. How many of the characters from the last exercise can be encoded without violating this rule?

9.26 A rook in chess can move any number of spaces horizontally or vertically. (See Figure 9.15.) How many ways are there to put one black rook and one white rook on an 8-by-8 chessboard so they can't capture each other (that is, neither can move onto the other's square)?

9.27 A queen in chess can move any number of spaces horizontally, vertically, or diagonally. (Again, see Figure 9.15.) How many ways are there to put one black queen and one white queen on an 8-by-8 chessboard so they can't capture each other (that is, neither can move onto the other's square)? (*Hint: think about how far the black queen is from the edge of the board.*)

9.28 (*programming required*) Write a program to verify your solution to the previous exercise.

9.29 You have a wireless-enabled laptop, phone, and tablet. Each device needs to be assigned a unique "send" frequency and a unique "receive" frequency to communicate with a base station. Let $S := \{1, \ldots, 8\}$ denote send frequencies and $R := \{a, \ldots, h\}$ receive frequencies. A *frequency assignment* is an element of $S \times R$. A set of frequency assignments is *noninterfering* if no elements of S or R appears twice. How many noninterfering frequency assignments are there for your three devices?

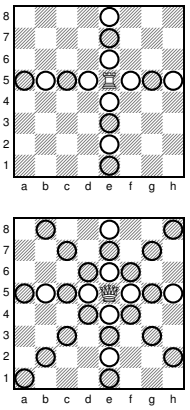


Figure 9.15: Two chess boards, showing the legal moves for a rook (above) and queen (below).

9.30 Write down an inclusion–exclusion formula for $|A \cup B \cup C \cup D|$.

9.31 How many integers between 1 and 1000, inclusive, are divisible by one or more of 3, 5, and 7?

9.32 How many integers between 1 and 1000, inclusive, are divisible by one or more of 6, 7, and 8?

9.33 How many integers between 1 and 10000, inclusive, are divisible by at least one of 2, 3, 5, or 7?

In Chapter 7, we encountered the totient function $\varphi : \mathbb{Z}^{\geq 1} \to \mathbb{Z}^{\geq 0}$, defined as

$$\varphi(n) := \textit{the number of } k \textit{ with } 1 \leq k \leq n \textit{ such that } k \textit{ and } n \textit{ have no common divisors}.$$

We can always compute the totient of n by brute force (just test all $k \in \{1, \ldots, n\}$ for common divisors using the Euclidean algorithm, for example). But the next few exercises will give a hint at another way to do this computation more efficiently. For a fixed integer n:

9.34 Suppose $m \in \mathbb{Z}^{\geq 1}$ evenly divides n. Define $M := \{k \in \{1, \ldots, n\} : m \mid k\}$. Argue that $|M| = \frac{n}{m}$.

9.35 (*A number-theoretic interlude.*) Let the prime factorization of n be $n = p_1^{e_1} \cdot p_2^{e_2} \cdots p_\ell^{e_\ell}$, for distinct prime numbers $\{p_1, \ldots, p_\ell\}$ and integers $e_1, \ldots, e_\ell \geq 1$. Let $k \leq n$ be arbitrary. Argue that k and n have no common divisors greater than 1 if and only if, for all i, we have $p_i \nmid k$.

9.36 Let n be an integer such that $n = p^i q^j$ for two distinct prime numbers p and q, and integers $i \geq 1$ and $j \geq 1$. (For example, we can write $544 = 17^1 \cdot 2^5$; here $p = 17$, $q = 2$, $i = 1$, and $j = 5$.) Let $P := \{k \in \{1, \ldots, n\} : p \mid k\}$ and $Q := \{k \in \{1, \ldots, n\} : q \mid k\}$. Argue that $\varphi(n) = n(1 - \frac{1}{p})(1 - \frac{1}{q})$ by using Inclusion–Exclusion to compute $|P \cup Q|$. (You should find the last two exercises helpful.)

In the sport of cricket, a team consists of 11 *players who come up to bat in pairs. Initially, players* #1 *and* #2 *bat. When one of those two players gets out, then player* #3 *replaces the one who got out. When one of the two batting players—player* #3 *and whichever player of* {#1, #2} *didn't get out—gets out, then player* #4 *joins the one who isn't out. This process continues until the* 10*th player gets out, leaving the last player not out (but stranded without a partner).*

Thus, in total, there are 11 *players who bat together in* 10 *partnerships. As an example, consider the lineup Anil, Brendan, Curtly, Don, Eoin, Freddie, Glenn, Hansie, Inzamam, Jacques, Kumar. We could have the following batting partnerships: Anil & Brendan; Anil & Curtly; Anil & Don; Don & Eoin; Don & Freddie; . . . ; Don & Kumar.*

9.37 How many different partnerships (pairs of players) are possible?

9.38 How many different sequences of partnerships (like the example list of partnerships given previously) are possible? (It doesn't matter which of the last two players gets out.)

9.39 A team's batting lineup may be truncated (by winning the game or by choosing not to bat any longer) at any point after the first pair starts batting. Now how many different sequences of partnerships are possible? Here, it *does* matter whether one of the last two players gets out or not (but not which of the two was the one who got out).

9.40 Suppose that, as in Example 9.11, a bank allows 4-digit PINs, but doesn't permit a PIN that starts with the same digit repeated twice (for example, 7730) or ends with the same digit repeated twice (for example, 0122). Now how many invalid PINs are there?

9.41 Let S_k denote the set of PINs that are k digits long, where the PIN may not start with three repeated digits or end with three repeated digits. In terms of k, what is $|S_k|$? (Example 9.11 computed $|S_4|$.)

Checkers *is a game, like chess, played on an 8-by-8 grid. Chinook, a recently built checkers-playing program that never loses a game,*[3] *computes all possible board positions with up to k tokens, for small k. Over the next few exercises, you'll compute the scope of that task for very small k—namely,* $k \in \{1, 2\}$*. Figuring out how many board positions have two tokens—note that two tokens can't occupy the same square!—will take a little more work.*

Briefly, the rules of checkers are as follows. Two players, Red and Black, move tokens *diagonally on an 8-by-8 grid; tokens can only occupy shaded squares. There are two types of tokens:* pieces *and* kings*. Any piece that has reached the opposite side of the board from its starting side (row 8 or row 1) becomes a king. (So Black cannot have a piece in row 8, because that piece would have become a king.) Note that Black occupying square C3 is different from Red occupying C3. (See Figure 9.16.)*

9.42 How many board positions have exactly one token (of either color)?

9.43 How many board positions have two kings, one of each color?

9.44 How many board positions have two Red kings? (Notice that two Red kings cannot be distinguished, so it doesn't matter "which" one comes first.)

9.45 How many board positions have two Black pieces?

9.46 How many board positions have two pieces, one of each color?

9.47 How many board positions have one Red king and one Red piece?

9.48 How many board positions have one Black king and one Red piece?

9.49 Use the last six exercises to determine how many total board positions have two tokens.

9.50 (*programming required*) Write a program, in a language of your choice, to verify your answer to the last few exercises (particularly the total count, in the last exercise).

9.51 How many subsets of $\{0,1\}^1 \cup \{0,1\}^2 \cup \{0,1\}^3$ are prefix free? (See Example 9.21.) You will probably find it easiest to solve this problem by writing a program.

A text-to-speech system takes written language (text) and reads it aloud as audio (speech). One of the simplest ways to build a text-to-speech system is to prerecord each syllable, and then paste together those sounds. (Pasting separate recordings is difficult, and this system as described will produce very robotic-sounding speech. But it's a start.) A syllable consists of a consonant or cluster of consonants called the onset*, then a vowel called the* nucleus*, and finally the consonant(s) called the* coda*. In many languages, only some combinations of choices for these parts are allowed—there are fascinating linguistic constraints based on ordering or place of articulation (for example, English allows stay but not tsay, and allows clay and play but not tlay) that we're almost entirely omitting here.*

9.52 A consonant can be described by a *place of articulation* (one of 11 choices: the lips, the palate, etc.); a *manner of articulation* (one of 8 choices: stopping the airflow, stopping the oral airflow with the nasal passage open, etc.); and a *voicing* (the vocal cords are either vibrating, or not). According to this description, how many consonants are there?

9.53 A vowel can be described as either *lax* or *tense*; as either *high* or *mid* or *low*; and as either *front* or *central* or *back*. According to this description, how many vowels are there?

9.54 As a (very!) rough approximation, Japanese syllables consist of one of 25 consonants followed by one of 5 vowels, with one consonant that can appear as a coda (or the coda can be left off). How many Japanese syllables are there?

9.55 As a rough (even rougher!) approximation, English syllables consist of an onset that is either one of 25 consonants or a *cluster* of any two of these consonants, followed by one of 16 vowels, followed optionally by one of 25 consonants. How many English syllables are there?

9.56 To cut down on the large number of syllables that you found in the last exercise, some systems are instead based on *demisyllables*—the first half or the second half of a syllable. (We glue the sounds together in the middle of the vowel.) That is, a demisyllable is either a legal onset followed by a vowel, or a vowel followed by a legal coda. How many demisyllables are there in English (making the same very rough assumptions as the last question)?

[3] Jonathan Schaeffer, Neil Burch, Yngvi Bjornsson, Akihiro Kishimoto, Martin Muller, Rob Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 14 September 2007.

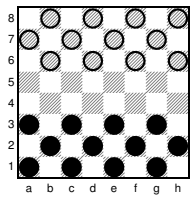


Figure 9.16: A checker board. Pieces can occupy any shaded square; a black piece that reaches row 8 or a red piece that reaches row 1 becomes a king.

9.3 *Using Functions to Count*

> The sun's shining bright
> Everything seems all right
> When we're poisoning pigeons in the park.
>
> Tom Lehrer (b. 1928), "Poisoning Pigeons In The Park"

Our focus in Section 9.2 was on counting sequences of choices (the Generalized Product Rule) and choices of choices (the Sum Rule). But what about counting other kinds of sets? Our basic plan is simple: *be lazy!* In this section, we'll introduce ways of counting the cardinality of a given set A *in terms of* $|B|$ *for some other set* B, by using functions that translate between the elements of A and the elements of B:

- *Mapping Rule:* There exists a bijection $f : A \to B$ if and only if $|A| = |B|$. Similarly, there exists an onto function $f : A \to B$ if and only if $|A| \geq |B|$, and there exists a one-to-one function $f : A \to B$ if and only if $|A| \leq |B|$.
- *Division Rule:* Suppose there exists a function $f : A \to B$ such that, for every $b \in B$, we have $|\{a \in A : f(a) = b\}| = k$. Then $|A| = k \cdot |B|$.

In particular, we'll hope to "translate" a choice from an arbitrary set into a sequence of choices from very simple sets—which, using the tools from Section 9.2, we know how to count. Here's a first example to illustrate the basic idea:

Example 9.23 (Number of valid Hamming codewords)

Problem: In Section 4.2, we introduced the Hamming code, an error-correcting code that encodes any 4-bit *message* $m \in \{0,1\}^4$ as a 7-bit *codeword* $x \in \{0,1\}^7$. Specifically, the encoding function *encode* : $\{0,1\}^4 \to \{0,1\}^7$ maps $\langle a,b,c,d\rangle$ to $\langle a,b,c,d,b \oplus c \oplus d, a \oplus b \oplus d, a \oplus b \oplus d\rangle$, where $\oplus$ is exclusive or. That is, a valid Hamming codeword x is an element of $\{0,1\}^7$ satisfying three conditions:

$$x_2 + x_3 + x_4 \equiv_2 x_5 \qquad x_1 + x_3 + x_4 \equiv_2 x_6 \qquad x_1 + x_2 + x_4 \equiv_2 x_7.$$

How many different valid codewords does the Hamming code have?

Solution: We can count the number of valid codewords by looking at all $2^7 = 128$ elements of $\{0,1\}^7$ and testing these three conditions (✓ = pass; ✗ = fail):

codeword	codeword	codeword	codeword	codeword	codeword	codeword	codeword
0000000 ✓✓✓	0010000 ✗✓✓	0100000 ✗✓✗	0110000 ✓✗✗	1000000 ✓✗✗	1010000 ✗✓✗	1100000 ✗✗✓	1110000 ✓✓✓
0000001 ✓✓✗	0010001 ✗✗✗	0100001 ✗✓✓	0110001 ✓✗✓	1000001 ✓✗✓	1010001 ✗✓✓	1100001 ✗✗✗	1110001 ✓✓✗
0000010 ✓✗✓	0010010 ✗✓✓	0100010 ✗✗✗	0110010 ✓✓✗	1000010 ✓✓✗	1010010 ✗✗✗	1100010 ✗✓✓	1110010 ✓✗✓
0000011 ✓✗✗	0010011 ✗✗✗	0100011 ✗✗✓	0110011 ✓✓✓	1000011 ✓✓✓	1010011 ✗✗✓	1100011 ✗✓✗	1110011 ✓✗✗
0000100 ✗✓✓	0010100 ✓✗✓	0100100 ✓✓✗	0110100 ✗✗✗	1000100 ✗✗✗	1010100 ✓✓✗	1100100 ✓✗✓	1110100 ✗✓✓
0000101 ✗✓✗	0010101 ✓✗✗	0100101 ✓✓✓	0110101 ✗✗✓	1000101 ✗✗✓	1010101 ✓✓✓	1100101 ✓✗✗	1110101 ✗✓✗
0000110 ✗✗✓	0010110 ✓✓✓	0100110 ✓✗✗	0110110 ✗✓✗	1000110 ✗✓✗	1010110 ✓✗✗	1100110 ✓✓✓	1110110 ✗✗✓
0000111 ✗✗✗	0010111 ✓✗✗	0100111 ✓✗✓	0110111 ✗✓✓	1000111 ✗✓✓	1010111 ✓✗✓	1100111 ✓✓✗	1110111 ✗✗✗
0001000 ✗✗✗	0011000 ✓✗✗	0101000 ✓✗✓	0111000 ✗✓✓	1001000 ✗✓✓	1011000 ✓✗✓	1101000 ✓✓✗	1111000 ✗✗✗
0001001 ✗✗✓	0011001 ✓✓✓	0101001 ✓✗✗	0111001 ✗✓✗	1001001 ✗✓✗	1011001 ✓✗✗	1101001 ✓✓✓	1111001 ✗✗✓
0001010 ✗✓✗	0011010 ✓✗✗	0101010 ✓✓✓	0111010 ✗✗✓	1001010 ✗✗✓	1011010 ✓✓✓	1101010 ✓✗✗	1111010 ✗✓✗
0001011 ✗✓✓	0011011 ✓✗✓	0101011 ✓✓✗	0111011 ✗✗✗	1001011 ✗✗✗	1011011 ✓✓✗	1101011 ✓✗✓	1111011 ✗✓✓
0001100 ✓✗✗	0011100 ✗✓✗	0101100 ✗✗✓	0111100 ✓✓✓	1001100 ✓✓✓	1011100 ✗✗✓	1101100 ✗✓✗	1111100 ✓✗✗
0001101 ✓✗✓	0011101 ✗✓✓	0101101 ✗✗✗	0111101 ✓✓✗	1001101 ✓✓✗	1011101 ✗✗✗	1101101 ✗✓✓	1111101 ✓✗✓
0001110 ✓✓✗	0011110 ✗✗✗	0101110 ✗✓✓	0111110 ✓✗✓	1001110 ✓✗✓	1011110 ✗✓✓	1101110 ✗✗✗	1111110 ✓✓✗
0001111 ✓✓✓	0011111 ✗✗✓	0101111 ✗✓✗	0111111 ✓✗✗	1001111 ✓✗✗	1011111 ✗✓✗	1101111 ✗✗✓	1111111 ✓✓✓

By checking every entry in the table, we see that there are 16 valid codewords.

Problem-solving tip: Use programming to help you! If you're going to use the simple-but-tedious way to count legal Hamming code codewords, via enumeration, write a program rather than doing it by hand. (For example, the table in Example 9.23 was generated with a Python program!)

This table-based approach is fine, but here's a less tedious way to count. By the definition of the encoding function, every possible message in $\{0,1\}^4$ is encoded as a different codeword in $\{0,1\}^7$. Furthermore, every valid codeword is the encoding of a message in $\{0,1\}^4$. Thus the number of valid codewords equals the number of messages, and there are $|\{0,1\}^4| = 16$ valid codewords.

9.3.1 *The Mapping Rule*

The approach that we used in Example 9.23 is based on *functions* that translate from one set to another. In the remainder of this section, we will formalize this style of reasoning as a general technique for counting problems. To build intuition about how to use functions to count, let's start with some small, informal examples:

Example 9.24 (Some mappings, informally)

- Let S be a collection of documents, where each document is labeled with one of 5 genres: *poem*, *essay*, *memoir*, *drama*, or *novel*.
  - Suppose every genre appears as the label for at least one document. Then $|S| \geq 5$. (We see 5 different kinds of labels on documents, and every document has only one label. Thus there must be at least 5 different documents.)
  - Suppose there's no genre that appears as the label for two distinct documents. Then $|S| \leq 5$. (No label is reused—that is, no label appears on more than one document—so we can only possibly observe 5 total labels. Every document is labeled, so we can't have more than 5 documents.)
- You're taking a class in which no two students' last names start with the same letter. Then there are at most 26 students in the class.
- You're in a club on campus that has at least one member from every state in the U.S. Then the club has at least 50 members.
- You're out to dinner with friends, and you and each of your friends order one of 8 desserts on the menu. Suppose that each dessert is ordered at least once, and no two of you order the same dessert. Then your group has exactly 8 people.

Taking it further: The document/genre scenario in Example 9.24 is an example of a *classification problem*, where we must *label* some given input data ("instances") as belonging to exactly one of k different *classes*. Classification problems are one of the major types of tasks encountered in the subfield of CS called *machine learning*. In machine learning, we try to build software systems that can "learn" how to better perform a task on the basis of some training data. Other problems in machine learning include *anomaly detection*, where we try to identify which instances from a set "aren't like" the others; or *clustering problems* (see p. 234), where we try to separate a collection of instances into coherent subgroups—for example, separating a collection of documents into "topics." Classification problems are very common in machine learning: for example, we might want to classify a written symbol as one of the 26 letters of the alphabet (*optical character recognition*); or classify a portion of an audio speech stream as one of 40,000 common English words (*speech recognition*); or classify an email message as either "spam" or "not spam" (*spam detection*).

Formalizing the rule

How can we generalize the intuition of Example 9.24 into a rule for counting? Think about the first scenario, the documents and the genres: we can view the labels on the documents in S as being given by a function

$$label : S \to \{poem, essay, memoir, drama, novel\}.$$

If there exists any function that behaves in the way that *label* did in Example 9.24—that is, either "covering" all of the possible outputs at least once each, or covering all of the possible outputs *at most* once each—then we can infer whether the set of possible inputs or the set of possible outputs is bigger.

The formal statements of the counting rules based on this intuition rely on the definition of three special types of functions that we defined in Chapter 2: onto functions, one-to-one functions, and bijections. (See Figure 9.17 for a reminder of the definitions.) Formally, the existence of a function $f : A \to B$ with one of these properties will let us relate $|A|$ and $|B|$:

Definition reminder: onto, one-to-one, and bijective functions.
Let A and B be two sets, and let $f : A \to B$ be a function. Then:

- f is *onto* if, for all $b \in B$, there exists an $a \in A$ such that $f(a) = b$.
- f is *one-to-one* if, for all $a \in A$ and $a' \in A$, if $f(a) = f(a')$ then $a = a'$.
- f is a *bijection* if it is both one-to-one and onto.

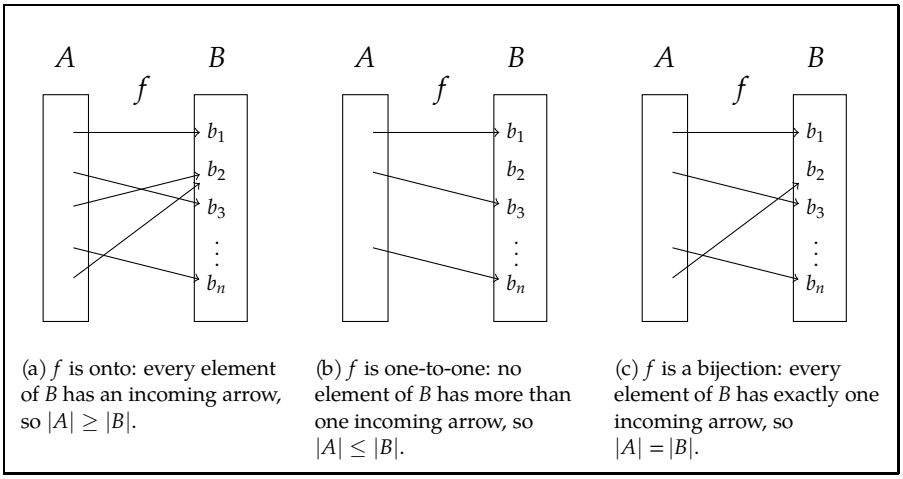
Slightly less formally: the function f is onto if "every possible output is hit"; f is one-to-one if "no output is hit more than once"; and f is a bijection if "every output is hit exactly once."

Figure 9.17: A reminder of Definitions 2.49, 2.50, and 2.51 (onto, one-to-one, and bijective functions).

Theorem 9.9 (Mapping Rule)
Let A and B be arbitrary sets. Then:

- *An onto function $f : A \to B$ exists if and only if $|A| \geq |B|$.*
- *A one-to-one function $f : A \to B$ exists if and only if $|A| \leq |B|$.*
- *A bijection $f : A \to B$ exists if and only if $|A| = |B|$.*

See Figure 9.18 for a visual representation of the Mapping Rule, and for the intuition as why it's correct: the number of arrows leaving A is precisely $|A|$; if $|A|$ arrows are enough to "cover" all elements of B, then $|B| \leq |A|$; and if $|A|$ arrows can be directed into $|B|$ elements without any duplication, then $|B| \geq |A|$. (And, actually, the third part of the Mapping Rule is implied by the first two parts: if there's a bijection $f : A \to B$ then f is both onto and one-to-one, so the first two parts of the Mapping Rule imply that $|A| \geq |B|$ *and* $|A| \leq |B|$, and thus that $|A| = |B|$.)



(a) f is onto: every element of B has an incoming arrow, so $|A| \geq |B|$.

(b) f is one-to-one: no element of B has more than one incoming arrow, so $|A| \leq |B|$.

(c) f is a bijection: every element of B has exactly one incoming arrow, so $|A| = |B|$.

Figure 9.18: The Mapping Rule. The number of arrows equals $|A|$.

A few examples

We'll start with another example—like those in Example 9.24—of the logic underlying the Mapping Rule, but this time using function terminology:

Example 9.25 (Students and assignments)
Let S be a set of 128 students in a computer science class, let A be a set of programming assignments, and suppose that $mine : S \to A$ is a function so that $mine(s)$ is the assignment that has the name of student s written on it. (Because *mine* is a function, each student's name is by definition on one and only one submitted assignment.)

- Suppose the function *mine* is onto. Then every assignment in A has at least one student's name on it—and therefore there are at least as many students as assignments: each name is written only once, and every assignment has a name on it. So $|A| \leq 128$. (There could be fewer than 128 if, for example, assignments were allowed to be submitted by pairs of students.)
- Suppose the function *mine* is one-to-one. Then no assignment has more than one name on it—and therefore there are at least as many students as assignments: each assignment has at most one name, so there can't be more names than assignments. So $|A| \geq 128$. (There could be more than 128 if, for example, there are assignments in the pile that were submitted by students in a different section of the course.)
- Suppose the function *mine* is both onto and one-to-one. Then each assignment has exactly one name written on it, and thus $|A| = |S| = 128$.

Let's also rewrite two of the informal scenarios from Example 9.24 to explicitly use functions and the Mapping Rule:

Example 9.26 (Classes, names, and states, formalized)

- Let S be the set of students taking a particular class. Define the function $f : S \to \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}$, where $f(s)$ is the first letter of the last name of student s. If no two students' last names start with the same letter, then $f(s) = f(s')$ only when $s = s'$—in other words, the function f is one-to-one. Then, by the Mapping Rule, $|S| \leq |\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}|$: there are at most 26 students in the class.
- Let T be the set of people in a particular club. Let $T' \subseteq T$ be those people in T who are from one of the 50 states. Because $T' \subseteq T$, we have $|T| \geq |T'|$.

  Define the function $g : T' \to \{Alabama, Alaska, \ldots, Wyoming\}$, where $g(x)$ is the home state of person x. If there is at least one student from every state, then for all $s \in \{Alabama, Alaska, \ldots, Wyoming\}$ there's an $x \in T'$ such that $g(x) = s$—in other words, the function g is onto. Then, by the Mapping Rule, $|T'| \geq |\{Alabama, Alaska, \ldots, Wyoming\}|$: there are at least 50 people in the club.

We'll close this section with an example of using the Mapping Rule to count the cardinality of a set that we have not yet been able to calculate. We'll do so by giving a

bijection between this new set (with previously unknown cardinality) and a set whose cardinality we *do* know.

The set that we'll analyze here is the *power set* of a set X—the set of all subsets of X, defined as $\mathscr{P}(X) := \{Y : Y \subseteq X\}$. (See Definition 2.31.) For example, $\mathscr{P}(\{0,1\})$ is $\{\{\}, \{0\}, \{1\}, \{0,1\}\}$. Let's look at the power set of $\{1,2,\ldots,8\}$:

Example 9.27 (Power set of $\{1,2,\ldots,8\}$)
Problem: What is $|\mathscr{P}(\{1,2,\ldots,8\})|$?

Solution: We'll give a bijection between $\{0,1\}^8$ and $\mathscr{P}(\{1,2,\ldots,8\})$—that is, we'll define a function $b : \{0,1\}^8 \to \mathscr{P}(\{1,2,\ldots,8\})$ that's a bijection. Here is the correspondence: for every 8-bit string $y \in \{0,1\}^8$, define $b(y)$ to be the subset $Y \subseteq \{1,2,\ldots,8\}$ such that $i \in Y$ if and only if the ith bit of y is 1. For example:

$$\begin{aligned} y = 11101010 &\quad\to\quad Y = \{1,2,3,5,7\} && \text{that is, } b(11101010) = \{1,2,3,5,7\}, \\ y = 00001000 &\quad\to\quad Y = \{5\} && \text{and } b(00001000) = \{5\}, \\ y = 00000000 &\quad\to\quad Y = \{\} && \text{and } b(00000000) = \{\}. \end{aligned}$$

Because every subset corresponds to some bitstring, and no subset corresponds to more than one bitstring, the function $b : \{0,1\}^8 \to \mathscr{P}(\{1,2,\ldots,8\})$ is a bijection between $\{0,1\}^8$ and $\mathscr{P}(\{1,2,\ldots,8\})$.

Because a bijection from $\{0,1\}^8$ to $\mathscr{P}(\{1,2,\ldots,8\})$ exists, the Mapping Rule says that $|\mathscr{P}(\{1,2,\ldots,8\})| = |\{0,1\}^8| = 2^8 = 256$.

The idea of the mapping from Example 9.27 applies for an arbitrary finite set X. Here is the general result:

Lemma 9.10 (Cardinality of the Power Set)
Let X be any finite set. Then $|\mathscr{P}(X)| = 2^{|X|}$.

Lemma 9.10 is the reason for the power set's name: the cardinality of $\mathscr{P}(X)$ is 2 to the *power* of $|X|$.

Proof. Let $n = |X|$. Let $X = \{x_1, x_2, \ldots, x_n\}$ be an arbitrary ordering of the elements of X. Define a function $f : \{0,1\}^n \to \mathscr{P}(X)$ as follows:

$$f(y) = \{x_i : \text{the } i\text{th bit of } y \text{ is } 1\}.$$

It is easy to see that f is onto: for any subset Y of X, there exists a $y \in \{0,1\}^n$ such that $f(y) = Y$. It is also easy to see that f is one-to-one: if $y \neq y'$ then there exists an i such that $y_i \neq y'_i$, so $[x_i \in f(y)] \neq [x_i \in f(y')]$. Therefore f is a bijection, and by the Mapping Rule we can conclude $|\mathscr{P}(X)| = |\{0,1\}^{|X|}| = 2^{|X|}$. □

Taking it further: Although our focus in this chapter is on finding the cardinality of *finite* sets, we can also apply the Mapping Rule to think about *infinite cardinalities.* Infinite sets are generally more the focus of mathematicians than of computer scientists, but there are some fascinating (and completely mind-bending) results that are relevant for computer scientists, too. For example, we can prove that the number of even integers *is the same* as the number of integers (even though the former is a proper subset of the latter!). But we can also prove that $|\mathbb{R}| > |\mathbb{Z}|$. More relevantly for computer science, we can prove that there are strictly more *problems* than there are *computer programs*, and therefore that *there are problems that cannot be solved by a computer.* See the discussion on p. 937.

9.3.2 *The Division Rule*

When we introduced the Inclusion–Exclusion Rule, we used an approach to counting that we might call *count first, apologize later:* to compute the cardinality of a set $A \cup B$, we found $|A| + |B|$ and then "fixed" our count by subtracting the number of elements that we'd counted twice—namely, subtracting $|A \cap B|$. Here we'll consider an analogous count-and-correct rule, called the *Division Rule*, that applies when we count every element of a set multiple times (and where each element is recounted the same number of times); we'll then correct our total by dividing by this "redundancy factor." Let's start with some informal examples:

Example 9.28 (Some redundant counting, informally)

- Suppose that the Juggling Club on campus sells 99 juggling torches to its members, in sets of three. Then there are 33 people who purchased torches.
- There are 42 people at a party. Suppose that every person shakes hands with every other person. How many handshakes have occurred? There are many ways to solve this problem, but here's an approach that uses division: each person shakes hands with all 41 other people, for a total of $(42 \text{ people}) \cdot (41 \text{ shakes/person}) = 1722$ shakes. But each handshake involves *two* people, so we've counted every shake exactly twice; thus there are actually a total of $861 = \frac{1722}{2} = \frac{42 \cdot 41}{2}$ handshakes.
- In Game 5 of the 1997 NBA Finals, the Chicago Bulls had 10 players who were on the court for some portion of the game. The number of minutes played by these ten were $\langle 45, 44, 26, 24, 24, 24, 23, 23, 4, 3 \rangle$. The total number of minutes played was $45 + 44 + 26 + 24 + 24 + 24 + 23 + 23 + 4 + 3 = 240$. In basketball, five players are on the court at a time. Thus the game lasted $\frac{240}{5} = 48$ minutes.

We'll phrase the Division Rule using the same general structure as the Mapping Rule, in terms of a function that maps from one set to another. Specifically, if we have a function $f : A \to B$ that always maps exactly the same number of elements of A to each element of B—for instance, exactly three torches are mapped to any particular juggler in Example 9.28—then $|A|$ and $|B|$ differ exactly by that factor:

Theorem 9.11 (Division Rule)
Let A and B be arbitrary sets. Suppose that there exists a function $f : A \to B$ such that, for every $b \in B$, there are exactly k elements $a_1, \ldots, a_k \in A$ such that $f(a_i) = b$. (That is, $|\{a \in A : f(a) = b\}| = k$ for all $b \in B$.) Then $|A| = k \cdot |B|$.

(The Division Rule with $k = 1$ simply *is* the bijection case of the Mapping Rule: what it means for $f : A \to B$ to be a bijection is precisely that $|\{a \in A : f(a) = b\}| = 1$ for every $b \in B$. If such a function f exists, then both the Mapping Rule and the Division Rule say that $|A| = 1 \cdot |B|$.)

Here are two simple examples to illustrate the formal version of the Division Rule:

Example 9.29 (Redundant counting, formally)

- Let M be the set of members of the Juggling Club, and let T be the set of torches bought by the members of the club. Consider the function $boughtBy : T \to M$. Assuming that each member bought precisely three torches—that is, assuming that $|\{t \in T : boughtBy(t) = m\}| = 3$ for every $m \in M$—then $|T| = 3 \cdot |M|$.
- Consider the sets $A = \{0, 1, \ldots, 31\}$ and $B = \{0, 1, \ldots, 15\}$. Define the function $f : A \to B$ as $f(n) = \lfloor n/2 \rfloor$. For each $b \in B$, there are exactly two input values whose output under f is b, namely $2b$ and $2b+1$. Thus by the Division Rule $|A| = 2 \cdot |B|$.

This basic idea—if we've counted each thing k times, then dividing our total count by k gives us the number of things—is pretty obvious, and it'll also turn out to be surprisingly useful. Here's a sequence of examples, starting with a warm-up exercise and continuing with two (slightly less obvious) applications of the Division Rule:

Example 9.30 (Rearranging PERL, PEER, and SMALLTALK)

Problem: How many different ways can you arrange the letters of ...

1. ... the name of the programming language PERL?
2. ... the word PEER?
3. ... the name of the programming language SMALLTALK?

Solution: PERL: There are 4 different letters, and any permutation of them is a different ordering. Thus there are $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ orderings. (See Theorem 9.7.)

PEER: We'll answer this question using the solution for PERL. Define the function L->E as follows: given a 4-character input string, it produces a 4-character output string in which every L has been replaced by an E. For example, L->E(PERL) = PERE. Let S denote the orderings of the word PERL, and let T denote the orderings of PEER. Note that the function L->E : $S \to T$ has the property that, for every $t \in T$, there are exactly two strings $x \in S$ such that L->E$(x) = t$. (For example, L->E(PERL) = PERE and L->E(PLRE) = PERE.) See Figure 9.19. Thus, by the Division Rule, there are $\frac{4!}{2} = \frac{24}{2} = 12$ ways to order the letters of PEER.

SMALLTALK: There are 9! different orderings of the nine "letters" in the word S M A_1 L_1 L_2 T A_2 L_3 K. (We are writing L_1 and L_2 and L_3 to denote three different "letters," and similarly for A_1 and A_2.) We will use the Division Rule repeatedly to "erase" subscripts:

- The function that erases subscripts on the As maps two inputs to each output: one with A_1 before A_2, and one with A_2 before A_1. Thus there are $\frac{9!}{2}$ different orderings of the "letters" in the word S M A L_1 L_2 T A L_3 K.
- The function that takes an ordering of S M A L_1 L_2 T A L_3 K and erases the subscripts on the Ls maps precisely *six* inputs to each output: one for each of the 3! possible orderings of the Ls.

Thus there are $\frac{9!}{2 \cdot 3!} = \frac{362{,}880}{12} = 30{,}240$ different orderings of the letters in the word S M A L L T A L K.

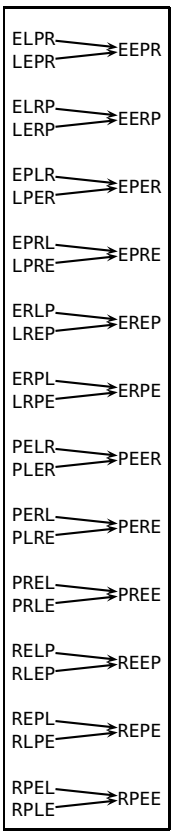


Figure 9.19: The 24 different orderings of PERL and the 12 different orderings of PEER. The function that replaces L by E is displayed by the arrows.

Counting orderings when some elements are indistinguishable

Although we phrased Example 9.30 in terms of the number of ways to rearrange the letters of some particular words, there's a very general idea that underlies the PEER and SMALLTALK examples. We'll state the underlying idea as a theorem:

Theorem 9.12 (Rearranging with duplicates)
The number of ways to rearrange a sequence containing k different distinct elements $\{x_1, \ldots, x_k\}$, where element x_i appears n_i times, is

$$\frac{(n_1 + n_2 + \cdots + n_k)!}{(n_1!) \cdot (n_2!) \cdot \ \cdots \ \cdot (n_k!)}.$$

For example, PERL has $k = 4$ distinct elements, which appear $n_{\mathsf{P}} = n_{\mathsf{E}} = n_{\mathsf{R}} = n_{\mathsf{L}} = 1$ time each; the theorem says that there are $\frac{(1+1+1+1)!}{1!\cdot 1!\cdot 1!\cdot 1!} = 4!$ ways to arrange the letters. On the other hand, SMALLTALK has $k = 6$ distinct elements, which appear $n_{\mathsf{A}} = 2$, $n_{\mathsf{L}} = 3$, and $n_{\mathsf{S}} = n_{\mathsf{M}} = n_{\mathsf{T}} = n_{\mathsf{K}} = 1$ times each; the theorem says that there are $\frac{(2+3+1+1+1+1)!}{2!\cdot 3!\cdot 1!\cdot 1!\cdot 1!\cdot 1!} = \frac{9!}{2!\cdot 3!}$ ways to arrange the letters. Let's prove the theorem:

Proof of Theorem 9.12. Let's handle a simpler case first: suppose that we have n different elements that we can put into any order, and precisely k of these n elements are indistinguishable. Then there are exactly $\frac{n!}{k!}$ different orderings of those n elements. To see this fact, imagine "decorating" each of those k items with some kind of artificial distinguishing mark, like the numerical subscripts of the letters of SMALLTALK from Example 9.30. Then there are $n!$ different orderings of the n elements. The *erase* function that eliminates our artificial distinguishing marks has $k!$ inputs that yield the same output—namely, one *for each ordering* of the k artificially marked elements. Therefore, by the Division Rule, there are $\frac{n!}{k!}$ different orderings of the elements, without the distinguishing markers.

The full theorem is just a mild generalization of this argument, to allow us to consider more than one set of indistinguishable elements. (In particular, we could give a formal proof by induction on the number of elements with $n_i \geq 2$.) In total, there are $(n_1 + n_2 + \cdots + n_k)!$ different orderings of the elements themselves, but there are $n_1!$ equivalent orderings of the first element, $n_2!$ of the second, and so forth. The function that "erases subscripts" as in Example 9.30 has $(n_1!) \cdot (n_2!) \cdot \ \cdots \ \cdot (n_k!)$ different equivalent orderings, and thus the total number of orderings is, by the Division Rule,

$$\frac{(n_1 + n_2 + \cdots + n_k)!}{(n_1!) \cdot (n_2!) \cdot \ \cdots \ \cdot (n_k!)}.$$

□

Here's another simple example that we can solve using this theorem:

Example 9.31 (Writing 232,848 as a sequence of prime factors)
Problem: How many ways can we write 232,848 as a product $p_1 p_2 \cdots p_k$, where each p_i is prime? (The *set* of prime factors, and *the number of occurrences of each factor*, are the same in every product, because the prime factorization of any positive integer is unique. But the *order* may change: for example, we can write $6 = 3 \cdot 2$ or $6 = 2 \cdot 3$.)

Solution: The prime factorization of 232,848 is $232{,}848 = 2^4 \cdot 3^3 \cdot 7^2 \cdot 11$. Thus a product of primes that equals 232,848 consists of 4 copies of two, 3 copies of three, 2 copies of seven, and one copy of eleven—in some order. (For example, $2 \cdot 2 \cdot 7 \cdot 3 \cdot 3 \cdot 7 \cdot 2 \cdot 11 \cdot 3 \cdot 2$.) By Theorem 9.12, the number of orderings of these elements is

$$\frac{(4+3+2+1)!}{4! \cdot 3! \cdot 2! \cdot 1!} = \frac{10!}{4! \cdot 3! \cdot 2!} = \frac{3{,}628{,}800}{24 \cdot 6 \cdot 2} = 12{,}600.$$

Problem-solving tip: There are often many different ways to solve a given problem—and you can use whatever approach makes the most sense *to you!* For example, Exercise 9.106 explores a completely different way to solve Example 9.32, based on the Generalized Product Rule instead of the Division Rule.

A slightly more complicated example

Here is one final example of the Division Rule, in which we'll use this approach on a slightly more complicated problem:

Example 9.32 (Assigning partners)

Problem: The professor divides the n students in a CS class into $\frac{n}{2}$ partnerships, with two students per partnership. (Assume that n is even.) The order of partners within a pair doesn't matter, nor does the order of the partnerships. (That is, the listings

Paul and George
John and Ringo

and

Ringo and John
George and Paul

represent exactly the same set of partnerships.) How many ways are there to divide the class into partnerships?

Solution: Let's line up the students in some order, and then pair the first two students, then pair the third and fourth, and so on. There are $n!$ different orderings of the students, but there are fewer than $n!$ possible partnerships, because we've double counted each set of pairs in two different ways:

- there are two equivalent orderings of the first pair of students, and two equivalent orderings of the second pair, and so on.
- the ordering of the pairs doesn't matter, so the partnerships themselves can be listed in any order at all (without changing who's paired with whom).

Each of the $\frac{n}{2}$ pairs can be listed in 2 orders, so—by the Product Rule—there are $2^{n/2}$ different possible within-pair orderings. And there are $(n/2)!$ different orderings of the pairs. Applying the Division Rule, then, we see that there are

$$\frac{n!}{(n/2)! \cdot 2^{n/2}} \qquad (*)$$

total possible ways to assign partners.

Let's make sure that $(*)$ checks out for some small values of n. For $n = 2$, there's just one pairing, and indeed $(*)$ is $\frac{2!}{1! \cdot 2^1} = \frac{2}{2} = 1$. For $n = 4$, the formula $(*)$ yields $\frac{4!}{2^3} = \frac{4 \cdot 3 \cdot 2}{8} = 3$ pairings; indeed, for the quartet Paul, John, George, and Ringo, there are three possible partners for Paul (and once Paul is assigned a partner there are no further choices to be made). See Figure 9.20 for an illustration: we try all $4! = 24$ orderings of the four people, then we reorder the names within each pair, and finally we reorder the pairs.

ordering	*reordered within pairs*	
AB CD	AB CD	AB + CD
AB DC	AB CD	
BA CD	AB CD	
BA DC	AB CD	
CD AB	CD AB	
CD BA	CD AB	
DC AB	CD AB	
DC BA	CD AB	
AC BD	AC BD	AC + BD
AC DB	AC BD	
BD AC	BD AC	
BD CA	BD AC	
CA BD	AC BD	
CA DB	AC BD	
DB AC	BD AC	
DB CA	BD AC	
AD BC	AD BC	AD + BC
AD CB	AD BC	
BC AD	BC AD	
BC DA	BC AD	
CB AD	BC AD	
CB DA	BC AD	
DA BC	AD BC	
DA CB	AD BC	

Figure 9.20: Partnerships for $n = 4$ students: the 4! orderings, then the orderings sorted within pairs, and then with the pairs sorted.

9.3.3 *The Pigeonhole Principle*

We'll close this section with a very simple—but also surprisingly useful—theorem based on the Mapping Rule, called the *pigeonhole principle.* Here are a few informal examples to introduce the underlying idea:

A *pigeonhole* refers to one of the "cells" in a grid of compartments that are open in the front, and which can house either snail mail or, back in the day, roosting pigeons. (There's also a related verb: to *pigeonhole* someone/something is to categorize that person/thing into one of a small number of—misleadingly simple—groups.)

Example 9.33 (What happens when there are more things than kinds of things)

- If there are more socks in your drawer than there are colors of socks in your drawer, then you must have two socks of the same color.
- If there are only 5 possible letter grades and there are 6 or more students in a class, then there must be two students who receive the same letter grade.
- If you take 9 or more CS courses during the 8 semesters that you're in college, then there must be at least one semester in which you doubled up on CS courses.
- In the antiquated language in which this result is generally stated: if there are n pigeonholes, and $n+1$ pigeons that are placed into those pigeonholes, then there must be at least one pigeonhole that contains more than one pigeon.

Here is the general statement of the theorem, along with its proof:

Theorem 9.13 (Pigeonhole Principle)
Let A and B be sets with $|A| > |B|$, and let $f : A \to B$ be any function. Then there exist distinct elements $a \in A$ and $a' \in A$ such that $f(a) = f(a')$.

Proof. We can prove the Pigeonhole Principle using the Mapping Rule. Given the sets A and B, and the function $f : A \to B$, the Mapping Rule tells us that

$$\text{if } f : A \to B \text{ is one-to-one, then } |A| \leq |B|. \tag{1}$$

Taking the contrapositive of (1), we have

$$\text{if } |A| > |B|, \text{ then } f : A \to B \text{ is not one-to-one.} \tag{2}$$

By assumption, we have that $|A| > |B|$, so $f : A \to B$ is not one-to-one. The theorem follows by the definition of a one-to-one function: the fact that $f : A \to B$ is not one-to-one means precisely that there is some $b \in B$ that's "hit" twice by f. In other words, there exist distinct $a \in A$ and $a' \in A$ such that $a \neq a'$ and $f(a) = f(a')$. □

A slight generalization of this idea is also sometimes useful: if there are n total objects, each of which has one of k types, then there must be a type that has at least $\lceil n/k \rceil$ objects. (We'll omit the proof, but the idea is very similar to Theorem 9.13.)

Theorem 9.14 (Pigeonhole Principle: Extended Version)
Let A and B be sets, and let $f : A \to B$ be any function. Then there exists some $b \in B$ such that the set $\{a \in A : f(a) = b\}$ contains at least $\lceil |A|/|B| \rceil$ elements.

(Another less formal way of stating this fact is "the maximum must exceed the average": the number of elements in A that "hit" a particular $b \in B$ is $|A|/|B|$ on average, and there must be some element of B that's hit at least this many times.)

We'll start with two simpler examples of the pigeonhole principle, and close with a slightly more complicated application. (In the last example, the slightly tricky part of applying the pigeonhole principle is figuring out what corresponds to the "holes.")

Example 9.34 (Congressional voting)
Suppose that there were 5 different bills upon which the House of Representatives voted yesterday. (There are 435 representatives in the U.S. House.) The pigeonhole principle implies that there are two representatives who voted identically on yesterday's bills. A representative's vote can be expressed as an element of $\{aye, nay, abstain\}^5$, which has cardinality $3^5 = 243$. Because $243 < 435$, the pigeonhole principle says that there are two representatives with the same voting record.

Example 9.35 (Logical equivalence)
Let S be a set of 17 different logical propositions over the Boolean variables p and q.

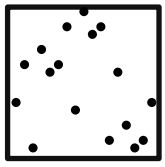
A truth table for a proposition $\varphi \in S$ is an element of $\{\text{True}, \text{False}\}^4$ (the rows of the truth table correspond to each of the four truth assignments for p and q), and there are only $|\{\text{True}, \text{False}\}^4| = 2^4 = 16$ different such values. Therefore, our 17 different propositions have only 16 different possible truth tables—so, by the pigeonhole principle, there must be two different propositions that have the same truth table.

Example 9.36 (Points in a square)
Problem: Suppose that there are $n^2 + 1$ points in a 1-by-1 square, as in Figure 9.21(a). Show that there must be two points within distance $\frac{\sqrt{2}}{n}$ of each other.

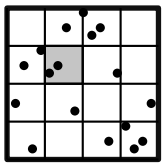
Solution: We will use the pigeonhole principle. Divide the unit square into n^2 equal-sized disjoint subsquares—each with dimension $\frac{1}{n}$-by-$\frac{1}{n}$. (To prevent overlap, we'll say that every shared boundary line is included in the square to the left or below the shared line.) There are n^2 subsquares, and $n^2 + 1$ points. By the pigeonhole principle, at least one subsquare contains two or more points. (See Figure 9.21(b).)

Notice that the farthest apart that two points in a subsquare can be is when they are at opposite corners of the subsquare. In this case, they are $\frac{1}{n}$ apart in x-coordinate and $\frac{1}{n}$ apart in y-coordinate—in other words, they are separated by a distance of

$$\sqrt{(\tfrac{1}{n})^2 + (\tfrac{1}{n})^2} = \sqrt{\tfrac{2}{n^2}} = \tfrac{\sqrt{2}}{n}.$$



(a) 17 points in a 1-by-1 square.



(b) The square divided into 16 subsquares, and one of the several doubly occupied subsquares.

Figure 9.21: Putting $n^2 + 1$ points in the unit square.

Taking it further: The pigeonhole principle can be used to show that *compression* of data files (for example, ZIP files or compressed image formats like GIF) must either lose information about the original data (so-called *lossy compression*) or must, for some input files, actually cause the "compressed" version to be larger than the original file. See the discussion on p. 938.

Computer Science Connections

Infinite Cardinalities (and Problems that Can't Be Solved by Any Program)

Recall the Mapping Rule: *for any two sets A and B, a bijection $f : A \to B$ exists if and only if $|A| = |B|$.* Although we were thinking about finite sets when we stated this rule, the statement holds even for infinite sets A and B; we can even think of this rule as *defining* what it means for two sets to have the same cardinality. Those sets S such that $|S| = |\mathbb{Z}|$, called *countable* sets, will turn out to be particularly important. Surprisingly, some sets that "seem" much bigger or much smaller than the integers have the same cardinality as $\mathbb{Z}$. For example, the set of nonnegative integers has the same cardinality as the set of all integers! (See Figure 9.22 for a bijection between these sets.) This fact is very strange—after all, we're looking at sets A and B *where A is a proper subset of B* and we've now established that $|A| = |B|$! But, indeed, because we have a bijection between A and B, they really are the same size.

Define the function $f : \mathbb{Z}^{\geq 0} \to \mathbb{Z}$ as $f(n) = \lceil \frac{n}{2} \rceil \cdot (-1)^n$. Then:

$$\begin{aligned} f(0) &= \lceil \tfrac{0}{2} \rceil \cdot (-1)^0 = 0 \cdot 1 = 0 \\ f(1) &= \lceil \tfrac{1}{2} \rceil \cdot (-1)^1 = 1 \cdot -1 = -1 \\ f(2) &= \lceil \tfrac{2}{2} \rceil \cdot (-1)^2 = 1 \cdot 1 = 1 \\ f(3) &= \lceil \tfrac{3}{2} \rceil \cdot (-1)^3 = 2 \cdot -1 = -2 \\ f(4) &= \lceil \tfrac{4}{2} \rceil \cdot (-1)^4 = 2 \cdot 1 = 2 \\ &\vdots \end{aligned}$$

Figure 9.22: A bijection between $\mathbb{Z}^{\geq 0}$ and $\mathbb{Z}$. Thus $|\mathbb{Z}^{\geq 0}| = |\mathbb{Z}|$.

p	r	i	n	t		"	h	e	l	l	o		w	o
112	114	105	110	116	32	34	104	101	108	108	111	32	119	111
1110000	1110010	1101001	1101110	1110100	100000	100010	1101000	1100101	1101100	1101100	1101111	100000	1110111	1101111

Figure 9.23: Converting a Python program into an integer. This program corresponds to the integer whose binary representation is $1110000\,1110010\,1101001\,1101110\cdots$.

Or consider a Python program p. Think of the source code of p as a file—which thus represents p as a sequence of characters, each of which is represented as a sequence of bits, which can therefore be interpreted as an integer written in binary. (See Figure 9.23.) Therefore there is a bijection f between the integers and the set of Python programs, where $f(i)$ is the ith-largest Python program (sorted numerically by its binary representation).

With all of these sets that have the same cardinality, it might be tempting to think that *all* infinite sets have the same cardinality as $\mathbb{Z}$. But they don't!

Theorem 9.15
The set of all subsets of $\mathbb{Z}^{\geq 0}$—that is, $\mathscr{P}(\mathbb{Z}^{\geq 0})$—is strictly bigger than $\mathbb{Z}^{\geq 0}$.

Proof. Suppose for a contradiction that $f : \mathbb{Z}^{\geq 0} \to \mathscr{P}(\mathbb{Z}^{\geq 0})$ is an onto function. We'll show that there's a set $S \in \mathscr{P}(\mathbb{Z}^{\geq 0})$ such that *for every $n \in \mathbb{Z}^{\geq 0}$ we have $f(n) \neq S$.* Define the set S as follows:

$$S := \{i \in \mathbb{Z}^{\geq 0} : i \notin f(i)\} \qquad \text{(So } i \in S \Leftrightarrow \text{the set } f(i) \text{ does not contain } i.\text{)}$$

Observe that the set S *differs from $f(i)$ for every i*: specifically, for every i we have $i \in S \Leftrightarrow i \notin f(i)$. Thus S is never "hit" by f—contradicting the assumption that f was onto. Therefore there is no onto function $f : \mathbb{Z}^{\geq 0} \to \mathscr{P}(\mathbb{Z}^{\geq 0})$, and, by the Mapping Rule, $|\mathbb{Z}^{\geq 0}| < |\mathscr{P}(\mathbb{Z}^{\geq 0})|$. (This argument is called a proof by *diagonalization*; see Figure 9.24.) □

We can think of any subset of $\mathbb{Z}$ as defining a *problem* that we might want to write a Python program to solve. For example, the set $\{0, 2, 4, 6, \ldots\}$ is the problem of identifying even numbers. The set $\{1, 2, 4, 8, 16, \ldots\}$ is exact powers of 2. The set $\{2, 3, 5, 7, 11, \ldots\}$ is prime numbers. What does all of this say? *There are more problems than there are Python programs!* And thus there are problems that cannot be solved by any program![4]

	0	1	2	3	4	
$f(0)$	**1**	0	1	0	1	$\cdots$
$f(1)$	0	**0**	0	1	1	$\cdots$
$f(2)$	0	1	**1**	0	1	$\cdots$
$f(3)$	1	1	0	**1**	1	$\cdots$
$f(4)$	1	0	1	0	**0**	$\cdots$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Figure 9.24: Diagonalization. Suppose that $f : \mathbb{Z}^{\geq 0} \to \mathscr{P}(\mathbb{Z}^{\geq 0})$. In a table, write row n corresponding to $f(n)$—so that $f(n)$ has a "1" in column j when $j \in f(n)$. Define $S := \{i : i \notin f(i)\}$—that is, the opposite of the diagonal element. For this table we have $0 \notin S$ (because $0 \in f(0)$), $1 \in S$ (because $1 \notin f(1)$), etc.

Problems that can't be solved by any computer program are called *uncomputable.* Section 4.4.4 identifies some particular uncomputable problems, or see a good book on computability, like

[4] Dexter Kozen. *Automata and Computability*. Springer, 1997; and Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, 3rd edition, 2012.

Computer Science Connections

Lossy and Lossless Compression

The task in *compression* is to take a large (potentially massively large!) piece of data and to represent it, somehow, using a smaller amount of space. Compression techniques are tremendously common, for a wide variety of data: text, images, audio, and video, for example. There are two fundamentally different approaches to compression of an original data file d into a compressed form d': *lossy* and *lossless* compression.

Lossy Compression. In *lossy compression*, d' does not represent exactly all of the information in d—that is, we've "lost" some information through compression. (That's why the compression is called "lossy.") In fact, many of the standard file formats for images, audio, and video are just standard methods for lossy compression. For example, JPEG is a lossy image compression format, and MP3 is a lossy audio compression format. The general goal with a lossy compression technique is to maintain, to the extent possible, "perceptual indistinguishability." For example, a digital audio stream can be represented precisely as a sequence of *intensities at each time t* ("how loud is the sound at time t?"). A lossy compression technique for sound might round the intensities: instead of representing an intensity as one of 2^{16} values ("a 16-bit sound," which is CD quality), we could round to the nearest of 2^8 values. (This idea is called *quantization*; see Example 2.56.) As long as the lost precision is smaller than the level of human perception, the new audio file would "sound the same" as the original.

Lossless Compression. In *lossless compression*, the precise contents of the original data file d can be reconstructed when the compressed data file d' is uncompressed. This approach is the one commonly used, for example, when compressing text using a program like ZIP.

The typical idea of lossless compression is to exploit redundancy in the stored data and to avoid wasting space storing the "same" information twice. For example, take the complete works of Shakespeare. By replacing every occurrence of `the` with `QQ` (two letters that don't occur consecutively in Shakespeare) the resulting file takes "only" about 99.2% of the original size. We can then set up a "translation table" telling us that `QQ` $\to$ `the` when we're decompressing. One interesting fact about lossless compression, though, is that it is *impossible* to actually compress every input file into a smaller size:

The word `the` appears over 20,000 times in the complete works of Shakespeare. The words `thee`, `them`, `their`, `they`, `there`, and `these` also appear over 1000 times each.

Theorem 9.16
Let C be any lossless compression function. Then there exists an input file d such that $C(d)$ takes up at least as much space as d.

Proof. Suppose that C compresses all n-bit inputs into $n-1$ or fewer bits. That is, $C : \{0,1\}^n \to \bigcup_{i=0}^{n-1} \{0,1\}^i$. Observe that the domain has size 2^n and the range has size $\sum_{i=0}^{n-1} 2^i = 2^n - 1$. By the pigeonhole principle, there must be two distinct input files d_1 and d_2 such that $C(d_1) = C(d_2)$. But this C cannot be a lossless compression technique: if the compressed versions of the files are identical, the decompressed versions must be identical too! □

Here's an example of a lossless "compression" function making a file bigger: I downloaded the complete works of Shakespeare from Project Gutenberg, `http://www.gutenberg.org`. It took 5,590,193 bytes uncompressed, and 2,035,948 bytes when run through `gzip`. But `shakespeare.zip.zip` (2,035,779 bytes), run through `gzip` three times, is actually bigger than `shakespeare.zip` (2,035,417 bytes).

9.3.4 Exercises

9.57 Use the idea of Example 9.23 to determine how many bitstrings $x \in \{0,1\}^7$ *fail all three* Hamming code tests—those marked "✗✗✗" in the table in Example 9.23, or satisfying these three conditions:

$$x_2 + x_3 + x_4 \not\equiv_2 x_5 \qquad x_1 + x_3 + x_4 \not\equiv_2 x_6 \qquad x_1 + x_2 + x_4 \not\equiv_2 x_7.$$

9.58 Prove that the set P of legal positions in a chess game satisfies $|P| \leq 13^{64}$. (*Hint: Define a one-to-one function from* $\{1, 2, \ldots, 13\}^{64}$ *to* P.)

Let Σ *be a nonempty set. A* string over Σ *is a sequence of elements of* Σ*—that is,* $x \in \Sigma^n$ *for some* $n \geq 0$.

9.59 How many strings of length n over the alphabet $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}, \texttt{␣}\}$ are there? How many contain exactly 2 "words" (that is, contain exactly one space ␣ that is not in the first or last position)?

9.60 Let $n \geq 3$. How many n-symbol strings over this alphabet contain exactly 3 "words"? (*Hint: use Example 9.4 to account for n-symbol strings with exactly two* ␣*s; then use Inclusion–Exclusion to prevent initial/final/consecutive spaces, as in* ␣ABC$\cdots$, $\cdots$XYZ␣, *and* $\cdots$JKL␣␣MNO$\cdots$.)

A string over the alphabet $\{\texttt{[}, \texttt{]}\}$ *is called a string of* balanced parentheses *if two conditions hold: (i) every* [*is later closed by a*]*; and (ii) every*] *closes a previous* [. (*You must close everything, and you never close something you didn't open.*) *Let* $B_n \subseteq \{\texttt{[}, \texttt{]}\}^n$ *denote the set of strings of balanced parentheses that contain n symbols.*

9.61 Show that $|B_n| \leq 2^n$: define a one-to-one function $f : B_n \to \{0,1\}^n$ and use the Mapping Rule.

9.62 Show that $|B_n| \geq 2^{n/4}$ by defining a one-to-one function $g : \{0,1\}^{n/4} \to B_n$ and using the Mapping Rule. (*Hint: consider* [][] *and* [[]].)

A certain college in the midwest requires its users' passwords to be 15 *characters long. Inspired by an XKCD comic (see* http://xkcd.com/936/*), a certain faculty member at this college now creates his passwords by choosing three* 5*-letter English words from the dictionary, without spaces. (An example password is* ADOBESCORNADORN*, from the words* ADOBE *and* SCORN *and* ADORN.*) There are* 8636 *five-letter words in the dictionary that he found.*

9.63 How many passwords can be made from any 15 (uppercase-only) letters? How many passwords can be made by pasting together three 5-letter words from this dictionary?

9.64 How many passwords can be made by pasting together three *distinct* 5-letter words from this dictionary? (For example, the password ADOBESCUBAADOBE is forbidden because ADOBE is repeated.)

The faculty member in question has a hard time remembering the order of the words in his password, so he's decided to ensure that the three words he chooses from this dictionary are different and appear in alphabetical order in his password. (*For example, the password* ADOBESCUBAFOXES *is forbidden because* SCUBA *is alphabetically after* FOXES.)

9.65 How many passwords fit this criterion? Solve this problem as follows. Let P denote the set of three-distinct-word passwords (the set from Exercise 9.64). Let A denote the set of three-distinct-alphabetical-word passwords. Define a function $f : P \to A$ that sorts. Then use the Division Rule.

9.66 After play-in games, the NCAA basketball tournament involves 64 teams, arranged in a *bracket* that specifies who plays whom in each round. (The winner of each game goes on to the next round; the loser is eliminated. See Figure 9.25.) How many different outcomes (that is, lists of winners of all games) of the tournament are there?

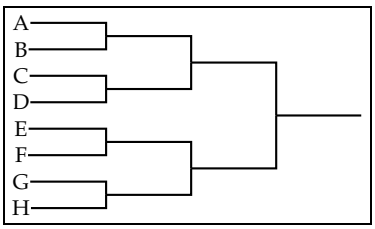


Figure 9.25: An 8-team tournament bracket. In the first round, A plays B, C plays D, etc. The A/B winner plays the C/D winner in the second round, and so forth.

A palindrome *over* Σ *is a string* $x \in \Sigma^n$ *that reads the same backward and forward—like* 0110, TESTSET*, or (ignoring spaces and punctuation)* SIT ON A POTATO PAN, OTIS!.

9.67 How many 6-letter palindromes (elements of $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}^6$) are there?

9.68 How many 7-letter palindromes (elements of $\{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}^7$) are there?

9.69 Let $n \geq 1$ be an integer, and let P_n denote the set of palindromes over Σ of length n. Define a bijection $f : P_n \to \Sigma^k$ (for some $k \geq 0$ that you choose). Prove that f is a bijection, and use this bijection to write a formula for $|P_n|$ for arbitrary $n \in \mathbb{Z}^{\geq 1}$.

Let n *be a positive integer. Recall an integer* $k \geq 1$ *is a* factor *of* n *if* $k \mid n$*. The integer* n *is called* squarefree *if there's no integer* $m \geq 2$ *such that* $m^2 \mid n$.

9.70 How many positive integer factors does 100 have? How many are squarefree?

9.71 How many positive integer factors does 12! have? (*Hint: calculate the prime factorization of* 12!.)

9.72 How many squarefree factors does 12! have? Explain your answer.

9.73 (*programming required*) Write a program that, given $n \in \mathbb{Z}^{\geq 1}$, finds all squarefree factors of n.

9.74 Consider two sets A and B. Consider the following claim: if there is a function $f : A \to B$ that is not onto, then $|A| < |B|$. Why does this claim not follow directly from the Mapping Rule?

The genre-counting problem (Example 9.24) considered a function $f : \{1, 2, \ldots, n\} \to \{1, 2, 3, 4, 5\}$. When $n = 5$ ...

9.75 How many different functions $f : \{1, 2, \ldots, 5\} \to \{1, 2, \ldots, 5\}$ are there?

9.76 How many one-to-one functions $f : \{1, 2, \ldots, 5\} \to \{1, 2, \ldots, 5\}$ are there?

9.77 How many bijections $f : \{1, 2, \ldots, 5\} \to \{1, 2, \ldots, 5\}$ are there?

9.78 Let $n \geq 1$ and $m \geq n$ be integers. Consider the set G of functions $g : \{1, 2, \ldots n\} \to \{1, 2, \ldots, m\}$. How many functions are in G? How many one-to-one functions are there in G? How many bijections?

9.79 Show that the number of bijections $f : A \to B$ is equal to the number of bijections $g : B \to A$. (*Hint: define a bijection between {bijections $f : A \to B$} and {bijections $g : B \to A$}, and use the bijection case of the mapping rule!*)

9.80 A *Universal Product Code* (*UPC*) is a numerical representation of the bar codes used in stores, with an error-detecting feature to handle misscanned codes. A UPC is a 12-digit number $\langle x_1, x_2, \ldots, x_{12} \rangle$ where $[\sum_{i=1}^{6} 3x_{2i-1} + x_{2i}] \bmod 10 = 0$. (That is, the even-indexed digits plus three times the odd-indexed digits should be divisible by 10.) Prove that there exists a bijection between the set of 11-digit numbers and the set of valid 12-digit UPC codes. Use this fact to determine the number of valid UPC codes.

9.81 A *strictly increasing sequence* of integers is $\langle i_1, i_2, \ldots, i_k \rangle$ where $i_1 < i_2 < \cdots < i_k$. How many strictly increasing sequences start with 1 and end with 1024? (That is, we have $i_1 = 1$ and $i_k = 1024$. The value of k can be anything you want; you should count both $\langle 1, 1024 \rangle$ and $\langle 1, 2, 3, 4, \ldots, 1023, 1024 \rangle$.)

A subsequence *of a sequence $x = \langle x_1, x_2, \ldots, x_n \rangle$ is a sequence $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$ of $k \geq 0$ elements of x, where $\langle i_1, i_2, \ldots, i_k \rangle$ is a strictly increasing sequence. For example,* PYTHON *is a subsequence of* PYTHAGOREAN *and* BASIC *is a subsequence of* BRAINSICKNESS.

9.82 Suppose the components of $x = \langle x_1, x_2, \ldots, x_n \rangle$ are all different (as in PYTHON but not PYTHAGOREAN). Use the Mapping Rule to figure out how many subsequences of x there are.

9.83 Suppose the components of $x = \langle x_1, x_2, \ldots, x_n \rangle$ are all different, *except for a single pair of identical elements that are separated by k other elements*. For example, PYTHAGOREAN has $n = 11$ and $k = 4$, because there are four entries (GORE) between the As (at index 5 and 10), which are the only repeated entries. In terms of n and k, how many subsequences of x are there?

As Example 9.23 describes, the Hamming Code adds 3 different parity bits to a 4-bit message m, where each added bit corresponds to the parity of a carefully chosen subset of the message bits, creating a 7-bit codeword c. Let k and n, respectively, denote the number of bits in the message and the codeword. (For the Hamming Code, we have $k = 4$ and $n = 7$.)

A decoding *algorithm takes a received (and possibly corrupted) codeword c' and determines which message has a corresponding codeword c that is most similar to c'. (See Section 4.2, or Figure 9.26 for a brief reminder. See also Exercises 4.25–4.28.) We can view the decoding algorithm as a function* decode $: \mathscr{P}(1, 2, \ldots, n-k) \to \{0, 1, 2, \ldots, n\}$*—where* decode$(S)$ *tells us which bit (if any) to flip in the received codeword when S is the set of mismatched parity bits. (If* decode$(S) = 0$*, then no bits should be flipped.)*

9.84 Argue using the Mapping Rule (that is, without reference to the precise function in Figure 9.26) that for the Hamming Code's parameters ($n = 7$ and $k = 4$) that there exists a bijection *decode* $: \mathscr{P}(\{1, 2, \ldots, n-k\}) \to \{0, 1, 2, \ldots, n\}$.

9.85 Suppose that we choose $n = 9$ and $k = 4$. Does there exist a bijection from $\mathscr{P}(\{1, 2, \ldots, n-k\})$ to $\{0, 1, 2, \ldots, n\}$? Why or why not?

9.86 Suppose that we choose $n = 31$. For what value(s) of k does there exist a bijection from $\mathscr{P}(\{1, 2, \ldots, n-k\})$ to $\{0, 1, 2, \ldots, n\}$? Prove your answer.

9.87 Prove that, for any n that is not one less than a power of 2, there does *not* exist a bijection from $\mathscr{P}(\{1, 2, \ldots, n-k\})$ to $\{0, 1, 2, \ldots, n\}$.

The Hamming code

For the message $m = \langle a, b, c, d \rangle$, we compute three parity bits:

- *parity bit #1:* $b \oplus c \oplus d$
- *parity bit #2:* $a \oplus c \oplus d$
- *parity bit #3:* $a \oplus b \oplus d$

and send $c := \langle a, b, c, d,$ parity #1, parity #2, parity #3$\rangle$.

Having received a (possibly corrupted) codeword c', we compute what the parity bits would have been for the received message bits, and check for mismatches between the computed and received parity bits:

parity bit mismatches	error (which bit to flip)
$\{\}$	no error!
$\{1\}$	parity #1
$\{2\}$	parity #2
$\{3\}$	parity #3
$\{1, 2\}$	bit c
$\{1, 3\}$	bit b
$\{2, 3\}$	bit a
$\{1, 2, 3\}$	bit d

Figure 9.26: Decoding the Hamming Code. Every single-bit error is corrected.

In the corporate and political worlds, there's a dubious technique called URL squatting, *where someone creates a website whose name is very similar to a popular site and uses it to skim the traffic generated by poor-typing internet users. For example, Google owns the addresses* `gogle.com` *and* `googl.com`, *which redirect to* `google.com`. *(But, as of this writing, someone else owns* `oogle.com`, `goole.com`, *and* `googe.com`.*) Consider an n-letter company name. How many single-typo manglings of the name are there if we consider the following kinds of errors? Consider only uppercase* letters *throughout. (If your answers depend on the particular n-letter company name, then say* how *they depend on that name. Note that no transposition errors are possible for the company name* `MMM`, *for example.)*

9.88 one-letter substitutions

9.89 one-letter insertions

9.90 one-pair transpositions (two adjacent letters written in the wrong order)

9.91 one-letter deletions

How many different ways can you arrange the letters of the following words?

9.92 `PASCAL`

9.93 `GRACEHOPPER`

9.94 `ALANTURING`

9.95 `CHARLESBABBAGE`

9.96 `ADALOVELACE`

9.97 `PEERTOPEERSYSTEM`

9.98 (*programming required*) Write a function that, given an input string, computes the number of ways to rearrange the string's letters. Use your program to verify your answers to the last few exercises.

9.99 (*programming required*) In Example 9.31, we analyzed the number of ways to write a particular integer n as the product of primes. (Because the prime factorization of n is unique, the only difference between these products is the order in which the primes appear.) Write a program, in a language of your choice, to compute the number x_n of ways we can write a given number n as $p_1 \cdot p_2 \cdots p_k$, where each p_i is prime. For what number $n \leq 10{,}000$ is x_n the greatest?

In Chapter 3, we discussed the application of Boolean logic to AI-based approaches to playing games like Tic-Tac-Toe. (See p. 344, or Figure 9.27 for a 2*-by-*2 *version of the game [Tic-Tac; the* 3*-by-*3 *version is Tic-Tac-Toe].)*

Specifically, recall the Tic-Tac-Toe game tree: the root of the tree is the empty board, and the children of any node in the tree are the boards that result from any move made in any of the empty squares. We talked briefly about why chess is hard to solve using an approach like this. (In brief: it's huge.) The next few problems will explore why a little bit of cleverness helps a lot in solving even something as simple as Tic-Tac-Toe.

9.100 Tic-Tac-Toe ends when either player completes a row, column, or diagonal. But for this question, assume that even after somebody wins the game, the board is completely filled in before the game ends. (That is, every leaf of the game tree has a completely filled board.) How many leaves are in the game tree?

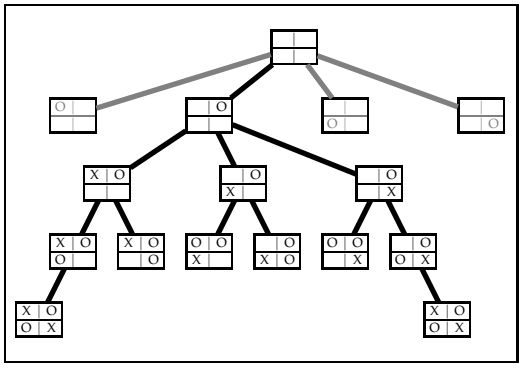


Figure 9.27: A portion of the game tree for Tic-Tac. (The missing 75% is rotated, but otherwise identical.)

9.101 Continue to assume that the board is completely filled in before the game ends. How many *distinct* leaves are there in the tree? (That is, suppose that the order in which O fills his or her squares doesn't matter; if the same squares are filled, the boards count as the same.)

9.102 Continue to assume that the board is completely filled in before the game ends. Extend your answer to Exercise 9.100: how many total boards appear in the game tree (as leaves or as internal nodes)? *(Hint: it may be easiest to compute the number of boards after k moves, and add up your numbers for* $k = 0, 1, \ldots, 9$.*)*

9.103 Continue to assume that the board is completely filled in before the game ends. How many *distinct* total boards—internal nodes or leaves—are there in the tree?

There are still two optimizations left that we haven't tried. The first is using the symmetry of the board to help us: for example, there are really only three first moves that can be made in Tic-Tac-Toe: a corner, the middle of the board, and the middle of a side. The second optimization is to truncate the tree when there's a winner. These are both a bit tedious to track by hand, but it's manageable with a small program.

9.104 (*programming required*) We can cut the size of the game tree down to less than a third of the original size—actually substantially more!—by exploiting symmetry in plays. (We're down to a third of the original size just within the first move.) Write a program to compute the entire Tic-Tac-Toe game tree, and use it to determine the number of unique boards (counting as equivalent two boards that match with respect to rotational or reflectional symmetry) in the game tree. How many boards are now in the tree?

9.105 (*programming required*) We can reduce the size of the game tree just a bit further by not expanding the portions of the game tree where one of the players has already won. Extend your implementation from the last exercise so that no moves are made in any board in which O or X has already won. How many boards are in the tree now?

Recall Example 9.32: we must put n students (where n is even) into $\frac{n}{2}$ partnerships. (We don't care about the order of the partnerships, nor about the order of partners within a pair.) Here is an alternative way of solving this problem:

9.106 Consider sorting the n people alphabetically by name. Repeat the following $\frac{n}{2}$ times: for the unmatched person p whose name is alphabetically first, choose a partner for p from the set of all other unmatched people. How many choices are there in iteration i? How many choices are there, in total?

9.107 Algebraically prove the following identity. (*Hint: what does $(n/2)! \cdot 2^{n/2}$ represent?*)

$$\prod_{i=1}^{n/2}(n-2i+1) = \frac{n!}{(n/2)! \cdot 2^{n/2}}$$

Think of an n-gene chromosome as a permutation of the numbers $\{1, 2, \ldots, n\}$, representing the order in which these n genes appear. The following questions ask you to determine how many chromosome-level rearrangement events of a particular form there are. (See, for example, Figure 3.38.)

9.108 A *prefix reversal* inverts the order of the first j genes, for some $j > 1$ and $j \leq n$. For example, for the chromosome $\langle 5, 9, 6, 2, 1, 4, 7, 3, 8\rangle$ we could get the result $\langle \underline{6, 9, 5}, 2, 1, 4, 7, 3, 8\rangle$ or $\langle \underline{1, 2, 6, 9, 5}, 4, 7, 3, 8\rangle$ from a prefix reversal. How many different prefix reversals are there for a 1000-gene chromosome?

9.109 A *reversal* inverts the order of the genes between index i and index j, for some i and $j > i$. For example, for the chromosome $\langle 5, 9, 6, 2, 1, 4, 7, 3, 8\rangle$ we could get the result $\langle \underline{6, 9, 5}, 2, 1, 4, 7, 3, 8\rangle$ or $\langle 5, 9, 6, \underline{4, 1, 2}, 7, 3, 8\rangle$ from a reversal. How many different reversals are there for a 1000-gene chromosome?

9.110 A *transposition* takes the genes between indices i and j and places them between indices k and $k+1$, for some i and $j > i$ and $k \notin \{i, i+1, \ldots, j\}$. For example, for the chromosome $\langle 5, 9, 6, 2, 1, 4, 7, 3, 8\rangle$ we could get the result $\langle 5, \underline{1, 4, 7, 3}, 9, 6, 2, 8\rangle$ or $\langle 1, 4, \underline{5, 9, 6, 2}, 7, 3, 8\rangle$ from a transposition. How many different transpositions are there for a 1000-gene chromosome?

A cellular automaton *is a formalism that's sometimes used to model complex systems—like the spatial distribution of populations, for example. Here is the model, in its simplest form. We start from an n-by-n* toroidal lattice *of cells: a two-dimensional grid, that "wraps around" so that that there's no edge. (Think of a donut.) Each cell is connected to its eight immediate neighbors.*

Cellular automata are a model of evolution over time: our model will proceed in a sequence of time steps*. At every time step, each cell u is in one of two states:* active *or* inactive*. A cell's state may change from time t to time $t+1$. More precisely, each cell u has an* update rule *that describes u's state at time $t+1$ given the state of u and each of u's neighbors at time t. (For example, see Figure 9.28.)*

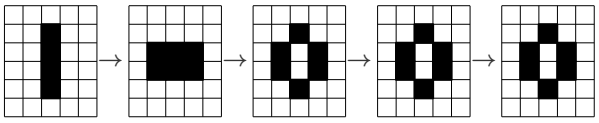


Figure 9.28: In the *Game of Life*, each cell has an identical update rule: an active cell with ≤ 1 live neighbors dies (from "loneliness"), a live cell with ≥ 4 live neighbors dies (from "overcrowding"), and a dead cell with exactly three living neighbors becomes alive.

9.111 An *update rule* is a function that takes the state of a cell and the state of its eight neighbors as input, and produces the new state of the cell as output. How many different update rules are there?

9.112 Let's call an update rule a *strictly cardinal update rule* if—as in the Game of Life—the state of a cell u at time $t+1$ depends only the following: (i) the state of cell u at time t, and (ii) the *number* of active neighbors of cell u at time t. How many different strictly cardinal update rules are there?

Suppose that we have an 10-by-10 lattice of 100 cells, and we have an update rule f_u for every cell u. (These update rules might be the same or differ from cell to cell.) Suppose the system begins in an initial configuration M_0. Suppose we start the system at time $t = 0$ in configuration M_0, and derive the configuration M_t at time $t \geq 1$ by computing

$$M_t(u) = f_u(\text{the states of } u\text{'s neighbors in } M_{t-1}).$$

Let's consider the possible outcomes of the sequence $M_0, M_1, M_2, \ldots$. Say that this sequence exhibits eventual convergence *if the following holds: there exists a time $t \geq 0$ such that, for all times $t' \geq t$, we have $M_{t'} = M_t$. (So the Life example in Figure 9.28 exhibits eventual convergence.) Otherwise, we'll say that this sequence* oscillates.

9.113 Given M_0 and the f_u's, we'd like to know what the long-run behavior of this system is: does it eventually converge or does it oscillate? Prove that, for a sufficiently large value of K, we have eventual convergence if and only if the following algorithm returns True. Also compute the smallest value of K for which this algorithm is guaranteed to be correct.

- Start with $M := M_0$ and $t := 0$.
- Repeat the following K times: update M to the next time step (that is, for each u compute the updated $M'(u)$ by evaluating f_u on u's neighbor cells in M).
- If M would be unchanged by one additional round of updates, return True. Else return False.

9.114 Suppose that we place 1234 items into 17 buckets. (For example, consider hashing 1234 items into a 17-cell hash table.) Call the number of items in a bucket its *occupancy*, and the *maximum occupancy* the number of items in the most-occupied bucket. What's the smallest possible maximum occupancy?

9.115 Consider a function $f : A \to B$. Fill in the blank with a statement relating $|A|$ and $|B|$, and then prove the resulting claim: if ____, then, for some $b \in B$, we have $|\{a \in A : f(a) = b\}| \geq 202$.

9.116 Suppose that we quantize a set of values from $S = \{1, 2, \ldots, n\}$ into $\{k_1, k_2, \ldots, k_5\} \subset S$. (See Example 2.56.) Namely, we choose these 5 values and then define a function $q : S \to \{k_1, k_2, \ldots, k_5\}$. The *maximum error* of this quantization is $\max_{x \in S} |x - q(x)|$. Use the Pigeonhole Principle (or the "the maximum must exceed the average" generalization) to determine the smallest possible maximum error.

Imagine a round-robin chess tournament for 150 *players, each of whom plays* 7 *games. (In other words, each player is guaranteed to participate in precisely* 7 *games with* 7 *different opponents. Remember that each game has two players.)*

9.117 There are 20 possible first moves for White in a chess game, and 20 possible first moves for Black in response. (See Example 9.15.) Prove that there must be two different games in the tournament that began with the same first two moves (one by White and one by Black).

9.118 Suppose that would-be draws in this tournament are resolved by a coin flip, so that every game has a winner and a loser. Prove that there must be two participants in such a tournament who have precisely the same sequence of wins and losses (for example, WWWLLLW).

A win–loss record reports a number of wins and a number of losses (for example, 6 *wins and* 1 *loss, or* 3 *wins and* 4 *losses), without reference to the order of these results.*

9.119 Continuing to suppose that there are no draws in this tournament, identify as large a value of k as you can for which the following claim is true, and prove that it's true for your value of k: there is some win–loss record that is shared by at least k competitors.

9.120 Now suppose that draws are allowed, so that competitors have a win–loss–draw record (for example, 2 wins, 1 loss, and 4 draws). Identify the largest k for which there is some win–loss–draw record that is shared by at least k competitors, and prove that this claim holds for the k you've identified.

9.4 Combinations and Permutations

> Not everything that can be counted counts, and not everything that counts can be counted.
>
> William Bruce Cameron (1921–2002)

So far in this chapter, we've been working to develop a toolbox of general techniques for counting problems: the Sum Rule and Inclusion–Exclusion, the (Generalized) Product Rule, the Mapping Rule, and the Division Rule. This section will be different; instead of a new technique, here we will devote our attention to a particularly common kind of counting problem: the number of ways to *choose* a subset from a given set of candidate elements. Let's start with an illustrative example:

Example 9.37 (Printing t-shirts)
Problem: Suppose you run a t-shirt shop. There is a collection of *jobs* that you're asked to run, but there's limited time so you must choose which ones to actually print. There are 17 requested jobs $\{\texttt{a}, \texttt{b}, \ldots, \texttt{q}\}$, but there is only time to print 4 different jobs. How many ways are there to select 4 of these 17 candidate jobs?

Solution: There are two answers, depending on how we interpret the problem: does the *order* of the printed jobs matter, or does it only matter *whether* a job was printed? (Are we choosing an ordered 4-tuple? Or an unordered subset of size 4?)

Order matters: Then the Generalized Product Rule immediately gives us the answer: there are 17 choices for the first job, 16 for the second job, 15 for the third, and 14 for the fourth; thus there are $17 \cdot 16 \cdot 15 \cdot 14$ total choices.
Another way to write $17 \cdot 16 \cdot 15 \cdot 14$ is $\frac{17!}{13!}$: every multiplicand between 1 and 13 appears in both the numerator and denominator, leaving only $\{17, 16, 15, 14\}$ uncancelled. We can justify the $\frac{17!}{13!}$ version of the answer using the Division Rule: we choose one of the 17! orderings of all 17 jobs, and then print the first 4 jobs in this order—but we've counted each 4-job ordering 13! times (once for each ordering of the 13 unprinted jobs), so we must divide by 13!.

Order doesn't matter: As in the previous case, there are $\frac{17!}{13!}$ ways of choosing an ordered sequence of 4 jobs. Because order doesn't matter, we have counted each set of four chosen jobs 4! times, once for each ordering of them. By the Division Rule, then, there are $\frac{17!}{13! \cdot 4!}$ ways of selecting 4 unordered jobs from a set of 17.

Two different fundamental notions of choice are illustrated by Example 9.37: *permutations*, in which the order of the chosen elements matters, and *combinations*, in which the order doesn't matter. These two notions will be our focus in this section. Here's another example to further illustrate combinations:

Example 9.38 (Arranging letters of a bitstring)
Problem: How many different ways can you arrange the symbols in the "word" `000111`? What about the "word" `00...011...1` containing k zeros and $n - k$ ones?

Solution: This problem is just another application of the techniques we used for `PERL` and `PEER` and `SMALLTALK` in Example 9.30. (We can think of the word `000111` just like a word like `DEEDED`: two different letters, appearing three times each.) There are 6 total characters in the word, each appearing 3 times, so the total number of arrangements is $\frac{6!}{3!\cdot 3!}$. (See Theorem 9.12.)

For the general version of the problem—the word `00...011...1`, with k zeros and $n-k$ ones—we have a total of n characters, so there are $n!$ ways of writing them down. But $k!$ orderings of the zeros, and $(n-k)!$ orderings of the ones, are identical. Hence, by the Division Rule, the total number of orderings is $\frac{n!}{k!\cdot(n-k)!}$.

Combinations

The quantity that we computed in Example 9.38 is called the number of *combinations* of k elements chosen from a set of n candidates:

Definition 9.2 (Combinations)
Consider nonnegative integers n and k with $k \leq n$. The quantity $\binom{n}{k}$ is defined as

$$\binom{n}{k} := \frac{n!}{k!\cdot(n-k)!},$$

and is read as "n choose k."

The quantity $\binom{n}{k}$ is also sometimes called a *binomial coefficient*, for reasons that we'll see in Section 9.4.3. It's also sometimes denoted $C(n,k)$ ("C" as in "Combination").

As we just argued in Example 9.38, the quantity $\binom{n}{k}$ denotes the number of ways to choose a k-element subset of a set of n elements. For convenience, define $\binom{n}{k} := 0$ whenever $n < 0$ or $k < 0$ or $k > n$: there are *zero* ways to choose a k-element subset of a set of n elements under these circumstances.

Taking it further: When there are annoying complications (or divide-by-zero errors or the like) in the boundary cases of a definition, it's often easiest to tweak the definition to make those cases less special. (Here, for example, instead of having $\binom{7}{8}$ be undefined, we treat $\binom{7}{8}$ as 0.)

A similar idea in programming can make life much simpler when you encounter data structures with complicated edge conditions—for example, a node in a linked list that might not have a successor. A *sentinel* is a "fake" element that you might add to the boundary of a data structure that makes the edge elements of the data structure less special. For example, in image processing, we might augment an n-by-m image with an extra 0th and $(m+1)$st column, and an extra 0th and $(n+1)$st row, of blank pixels. Once these "border pixels" are added, *every pixel in the image has a neighbor in each cardinal direction.* Thus there's no special code required for edge pixels in code to, for example, apply a blur filter to the image.

Here are a few small examples of counting problems that use combinations:

Example 9.39 (8-bit strings with 2 ones)
How many different 8-bit strings have exactly 2 ones?

We solved this precise problem in Example 9.3 using the Sum Rule, but combinations give us an easier way to answer this question. We must choose 2 out of 8 indices to make equal to one. There are $\binom{8}{2} = \frac{8!}{2!\cdot(8-2)!} = \frac{8!}{2!\cdot 6!} = \frac{8\cdot 7}{2} = 28$ such choices of indices, and thus $\binom{8}{2}$ different 8-bit bitstrings with exactly 2 ones. These 28 strings are shown in Figure 9.29.

```
11000000
10100000
10010000
10001000
10000100
10000010
10000001
01100000
01010000
01001000
01000100
01000010
01000001
00110000
00101000
00100100
00100010
00100001
00011000
00010100
00010010
00010001
00001100
00001010
00001001
00000110
00000101
00000011
```

Figure 9.29: All 8-bit bitstrings with exactly 2 ones.

Example 9.40 (32-bit strings with < 3 ones)
How many different 32-bit strings have fewer than 3 ones?

We will use the Sum Rule, plus the formula for combinations. (We can partition the set of 32-bit strings that have fewer than 3 ones into those with 0, 1, or 2 ones.) Thus there are $\binom{32}{0} + \binom{32}{1} + \binom{32}{2} = 1 + 32 + \frac{32 \cdot 31}{2} = 1 + 32 + 496 = 529$ total such strings.

(Recall that $0! = 1$, so $\binom{32}{0} = \frac{32!}{0! \cdot (32-0)!} = \frac{32!}{0! \cdot 32!} = \frac{32!}{1 \cdot 32!} = \frac{32!}{32!} = 1$.)

Finally, here's an example of counting using combinations that relates counting to probability. (There's much more about probability in Chapter 10.) If we flip an *unbiased coin* (in other words, a coin that comes up heads with probability $\frac{1}{2}$ and tails with probability $\frac{1}{2}$ each time we flip it), then every sequence of coin flips is equally likely. The probability that an "event" E happens when we flip an unbiased coin is the fraction of possible flip sequences for which E actually occurs.

Example 9.41 (Exactly 50% heads)
Suppose we flip an unbiased coin 10 times. What is the probability that precisely 5 flips come up heads?

There are $2^{10} = 1024$ total sequences, of which $\binom{10}{5} = \frac{10!}{5! \cdot 5!} = 252$ have precisely 5 heads. Thus there's a $\frac{252}{1024} \approx 0.2461$ chance of exactly half of the flips being heads.

9.4.1 *Four Different Ways to Select k out of n Options*

In Example 9.37, we saw two different ways in which we can imagine choosing a subset of k distinct elements from a set S of n candidates, depending on whether the *order* in which we choose those k elements matters.

There is another dichotomy that can arise in counting problems: we can imagine circumstances in which we choose k elements from a set S, but where *repetition* is allowed (that is, we can choose the same element more than once). In other scenarios, repetition might not make sense. Here are some examples of all four situations (see also Figure 9.30):

order matters, repetition allowed (9 *ways*)	*order matters, repetition not allowed* (6 *ways*)	*order irrelevant, repetition allowed* (6 *ways*)	*order irrelevant, repetition not allowed* (3 *ways*)
A, then A		A and A	
A, then B	A, then B	A and B	A and B
B, then A	B, then A		
A, then C	A, then C	A and C	A and C
C, then A	C, then A		
B, then B		B and B	
B, then C	B, then C	B and C	B and C
C, then B	C, then B		
C, then C		C and C	

Figure 9.30: Four ways of choosing 2 elements from the candidates A, B, and C—depending on whether we can choose the same element more than once, and whether the order of choices matters.

- You order a two-scoop ice cream cone from a list of flavors. Order matters: a chocolate scoop on top of a mint scoop $\neq$ mint on top of chocolate. Repetition is allowed: you can choose vanilla for both scoops.
- Your soccer game is tied, and you must choose 5 of your 11 players to take penalty kicks to break the tie. Order matters: the kicks are taken in sequence, so Pelé then Maradona $\neq$ Maradona then Pelé. Repetition is forbidden: each player is allowed to take only one kick.
- You order a three-salad salad sampler from a list of salads. Order doesn't matter: salads are served on a round plate, so it doesn't matter which one is "first." Repetition is allowed: you can choose the Caesar as two or all three of your salads.

• You select a starting lineup of 5 basketball players from your 13-person team. Order doesn't matter: all 5 chosen players are equivalent in starting the game. Repetition is forbidden: you must choose five different players.

Here we will consider all four types of counting problems—ordered/unordered choice with/without repetition—and do a few examples. See Figure 9.31 for a summary of the number of ways to make these different types of choices.

	order matters	*order doesn't matter*
repetition forbidden	$\frac{n!}{(n-k)!}$	$\binom{n}{k}$
repetition allowed	n^k	$\binom{n+k-1}{k}$

Figure 9.31: Four ways of selecting k of n items, and the number of ways to make that selection.

When order matters and repetition is forbidden

Suppose that we choose a *sequence* of k *distinct* elements from a set S: that is, the *order of the selected elements matters* and *repetition is not allowed.* (For example, in a player draft for a sports league, no player can be chosen more than once—"repetition is forbidden"—and the outcome of the draft depends not just on whether Babe Ruth was chosen, but also whether it was the Eagles or the Wildcats that selected him.)

In other words, we make k successive selections from S, but no candidate can be chosen more than once. Such a sequence is sometimes called a *k-permutation* of S—an ordered sequence of k distinct elements of S. (Recall from Definition 9.1 that a *permutation* of a set S is an ordering of S's elements.)

Some people denote the number of ways of choosing an ordered sequence of k distinct selections from a set of n options by $P(n,k)$, because "permutation" starts with "P."

There are $\frac{n!}{(n-k)!}$ different k-permutations of an n-element set S, by the Generalized Product Rule. (Specifically, there are

$$\underbrace{(n)}_{\text{choices of first element}} \cdot \underbrace{(n-1)}_{\text{choices of second element}} \cdot \cdots \cdot \underbrace{(n-k+1)}_{\text{choices of } k\text{th element}}$$

total choices, and $\frac{n!}{(n-k)!} = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot (n-k+1)$.)

Example 9.42 (4 **of** 10)
Suppose that you are asked to place four of the cards $\{A\heartsuit, 2\heartsuit, \cdots, 10\heartsuit\}$ on the table, arranged from left to right in an order of your choosing. There are $10 \cdot 9 \cdot 8 \cdot 7 = \frac{10!}{(10-4)!}$ such arrangements: order matters ($A234\heartsuit \neq 432A\heartsuit$) and repetition is not allowed ($4444\heartsuit$ isn't a valid arrangement, because you only have one $4\heartsuit$ card).

When order matters and repetition is allowed

Suppose that we simply choose a sequence of k (not necessarily distinct) elements: that is, *order matters* and *repetition is allowed.* In other words, we make k successive selections from S, and we're allowed to make the same choice multiple times. (For example, suppose you and $k-1$ friends go to a Chinese restaurant with n items on the menu, and each of you orders something for dinner. You're allowed to order the same dish as your friends—"repetition is allowed"—but you getting the Tofu with Black Bean Sauce and your vegan friend getting Twice-Cooked Pork is definitely different from the other way around.)

Then there are n^k different ways to make this choice, by the Product Rule: at every stage, there are n possible choices, and there are k stages.

Example 9.43 (4 of 10, a second way)
Suppose that you are asked to create a 4-digit integer. There are 10^4 such integers: order matters ($1234 \neq 4321$) and repetition is allowed (4444 is a valid 4-digit number).

When order doesn't matter and repetition is forbidden

Suppose that we choose an *unordered* set of k *distinct* elements: that is, *order does not matter* and *repetition is not allowed*. (For example, suppose you and $n-1$ friends enter a raffle in which k identical new cell phones will be given away. Each of you puts your name on one of n cards that are placed in a hat, and k cards are drawn to choose the winners. Cards for winners are not put back into the hat after they're drawn, so nobody can win twice—"repetition is forbidden"—but Alice and Bob winning is the same as Bob and Alice winning.)

When we choose an unordered set of k distinct elements from a set of n options, there are $\binom{n}{k}$ different ways to make this choice, by the definition of combination. Such a subset is sometimes called a *k-combination* of S—an unordered set of k distinct elements of S. (Recall from Definition 9.2 that a *combination* of elements from a set S is precisely an unordered subset of elements from S.)

Example 9.44 (4 of 10, another way)
Suppose that you're asked to create a 10-bit number with exactly 4 ones. You do so by starting with `0000000000` and choosing 4 indices to change from `0` to `1`. There are $\binom{10}{4}$ such bitstrings: the order in which you choose a bit to make a 1 doesn't matter (changing bit #2 and then bit #7 to 1 yields the same bitstring as changing bit #7 and then bit #2 to 1) and repetition is not allowed (you have to change 4 *different* bits to 1).

When order doesn't matter and repetition is allowed

While these three types of selecting k out of n elements are the most frequent, the fourth possibility can sometimes arise, too: *order doesn't matter* but *repetition is allowed*. Let's build some intuition for this case with a longer example:

Example 9.45 (Taking notes on six sheets of paper in three classes)
Problem: You discover that your school notebook has only $k = 6$ sheets of paper left in it. You are attending $n = 3$ different classes today: Archaeology (A), Buddhism (B), and Computer Science (C). How many ways are there to allocate your six sheets of paper across your three classes? (No paper splitting or hoarding: each sheet must be allocated to one and only one class!)

(Here's another way to phrase the question: you must choose how many pages to assign to A, how many to B, and how many to C. That is, you must choose three nonnegative integers a, b, and c with $a+b+c = 6$. How many ways can you do it?)

Problem-solving tip: When you encounter a problem that seems completely novel, run through the techniques you know about and try them on for size, even if they're not an obvious fit. The type of counting in Example 9.45 doesn't seem like it has a lot to do with combinations, but by changing the way you view this problem it can be transformed into a problem you've seen before.

Solution: The 28 ways of allocating your paper are shown in the following tables, sorted by the number of pages allocated to Archaeology (and breaking ties by the number of pages allocated to Buddhism). The allocations are shown in three ways:

- Pages are represented by the class name.
- Pages are represented by □, with | marking divisions between classes: we allocate the number of pages before the first divider to *A*, the number between the dividers to *B*, and the number after the second divider to *C*.
- Pages are represented by 0, with 1 marking divisions between classes: as in the □-and-| representation, we allocate pages before the first 1 to *A*, those between the 1s to *B*, and those after the second 1 to *C*.

Here are the 28 different allocations:

AAAAAA		
AAAAA	B	
AAAAA		C
AAAA	BB	
AAAA	B	C
AAAA		CC
AAA	BBB	
AAA	BB	C
AAA	B	CC
AAA		CCC
AA	BBBB	
AA	BBB	C
AA	BB	CC
AA	B	CCC
AA		CCCC
A	BBBBB	
A	BBBB	C
A	BBB	CC
A	BB	CCC
A	B	CCCC
A		CCCCC
	BBBBBB	
	BBBBB	C
	BBBB	CC
	BBB	CCC
	BB	CCCC
	B	CCCCC
		CCCCCC

A	B	C
□□□□□□		
□□□□□	□	
□□□□□		□
□□□□	□□	
□□□□	□	□
□□□□		□□
□□□	□□□	
□□□	□□	□
□□□	□	□□
□□□		□□□
□□	□□□□	
□□	□□□	□
□□	□□	□□
□□	□	□□□
□□		□□□□
□	□□□□□	
□	□□□□	□
□	□□□	□□
□	□□	□□□
□	□	□□□□
□		□□□□□
	□□□□□□	
	□□□□□	□
	□□□□	□□
	□□□	□□□
	□□	□□□□
	□	□□□□□
		□□□□□□

00000011
00000101
00000110
00001001
00001010
00001100
00010001
00010010
00010100
00011000
00100001
00100010
00100100
00101000
00110000
01000001
01000010
01000100
01001000
01010000
01100000
10000001
10000010
10000100
10001000
10010000
10100000
11000000

All three versions of this table accurately represent the full set of 28 allocations, but let's concentrate on the representation in the second and third columns—particularly the third. The 0-and-1 representation in the third column contains *exactly* the same strings as Figure 9.29, which listed all $28 = \binom{8}{2}$ of the 8-bit strings that contain exactly 2 ones.

In a moment, we'll state a theorem that generalizes this example into a formula for the number of ways to select k out of n elements when order doesn't matter but repetition is allowed. But, first, here's a slightly different way of thinking about the result in Example 9.45 that may be more intuitive.

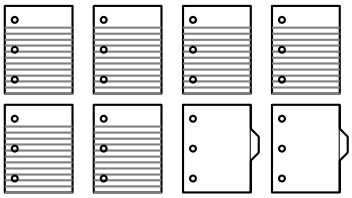


Figure 9.32: Any ordering of 6 pieces of paper and 2 divider tabs defines three sections (before, between, and after the dividers).

Suppose that we're trying to allocate a total of k pages among n classes. Imagine placing the k pages into a three-ring binder along with $n-1$ "divider tabs" (the kind that separate sections of a binder), as in Figure 9.32. There are now $n+k-1$ things in your binder. (In Example 9.45, there were 6 pages and 2 dividers, so 8 total things are in the binder.) The ways of allocating the pages precisely correspond to the ways of ordering the things in the binder—that is, choosing which of the $n+k-1$ things in the binder should be blank sheets of paper, and which should be dividers. So there are $\binom{n+k-1}{k}$ ways of doing so. In Example 9.45, we had $n=3$ and $k=6$, so there were $\binom{8}{6}=28$ ways of doing this allocation.

While the description in Example 9.45 wasn't stated in precisely these terms, our paper-allocation task was really a task about choosing with repetition: six times (once for each piece of paper), we select one of the elements of the set $\{A, B, C\}$ of classes. We may select the same class as many times as we wish ("repetition is allowed"), and the pieces of paper are indistinguishable ("order doesn't matter"). Here is the general statement of the number of ways to select k out of n elements for this scenario:

Theorem 9.17 (Choosing with repetition when order doesn't matter)
The number of ways to select k out of n elements when order doesn't matter but repetition is allowed is $\binom{n+k-1}{k}$.

Proof. We'll give a proof based on the Mapping Rule. We can represent a particular choice of k elements from the set of n candidates as a sequence $x \in (\mathbb{Z}^{\geq 0})^n$ such that $\sum_{i=1}^{n} x_i = k$. (Specifically, x_i tells us how many times we chose element i.) Define

$$X := \{x \in (\mathbb{Z}^{\geq 0})^n : \textstyle\sum_{i=1}^{n} x_i = k\}$$
$$\text{and } S := \{x \in \{0,1\}^{n+k-1} : x \text{ contains exactly } n-1 \text{ ones and } k \text{ zeros}\}.$$

We claim that there is a bijection between X and S. Specifically, define $f : X \to S$ as

$$f(x_1, x_2, \ldots, x_n) = \underbrace{00\cdots0}_{x_1 \text{ times}}\ 1\ \underbrace{00\cdots0}_{x_2 \text{ times}}\ 1\ \cdots\ 1\ \underbrace{00\cdots0}_{x_n \text{ times}}$$

(This representation is precisely the one in Example 9.45.) It's easy to see that f is a bijection: every element of S corresponds to one and only one element of X. As we argued in Example 9.38, the cardinality of S is $\binom{n+k-1}{k}$. □

Here's another example of this type of choice:

Example 9.46 (4 of 10, one last way)
Suppose that you have decided to buy 4 total drinks for a group of 10 of your friends. (You may buy multiple drinks for the same friend.) You can think of lining your friends up and performing a total of 13 successive actions, each of which is either (a) buying a drink for the friend immediately in front of you, or (b) shouting "next!". Of your 13 actions, 4 must be drink purchases. (The other 9 must be shouts of "next!") There are $\binom{13}{4}$ ways to choose these actions.

Choosing k of n elements, summarized

We've now discussed four notions of choosing k elements from a set of n candidates, depending on whether we could choose the same option more than once and whether the order of our choices mattered:

- order matters and repetition is allowed: n^k ways.
- order matters and repetition is forbidden: $\frac{n!}{(n-k)!}$ ways.
- order doesn't matter and repetition is allowed: $\binom{n+k-1}{k}$ ways.
- order doesn't matter and repetition is forbidden: $\binom{n}{k}$ ways.

(Or see Figure 9.31 for a summary.) We've also considered the same example—choosing 4 of 10 options—in each setting, and the number of ways to do so was different in each of the four different scenarios:

- order matters and repetition is allowed: $10{,}000 = 10^4$ ways.
- order matters and repetition is forbidden: $5040 = 10 \cdot 9 \cdot 8 \cdot 7$ ways.
- order doesn't matter and repetition is allowed: $715 = \binom{13}{4}$ ways.
- order doesn't matter and repetition is forbidden: $210 = \binom{10}{4}$ ways.

Taking it further: In CS, we frequently encounter tasks where we must identify the best solution from a set of possibilities. For example, we might want to find the *longest increasing subsequence* (*LIS*) of a sequence of n integers. A *brute-force algorithm* is one that solves the problem by literally trying every possible solution and selecting the best. (For LIS, there are 2^n subsequences, so this algorithm is very slow.) But if there's a certain kind of structure and enough repetition in the subproblems that arise in a naïve recursive solution, a more advanced algorithmic design technique called *dynamic programming* can yield a much faster algorithm. And counting the number of subproblems—and the number of distinct subproblems!—is what establishes when algorithms using brute force or dynamic programming are good enough. See the discussion on p. 959.

9.4.2 *Some Properties of $\binom{n}{k}$, and Combinatorial Proofs*

Of the four ways of choosing k elements from n candidates that we explored in Section 9.4.1, perhaps the most common is the setting when order doesn't matter and repetition is forbidden. In this section, we'll explore some of the remarkable mathematical properties of the numbers—the values of $\binom{n}{k}$—that arise in this scenario.

The properties that we'll prove here (and those that you'll establish in the exercises) will be equalities of the form $x = y$ for two expressions x and y. We'll generally be able to give two very different styles of proof that $x = y$. One type of proof uses algebra, typically using the definition of $\binom{n}{k}$ and algebraic manipulations to show that x and y are equal. The other type of proof will be a more story-based approach, called a *combinatorial proof*, where we argue that $x = y$ by explaining how x and y are really just two ways of looking at the same set:

Definition 9.3 (Combinatorial Proof)
A combinatorial proof *establishes that two quantities x and y are equal by defining a set S and proving that $|S| = x$ and $|S| = y$ by counting $|S|$ in two different ways.*

The algebraic approach is perhaps apparently more straightforward, but combinatorial proofs can be more fun. Here's a first example:

Theorem 9.18 (A symmetry in choosing)
For any positive integer n and any integer $k \in \{0, 1, \ldots, n\}$, we have $\binom{n}{k} = \binom{n}{n-k}$.

Proof #1 of $\binom{n}{k} = \binom{n}{n-k}$, via algebra. We simply follow our noses through the definition:

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{k! \cdot (n-k)!} && \text{definition of combinations} \\
&= \frac{n!}{(n-k)! \cdot k!} && \text{commutativity of multiplication} \\
&= \frac{n!}{(n-k)! \cdot (n-(n-k))!} && \text{antisimplification: } k = n-(n-k) \\
&= \binom{n}{n-k}. && \text{definition of combinations}
\end{aligned}$$

□

Here is a second proof of Theorem 9.18—this time a combinatorial proof. The basic idea is that we will construct a set S such that we can prove that $|S| = \binom{n}{k}$ and we can prove that $|S| = \binom{n}{n-k}$. (Thus we can conclude $\binom{n}{k} = \binom{n}{n-k}$.)

Proof #2 of $\binom{n}{k} = \binom{n}{n-k}$, via a combinatorial proof: Suppose that n students submit implementations of Bubble Sort in a computer science class. The instructor has k gold stars, and he will affix a gold star to each of k different implementations. Let S be the set of ways to affix gold stars. Here are two ways of computing $|S|$:

- First, we claim that $|S| = \binom{n}{k}$. Specifically, the instructor will choose k of the n submissions and affix gold stars to the k chosen elements. There are $\binom{n}{k}$ ways of doing so.
- Second, we claim that $|S| = \binom{n}{n-k}$. Specifically, the instructor will choose $n-k$ of the n submissions that he will *not* adorn with gold stars. The remaining unchosen submissions will be adorned. There are $\binom{n}{n-k}$ ways of choosing the unadorned submissions.

But $|S|$ is the same regardless of how we count it! So $\binom{n}{k} = |S| = \binom{n}{n-k}$ and the theorem follows. □

(Another way to think about the combinatorial proof: an n-bit string with k ones *is* an n-bit string with $n-k$ zeros; the number of choices for where the ones go is identical to the number of choices for where the zeros go.)

A combinatorial proof requires creativity—*what set S should we consider?*—but the argument that the proof is correct is generally comparatively straightforward. Thus the challenge in proving an identity with a combinatorial proof is a challenge of narrative: we must find a story in which the two sides of the equation both capture the set described by that story.

Problem-solving tip: The hard part in a combinatorial proof is coming up with a story that explains both sides of the equation. Understanding what the more complicated side of the equation means is often a good place to start.

Pascal's Identity

Here's another example claim with both algebraic and combinatorial proofs:

Theorem 9.19 (Pascal's Identity)
For any integer $n \geq 1$ and any $k \in \{0, 1, \ldots, n\}$:

$$\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}.$$

Pascal's identity is named after Blaise Pascal, a 17th-century French mathematician. The programming language Pascal was also named in his honor.

Proof #1 of Pascal's Identity (algebra). Observe that if $k = 0$ or $k = n$, the identity follows immediately: by definition, we have $\binom{n}{0} = 1 = 1 + 0 = \binom{n-1}{0} + \binom{n-1}{-1}$ and similarly $\binom{n}{n} = 1 = 0 + 1 = \binom{n-1}{n} + \binom{n-1}{n-1}$. For the non-boundary cases, we'll manipulate the left-hand side until it's equal to the right-hand side:

$$\begin{aligned}
&\binom{n-1}{k} + \binom{n-1}{k-1} \\
&= \frac{(n-1)!}{k! \cdot (n-1-k)!} + \frac{(n-1)!}{(k-1)! \cdot (n-k)!} && \text{definition of combinations} \\
&= \frac{(n-1)!}{k! \cdot (n-1-k)!} \cdot \frac{n-k}{n-k} + \frac{(n-1)!}{(k-1)! \cdot (n-k)!} \cdot \frac{k}{k} && \text{multiplying by } 1 = \tfrac{x}{x} \\
&= \frac{(n-1)! \cdot (n-k)}{k! \cdot (n-k)!} + \frac{(n-1)! \cdot k}{k! \cdot (n-k)!} && (k-1)! \cdot k = k! \text{ and } (n-1-k)! \cdot (n-k) = (n-k)! \\
&= \frac{(n-1)! \cdot [(n-k)+k]}{k! \cdot (n-k)!} && \text{factoring} \\
&= \frac{n!}{k! \cdot (n-k)!} && n-k+k = n, \text{ and } (n-1)! \cdot n = n! \\
&= \binom{n}{k}. \qquad \square && \text{definition of combinations}
\end{aligned}$$

Proof #2 of Pascal's Identity (combinatorial proof). For the case of $k = 0$ or $k = n$, the argument is the same as in Proof #1. Otherwise, consider a set of $n \geq 1$ employees, one of whom is named Babbage. How many ways can we select a subset of k different employees? Here are two different ways of counting the number of these subsets:

- We choose k of the n employees. There are $\binom{n}{k}$ ways to do so.
- We decide whether to include Babbage, and then fill in the rest of the team:
  - If we pick Babbage, we need to pick $k-1$ further employees from the $n-1$ other (non-Babbage) employees; thus there are $\binom{n-1}{k-1}$ ways to select a team that includes Babbage.
  - If we don't pick Babbage, we pick all k employees from the $n-1$ others; thus there are $\binom{n-1}{k}$ ways to select a team that does not include Babbage.

  By the Sum Rule, there are therefore $\binom{n-1}{k-1} + \binom{n-1}{k-1}$ ways to choose a team.

Because we've counted the cardinality of one set in two different ways, the two sizes must be equal. Therefore $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ and the theorem follows. □

Taking it further: World War II was perhaps the first major historical moment in which computer science—and, by the end of the war, the computer—was central to the story. The German military used a complex cryptographic device called the *Enigma machine* for encryption of military communication during the war. The Enigma machine, which was partially mechanical and partially electrical, had a large (though not unfathomably large) set of possible physical configurations, each corresponding to a different cryptographic "key." Among the first applications of an electronic computer—and the reason that one of the first computers was designed and built in the first place—was in breaking these codes, in part by exhaustively exploring the set of possible keys. As such, understanding the number of different keys in the system (a counting problem!) was crucial to the Allies' success in breaking the Enigma code. For more, see the discussion on p. 960.

9.4.3 *The Binomial Theorem*

The quantity $\binom{n}{k}$ is sometimes called a *binomial coefficient*, for reasons that we'll see in this section. First, a reminder: the product of two binomials $(x+y)$ and $(a+b)$ is $xa+xb+ya+yb$. (You may have once learned the "FOIL" mnemonic for the terms of the product: first = xa; outer = xb; inner = ya; and last = yb.) Thus when we square $x+y$—that is, multiply it by itself—we get

A *binomial* (Latin *bi* "two" + *nom* "name") is a special kind of polynomial—*poly* "many" + *nom* "name"—that has precisely two terms.

$$(x+y)\cdot(x+y) \;=\; xx+xy+yx+yy \;=\; 1\cdot x^2 + 2\cdot xy + 1\cdot y^2.$$

Observe that the three coefficients of these terms, in order, are $\langle 1,2,1\rangle = \langle \binom{2}{0}, \binom{2}{1}, \binom{2}{2}\rangle$. The *binomial theorem* is a general statement of this pattern: when we multiply out the expression $(x+y)^n$, the coefficient of the $x^k y^{n-k}$ term is $\binom{n}{k}$:

Theorem 9.20 (The Binomial Theorem)
For any $a \in \mathbb{R}$, any $b \in \mathbb{R}$, and any $n \in \mathbb{Z}^{\geq 0}$, we have

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}.$$

Before we prove the binomial theorem, let's start with some intuition about *why* these coefficients arise. For example, let's compute $(x+y)^4 = (x+y)\cdot(x+y)\cdot(x+y)\cdot(x+y)$, without doing any simplification by combining like terms:

$$\begin{aligned}
&(x+y)\cdot(x+y)\cdot(x+y)\cdot(x+y)\\
&= (xx+xy+yx+yy)\cdot(x+y)\cdot(x+y)\\
&= (xxx+xyx+yxx+yyx+xxy+xyy+yxy+yyy)\cdot(x+y)\\
&= xxxx+xyxx+yxxx+yyxx+xxyx+xyyx+yxyx+yyyx\\
&\quad +xxxy+xyxy+yxxy+yyxy+xxyy+xyyy+yxyy+yyyy.
\end{aligned}$$

Every term of the resulting expression consists of 4 multiplicands, one from each of the 4 copies of $(x+y)$. How many of these 16 terms contain, say, 2 copies of x and 2 copies of y? There are 6—$yyxx$, $xyyx$, $yxyx$, $xyxy$, $yxxy$, and $xxyy$—which is just the

number of elements of $\{x,y\}^4$ that contain precisely two copies of x. While the symbols are different, it's easy to see that this quantity is precisely the number of elements of $\{0,1\}^4$ that contain precisely two ones—which is just $\binom{4}{2}$.

We will prove the Binomial Theorem in generality in a moment, but to build a little bit of intuition for the proof, let's look at a special case first:

Problem-solving tip: When you're asked to solve a problem for a general value of n, one good way to get started is to try to solve it for a specific small value of n—and then try to generalize your solution to an arbitrary n. It's often easier to generalize from a particular n to a general n than to give a fully generally answer "from scratch."

Example 9.47 (The coefficients of $(x+y)^3$)
We're going to show that $(x+y)^3 = x^3+3x^2y+3xy^2+y^3$ *in the same style that we'll use in the full proof of the Binomial Theorem.* We'll start with the observation, made previously, that $(x+y)^2 = x^2+2xy+y^2 = \binom{2}{0}x^2+\binom{2}{1}xy+\binom{2}{2}y^2$. A key step will make use of Theorem 9.19 to move from the coefficients of $(x+y)^2$ to the coefficients of $(x+y)^3$.

$$\begin{aligned}(x+y)^3 &= (x+y)\cdot(x+y)^2\\ &= (x+y)\cdot\left[\binom{2}{0}x^2+\binom{2}{1}xy+\binom{2}{2}y^2\right]\\ &= \underbrace{\binom{2}{0}x^3+\binom{2}{1}x^2y+\binom{2}{2}xy^2}_{x\cdot\left(\binom{2}{0}x^2+\binom{2}{1}xy+\binom{2}{2}y^2\right)} + \underbrace{\binom{2}{0}x^2y+\binom{2}{1}xy^2+\binom{2}{2}y^3}_{y\cdot\left(\binom{2}{0}x^2+\binom{2}{1}xy+\binom{2}{2}y^2\right)}\end{aligned}$$

which, collecting like terms, simplifies to

$$(x+y)^3 = \binom{2}{0}x^3+\left[\binom{2}{1}+\binom{2}{0}\right]x^2y+\left[\binom{2}{2}+\binom{2}{1}\right]xy^2+\binom{2}{2}y^3.$$

By Theorem 9.19, we have that $\binom{2}{1}+\binom{2}{0} = \binom{3}{1}$ and $\binom{2}{2}+\binom{2}{1} = \binom{3}{2}$, so

$$(x+y)^3 = \binom{2}{0}x^3+\binom{3}{1}x^2y+\binom{3}{2}xy^2+\binom{2}{2}y^3$$

Because $\binom{n}{n} = 1$ and $\binom{n}{0} = 1$ for any n, we have that $\binom{2}{0} = \binom{3}{0}$ and $\binom{2}{2} = \binom{3}{3}$, and thus

$$\begin{aligned}(x+y)^3 &= \binom{3}{0}x^3+\binom{3}{1}x^2y+\binom{3}{2}xy^2+\binom{3}{3}y^3\\ &= x^3+3x^2y+3xy^2+y^3.\end{aligned}$$

□

The combination notation can sometimes obscure the structure of the proof; for further intuition, here is what this proof looks like, without the notational overhead:

$$\begin{aligned}(x+y)^3 &= (x+y)\cdot(x+y)^2\\ &= (x+y)\cdot(x^2+2xy+y^2)\\ &= (x^3+2x^2y+xy^2)+(x^2y+2xy^2+y^3)\\ &= x^3+(2+1)x^2y+(1+2)xy^2+y^3\\ &= x^3+3x^2y+3xy^2+y^3.\end{aligned}$$

Proof of the Binomial Theorem

We're now ready to give a proof of the general form of the Binomial Theorem. Our

proof will use mathematical induction on the exponent, and the structure of the inductive case of the proof will precisely mimic that of Example 9.47.

Proof of Binomial Theorem. Let a and b be arbitrary real numbers. We wish to prove that, for any integer $n \geq 0$,

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}.$$

We proceed by induction on n.

The base case ($n = 0$) is straightforward: anything to the 0th power is 1, so in particular $(a+b)^0 = 1$. And $\sum_{i=0}^{0} \binom{0}{i} a^i b^{0-i} = \binom{0}{0} \cdot 1 \cdot 1 = 1$.

For the inductive case ($n \geq 1$), we assume the inductive hypothesis $(a+b)^{n-1} = \sum_{i=0}^{n-1} \binom{n-1}{i} a^i b^{n-1-i}$. We must prove that $(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}$. Our proof echoes the structure of Example 9.47:

$$\begin{aligned}
(a+b)^n &= (a+b) \cdot (a+b)^{n-1} && \text{definition of exponentiation} \\
&= (a+b) \cdot \sum_{i=0}^{n-1} \binom{n-1}{i} a^i b^{n-1-i} && \text{inductive hypothesis} \\
&= a \cdot \left[\sum_{i=0}^{n-1} \binom{n-1}{i} a^i b^{n-1-i} \right] + b \cdot \left[\sum_{i=0}^{n-1} \binom{n-1}{i} a^i b^{n-1-i} \right] && \text{distributing the multiplication} \\
&= \left[\sum_{i=0}^{n-1} \binom{n-1}{i} a^{i+1} b^{n-1-i} \right] + \left[\sum_{i=0}^{n-1} \binom{n-1}{i} a^i b^{n-i} \right] && \text{distributing the multiplication, again} \\
&= \left[\sum_{j=1}^{n} \binom{n-1}{j-1} a^j b^{n-j} \right] + \left[\sum_{i=0}^{n-1} \binom{n-1}{i} a^i b^{n-i} \right]. && \text{reindexing the first summation } (j := i+1)
\end{aligned}$$

By separating out the $i = 0$ and $j = n$ terms from the two summations, and then combining like terms, we have

$$\begin{aligned}
(a+b)^n &= \left[\sum_{j=1}^{n-1} \binom{n-1}{j-1} a^j b^{n-j} \right] + \left[\sum_{i=1}^{n-1} \binom{n-1}{i} a^i b^{n-i} \right] + \binom{n-1}{n-1} a^n b^{n-n} + \binom{n-1}{0} a^0 b^{n-0} \\
&= \left[\sum_{j=1}^{n-1} \left(\binom{n-1}{j-1} + \binom{n-1}{j} \right) a^j b^{n-j} \right] + \binom{n-1}{n-1} a^n b^{n-n} + \binom{n-1}{0} a^0 b^{n-0}.
\end{aligned}$$

Applying Theorem 9.19 to substitute $\binom{n}{j}$ for $\binom{n-1}{j-1} + \binom{n-1}{j}$ and using the fact that $\binom{n-1}{n-1} = 1 = \binom{n}{n}$ and $\binom{n-1}{0} = 1 = \binom{n}{0}$, we have

$$\begin{aligned}
(a+b)^n &= \left[\sum_{j=1}^{n-1} \binom{n}{j} a^j b^{n-j} \right] + \binom{n-1}{n-1} a^n b^{n-n} + \binom{n-1}{0} a^0 b^{n-0} && \binom{n}{j} = \binom{n-1}{j-1} + \binom{n-1}{j} \\
&= \left[\sum_{j=1}^{n-1} \binom{n}{j} a^j b^{n-j} \right] + \binom{n}{n} a^n b^{n-n} + \binom{n}{0} a^0 b^{n-0} && \binom{n-1}{n-1} = 1 = \binom{n}{n} \text{ and } \binom{n-1}{0} = 1 = \binom{n}{0} \\
&= \left[\sum_{j=0}^{n} \binom{n}{j} a^j b^{n-j} \right], && \text{incorporating the } j = 0 \text{ and } j = n \text{ terms back into the summation}
\end{aligned}$$

which proves the theorem. □

9.4.4 *Pascal's Triangle*

Much of this section has been devoted to understanding the binomial coefficients, through the Binomial Theorem and through combinatorial proofs of a number of their other properties. We'll close our discussion of binomial coefficients with a visual representation of these quantities, called *Pascal's triangle.* Pascal's triangle arranges the binomial coefficients in a classical and very useful way: the nth row of Pascal's triangle consists of all of the $n+1$ binomial coefficients $\binom{n}{0}, \binom{n}{1}, \cdots, \binom{n}{n}$, in order. Figure 9.33 shows the first nine rows of Pascal's triangle:

Like Pascal's identity, Pascal's triangle is named after the 17th-century French mathematician Blaise Pascal.

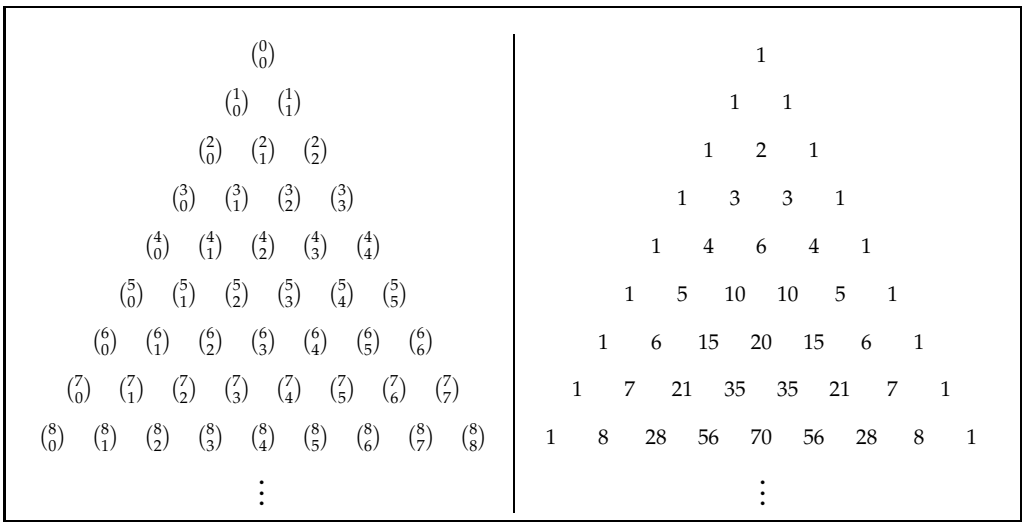


Figure 9.33: The first several rows of Pascal's triangle, in both "choose" notation and in numerical form.

Many of the properties of the binomial coefficients that we've established previously can be seen by looking at patterns visible in Pascal's triangle—as can some others that we'll prove here, or that you'll prove in the exercises.

For example, Figure 9.34 gives visualizations of two properties that we've already proven. Theorem 9.18 states that $\binom{n}{k} = \binom{n}{n-k}$; this theorem is reflected by the fact that the numerical values of Pascal's triangle are symmetric around a vertical line drawn down through the middle of the triangle. And Theorem 9.19 ("Pascal's Identity"), which states that $\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}$, is illustrated by the fact that each entry in Pascal's triangle is the sum of the two elements immediately above it (up-and-left and up-and-right).

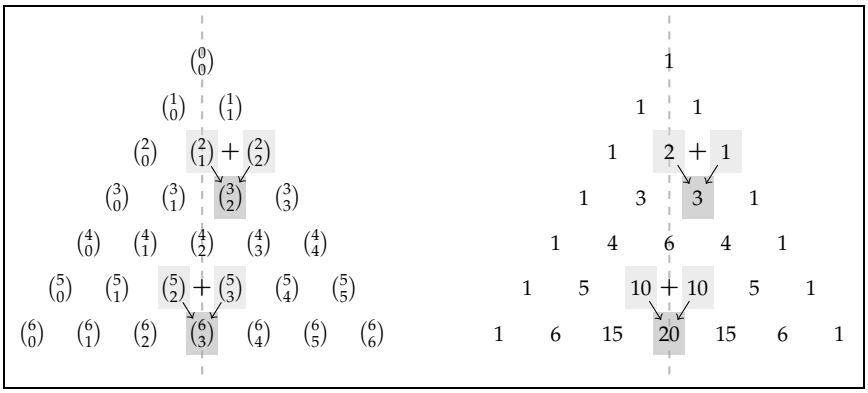


Figure 9.34: Theorems 9.18 and 9.19 reflected in Pascal's triangle.

There are many other notable properties of the binomial coefficients, many of which we can see more easily by looking at Pascal's triangle. Here's one example; a number of other properties are left to you in the exercises. Let's look at the *row sums* of Pas-

cal's triangle—that is, computing $\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n}$ for different values of n. (See Figure 9.35.)

From calculating the row sum for a few small values of n, we see that the nth row appears to have value equal to 2^n. (Incidentally, the sum of the *squares* of the numbers in any particular row in Pascal's triangle also has a special form, as you'll see in Exercise 9.170.) Indeed, the power-of-two pattern for the row sums of Pascal's triangle that we observe in Figure 9.35 holds for arbitrary n—and we'll prove this theorem here, in several different ways.

$$\begin{array}{rl} 1 & =1 \\ 1+1 & =2 \\ 1+2+1 & =4 \\ 1+3+3+1 & =8 \\ 1+4+6+4+1 & =16 \\ 1+5+10+10+5+1 & =32 \\ 1+6+15+20+15+6+1 & =64 \\ \vdots & \end{array}$$

Figure 9.35: The row sums of Pascal's triangle.

Theorem 9.21 (Sum of a row of Pascal's triangle)
$\sum_{i=0}^{n} \binom{n}{i} = 2^n$.

Proof #1 (algebraic/inductive) [sketch]. We can gain a bit of intuition for this claim from Theorem 9.19 (Pascal's Identity): each entry $\binom{n}{k}$ in the nth row is added into *exactly two* entries in the $(n+1)$st row, namely $\binom{n+1}{k}$ and $\binom{n+1}{k+1}$. Therefore the values in row #n of Pascal's triangle each contribute *twice* to the values in row #$(n+1)$, and therefore the $(n+1)$st row's sum is twice the sum of the nth row. This intuition can be turned into an inductive proof, which you'll give in Exercise 9.169. □

Proof #2 (combinatorial). Let $S := \{1, 2, \ldots, n\}$ be a set with n elements. Let's count the number of subsets of S in two different ways.

On one hand, there are 2^n such subsets: there is a bijection between subsets of S and $|S|$-bit strings. (See Lemma 9.10.)

On the other hand, let's account for the subsets of S by *first* choosing a size k of the subset, and then counting the number of subsets of that size. By the Sum Rule, the total number of subsets of S is exactly

$$\sum_{k=0}^{n} (\text{the number of subsets of } S \text{ of size } k).$$

By definition, there are exactly $\binom{n}{k}$ subsets of size k. Therefore the total number of subsets is $\sum_{k=0}^{n} \binom{n}{k}$. Thus $2^n = \sum_{k=0}^{n} \binom{n}{k}$. □

Proof #3 (making clever use of the Binomial Theorem). We'll start from the right-hand side of the theorem statement, and begin with a completely unexpected, but obviously true, antisimplification:

$$\begin{aligned} 2^n &= (1+1)^n && \text{obviously } 2 = 1+1\text{; therefore } 2^n = (1+1)^n \\ &= \sum_{i=0}^{n} \binom{n}{i} 1^i 1^{n-i} && \text{binomial theorem} \\ &= \sum_{i=0}^{n} \binom{n}{i}. && 1^k = 1 \text{ for any value of } k \end{aligned}$$ □

You'll explore some of the many other interesting and useful properties of Pascal's triangle, and of the binomial coefficients in general, in the exercises.

Computer Science Connections

Brute Force Algorithms and Dynamic Programming

In an *optimization problem*, we're given a set S of valid solutions and some measure of quality $f : S \to \mathbb{R}$, and asked to compute the element $x \in S$ that's the best according to f. (That is, we want to find the $x \in S$ that optimizes $f(x)$.) Two examples are shown in Figure 9.36: the *traveling salesman problem (TSP)*—the problem solved every day by delivery drivers, who have to visit a given list of addresses and return to the depot—and the *cheapest vertical seam (CVS)* problem, which arises in a remarkable computer graphics application.[5] (For an example of the latter problem, see Figure 9.37.)

For both TSP and CVS, there are very simple, but very slow, *brute-force algorithms* that solve the problem by computing the list of all possible solutions (all orderings of the cities; all top-to-bottom paths) and identifying the best of these possible solutions. It's by now a reasonably straightforward counting exercise to show that there are $n!$ orderings and between $2^n \cdot n$ and $3^n \cdot n$ paths (it takes some work to avoid counting paths that fall off the left/right edges of the grid). These running times are unimpressive—even n around 100 would require decades of computing time—and this is, more or less, the best known algorithm for TSP! (See p. 326.)

But we can do better for CVS, with another view of the problem. Given a grid G, define $best(i,j)$ as the *cost of the cheapest path from grid cell $\langle i,j\rangle$ to the bottom of the grid.* Then we can solve the CVS problem using a recursive algorithm that computes $best(i,j)$ for every cell $\langle i,j\rangle$, as in Figure 9.38. Unfortunately, this algorithm is just as slow as the brute-force approach: to compute $\textbf{best}(i,j)$, we make three recursive calls, at least two of which remain inside the grid. Thus the running time $T(i)$ to find $\textbf{best}(n-i,j)$ with i rows beneath cell $\langle i,j\rangle$ is given by the recurrence $T(1) = 1$ and $T(i) \geq 2T(i-1)+1$—which satisfies $T(n) \geq 2^n$, just as slow as before.

But a key algorithmic observation is that the number of *different* cells in the grid is much smaller—only n^2 different cells! So, while the algorithm in Figure 9.38 does take $\Omega(2^n)$ time, it actually "should" require only $\Theta(n^2)$ time—*as long as we avoid recomputing* $\textbf{best}(i,j)$ *multiple times for the same value of* $\langle i,j\rangle$*!* Once we've figured out $\textbf{best}(3,7)$ (because we needed that value to figure out $\textbf{best}(4,6)$), we don't bother recomputing $\textbf{best}(3,7)$ when we need it again (while we're computing $\textbf{best}(4,7)$ and $\textbf{best}(4,8)$); instead, we just remember the value and reuse it without doing any further computation.

The most straightforward way to implement this basic idea is called *memoization*: we build a data structure in which we check to see whether we've already stored the value of $\textbf{best}(i,j)$ before computing the value via the three recursive calls, and we always add all values we compute to the data structure before returning them. A slightly more efficient way of implementing this idea is called *dynamic programming*, where we transform this recursive solution into one using loops—and build up the values of $\textbf{best}(i,j)$ from the bottom up. (See Figure 9.39).

In general, dynamic programming is an algorithmic design technique that can save us a massive amount of computation—as long as the number of *different* problems encountered in the recursive solution is small.

Traveling Salesman Problem (TSP):
Input: A set C of n cities, and distance function d giving the driving time between any two cities.
Output: An ordering π of C such that the sum of the driving times $\sum_i d(\pi_i, \pi_{i+1})$ is minimized.

Cheapest Vertical Seam (CVS):
Input: An n-by-n grid of integers.
Output: A path from the top row to the bottom row, moving in direction $\{\swarrow, \downarrow, \searrow\}$ at each step, such that the sum of the integers along the path is minimized.

Figure 9.36: Two problems.

[5] Shai Avidan and Ariel Shamir. Seam carving for content-aware image resizing. In *ACM SIGGRAPH*, 2007.

Figure 9.37: A small example of CVS.

best(i,j): *// Assume $G_{1\ldots n,1\ldots n}$ is given.*
1: **if** $i = n$ **then**
2: **return** $G_{i,j}$ *(in the last row)*
3: **else if** $j \leq 0$ or $j \geq n$ **then**
4: **return** $+\infty$ *(outside the grid)*
5: **else**
6: **return** the minimum of:

$$\begin{cases} G_{i,j} + \textbf{best}(i+1, j-1), \\ G_{i,j} + \textbf{best}(i+1, j), \\ G_{i,j} + \textbf{best}(i+1, j+1). \end{cases}$$

Figure 9.38: A recursive algorithm for CVS. (To solve CVS itself, return the smallest $\textbf{best}(1,j)$ for every $1 \leq j \leq n$.)

CVS$(G_{1\ldots n,1\ldots n})$:
1: **for** $j := 1, \ldots, n$:
2: $T[n,j] := G_{i,j}$
3: **for** $i := n-1, \ldots, 1$:
4: **for** $j := 1, \ldots, n$:
5: $T[i,j] :=$ the minimum of:
(Treat $T[\cdot, j] = \infty$ if j out of range.)

$$\begin{cases} G_{i,j} + T[i+1, j-1], \\ G_{i,j} + T[i+1, j], \\ G_{i,j} + T[i+1, j+1]. \end{cases}$$

6: **return** $\min_j T[1,j]$.

Figure 9.39: A dynamic programming algorithm for CVS.

Computer Science Connections

The Enigma Machine and the First Computer

The Enigma machine was a physical cryptographic device used by the Germans during World War II to communicate between German high command and their military units in the field. The basic structure of the machine involved *rotors* and *cables*. A *rotor* was a 26-slot physical wheel that encoded a permutation π; when the wire corresponding to input i is active, the output wire corresponding to π_i is active. A *plugboard* allowed an arbitrary matching of keys on the keyboard to the inputs to the rotors—a *cable* was what actually connected a key to the first rotor. (The machine did not require any cables in the plugboard; if there was no cable, then the key pressed was what went into the rotor in the first place.) The basic encryption in the Enigma machine proceeded as follows:

1. The user pressed a key, say `A`, on the keyboard. If there was a cable from the `A` key, then the key would be remapped to the other end of the cable; otherwise the procedure proceeded using the `A`. (See Figure 9.40.)
2. The pressed key was permuted by rotor #1; the output of rotor #1 was permuted by rotor #2; the output of rotor #2 was permuted by rotor #3. (See Figure 9.41.) The output of rotor #3 was "reflected" by a fixed permutation, and then the reflector's output pass through the three rotors, in reverse order and backward: the output of the reflector was permuted by rotor #3, then by #2, and then by #1. (See Figure 9.42.)
3. A light corresponding to the output of rotor #1, passed through the plugboard cable if present, lights up; the illuminated letter is the encoding.

The tricky part is that the rotors rotate by one notch when the key is pressed, so that the encoding changes with every keypress.

The "secret key" that the two communicating entities needed to agree upon was which rotors to use in which order ($5 \cdot 4 \cdot 3 = 60$; there were 5 standard rotors in an Enigma), what the initial position of the rotors should be ($26^3 = 17{,}576$), and what plugboard matching to use ($\frac{26!}{13! \cdot 2^{13}} \approx 8 \times 10^{12}$ choices if all 26 letters were matched; see Example 9.32). Interestingly, almost all of the complexity came from the plugboards.

Perhaps surprisingly, the fact that there were so many possible settings for the Enigma led to the invention of one of the first programmable computers, by Alan Turing at Bletchley Park, in England, during the war. Turing built a machine that could test many of these configurations, by brute force. (If there were fewer possibilities, it could have been cracked by hand; if there were many more, it couldn't have been cracked by brute force.) Turing and his team developed a device called the Bombe to exhaustively try to compute the shared German secret key—each day!

Many other cryptographic tricks related to the way the Enigma was being used were also part of the analysis. For example, the construction of the device meant that no letter could encrypt to itself; this fact was exploited in the analysis. Another crucial part of the code breaking was a *known plaintext attack* on the Enigma: the British also used knowledge of what the Germans tended to communicate (like weather reports) to narrow their search.

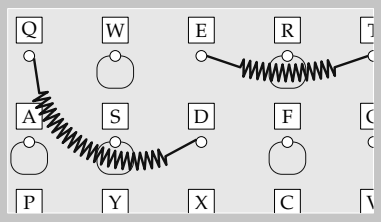


Figure 9.40: The effect of the plugboard. Each of the 26 keys is either mapped to itself (like `W` here), or is matched with another key (like `Q` ↔ `D` here). Pressing an unmatched key x yields x itself; pressing a matched key x yields whatever letter is matched to x.

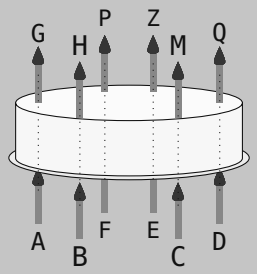


Figure 9.41: The effect of a rotor. Each rotor encodes a permutation of the letters; when the input letter i comes into the rotor, the output π_i comes out. (Here, for example, an input `B` turns into an output of `H`.) After each keypress, the top portion of the rotor would rotate by one notch, so that `B` would now turn into `G`.

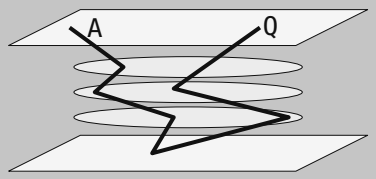


Figure 9.42: The Enigma machine's operation. The operator types an `A`, which (after going through the plugboard) is permuted by rotor #1, rotor #2, rotor #3, the fixed permutation of the machine, rotor #3, rotor #2, and rotor #1. It then (after passing through the plugboard) lights up the output, `Q`. The rotors advance by one notch, and encoding continues with the next letter.

9.4.5 Exercises

For two strings x and y, let's call a shuffle *of x and y any interleaving of the letters of the two strings (that maintains the order of the letters within each string, but may repeatedly alternate between blocks of x letters and blocks of y letters). For example, the words* ALE *and* LID *can be shuffled into* ALLIED *or* ALLIDE *or* ALLIDE *or* LIDALE*. How many* different *strings can be produced as shuffles of the following pairs of words?*

9.121 BACK and FORTH

9.122 DAY and NIGHT

9.123 SUPPLY and DEMAND

9.124 LIFE and DEATH

9.125 ON and ON

9.126 OUT and OUT

9.127 (*programming required*) Write a program, in a language of your choice, that computes all shuffles of two given words x and y. A recursive approach works well: a shuffle consists either of the first character of x followed by a shuffle of $x_{2\ldots|x|}$ and y, or the first character of y followed by a shuffle of x and $y_{2\ldots|y|}$. (Be sure to eliminate any duplicates from your resulting list.)

The next few questions ask you to think about shuffles of generic strings, instead of particular words. (Assume that the alphabet is an arbitrarily large set—you are not restricted to the 26 *letters in English.) Consider two strings x and y, and let $n := |x| + |y|$ be the total number of characters between them. Note that the number of distinct shuffles of x and y may depend* both *on the lengths of x and y* and *on the particular strings themselves; for example, if some letters are shared between or within the two strings, there may be fewer possible shuffles.*

9.128 In terms of n, what is the *maximum* possible number of different shuffles of x and y?

9.129 In terms of n, what's the *minimum* possible number of distinct shuffles of x and y?

9.130 What is the largest possible number of different shuffles of *three* strings of length a, b, and c?

9.131 How many 42-bit strings have exactly 16 ones?

9.132 How many 23-bit strings have at most 3 ones? (The coincidental arithmetic structure of the answer actually turns out to be helpful for error-correcting codes; see Exercise 4.30.)

9.133 How many 32-bit strings have a number of ones within ± 2 of the number of zeros?

9.134 The set of 64-bit strings with k ones is largest for $k = 32$. What's the smallest m for which

$$|\{\text{the number of 64-bit strings with} \leq m \text{ ones}\}| \geq |\{\text{the number of 64-bit strings with 32 ones}\}|?$$

9.135 What is the smallest even integer n for which the following statement is true? If we flip an unbiased coin n times, as in Example 9.41, the probability that we get exactly $\frac{n}{2}$ heads is less than 10%.

A bridge hand consists of 13 *cards from a standard* 52*-card deck, with* 13 *ranks* (2 *through ace*) *and* 4 *suits* (♣, ♢, ♡, *and* ♠). *(That is, the cards in the deck are $\{2, 3, \ldots, 10, J, Q, K, A\} \times \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$.) How many different bridge hands are there that meet the following conditions?*

9.136 A *void in spades:* a 13-card hand that contains only cards from the suits ♣, ♢, and ♡.

9.137 A *singleton in hearts:* exactly one of the 13 cards comes from the suit ♡.

9.138 All four kings.

9.139 No queens at all.

9.140 Exactly two jacks.

9.141 Exactly two jacks and exactly two queens.

9.142 A bridge hand has *high honors* if it contains the five highest-ranked cards $\{10, J, Q, K, A\}$ in the same suit. How many bridge hands have high honors? (*Warning: be careful about double counting!*)

Many bridge players evaluate their hands by the following system of points*. First, give yourself one* high-card point *for a jack, two for a queen, three for a king, and four for an ace. Furthermore, give yourself three* distribution points *for each void (a suit in which you have zero cards), two points for a singleton (a suit with one card), and one point for a doubleton (a suit with two cards).*

9.143 How many bridge hands have a high-card point count of zero?

9.144 How many bridge hands have a high-card point count of zero *and* a distribution point count of zero? What fraction of all bridge hands is this?

How many ways are there to choose 32 *out of* 202 *options if . . .*

9.145 . . . repetition is allowed and order matters?

9.146 . . . repetition is forbidden and order matters?

9.147 . . . repetition is allowed and order doesn't matter?

9.148 . . . repetition is forbidden and order doesn't matter?

The first 10 *prime numbers are* $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\}$. *How many different integers have exactly* ...

9.149 ... 5 prime factors (all from this set), where all of these factors are different?

9.150 ... 5 prime factors (all from this set)? (Note that $32 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$ is an example.)

How many different integers have exactly 10 *prime factors* ...

9.151 ... all of which come from the set of the first 20 prime numbers?

9.152 ... all of which come from the set of the first 20 prime numbers, and where all 10 of these factors are different from each other?

Suppose that we have two sequences $\langle x_1, x_2, \ldots, x_n \rangle$ *and* $\langle y_1, y_2, \ldots, y_{2n} \rangle$ *of data points—perhaps representing a sequence of intensities from two streams of speech. We wish to* align x *to* y *by matching up elements of* x *to elements of* y. (*For example,* y *might represent a reference stream, where we're trying to match* x *up to it.*) *We insist that each element of* x *is assigned to one and only one element of* y. (*See Figure 9.43.*)

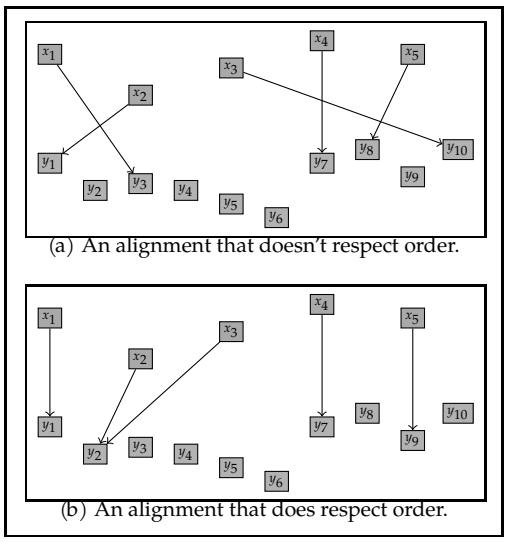
9.153 How many ways are there to assign each of the n elements of x to one of the $2n$ elements of y?

9.154 How many ways are there to assign each of the n elements of x to one of the $2n$ elements of y so that no element of y is matched to more than one element of x?

In many applications, we can only consider alignments of the elements of x *and* y *that "maintain order": that is, we can't have* x_5 *assigned to an element of* y *that comes after the element assigned to* x_6. (*If* $f : \{1, \ldots, n\} \to \{1, \ldots, 2n\}$ *represents the alignment, then we require that* $i \leq j$ *implies that* $f(i) \leq f(j)$.)

9.155 How many ways are there to assign each of the n elements of x to one of the $2n$ elements of y in a way that maintains order?

9.156 How many ways are there to assign each of the n elements of x to one of the $2n$ elements of y in a way that maintains order so that no element of y is matched to more than one element of x?



(a) An alignment that doesn't respect order.

(b) An alignment that does respect order.

Figure 9.43: An alignment between two sequences, for Exercises 9.153–9.156. (Thanks to Roni Khardon, from whom I learned a version of the exercises.)

9.157 Consider the equation $a + b + c = 202$. How many solutions are there where a, b, and c are all nonnegative integers?

9.158 How many different solutions are there to the equation $a + b + c + d + e = 8$, where all of $\{a, b, c, d, e\}$ have to be nonnegative integers?

9.159 What about for $a + b + c + d + e = 88$, again where all variables must be nonnegative integers?

9.160 What about for $a + 2b + c = 128$, again where a, b, and c must be nonnegative integers? (*Hint: sum over the possible values of* b *and use Theorem 9.17.*)

The Association for Computing Machinery (the ACM)—a major professional society for computer scientists—puts on student programming competitions regularly. Teams of students spend a few hours working on some programming problems (of various levels of difficulty).

9.161 Suppose that, at a certain college in the midwest, there are 141 computer science majors. A programming contest team consists of 3 students. How many ways are there to choose a team?

9.162 Suppose that, at a certain programming contest, teams are given 10 problems to try to solve. When the contest begins, each of the 3 members of the team has to choose a problem to think about first. (More than one team member can think about the same problem.) How many ways are there for the 3 team members to choose a problem to think about first?

9.163 In most programming contests, teams are scored by the number of problems they correctly solve. (There are tiebreakers based on time and certain penalties.) A team can submit multiple solutions to the same problem. Suppose that a particular team has calculated that they have time to code up and submit 20 different attempted answers to the 10 questions in the contest. How many different ways can they allocate their 20 submissions across the 10 problems? (The order of their submissions doesn't matter.)

See the discussion on p. 730, or

9.164 Solve the following problem, posed by Adi Shamir in his original paper on secret sharing:[6]

> Eleven scientists are working on a secret project. They wish to lock up the documents in a cabinet so that the cabinet can be opened if and only if six or more of the scientists are present. What is the smallest number of locks needed? What is the smallest number of keys to the locks each scientist must carry?

[6] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

9.165 In machine learning, we try to use a collection of *training data*—for example, a large collection of ⟨image, letter⟩ pairs of images of handwritten letters and the English letter that they represent—to compute a predictor that will do well on predicting answers on a set of novel *test data.* One danger in such a system is *overfitting:* we might build a predictor that's overly affected by idiosyncrasies of the training data. One way to address the risk of overfitting is a technique called *cross-validation:* we divide the training data into several subsets, and then, for each subset S, train our predictor based on $\sim S$ and test it on S. We might then average the parameters of our predictor across the subsets S. In *ten-fold cross-validation* on a n-element training set, we would split our n training examples into disjoint sets $S_1, S_2, \ldots, S_{10}$ where $|S_i| = \frac{n}{10}$.

How many ways are there to split an n-element set into disjoint subsets $S_1, S_2, \ldots, S_{10}$ of size $\frac{n}{10}$ each? (Note the order of the subsets themselves doesn't matter, nor does the order of the elements within a subset.)

9.166 Consider the set of bitstrings $x \in \{0,1\}^{n+k}$ with n zeros and k ones with the additional condition that *no ones are adjacent.* (For $n=3$ and $k=2$, for example, the legal bitstrings are `00101`, `01001`, `01010`, `10001`, `10010`, and `10100`.) Prove by induction on n that the number of such bitstrings is $\binom{n+1}{k}$.

9.167 Consider the set of bitstrings $x \in \{0,1\}^{n+k}$ with n zeros and k ones with the additional condition that *every block of ones has even length.* (For $n=3$ and $k=2$, for example, the legal bitstrings are `00011`, `00110`, `01100`, `11000`.) Prove that, for any even k, the number of such bitstrings is $\binom{n+(k/2)}{n}$.

9.168 Prove that $k \cdot \binom{n}{k} = n \cdot \binom{n-1}{k-1}$ twice, using both an algebraic and a combinatorial proof.

9.169 Using induction on n, prove Theorem 9.21—that is, prove that

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n.$$

9.170 Prove the following identity about the squares of the binomial coefficients. (For example, for $n=4$, this identity states that $\binom{4}{0}^2 + \binom{4}{1}^2 + \binom{4}{2}^2 + \binom{4}{3}^2 + \binom{4}{4}^2 = 1^2 + 4^2 + 6^2 + 4^2 + 1^2 = 70$ is equal to $\binom{8}{4}$. And, indeed, $\binom{8}{4} = \frac{8!}{4!4!} = 70$.) Use a combinatorial proof.

$$\sum_{k=0}^{n} \binom{n}{k}^2 = \binom{2n}{n}.$$

9.171 Prove the following identity by algebraic manipulation:

$$\binom{n}{m}\binom{m}{k} = \binom{n}{k}\binom{n-k}{m-k}.$$

9.172 Now prove the identity from Exercise 9.171 with a combinatorial proof. (*Hint: think about choosing a team of m people from a pool of n candidates, and picking k managers from the team that you've chosen.*)

9.173 Prove the following identity, using an algebraic, inductive, or combinatorial proof:

$$\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}.$$

Recall that $\binom{a}{b} = 0$ for any $b < 0$ or $b > a$, so many of the terms of the summation are zero. For example, for $m=3$ and $n=5$, the claim states that $\binom{6}{4} = \binom{0}{3} + \binom{1}{3} + \binom{2}{3} + \binom{3}{3} + \binom{4}{3} + \binom{5}{3} = 0+0+0+\binom{3}{3}+\binom{4}{3}+\binom{5}{3}$.

9.174 Prove the following identity about the binomial coefficients and the Fibonacci numbers (where f_i is the ith Fibonacci number), by induction on n:

$$\sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n-k}{k} = f_{n+1}.$$

9.175 Prove *van der Monde's identity*:

$$\binom{n+m}{k} = \sum_{r=0}^{k} \binom{m}{k-r} \cdot \binom{n}{r}.$$

(*Hint: suppose you have a deck of n red cards and m black cards, from which you choose a hand of k total cards.*)

A common subsequence *of two strings x and y is a string z that's a subsequence of both. A subsequence of an n-character string corresponds to a subset of $\{1, 2, \ldots, n\}$, indicating which indices are included (and which aren't). (See Exercise 9.82.) For example,* BASIC *is a common subsequence of* BRAINSICKNESS *and* BIOACOUSTICS.

9.176 Suppose that you have been asked to find the *number of common subsequences* of two n-character strings $x, y \in \Sigma^n$, by brute force. An algorithm to do so is shown in Figure 9.44(a). How many times do we execute Line 3 (testing whether $a = b$)?

9.177 Using the fact that common subsequences must have the same length, we can modify the algorithm as shown in Figure 9.44(b). Now how many times do we execute Line 4 (testing whether $a = b$)?

9.178 Using *Stirling's approximation* of the factorial function, which states that $n! \approx \sqrt{2\pi n}(n/e)^n$ (where $\pi = 3.1415\cdots$ and $e = 2.7182\cdots$), argue that Figure 9.44(b) is an improvement on Figure 9.44(a).

```
1: for each subsequence a of x:
2:     for each subsequence b of y:
3:         check if a = b
```

(a) A brute-force algorithm.

```
1: for k = 0 ... n:
2:     for each subsequence a of x of length k:
3:         for each subsequence b of y of length k:
4:             check if a = b
```

(b) A length-aware brute-force algorithm.

Figure 9.44: Two algorithms for common subsequences.

9.179 Use the Binomial Theorem to prove the following identity:

$$\sum_{k=0}^{n} (-1)^k \cdot \binom{n}{k} = 0.$$

9.180 Use the Binomial Theorem to prove the following identity:

$$\sum_{k=0}^{n} \frac{\binom{n}{k}}{2^k} = \left(\tfrac{3}{2}\right)^n.$$

9.181 In Section 9.2.2, we introduced the *Inclusion–Exclusion* rule for counting the union of 2 or 3 sets:

$$|A \cup B| = |A| + |B| - |A \cap B|$$
$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

Exercise 9.30 asked you to give a formula for a 4-set intersection, but here's a completely general solution:

$$\left|\bigcup_{i=1}^{k} A_i\right| = \sum_{i=1}^{k} \left[(-1)^{i+1} \cdot \sum_{j_1 < j_2 < \cdots < j_i} |A_{j_1} \cap A_{j_2} \cap \cdots \cap A_{j_i}|\right].$$

(Recall that $\bigcup_{i=1}^{k} A_i = A_1 \cup A_2 \cup \cdots \cup A_k$.) Argue that this formula correctly expresses the Inclusion–Exclusion Rule for any number of sets. (*Hint: figure out how many ℓ-set intersections each element x appears in. Then use the Binomial Theorem—specifically, Exercise 9.179.*)

9.182 In Example 8.4, we looked at the subset relation for a set S: that is, we defined the set of pairs

$$subset := \{\langle A, B\rangle \in \mathscr{P}(S) \times \mathscr{P}(S) : [\forall x \in S : x \in A \Rightarrow x \in B]\}.$$

For any particular set $B \in \mathscr{P}(S)$, the *number of sets A such that $\langle A, B\rangle \in subset$* is precisely $2^{|B|}$. The total number of pairs in the *subset* relation on S is thus 2^k times *the number of subsets of S of size k*, summed over all k. We've already seen that the number of subsets of S of size k is $\binom{|S|}{k}$. Thus the total number of pairs in the *subset* relation on S is

$$\sum_{k=0}^{|S|} (\text{number of subsets of } S \text{ of size } k) \cdot 2^k = \sum_{k=0}^{|S|} \binom{|S|}{k} \cdot 2^k.$$

Use the Binomial Theorem to compute a simple formula for this summation.

9.5 Chapter at a Glance

Counting is the problem of, given a potentially convoluted description of a set S, computing the cardinality of S. Our general strategy for counting is to develop techniques for counting simple sets like unions and sequences, and then to handle more complicated counting problems by "translating" them into these simple problems.

Counting Unions and Sequences

The *Sum Rule* describes how to compute the cardinality of the union of sets: if A and B are disjoint sets, then $|A \cup B| = |A| + |B|$. More generally, if the sets $A_1, A_2, \ldots, A_k$ are all disjoint, then $\left|\bigcup_{i=1}^{k} A_i\right| = \sum_{i=1}^{k} |A_i|$. If the sets A and B are not disjoint, then the Sum Rule doesn't apply. Instead, we can use *Inclusion–Exclusion* to count $|A \cup B|$. This rule states that $|A \cup B| = |A| + |B| - |A \cap B|$ for any sets A and B. For three sets,

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|.$$

To compute the cardinality of the Cartesian product of sets, we can use the *Product Rule*: for sets A and B, we have $|A \times B| = |A| \cdot |B|$. More generally, for arbitrary sets $A_1, A_2, \ldots, A_k$, we have $|A_1 \times A_2 \times \cdots \times A_k| = \prod_{i=1}^{k} |A_i|$. Applying the Product Rule to a set $S \times S \times \cdots \times S$, we see that, for any set S and any $k \in \mathbb{Z}^{\geq 1}$, we have $|S^k| = |S|^k$. If the set of options for one choice depends on previous choices, then we cannot directly apply the Product Rule. However, the basic idea still applies: the *Generalized Product Rule* says that $|S| = \prod_{i=1}^{k} n_i$ if S denotes a set of sequences of length k, where, for each choice of the first $i-1$ components of the sequence, there are exactly n_i choices for the ith component.

A *permutation* of a set S is sequence of elements from S that contains no repetitions and has length $|S|$. In other words, a permutation of S is an ordering of the elements of S. By the Generalized Product Rule, there are precisely $n! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 1$ permutations of an n-element set.

Using Functions to Count

Let A and B be arbitrary sets. We can use a function $f : A \to B$ to relate $|A|$ and $|B|$. The *Mapping Rule* says that:

- There exists a function $f : A \to B$ that's onto if and only if $|A| \geq |B|$.
- There exists a function $f : A \to B$ that's one-to-one if and only if $|A| \leq |B|$.
- There exists a function $f : A \to B$ that's a bijection if and only if $|A| = |B|$.

The Mapping Rule implies, among other things, that the power set $\mathscr{P}(S)$ of a set S has cardinality $|\mathscr{P}(S)| = 2^{|S|}$.

The *Division Rule* says the following: suppose that there exists a function $f : A \to B$ such that, for every $b \in B$, there are exactly k elements $a_1, \ldots, a_k \in A$ such that $f(a_i) = b$. Then $|A| = k \cdot |B|$. The Division Rule implies, among other things, that the number of ways to rearrange a sequence containing k different distinct elements $\{x_1, \ldots, x_k\}$,

where element x_i appears n_i times, is

$$\frac{(n_1 + n_2 + \cdots + n_k)!}{(n_1!) \cdot (n_2!) \cdot \cdots \cdot (n_k!)}.$$

The *pigeonhole principle* says that if A and B are sets with $|A| > |B|$, and $f : A \to B$, then there exist distinct a and $a' \in A$ such that $f(a) = f(a')$. That is, if there are more pigeons than holes, and we place the pigeons into the holes, then there must be (at least) one hole containing more than one pigeon.

Combinations and Permutations

Consider nonnegative integers n and k with $k \leq n$. The quantity $\binom{n}{k}$ is defined as

$$\binom{n}{k} := \frac{n!}{k! \cdot (n-k)!},$$

and is read as "n choose k." The quantity $\binom{n}{k}$ denotes the number of ways to choose a k-element subset of a set of n elements, called a *combination,* when each element can only be selected at most once and the order of the selected elements doesn't matter. The quantity $\binom{n}{k}$ is also sometimes called a *binomial coefficient.*

Depending on whether we allow the same candidate to be chosen more than once and whether we care about the order in which the candidates are chosen, there are many versions of selecting k out of a set of n candidates:

- If the order of the selected elements doesn't matter and repetition of the chosen elements is not allowed, then there are $\binom{n}{k}$ ways to choose.
- If order matters and repetition is not allowed, there are $\frac{n!}{(n-k)!}$ ways.
- If order matters and repetition is allowed, there are n^k ways.
- If order doesn't matter and repetition is allowed, there are $\binom{n+k-1}{k}$ ways.

A *combinatorial proof* establishes that two quantities x and y are equal by defining a set S and proving that $|S| = x$ and $|S| = y$ by counting $|S|$ in two different ways. We can give combinatorial proofs of the following facts about the binomial coefficients, among others:

$$\binom{n}{k} = \binom{n}{n-k} \qquad \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \qquad \sum_{i=0}^{n} \binom{n}{i} = 2^n.$$

The *binomial theorem* states that, for any $a, b \in \mathbb{R}$ and any $n \in \mathbb{Z}^{\geq 0}$,

$$(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}.$$

We can prove the binomial theorem by induction on the exponent n.

Many of the interesting properties of the binomial coefficients can be seen by looking at patterns visible in *Pascal's triangle,* which arranges the binomial coefficients so that the nth row contains the $n+1$ binomial coefficients $\binom{n}{0}, \binom{n}{1}, \cdots, \binom{n}{n}$. See Figure 9.45 for the first few rows of Pascal's triangle.

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

$$\binom{5}{0} \quad \binom{5}{1} \quad \binom{5}{2} \quad \binom{5}{3} \quad \binom{5}{4} \quad \binom{5}{5}$$

$$\binom{6}{0} \quad \binom{6}{1} \quad \binom{6}{2} \quad \binom{6}{3} \quad \binom{6}{4} \quad \binom{6}{5} \quad \binom{6}{6}$$

$$\vdots$$

Figure 9.45: The first several rows of Pascal's triangle.

Key Terms and Results

Key Terms

Counting Unions and Sequences

- Sum Rule
- Product Rule
- double counting
- Inclusion–Exclusion
- Generalized Product Rule
- permutation

Using Functions to Count

- Mapping Rule
- Division Rule
- pigeonhole principle

Combinations and Permutations

- combinations
- permutations
- $\binom{n}{k}$ / binomial coefficient
- binomial theorem
- combinatorial proof
- Pascal's triangle

Key Results

Counting Unions and Sequences

1. The Sum Rule: if the sets $A_1, A_2, \ldots, A_k$ are all disjoint, then $\left|\bigcup_{i=1}^{k} A_i\right| = \sum_{i=1}^{k} |A_i|$. The Inclusion–Exclusion Rule allows us to handle nondisjoint sets; for example, for any sets A, B we have $|A \cup B| = |A| + |B| - |A \cap B|$.
2. The Product Rule: $|A_1 \times A_2 \times \cdots \times A_k| = \prod_{i=1}^{k} |A_i|$. For any set S and any $k \in \mathbb{Z}^{\geq 1}$, we have $|S^k| = |S|^k$.
3. The Generalized Product Rule: if S is a set of sequences of length k, where, for each choice of the first $i-1$ components of the sequence, there are exactly n_i choices for the ith component, then $|S| = \prod_{i=1}^{k} n_i$.

Using Functions to Count

1. The Mapping Rule: an onto function $f : A \to B$ means $|A| \geq |B|$; a one-to-one function $f : A \to B$ means $|A| \leq |B|$; and a bijection $f : A \to B$ means $|A| = |B|$.
2. For any set S, $|\mathscr{P}(S)| = 2^{|S|}$.
3. The Division Rule: if $f : A \to B$ satisfies $|\{a \in A : f(a) = b\}| = k$ for all $b \in B$, then $|A| = k \cdot |B|$.
4. The number of ways to arrange a sequence containing elements $\{x_1, \ldots, x_k\}$, where x_i appears n_i times, is $\frac{(n_1+n_2+\cdots+n_k)!}{(n_1!)\cdot(n_2!)\cdot\cdots\cdot(n_k!)}$.
5. Pigeonhole principle: if $f : A \to B$ and $|A| > |B|$, then there exist $a, a' \neq a \in A$ such that $f(a) = f(a')$.

Combinations and Permutations

1. There are four versions of selecting k out of n candidates, depending on whether the order of the chosen elements matters and whether we can choose the same element twice. (See Figure 9.31.) The binomial coefficient $\binom{n}{k}$ denotes the number of ways to choose when repetition is forbidden and order doesn't matter (called *combinations*).
2. Some useful properties: $\binom{n}{k} = \binom{n}{n-k}$ and $\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}$ and $\sum_{i=0}^{n} \binom{n}{i} = 2^n$.
3. The binomial theorem: $(a+b)^n = \sum_{i=0}^{n} \binom{n}{i} a^i b^{n-i}$.