Mathematics

11

This chapter deals with mathematical topics that are recurrent in competitive programming. We will both discuss theoretical results and learn how to use them in practice in algorithms.

Section 11.1 discusses number-theoretical topics. We will learn algorithms for finding prime factors of numbers, techniques related to modular arithmetic, and efficient methods for solving integer equations.

Section 11.2 explores ways to approach combinatorial problems: how to efficiently count all valid combinations of objects. The topics of this section include binomial coefficients, Catalan numbers, and inclusion-exclusion.

Section 11.3 shows how to use matrices in algorithm programming. For example, we will learn how to make a dynamic programming algorithm more efficient by exploiting an efficient way to calculate matrix powers.

Section 11.4 first discusses basic techniques for calculating probabilities of events and the concept of Markov chains. After this, we will see examples of algorithms that are based on randomness.

Section 11.5 focuses on game theory. First, we will learn to optimally play a simple stick game using nim theory, and after this, we will generalize the strategy to a wide range of other games.

11.1 Number Theory

Number theory is a branch of mathematics that studies integers. In this section, we will discuss a selection of number-theoretical topics and algorithms, such as finding prime numbers and factors, and solving integer equations.

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_11

11.1.1 Primes and Factors

An integer a is called a *factor* or a *divisor* of an integer b if a divides b. If a is a factor of b, we write $a \mid b$, and otherwise we write $a \nmid b$. For example, the factors of 24 are 1, 2, 3, 4, 6, 8, 12, and 24.

An integer $n > 1$ is a *prime* if its only positive factors are 1 and n. For example, 7, 19, and 41 are primes, but 35 is not a prime, because $5 \cdot 7 = 35$. For every integer $n > 1$, there is a unique *prime factorization*

$$n = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_k^{\alpha_k},$$

where $p_1, p_2, \ldots, p_k$ are distinct primes and $\alpha_1, \alpha_2, \ldots, \alpha_k$ are positive integers. For example, the prime factorization for 84 is

$$84 = 2^2 \cdot 3^1 \cdot 7^1.$$

Let $\tau(n)$ denote the number of factors of an integer n. For example, $\tau(12) = 6$, because the factors of 12 are 1, 2, 3, 4, 6, and 12. To calculate the value of $\tau(n)$, we can use the formula

$$\tau(n) = \prod_{i=1}^{k} (\alpha_i + 1),$$

because for each prime p_i, there are $\alpha_i + 1$ ways to choose how many times it appears in the factor. For example, since $12 = 2^2 \cdot 3$, $\tau(12) = 3 \cdot 2 = 6$.

Then, let $\sigma(n)$ denote the sum of factors of an integer n. For example, $\sigma(12) = 28$, because $1 + 2 + 3 + 4 + 6 + 12 = 28$. To calculate the value of $\sigma(n)$, we can use the formula

$$\sigma(n) = \prod_{i=1}^{k} (1 + p_i + \cdots + p_i^{\alpha_i}) = \prod_{i=1}^{k} \frac{p_i^{\alpha_i+1} - 1}{p_i - 1},$$

where the latter form is based on the geometric progression formula. For example, $\sigma(12) = (2^3 - 1)/(2 - 1) \cdot (3^2 - 1)/(3 - 1) = 28$.

Basic Algorithms If an integer n is not prime, it can be represented as a product $a \cdot b$, where $a \leq \sqrt{n}$ or $b \leq \sqrt{n}$, so it certainly has a factor between 2 and $\lfloor \sqrt{n} \rfloor$. Using this observation, we can both test if an integer is prime and find its prime factorization in $O(\sqrt{n})$ time.

The following function `prime` checks if a given integer n is prime. The function attempts to divide n by all integers between 2 and $\lfloor \sqrt{n} \rfloor$, and if none of them divides n, then n is prime.

```
bool prime(int n) {
    if (n < 2) return false;
    for (int x = 2; x*x <= n; x++) {
        if (n%x == 0) return false;
    }
    return true;
}
```

Then, the following function `factors` constructs a vector that contains the prime factorization of n. The function divides n by its prime factors and adds them to the vector. The process ends when the remaining number n has no factors between 2 and $\lfloor\sqrt{n}\rfloor$. If $n > 1$, it is prime and the last factor.

```
vector<int> factors(int n) {
    vector<int> f;
    for (int x = 2; x*x <= n; x++) {
        while (n%x == 0) {
            f.push_back(x);
            n /= x;
        }
    }
    if (n > 1) f.push_back(n);
    return f;
}
```

Note that each prime factor appears in the vector as many times as it divides the number. For example, $12 = 2^2 \cdot 3$, so the result of the function is [2, 2, 3].

Properties of Primes It is easy to show that there is an infinite number of primes. If the number of primes would be finite, we could construct a set $P = \{p_1, p_2, \ldots, p_n\}$ that would contain all the primes. For example, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, and so on. However, using such a set P, we could form a new prime

$$p_1 p_2 \cdots p_n + 1$$

that would be larger than all elements in P. This is a contradiction, and the number of primes has to be infinite.

The *prime-counting function* $\pi(n)$ gives the number of primes up to n. For example, $\pi(10) = 4$, because the primes up to 10 are 2, 3, 5, and 7. It is possible to show that

$$\pi(n) \approx \frac{n}{\ln n},$$

which means that primes are quite frequent. For example, an approximation for $\pi(10^6)$ is $10^6 / \ln 10^6 \approx 72382$, and the exact value is 78498.

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	1	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1

Fig. 11.1 Outcome of the sieve of Eratosthenes for $n = 20$

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	3	2	5	2	7	2	3	2	11	2	13	2	3	2	17	2	19	2

Fig. 11.2 An extended sieve of Eratosthenes that contains the smallest prime factor of each number

11.1.2 Sieve of Eratosthenes

The *sieve of Eratosthenes* is a preprocessing algorithm that constructs an array `sieve` from which we can efficiently check if any integer x between $2 \ldots n$ is prime. If x is prime, then `sieve`$[x] = 0$, and otherwise `sieve`$[x] = 1$. For example, Fig. 11.1 shows the contents of `sieve` for $n = 20$.

To construct the array, the algorithm iterates through the integers $2 \ldots n$ one by one. Always when a new prime x is found, the algorithm records that the numbers $2x$, $3x$, $4x$, etc., are not primes. The algorithm can be implemented as follows, assuming that every element of `sieve` is initially zero:

```
for (int x = 2; x <= n; x++) {
    if (sieve[x]) continue;
    for (int u = 2*x; u <= n; u += x) {
        sieve[u] = 1;
    }
}
```

The inner loop of the algorithm is executed $\lfloor n/x \rfloor$ times for each value of x. Thus, an upper bound for the running time of the algorithm is the harmonic sum

$$\sum_{x=2}^{n} \lfloor n/x \rfloor = \lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \lfloor n/4 \rfloor + \cdots = O(n \log n).$$

In fact, the algorithm is more efficient, because the inner loop will be executed only if the number x is prime. It can be shown that the running time of the algorithm is only $O(n \log \log n)$, a complexity very near to $O(n)$. In practice, the sieve of Eratosthenes is very efficient; Table 11.1 shows some real running times.

There are several ways to extend the sieve of Eratosthenes. For example, we can calculate for each number k its smallest prime factor (Fig. 11.2). After this, we can efficiently factorize any number between $2 \ldots n$ using the sieve. (Note that a number n has $O(\log n)$ prime factors.)

Table 11.1 Running times of the sieve of Eratosthenes

Upper bound n	Running time (s)
10^6	0.01
$2 \cdot 10^6$	0.03
$4 \cdot 10^6$	0.07
$8 \cdot 10^6$	0.14
$16 \cdot 10^6$	0.28
$32 \cdot 10^6$	0.57
$64 \cdot 10^6$	1.16
$128 \cdot 10^6$	2.35

11.1.3 Euclid's Algorithm

The *greatest common divisor* of integers a and b, denoted $\gcd(a, b)$, is the largest integer that divides both a and b. For example, $\gcd(30, 12) = 6$. A related concept is the *lowest common multiple*, denoted $\text{lcm}(a, b)$, which is the smallest integer that is divisible by both a and b. The formula

$$\text{lcm}(a, b) = \frac{ab}{\gcd(a, b)}$$

can be used to calculate lowest common multiples. For example, $\text{lcm}(30, 12) = 360/\gcd(30, 12) = 60$.

One way to find $\gcd(a, b)$ is to divide a and b into prime factors, and then choose for each prime the largest power that appears in both factorizations. For example, to calculate $\gcd(30, 12)$, we can construct the factorizations $30 = 2 \cdot 3 \cdot 5$ and $12 = 2^2 \cdot 3$, and conclude that $\gcd(30, 12) = 2 \cdot 3 = 6$. However, this technique is not efficient if a and b are large numbers.

Euclid's algorithm provides an efficient way to calculate the value of $\gcd(a, b)$. The algorithm is based on the formula

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0. \end{cases}$$

For example,

$$\gcd(30, 12) = \gcd(12, 6) = \gcd(6, 0) = 6.$$

The algorithm can be implemented as follows:

```
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a%b);
}
```

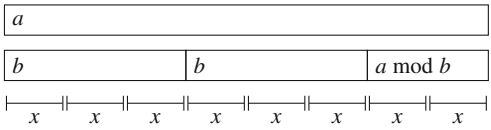


Fig. 11.3 Why does Euclid's algorithm work?

Why does the algorithm work? To understand this, consider Fig. 11.3. where $x = \gcd(a, b)$. Since x divides both a and b, it must also divide $a \bmod b$, which shows why the recursive formula holds.

It can be proved that Euclid's algorithm works in $O(\log n)$ time, where $n = \min(a, b)$.

Extended Euclid's Algorithm Euclid's algorithm can also be extended so that it gives integers x and y for which

$$ax + by = \gcd(a, b).$$

For example, when $a = 30$ and $b = 12$,

$$30 \cdot 1 + 12 \cdot (-2) = 6.$$

We can solve also this problem using the formula $\gcd(a, b) = \gcd(b, a \bmod b)$. Suppose that we have already solved the problem for $\gcd(b, a \bmod b)$, and we know values x' and y' for which

$$bx' + (a \bmod b)y' = gcd(a, b).$$

Then, since $a \bmod b = a - \lfloor a/b \rfloor \cdot b$,

$$bx' + (a - \lfloor a/b \rfloor \cdot b)y' = gcd(a, b),$$

which equals

$$ay' + b(x' - \lfloor a/b \rfloor \cdot y') = gcd(a, b).$$

Thus, we can choose $x = y'$ and $y = x' - \lfloor a/b \rfloor \cdot y'$. Using this idea, the following function returns a tuple $(x, y, \gcd(a, b))$ that satisfies the equation.

```
tuple<int,int,int> gcd(int a, int b) {
    if (b == 0) {
        return {1,0,a};
    } else {
        int x,y,g;
        tie(x,y,g) = gcd(b,a%b);
        return {y,x-(a/b)*y,g};
    }
}
```

We can use the function as follows:

```
int x,y,g;
tie(x,y,g) = gcd(30,12);
cout << x << " " << y << " " << g << "\n"; // 1 -2 6
```

11.1.4 Modular Exponentiation

There is often a need to efficiently calculate the value of $x^n \bmod m$. This can be done in $O(\log n)$ time using the following recursive formula:

$$x^n = \begin{cases} 1 & n = 0 \\ x^{n/2} \cdot x^{n/2} & n \text{ is even} \\ x^{n-1} \cdot x & n \text{ is odd} \end{cases}$$

For example, to calculate the value of x^{100}, we first calculate the value of x^{50} and then use the formula $x^{100} = x^{50} \cdot x^{50}$. Then, to calculate the value of x^{50}, we first calculate the value of x^{25} and so on. Since n always halves when it is even, the calculation takes only $O(\log n)$ time.

The algorithm can be implemented as follows:

```
int modpow(int x, int n, int m) {
    if (n == 0) return 1%m;
    long long u = modpow(x,n/2,m);
    u = (u*u)%m;
    if (n%2 == 1) u = (u*x)%m;
    return u;
}
```

11.1.5 Euler's Theorem

Two integers a and b are called *coprime* if $\gcd(a, b) = 1$. *Euler's totient function* $\varphi(n)$ gives the number of integers between $1 \ldots n$ that are coprime to n. For example, $\varphi(10) = 4$, because 1, 3, 7, and 9 are coprime to 10.

Any value of $\varphi(n)$ can be calculated from the prime factorization of n using the formula

$$\varphi(n) = \prod_{i=1}^{k} p_i^{\alpha_i - 1}(p_i - 1).$$

For example, since $10 = 2 \cdot 5$, $\varphi(10) = 2^0 \cdot (2-1) \cdot 5^0 \cdot (5-1) = 4$.

Euler's theorem states that

$$x^{\varphi(m)} \bmod m = 1$$

for all positive coprime integers x and m. For example, Euler's theorem tells us that $7^4 \bmod 10 = 1$, because 7 and 10 are coprime and $\varphi(10) = 4$.

If m is prime, $\varphi(m) = m - 1$, so the formula becomes

$$x^{m-1} \bmod m = 1,$$

which is known as *Fermat's little theorem*. This also implies that

$$x^n \bmod m = x^{n \bmod (m-1)} \bmod m,$$

which can be used to calculate values of x^n if n is very large.

Modular Multiplicative Inverses The *modular multiplicative inverse* of x with respect to m is a value $\text{inv}_m(x)$ such that

$$x \cdot \text{inv}_m(x) \bmod m = 1.$$

For example, $\text{inv}_{17}(6) = 3$, because $6 \cdot 3 \bmod 17 = 1$.

Using modular multiplicative inverses, we can divide numbers modulo m, because division by x corresponds to multiplication by $\text{inv}_m(x)$. For example, since we know that $\text{inv}_{17}(6) = 3$, we can calculate the value of $36/6 \bmod 17$ in another way using the formula $36 \cdot 3 \bmod 17$.

A modular multiplicative inverse exists exactly when x and m are coprime. In this case, it can be calculated using the formula

$$\text{inv}_m(x) = x^{\varphi(m)-1},$$

which is based on Euler's theorem. In particular, if m is prime, $\varphi(m) = m - 1$ and the formula becomes

$$\text{inv}_m(x) = x^{m-2}.$$

For example,

$$\text{inv}_{17}(6) \bmod 17 = 6^{17-2} \bmod 17 = 3.$$

The above formula allows us to efficiently calculate modular multiplicative inverses using the modular exponentiation algorithm (Sect. 11.1.4).

11.1.6 Solving Equations

Diophantine Equations A *Diophantine equation* is an equation of the form

$$ax + by = c,$$

where a, b, and c are constants and the values of x and y should be found. Each number in the equation has to be an integer. For example, one solution to the equation

$$5x + 2y = 11$$

is $x = 3$ and $y = -2$.

We can efficiently solve a Diophantine equation by using the extended Euclid's algorithm (Sect. 11.1.3) which gives integers x and y that satisfy the equation

$$ax + by = \gcd(a, b).$$

A Diophantine equation can be solved exactly when c is divisible by $\gcd(a, b)$.

As an example, let us find integers x and y that satisfy the equation

$$39x + 15y = 12.$$

The equation can be solved, because $\gcd(39, 15) = 3$ and $3 \mid 12$. The extended Euclid's algorithm gives us

$$39 \cdot 2 + 15 \cdot (-5) = 3,$$

and by multiplying this by 4, the equation becomes

$$39 \cdot 8 + 15 \cdot (-20) = 12,$$

so a solution to the equation is $x = 8$ and $y = -20$.

A solution to a Diophantine equation is not unique, because we can form an infinite number of solutions if we know one solution. If a pair (x, y) is a solution, then also all pairs

$$\left(x + \frac{kb}{\gcd(a, b)}, y - \frac{ka}{\gcd(a, b)}\right)$$

are solutions, where k is any integer.

Chinese Remainder Theorem The *Chinese remainder theorem* solves a group of equations of the form

$$\begin{aligned} x &= a_1 \bmod m_1 \\ x &= a_2 \bmod m_2 \\ &\cdots \\ x &= a_n \bmod m_n \end{aligned}$$

where all pairs of $m_1, m_2, \ldots, m_n$ are coprime.

It turns out that a solution to the equations is

$$x = a_1 X_1 \text{inv}_{m_1}(X_1) + a_2 X_2 \text{inv}_{m_2}(X_2) + \cdots + a_n X_n \text{inv}_{m_n}(X_n),$$

where

$$X_k = \frac{m_1 m_2 \cdots m_n}{m_k}.$$

In this solution, for each $k = 1, 2, \ldots, n$,

$$a_k X_k \text{inv}_{m_k}(X_k) \bmod m_k = a_k,$$

because

$$X_k \text{inv}_{m_k}(X_k) \bmod m_k = 1.$$

Since all other terms in the sum are divisible by m_k, they have no effect on the remainder and $x \bmod m_k = a_k$.

For example, a solution for

$$\begin{aligned} x &= 3 \bmod 5 \\ x &= 4 \bmod 7 \\ x &= 2 \bmod 3 \end{aligned}$$

is

$$3 \cdot 21 \cdot 1 + 4 \cdot 15 \cdot 1 + 2 \cdot 35 \cdot 2 = 263.$$

Once we have found a solution x, we can create an infinite number of other solutions, because all numbers of the form

$$x + m_1 m_2 \cdots m_n$$

are solutions.

11.2 Combinatorics

Combinatorics studies methods for counting combinations of objects. Usually, the goal is to find a way to count the combinations efficiently without generating each combination separately. In this section, we discuss a selection of combinatorial techniques that can be applied to a large number of problems.

11.2.1 Binomial Coefficients

The *binomial coefficient* $\binom{n}{k}$ gives the number of ways we can choose a subset of k elements from a set of n elements. For example, $\binom{5}{3} = 10$, because the set $\{1, 2, 3, 4, 5\}$ has 10 subsets of 3 elements:

$$\{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 5\}, \{1, 3, 4\}, \{1, 3, 5\},$$

$$\{1, 4, 5\}, \{2, 3, 4\}, \{2, 3, 5\}, \{2, 4, 5\}, \{3, 4, 5\}$$

Binomial coefficients can be recursively calculated using the formula

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

with the base cases

$$\binom{n}{0} = \binom{n}{n} = 1.$$

To see why this formula works, consider an arbitrary element x in the set. If we decide to include x in our subset, the remaining task is to choose $k - 1$ elements from $n - 1$ elements. Then, if we do not include x in our subset, we have to choose k elements from $n - 1$ elements.

Another way to calculate binomial coefficients is to use the formula

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

which is based on the following reasoning: There are $n!$ permutations of n elements. We go through all permutations and always include the first k elements of the permutation in the subset. Since the order of the elements in the subset and outside the subset does not matter, the result is divided by $k!$ and $(n - k)!$

For binomial coefficients,

$$\binom{n}{k} = \binom{n}{n-k},$$

because we actually divide a set of n elements into two subsets: the first contains k elements and the second contains $n - k$ elements.

The sum of binomial coefficients is

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = 2^n.$$

The reason for the name "binomial coefficient" can be seen when the binomial $(a + b)$ is raised to the nth power:

$$(a+b)^n = \binom{n}{0}a^n b^0 + \binom{n}{1}a^{n-1}b^1 + \cdots + \binom{n}{n-1}a^1 b^{n-1} + \binom{n}{n}a^0 b^n.$$

Fig. 11.4 First 5 rows of Pascal's triangle

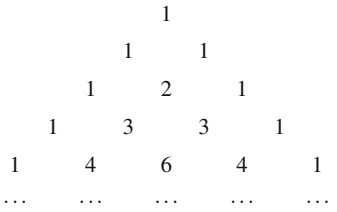
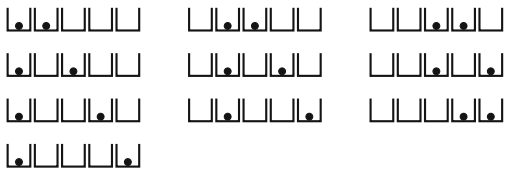


Fig. 11.5 Scenario 1: Each box contains at most one ball



Binomial coefficients also appear in *Pascal's triangle* (Fig. 11.4) where each value equals the sum of two above values.

Multinomial Coefficients The *multinomial coefficient*

$$\binom{n}{k_1, k_2, \ldots, k_m} = \frac{n!}{k_1! k_2! \cdots k_m!},$$

gives the number of ways a set of n elements can be divided into subsets of sizes $k_1, k_2, \ldots, k_m$, where $k_1 + k_2 + \cdots + k_m = n$. Multinomial coefficients can be seen as a generalization of binomial coefficients; if $m = 2$, the above formula corresponds to the binomial coefficient formula.

Boxes and Balls "Boxes and balls" is a useful model, where we count the ways to place k balls in n boxes. Let us consider three scenarios:

Scenario 1: Each box can contain at most one ball. For example, when $n = 5$ and $k = 2$, there are 10 combinations (Fig. 11.5). In this scenario, the number of combinations is directly the binomial coefficient $\binom{n}{k}$.

Scenario 2: A box can contain multiple balls. For example, when $n = 5$ and $k = 2$, there are 15 combinations (Fig. 11.6). In this scenario, the process of placing the balls in the boxes can be represented as a string that consists of symbols "o" and "$\rightarrow$." Initially, assume that we are standing at the leftmost box. The symbol "o" means that we place a ball in the current box, and the symbol "$\rightarrow$" means that we move to the next box to the right. Now each solution is a string of length $k + n - 1$ that contains k symbols "o" and $n - 1$ symbols "$\rightarrow$." For example, the upper-right solution in Fig. 11.6 corresponds to the string "$\rightarrow \rightarrow$ o $\rightarrow$ o $\rightarrow$." Thus, we can conclude that the number of combinations is $\binom{k+n-1}{k}$.

Scenario 3: Each box may contain at most one ball, and in addition, no two adjacent boxes may both contain a ball. For example, when $n = 5$ and $k = 2$, there are 6 combinations (Fig. 11.7). In this scenario, we can assume that k balls are initially placed in the boxes and there is an empty box between each two adjacent boxes. The

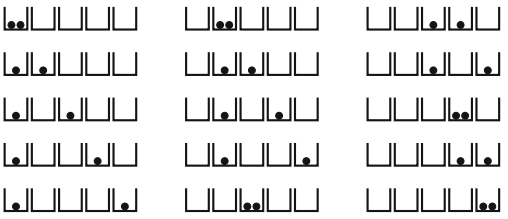


Fig. 11.6 Scenario 2: A box may contain multiple balls

Fig. 11.7 Scenario 3: Each box contains at most one ball and no two adjacent boxes contain a ball

remaining task is to choose the positions for the remaining empty boxes. There are $n - 2k + 1$ such boxes and $k + 1$ positions for them. Thus, using the formula of Scenario 2, the number of solutions is $\binom{n-k+1}{n-2k+1}$.

11.2.2 Catalan Numbers

The *Catalan number* C_n gives the number of valid parenthesis expressions that consist of n left parentheses and n right parentheses. For example, $C_3 = 5$, because we can construct a total of five parenthesis expressions using three left parentheses and three right parentheses:

- ( ) ( ) ( )
- (( )) ( )
- ( ) (( ))
- ((( )))
- (( ) ( ))

What is exactly a *valid parenthesis expression*? The following rules precisely define all valid parenthesis expressions:

- An empty parenthesis expression is valid.
- If an expression A is valid, then also the expression (A) is valid.
- If expressions A and B are valid, then also the expression AB is valid.

Another way to characterize valid parenthesis expressions is that if we choose any prefix of such an expression, it has to contain at least as many left parentheses as right parentheses, and the complete expression has to contain an equal number of left and right parentheses.

Catalan numbers can be calculated using the formula

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}$$

where we consider the ways to divide the parenthesis expression into two parts that are both valid parenthesis expressions, and the first part is as short as possible but not empty. For each i, the first part contains $i+1$ pairs of parentheses and the number of valid expressions is the product of the following values:

- C_i: the number of ways to construct a parenthesis expression using the parentheses of the first part, not counting the outermost parentheses
- C_{n-i-1}: the number of ways to construct a parenthesis expression using the parentheses of the second part

The base case is $C_0 = 1$, because we can construct an empty parenthesis expression using zero pairs of parentheses.

Catalan numbers can also be calculated using the formula

$$C_n = \frac{1}{n+1}\binom{2n}{n},$$

which can be explained as follows:

There are a total of $\binom{2n}{n}$ ways to construct a (not necessarily valid) parenthesis expression that contains n left parentheses and n right parentheses. Let us calculate the number of such expressions that are *not* valid.

If a parenthesis expression is not valid, it has to contain a prefix where the number of right parentheses exceeds the number of left parentheses. The idea is to pick the shortest such prefix and reverse each parenthesis in the prefix. For example, the expression `())()(` has the prefix `())`, and after reversing the parentheses, the expression becomes `)((()(`. The resulting expression consists of $n+1$ left and $n-1$ right parentheses. In fact, there is a unique way to produce any expression of $n+1$ left and $n-1$ right parentheses in the above manner. The number of such expressions is $\binom{2n}{n+1}$, which equals the number of nonvalid parenthesis expressions. Thus, the number of valid parenthesis expressions can be calculated using the formula

$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n}.$$

Counting Trees We can also count certain tree structures using Catalan numbers. First, C_n equals the number of binary trees of n nodes, assuming that left and right children are distinguished. For example, since $C_3 = 5$, there are 5 binary trees of 3 nodes (Fig. 11.8). Then, C_n also equals the number of general rooted trees of $n+1$ nodes. For example, there are 5 rooted trees of 4 nodes (Fig. 11.9).

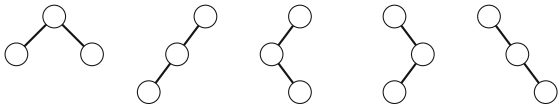


Fig. 11.8 There are 5 binary trees of 3 nodes

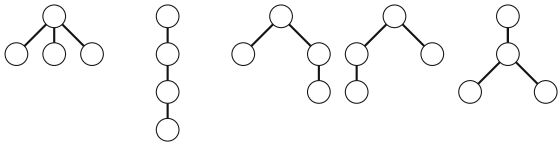


Fig. 11.9 There are 5 rooted trees of 4 nodes

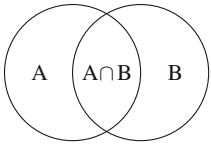


Fig. 11.10
Inclusion-exclusion principle for two sets

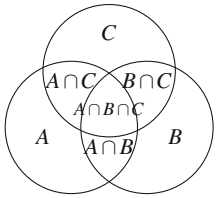


Fig. 11.11
Inclusion-exclusion principle for three sets

11.2.3 Inclusion-Exclusion

Inclusion-exclusion is a technique that can be used for counting the size of a union of sets when the sizes of the intersections are known, and vice versa. A simple example of the technique is the formula

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

where A and B are sets and $|X|$ denotes the size of X. Figure 11.10 illustrates the formula. In this case, we want to calculate the size of the union $A \cup B$ that corresponds to the area of the region that belongs to at least one circle in Fig. 11.10. We can calculate the area of $A \cup B$ by first summing up the areas of A and B and then subtracting the area of $A \cap B$ from the result.

The same idea can be applied when the number of sets is larger. When there are three sets, the inclusion-exclusion formula is

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

which corresponds to Fig. 11.11.

In the general case, the size of the union $X_1 \cup X_2 \cup \cdots \cup X_n$ can be calculated by going through all possible intersections that contain some of the sets $X_1, X_2, \ldots, X_n$.

If an intersection contains an odd number of sets, its size is added to the answer, and otherwise its size is subtracted from the answer.

Note that there are similar formulas for calculating the size of an intersection from the sizes of unions. For example,

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

and

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

Counting Derangements As an example, let us count the number of *derangements* of $\{1, 2, \ldots, n\}$, i.e., permutations where no element remains in its original place. For example, when $n = 3$, there are two derangements: (2, 3, 1) and (3, 1, 2).

One approach for solving the problem is to use inclusion-exclusion. Let X_k be the set of permutations that contain the element k at position k. For example, when $n = 3$, the sets are as follows:

$$\begin{aligned} X_1 &= \{(1, 2, 3), (1, 3, 2)\} \\ X_2 &= \{(1, 2, 3), (3, 2, 1)\} \\ X_3 &= \{(1, 2, 3), (2, 1, 3)\} \end{aligned}$$

The number of derangements equals

$$n! - |X_1 \cup X_2 \cup \cdots \cup X_n|,$$

so it suffices to calculate $|X_1 \cup X_2 \cup \cdots \cup X_n|$. Using inclusion-exclusion, this reduces to calculating sizes of intersections. Moreover, an intersection of c distinct sets X_k has $(n - c)!$ elements, because such an intersection consists of all permutations that contain c elements in their original places. Thus, we can efficiently calculate the sizes of the intersections. For example, when $n = 3$,

$$\begin{aligned} |X_1 \cup X_2 \cup X_3| &= |X_1| + |X_2| + |X_3| \\ &\quad -|X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| \\ &\quad +|X_1 \cap X_2 \cap X_3| \\ &= 2 + 2 + 2 - 1 - 1 - 1 + 1 \\ &= 4, \end{aligned}$$

so the number of derangements is $3! - 4 = 2$.

It turns out that the problem can also be solved *without* using inclusion-exclusion. Let $f(n)$ denote the number of derangements for $\{1, 2, \ldots, n\}$. We can use the following recursive formula:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases}$$

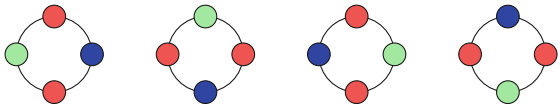


Fig. 11.12 Four symmetric necklaces

The formula can be proved by considering the possibilities how the element 1 changes in the derangement. There are $n-1$ ways to choose an element x that replaces the element 1. In each such choice, there are two options:

Option 1: We also replace the element x with the element 1. After this, the remaining task is to construct a derangement of $n-2$ elements.

Option 2: We replace the element x with some other element than 1. Now we have to construct a derangement of $n-1$ element, because we cannot replace the element x with the element 1, and all other elements must be changed.

11.2.4 Burnside's Lemma

Burnside's lemma can be used to count the number of distinct combinations so that symmetric combinations are counted only once. Burnside's lemma states that the number of combinations is

$$\frac{1}{n}\sum_{k=1}^{n} c(k),$$

where there are n ways to change the position of a combination, and there are $c(k)$ combinations that remain unchanged when the kth way is applied.

As an example, let us calculate the number of necklaces of n pearls, where each pearl has m possible colors. Two necklaces are symmetric if they are similar after rotating them. For example, Fig. 11.12 shows four symmetric necklaces, which should be counted as a single combination.

There are n ways to change the position of a necklace, because it can be rotated $k = 0, 1, \ldots, n-1$ steps clockwise. For example, if $k = 0$, all m^n necklaces remain the same, and if $k = 1$, only the m necklaces where each pearl has the same color remain the same. In the general case, a total of $m^{\gcd(k,n)}$ necklaces remain the same, because blocks of pearls of size $\gcd(k,n)$ will replace each other. Thus, according to Burnside's lemma, the number of distinct necklaces is

$$\frac{1}{n}\sum_{k=0}^{n-1} m^{\gcd(k,n)}.$$

For example, the number of distinct necklaces of 4 pearls and 3 colors is

$$\frac{3^4+3+3^2+3}{4} = 24.$$

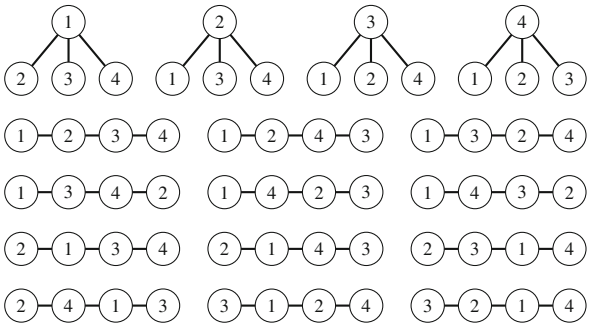


Fig. 11.13 There are 16 distinct labeled trees of 4 nodes

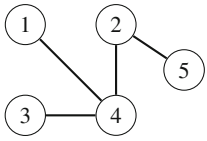


Fig. 11.14 Prüfer code of this tree is [4, 4, 2]

11.2.5 Cayley's Formula

Cayley's formula states that there are a total of n^{n-2} distinct labeled trees of n nodes. The nodes are labeled $1, 2, \ldots, n$, and two trees are considered distinct if either their structure or labeling is different. For example, when $n = 4$, there are $4^{4-2} = 16$ labeled trees, shown in Fig. 11.13.

Cayley's formula can be proved using Prüfer codes. A *Prüfer code* is a sequence of $n - 2$ numbers that describes a labeled tree. The code is constructed by following a process that removes $n - 2$ leaves from the tree. At each step, the leaf with the smallest label is removed, and the label of its only neighbor is added to the code. For example, the Prüfer code of the tree in Fig. 11.14 is [4, 4, 2], because we remove leaves 1, 3, and 4.

We can construct a Prüfer code for any tree, and more importantly, the original tree can be reconstructed from a Prüfer code. Hence, the number of labeled trees of n nodes equals n^{n-2}, the number of Prüfer codes of length n.

11.3 Matrices

A *matrix* is a mathematical concept that corresponds to a two-dimensional array in programming. For example,

$$A = \begin{bmatrix} 6 & 13 & 7 & 4 \\ 7 & 0 & 8 & 2 \\ 9 & 5 & 4 & 18 \end{bmatrix}$$

is a matrix of size 3×4, i.e., it has 3 rows and 4 columns. The notation $[i, j]$ refers to the element in row i and column j in a matrix. For example, in the above matrix, $A[2, 3] = 8$ and $A[3, 1] = 9$.

A special case of a matrix is a *vector* that is a one-dimensional matrix of size $n \times 1$. For example,

$$V = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

is a vector that contains three elements.

The *transpose* A^T of a matrix A is obtained when the rows and columns of A are swapped, i.e., $A^T[i, j] = A[j, i]$:

$$A^T = \begin{bmatrix} 6 & 7 & 9 \\ 13 & 0 & 5 \\ 7 & 8 & 4 \\ 4 & 2 & 18 \end{bmatrix}$$

A matrix is a *square matrix* if it has the same number of rows and columns. For example, the following matrix is a square matrix:

$$S = \begin{bmatrix} 3 & 12 & 4 \\ 5 & 9 & 15 \\ 0 & 2 & 4 \end{bmatrix}$$

11.3.1 Matrix Operations

The sum $A + B$ of matrices A and B is defined if the matrices are of the same size. The result is a matrix where each element has the sum of the corresponding elements in A and B. For example,

$$\begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} + \begin{bmatrix} 4 & 9 & 3 \\ 8 & 1 & 3 \end{bmatrix} = \begin{bmatrix} 6+4 & 1+9 & 4+3 \\ 3+8 & 9+1 & 2+3 \end{bmatrix} = \begin{bmatrix} 10 & 10 & 7 \\ 11 & 10 & 5 \end{bmatrix}.$$

Multiplying a matrix A by a value x means that each element of A is multiplied by x. For example,

$$2 \cdot \begin{bmatrix} 6 & 1 & 4 \\ 3 & 9 & 2 \end{bmatrix} = \begin{bmatrix} 2 \cdot 6 & 2 \cdot 1 & 2 \cdot 4 \\ 2 \cdot 3 & 2 \cdot 9 & 2 \cdot 2 \end{bmatrix} = \begin{bmatrix} 12 & 2 & 8 \\ 6 & 18 & 4 \end{bmatrix}.$$

The product AB of matrices A and B is defined if A is of size $a \times n$ and B is of size $n \times b$, i.e., the width of A equals the height of B. The result is a matrix of size $a \times b$ whose elements are calculated using the formula

$$AB[i, j] = \sum_{k=1}^{n} (A[i, k] \cdot B[k, j]).$$

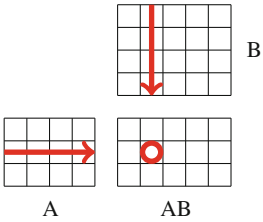


Fig. 11.15 Intuition behind the matrix multiplication formula

The idea is that each element of AB is a sum of products of elements of A and B according to Fig. 11.15. For example,

$$\begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 6 \\ 2 & 9 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 4 \cdot 2 & 1 \cdot 6 + 4 \cdot 9 \\ 3 \cdot 1 + 9 \cdot 2 & 3 \cdot 6 + 9 \cdot 9 \\ 8 \cdot 1 + 6 \cdot 2 & 8 \cdot 6 + 6 \cdot 9 \end{bmatrix} = \begin{bmatrix} 9 & 42 \\ 21 & 99 \\ 20 & 102 \end{bmatrix}.$$

We can directly use the above formula to calculate the product C of two $n \times n$ matrices A and B in $O(n^3)$ time[1]:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        for (int k = 1; k <= n; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

Matrix multiplication is associative, so $A(BC) = (AB)C$ holds, but it is not commutative, so usually $AB \neq BA$.

An *identity matrix* is a square matrix where each element on the diagonal is 1 and all other elements are 0. For example, the following matrix is the 3×3 identity matrix:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Multiplying a matrix by an identity matrix does not change it. For example,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 3 & 9 \\ 8 & 6 \end{bmatrix}.$$

[1] While the straightforward $O(n^3)$ time algorithm is sufficient in competitive programming, there are *theoretically* more efficient algorithms. In 1969, Strassen [31] discovered the first such algorithm, now called *Strassen's algorithm*, whose time complexity is $O(n^{2.81})$. The best current algorithm, proposed by Le Gall [11] in 2014, works in $O(n^{2.37})$ time.

The power A^k of a matrix A is defined if A is a square matrix. The definition is based on matrix multiplication:

$$A^k = \underbrace{A \cdot A \cdot A \cdots A}_{k \text{ times}}$$

For example,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^3 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix} = \begin{bmatrix} 48 & 165 \\ 33 & 114 \end{bmatrix}.$$

In addition, A^0 is an identity matrix. For example,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The matrix A^k can be efficiently calculated in $O(n^3 \log k)$ time using the algorithm in Sect. 11.1.4. For example,

$$\begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^8 = \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4 \cdot \begin{bmatrix} 2 & 5 \\ 1 & 4 \end{bmatrix}^4.$$

11.3.2 Linear Recurrences

A *linear recurrence* is a function $f(n)$ whose initial values are $f(0)$, $f(1)$, $\ldots$, $f(k-1)$ and larger values are calculated recursively using the formula

$$f(n) = c_1 f(n-1) + c_2 f(n-2) + \cdots + c_k f(n-k),$$

where $c_1, c_2, \ldots, c_k$ are constant coefficients.

Dynamic programming can be used to calculate any value of $f(n)$ in $O(kn)$ time by calculating all values of $f(0)$, $f(1)$, $\ldots$, $f(n)$ one after another. However, as we will see next, we can also calculate the value of $f(n)$ in $O(k^3 \log n)$ time using matrix operations. This is an important improvement if k is small and n is large.

Fibonacci Numbers A simple example of a linear recurrence is the following function that defines the Fibonacci numbers:

$$\begin{aligned} f(0) &= 0 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2) \end{aligned}$$

In this case, $k = 2$ and $c_1 = c_2 = 1$.

To efficiently calculate Fibonacci numbers, we represent the Fibonacci formula as a square matrix X of size 2×2, for which the following holds:

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \end{bmatrix}$$

Thus, values $f(i)$ and $f(i+1)$ are given as "input" for X, and X calculates values $f(i+1)$ and $f(i+2)$ from them. It turns out that such a matrix is

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}.$$

For example,

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} f(5) \\ f(6) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 13 \end{bmatrix} = \begin{bmatrix} f(6) \\ f(7) \end{bmatrix}.$$

Thus, we can calculate $f(n)$ using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

The value of X^n can be calculated in $O(\log n)$ time, so the value of $f(n)$ can also be calculated in $O(\log n)$ time.

General Case Let us now consider the general case where $f(n)$ is any linear recurrence. Again, our goal is to construct a matrix X for which

$$X \cdot \begin{bmatrix} f(i) \\ f(i+1) \\ \vdots \\ f(i+k-1) \end{bmatrix} = \begin{bmatrix} f(i+1) \\ f(i+2) \\ \vdots \\ f(i+k) \end{bmatrix}.$$

Such a matrix is

$$X = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ c_k & c_{k-1} & c_{k-2} & \cdots & c_1 \end{bmatrix}.$$

In the first $k-1$ rows, each element is 0 except that one element is 1. These rows replace $f(i)$ with $f(i+1)$, $f(i+1)$ with $f(i+2)$, and so on. Then, the last row contains the coefficients of the recurrence to calculate the new value $f(i+k)$.

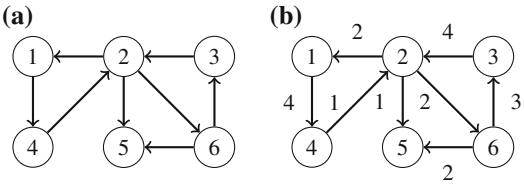


Fig. 11.16 Example graphs for matrix operations

Now, $f(n)$ can be calculated in $O(k^3 \log n)$ time using the formula

$$\begin{bmatrix} f(n) \\ f(n+1) \\ \vdots \\ f(n+k-1) \end{bmatrix} = X^n \cdot \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(k-1) \end{bmatrix}.$$

11.3.3 Graphs and Matrices

The powers of adjacency matrices of graphs have interesting properties. When M is an adjacency matrix of an unweighted graph, the matrix M^n gives for each node pair (a, b) the number of paths that begin at node a, end at node b, and contain exactly n edges. It is allowed that a node appears on a path several times.

As an example, consider the graph in Fig. 11.16a. The adjacency matrix of this graph is

$$M = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Then, the matrix

$$M^4 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 0 & 2 & 2 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

gives the number of paths that contain exactly 4 edges. For example, $M^4[2, 5] = 2$, because there are two paths of 4 edges from node 2 to node 5: $2 \to 1 \to 4 \to 2 \to 5$ and $2 \to 6 \to 3 \to 2 \to 5$.

Using a similar idea in a weighted graph, we can calculate for each node pair (a, b) the shortest length of a path that goes from a to b and contains exactly n

edges. To calculate this, we define matrix multiplication in a new way, so that we do not calculate numbers of paths but minimize lengths of paths.

As an example, consider the graph in Fig. 11.16b. Let us construct an adjacency matrix where ∞ means that an edge does not exist, and other values correspond to edge weights. The matrix is

$$M = \begin{bmatrix} \infty & \infty & \infty & 4 & \infty & \infty \\ 2 & \infty & \infty & \infty & 1 & 2 \\ \infty & 4 & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & \infty & 2 & \infty \end{bmatrix}.$$

Instead of the formula

$$AB[i, j] = \sum_{k=1}^{n} (A[i, k] \cdot B[k, j])$$

we now use the formula

$$AB[i, j] = \min_{k=1}^{n} (A[i, k] + B[k, j])$$

for matrix multiplication, so we calculate minima instead of sums, and sums of elements instead of products. After this modification, matrix powers minimize path lengths in the graph. For example, as

$$M^4 = \begin{bmatrix} \infty & \infty & 10 & 11 & 9 & \infty \\ 9 & \infty & \infty & \infty & 8 & 9 \\ \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & 8 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 12 & 13 & 11 & \infty \end{bmatrix},$$

we can conclude that the minimum length of a path of 4 edges from node 2 to node 5 is 8. Such a path is $2 \to 1 \to 4 \to 2 \to 5$.

11.3.4 Gaussian Elimination

Gaussian elimination is a systematic way to solve a group of linear equations. The idea is to represent the equations as a matrix and then apply a sequence of simple matrix row operations that both preserve the information of the equations and determine a value for each variable.

Suppose that we are given a group of n linear equations, each of which contains n variables:

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= b_2 \\ \cdots & \\ a_{n,1}x_1 + a_{n,2}x_2 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

We represent the equations as a matrix as follows:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{bmatrix}$$

To solve the equations, we want to transform the matrix to

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & c_1 \\ 0 & 1 & \cdots & 0 & c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & c_n \end{bmatrix},$$

which tells us that the solution is $x_1 = c_1, x_2 = c_2, \ldots, x_n = c_n$. To do this, we use three types of matrix row operations:

1. Swap the values of two rows.
2. Multiply each value in a row by a nonnegative constant.
3. Add a row, multiplied by a constant, to another row.

Each above operation preserves the information of the equations, which guarantees that the final solution agrees with the original equations. We can systematically process each matrix column so that the resulting algorithm works in $O(n^3)$ time.

As an example, consider the following group of equations:

$$\begin{aligned} 2x_1 + 4x_2 + x_3 &= 16 \\ x_1 + 2x_2 + 5x_3 &= 17 \\ 3x_1 + x_2 + x_3 &= 8 \end{aligned}$$

In this case the matrix is as follows:

$$\begin{bmatrix} 2 & 4 & 1 & 16 \\ 1 & 2 & 5 & 17 \\ 3 & 1 & 1 & 8 \end{bmatrix}$$

We process the matrix column by column. At each step, we make sure that the current column has a one in the correct position and all other values are zeros. To process the first column, we first multiply the first row by $\frac{1}{2}$:

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 1 & 2 & 5 & 17 \\ 3 & 1 & 1 & 8 \end{bmatrix}$$

Then we add the first row to the second row (multiplied by -1) and the first row to the third row (multiplied by -3):

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 0 & 0 & \frac{9}{2} & 9 \\ 0 & -5 & -\frac{1}{2} & -16 \end{bmatrix}$$

After this, we process the second column. Since the second value in the second row is zero, we first swap the second and third row:

$$\begin{bmatrix} 1 & 2 & \frac{1}{2} & 8 \\ 0 & -5 & -\frac{1}{2} & -16 \\ 0 & 0 & \frac{9}{2} & 9 \end{bmatrix}$$

Then we multiply the second row by $-\frac{1}{5}$ and add it to the first row (multiplied by -2):

$$\begin{bmatrix} 1 & 0 & \frac{3}{10} & \frac{8}{5} \\ 0 & 1 & \frac{1}{10} & \frac{16}{5} \\ 0 & 0 & \frac{9}{2} & 9 \end{bmatrix}$$

Finally, we process the third column by first multiplying it by $\frac{2}{9}$ and then adding it to the first row (multiplied by $-\frac{3}{10}$) and to the second row (multiplied by $-\frac{1}{10}$):

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 2 \end{bmatrix}$$

Now the last column of the matrix tells us that the solution to the original group of equations is $x_1 = 1, x_2 = 3, x_3 = 2$.

Note that Gaussian elimination only works if the group of equations has a *unique* solution. For example, the group

$$\begin{aligned} x_1 + \ x_2 &= 2 \\ 2x_1 + 2x_2 &= 4 \end{aligned}$$

has an infinite number of solutions, because both the equations contain the same information. On the other hand, the group

$$x_1 + x_2 = 5$$
$$x_1 + x_2 = 7$$

cannot be solved, because the equations are contradictory. If there is no unique solution, we will notice this during the algorithm, because at some point we will not be able to successfully process a column.

11.4 Probability

A *probability* is a real number between 0 and 1 that indicates how probable an event is. If an event is certain to happen, its probability is 1, and if an event is impossible, its probability is 0. The probability of an event is denoted $P(\cdots)$ where the three dots describe the event. For example, when throwing a dice, there are six possible outcomes $1, 2, \ldots, 6$, and $P(\text{“the outcome is even”}) = 1/2$.

To calculate the probability of an event, we can either use combinatorics or simulate the process that generates the event. As an example, consider an experiment where we draw the three top cards from a shuffled deck of cards.[2] What is the probability that each card has the same value (e.g., ♠8, ♣8, and ♢8)?

One way to calculate the probability is to use the formula

$$\frac{\text{number of desired outcomes}}{\text{total number of outcomes}}.$$

In our example, the desired outcomes are those in which the value of each card is the same. There are $13\binom{4}{3}$ such outcomes, because there are 13 possibilities for the value of the cards and $\binom{4}{3}$ ways to choose 3 suits from 4 possible suits. Then, there are a total of $\binom{52}{3}$ outcomes, because we choose 3 cards from 52 cards. Thus, the probability of the event is

$$\frac{13\binom{4}{3}}{\binom{52}{3}} = \frac{1}{425}.$$

Another way to calculate the probability is to simulate the process that generates the event. In our example, we draw three cards, so the process consists of three steps. We require that each step of the process is successful.

Drawing the first card certainly succeeds, because any card is fine. The second step succeeds with probability $3/51$, because there are 51 cards left and 3 of them

[2] A deck of cards consists of 52 cards. Each card has a suit (spade ♠, diamond ♢, club ♣, or heart ♡) and a value (an integer between 1 and 13).

have the same value as the first card. In a similar way, the third step succeeds with probability $2/50$. Thus, the probability that the entire process succeeds is

$$1 \cdot \frac{3}{51} \cdot \frac{2}{50} = \frac{1}{425}.$$

11.4.1 Working with Events

A convenient way to represent events is to use sets. For example, the possible outcomes when throwing a dice are $\{1, 2, 3, 4, 5, 6\}$, and any subset of this set is an event. The event "the outcome is even" corresponds to the set $\{2, 4, 6\}$.

Each outcome x is assigned a probability $p(x)$, and the probability $P(X)$ of an event X can be calculated using the formula

$$P(X) = \sum_{x \in X} p(x).$$

For example, when throwing a dice, $p(x) = 1/6$ for each outcome x, so the probability of the event "the outcome is even" is

$$p(2) + p(4) + p(6) = 1/2.$$

Since the events are represented as sets, we can manipulate them using standard set operations:

- The *complement* $\bar{A}$ means "A does not happen." For example, when throwing a dice, the complement of $A = \{2, 4, 6\}$ is $\bar{A} = \{1, 3, 5\}$.
- The *union* $A \cup B$ means "A or B happen." For example, the union of $A = \{2, 5\}$ and $B = \{4, 5, 6\}$ is $A \cup B = \{2, 4, 5, 6\}$.
- The *intersection* $A \cap B$ means "A and B happen." For example, the intersection of $A = \{2, 5\}$ and $B = \{4, 5, 6\}$ is $A \cap B = \{5\}$.

Complement The probability of $\bar{A}$ is calculated using the formula

$$P(\bar{A}) = 1 - P(A).$$

Sometimes, we can solve a problem easily using complements by solving the opposite problem. For example, the probability of getting at least one six when throwing a dice ten times is

$$1 - (5/6)^{10}.$$

Here $5/6$ is the probability that the outcome of a single throw is not six, and $(5/6)^{10}$ is the probability that none of the ten throws is a six. The complement of this is the answer to the problem.

Union The probability of $A \cup B$ is calculated using the formula

$$P(A \cup B) = P(A) + P(B) - P(A \cap B).$$

For example, consider the events $A =$ "the outcome is even" and $B =$ "the outcome is less than 4" when throwing a dice. In this case, the event $A \cup B$ means "the outcome is even or less than 4," and its probability is

$$P(A \cup B) = P(A) + P(B) - P(A \cap B) = 1/2 + 1/2 - 1/6 = 5/6.$$

If the events A and B are *disjoint*, i.e., $A \cap B$ is empty, the probability of the event $A \cup B$ is simply

$$P(A \cup B) = P(A) + P(B).$$

Intersection The probability of $A \cap B$ can be calculated using the formula

$$P(A \cap B) = P(A)P(B|A),$$

where $P(B|A)$ is the *conditional probability* that B happens assuming that we know that A happens. For example, using the events of our previous example, $P(B|A) = 1/3$, because we know that the outcome belongs to the set $\{2, 4, 6\}$, and one of the outcomes is less than 4. Thus,

$$P(A \cap B) = P(A)P(B|A) = 1/2 \cdot 1/3 = 1/6.$$

Events A and B are *independent* if

$$P(A|B) = P(A) \quad \text{and} \quad P(B|A) = P(B),$$

which means that the fact that B happens does not change the probability of A, and vice versa. In this case, the probability of the intersection is

$$P(A \cap B) = P(A)P(B).$$

11.4.2 Random Variables

A *random variable* is a value that is generated by a random process. For example, when throwing two dice, a possible random variable is

$$X = \text{"the sum of the outcomes"}.$$

For example, if the outcomes are [4, 6] (meaning that we first throw a four and then a six), then the value of X is 10.

We denote by $P(X = x)$ the probability that the value of a random variable X is x. For example, when throwing two dice, $P(X = 10) = 3/36$, because the total

Fig. 11.17 Possible ways to place two balls in four boxes

number of outcomes is 36 and there are three possible ways to obtain the sum 10: [4, 6], [5, 5], and [6, 4].

Expected Values The *expected value* $E[X]$ indicates the average value of a random variable X. The expected value can be calculated as a sum

$$\sum_x P(X = x)x,$$

where x goes through all possible values of X.

For example, when throwing a dice, the expected outcome is

$$1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 4 + 1/6 \cdot 5 + 1/6 \cdot 6 = 7/2.$$

A useful property of expected values is *linearity*. It means that the sum $E[X_1 + X_2 + \cdots + X_n]$ always equals the sum $E[X_1] + E[X_2] + \cdots + E[X_n]$. This holds even if random variables depend on each other. For example, when throwing two dice, the expected sum of their values is

$$E[X_1 + X_2] = E[X_1] + E[X_2] = 7/2 + 7/2 = 7.$$

Let us now consider a problem where n balls are randomly placed in n boxes, and our task is to calculate the expected number of empty boxes. Each ball has an equal probability to be placed in any of the boxes.

For example, Fig. 11.17 shows the possibilities when $n = 2$. In this case, the expected number of empty boxes is

$$\frac{0+0+1+1}{4} = \frac{1}{2}.$$

Then, in the general case, the probability that a single box is empty is

$$\left(\frac{n-1}{n}\right)^n,$$

because no ball should be placed in it. Hence, using linearity, the expected number of empty boxes is

$$n \cdot \left(\frac{n-1}{n}\right)^n.$$

Distributions The *distribution* of a random variable X shows the probability of each value that X may have. The distribution consists of values $P(X = x)$. For example, when throwing two dice, the distribution for their sum is:

x	2	3	4	5	6	7	8	9	10	11	12
$P(X = x)$	1/36	2/36	3/36	4/36	5/36	6/36	5/36	4/36	3/36	2/36	1/36

In a *uniform distribution*, the random variable X has n possible values $a, a + 1, \ldots, b$ and the probability of each value is $1/n$. For example, when throwing a dice, $a = 1$, $b = 6$, and $P(X = x) = 1/6$ for each value x.

The expected value of X in a uniform distribution is

$$E[X] = \frac{a+b}{2}.$$

In a *binomial distribution*, n attempts are made and the probability that a single attempt succeeds is p. The random variable X counts the number of successful attempts, and the probability of a value x is

$$P(X = x) = p^x (1-p)^{n-x} \binom{n}{x},$$

where p^x and $(1-p)^{n-x}$ correspond to successful and unsuccessful attempts, and $\binom{n}{x}$ is the number of ways we can choose the order of the attempts.

For example, when throwing a dice ten times, the probability of throwing a six exactly three times is $(1/6)^3 (5/6)^7 \binom{10}{3}$.

The expected value of X in a binomial distribution is

$$E[X] = pn.$$

In a *geometric distribution*, the probability that an attempt succeeds is p, and we continue until the first success happens. The random variable X counts the number of attempts needed, and the probability of a value x is

$$P(X = x) = (1-p)^{x-1} p,$$

where $(1-p)^{x-1}$ corresponds to the unsuccessful attempts and p corresponds to the first successful attempt.

For example, if we throw a dice until we get a six, the probability that the number of throws is exactly 4 is $(5/6)^3 1/6$.

The expected value of X in a geometric distribution is

$$E[X] = \frac{1}{p}.$$

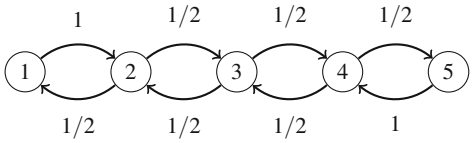


Fig. 11.18 A Markov chain for a building that consists of five floors

11.4.3 Markov Chains

A *Markov chain* is a random process that consists of states and transitions between them. For each state, we know the probabilities of moving to other states. A Markov chain can be represented as a graph whose nodes correspond to the states and edges describe the transitions.

As an example, consider a problem where we are in floor 1 in an n floor building. At each step, we randomly walk either one floor up or one floor down, except that we always walk one floor up from floor 1 and one floor down from floor n. What is the probability of being in floor m after k steps?

In this problem, each floor of the building corresponds to a state in a Markov chain. For example, Fig. 11.18 shows the chain when $n = 5$.

The probability distribution of a Markov chain is a vector $[p_1, p_2, \ldots, p_n]$, where p_k is the probability that the current state is k. The formula $p_1 + p_2 + \cdots + p_n = 1$ always holds.

In the above scenario, the initial distribution is $[1, 0, 0, 0, 0]$, because we always begin in floor 1. The next distribution is $[0, 1, 0, 0, 0]$, because we can only move from floor 1 to floor 2. After this, we can either move one floor up or one floor down, so the next distribution is $[1/2, 0, 1/2, 0, 0]$, and so on.

An efficient way to simulate the walk in a Markov chain is to use dynamic programming. The idea is to maintain the probability distribution, and at each step go through all possibilities how we can move. Using this method, we can simulate a walk of m steps in $O(n^2 m)$ time.

The transitions of a Markov chain can also be represented as a matrix that updates the probability distribution. In the above scenario, the matrix is

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix}.$$

When we multiply a probability distribution by this matrix, we get the new distribution after moving one step. For example, we can move from the distribution

[1, 0, 0, 0, 0] to the distribution [0, 1, 0, 0, 0] as follows:

$$\begin{bmatrix} 0 & 1/2 & 0 & 0 & 0 \\ 1 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1 \\ 0 & 0 & 0 & 1/2 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

By calculating matrix powers efficiently, we can calculate the distribution after m steps in $O(n^3 \log m)$ time.

11.4.4 Randomized Algorithms

Sometimes we can use randomness for solving a problem, even if the problem is not related to probabilities. A *randomized algorithm* is an algorithm that is based on randomness. There are two popular types of randomized algorithms:

- A *Monte Carlo algorithm* is an algorithm that may sometimes give a wrong answer. For such an algorithm to be useful, the probability of a wrong answer should be small.
- A *Las Vegas algorithm* is an algorithm that always gives the correct answer, but its running time varies randomly. The goal is to design an algorithm that is efficient with high probability.

Next we will go through three example problems that can be solved using such algorithms.

Order Statistics The *kth order statistic* of an array is the element at position k after sorting the array in increasing order. It is easy to calculate any order statistic in $O(n \log n)$ time by first sorting the array, but is it really needed to sort the entire array just to find one element?

It turns out that we can find order statistics using a Las Vegas algorithm, whose expected running time is $O(n)$. The algorithm chooses a random element x from the array and moves elements smaller than x to the left part of the array, and all other elements to the right part of the array. This takes $O(n)$ time when there are n elements.

Assume that the left part contains a elements and the right part contains b elements. If $a = k$, element x is the kth order statistic. Otherwise, if $a > k$, we recursively find the kth order statistic for the left part, and if $a < k$, we recursively find the rth order statistic for the right part where $r = k - a - 1$. The search continues in a similar way, until the desired element has been found.

When each element x is randomly chosen, the size of the array about halves at each step, so the time complexity for finding the kth order statistic is about

$$n + n/2 + n/4 + n/8 + \cdots = O(n).$$

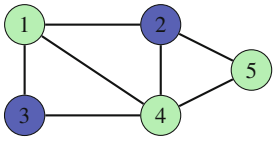


Fig. 11.19 A valid coloring of a graph

Note that the worst case of the algorithm requires $O(n^2)$ time, because it is possible that x is always chosen in such a way that it is one of the smallest or largest elements in the array and $O(n)$ steps are needed. However, the probability of this is so small that we may assume that this never happens in practice.

Verifying Matrix Multiplication Given matrices A, B, and C, each of size $n \times n$, our next problem is to *verify* if $AB = C$ holds. Of course, we can solve the problem by just calculating the product AB in $O(n^3)$ time, but one could hope that verifying the answer would be easier than to calculate it from scratch.

It turns out that we can solve the problem using a Monte Carlo algorithm whose time complexity is only $O(n^2)$. The idea is simple: we choose a random vector X of n elements and calculate the matrices ABX and CX. If $ABX = CX$, we report that $AB = C$, and otherwise we report that $AB \neq C$.

The time complexity of the algorithm is $O(n^2)$, because we can calculate the matrices ABX and CX in $O(n^2)$ time. We can calculate the matrix ABX efficiently by using the representation $A(BX)$, so only two multiplications of $n \times n$ and $n \times 1$ size matrices are needed.

The drawback of the algorithm is that there is a small chance that the algorithm makes a mistake when it reports that $AB = C$. For example,

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \neq \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix},$$

but

$$\begin{bmatrix} 6 & 8 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 8 & 7 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \end{bmatrix}.$$

However, in practice, the probability that the algorithm makes a mistake is small, and we can decrease the probability by verifying the result using multiple random vectors X before reporting that $AB = C$.

Graph Coloring Given a graph that contains n nodes and m edges, our final problem is to find a way to color the nodes using two colors so that for at least $m/2$ edges, the endpoints have different colors. For example, Fig. 11.19 shows a valid coloring of a graph. In this case the graph contains seven edges, and the endpoints of five of them have different colors in the coloring.

The problem can be solved using a Las Vegas algorithm that generates random colorings until a valid coloring has been found. In a random coloring, the color of each node is independently chosen so that the probability of both colors is $1/2$. Hence, the expected number of edges whose endpoints have different colors is $m/2$.

Since it is expected that a random coloring is valid, we will quickly find a valid coloring in practice.

11.5 Game Theory

In this section, we focus on two-player games where the players move alternately and have the same set of moves available, and there are no random elements. Our goal is to find a strategy that we can follow to win the game no matter what the opponent does, if such a strategy exists.

It turns out that there is a general strategy for such games, and we can analyze the games using *nim theory*. First, we will analyze simple games where players remove sticks from heaps, and after this, we will generalize the strategy used in those games to other games.

11.5.1 Game States

Let us consider a game that starts with a heap of n sticks. Two players move alternately, and on each move, the player has to remove 1, 2, or 3 sticks from the heap. Finally, the player who removes the last stick wins the game.

For example, if $n = 10$, the game may proceed as follows:

- Player A removes 2 sticks (8 sticks left).
- Player B removes 3 sticks (5 sticks left).
- Player A removes 1 stick (4 sticks left).
- Player B removes 2 sticks (2 sticks left).
- Player A removes 2 sticks and wins.

This game consists of states $0, 1, 2, \ldots, n$, where the number of the state corresponds to the number of sticks left.

A *winning state* is a state where the player will win the game if they play optimally, and a *losing state* is a state where the player will lose the game if the opponent plays optimally. It turns out that we can classify all states of a game so that each state is either a winning state or a losing state.

In the above game, state 0 is clearly a losing state, because the player cannot make any moves. States 1, 2, and 3 are winning states, because the player can remove 1, 2, or 3 sticks and win the game. State 4, in turn, is a losing state, because any move leads to a state that is a winning state for the opponent.

More generally, if there is a move that leads from the current state to a losing state, it is a winning state, and otherwise it is a losing state. Using this observation, we can classify all states of a game starting with losing states where there are no possible moves. Figure 11.20 shows the classification of states $0 \ldots 15$ (W denotes a winning state and L denotes a losing state).

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*	*L*	*W*	*W*	*W*

Fig. 11.20 Classification of states 0 . . . 15 in the stick game

Fig. 11.21 State graph of the divisibility game

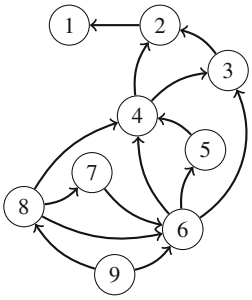


Fig. 11.22 Classification of states 1 . . . 9 in the divisibility game

1	2	3	4	5	6	7	8	9
L	*W*	*L*	*W*	*L*	*W*	*L*	*W*	*L*

It is easy to analyze this game: a state k is a losing state if k is divisible by 4, and otherwise it is a winning state. An optimal way to play the game is to always choose a move after which the number of sticks in the heap is divisible by 4. Finally, there are no sticks left and the opponent has lost. Of course, this strategy requires that the number of sticks is *not* divisible by 4 when it is our move. If it is, there is nothing we can do, and the opponent will win the game if they play optimally.

Let us then consider another stick game, where in each state k, it is allowed to remove any number x of sticks such that x is smaller than k and divides k. For example, in state 8 we may remove 1, 2, or 4 sticks, but in state 7 the only allowed move is to remove 1 stick. Figure 11.21 shows the states 1 . . . 9 of the game as a *state graph*, whose nodes are the states and edges are the moves between them:

The final state in this game is always state 1, which is a losing state, because there are no valid moves. Figure 11.22 shows the classification of states 1 . . . 9. It turns out that in this game, all even-numbered states are winning states, and all odd-numbered states are losing states.

11.5.2 Nim Game

The *nim game* is a simple game that has an important role in game theory, because many other games can be played using the same strategy. First, we focus on nim, and after this, we generalize the strategy to other games.

There are n heaps in nim, and each heap contains some number of sticks. The players move alternately, and on each turn, the player chooses a heap that still contains sticks and removes any number of sticks from it. The winner is the player who removes the last stick.

The states in nim are of the form $[x_1, x_2, \ldots, x_n]$, where x_i denotes the number of sticks in heap i. For example, $[10, 12, 5]$ is a state where there are three heaps with 10, 12, and 5 sticks. The state $[0, 0, \ldots, 0]$ is a losing state, because it is not possible to remove any sticks, and this is always the final state.

Analysis It turns out that we can easily classify any nim state by calculating the *nim sum* $s = x_1 \oplus x_2 \oplus \cdots \oplus x_n$, where $\oplus$ denotes the xor operation. The states whose nim sum is 0 are losing states, and all other states are winning states. For example, the nim sum of $[10, 12, 5]$ is $10 \oplus 12 \oplus 5 = 3$, so the state is a winning state.

But how is the nim sum related to the nim game? We can explain this by looking at how the nim sum changes when the nim state changes.

Losing states: The final state $[0, 0, \ldots, 0]$ is a losing state, and its nim sum is 0, as expected. In other losing states, any move leads to a winning state, because when a single value x_i changes, the nim sum also changes, so the nim sum is different from 0 after the move.

Winning states: We can move to a losing state if there is any heap i for which $x_i \oplus s < x_i$. In this case, we can remove sticks from heap i so that it will contain $x_i \oplus s$ sticks, which will lead to a losing state. There is always such a heap, where x_i has a one bit at the position of the leftmost one bit of s.

Example As an example, consider the state $[10, 12, 5]$. This state is a winning state, because its nim sum is 3. Thus, there has to be a move which leads to a losing state. Next we will find out such a move.

The nim sum of the state is as follows:

10	1010
12	1100
5	0101
3	0011

In this case, the heap with 10 sticks is the only heap that has a one bit at the position of the leftmost one bit of the nim sum:

10	10<u>1</u>0
12	1100
5	0101
3	00<u>1</u>1

The new size of the heap has to be $10 \oplus 3 = 9$, so we will remove just one stick. After this, the state will be $[9, 12, 5]$, which is a losing state:

9	1001
12	1100
5	0101
0	0000

Misère Game In a *misère* nim game, the goal of the game is opposite, so the player who removes the last stick loses the game. It turns out that the misère nim game can be optimally played almost like the standard nim game.

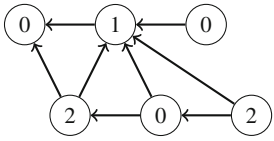


Fig. 11.23 Grundy numbers of game states

The idea is to first play the misère game like the standard game, but change the strategy at the end of the game. The new strategy will be introduced in a situation where each heap would contain at most one stick after the next move. In the standard game, we should choose a move after which there is an even number of heaps with one stick. However, in the misère game, we choose a move so that there is an odd number of heaps with one stick.

This strategy works because a state where the strategy changes always appears in the game, and this state is a winning state, because it contains exactly one heap that has more than one stick so the nim sum is not 0.

11.5.3 Sprague–Grundy Theorem

The *Sprague–Grundy theorem* generalizes the strategy used in nim to all games that fulfill the following requirements:

- There are two players who move alternately.
- The game consists of states, and the possible moves in a state do not depend on whose turn it is.
- The game ends when a player cannot make a move.
- The game surely ends sooner or later.
- The players have complete information about the states and allowed moves, and there is no randomness in the game.

Grundy Numbers The idea is to calculate for each game state a *Grundy number* that corresponds to the number of sticks in a nim heap. When we know the Grundy numbers of all states, we can play the game like the nim game.

The Grundy number of a game state is calculated using the formula

$$\text{mex}(\{g_1, g_2, \ldots, g_n\}),$$

where $g_1, g_2, \ldots, g_n$ are the Grundy numbers of the states to which we can move from the state, and the mex function gives the smallest nonnegative number that is not in the set. For example, $\text{mex}(\{0, 1, 3\}) = 2$. If a state has no possible moves, its Grundy number is 0, because $\text{mex}(\emptyset) = 0$.

For example, Fig. 11.23 shows a state graph of a game where each state is assigned its Grundy number. The Grundy number of a losing state is 0, and the Grundy number of a winning state is a positive number.

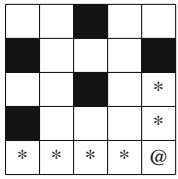


Fig. 11.24 Possible moves on the first turn

0	1	■	0	1
■	0	1	2	■
0	2	■	1	0
■	3	0	4	1
0	4	1	3	2

Fig. 11.25 Grundy numbers of game states

Consider a state whose Grundy number is x. We can think that it corresponds to a nim heap that has x sticks. In particular, if $x > 0$, we can move to states whose Grundy numbers are $0, 1, \ldots, x-1$, which simulates removing sticks from a nim heap. There is one difference, though it may be possible to move to a state whose Grundy number is larger than x and "add" sticks to a heap. However, the opponent can always cancel any such move, so this does not change the strategy.

As an example, consider a game where the players move a figure in a maze. Each square of the maze is either floor or wall. On each turn, the player has to move the figure some number of steps left or up. The winner of the game is the player who makes the last move. Figure 11.24 shows a possible initial configuration of the game, where @ denotes the figure and * denotes a square where it can move. The states of the game are all floor squares of the maze. Figure 11.25 shows the Grundy numbers of the states in this configuration.

According to the Sprague–Grundy theorem, each state of the maze game corresponds to a heap in the nim game. For example, the Grundy number of the lower-right square is 2, so it is a winning state. We can reach a losing state and win the game by moving either four steps left or two steps up.

Subgames Assume that our game consists of subgames, and on each turn, the player first chooses a subgame and then a move in the subgame. The game ends when it is not possible to make any move in any subgame. In this case, the Grundy number of a game equals the nim sum of the Grundy numbers of the subgames. The game can then be played like a nim game by calculating all Grundy numbers for subgames and then their nim sum.

As an example, consider a game that consists of three mazes. On each turn, the player chooses one of the mazes and then moves the figure in the maze. Figure 11.26 shows an initial configuration of the game, and Fig. 11.27 shows the corresponding Grundy numbers. In this configuration, the nim sum of the Grundy numbers is $2 \oplus 3 \oplus 3 = 2$, so the first player can win the game. One optimal move is to move two steps up in the first maze, which produces the nim sum $0 \oplus 3 \oplus 3 = 0$.

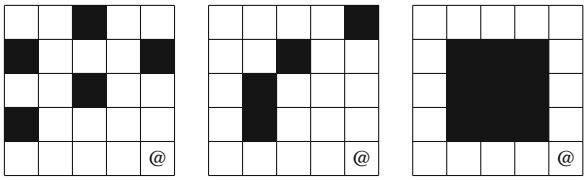


Fig. 11.26 A game that consists of three subgames

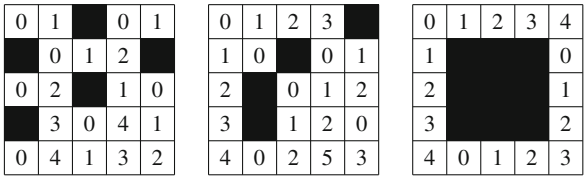


Fig. 11.27 Grundy numbers in subgames

Grundy's Game Sometimes a move in a game divides the game into subgames that are independent of each other. In this case, the Grundy number of a game state is

$$\text{mex}(\{g_1, g_2, \ldots, g_n\}),$$

where there are n possible moves and

$$g_k = a_{k,1} \oplus a_{k,2} \oplus \ldots \oplus a_{k,m},$$

meaning that move k divides the game into m subgames whose Grundy numbers are $a_{k,1}, a_{k,2}, \ldots, a_{k,m}$.

An example of such a game is *Grundy's game*. Initially, there is a single heap that has n sticks. On each turn, the player chooses a heap and divides it into two nonempty heaps such that the heaps are of different size. The player who makes the last move wins the game.

Let $g(n)$ denote the Grundy number of a heap of size n. The Grundy number can be calculated by going through all ways to divide the heap into two heaps. For example, when $n = 8$, the possibilities are $1 + 7$, $2 + 6$, and $3 + 5$, so

$$g(8) = \text{mex}(\{g(1) \oplus g(7), g(2) \oplus g(6), g(3) \oplus g(5)\}).$$

In this game, the value of $g(n)$ is based on the values of $g(1), \ldots, g(n-1)$. The base cases are $g(1) = g(2) = 0$, because it is not possible to divide the heaps of 1 and 2 sticks into smaller heaps. The first Grundy numbers are:

$$\begin{aligned} g(1) &= 0 \\ g(2) &= 0 \\ g(3) &= 1 \\ g(4) &= 0 \end{aligned}$$

$$g(5) = 2$$
$$g(6) = 1$$
$$g(7) = 0$$
$$g(8) = 2$$

The Grundy number for $n = 8$ is 2, so it is possible to win the game. The winning move is to create heaps $1 + 7$, because $g(1) \oplus g(7) = 0$.