p_{2n}. Furthermore, this married couple $S = \{p_0, p_{2n}\}$ is not the host and hostess, since the host is not in $\{p_0, \dots, p_{2n}\}$.

Everyone not in S shakes with exactly one person in S, namely p_{2n}. If we delete S to obtain a smaller party, then we have n couples remaining (including the host and hostess), no person shakes with a spouse, and each person shakes with one fewer person than in the full party. Hence in the smaller party the people other than the host shake hands with different numbers of people.

By deleting the set S, we thus obtain an n-party (deleting the leftmost couple in the picture for $n = 3$ yields the picture for $n = 2$). Applying the induction hypothesis to this n-party tells us that, outside of the couple S, the hostess shakes with $n - 1$ people. Since she also shakes with $p_{2n} \in S$, in the full $(n + 1)$-party she shakes with n people. ∎

The first argument in Example A.28 falls into the induction trap, because it does not consider all possible $(n + 1)$-parties. It considers only those obtained by adding a couple to an n-party in a certain way, without proving that every $(n + 1)$-party is obtained in this way.

Starting with an arbitrary $(n + 1)$-party forces us to prove that every $(n + 1)$-party arises in this way in order to obtain a configuration where we can apply the induction hypothesis. We cannot discard just any married couple to obtain the smaller party. We must find a couple S such that everyone outside S shakes with exactly one person in S. Only then will the smaller party satisfy the hypotheses needed to be an n-party.

The need to show that our smaller object satisfies the conditions in the induction hypothesis replaces the need to prove that all objects of the larger size were generated by growing from an object of the smaller size.

Sometimes the proof of the induction step uses more than one earlier instance. If we always use both $P(n - 2)$ and $P(n - 1)$ to prove $P(n)$, then we must verify both $P(1)$ and $P(2)$ to get started. The proof of the induction step is not valid for $n = 2$, since there is no $P(0)$ to use.

A.29. Example. Let $a_1, a_2, \dots$ be defined by $a_1 = 2$, $a_2 = 8$, and $a_n = 4(a_{n-1} - a_{n-2})$ for $n \geq 3$. We seek a formula for a_n in terms of n.

We may try to guess a formula that fits the data. The definition yields $a_3 = 24$, $a_4 = 64$, and $a_5 = 160$. All these satisfy $a_n = n2^n$. Having guessed this as a possible formula for a_n, we can try to use induction to prove it.

When $n = 1$, we have $a_1 = 2 = 1 \cdot 2^1$. When $n = 2$, we have $a_2 = 8 = 2 \cdot 2^2$. In both cases, the formula is correct.

In the induction step, we prove that the desired formula is correct for $n \geq 3$. We use the hypothesis that the formula is correct for the preceding instances $n - 1$ and $n - 2$. This allows us to compute a_n using its expression in terms of earlier values:

$$a_n = 4(a_{n-1} - a_{n-2}) = 4[(n-1)2^{n-1} - (n-2)2^{n-2}] = (2n-2)2^n - (n-2)2^n = n2^n.$$

The validity of the formula for a_n follows from its validity for a_{n-1} and a_{n-2}, which completes the proof. ∎

In this proof, we must verify the formula for $n = 1$ and $n = 2$ in the basis step; the proof of the induction step is not valid when $n = 2$. Example A.29 specifies $a_1, a_2, \ldots$ by a **recurrence relation**. The general term a_n is specified using earlier terms. Similarly, the proof of Proposition A.26 yields a recurrence for the number r_n of regions formed by n lines; $r_n = r_{n-1} + n$, with $r_1 = 2$.

If the recurrence relation uses k earlier terms to compute a_n, then we must provide k initial values in order to specify the terms exactly; this is a recurrence of **order** k. Statements proved by induction about recurrences of order k typically require verification of k instances in the basis step. Standard techniques from enumerative combinatorics yield solutions to many recurrence relations without guessing formulas or directly using induction.

We also sometimes use recursive computation in graph theory. We may have a value for each graph instead of just one for each "size" as in a sequence. If we can express the value for a graph G as a formula in terms of graphs with fewer edges (and specify the values for graphs with no edges), then again we have a recurrence. We use this technique to count spanning trees (Section 2.2) and proper colorings (Section 5.3).

FUNCTIONS

A function transforms elements of one set into elements of another.

A.30. Definition. A **function** f *from* a set A *to* a set B assigns to each $a \in A$ a single element $f(a)$ in B, called the **image** of a under f. For a function f from A to B (written $f \colon A \to B$), the set A is the **domain** and the set B is the **target**. The **image** of a function f with domain A is $\{f(a) \colon a \in A\}$.

We take many elementary functions as familiar, such as the absolute value function and polynomials (both defined on $\mathbb{R}$). "Size" is a function whose domain is the set of finite sets and whose target is $\mathbb{N} \cup \{0\}$.

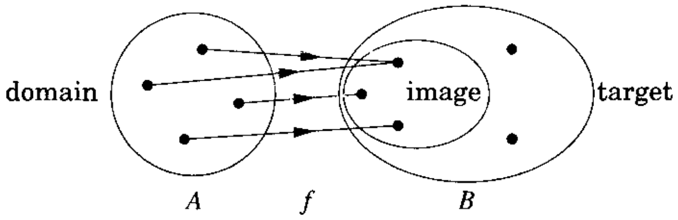
A.31. Definition. For $x \in \mathbb{R}$, the **floor** $\lfloor x \rfloor$ is the greatest integer that is at most x. The **ceiling** $\lceil x \rceil$ is the smallest integer that is at least x. A **sequence** is a function f whose domain is $\mathbb{N}$.

The floor function and ceiling function map $\mathbb{R}$ to $\mathbb{Z}$. When the target of a sequence is A, we have a sequence of elements in A, and we express the sequence as $a_1, a_2, a_3, \ldots$, where $a_n = f(n)$. We have used induction to prove sequences of statements and to prove formulas specifying sequences of numbers.

We may want to know how fast a function from $\mathbb{R}$ to $\mathbb{R}$ grows, particularly when analyzing algorithms. For example, we say that the growth of a function g is (at most) **quadratic** if it is bounded by a quadratic polynomial for all sufficiently large inputs. A more precise discussion of growth rates of functions appears in Appendix B.

A.32. Remark. *Schematic representation.* A function $f\colon A \to B$ is **defined on** A and **maps** A into B. To visualize a function $f\colon A \to B$, we draw a region representing A and a region representing B, and from each $x \in A$ we draw an arrow to $f(x)$ in B. In digraph language, this produces an orientation of a bipartite graph with partite sets A and B in which every element of A is the tail of exactly one edge.

The image of a function is contained in its target. Thus we draw the region for the image inside the region for the target. ■



To describe a function, we must specify $f(a)$ for each $a \in A$. We can list the pairs $(a, f(a))$, provide a formula for computing $f(a)$ from a, or describe the rule for obtaining $f(a)$ from a in words.

A.33. Definition. A function $f\colon A \to B$ is a **bijection** if for every $b \in B$ there is exactly one $a \in A$ such that $f(a) = b$.

Under a bijection, each element of the target is the image of exactly one element of the domain. Thus when a bijection is represented as in Remark A.32, every element of the target is the head of exactly one edge.

A.34. Example. *Pairing spouses.* Let M be the set of men at a party, and let W be the set of women. If the attendees consist entirely of married couples, then we can define a function $f\colon M \to W$ by letting $f(x)$ be the spouse of x. For each woman $w \in W$, there is exactly one $x \in M$ such that $f(x) = w$. Hence f is a bijection from M to W. ■

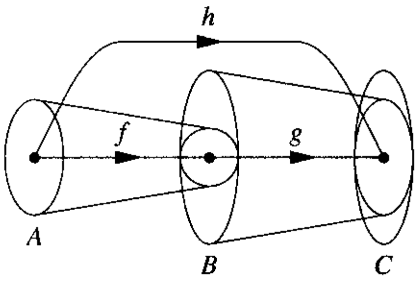
Bijections pair up elements from different sets. Thus we also describe a bijection from A to B as a **one-to-one correspondence** between A and B. Occasionally in the text we say informally that elements of one set "correspond" to elements of another; by this we mean that there is a natural one-to-one correspondence between the two sets.

When A has n elements, listing them as $a_1, \ldots, a_n$ defines a bijection from $[n]$ to A. Viewing the correspondence in the other direction defines a bijection from A to $[n]$. All bijections can be "inverted".

A.35. Definition. If f is a bijection from A to B, then the **inverse** of f is the function $g\colon B \to A$ such that, for each $b \in B$, $g(b)$ is the unique element $x \in A$ such that $f(x) = b$. We write f^{-1} for the function g.

When the target of a function is the domain of a second function, we can create a new function by applying the first and then the second. This yields a function from the domain of the first function into the target of the second.

A.36. Definition. If $f\colon A \to B$ and $g\colon B \to C$, then the **composition** of g with f is a function $h\colon A \to C$ defined by $h(x) = g(f(x))$ for $x \in A$. When h is the composition of g with f, we write $h = g \circ f$.



From the definitions, it is easy to verify that the composition of two bijections is a bijection. We use this in Proposition 1.1.24 in verifying for graphs that a composition of isomorphisms is an isomorphism.

COUNTING AND BINOMIAL COEFFICIENTS

A discussion of counting quickly leads to summations and products. These can be written concisely using appropriate notation.

A.37. Remark. We express summation using $\sum$, the uppercase Greek letter "sigma". When a and b are integers, the value of $\sum_{i=a}^{b} f(i)$ is the sum of the numbers $f(i)$ over the integers i satisfying $a \le i \le b$. Here i is the **index of summation**, and the formula $f(i)$ is the **summand**.

We write $\sum_{j\in S} f(j)$ to sum a real-valued function f over the elements of a set S in its domain. When no subset is specified, as in $\sum_j x_j$, we sum over the entire domain. When the summand has only one symbol that can vary, we may omit the subscript on the summation symbol, as in $\sum x_i$.

Similar comments apply to indexed products using $\prod$, which is the uppercase Greek letter "pi". ■

Two simple rules help organizing the counting of finite sets by breaking problems into subproblems. These rules follow from the definition of size and properties of bijections.

A.38. Definition. The **rule of sum** states that if A is a finite set and $B_1, \dots, B_m$ is a partition of A, then $|A| = \sum_{i=1}^{m} |B_i|$.

Let T be a set whose elements can be described using a procedure involving steps $S_1, \dots, S_k$ such that step S_i can be performed in r_i ways, regardless of *how* steps $S_1, \dots, S_{i-1}$ are performed. The **rule of product** states that $|T| = \prod_{i=1}^{k} r_i$.

For example, there are q^k lists of length k from a set of size q. There are q choices for each position, regardless of the choices in other positions. By the product rule, there are q^k ways to form the k-tuple.

A.39. Definition. A **permutation** of a finite set S is a bijection from S to S. The **word form** of a permutation f of $[n]$ is the list $f(1), \dots, f(n)$ in that order. An **arrangement** of elements from a set S is a list of elements of S (in order). We write $n!$ (read as "n **factorial**") to mean $\prod_{i=1}^{n} i$, with the convention that $0! = 1$.

The word form of a permutation of $[n]$ includes the full description of the permutation. For counting purposes we refer to the word form *as* the permutation; thus 614325 is a permutation of [6]. With this viewpoint, a permutation of $[n]$ is an arrangement of all the elements of $[n]$.

A.40. Theorem. An n-element set has $n!$ permutations (arrangements without repetition). In general, the number of arrangements of k distinct elements from a set of size n is $n(n-1)\cdots(n-k+1)$.

Proof: We count the lists of k distinct elements from a set S of size n. There is no such list when $k > n$, which agrees with the formula. We construct the lists one element at a time, specifying the element in position $i+1$ after specifying the elements in earlier positions.

There are n ways to choose the image of 1. For each way we do this, there are $n-1$ ways to choose the image of 2. In general, after we have chosen the first i images, avoiding them leaves $n-i$ ways to choose the next image, no matter how we made the first i choices. The rule of product yields $\prod_{i=0}^{k-1}(n-i)$ for the number of arrangements. ■

Often the order of elements in a list is unimportant.

A.41. Definition. A **selection** of k elements from $[n]$ is a k-element subset of $[n]$. The number of such selections is "n choose k", written as $\binom{n}{k}$.

If $k < 0$ or $k > n$, then $\binom{n}{k} = 0$; in these cases there are no selections of k elements from $[n]$. When $0 \le k \le n$, we obtain a simple formula.

A.42. Theorem. For integers n, k with $0 \le k \le n$, $\binom{n}{k} = \frac{1}{k!}\prod_{i=0}^{k-1}(n-i)$.

Proof: We relate selections to arrangements. We count the arrangements of k elements from $[n]$ in two ways. Picking elements for positions as in Theorem A.40 yields $n(n-1)\cdot(n-k+1)$ as the number of arrangements.

Alternatively, we can select the k-element subset first and then write it in some order. Since by definition there are $\binom{n}{k}$ selections, the product rule yields $\binom{n}{k}k!$ for the number of arrangements.

In each case, we are counting the set of arrangements, so we conclude that $n(n-1)\cdots(n-k+1) = \binom{n}{k}k!$. Dividing by $k!$ completes the proof. ■

The formula for $\binom{n}{k}$ can be written as $\frac{n!}{k!(n-k)!}$, but the form in the statement of Theorem A.42 tends to be more useful, especially when k is small. For example, $\binom{n}{2} = n(n-1)/2$ and $\binom{n}{3} = n(n-1)(n-2)/6$, the former being the number of edges in a complete graph with n vertices. This form more directly reflects the counting argument and cancels the $(n-k)!$ appearing in both the numerator and denominator.

The numbers $\binom{n}{k}$ are called the **binomial coefficients** due to their appearance as coefficients in the nth power of a sum of two terms.

A.43. Theorem. (Binomial Theorem) For $n \in \mathbb{N}$, $(x+y)^n = \sum_{k=0}^{n}\binom{n}{k}x^k y^{n-k}$.

Proof: The proof interprets the process of multiplying out the factors in the product $(x+y)(x+y)\cdots(x+y)$. To form a term in the product, we must choose x or y from each factor. The number of factors that contribute x is some integer k in $\{0,\ldots,n\}$, and the remaining $n-k$ factors contribute y. The number of terms of the form $x^k y^{n-k}$ is the number of ways to choose k of the factors to contribute x. Summing over k accounts for all the terms. ■

Using the definition of size and the composition of bijections, it follows that finite sets A and B have the same size if and only if there is a bijection from A to B. Thus we can compute the size of a set by establishing a bijection from it to a set of known size.

Simple examples include the statements that a complete graph has $\binom{n}{2}$ edges and that therefore there are $2^{\binom{n}{2}}$ simple graphs with vertex set $[n]$. Proposition 1.3.10 uses a bijection to count 6-cycles in the Petersen graph. Exercise 1.3.32 uses a bijection to count graphs with vertex set $[n]$ and even vertex degrees. Theorem 2.2.3 uses a bijection to count trees with vertex set $[n]$.

A.44. Lemma. For $n \in \mathbb{N}$, the number of subsets of $[n]$ with even size equals the number of subsets of $[n]$ with odd size.

Proof: *Proof 1* (bijection). For each subset with even size, delete the element n if it appears, and add n if it does not appear. This always changes the size by 1 and produces a subset with odd size. The map is a bijection, since each odd subset containing n arises only from one even subset omitting n, and each odd subset omitting n arises only from one even subset containing n.

Proof 2 (binomial theorem). Setting $x=-1$ and $y=1$ in Theorem A.43. yields $\sum_{k=0}^{n}\binom{n}{k}(-1)^k = (-1+1)^n = 0$. (Note that we proved Theorem A.43 using bijections.) ■

We prove a few identities involving binomial coefficients to illustrate combinatorial arguments involving bijections and the idea of counting a set in two ways. We can prove an equality by showing that both sides count the same set.

A.45. Lemma. $\binom{n}{k} = \binom{n}{n-k}$.

Proof: *Proof 1* (counting two ways). By definition, $[n]$ has $\binom{n}{k}$ subsets of size k. Another way to count selections of k elements is to count selections of $n-k$ elements to omit, and there are $\binom{n}{n-k}$ of these.

Proof 2 (bijections). The left side counts the k-element subsets of $[n]$, the right side counts the $n-k$-element subsets, and the operation of "complementation" establishes a bijection between the two collections. ■

Often, "counting two ways" means grouping the elements in two ways. Sometimes one of the counts only gives a bound on the size of the set. In this case the counting argument proves an inequality; there are several instances of this phenomenon in Chapter 3 (see also Exercise 1.3.31). Here we stick to equalities.

A.46. Lemma. (The Chairperson Identity) $k\binom{n}{k} = n\binom{n-1}{k-1}$.

Proof: Each side counts the k-person committees with a designated chairperson that can be formed from a set of n people. On the left, we select the committee and then select the chair from it; on the right, we select the chair first and then fill out the rest of the committee. ■

Many students see the next formula as the first application of induction, but it also is easily proved by counting a set in two ways.

A.47. Lemma. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Proof: The right side is $\binom{n+1}{2}$; we can view this as counting the nontrivial intervals with endpoints in the set $\{1, \dots, n+1\}$. On the other hand, we can group the intervals by length; there is one interval with length n, two with length $n-1$, and so on up to n intervals with length 1. ■

Lemma A.47 generalizes to $\sum_{i=k}^{n} \binom{i}{k} = \binom{n+1}{k+1}$. To prove this by counting in two ways, partition the set of $k+1$-element subsets of $[n+1]$ into groups so that the size of the ith group will be $\binom{i}{k}$.

Finally, a recursive computation for the binomial coefficients.

A.48. Lemma. (Pascal's Formula) If $n \geq 1$, then $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

Proof: We count the k-sets in $[n]$. There are $\binom{n-1}{k}$ such sets not containing n and $\binom{n-1}{k-1}$ such sets containing n. ■

Given the initial conditions for $n = 0$, which are $\binom{0}{0} = 1$ and $\binom{0}{k} = 0$ for $k \neq 0$, Pascal's Formula can be used to give inductive proofs of many statements about binomial coefficients, including Theorems A.42–A.43.

A.49. Remark. *Multinomial coefficients.* Binomial coefficients and the binomial theorem generalize to multinomials. When $\sum n_i = n$, the **multinomial coefficient** $\binom{n}{n_1,\ldots,n_k}$ is the coefficient of $\prod x_i^{n_i}$ in the expansion of $(\sum_{i=1}^k x_i)^n$. It has the value $n!/\prod n_i!$. Terms of the form $\prod x_i^{n_i}$ arise in the expansion only when $\sum n_i = n$. Otherwise, there is nothing to count, and we say that $\binom{n}{n_1,\ldots,n_k} = 0$ when $\sum n_i \neq n$.

The contributions to this coefficient correspond to n-tuples that are arrangements of n objects, using n_i copies of object i for each i. Having a copy of object i in position j corresponds to choosing the term x_i from the jth factor $(x_1 + \cdots + x_k)$.

The formula $n!/\prod n_i!$ is derived by counting these arrangements. There are $n!$ arrangements of n distinct items. If we view these objects as distinct, then we count each arrangement $\prod n_i!$ times, since permuting the copies of a single object does not change the arrangement.

In Corollary 2.2.4, these arrangements correspond to trees with vertex set $[n]$ and specified vertex degrees. When we set $x_i = 1$ for all i, we obtain the total number of n-tuples formed from k types of letters over all multiplicities of repetition; the result is k^n. ■

RELATIONS

Given two objects s and t, not necessarily of the same type, we may ask whether they satisfy a given relationship. Let S denote the set of objects of the first type, and let T denote the set of objects of the second type. Some of the ordered pairs (s, t) may satisfy the relationship, and some may not. The next definition makes this notion precise.

A.50. Definition. When S and T are sets, a **relation** between S and T is a subset of the product $S \times T$. A **relation on** S is a subset of $S \times S$.

We usually specify a relation by a condition on pairs. In Section 1.1, we define several relations associated with a graph G. The *incidence relation* between $S = V(G)$ and $T = E(G)$ is the set of ordered pairs (v, e) such that $v \in V(G)$, $e \in E(G)$, and v is an endpoint of edge e. The *adjacency relation* on the set $V(G)$ is the set of ordered pairs (x, y) of vertices such that x and y are the endpoints of an edge.

A.51. Remark. Let R be a relation defined on a set S. When discussing several items from S, we use the adjective **pairwise** to specify that each pair among these items satisfies R. Thus we can talk about a family of pairwise disjoint sets, or a family of pairwise isomorphic graphs. An independent set in a graph is a set of pairwise nonadjacent vertices. A set of **distinct** objects is a set of pairwise unequal objects.

We need the term "pairwise" because the relation is defined for pairs. For the same reason, we don't use "pairwise" when discussing only two objects.

When two graphs are isomorphic, we don't say they are pairwise isomorphic. Similarly, we say that the endpoints of an edge are adjacent, not pairwise adjacent; the adjacency relation is satisfied by certain pairs of vertices. ∎

To specify a relation between S and T, we can list the ordered pairs satisfying it. Usually it is more convenient to let S index the rows and T the columns of a grid of positions called a **matrix**. We can then specify the relation by recording, in the position for row s and column t, a 1 if (s, t) satisfies the relation and a 0 if (s, t) does not satisfy the relation. Thus the adjacency and incidence matrices of a graph are the matrices recording the adjacency and incidence relations (see Definition 1.1.17).

The condition "have the same parity" defines a relation on $\mathbb{Z}$. If x, y are both even or both odd, then (x, y) satisfies this relation; otherwise it does not. The key properties of parity lead us to an important class of relations.

A.52. Definition. An **equivalence relation** on a set S is a relation R on S such that for all choices of distinct $x, y, z \in S$,
a) $(x, x) \in R$ (**reflexive property**).
b) $(x, y) \in R$ implies $(y, x) \in R$ (**symmetric property**).
c) $(x, y) \in R$ and $(y, z) \in R$ imply $(x, z) \in R$ (**transitive property**).

For every set S, the **equality relation** $R = \{(x, x)\colon x \in S\}$ is an equivalence relation on S. In Proposition 1.1.24 we show that the isomorphism relation is an equivalence relation on graphs. The notation $G \cong H$ for this relation suggests "equal in some sense".

A.53. Definition. Given an equivalence relation on S, the set of elements equivalent to $x \in S$ is the **equivalence class** containing x.

The equivalence classes of an equivalence relation on S form a partition of S; elements x and y belong to the same class if and only if (x, y) satisfies the relation. The converse assertion also holds. If $A_1, \dots, A_k$ is a partition of S, then the condition "x and y are in the same set in the partition" defines an equivalence relation on S.

Parity partitions the integers into two equivalence classes by their remainder upon division by 2. This notion generalizes to any natural number.

A.54. Definition. Given a natural number n, the integers x and y are **congruent modulo** n if $x - y$ is divisible by n. We write this as $x \equiv y \bmod n$. The number n is the **modulus**.

A.55. Theorem. For $n \in \mathbb{N}$, congruence mod n is an equivalence relation on $\mathbb{Z}$.

Proof: Reflexive property: $x - x$ equals 0, which is divisible by n.

Symmetric property: If $x \equiv y \bmod n$, then by definition $n|(x - y)$. Since $y - x = -(x - y)$, and since n divides $-m$ if and only if n divides m, we also have $n|(y - x)$, and hence $y \equiv x \bmod n$.

Transitive property: If $n|(x-y)$ and $n|(y-z)$, then integers a, b exist such that $x-y=an$ and $y-z=bn$. Adding these equations yields $x-z=an+bn=(a+b)n$, so $n|(x-z)$. Thus the relation is transitive. ■

A.56. Definition. The equivalence classes of the relation "congruence modulo n" on $\mathbb{Z}$ are the **remainder classes** or **congruence classes** modulo n. The set of congruence classes is written as $\mathbb{Z}_n$ or $\mathbb{Z}/n\mathbb{Z}$.

There are n remainder classes modulo n. For $0 \le r < n$, the rth class in $\mathbb{Z}_n$ is $\{kn+r\colon k \in \mathbb{Z}\}$. Numbers a and b lie in the rth class if and only if they both have remainder r upon division by n. Thus "$m \equiv r \bmod n$" has the same meaning as "m is r more than a multiple of n".

THE PIGEONHOLE PRINCIPLE

The pigeonhole principle is a simple notion that leads to elegant proofs and can reduce case analysis. In every set of numbers, the average is between the minimum and the maximum. When dealing with integers, the pigeonhole principle allows us to take the ceiling or floor of the average in the desired direction.

A.57. Lemma. (Pigeonhole Principle) If a set consisting of more than kn objects is partitioned into n classes, then some class receives more than k objects.

Proof: The contrapositive states that if every class receives at most k objects, then in total there are at most kn objects. ■

The pigeonhole principle can reduce case analysis by allowing us to use additional information about an extreme element of a set. This simple idea can crop up unexpectedly, but its use can be quite effective. When we find that we need the pigeonhole principle, there is no trouble applying it: we need a sufficiently big value in our set, and the pigeonhole principle provides it.

Some applications of the pigeonhole principle are rather subtle. Section 8.3 presents several of these. The subtlety arises when it is unclear how to define the objects and the classes so that the pigeonhole principle will apply.

Proposition 1.3.15 proves the next proposition using Remark A.13. Here we use the pigeonhole principle instead.

A.58. Proposition. If G is a simple n-vertex graph with $\delta(G) \ge (n-1)/2$, then G is connected.

Proof: Choose $u, v \in V(G)$. If $u \not\leftrightarrow v$, then at least $n-1$ edges join $\{u, v\}$ to the remaining vertices, since $\delta(G) \ge (n-1)/2$. There are $n-2$ other vertices, so the pigeonhole principle implies that one of them receives two of these edges. Since G is simple, this vertex is a common neighbor of u and v.

For every two vertices $u, v \in V(G)$, we have proved that u and v are adjacent or have a common neighbor. Thus G is connected. ■

The pigeonhole principle can also be useful in statements about trees, where the number of vertices is one more than the number of edges. If each vertex selects an edge in some way, then some edge must be selected twice. The idea is to design the selection so that when an edge is selected twice, the desired outcome occurs. Applications of this idea occur in Lemma 8.1.10 and Theorem 8.3.2.

The pigeonhole principle is the discrete version of the statement that the average of a set of numbers is between the minimum and the maximum. This statement is made explicit for vertex degrees in Corollary 1.3.4. Other applications are sprinkled throughout the book.

Appendix B

Optimization and Complexity

A salesman plans to visit $n - 1$ other cities and return home. The natural objective is to minimize the total travel time. If we assign each edge of K_n a weight equal to the travel time between the corresponding cities, then we seek the spanning cycle of minimum total weight. This is the famous **Traveling Salesman Problem (TSP)**. Seemingly analogous to the Minimum Spanning Tree problem, the TSP as yet has no good algorithm.

Similarly, although we have a good algorithm for finding maximum matchings, we have none for finding the maximum size of an independent set of vertices. Since the former is the special case of the latter for line graphs, it is not too surprising that it is easier to solve.

INTRACTABILITY

We defined a *good algorithm* (Definition 3.2.3) to be an algorithm that runs (correctly) in time bounded by a polynomial function of the input size. One algorithm for the TSP considers all spanning cycles and selects the cheapest one. This is not a good algorithm, because K_n has $(n - 1)!/2$ spanning cycles, and this has grows faster than every polynomial function of n. The computation takes too long for graphs of any substantial size. Practical applications require solving TSPs on graphs with hundreds or thousands of vertices.

No one has found a good algorithm, and no one has proved that none exists. The TSP belongs to a large class of problems having the property that a good algorithm for any one of them will yield a good algorithm for every one of them. A good algorithm for B yields a good algorithm for A if we can "reduce" problem A to problem B.

As an easy example of this, we can use a good algorithm for the TSP (problem B) to recognize Hamiltonian graphs (problem A). From a graph G, form an instance of the TSP on vertex set $V(G)$ by assigning weight 0 to vertex pairs that are edges of G and weight 1 to pairs that are not. The graph G has a Hamiltonian cycle if and only if the optimal solution to this instance of the TSP has cost 0. The time for the transformation is polynomial in $n(G)$, so a good algorithm for the TSP produces a good algorithm to test for spanning cycles. We conclude that the TSP is at least as hard as the Hamiltonian cycle problem.

In the formal discussion, we consider only **decision problems**, where the answer is YES or NO. This makes sense for recognizing Hamiltonian graphs, but the TSP is an optimization problem. When formulated as a decision problem (called MINIMUM SPANNING CYCLE), the input for the TSP is a weighted graph G and a number k, and the problem is to test whether G has a spanning cycle with weight at most k. Repeated applications of this decision problem (at most a polynomial number of applications) can be used to find the minimum weight of a spanning cycle. Similarly, MAXIMUM INDEPENDENT SET takes a graph G and an integer k as input and tests $\alpha(G) \geq k$.

We judge a graph algorithm by its maximum (*worst-case*) running time over inputs on n vertices, as a function of n. The **complexity** of a decision problem is the minimum worst-case running time over all solution algorithms, again as a function of the size of the problem.[†] In describing the growth of a function g, we compare it with a reference function f. We define several sets of functions in terms of f. The sets $O(f)$ and $\Omega(f)$ describe functions bounded above and below by multiples of f. Functions in $\Theta(f)$ grow at about the same rate as f, those in $o(f)$ grow more slowly, and those in $\omega(f)$ grow more quickly.

$$\begin{aligned}
O(f) &= \{g\colon \exists c, a \in \mathbb{R} \text{ such that } |g(x)| \leq c|f(x)| \text{for} x > a\} \\
\Omega(f) &= \{g\colon \exists c, a \in \mathbb{R} \text{ such that } |g(x)| \geq c|f(x)| \text{for} x > a\} \\
\Theta(f) &= O(f) \cap \Omega(f) \\
o(f) &= \{g\colon |g(x)|/|f(x)| \to 0\} \\
\omega(f) &= \{g : |g(x)|/|f(x)| \to \infty\}
\end{aligned}$$

The class of problems with polynomial complexity (solved by a good algorithm) is called "P". We have discussed only deterministic algorithms: each input leads to exactly one polynomial-time computation.

Now we consider nondeterministic algorithms. For many decision problems with no known good algorithm, short proofs exist for YES answers. For example, if we guess the right order of vertices in the HAMILTONIAN CYCLE problem (specified by a sequence of $O(n \log n)$ bits), then we can verify rapidly that this order forms a spanning cycle.

A **nondeterministic polynomial-time algorithm** tries all values of a polynomial-length sequence of bits simultaneously, applying a polynomial-time computation to each guess (polynomial in the length of the input). If any guess demonstrates a YES answer to the decision problem, then the algorithm says YES. Otherwise, the answer is NO. This amounts to saying that when the answer is YES, there is a polynomial-time proof of this. The nondeterminism lies not in the answer but in the choice of the computation path.

[†]Technically, the **size** of a problem instance is its length in some encoding of the problem. Measuring the size of a graph problem by the number of vertices suffices for our purposes. A polynomial in n is also a polynomial in n^2 or n^3, so the distinction is unimportant unless the problem involves exponentially large edge weights.

The class of problems solvable by nondeterministic polynomial-time algorithms is called "NP". A machine that has the power to follow many computation paths in parallel can also follow one; hence $P \subseteq NP$. It is commonly believed that $P \neq NP$. This has not been proved, so NP cannot be taken to mean "nonpolynomial". Instead, we use the informal term **intractable** for the problems in NP that are essentially as hard as all the problems in NP.

A problem is **NP-hard** if a polynomial-time algorithm for it could be used to construct a polynomial-time algorithm for each problem in NP. It is **NP-complete** if it belongs to NP and is NP-hard. If some NP-complete problem belongs to P, then $P = NP$. No polynomial-time algorithm is known for any of the many NP-complete problems. This supports the prevailing belief that $P \neq NP$. Garey–Johnson [1979] presents a thorough introduction to this topic.

Given one NP-complete problem, NP-completeness of other problems follows by reduction arguments as suggested earlier. We present several such arguments in this appendix. Here we list complexities for some of the problems discussed in this book.

Standard style in computer science uses uppercase names for decision problems. For a problem whose name includes an optimization, the decision problem is testing whether the value is as extreme as a number given as part of the input. Parameters in the name, however, are fixed as part of the statement of the problem.

This is an important distinction. For example, k-INDEPENDENT SET for fixed k is in P, since the number of k-sets of vertices is a polynomial in n of degree k when k is fixed, and we can simply test them all for independence. On the other hand, MAXIMUM INDEPENDENT SET is NP-complete; this is the problem of testing whether G has an independent set of size at least k where k is part of the input (and can grow with n).

Problems in P	NP-complete Problems
k-INDEPENDENT SET	MAXIMUM INDEPENDENT SET
GIRTH (SHORTEST CYCLE)	CIRCUMFERENCE (LONGEST CYCLE)
EULERIAN CIRCUIT	HAMILTONIAN CYCLE
DIAMETER	LONGEST PATH, HAMILTONIAN PATH
CONNECTIVITY	
2-COLORABILITY	k-COLORABILITY (for any fixed $k \geq 3$)
MAXIMUM MATCHING	$\Delta(G)$-EDGE-COLORABILITY
PLANARITY	GENUS

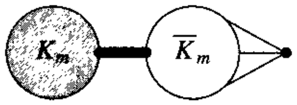
NP-completeness is related to the lack of an easily testable necessary and sufficient condition for YES answers. A **good characterization** is a characterization by a condition checkable in polynomial time. The characterization for Eulerian graphs is good, and GIRTH, DIAMETER, and 2-COLORABILITY can all be solved in polynomial time using Breadth-First Search. Polynomial behavior is less obvious for CONNECTIVITY, but min-max relations like Menger's Theorem generally lead to polynomial-time optimization algorithms, often based on network flow methods (Section 4.3).

HEURISTICS AND BOUNDS

Our traveling salesman still awaits instruction. NP-completeness does not eliminate the need for an answer. We seek heuristic algorithms that find solutions close to optimal. Perhaps we can prove a guarantee about how far from the optimum the result may be. For example, we may be content to have a solution whose cost is at most twice the optimum, if we have an algorithm that can quickly generate such a solution. An **approximation algorithm** always generates a solution whose ratio to the optimum is bounded by a constant.[†]

Greediness is a simple heuristic. For the minimum spanning tree problem, the result is optimal. On other problems, greedy algorithms may perform very badly. Consider MAXIMUM INDEPENDENT SET. We may generate an independent set iteratively by picking a vertex and deleting it and its neighbors. How should we pick the next vertex? If we always choose right, then the result is a maximum independent set. A greedy heuristic is to pick a vertex of minimum degree in what remains, since this leaves the largest set of candidates for the independent set. The result can be arbitrarily bad.

B.1. Example. *Defeating the greedy algorithm.* Consider $(K_1 + K_m) \vee \overline{K}_m$. This graph has one vertex of degree m, m vertices of degree $m+1$, and m vertices of degree $2m-1$. The greedy heuristic picks the vertex of minimum degree and deletes it and its neighbors, leaving a clique. Hence the greedy algorithm finds an independent set of size 2, when in fact $\alpha(G) = m$. ■



Nevertheless, the greedy algorithm works well on large graphs generated randomly. In this model (see Section 8.5), it almost always finds an independent set of size at least half the maximum. Exercise 12 presents two heuristics for MINIMUM VERTEX COVER; one fails as in Example B.1, but the other yields an approximation algorithm.

Next we consider simple heuristics for the TSP, where $\{v_1, \ldots, v_n\}$ are the vertices and w_{ij} denotes the weight (cost) of edge v_iv_j. From an arbitrary starting vertex, it seems reasonable to move to a new vertex via the least-cost incident edge. We iteratively move to the closest unvisited neighbor of the current vertex. This is a "greedy" algorithm and runs quickly. It is the **nearest-neighbor** heuristic.

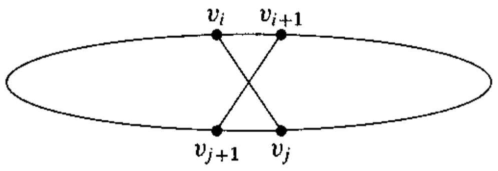
[†]Better yet: an **approximation scheme** is a family of algorithms indexed by a parameter ϵ, say $\epsilon = 1/k$ for the kth algorithm, such that the kth algorithm has performance ratio bounded by $1+\epsilon$. Each algorithm has polynomial complexity, but the degree of the polynomial grows with k. Improving the performance ratio requires more time.

B.2. Example. *Failure of the Nearest-neighbor heuristic.* Consider a TSP with weight 0 on a Hamiltonian path P, weight n^2 on all other edges incident to the endpoints of P, and weight 1 on all remaining edges. This example has many spanning cycles of weight n, but the nearest-neighbor heuristic yields a cycle of weight at least n^2 from any starting vertex. Thus the cost of the cycle produced by the algorithm is not bounded by a constant multiple of the optimal cost, and it is not an approximation algorithm. ∎

There are many similar heuristics. We could try to grow a cycle one vertex at a time, greedily absorbing the vertex whose insertion in the cycle causes the least increase in cost. This **nearest-insertion** heuristic has a better chance than nearest-neighbor, because at stage i of the nearest-neighbor heuristic we make a choice among $n - i$ alternatives, whereas at stage i in nearest-insertion we choose among $(n - i)i$ alternatives (which to add and where to insert it). Nevertheless, this also is not an approximation algorithm (Exercise 7).

Another approach is to start with a candidate spanning cycle and try to improve it. Maintaining a feasible solution (an actual cycle) and considering small changes to improve it is called **local search**. Allowing changes takes us beyond greedy algorithms and may perform better.

To improve the current cycle, we consider changing a pair of edges. If $(v_1, \dots, v_n)$ is our cycle, we could substitute the edges $v_i v_j$ and $v_{i+1}v_{j+1}$ for $v_i v_{i+1}$ and $v_j v_{j+1}$ to obtain a new cycle (the other possible switch leads to two disjoint cycles instead of one cycle). The switch is beneficial if $w_{i,j} + w_{i+1,j+1} < w_{i,i+1} + w_{j,j+1}$. The current cycle has $\binom{n}{2} - n = \binom{n-1}{2}$ pairs of nonincident edges to consider switching. The algorithm of Lin–Kernighan [1973], which has proved remarkably difficult to improve in practice, considers switches among three edges at a time.



The following theorem seems to doom efforts to find an approximation algorithm for the general TSP.

B.3. Theorem. (Sahni–Gonzalez [1976]) If there is a constant $c \geq 1$ and a polynomial-time algorithm A such that A produces for each instance of the TSP a spanning cycle with cost at most c times the optimum, then P=NP.

Proof: We show that such an algorithm A could be used to construct a polynomial-time algorithm for HAMILTONIAN CYCLE, which is NP-complete (Corollary B.11). Given an n-vertex graph G, construct an instance of TSP on the same vertex set by letting $w_{ij} = 1$ if $v_i v_j \in E(G)$, and $w_{ij} = cn$ otherwise.

In this instance of TSP, every spanning cycle with cost at most cn has cost exactly n and corresponds to a spanning cycle of the original input G. Since A