# Chapter 2

# Combinatorial Analysis

Our first main topic is a review of combinatorics. Combinatorics is one of the oldest branches of mathematics. Perhaps this is not surprising, given that it deals with countable objects. But for many, it is also among the more frustrating branches of mathematics. There are many ways of counting, and the difficulty of solving a combinatorial problem largely depends on *how* you count.

We will therefore adopt the approach of constructing basic combinatorial objects, such as permutations and combinations, from even more basic ones, such as products and sums. In this way, we will motivate the techniques of combinatorics using the most fundamental building blocks, allowing ourselves to fall back on these building blocks whenever we are in need of unraveling the complexities of a counting problem.

## 2.1 Product and sum rules

Two techniques at the heart of combinatorial analysis seem thoroughly trivial, but are not. They are the *product rule* and the *sum rule*.

**Definition 2.1** (Product rule)**.** *Consider choosing a pair of objects. If there are $n_1$ choices for object 1, and $n_2$ choices for object 2, then there are $n_1 n_2$ choices for the pair of objects 1 and 2.*

**Example 2.2** (Genetic code: Roberts 2.2)**.** A DNA strand is made up of string of molecules, each containing a subcomponent called a nucleotide or "base." There are four different types of bases.

- A: Adenine
- C: Cytosine
- G: Guanine
- T: Thymine

For instance, a possible string might consist of ACGTTCAGTC...

The exact sequence of bases is a recipe that determines which *amino acids* will be employed, to create the proteins that do most of the real work in living organisms.

There are 20 different types of amino acids. What is the shortest string of bases that provides enough information to encode for all 20 amino acids?

There are $n_1 = 4$ ways of choosing base #1, $n_2 = 4$ ways of choosing base #2, etc. Thus, by the product rule (applied recursively in this case), there are $4^k$ ways of choosing a $k$-tuple of bases. So we need to find the smallest integer $k$ such that $4^k \geq 20$. The result is $k = 3$.

And sure enough, in all known living cells, it is the triplets of bases that uniquely specify which amino acid is to be used.

**Definition 2.3** (Sum rule)**.** *Consider choosing a single object from one out of two collections. If there are $n_1$ choices in collection 1, and $n_2$ choices in collection 2, then there are $n_1 + n_2$ choices for the object.*

**Example 2.4** (Committee selection)**.** The university's committee on information technology needs a faculty representative from either the math or computer science department. There are 8 math faculty members and 10 CS faculty members. How many possibilities are there for this representative?

There are $n_1 = 8$ ways of choosing a math faculty member, and $n_2 = 10$ ways of choosing a CS faculty member. Thus, by the sum rule, there are $n_1 + n_2 = 18$ ways of choosing either a math or a CS faculty member.

The product and sum rules follow in a seemingly obvious way from the properties of intersections and unions of sets — so obvious that we do not even bother to call them theorems. But these rules turn out to be amazingly powerful, particularly when applied in composition with each other. Arguably, much of combinatorics is really nothing more than finding the correct repeated applications of the product and sum rules.

**Example 2.5** (Login passwords: Rosen 5.1, example 15)**.** Imagine the following rules for login passwords at a secure web site. The password must be 6 to 8 characters long, where a character is either a letter (case-insensitive) or a digit. The password must contain at least one digit. Given these constraints, how many passwords are possible?

Let

$$\begin{aligned} P &= \text{total \# of passwords} \\ P_i &= \text{total \# of passwords with exactly } i \text{ characters.} \end{aligned}$$

Then by the **sum rule**,

$$P = P_6 + P_7 + P_8.$$

Now let

$$\begin{aligned} C_i &= \text{total \# of } i\text{-character strings} \\ L_i &= \text{total \# of } i\text{-letter strings.} \end{aligned}$$

Since a password must contain at least one digit and cannot simply be a string of letters, it follows from the **sum rule** that

$$C_i = P_i + L_i.$$

There are 26 ways of generating each of $i$ letters, and 36 ways of generating each of $i$ alphanumeric characters (**sum rule** again!), so then from the **product rule**,

$$L_i = (26)^i, \qquad C_i = (36)^i.$$

It follows that

$$\begin{aligned} P &= P_6 + P_7 + P_8 \\ &= (C_6 - L_6) + (C_7 - L_7) + (C_8 - L_8) \\ &= (36^6 - 26^6) + (36^7 - 26^7) + (36^8 - 26^8) \\ &= 2{,}684{,}483{,}063{,}360 \end{aligned}$$

. . . which is probably large enough to be secure.

## 2.2 Permutations and combinations

Certain combinatorial operations are needed so frequently that it would be inefficient to have to break them down into product or sum rules each time they are needed. We now discuss some common "shortcuts" of this kind.

**Definition 2.6** (Permutation)**.** *A permutation of $n$ objects is an ordered arrangement of them.*

From a straightforward application of the product rule, the number of permutations $= n!$

**Example 2.7** (Job interviews: Roberts exercise 2.3.9)**.** 10 job candidates are called for interviews. 5 of them are told to come in the morning, and the other 5 in the afternoon. In how many possible ways can we order the interviews?

Let

$$\begin{aligned} n &= \#\text{ of possible orders} \\ n_1 &= \#\text{ of possible orders among 5 morning candidates} \\ n_2 &= \#\text{ of possible orders among 5 afternoon candidates.} \end{aligned}$$

From the **product rule**, $n = n_1 n_2$. But $n_1$ and $n_2$ are simply permutations:

$$\begin{aligned} n_1 &= 5! \\ n_2 &= 5! \end{aligned}$$

so $n = n_1 n_2 = 14{,}400$.

**Definition 2.8** ($r$-permutation)**.** *An $r$-permutation of $n$ objects is an arrangement of them, where only $r$ specific objects are ordered and the remaining objects are unordered.*

There are several ways of calculating the number of $r$-permutations. Here is one of the most concise. From the **product rule**,

$$\begin{aligned} &\#\text{ of ordered arrangements of } n \text{ objects} = \\ &\quad (\#\text{ of } r\text{-permutations}) \times (\#\text{ of ordered arrangements of remaining } n-r \text{ objects}). \end{aligned}$$

But the number of ordered arrangements is simply a permutation, so

$$\#\text{ of } r\text{-permutations} = \frac{\#\text{ of ordered arrangements of } n \text{ objects}}{\#\text{ of ordered arrangements of remaining } n-r \text{ objects}} = \frac{n!}{(n-r)!}.$$

**Definition 2.9** (Combination)**.** *A combination of $n$ objects is a selection of $r$ out of the $n$ objects, disregarding order.*

(Note that a combination is also sometimes called an $r$-combination, since like an $r$-permutation it involves $r$ out of $n$ elements.)

From the **product rule**,

$$\#\text{ of } r\text{-permutations} = (\#\text{ of combinations}) \times (\#\text{ of ordered arrangements of } r \text{ objects}),$$

so

$$\#\text{ of combinations} = \frac{n!}{r!(n-r)!} = \binom{n}{r}.$$

**Example 2.10** (Committee selection #2: Rosen 5.3, exercise 34)**.** A graduate program has 10 male and 15 female students. How many ways are there to choose a 6-member social committee, if it must have more women than men?

Let

$$\begin{aligned} n &= \#\text{ of ways of forming 6-member committee with more women then men} \\ n_i &= \#\text{ of ways of forming 6-member committee with } i \text{ women (and } 6-i \text{ men).} \end{aligned}$$

Then, from the **sum rule**,

$$n = n_4 + n_5 + n_6.$$

Now let

$$\begin{aligned} w_i &= \#\text{ of ways of choosing } i \text{ out of 15 women} \\ m_i &= \#\text{ of ways of choosing } i \text{ out of 10 men.} \end{aligned}$$

From the **product rule**,

$$n_i = w_i\, m_{6-i}.$$

But $w_i$ and $m_i$ are simply combinations,

$$w_j = \binom{15}{j}, \qquad m_j = \binom{10}{j},$$

so

$$n = \binom{15}{4}\binom{10}{2} + \binom{15}{5}\binom{10}{1} + \binom{15}{6}\binom{10}{0} = 96{,}460.$$

## 2.3 Inclusion-exclusion principle

The principle of inclusion-exclusion is often seen first in a probability course. But fundamentally it is a counting principle.

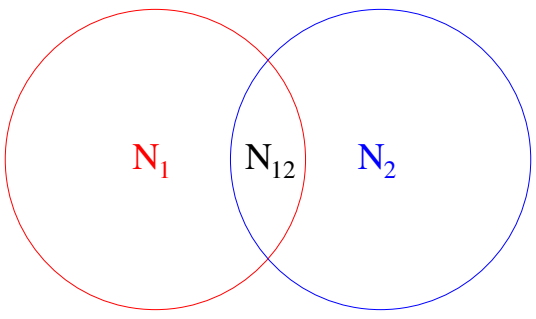
**Theorem 2.11** (Inclusion-exclusion)**.** *Suppose that*

$$\begin{aligned} N_1 &\text{ is a set of objects with attribute \#1} \\ N_2 &\text{ is a set of objects with attribute \#2} \\ N_{12} = N_1 \cap N_2 &\text{ is a set of objects with both attributes.} \end{aligned}$$

*Then, the number of objects having* either *attribute #1* or *attribute #2 (or both) is*

$$\big|N_1 \cup N_2\big| = \big|N_1\big| + \big|N_2\big| - \big|N_{12}\big|.$$

*Proof.* Informally, it is clear from the following picture that the theorem holds:



More formally,

$$\begin{aligned}\big|N_1 \cup N_2\big| &= \big|(N_1 \setminus N_{12}) \cup N_{12} \cup (N_2 \setminus N_{12})\big| \\ &= \big(|N_1| - |N_{12}|\big) + |N_{12}| + \big(|N_2| - |N_{12}|\big) \\ &= |N_1| + |N_2| - |N_{12}|.\end{aligned}$$

□

**Example 2.12** (Divisors of integers)**.** How many positive integers not exceeding 1000 are divisible by 7 or 11?

Let $n_1$ be the number of such integers that are divisible by 7, let $n_2$ be the number that are divisible by 11, and let $n_{12}$ be the number that are divisble by both (or equivalently, by 77). Then

$$\begin{aligned}n_1 &= \left\lfloor \frac{1000}{7} \right\rfloor = 142 \\ n_2 &= \left\lfloor \frac{1000}{11} \right\rfloor = 90 \\ n_{12} &= \left\lfloor \frac{1000}{7 \cdot 11} \right\rfloor = 12,\end{aligned}$$

where $\lfloor x \rfloor$ is the "floor" function, denoting the integer part of $x$. By the principle of inclusion-exclusion, the number of integers not exceeding 1000 that are divisible by 7 or 11 is then

$$n_1 + n_2 - n_{12} = 142 + 90 - 12 = 220.$$

A simple induction argument generalizes the principle of inclusion-exclusion as follows.

**Theorem 2.13** (Generalized inclusion-exclusion)**.** *If $N_i$ is a set of objects that have attribute $i$, for $1 \le i \le r$,*

$$\begin{aligned}\big|N_1 \cup \cdots \cup N_r\big| &= \sum_{i=1}^{r} |N_i| - \sum_{i_1<i_2} \big|N_{i_1} \cap N_{i_2}\big| + \cdots + (-1)^{k+1} \sum_{i_1<\cdots<i_k} \big|N_{i_1} \cap \cdots \cap N_{i_k}\big| \\ &\qquad + \cdots + (-1)^{r+1}\big|N_1 \cap \cdots \cap N_r\big| \\ &= \sum_{k=1}^{r} (-1)^{k+1} \sum_{i_1<\cdots<i_k} \left| \bigcap_{j=1}^{k} N_{i_j} \right|.\end{aligned}$$

*Proof.* Rather than proving this by induction, we will use an approach that supplies a different kind of intuition. Consider an object that has $m$ out of the $r$ possible attributes. On the left-hand side of the equation above, every object is counted exactly once. On the right-hand side, such an object is counted

$m$ times among single attributes ($k = 1$),
$\binom{m}{2}$ times among pairs of attributes ($k = 2$),
$\binom{m}{3}$ times among triplets of attributes ($k = 2$),
etc.

But using the binomial theorem,

$$m - \binom{m}{2} + \binom{m}{3} - \cdots = 1 - \left[\binom{m}{0} - \binom{m}{1} + \binom{m}{2} - \binom{m}{3} + \cdots\right] = 1 - (1-1)^m = 1,$$

so the object is counted exactly once, as it should be. This argument holds for any $m$. Therefore, all objects are counted exactly once on both sides of the equation, and the equality holds.

This is an example of a *combinatorial proof*. In such a proof, rather than using analytical methods to prove an equality, one interprets both sides of the equality as different ways of counting the identical quantity. □

The following classic probability puzzler is a delightful example of the generalized inclusion-exclusion principle.

**Example 2.14** (The coatroom)**.** Imagine that $r$ people arrive at a party, and each one leaves his or her coat in the coatroom. When the people leave, they are too drunk to remember whose coat is which, and so each person picks a coat at random. What is the probability $p$ that each person gets *someone else's* coat? And what happens to $p$ asymptotically, when $r$ tends to infinity?

First, some intuition. It might be reasonable to expect that when $r$ is very large, then just by pure chance, at least one person would get his or her own coat back. In that case, $p$ would tend to zero. But how do we show this?

Although this is a probability problem, we may just as well think of it as a combinatorics problem, by considering the number of ways in which each person can get someone else's coat. Then $p$ will simply be that quantity divided by $r!$, since every one of the $r!$ permutations of the coats is equally probable.

So how do we calculate the number of ways in which each person gets someone else's coat? We might try to start with the product rule: the first person leaving the party has $r-1$ possible options, the second person leaving the party has... is it $r-2$ possible options?... or $r-1$ possible options, if that first person has already taken the second person's coat?... Clearly, it is not a straightforward application of the product rule.

Instead, let $N$ be the set of ways in which *at least* one person recovers his or her own coat, and let $N_i$ be the set of ways in which the $i$th person recovers his or her own coat. Then,

$$|N| = \left|N_1 \cup \cdots \cup N_r\right|,$$

but from generalized inclusion-exclusion,

$$\left|N_1 \cup \cdots \cup N_r\right| = \sum_{k=1}^{r} (-1)^{k+1} \sum_{i_1 < \cdots < i_k} \left|\bigcap_{j=1}^{k} N_{i_j}\right|.$$

For a given value of $k$, what does $\left|N_{i_1} \cap \cdots \cap N_{i_k}\right|$ represent? This is the number of ways in which individuals $i_1$ through $i_k$ can each recover their own coats, which is just the number of permutations of the remaining $r-k$ coats, or $(r-k)!$. Thus,

$$\begin{aligned} |N| &= \sum_{k=1}^{r} (-1)^{k+1} \sum_{i_1 < \cdots < i_k} (r-k)! \\ &= \sum_{k=1}^{r} (-1)^{k+1} \binom{r}{k} (r-k)! \\ &= \sum_{k=1}^{r} (-1)^{k+1} \frac{r!}{k!}. \end{aligned}$$

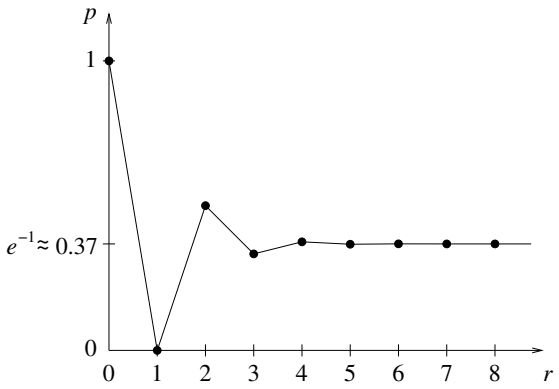
Recall that every one of the $r!$ permutations of the coats are equally likely, so the probability that at least one person gets his or her own coat back is

$$\frac{|N|}{r!} = \sum_{k=1}^{r} \frac{(-1)^{k+1}}{k!}.$$

The probability $p$ that *nobody* gets his or her own coat back is the complement of this:

$$p = 1 - \frac{|N|}{r!} = 1 - \sum_{k=1}^{r} \frac{(-1)^{k+1}}{k!} = \sum_{k=0}^{r} \frac{(-1)^{k}}{k!}.$$

It is instructive to plot $p$ for increasing $r$.




When $r = 0$ (silly case), of course no one can get their coat back, so $p = 1$. When $r = 1$ (almost as silly), the person must get his or her coat back, so $p = 0$. When $r = 2$, there is an equal chance of both people recovering their own coats and of each one getting the other's, so $p = 1/2$. But very quickly $p$ converges to its asymptotic value, which is nothing but the Taylor series for $e^{-1}$. So our intuition was wrong: when $r$ is large, $p$ does not tend to zero at all!

## 2.4 Pigeonhole principle

So far, we have just been counting. We now turn to a combinatorial principle that allows us to prove certain propositions directly. The pigeonole principle is another property that, while seemingly trivial, has nontrivial consequences.

**Theorem 2.15** (Pigeonhole principle)**.** *If $k + 1$ or more objects are placed in $k$ boxes, at least one box must contain two or more objects.*

This property becomes particularly useful when generalized as follows.

**Theorem 2.16** (Generalized pigeonhole principle)**.** *If $m$ or more objects are placed in $k$ boxes, at least one box must contain $\lceil m/k \rceil$ or more objects.*

Here, $\lceil x \rceil$ is the "ceiling" function, denoting the smallest integer greater than or equal to $x$.

Essentially, the pigeonhole principle formalizes the observation that we minimize the number of objects in the most populated box when the distribution of objects is as flat as possible. This results in a very useful bound on the number of objects in each box.

**Example 2.17** (Course grades: Rosen 5.2, example 6)**.** Students in a class can receive 5 possible grades: A, B, C, D and F. What is the minimum number of students needed in the class, to insure that at least 6 of them have the same grade?

Let

$$\begin{aligned} m &= \#\text{ of students (objects)} \\ k &= \#\text{ of grades (boxes).} \end{aligned}$$

We know that $k = 5$, and want to find the smallest $m$ satisfying

$$\left\lceil \frac{m}{5} \right\rceil = 6.$$

This gives $m = 26$. So as long as there are 26 students, no matter how well or badly the students do in the course, at least 6 of them must get the same grade. (Of course, the pigeonhole principle will not tell us anything about what that grade will be!)

Some of the most powerful applications of the pigeonhole principle are less straightforward.

**Example 2.18** (Computer factory: Roberts example 2.34)**.** A computer factory produces, over a given 30-day period,

- at least one laptop computer every day
- an integer number of laptops every day (so no machine is ever left unfinished at the end of the day)
- *on average*, no more than 1.5 laptops per day (so no more than 45 over the entire 30-day period)

Can we show that there is a period of consecutive days where *exactly* 14 laptops are produced?

Let $a_i = \#$ of laptops produced cumulatively through the $i$th day. From the conditions above, we know that

$$1 \leq a_1 < a_2 < \cdots < a_{30} \leq 45,$$

where $a_i \in \mathbb{Z}$.

Now consider the following sequence:

$$a_1, a_2, \ldots, a_{30}, a_1 + 14, a_2 + 14, \ldots, a_{30} + 14$$

Notice that there are 60 integers in this sequence, all between 1 and $45 + 14 = 59$. Therefore, by the pigeonhole principle, at least two numbers in the sequence must be the same. However, due to the strict inequalities in the equation above, $\forall i \neq j$, $a_i \neq a_j$, and of course $a_i + 14 \neq a_j + 14$ as well. Thus,

$$\exists i \neq j \quad \text{such that} \quad a_i = a_j + 14.$$

It follows that exactly 14 computers are produced, starting on day $j + 1$ and ending on day $i$.

Applications of the pigeonhole principle can be even more subtle, and much of the art of discrete modeling involves finding the appropriate "hidden" techniques for solving a problem. Here is a further example, from number theory.

**Example 2.19** (Divisors of integers #2: Rosen 5.2, example 11.)**.** Given $n+1$ distinct integers chosen from the set $\{1, 2, \ldots, 2n\}$, show that one of these must divide another.

Let $a_1, \ldots, a_{n+1}$ be the $n+1$ integers chosen. For any $a_i$, write

$$a_i = 2^{k_i} q_i,$$

where $k_i$ is a nonnegative integer and $q_i$ is an odd integer. Note that $k_i$ could be 0, if $a_i$ is odd. If $a_i$ is even, on the other hand, what we are doing is stripping away as many powers of 2 as possible, and then $q_i$ is the odd number that remains.

Note that $q_1, \ldots, q_{n+1}$ are all odd positive integers less than $2n$. There are only $n$ such integers available, so by the pigeonhole principle, there must exist two that are equal. Call these $q_i = q_j$, where $i \neq j$. Then

$$a_i = 2^{k_i - k_j} a_j.$$

But we said that $a_i$ and $a_j$ are distinct integers. Consequently, one must divide the other, and moreover their ratio must be a power of 2.

## 2.5 Generating functions

Herbert Wilf, in his book *generatingfunctionology*, describes a generating function as "a clothesline on which we hang a sequence of numbers." More formally, generating functions are power series whose coefficients encode information about a sequence of numbers $a_k$. Generating functions are a tremendously powerful tool in discrete mathematics: some applications include problems of counting, solving recurrence relations, and proving or solving combinatorial identities. We consider all three of these below.

**Definition 2.20.** *Given an infinite sequence $a_0, a_1, a_2, \ldots$, the* **ordinary generating function** *is defined as*

$$G(x) = \sum_{k=0}^{\infty} a_k x^k = a_0 + a_1 x + a_2 x^2 + \cdots$$

**Example 2.21.** *Consider the infinite sequence $1, 1, 1, \ldots$.*

$$G(x) = 1 + x + x^2 + \cdots = \frac{1}{1-x} \qquad \text{for } |x| < 1.$$

It is straightforward to generalize this to finite sequences: simply set $a_k = 0$ for all $k$ greater than a certain value.

**Example 2.22.** *Consider the finite sequence*

$$\binom{n}{0}, \binom{n}{1}, \ldots, \binom{n}{n}.$$

*Then*

$$G(x) = \binom{n}{0} + \binom{n}{1} x + \binom{n}{2} x^2 + \cdots + \binom{n}{n} x^n = (1+x)^n$$

*by the binomial theorem.*

We now discuss the three applications of generating functions that we mentioned.

### 2.5.1 Counting

Suppose that we want to count the number of nonnegative integer solutions to an equation of the form

$$k_1 + k_2 + \cdots = k.$$

For this purpose, consider the ordinary generating function formed by the product of polynomials

$$G(x) = (1 + x + x^2 + \cdots)(1 + x + x^2 + \cdots)\cdots$$

$G(x)$ can be expanded as

$$G(x) = a_0 + a_1 x + a_2 x^2 + \cdots$$

Notice that the $k$th coefficient $a_k$ counts the number of ways to obtain $x^k$ by multiplying out the terms in $G(x)$, i.e., the number of ways of solving

$$(x^{k_1})(x^{k_2})\cdots = x^k.$$

Consequently, $a_k$ gives exactly the number of nonnegative integer solutions that we are asking for.

But there is a major problem with this formulation. When $G(x)$ is an infinite product of infinite polynomials, all coefficients $a_1$, $a_2$, etc. will be infinite as well! That reflects the fact that our original nonnegative integer equation has an infinite number of solutions for $k > 0$.

So let us modify that original equation, by imposing additional constraints on the values that any given $k_i$ can take on. For instance, $k_i$ might be nonzero only for $i \in \{1, \ldots, n\}$, or the value $k_i$ might be restricted to a finite subset of integers. We would then modify the generation function by omitting the corresponding powers of $x$ in the polynomials making up $G(x)$. As long as our equation has a finite number of solutions, the coefficient $a_k$ will be finite as well.

**Example 2.23.** Count the number of nonnegative integer solutions to

$$k_1 + k_2 + \cdots = 4$$

where

$$\begin{aligned} k_1 &\in \{0, 1, 2, 3, \ldots\}, \\ k_2 &\in \{0, 2, 4, 6, \ldots\}, \\ k_3 &\in \{0, 3, 6, 9, \ldots\}, \\ k_4 &\in \{0, 4, 8, 12, \ldots\}, \\ &\text{etc.} \end{aligned}$$

The generating function is given by

$$\begin{aligned} G(x) &= (1 + x + x^2 + \cdots)(1 + x^2 + x^4 + \cdots)(1 + x^3 + x^6 + \cdots)(1 + x^4 + x^8 + \cdots)\cdots \\ &= 1 + x + 2x^2 + 3x^3 + 5x^4 + \cdots \end{aligned}$$

In this expansion, the coefficient $a_4$ of $x^4$ is 5, so there are 5 possible solutions for $(k_1, k_2, k_3, k_4, \ldots)$. It is easy to confirm by explicit counting that this is correct, and that the solutions are

$$\begin{aligned} &(4, 0, 0, 0, 0, \ldots) \qquad (0, 4, 0, 0, 0, \ldots) \\ &(2, 2, 0, 0, 0, \ldots) \qquad (0, 0, 0, 4, 0, \ldots) \\ &(1, 0, 3, 0, 0, \ldots) \end{aligned}$$

Notice the general approach here. If one can express a counting problem as the number of integer solutions to an equation

$$k_1 + k_2 + \cdots = k,$$

with appropriate constraints on $k_1, k_2, \ldots$, i.e.,

$$k_i \in \{k_{i1}, k_{i2}, \ldots\}$$

for any given $i$, then the generating function is expressed as

$$G(x) = \prod_{i=1}^{\infty} \left( x^{k_{i1}} + x^{k_{i2}} + \cdots \right)$$

and the number of integer solutions to the equation is given by the coefficient of $x_k$ in $G(x)$.

We now consider a range of counting problems that can be solved in this way.

**Partitioning objects into groups**

Take the problem of partitioning $k$ indistinguishable objects into at most $n$ groups. In how many ways can we do this?

This is equivalent to choosing an ordered $n$-tuple of nonnegative integers $(k_1, \ldots, k_n)$ that sum to $k$. (Since we asked for "at most" $n$ groups, some of these integers could be zero.) We therefore ask for the number of nonnegative integer solutions to

$$k_1 + \cdots + k_n = k.$$

Since we now have the constraint that $k_i$ is nonzero only for $i \in \{1, \ldots, n\}$, the generating function is given by

$$G(x) = \underbrace{(1 + x + x^2 + \cdots)(1 + x + x^2 + \cdots) \cdots (1 + x + x^2 + \cdots)}_{n \text{ times}} = \left( \frac{1}{1 - x} \right)^n \qquad \text{for } |x| < 1,$$

and we need the coefficient $a_k$ in the expansion

$$G(x) = a_0 + a_1 x + a_2 x^2 + \cdots$$

If we think of this expansion as a Taylor series, then clearly

$$a_k = \frac{G^{(k)}(0)}{k!},$$

and so we simply need to differentiate $G(x) = (1 - x)^{-n}$ $k$ times and set its argument to 0:

$$G^{(k)}(x) = \frac{(n + k - 1)!}{(n - 1)!} (1 - x)^{-n-k} \qquad \text{for } |x| < 1$$

$$\implies a_k = \frac{G^{(k)}(0)}{k!} = \frac{(n + k - 1)!}{k!(n - 1)!} = \binom{n + k - 1}{n - 1}.$$

**Example 2.24** (Voting outcomes)**.** In a secret ballot, 10 votes are cast, each for one of 3 possible candidates. What is the number of possible outcomes?

This is precisely a problem of partitioning $k = 10$ objects (votes) into at most $n = 3$ groups (candidates). An outcome here is an ordered triplet $(k_1, k_2, k_3)$ of votes, where $k_1 + k_2 + k_3 = 10$. The number of nonnegative integer solutions to the equation is

$$a_{10} = \binom{3 + 10 - 1}{3 - 1} = \binom{12}{2} = 66.$$

That is the answer we need, but it is also helpful to use this example to highlight the approach. We are constructing a generating function

$$\begin{aligned} G(x) &= (1 + x + x^2 + \cdots)(1 + x + x^2 + \cdots)(1 + x + x^2 + \cdots) \\ &= 1 + 3x + 6x^2 + \cdots, \end{aligned}$$

whose coefficients perform the count that we need. For instance, the coefficient of $x^2$ comes from the 6 triplets of nonnegative integers $(k_1, k_2, k_3)$ satisfying

$$(x^{k_1})(x^{k_2})(x^{k_3}) = x^2,$$

namely

$$\begin{array}{ll} (2,0,0) & (0,1,1) \\ (0,2,0) & (1,0,1) \\ (0,0,2) & (1,1,0) \end{array}$$

Therefore, $a_{10}$ counts the number of triplets $(k_1, k_2, k_3)$ satisfying

$$(x^{k_1})(x^{k_2})(x^{k_3}) = x^{10},$$

and by differentiating $G(x)$ 10 times, we recover that coefficient.

Suppose we wanted to partition $k$ objects into exactly $n$ *nonempty* groups. Then, we would instead use the generating function

$$G(x) = \underbrace{(x + x^2 + \cdots)(x + x^2 + \cdots)\cdots(x + x^2 + \cdots)}_{n \text{ times}} = \left(\frac{x}{1 - x}\right)^n \qquad \text{for } |x| < 1,$$

since that rules out having any $n$-tuples with an element of 0. Note that this just shifts over all coefficients by $n$ in the Taylor expansion of $G(x)$. So partitioning $k$ objects into $n$ nonempty groups is exactly equivalent to partitioning $k - n$ objects into $n$ (possibly empty) groups: essentially, we put one object in each group, and then solve our previous problem.

### Choosing $k$ objects from among $n$ groups

Now imagine that we have $n$ groups of objects, with group sizes $s_1, \ldots, s_n$, and the objects within each group are indistinguishable from each other. If we choose $k$ of those objects, in how many ways can we draw them from the different groups?

This is somewhat similar to the previous problem, in that we are looking for the number of nonnegative integer solutions to the equation

$$k_1 + \cdots + k_n = k.$$

But now, in each group $i$, $k_i$ is bounded by the number of objects $s_i$ in that group.

Setting up the generating function is not hard. As before, we let $G(x)$ be a product of $n$ terms, one for each group, but now each of the $n$ contributions is finite rather than infinite,

$$\begin{aligned} G(x) &= (1 + x + \cdots + x^{s_1}) \cdots (1 + x + \cdots + x^{s_n}) \\ &= \left(\frac{1 - x^{s_1+1}}{1 - x}\right) \cdots \left(\frac{1 - x^{s_n+1}}{1 - x}\right), \end{aligned}$$

reflecting the fact that group $i$ can contribute no more than $s_i$ objects to the collection.

Given the generating function, we can once again produce the answer by differentiating appropriately:

$$a_k = \frac{G^{(k)}(0)}{k!}.$$

The main difference is that in this case, differentiating the generating function is harder. Note, however, that if $k \leq \min(s_1, ..., s_n)$, we can instead solve the problem by using an infinite series product $G(x) = (1 - x)^{-n}$. That is because the maximum number of objects in a group no longer constrains us, and so the problem becomes identical to the earlier one of partitioning objects into groups.

#### Choosing $k$ objects out of a set of $n$ objects

Say that we have a set of $n$ different (distinguishable) objects, and we want to choose $k$ of them. In how many ways can we do it? Of course, this is just a combination, and we already know how to calculate it without generating functions. But it is instructive to see how to do it with generating functions.

Let

$$G(x) = \underbrace{(1 + x)(1 + x) \cdots (1 + x)}_{n \text{ times}} = (1 + x)^n.$$

As before, when expanded as a polynomial, the function's coefficients $a_k$ simply count the number of ways that $k$ objects can be placed in $n$ groups, but now each group can have only 0 or 1 objects: 1 if that is among the chosen objects, 0 if it is not. So when $G(x)$ is expanded, the coefficient $a_k$ of the term $x^k$ counts the number of ways in which $k$ objects can be chosen and the remaining $n - k$ objects not chosen. To find $a_k$, we differentiate as usual:

$$\begin{aligned} a_k &= \frac{G^{(k)}(0)}{k!} \\ &= \frac{n(n-1)\cdots(n-k+1)}{k!} \\ &= \binom{n}{k}, \end{aligned}$$

and we recover the known result on the number of combinations.

### 2.5.2 Solving recurrence relations

**Example 2.25.** Consider the recurrence relation $a_{k+1} = 2a_k + 1$, where $a_0 = 0$. How can we find a closed-form expression for $a_k$?

Let $G(x)$ be the generating function for the sequence $a_k$, so

$$\begin{aligned} G(x) &= \sum_{k=0}^{\infty} a_k x^k \\ &= 0 + \sum_{k=1}^{\infty} a_k x^k \\ &= \sum_{k=0}^{\infty} a_{k+1} x^{k+1}. \end{aligned}$$

But based on our recurrence, it follows that

$$\begin{aligned} G(x) &= \sum_{k=0}^{\infty} (2a_k + 1) x^{k+1} \\ &= 2x \sum_{k=0}^{\infty} a_k x^k + x \sum_{k=0}^{\infty} x^k \\ &= 2xG(x) + \frac{x}{1-x} \qquad \text{for } |x| < 1. \end{aligned}$$

Thus,

$$G(x)(1-2x) = \frac{x}{1-x} \qquad \Longrightarrow \qquad G(x) = \frac{x}{(1-x)(1-2x)} = \frac{1}{1-2x} - \frac{1}{1-x}$$

using partial fractions. Now that we have the generating function, we can extract $a_k$ just as in our counting problems, by differentiating $G(x)$. It is not difficult to show that

$$G^{(k)}(x) = 2^k k!(1-2x)^{-1-k} - k!(1-x)^{-1-k},$$

in which case

$$a_k = \frac{G^{(k)}(0)}{k!} = 2^k - 1.$$

**Example 2.26.** Now consider the recurrence relation $a_{k+1} = 2a_k + k$, where $a_0 = 0$. What is $a_k$?

As before, let $G(x)$ be the generating function for the sequence $a_k$, so

$$\begin{aligned} G(x) &= \sum_{k=0}^{\infty} a_k x^k \\ &= 0 + \sum_{k=1}^{\infty} a_k x^k \\ &= \sum_{k=0}^{\infty} a_{k+1} x^{k+1}. \end{aligned}$$

Based on our recurrence, it follows that

$$\begin{aligned} G(x) &= \sum_{k=0}^{\infty} (2a_k + k) x^{k+1} \\ &= 2x \sum_{k=0}^{\infty} a_k x^k + x \sum_{k=0}^{\infty} k x^k. \end{aligned}$$

How do we calculate the infinite sum $\sum_{k=0}^{\infty} kx^k$? The easiest way is to express it in terms of a derivative

$$\begin{aligned}\sum_{k=0}^{\infty} kx^k &= \left(\frac{d}{dx}\sum_{k=0}^{\infty} x^{k+1}\right) - \sum_{k=0}^{\infty} x^k \\ &= \left(\frac{d}{dx}\frac{x}{1-x}\right) - \frac{1}{1-x} \\ &= \frac{1}{(1-x)^2} - \frac{1}{1-x} \\ &= \frac{x}{(1-x)^2},\end{aligned}$$

as long as $|x| < 1$. Then

$$\begin{aligned}G(x) &= 2xG(x) + \frac{x^2}{(1-x)^2} \\ \implies \quad G(x) &= \frac{x^2}{(1-x)^2(1-2x)} = \frac{1}{1-2x} - \frac{1}{(1-x)^2}\end{aligned}$$

using partial fractions. Differentiation yields

$$G^{(k)}(x) = 2^k k!(1-2x)^{-1-k} - (k+1)!(1-x)^{-2-k},$$

so

$$a_k = \frac{G^{(k)}(0)}{k!} = 2^k - k - 1.$$

### 2.5.3 Proving combinatorial identities

**Example 2.27.** One of the homework problems involved using a combinatorial proof to show that

$$\sum_{k=1}^{n} k\binom{n}{k}^2 = n\binom{2n-1}{n-1}.$$

How can we show this using generating functions? Write it as

$$\sum_{k=1}^{n} \frac{k}{n}\binom{n}{k}^2 = \binom{2n-1}{n-1},$$

and then observe that the right-hand side is the coefficient $a_{n-1}$ of $x^{n-1}$ in the generating function

$$G(x) = (1+x)^{2n-1}.$$

In order to turn this into the form on the left-hand side, we somehow need to produce a product of two binomial coefficients. That suggests breaking up $G(x)$ into two parts:

$$G(x) = (1+x)^{n-1}(1+x)^n,$$

and then from the binomial theorem,

$$\begin{aligned} G(x) &= \sum_{k=0}^{n-1} \binom{n-1}{k} x^k \sum_{l=0}^{n} \binom{n}{l} x^l \\ &= \sum_{k=1}^{n} \binom{n-1}{k-1} x^{k-1} \sum_{l=0}^{n} \binom{n}{l} x^l \\ &= \sum_{k=1}^{n} \sum_{l=0}^{n} \frac{k}{n} \binom{n}{k} \binom{n}{l} x^{k+l-1}. \end{aligned}$$

Recall that we were interested in the coefficient $a_{n-1}$ of $x^{n-1}$ in the polynomial expansion of $G(x)$. That corresponds to all sum contributions where $n = k + l$, or alternatively $l = n - k$, so it follows that

$$\begin{aligned} a_{n-1} &= \sum_{k=1}^{n} \frac{k}{n} \binom{n}{k} \binom{n}{n-k} \\ &= \sum_{k=1}^{n} \frac{k}{n} \binom{n}{k}^2. \end{aligned}$$

## 2.6 Algorithms and complexity

Finally, we turn to some discussion of algorithmic analysis.

Algorithmic development has been the engine driving much of the progress in discrete mathematics over the past 50 years, and enabling much of today's discrete modeling capabilities. Much has been made of the improvements to computational power due to Moore's law, which very roughly states that hardware capabilities double every two years. But the fact remains that algorithmic improvements have largely outpaced Moore's law. *For many numerical problems, a modern algorithm on a 1950s computer would run much faster than a 1950s algorithm on a modern computer.*

So how do we characterize algorithmic performance?

### 2.6.1 Asymptotic growth of functions

When we study the running time of an algorithm, we typically are concerned with how this grows asymptotically, as the problem input size becomes large. Let us review the notation for asymptotic growth of functions.

- $f(n) = o(g(n))$:
$$\forall c > 0, \exists n_0 : \forall n > n_0,\ f(n) < c\, g(n).$$
This means $f(n)$ scales (in $n$) strictly slower than $g(n)$ (very roughly: "$f < g$").
- $f(n) = O(g(n))$:
$$\exists c, n_0 : \forall n > n_0,\ f(n) \leq c\, g(n).$$
This means $f(n)$ scales no faster than $g(n)$ (very roughly: "$f \leq g$").
- $f(n) = \Theta(g(n))$:
$$\exists c_1, c_2, n_0 : \forall n > n_0,\ c_1\, g(n) \leq f(n) \leq c_2\, g(n).$$
This means $f(n)$ scales at the same rate as $g(n)$ (very roughly, "$f = g$").

- $f(n) = \Omega(g(n))$:
$$\exists c, n_0 : \forall n > n_0,\ f(n) \geq c\, g(n).$$
This means $f(n)$ scales no slower than $g(n)$ (very roughly: "$f \geq g$").

- $f(n) = \omega(g(n))$:
$$\forall c > 0, \exists n_0 : \forall n > n_0,\ f(n) > c\, g(n).$$
This means $f(n)$ scales strictly faster than $g(n)$ (very roughly: "$f > g$").

**Example 2.28.** Let $f(n) = 3n$ and $g(n) = n^2 + 2n + 1$. When $n$ is large, $f(n)$ grows linearly whereas $g(n)$ grows quadratically. In this case, both of the first two properties hold:

- $f(n) = o(g(n))$, which can be seen formally by letting $n_0 = 3/c$:
$$\forall n > 3/c, \quad 3n < cn^2 < c(n^2 + 2n + 1).$$

- $f(n) = O(g(n))$, which can be seen formally by letting $c = 3$ and $n_0 = 1$:
$$\forall n > 1, \quad 3n < 3(n^2 + 2n + 1).$$

**Example 2.29.** Let $f(n) = 3n^2 + 2n + 8$ and $g(n) = n^2$. When $n$ is large, both $f(n)$ and $g(n)$ grow quadratically. So in this case, the middle three properties hold:

- $f(n) = O(g(n))$, which can be seen formally by letting $c = 4$ and $n_0 = 4$:
$$\forall n > 4, \quad 3n^2 + 2n + 8 \leq 4n^2.$$

- $f(n) = \Theta(g(n))$, which can be seen formally by letting $c_1 = 3$, $c_2 = 4$ and $n_0 = 4$
$$\forall n > 4, \quad 3n^2 \leq 3n^2 + 2n + 8 \leq 4n^2.$$

- $f(n) = \Omega(g(n))$, which can be seen formally by letting $c = 3$ and $n_0 = 1$:
$$\forall n > 1,\ 3n^2 + 2n + 8 \geq 3n^2.$$

### 2.6.2 Computational complexity

Let $n$ be the size of the input to a problem, and let $f(n)$ be the running time of an algorithm to solve it. Some common classifications of algorithms are as follows:

- $\Theta(1)$: **constant time**
- $\Theta(\ln n)$: **logarithmic time**
- $\Theta(n)$: **linear time**
- $\Theta(n^\alpha)$: **polynomial time**
- $\Theta(\alpha^n), \alpha > 1$: **exponential time**

Generally, if an algorithm runs in polynomial time, it is considered to be *computationally tractable* — although now that datasets can often consist of billions of records, linear time is becoming an increasing necessity. Conversely, if an algorithm requires exponential time, it is *computationally intractable*. Even if the problem input is only of a modest size, the algorithm will in most cases take impractically long to run.

Of course, there are more and less efficient algorithms for solving any given problem. However, when we speak of the *intrinsic complexity* of a problem, we refer to the scaling of the *most efficient* algorithm to solve it.

Three important classes of intrinsic computational complexity are:

- **P**: the class of problems that can be solved in polynomial time. For instance, even using a crude algorithm, a list of $n$ numbers can be sorted in $O(n^2)$ time. Thus, sorting is in the P class.
- **NP**: the class of problems whose solutions can be *verified* in polynomial time. ("I may not be able to find a needle in a haystack, but I can verify quickly whether something I've found is a needle.")
- **NP-complete**: the class of problems to which *all* NP problems can be reduced in polynomial time. If a (decision) problem $C$ is NP-complete, then $C$ is in NP and there is a recipe for reducing *every* problem in NP to $C$ in polynomial time. Thus, solving an NP-complete problem is "as hard as" solving the hardest NP problems. If a polynomial time algorithm were to exist for solving an NP-complete problem, then a polynomial time algorithm would exist for solving *any* NP problem. ("If I can verify quickly that this is a needle, then I can even *find* it quickly in a haystack!")

Many problems have been proven NP-complete, but a general polynomial-time algorithm has never been found for any one of them. Finding even a single one would imply that P=NP, and thus *all* NP problems are tractable. This is widely believed to be false. But at the same time, no one has yet proven that P≠NP: it remains an open conjecture.

It is important to stress again that the complexity class describes the intrinsic complexity of the problem rather than the complexity of a given algorithm for solving the problem. We illustrate this point next.

### 2.6.3 Complexity of problems vs. complexity of algorithms

Consider the standard long-hand method for multiplying two $n$-digit numbers, by multiplying digits and carrying when necessary. For example, when $n = 4$, we might have:

$$
\begin{array}{cccccccc}
 & & & 1 & 2 & 3 & 4 \\
 & & \times & 4 & 3 & 2 & 1 \\
\hline
 & & & 1 & 2 & 3 & 4 \\
 & & 2 & 4 & 6 & 8 & \\
 & & & {\scriptstyle 1} & & & \\
 & 3 & 6 & 9 & 2 & & \\
 & {\scriptstyle 1} & {\scriptstyle 1} & & & & \\
4 & 8 & 2 & 6 & & & \\
\hline
5 & 3 & 3 & 2 & 1 & 1 & 4
\end{array}
$$

It is straightforward to show that the overall complexity $f(n)$ of this algorithm is $f(n) = \Theta(n^2)$. Is this really the intrinsic complexity of the problem? The Russian mathematician Kolmogorov

apparently thought so, but within a week of his 1960 conjecture to this effect, Karatsuba disproved it with one of the first examples of the algorithmic trick of *divide and conquer*. The idea is as follows.

Say that a problem of input size $n$ can be broken up into $a$ independent subproblems, each of size $n/b$. Let

$$\begin{aligned} f(n) &= \#\text{ of operations to solve problem} \\ g(n) &= \#\text{ of operations to split problem into independent subproblems.} \end{aligned}$$

It then follows that

$$f(n) = g(n) + af\left(\frac{n}{b}\right).$$

If we make the simplifying assumption that $n$ is an integer power of $b$, we have the following theorem. (A special case of this will be proved in the homework.)

**Theorem 2.30.** *If $f(n) = g(n) + af(n/b)$, where $g(n) = o\left(n^{\ln a/\ln b}\right)$, then $f(n) = O\left(n^{\ln a/\ln b}\right)$.*

How do we apply this in the case of multiplication? If $x_1$ and $x_2$ are the two $n$-digit numbers to multiply, and if we assume $n$ is even, let

$$\begin{aligned} x_1 &= 10^{n/2}y_1 + z_1 \\ x_2 &= 10^{n/2}y_2 + z_2. \end{aligned}$$

For instance, in the example above,

$$\begin{aligned} x_1 &= 1234: \qquad y_1 = 12,\ z_1 = 34 \\ x_2 &= 4321: \qquad y_2 = 43,\ z_2 = 21. \end{aligned}$$

Now multiplying the two expressions,

$$\begin{aligned} x_1x_2 &= 10^n y_1y_2 + 10^{n/2}(y_1z_2 + y_2z_1) + z_1z_2 \\ &= 10^n y_1y_2 + 10^{n/2}[(y_1 + z_1)(y_2 + z_2) - y_1y_2 - z_1z_2] + z_1z_2. \end{aligned}$$

The second step seems at first to make things far more complicated. But on closer inspection: since both $y_1y_2$ and $z_1z_2$ are reused in the calculation, it involves only *three* distinct multiplications of $(n/2)$-digit numbers. Splitting the problem into subproblems, as well as the addition operations, take only linear time, so $g(n) = O(n)$. This satisfies the conditions of the theorem with $a = 3$ and $b = 2$, so

$$f(n) = O\left(n^{\ln 3/\ln 2}\right),$$

where $\ln 3/\ln 2 \approx 1.585$. Consequently, the complexity is reduced by almost a factor of $\sqrt{n}$.

Is $\Theta\left(n^{1.585}\right)$ the intrinsic complexity of multiplying two $n$-digit numbers? In 1971, Schönhage and Strassen found an algorithm with running time $f(n) = O(n \log n \log\log n)$. This essentially brings the exponent arbitrarily close to 1, so that $f(n) = O\left(n^{1+\epsilon}\right)$ for any $\epsilon > 0$. In 2019, Harvey and van der Hoeven achieved a new breakthrough: an algorithm with $f(n) = O(n \log n)$, also known as *linearithmic* running time. Both of these are considered "galactic algorithms", meaning that they have such enormous constants in $f(n)$ that they are of no direct practical use. On the other hand, Karatsuba's approach, while still involving large constants, is useful in cryptographic applications where operations on RSA keys routinely use numbers of size $2^{2048}$.

It is conjectured that the 2019 algorithm by Harvey and van der Hoeven is theoretically optimal, in that the intrinsic complexity of multiplication is $f(n) = \Theta(n \log n)$, though this has not been proven. Who would have thought that there still remain open problems in basic integer arithmetic!

### 2.6.4 Average-case complexity

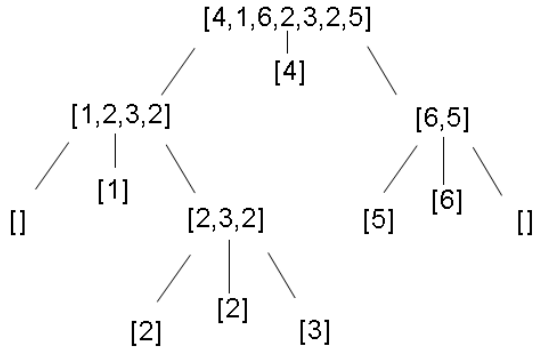
So far, all discussion of computational complexity has focused on the worst-case scenario, in that we ask about running time for the hardest possible input to the problem. But sometimes we care more about the average-case scenario. What is the "typical" number of operations needed?

There are (at least) two possible meanings to average-case complexity:

1. Complexity averaged over possible inputs to the problem.
2. Complexity averaged over possible executions of algorithm, if the algorithm itself has randomness. We could consider this even with the "worst possible" input.

An an example of both of these, consider the problem of sorting a list of $n$ elements, using the divide-and-conquer method known as *Quicksort*. This procedure recursively partitions an ordered list $l$ into groups $l_{\text{left}}$ and $l_{\text{right}}$, using the first item $l(1)$ in the list as a *pivot*: $l_{\text{left}}$ contains all elements that are less than or equal to $l(1)$, and $l_{\text{right}}$ contains all elements that are greater than $l(1)$.

For instance, a list $l = [4, 1, 6, 2, 3, 2, 5]$ is split into two lists based on its first item $l(1) = 4$. Those two lists are $l_{\text{left}} = [1, 2, 3, 2]$ and $l_{\text{right}} = [6, 5]$, each of which is then sorted recursively:




Note that in certain cases, $l_{\text{left}}$ or $l_{\text{right}}$ ends up being an empty list []. At the end, the leaves of the sorting tree are merged from left to right, yielding [1,2,2,3,4,5,6] as desired.

Let us analyze this procedure under the first definition of average-case complexity. Say that at a given step, $l(1)$ happens to be the $r$th smallest element in the list. In a list of length $n$, $n-1$ operations are needed to compare the pivot to the other elements, so

$$f(n) = n - 1 + f(r-1) + f(n-r).$$

If we are always unlucky (worst-case), the input will be such that $r = 1$ or $r = n$ all the time, so

$$f(n) = n - 1 + f(n-1),$$

in which case

$$f(n) = 1 + 2 + 3 + \cdots + n - 1 = \frac{n(n-1)}{2} = \Theta(n^2).$$

But for random inputs, $r$ is equally likely to be anything. So in expectation, the running time is

$$\begin{aligned} \mathbb{E}[f(n)] &= n - 1 + \frac{1}{n}\sum_{r=1}^{n}[f(r-1) + f(n-r)] \\ &= n - 1 + \frac{2}{n}\sum_{r=1}^{n} f(r-1). \end{aligned}$$

As you will show in the homework, this gives $\mathbb{E}[f(n)] = \Theta(n \log n)$. Thus, the typical-case behavior of the algorithm is close to linear — far better than its worst-case behavior.

That was an example of the first meaning of average-case complexity: the average over all possible inputs to the problem. But now say that an adversary is trying to thwart us, by creating lists that are hard to sort. What can we do?

Consider the second meaning of average-case complexity: the average over all executions of the algorithm. Say that we *randomize* the algorithm: instead of always pivoting about $l(1)$, we pivot about some $l(p)$, where $p$ is chosen uniformly at random between 1 and $n$. It is not hard to see that the analysis is identical to that for random inputs, and so again, $\mathbb{E}[f(n)] = \Theta(n \log n)$ on the average. But this time, there is no way for an adversary to thwart us because he or she has no way of knowing which $p$ we will pick! *Regardless of input* to the algorithm, the result holds.