Chapter 2

第 2 章

Recurrences

递归

2.1 Karatsuba's Algorithm: One Example of Recursive Algorithms

2.1 卡拉苏巴算法:递归算法的一个例子

We learned multiplication in primary school. When we do 12×34 , we essentially calculate $1 \times 3 \times 100 + (2 \times 3 + 1 \times 4) \times 10 + 2 \times 4$. In general, we use the formula

我们在小学时学过乘法。当我们做 12×34 时，本质上是在计算 $1 \times 3 \times 100 + (2 \times 3 + 1 \times 4) \times 10 + 2 \times 4$ 。一般来说，我们使用公式

$$\left(\sum_{i=0}^{n-1} x_i \cdot b^i\right) \cdot \left(\sum_{i=0}^{n-1} y_i \cdot b^i\right) = \sum_{i=0}^{n-1} \left(\sum_{j=0}^{i} x_j \, y_{i-j}\right) \cdot b^i + \sum_{i=n}^{2(n-1)} \left(\sum_{j=i-(n-1)}^{n-1} x_j \, y_{i-j}\right) \cdot b^i,$$

where $b = 10$ for decimal calculations, while b is often equal to some power of 2 in computers. This naive algorithm needs to do n^2 "basic" multiplications (i.e., multiplications over $[0, b-1]$) in order to multiply two number from $[0, b^n - 1]$. Naturally, many people suspected that the above "textbook multiplication" algorithm might be the best we could do, because there seemed to be no obvious way to do multiplications faster.

其中 $b = 10$ 用于十进制计算，而 b 在计算机中通常等于 2 的某个幂。这种朴素算法需要进行 n^2 次“基本”乘法(即对 $[0, b-1]$ 进行的乘法)才能将两个 $[0, b^n - 1]$ 中的数相乘。自然而然地，许多人怀疑上述“教科书式乘法”算法可能是我们能做到的最好的算法了，因为似乎没有明显的方法能更快地进行乘法运算。

Surprisingly, Karatsubal proposed a counter-intuitive algorithm that easily beats the "textbook multiplication" algorithm. The idea is quite simple: when you do 12×34 , you should not waste your time doing two "basic" multiplications for $(2 \times 3 + 1 \times 4)$. Instead, you should just calculate $(1+2) \times (3+4) - 1 \times 3 - 2 \times 4$, which is identical to $(2 \times 3 + 1 \times 4)$. Noticing that you have to calculate 1×3 and 2×4 for other parts of the expression anyway, for this specific part you just need to do one "basic" multiplication! Therefore, we have reduced the total amount of work.

令人惊讶的是，卡拉苏巴提出了一种违反直觉的算法，它轻松击败了“教科书式乘法”算法。其思路相当简单:当你做 12×34 时，不应该浪费时间对 $(2 \times 3 + 1 \times 4)$ 进行两次“基本”乘法。相反，你应该只计算 $(1+2) \times (3+4) - 1 \times 3 - 2 \times 4$ ，它与 $(2 \times 3 + 1 \times 4)$ 相同。注意到无论如何你都必须为表达

式的其他部分计算 1×3 和 2×4，对于这个特定部分你只需要进行一次“基本”乘法！因此，我们减少了总的工作量。

[1]

Below we present a straightforward recursive algorithm based on Karatsuba's idea.

下面我们给出一个基于卡拉苏巴想法的直接递归算法。

Algorithm: Karatsuba(x, y)

算法:卡拉苏巴(x，y)

Input: x and y, two positive integers.

输入: x 和 y，两个正整数。

Output: the product xy.

输出:乘积 xy。

if $(x < 2^b)$ or $(y < 2^b)$

如果 $(x < 2^b)$ 或 $(y < 2^b)$

return xy;

返回 xy；

split x in the middle and get (h_x, l_x);

将 x 从中间分开得到 (h_x, l_x)；

split y in the middle and get (h_y, l_y);

将 y 从中间分开得到 (h_y, l_y)；

$$r_0 = \text{Karatsuba}(l_x, l_y);$$

$$r_1 = \text{Karatsuba}\left((l_x + h_x), (l_y + h_y)\right);$$

$$r_2 = \text{Karatsuba}(h_x, h_y);$$

return the number $r_2 \cdot 2^{2b} + (r_1 - r_2 - r_0) \cdot 2^b + r_0$.

返回数字 $r_2 \cdot 2^{2b} + (r_1 - r_2 - r_0) \cdot 2^b + r_0$。

---

[1] [1] Anatoly Karatsuba (1937-2008) was a Russian mathematician. When he was a student, he attended Kol-mogrov's seminar at the Moscow State University, and proposed an idea for accelerating the multiplication algorithm. Kolmogrov wrote his idea into a paper, with Karatsuba as the sole author, and got it published. As a professional mathematician, Karatsuba mainly worked on number theory.

[1] 阿纳托利·卡拉苏巴(1937 - 2008)是一位俄罗斯数学家。他还是学生时，参加了莫斯科国立大学柯尔莫哥洛夫的研讨会，并提出了一种加速乘法算法的想法。柯尔莫哥洛夫将他的想法写成一篇论文，卡拉苏巴为唯一作者并发表了该论文。作为一名专业数学家，卡拉苏巴主要从事数论研究。

Example 2.1.1 Suppose that we are running Karatsuba's algorithm to compute 0101×0010 . (All numbers in this example, like 0101 and 0010, are binary.) The algorithm recursively calls itself three times:

示例 2.1.1 假设我们正在运行 Karatsuba 算法来计算 0101×0010 。(此示例中的所有数字，如 0101 和 0010，均为二进制数。)该算法会递归调用自身三次:

- To compute 01×10 , it recursively calls itself three times:
- 为了计算 01×10 ，它会递归调用自身三次:

$$\text{- It computes } 1 \times 0 = 0 \text{ .}$$

$$\text{- It computes } (1 + 0) \times (0 + 1) = 1 \text{ .}$$

$$\text{- It computes } 0 \times 1 = 0 \text{ .}$$

Therefore, it calculates $1 - 0 - 0 = 1$ and returns 010 for 01×10 .

因此，它计算出 $1 - 0 - 0 = 1$ 并为 01×10 返回 010 。

- To compute $(01 + 01) \times (10 + 00) = 10 \times 10$, it recursively calls itself three times:
- 为了计算 $(01 + 01) \times (10 + 00) = 10 \times 10$ ，它会递归调用自身三次:
- It computes $0 \times 0 = 0$.
- 它计算 $0 \times 0 = 0$ 。
- It computes $(0 + 1) \times (0 + 1) = 1$.
- 它计算 $(0 + 1) \times (0 + 1) = 1$ 。
- It computes $1 \times 1 = 1$.
- 它计算 $1 \times 1 = 1$ 。

Therefore, it calculates $1 - 1 - 0 = 0$ and returns 100 for $(01 + 01) \times (10 + 00)$.

因此，它计算出 $1 - 1 - 0 = 0$ 并为 $(01 + 01) \times (10 + 00)$ 返回 100 。

- To compute 01×00 , it recursively calls itself three times:
- 为了计算 01×00 ，它会递归调用自身三次:
- It computes $1 \times 0 = 0$.
- 它计算 $1 \times 0 = 0$ 。
- It computes $(1 + 0) \times (0 + 0) = 0$.
- 它计算 $(1 + 0) \times (0 + 0) = 0$ 。
- It computes $0 \times 0 = 0$.
- 它计算 $0 \times 0 = 0$ 。

Therefore, it calculates $0-0-0=0$ and returns 000 for 01×00 .

因此，它计算出 $0-0-0=0$ 并为 01×00 返回 000 。

Finally, it calculates $100-010-000=010$ and returns 001010 for 0101×0010 .

最后，它计算出 $100-010-000=010$ 并为 0101×0010 返回 001010 。

Now, assume that each "basic" multiplication is on a couple of b -bit integers, and that Karat-suba's algorithm needs $T(n)$ "basic" multiplications on a couple of n -bit numbers. Assuming the multiplication of l_x+h_x and l_y+h_y requires no more "basic" multiplications than the multiplication of l_x and l_y (or that of h_x and h_y), we get the following equation:

现在，假设每次“基本”乘法都是对一对 b 位整数进行的，并且 Karatsuba 算法对一对 n 位数字需要 $T(n)$ 次“基本”乘法。假设 l_x+h_x 和 l_y+h_y 的乘法所需的“基本”乘法次数不超过 l_x 和 l_y 的乘法(或 h_x 和 h_y 的乘法)，我们得到以下等式:

$$T(n)=3T\left(\frac{n}{2}\right). \qquad (2.1)$$

It is trivial to derive from the above equation that $T(n)=3^{\log_2\frac{n}{b}}T(b)=\left(\frac{n}{b}\right)^{\log_2 3}\approx n^{\log_2 3}$, assuming b is small and n is big. In contrast, the naive algorithm needs n^2 "basic" multiplications. For large n , the difference can be very significant.

假设 b 较小且 n 较大，从上述等式很容易推导出 $T(n)=3^{\log_2\frac{n}{b}}T(b)=\left(\frac{n}{b}\right)^{\log_2 3}\approx n^{\log_2 3}$ 。相比之下，朴素算法需要 n^2 次“基本”乘法。对于较大的 n ，差异可能非常显著。

Karatsuba's algorithm is the beginning of a search for formulas that help reduce the computational cost of multiplication. Many more formulas have been found since then. For example, in 2009, Fan et al. developed a number of Karatsuba-like formulas, including some asymmetric ones. Interested readers can refer to [21].

卡拉苏巴算法开启了寻找有助于降低乘法计算成本公式的征程。自那时起，人们又发现了更多公式。例如，2009 年，范等人开发了许多类似卡拉苏巴的公式，包括一些非对称公式。感兴趣的读者可参考文献[21]。

A more popular algorithm for multiplication (of polynomials in general, and of integers in particular) is Fast Fourier Transform (FFT), which has an even lower computational cost. Just like Karatsuba's algorithm, FFT is also a divide-and-conquer algorithm. A nice introduction to FFT can be found in standard textbooks of algorithms, like [12].

一种更流行的乘法算法(一般用于多项式，特别是整数)是快速傅里叶变换(FFT)，其计算成本更低。与卡拉苏巴算法一样，FFT 也是一种分治算法。在标准算法教材中，如[12]，可以找到关于 FFT 的精彩介绍。

One might find it interesting to solve recurrence relations like Eq. (6.1). It turns out that there is no universal approach to solve all recurrence equations. Hereafter, we examine a few types of recurrences that we know how to solve.

求解像式(6.1)这样的递推关系可能会很有趣。事实证明，没有一种通用方法能解决所有递推方程。在此之后，我们将研究几种我们知道如何求解的递推类型。

2.2 Recurrences, Total and Particular Solutions

2.2 递推关系、通解和特解

We say a recurrence is linear if $T(n)$ is equal to a linear combination of $T(m)$s$(m < n)$. For a large class of linear recurrences, we happen to know the structure of their solutions, which is quite analogous to that of solutions to linear systems.

如果 a_n 等于 a_{n-1} 的线性组合，我们就说这个递推关系是线性的。对于一大类线性递推关系，我们恰好知道它们解的结构，这与线性方程组的解的结构非常相似。

2

Theorem 2.2.1 The set of solutions to the linear recurrence

定理 2.2.1 线性递推关系的解的集合

$$T(n) = \sum_{i=1}^{k} c_i\, T(a_i n + b_i) + d(n), (n \geq n_0) \qquad (2.2)$$

(where $a_i n + b_i < n$ for all i and all $n \geq n_0$) is

(其中对于所有 $n \geq k$ 和所有 $i = 0,1,\cdots,k-1$， $a_n = \sum_{i=0}^{k-1} c_i\, a_{n-i}$)是

$$\left\{ T_0(n) + T_1(n) \mid T_0(n) = \sum_{i=1}^{k} c_i\, T_0(a_i n + b_i) \right\},$$

where $T_1(n)$ is an arbitrary solution to Equation (2.2). Since $T_1(n)$ itself belongs to the set of solutions, we usually call it a particular solution and the set of all solutions the total solution.

其中 a_n^p 是方程(2.2)的任意一个解。由于 a_n^p 本身属于解的集合，我们通常称它为一个特解，而所有解的集合称为通解。

Proof: Consider

证明:考虑

$$T(n) = \sum_{i=1}^{k} c_i\, T(a_i n + b_i). \qquad (2.3)$$

---

[2] [2] Hereafter, we are sloppy, using notations like $T\left(\frac{n}{2}\right)$, despite that T is defined on positive integers but $\frac{n}{2}$ may not be an integer. In order to be rigorous, we would have to replace $\frac{n}{2}$ with $\left\lfloor \frac{n}{2} \right\rfloor$, and take care of a lot more details. To keep it simple, we use expressions like $T\left(\frac{n}{2}\right)$ as if they were legal.

在此之后，我们会比较随意，使用像 a_n 这样的符号，尽管 a_n 是在正整数上定义的，但 n 可能不是整数。为了严谨起见，我们必须用 n' 代替 n ，并处理更多细节。为了简单起见，我们使用像 a_n 这样的表达式，就好像它们是合法的一样。

Equation (2.3) has all terms being of degree 1, and is thus called a homogeneous equation. Suppose that $T_0(n)$ is a solution to Equation (2.3), and that $T_1(n)$ is a solution to Equation (2.2). Easy to get that

方程(2.3)的所有项都是一次的，因此被称为齐次方程。假设 a_n^h 是方程(2.3)的一个解，并且 a_n 是方程(2.2)的一个解。很容易得到

$$T_1(n) + T_0(n) = \sum_{i=1}^{k} c_i\, T_1(a_i n + b_i) + d(n) + \sum_{i=1}^{k} c_i\, T_0(a_i n + b_i)$$

$$= \sum_{i=1}^{k} c_i \left(T_1(a_i n + b_i) + T_0(a_i n + b_i)\right) + d(n).$$

Hence, $T_1(n) + T_0(n)$ must also be a solution to Equation (2.2).

因此， a_n^h 也一定是方程(2.2)的一个解。

Conversely, if $T_2(n)$ is a solution to Equation (2.2), we can easily verify that $T_2(n) - T_1(n)$ is a solution to Equation (2.3). Therefore, $T_2(n)$ can be written as the sum of $T_1(n)$, a particular solution, and a solution to the homogeneous equation.

反之，如果 a_n 是方程(2.2)的一个解，我们可以很容易地验证 $a_n - a_n^p$ 是方程(2.3)的一个解。因此， a_n 可以写成一个特解 a_n^p 和齐次方程的一个解的和。

Example 2.2.1 Suppose that we would like to solve the recurrence

例 2.2.1 假设我们要解递推关系

$$T(n) = 2T\left(\frac{n}{2}\right) - 1.$$

Clearly the corresponding homogeneous equation has solutions $T_0(n) = kn$ where k is an arbitrary constant. A particular solution is $T_1(n) = 1$. Hence, the total solution is $T(n) = kn + 1$.

显然，相应的齐次方程的解是 $a_n^h = c \cdot 2^n$ ，其中 c 是任意常数。一个特解是 $a_n^p = 3$ 。因此，通解是 $a_n = c \cdot 2^n + 3$ 。

Nevertheless, homogeneous equations are not always so easy to solve. In many cases, we do not know how to solve it and have to rely on the "guess-and-prove" strategy. Below we discuss one (unusual) type of homogeneous equations for which we do have a systematic approach.

然而，齐次方程并不总是那么容易求解。在很多情况下，我们不知道如何求解，不得不依靠“猜测并证明”的策略。下面我们讨论一种(不寻常的)齐次方程类型，对于这种方程我们确实有一个系统的方法。

Definition 2.2.1 Consider the homogeneous equation

定义 2.2.1 考虑齐次方程

$$T(n) = c_1 T(n-1) + c_2 T(n-2) + \cdots + c_k T(n-k), \qquad (2.4)$$

where k is a constant positive integer, $c_1, \ldots, c_k$ are all constants, and $c_1 c_k \neq 0$. We say Equation [2.4] is a linear homogeneous recurrence with constant coefficients of order k . Its characteristic equation is

其中 k 是一个正整数常量， $c_1, \ldots, c_k$ 均为常量，且 $c_1 c_k \neq 0$ 。我们称方程[2.4]是一个 k 阶的常系数线性齐次递推关系。它的特征方程是

$$x^k - c_1 x^{k-1} - c_2 x^{k-2} - \cdots - c_k = 0. \qquad (2.5)$$

If you have studied linear algebra, please compare the above definition with the characteristic equation of a real symmetric matrix. If you have studied differential equations, please compare the above definition with the characteristic equation of a linear differential equation with constant coefficients of order k. The comparison will help you understand the deep connection between Equations (2.4) and (2.5), which is reflected by the theorem below.

如果你学过线性代数，请将上述定义与实对称矩阵的特征方程作比较。如果你学过微分方程，请将上述定义与 k 阶常系数线性微分方程的特征方程作比较。这种比较将有助于你理解方程(2.4)和(2.5)之间的深层联系，这一点将在下面的定理中体现。

Theorem 2.2.2 If Equation 2.5) has k distinct roots $r_1, \dots, r_k$, then the solutions to Equation (2.4) are

定理 2.2.2 如果方程 2.5)有 k 个不同的根 $r_1, \dots, r_k$，那么方程(2.4)的解为

$$T_0(n) = a_1 r_1^n + a_2 r_2^n + \cdots + a_k r_k^n,$$

where $a_1, \dots, a_k$ are constants. There is no other solution to Equation (2.4).

其中 $a_1, \dots, a_k$ 是常量。方程(2.4)不存在其他解。

Proof: First, since $r_1, \dots, r_k$ are roots of Equation (2.5), for $i = 1, \dots, k$, we have

证明:首先，由于 $r_1, \dots, r_k$ 是方程(2.5)的根，对于 $i = 1, \dots, k$，我们有

$$r_i^k - c_1 r_i^{k-1} - c_2 r_i^{k-2} - \cdots - c_k = 0 \Rightarrow r_i^n = c_1 r_i^{n-1} + c_2 r_i^{n-2} + \cdots + c_k r_i^{n-k}.$$

Hence,

因此，

$$\begin{aligned} T_0(n) &= a_1 r_1^n + a_2 r_2^n + \cdots + a_k r_k^n \\ &= a_1\left(c_1 r_1^{n-1} + c_2 r_1^{n-2} + \cdots + c_k r_1^{n-k}\right) + \cdots + a_k\left(c_1 r_k^{n-1} + c_2 r_k^{n-2} + \cdots + c_k r_k^{n-k}\right) \\ &= c_1(a_1 r_1^{n-1} + a_2 r_2^{n-1} + \cdots + a_k r_k^{n-1}) + \cdots + c_k\left(a_1 r_1^{n-k} + a_2 r_2^{n-k} + \cdots + a_k r_k^{n-k}\right) \\ &= c_1 T_0(n-1) + \cdots + c_k T_0(n-k), \end{aligned}$$

which means $T_0(n)$ satisfies Equations (2.4).

这意味着 $T_0(n)$ 满足方程(2.4)。

Next, we show there is no other solution to Equation (2.4). Suppose $S(n)$ is a solution to Equation (2.4), i.e.,

接下来，我们证明方程(2.4)不存在其他解。假设 $S(n)$ 是方程(2.4)的一个解，即

$$S(n) = c_1 S(n-1) + c_2 S(n-2) + \cdots + c_k S(n-k).$$

We need to show that $S(n) = T_0(n)$ for all n, if the coefficients $a_1, \dots, a_k$ in $T_0(n)$ are chosen properly.

我们需要证明对于所有的 n，如果在 $T_0(n)$ 中适当选择系数 $a_1, \dots, a_k$，那么 $S(n) = T_0(n)$。

To choose the coefficients $a_1, \dots, a_k$ in $T_0(n)$, consider the linear system (with unknowns $a_1, \dots, a_n$)

为了在 $T_0(n)$ 中选择系数 $a_1, \dots, a_k$，考虑线性方程组(未知数为 $a_1, \dots, a_n$)

$$\begin{cases} a_1r_1 + a_2r_2 + \cdots + a_kr_k & = S(1) \\ a_1r_1^2 + a_2r_2^2 + \cdots + a_kr_k^2 & = S(2) \\ \cdots & \\ a_1r_1^k + a_2r_2^k + \cdots + a_kr_k^k & = S(k). \end{cases}$$

The coefficient matrix of the left hand side can be viewed as the product of a Vandermonde matrix and a diagonal matrix:

左边的系数矩阵可以看作是一个范德蒙德矩阵和一个对角矩阵的乘积:

$$\begin{bmatrix} r_1 & r_2 & \cdots & r_k \\ r_1^2 & r_2^2 & \cdots & r_k^2 \\ \cdots & \cdots & & \\ r_1^k & r_2^k & \cdots & r_k^k \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ r_1 & r_2 & \cdots & r_k \\ \cdots & \cdots & & \\ r_1^{k-1} & r_2^{k-1} & \cdots & r_k^{k-1} \end{bmatrix} \begin{bmatrix} r_1 & 0 & \cdots & 0 \\ 0 & r_2 & \cdots & 0 \\ & \cdots & \cdots & \\ 0 & 0 & \cdots & r_k \end{bmatrix}.$$

We all know that a Vandermonde matrix must be full-rank. We realize that the diagonal matrix must also be full-rank, because the roots $r_1, \ldots, r_k$ can't be equal to 0 . Consequently, the coefficient matrix must be full-rank, and the linear system must have a (unique) solution. When we use this solution as the coefficients $a_1, \ldots, a_k$ in $T_0(n)$, we are guaranteed that $T_0(1) = S(1), T_0(2) = S(2), \ldots, T_0(k) = S(k)$. A simple induction on n allows us to extend this result to $T_0(n) = S(n)$ for all positive integer n , because both $T_0(\ \ )$ and $S(\ \ )$ satisfy the same recurrence.

我们都知道范德蒙德矩阵一定是满秩的。我们意识到对角矩阵也一定是满秩的，因为根 $r_1, \ldots, r_k$ 不能等于 0。因此，系数矩阵一定是满秩的，并且线性方程组一定有(唯一)解。当我们将这个解用作 $T_0(n)$ 中的系数 $a_1, \ldots, a_k$ 时，我们保证 $T_0(1) = S(1), T_0(2) = S(2), \ldots, T_0(k) = S(k)$ 。对 n 进行简单的归纳，我们可以将这个结果推广到所有正整数 n 的 $T_0(n) = S(n)$ ，因为 $T_0(\ \ )$ 和 $S(\ \ )$ 都满足相同的递推关系。

Example 2.2.2 A function $T: \mathbb{Z}^+ \to \mathbb{Z}^+$ satisfies the following conditions:

例 2.2.2 一个函数 $T: \mathbb{Z}^+ \to \mathbb{Z}^+$ 满足以下条件:

- For all $n_1, n_2 \in \mathbb{Z}^+$ such that $\gcd(n_1, n_2) = 1, T(n_1n_2) = T(n_1)T(n_2)$.
- 对于所有满足 $\gcd(n_1, n_2) = 1, T(n_1n_2) = T(n_1)T(n_2)$ 的 $n_1, n_2 \in \mathbb{Z}^+$ 。
- For all $m, n \in \mathbb{Z}^+ (n \geq 2), T(m^n) = \frac{\left(T(m^{n-1})\right)^3}{\left(T(m^{n-2})\right)^2}$.
- 对于所有的 $m, n \in \mathbb{Z}^+ (n \geq 2), T(m^n) = \frac{\left(T(m^{n-1})\right)^3}{\left(T(m^{n-2})\right)^2}$ 。

Find the function T .

求函数 T 。

Solution: Consider an arbitrary prime p . Since $T(p^n) = \frac{\left(T(p^{n-1})\right)^3}{\left(T(p^{n-2})\right)^2}$, defining $S(n) = \log T(p^n)$, we get that $S(n) = 3S(n-1) - 2S(n-2)$. The characteristic equation of this recurrence is $x^2 - 3x + 2 = 0$, which has two distinct roots 1 and 2 . Hence, the solution to this recurrence is $S(n) = a_p 2^n + b_p$, where a_p, b_p are constants.

解:考虑任意一个素数 p。由于 $T(p^n)=\frac{\left(T(p^{n-1})\right)^3}{\left(T(p^{n-2})\right)^2}$，定义 $S(n)=\log T(p^n)$，我们得到 $S(n)=3S(n-1)-2S(n-2)$。这个递推关系的特征方程是 $x^2-3x+2=0$，它有两个不同的根 1 和 2。因此，这个递推关系的解是 $S(n)=a_p 2^n+b_p$，其中 a_p, b_p 是常数。

[3]

From $T(1)T(n)=T(n)$, we easily get that $T(1)=1$. Hence $S(0)=0$, which means $b_p=-a_p$. Thus, $S(n)=a_p(2^n-1)$, i.e., $T(p^n)=2^{a_p(2^n-1)}=\left(T(p)\right)^{2^n-1}$. For a general positive integer $p_1^{n_1}\dots p_k^{n_k}$ where $p_1,\dots,p_k$ are primes, $T\left(p_1^{n_1}\dots p_k^{n_k}\right)=\left(T(p_1)\right)^{2^{n_1}-1}\dots\left(T(p_k)\right)^{2^{n_k}-1}$. The value of $T(p)$ for each prime p can be arbitrarily chosen from positive integers.

由 $T(1)T(n)=T(n)$，我们很容易得到 $T(1)=1$。因此 $S(0)=0$，这意味着 $b_p=-a_p$。从而，$S(n)=a_p(2^n-1)$，即 $T(p^n)=2^{a_p(2^n-1)}=\left(T(p)\right)^{2^n-1}$。对于一个一般的正整数 $p_1^{n_1}\dots p_k^{n_k}$，其中 $p_1,\dots,p_k$ 是素数，$T\left(p_1^{n_1}\dots p_k^{n_k}\right)=\left(T(p_1)\right)^{2^{n_1}-1}\dots\left(T(p_k)\right)^{2^{n_k}-1}$。对于每个素数 p，$T(p)$ 的值可以从正整数中任意选取。

Theorem 2.2.2 perfectly solves Equation (2.4) when the characteristic equation has distinct roots. But in reality, we often have double roots and triple roots. What can we do then? Fortunately, we have another theorem that takes care of multiple roots.

定理 2.2.2 在特征方程有不同根时完美地解决了方程(2.4)。但实际上，我们经常会遇到二重根和三重根。那我们该怎么办呢？幸运的是，我们还有另一个处理重根的定理。

Theorem 2.2.3 If Equations (2.5) has ℓ roots $r_1,\dots,r_\ell$, where r_i is of multiplicity m_i for $i=1,\dots,\ell$, then the solutions to Equation (2.4) are

定理 2.2.3 如果方程(2.5)有 ℓ 个根 $r_1,\dots,r_\ell$，其中 r_i 对于 $i=1,\dots,\ell$ 的重数为 m_i，那么方程(2.4)的解为

$$T_0(n)=\sum_{i=1}^{\ell}\left(a_{i,1}+a_{i,2}n+\cdots+a_{i,m_i}n^{m_i-1}\right)r_i^n,$$

where $a_{ij}s$ are all constants. There is no other solution to Equation (2.4).

其中 $a_{ij}s$ 都是常数。方程(2.4)没有其他解。

The proof of Theorem 2.2.3 is quite similar to that of Theorem 2.2.2. There are two major differences:

定理 2.2.3 的证明与定理 2.2.2 的证明非常相似。有两个主要区别:

- Since r_i is a root of multiplicity m_i, on top of
- 由于 r_i 是重数为 m_i 的根，除此之外

$$r_i^k-c_1r_i^{k-1}-c_2r_i^{k-2}-\cdots-c_k=0,$$

[3] [3] If any of the roots is equal to 0, then c_k, the constant term of the characteristic equation, must be 0, contradicting our requirement that $c_1c_k\neq 0$.

[3] 如果任何一个根等于 0，那么 c_k，即特征方程的常数项，必须为 0，这与我们 $c_1c_k\neq 0$ 的要求相矛盾。

we also have

我们还有

$$kr_i^{k-1} - c_1(k-1)r_i^{k-2} - c_2(k-2)r_i^{k-3} - \cdots - c_{k-1} = 0,$$

$k(k-1)\ldots(k-m_i+2)r_i^{k-m_i+1} - c_1(k-1)\ldots(k-m_i+1)r_i^{k-m_i} - \cdots c_{k-m_i+1}(m_i-1)! = 0$. These equations help us to show that $T_0(n)$ satisfies the recurrence.

$k(k-1)\ldots(k-m_i+2)r_i^{k-m_i+1} - c_1(k-1)\ldots(k-m_i+1)r_i^{k-m_i} - \cdots c_{k-m_i+1}(m_i-1)! = 0$. 这些方程有助于我们证明 $T_0(n)$ 满足该递推关系。

- A generalization of Vandermonde matrix is confluent Vandermonde matrix [59], which is also full-rank.
- 范德蒙德矩阵的一种推广是合流范德蒙德矩阵[59]，它也是满秩的。

Theorems 2.2.2 and 2.2.3 tell us how to solve homogeneous equations. To get the total solution of Equation (2.2), we still need to find a particular solution. The bad news is that we do not have a systematic approach to finding a particular solution. The best we can do is only slightly better than the guess-and-prove method: the method of undetermined coefficients. It is slightly better in that we only need to guess what kind of function could be a particular solution, not the parameters of the function. As long as our guess is correct, the method of undetermined coefficients can help us correctly calculate the parameters of the function. Below we demonstrate how the method of undetermined coefficients works.

定理 2.2.2 和 2.2.3 告诉我们如何求解齐次方程。为了得到方程(2.2)的通解，我们仍然需要找到一个特解。坏消息是我们没有找到特解的系统方法。我们所能做的最好的方法只是比猜测并证明的方法稍好一点:待定系数法。它稍好一点是因为我们只需要猜测哪种函数可能是特解，而不是函数的参数。只要我们的猜测正确，待定系数法就能帮助我们正确计算函数的参数。下面我们演示待定系数法是如何工作的。

Example 2.2.3 Solve the recurrence

例 2.2.3 求解递推关系

$$T(n) = \frac{e^3}{2}T(n-1) + \frac{e^6}{2}T(n-2) + \left(1 - \frac{e^3+e^6}{2}\right)en + \frac{e^4}{2} + e^7.$$

Solution: The characteristic equation is

解:特征方程为

$$x^2 - \frac{e^3}{2}x - \frac{e^6}{2} = 0.$$

Solving this equation, we get two roots $r_1 = e^3$ and $r_2 = -\frac{e^3}{2}$. Consequently, the solution to the homogeneous equation is $T_0(n) = a_1e^{3n} + a_2\left(-\frac{e^3}{2}\right)^n$, where a_1, a_2 are constants.

解这个方程，我们得到两个根 $r_1 = e^3$ 和 $r_2 = -\frac{e^3}{2}$。因此，齐次方程的解为 $T_0(n) = a_1e^{3n} + a_2\left(-\frac{e^3}{2}\right)^n$，其中 a_1, a_2 是常数。

We guess a particular solution could be linear 4 So we plug $T(n) = un + v$ into the original equation and get that

我们猜测一个特解可能是线性的 4 所以我们将 $T(n)=un+v$ 代入原方程，得到

$$un+v=\frac{e^3}{2}(u(n-1)+v)+\frac{e^6}{2}(u(n-2)+v)+\left(1-\frac{e^3+e^6}{2}\right)en+\frac{e^4}{2}+e^7$$

$$\Rightarrow u=\frac{e^3+e^6}{2}\cdot(u-e)+e;\ v=-\frac{u(e^3+2e^6)}{2}+\frac{e^3+e^6}{2}\cdot v+\frac{e^4}{2}+e^7$$

Solving the last equation system, we get that $u=e, v=0$, which means $T_1(n)=en$ is a particular solution to the original recurrence. Hence, the total solution is

解最后一个方程组，我们得到 $u=e, v=0$ ，这意味着 $T_1(n)=en$ 是原递推关系的一个特解。因此，通解为

$$T(n)=T_0(n)+T_1(n)=a_1e^{3n}+a_2\left(-\frac{e^3}{2}\right)^n+en.$$

Theorems 2.2.1, 2.2.2, and 2.2.3, give us a very powerful tool to solve recurrences. Together with methods discussed in future sections, they enable us to solve recurrences in systematics ways. Nevertheless, these theorems and methods are not all we can use when solving recurrences. You will always encounter "strange" recurrences that can never be solved in a systematic way. What can you do in that case? You should bravely guess a solution, just like in the solution of Example 2.2.3. Below we do another example that requires you to guess (and prove).

定理 2.2.1、2.2.2 和 2.2.3 为我们提供了一个非常强大的工具来求解递推关系。与后续章节讨论的方法一起，它们使我们能够以系统的方式求解递推关系。然而，这些定理和方法并不是我们求解递推关系时所能使用的全部。你总会遇到“奇怪”的递推关系，它们永远无法以系统的方式求解。在那种情况下你能做什么呢？你应该勇敢地猜测一个解，就像在例 2.2.3 的解中那样。下面我们再做一个例子，需要你进行猜测(并证明)。

Example 2.2.4 (USAMO 1990, Day 1, Problem 2) Suppose $T(1)=\sqrt{x^2+48}$ and $T(n+1)=\sqrt{x^2+6T(n)}$ for $n\geq 1$. Solve the equation $T(n)=2x$ for all positive integer n and all real number x .

例 2.2.4(1990 年美国数学奥林匹克竞赛，第一天，问题 2)假设 $T(1)=\sqrt{x^2+48}$ 且 $T(n+1)=\sqrt{x^2+6T(n)}$ 对于 $n\geq 1$ 。求解方程 $T(n)=2x$ 对于所有正整数 n 和所有实数 x 。

4

Solution: Obviously $n=1, x=4$ is a solution. Further calculations tell us that, $\forall n_0\in\mathbb{Z}^+$, $n=n_0, x=4$ is a solution. Now we show that there is no other solution. Clearly, $x\geq 0$.

解:显然 $n=1, x=4$ 是一个解。进一步的计算告诉我们， $\forall n_0\in\mathbb{Z}^+$ ， $n=n_0, x=4$ 是一个解。现在我们证明没有其他解。显然， $x\geq 0$ 。

If $x>4$, then $3x^2>48\Rightarrow 4x^2>x^2+48\Rightarrow T(1)<2x$. For each positive integer n , we have $T(n)<2x\Rightarrow x^2+6T(n)<x^2+12x<x^2+3x^2=4x^2\Rightarrow T(n+1)<2x.$

如果 $x>4$ ，那么 $3x^2>48\Rightarrow 4x^2>x^2+48\Rightarrow T(1)<2x$ 。对于每个正整数 n ，我们有 $T(n)<2x\Rightarrow x^2+6T(n)<x^2+12x<x^2+3x^2=4x^2\Rightarrow T(n+1)<2x.$

[4] [4] Don’t ask me why. I have no idea about how to make guesses.

[4] 别问我为什么。我不知道该怎么猜测。

If $x < 4$, then $3x^2 < 48 \Rightarrow 4x^2 < x^2 + 48 \Rightarrow T(1) > 2x$. For each positive integer n , we have $T(n) > 2x \Rightarrow x^2 + 6T(n) > x^2 + 12x > x^2 + 3x^2 = 4x^2 \Rightarrow T(n+1) > 2x$.

如果 $x < 4$ ，那么 $3x^2 < 48 \Rightarrow 4x^2 < x^2 + 48 \Rightarrow T(1) > 2x$ 。对于每个正整数 n ，我们有 $T(n) > 2x \Rightarrow x^2 + 6T(n) > x^2 + 12x > x^2 + 3x^2 = 4x^2 \Rightarrow T(n+1) > 2x$ 。

Consequently, our initial guess of $x = 4$ (for all positive integer n) is the only solution.

因此，我们最初对 $x = 4$ 的猜测(对于所有正整数 n)是唯一的解。

Sometimes we don't even have the recurrence available, so that you need to make a guess about it in the first place.

有时我们甚至没有可用的递归关系，所以你首先需要对其进行猜测。

Example 2.2.5 (IMO 1998 Shortlist C3, Generalized [71]) Cards numbered 1 to n are arranged at random in a row with $n \geq 5$. In a move, one may choose any block of consecutive cards whose numbers are in ascending or descending order, and switch the block around. For example, if $n = 9$, then916532748may be changed to913562748. Prove that in at most $2n - 6$ moves, one can arrange the n cards so that their numbers are in ascending or descending order.

例 2.2.5(1998 年国际数学奥林匹克竞赛短名单 C3，推广形式[71])编号从 1 到 n 的卡片随机排成一行，有 $n \geq 5$ 。在一次移动中，可以选择任何一组连续的卡片，其数字按升序或降序排列，然后将这组卡片翻转。例如，如果 $n = 9$ ，那么 916532748 可以变为 913562748。证明在最多 $2n - 6$ 次移动中，可以将 n 张卡片排列成其数字按升序或降序排列。

Solution: For $n \geq 5$, we have the following simple recursive algorithm that can arrange $n + 1$ cards: First, we do a recursive call to arrange first n cards. Assume that, after the recursive call, we still need to place the last card between the i th and $(i + 1)$ st cards. We use one move to reverse the order of the $(i + 1)$ st through n th cards; then we use another move to reverse the order of the $(i + 1)$ st through $(n + 1)$ st cards. Easy to see that we are done after these two additional moves. (Note that we could have arranged the last n cards and then inserted the first card at the correct location. In general, two moves are sufficient for inserting the first/last card to n cards that have already been sorted.)

解:对于 $n \geq 5$ ，我们有以下简单的递归算法来排列 $n + 1$ 张卡片:首先，我们进行一次递归调用以排列前 n 张卡片。假设在递归调用之后，我们仍需要将最后一张卡片放置在第 i 张和第 $(i + 1)$ 张卡片之间。我们用一次移动来反转从第 $(i + 1)$ 张到第 n 张卡片的顺序；然后我们再用一次移动来反转从第 $(i + 1)$ 张到第 $(n + 1)$ 张卡片的顺序。很容易看出，在这额外的两次移动之后我们就完成了。(注意，我们本可以先排列最后 n 张卡片，然后将第一张卡片插入到正确的位置。一般来说，对于将第一张/最后一张卡片插入到已经排好序的 n 张卡片中，两次移动就足够了。)

Denote by $T(n)$ the number of moves needed by the above algorithm (in the worst case), when the input is n cards. We immediately get that $T(n + 1) = T(n) + 2$. The total solution to this recurrence is $T(n) = 2n + c$, where c is a constant. In order to show that $c = -6$, we need to figure out the value of $T(5)$. Obviously, as long as $T(5) = 4$, we can obtain $c = -6$. It might be a bit surprising that showing $T(5) = 4$, or, equivalently, arranging 5 cards in 4 moves, is not trivial. To achieve this objective, we distinguish two cases:

用 $T(n)$ 表示上述算法(在最坏情况下)处理输入为 n 张卡片时所需的移动次数。我们立即得到 $T(n + 1) = T(n) + 2$ 。这个递归关系的通解是 $T(n) = 2n + c$ ，其中 c 是一个常数。为了证明 $c = -6$ ，我们需要确定 $T(5)$ 的值。显然，只要 $T(5) = 4$ ，我们就能得到 $c = -6$ 。可能有点令人惊讶的是，证明 $T(5) = 4$ ，或者等价地，用 4 次移动排列 5 张卡片，并非易事。为了实现这个目标，我们区分两种情况:

Case A: The first or last card is numbered 1 or 5. W. L. O. G., assume the initial configuration is $1\ \ w\ \ x\ \ y\ \ z$. In a single move, we can easily arrange the last three cards in ascending/descending order. (Do you see how?) Next, we use two moves to insert w to them. After that, if the last four cards are descending, we use another move to reverse them; otherwise, we do nothing. In total, we need either 3 or 4 moves.

情形 A:第一张或最后一张牌的数字是 1 或 5。不失一般性，假设初始配置是 $1\ \ w\ \ x\ \ y\ \ z$。在一次移动中，我们可以轻松地将最后三张牌按升序/降序排列。(你明白怎么做吗？)接下来，我们用两次移动将 w 插入其中。之后，如果最后四张牌是降序的，我们再用一次移动将它们反转；否则，我们什么也不做。总共，我们需要 3 次或 4 次移动。

Case B: Neither 1 nor 5 is the first or the last card. This includes two subcases.

情形 B:1 和 5 都既不是第一张牌也不是最后一张牌。这包括两个子情形。

Subcase B1: One of 1 and 5 is right in the middle. W. L. O. G., assume the initial configuration is $x15yz$. If $y < z$, then we do two moves: $x15yz \to x1y5z \to x1yz5$. If $y > z$, then we do a single move: $x\ 1\ 5\ y\ z \to x\ 1\ z\ y\ 5$. After these two/one move, the last four cards are already sorted in ascending order. Next, we use two additional moves to insert x to them. So, in total, we have used either 3 or 4 moves.

子情形 B1:1 和 5 中的一个恰好在中间。不失一般性，假设初始配置是 $x15yz$。如果 $y < z$，那么我们进行两次移动: $x15yz \to x1y5z \to x1yz5$。如果 $y > z$，那么我们进行一次移动: $x\ 1\ 5\ y\ z \to x\ 1\ z\ y\ 5$。在这两次/一次移动之后，最后四张牌已经按升序排列。接下来，我们再用两次移动将 x 插入其中。所以，总共我们使用了 3 次或 4 次移动。

Subcase B2: Neither 1 nor 5 is right in the middle. W. L. O. G., assume the initial configuration is $x\ 1\ y\ 5z$. If $y < z$, then we do a single move: $x\ 1\ y\ 5z \to x\ 1\ y\ z\ 5$. If $y > z$, then we do two moves: $x\ 1\ y\ 5\ z \to x\ 1\ 5\ y\ z \to x\ 1\ z\ y\ 5$. After these one/two moves, the last four cards are already sorted in ascending order. Next, we use two additional moves to insert x to them. Again, in total, we have used either 3 or 4 moves.

子情况 B2:1 和 5 都不在中间位置。不失一般性，假设初始配置为 $x\ 1\ y\ 5z$。如果 $y < z$，那么我们进行一步移动: $x\ 1\ y\ 5z \to x\ 1\ y\ z\ 5$。如果 $y > z$，那么我们进行两步移动: $x\ 1\ y\ 5\ z \to x\ 1\ 5\ y\ z \to x\ 1\ z\ y\ 5$。在这一步或两步移动之后，最后四张牌已经按升序排列。接下来，我们再用两步移动将 x 插入其中。同样，总共我们使用了 3 步或 4 步移动。

2.3 The Generating Function Method

2.3 生成函数法

You might have studied, or at least heard of, generating functions, which was first introduced by de Moivre [5] They are a valuable tool in combinatorics. In this section, we start from scratch and show you how to solve recurrences using generating functions.

你可能已经学过，或者至少听说过生成函数，它最早由棣莫弗(de Moivre)引入 [5] 它们是组合数学中的一个重要工具。在本节中，我们将从头开始，向你展示如何使用生成函数来求解递推关系。

Definition 2.3.1 For function $T(n)$ defined on natural numbers [6] , its (ordinary) generating function is

定义 2.3.1 对于定义在自然数 [6] 上的函数 $T(n)$，其(普通)生成函数为

$$G(z) = \sum_{i=0}^{\infty} T\,(i) z^i.$$

What is the generating function $G(z)$? Why do we need it? It is just another way to express the function $T(n)$ —it carries exactly the same information as $T(n)$, although it looks a bit different. We can easily convert $T(n)$ to its generating function $G(z)$, and vice versa. The advantage of having another expression of the same mathematical object is that sometimes we can process it more easily. For example, in certain situations (which we shall see shortly), finding the generating function of a solution to a recurrence is easier than finding the solution itself. In that case, our strategy is to figure out the generating function first, and then convert it to the actual solution.

生成函数 $G(z)$ 是什么？我们为什么需要它？它只是表达函数 $T(n)$ 的另一种方式——它携带的信息与 $T(n)$ 完全相同，尽管看起来有点不同。我们可以很容易地将 $T(n)$ 转换为它的生成函数 $G(z)$ ，反之亦然。拥有同一个数学对象的另一种表达式的好处是，有时我们可以更轻松地对其进行处理。例如，在某些情况下(我们很快就会看到)，找到递推关系解的生成函数比找到解本身更容易。在那种情况下，我们的策略是先找出生成函数，然后将其转换为实际的解。

5

The coefficient of x^n on the left hand side must be equal to that on the right hand side:

左边 x^n 的系数必须等于右边的系数:

$$\sum_{0 \le k \le n} \binom{n+1}{k} \binom{n+1}{n-k} = \sum_{0 \le k \le n} \binom{n+2}{k} \binom{n}{n-k},$$

which is clearly equivalent to what we need to show.

这显然等同于我们需要证明的内容。

---

[5] [5] Abraham de Moivre (1667-1754) was a French mathematician. When he was a student, he received only a moderate amount of formal mathematical training through private lessons. But he became a professional mathematician anyway. Being a protestant, he had to flee to England due to religious persecutions. Living in England, he was very close to Newton, who claimed that "he knows all these things (mathematics) better than I do." In addition to generating functions, de Moivre had another great contribution to mathematics, de Moivre's Formula: $(\cos x + i\sin x)^n = \cos nx + i\sin nx$.

[5] 亚伯拉罕·棣莫弗(1667 - 1754)是一位法国数学家。他学生时代时，仅通过私人授课接受了适度的正规数学训练。但无论如何，他成为了一名职业数学家。作为一名新教徒，由于宗教迫害，他不得不逃往英国。在英国生活期间，他与牛顿关系密切，牛顿声称“他对这些东西(数学)的了解比我还深”。除了生成函数，棣莫弗对数学还有另一大贡献，即棣莫弗公式: $(\cos x + i\sin x)^n = \cos nx + i\sin nx$ 。

[6] For convenience, we allow n to be equal to 0 here, because otherwise the generating function would not have a constant term.

[6] 为方便起见，我们在此允许 n 等于 0，因为否则生成函数将没有常数项。

Problem Set 4

问题集 4

Problem 59 [Difficulty Estimate=1.6] Prove that Equation (2.6) has only two solutions, $T(n) = 0$ and the one given in Formula 2.7).

问题 59 [难度估计 = 1.6] 证明方程(2.6)只有两个解， $T(n) = 0$ 和公式 2.7 中给出的那个解。

Problem 60 [Difficulty Estimate = 1.1] To calculate the product of two polynomials $f(x) = ax + b$ and $g(x) = ux^2 + vx + w$, a naive algorithm needs 6 multiplications of coefficients. How can you do better?

问题 60 [难度估计 = 1.1] 为了计算两个多项式 $f(x) = ax + b$ 和 $g(x) = ux^2 + vx + w$ 的乘积，一种朴素的算法需要进行 6 次系数乘法。你怎样能做得更好？

Problem 61 [Difficulty Estimate = 2.1] Prove the number of linearly independent solutions to Equation (2.4) is equal to the number of distinct roots of its characteristic equation.

问题 61 [难度估计 = 2.1] 证明方程(2.4)线性无关解的数量等于其特征方程不同根的数量。

Problem 62 [Difficulty Estimate = 0.6] Solve the recurrence $T(n) = 3T(n-1) + (-3)^n$.

问题 62 [难度估计 = 0.6] 求解递推关系 $T(n) = 3T(n-1) + (-3)^n$ 。

Problem 63 (Chapter 1 Exercise 20 of [26]) [Difficulty Estimate=2.4] Solve the recurrence $T(2n) = 4T(n) + an + b,\ T(2n+1) = 4T(n) + cn + d$, where $T(1) = u$ is given.

问题 63([26]中第 1 章练习 20)[难度估计=2.4] 求解递归关系 $T(2n) = 4T(n) + an + b,\ T(2n+1) = 4T(n) + cn + d$ ， 其中 $T(1) = u$ 为已知条件。

Problem 64 [Difficulty Estimate=2.3] Solve the recurrence

问题 64 [难度估计=2.3] 求解递归关系

$$T(n) = \begin{cases} 3T(n-1) - 5T(n-2) & \text{if } n \equiv 0 \pmod 4 \\ 6T(n-1) - 5T(n-4) & \text{if } n \equiv 1 \pmod 4 \\ T(n-2) & \text{if } n \equiv 2 \pmod 4 \\ 2T(n-2) & \text{if } n \equiv 3 \pmod 4. \end{cases}$$

Problem 65 [Difficulty Estimate = 2.0] Solve the recurrence $T(n+2) = \frac{T(n+1)T(n)}{\sqrt{T(n+1)^2 + T(n)^2 + 1}}$, where $T(1) = 1, T(2) = 5$.

问题 65 [难度估计 = 2.0] 求解递归关系 $T(n+2) = \frac{T(n+1)T(n)}{\sqrt{T(n+1)^2 + T(n)^2 + 1}}$， 其中 $T(1) = 1, T(2) = 5$ 。

Problem 66 ([85], Problem 6) Let $a_0 = 0, a_1 = 1$, and for $n \geq 2$, let $a_n = 17a_{n-1} - 70a_{n-2}$. For $n > 6$, show that the first (most significant) digit of a_n (when written in base 10) is 3 .

问题 66([85]，问题 6)设 $a_0 = 0, a_1 = 1$ ，对于 $n \geq 2$ ， 设 $a_n = 17a_{n-1} - 70a_{n-2}$ 。对于 $n > 6$ ， 证明 a_n (以十进制表示时)的首位(最高有效位)数字为 3 。

2.5 Asymptotic Complexity, Recursion Trees, and the Mas- ter Theorem

2.5 渐近复杂度、递归树与主定理

So far we have studied some powerful techniques for solving recurrences. Unfortunately, even with these techniques (plus other techniques, which we did not have time to cover), we still have a lot of recurrences that are not solvable in the sense that we can't figure out any particular solution. Nevertheless, as we have seen, we often have to solve recurrences in order to analyze the complexity of recursive algorithms. One approach frequently used is to look at so called asymptotic solutions, rather than precise solutions. Below we first review some notation.

到目前为止，我们已经研究了一些求解递归关系的强大技术。不幸的是，即使有了这些技术(再加上其他我们没时间涵盖的技术)，我们仍然有很多递归关系无法求解，也就是说我们找不到任何特定的解。然而，正如我们所看到的，为了分析递归算法的复杂度，我们经常需要求解递归关系。一种常用的方法是考虑所谓的渐近解，而不是精确解。下面我们首先回顾一些符号。

Definition 2.5.1 Let $T(n)$ and $f(n)$ be positive valued functions defined on positive integers. We say $T(n) = O\big(f(n)\big)$ if there exist $N > 0$ and $M > 0$ such that for every $n > N, T(n) \leq Mf(n)$. We say $T(n) = \Theta\big(f(n)\big)$ if $T(n) = O\big(f(n)\big)$ and $f(n) = O\big(T(n)\big)$.

定义 2.5.1 设 $T(n)$ 和 $f(n)$ 是定义在正整数上的正数值函数。如果存在 $N > 0$ 和 $M > 0$ 使得对于每个 $n > N, T(n) \leq Mf(n)$ ，我们就说 $T(n) = O\big(f(n)\big)$ 。如果 $T(n) = O\big(f(n)\big)$ 且 $f(n) = O\big(T(n)\big)$ ，我们就说 $T(n) = \Theta\big(f(n)\big)$ 。

Intuitively, we are examining the long-term growth of function $T(n)$, i.e., we are interested in the growth of $T(n)$ only when n is large enough. $T(n) = O\big(f(n)\big)$ means that, when n is really large, $T(n)$ grows no faster than a constant times $f(n)$. Why do we require $T(n)$ to grow no faster than a constant times $f(n)$? Why not just no faster than $f(n)$ itself? Well, a constant times $f(n)$ is considered growing exactly as fast as $f(n)$ itself, because we are facing huge differences between functions, and a mere constant factor does not matter at all.

直观地说，我们在研究函数 $T(n)$ 的长期增长情况，也就是说，我们只对 n 足够大时 $T(n)$ 的增长情况感兴趣。 $T(n) = O\big(f(n)\big)$ 意味着，当 n 非常大时， $T(n)$ 的增长速度不超过常数乘以 $f(n)$ 的增长速度。为什么我们要求 $T(n)$ 的增长速度不超过常数乘以 $f(n)$ 呢？为什么不只是不超过 $f(n)$ 本身呢？嗯，常数乘以 $f(n)$ 被认为与 $f(n)$ 本身的增长速度完全相同，因为我们面对的是函数之间的巨大差异，一个纯粹的常数因子根本无关紧要。

To see this, consider, e.g., the functions $f(n) = 2 \times 10^{100}n + 3 \times 10^{200}$, and $g(n) = n$. When n is very large, $f(n)$ is roughly 2×10^{100} times $g(n)$. You might think 2×10^{100} is a lot, but it is not. In comparison, consider $h(n) = n^2$. When n is very large, $h(n)$ can be $10^{10^{10}}$ times $g(n)$, $10^{10^{10^{10}}}$ times $g(n)$, ... arbitrarily many times $g(n)$. So there is a huge difference between $h(n)$ and $g(n)$, but not between $f(n)$ and $g(n)$. So we should place $f(n)$ and $g(n)$ in the same category, and $h(n)$ in a different one. We say the latter category of functions grow faster than the former, asymptotically.

要明白这一点，例如，考虑函数 $f(n) = 2 \times 10^{100}n + 3 \times 10^{200}$ 和 $g(n) = n$ 。当 n 非常大时， $f(n)$ 大致是 2×10^{100} 乘以 $g(n)$ 。你可能会认为 2×10^{100} 是个很大的数，但其实并非如此。相比之下，考虑 $h(n) = n^2$ 。当 n 非常大时， $h(n)$ 可以是 $10^{10^{10}}$ 乘以 $g(n)$ ， $10^{10^{10^{10}}}$ 乘以 $g(n), \ldots$ ，甚至可以是 $g(n)$ 的任意倍数。所以 $h(n)$ 和 $g(n)$ 之间有巨大差异，但 $f(n)$ 和 $g(n)$ 之间没有。因此，我们应该将 $f(n)$

和 $g(n)$ 归为同一类，而将 $h(n)$ 归为另一类。我们说后一类函数的增长速度比前一类快，是渐近意义上的。

When we compare the growths of positive-valued functions defined on positive integers, the big- O notation defined above is analogous to $\leq$ for comparison of real numbers, meaning that $T(n)$ grows asymptotically no faster than $f(n)$. For example, if $g(n) = n$ and $h(n) = n^2$, then $g(n) = O\big(h(n)\big)$. The big- Θ for growths of functions is similar to $=$ for real numbers, meaning that $T(n)$ grows asymptotically as fast as $f(n)$. For example, if $f(n) = 2 \times 10^{100}n + 3 \times 10^{200}$, and $g(n) = n$, then $f(n) = \Theta\big(g(n)\big)$.

当我们比较定义在正整数上的正值函数的增长情况时，上面定义的大 O 记号类似于用于比较实数的 $\leq$ ，这意味着 $T(n)$ 的渐近增长速度不超过 $f(n)$ 。例如，如果 $g(n) = n$ 和 $h(n) = n^2$ ，那么 $g(n) = O\big(h(n)\big)$ 。用于函数增长的大 Θ 类似于用于实数的 $=$ ，这意味着 $T(n)$ 的渐近增长速度与 $f(n)$ 相同。例如，如果 $f(n) = 2 \times 10^{100}n + 3 \times 10^{200}$ ，且 $g(n) = n$ ，那么 $f(n) = \Theta\big(g(n)\big)$ 。

These notations are somewhat unusual in that they do not mean what they appear to mean. Specifically, don't regard $T(n) = O\big(f(n)\big)$ as an equation and try to manipulate it like an equation. You can't derive $O\big(f(n)\big) = T(n)$ from $T(n) = O\big(f(n)\big)$. In fact, $O\big(f(n)\big) = T(n)$ is meaningless and we should absolutely avoid writing anything like that.

这些记号有些不同寻常，因为它们并不意味着它们表面看起来的意思。具体来说，不要把 $T(n) = O\big(f(n)\big)$ 当作一个等式并试图像处理等式那样对其进行运算。你不能从 $T(n) = O\big(f(n)\big)$ 推出 $O\big(f(n)\big) = T(n)$ 。事实上， $O\big(f(n)\big) = T(n)$ 是没有意义的，我们绝对应该避免写出那样的东西。

6

Another point to notice is that while the definition of big-O notation looks like that of sequence limits in calculus, we can't use sequence limits to define it. Specifically, we can't say $T(n) = O\big(f(n)\big)$ if and only if there exists $M > 0$ such that $\lim\limits_{n\to\infty} \frac{T(n)}{f(n)} \leq M$. The reason is that $\lim\limits_{n\to\infty}$ may not exist. If it exists, and it is less than or equal to a constant M , then we are confident that $T(n) = O\big(f(n)\big)$. However, in case it does not exist, we may still have $T(n) = O\big(f(n)\big)$. For example, consider

另一个需要注意的点是，虽然大 O 符号的定义看起来与微积分中的序列极限定义相似，但我们不能用序列极限来定义它。具体来说，我们不能说当且仅当存在某个 $M > 0$ 使得 $\lim\limits_{n\to\infty} \frac{T(n)}{f(n)} \leq M$ 时， $T(n) = O\big(f(n)\big)$ 。原因是 $\lim\limits_{n\to\infty}$ 可能不存在。如果它存在且小于或等于某个常数 M ，那么我们可以确定 $T(n) = O\big(f(n)\big)$ 。然而，如果它不存在，我们仍然可能有 $T(n) = O\big(f(n)\big)$ 。例如，考虑

$$T(n) = \begin{cases} n & \text{if } n \text{ is odd} \\ 2n & \text{if } n \text{ is even,} \end{cases}$$

and $f(n) = n$. Obviously $\lim\limits_{n\to\infty}$ does not exist, but $T(n) = O\big(f(n)\big)$.

以及 $f(n) = n$ 。显然 $\lim\limits_{n\to\infty}$ 不存在，但 $T(n) = O\big(f(n)\big)$ 。

---

6 9 These notations are used both in algorithm analysis and in mathematical disciplines like number theory. There are minor differences between the interpretations of these notations in different fields. We follow the tradition of algorithm analysis.

9 这些记号在算法分析和数论等数学学科中都有使用。这些记号在不同领域的解释略有不同。我们遵循算法分析的传统。

Example 2.5.1 Prove the following statements.

例 2.5.1 证明以下命题。

1. If $d \leq d'$, any degree-d polynomial is $O(n^{d'})$.

1. 如果 $d \leq d'$，那么任何 d 次多项式都是 $O(n^{d'})$。

2. Any degree-d polynomial is $\Theta(n^d)$.

2. 任何 d 次多项式都是 $\Theta(n^d)$。

3. If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.

3. 如果 $f(n) = O(g(n))$ 且 $g(n) = O(h(n))$，那么 $f(n) = O(h(n))$。

4. Any polynomial $f(n)$ grows no faster than the exponential function, i.e., $f(n) = O(e^n)$.

4. 任何多项式 $f(n)$ 的增长速度都不超过指数函数，即 $f(n) = O(e^n)$。

5. Any logarithm function $f(n) = a\log_b n (a > 0, b > 1)$ grows no faster than a polynomial, i.e., $f(n) = O(n^\epsilon)$ for all $\epsilon > 0$.

5. 任何对数函数 $f(n) = a\log_b n (a > 0, b > 1)$ 的增长速度都不超过多项式，即对于所有的 $\epsilon > 0$，$f(n) = O(n^\epsilon)$。

Solution: We skip the first three and show the last two. For all $d > 0$,

解:我们跳过前三个，证明后两个。对于所有的 $d > 0$，

$$e^n = \sum_{i=0}^{\infty} \frac{n^i}{i!} \Rightarrow n^d < d!\, e^n.$$

Hence, $n^d = O(e^n)$, and thus by the second and third statements in this example, we get that any degree-d polynomial is $O(e^n)$.

因此，$n^d = O(e^n)$，从而根据本例中的第二个和第三个命题，我们得出任何 d 次多项式都是 $O(e^n)$。

In $n = O(e^n)$, we replace n with $\epsilon \ln n$, and get that $\epsilon \ln n = O\left(e^{\epsilon \ln n}\right) = O(n^\epsilon)$. Since $a\log_b n = \frac{a}{\epsilon \ln b} \cdot \epsilon \ln n = O(\epsilon \ln n)$, by the third statement above, we get that $a\log_b n = O(n^\epsilon)$.

在 $n = O(e^n)$ 中，我们用 $\epsilon \ln n$ 替换 n，得到 $\epsilon \ln n = O\left(e^{\epsilon \ln n}\right) = O(n^\epsilon)$。由于 $a\log_b n = \frac{a}{\epsilon \ln b} \cdot \epsilon \ln n = O(\epsilon \ln n)$，根据上述第三个命题，我们得出 $a\log_b n = O(n^\epsilon)$。

Example 2.5.2 Recall that a divide-and-conquer algorithm divides the original problem into smaller ones, which are easier to solve. Please design a divide-and-conquer algorithm for finding the kth smallest element in an n-array. Note that a naive algorithm sorts the elements before finding the k th smallest one and thus requires $O(n\log n)$ comparisons. Can you do better?

例 2.5.2 回想一下，分治算法会将原问题分解为更小的问题，这些问题更容易解决。请设计一个分治算法，用于在一个 n 数组中找到第 k 小的元素。注意，一个简单的算法会在找到第 k 小的元素之前对元素进行排序，因此需要 $O(n\log n)$ 次比较。你能做得更好吗？

Solution: We examine the array from the beginning, and always keep the k smallest elements we have seen so far, and make sure these k elements are always sorted. When we are done, we have found the k smallest elements, not just the k th smallest one, of the entire array.

解决方案:我们从数组开头开始检查，并始终保留到目前为止所见到的 k 个最小元素，并确保这 k 个元素始终是有序的。当我们完成这个过程时，我们就找到了整个数组中 k 个最小元素，而不仅仅是第 k 小的元素。

Easy to see we need only $O(n\log k)$ comparisons.

很容易看出我们只需要 $O(n\log k)$ 次比较。

Asymptotic notations follow some simple rules, like:

渐近符号遵循一些简单规则，比如:

$$f_1(n) = O\big(g_1(n)\big), f_2(n) = O\big(g_2(n)\big) \Rightarrow f_1(n) + f_2(n) = O\big(g_1(n) + g_2(n)\big);$$

$$f_1(n) = O\big(g_1(n)\big), f_2(n) = O\big(g_2(n)\big) \Rightarrow f_1(n)f_2(n) = O\big(g_1(n)g_2(n)\big).$$

Furthermore, the effect of summation operation on big-O has a property similar to that of set union operation on cardinality:

此外，求和运算对大 O 的影响具有与基数的集合并集运算类似的性质:

Definition 2.5.2 Suppose S is a set of positive valued functions defined on natural numbers. We say $f(n) \in S$ is a maximum of S if for all $g(n) \in S, g(n) = O\big(f(n)\big)$. In this case, we write $\max S = f(n)$.

定义 2.5.2 假设 S 是一组定义在自然数上的正数值函数。如果对于所有的 $g(n) \in S, g(n) = O\big(f(n)\big)$，我们称 $f(n) \in S$ 是 S 中的最大值。在这种情况下，我们写作 $\max S = f(n)$。

Proposition 2.5.1 If S is a finite set of positive valued functions defined on natural numbers, and $|S| = O(1)$, then $\sum_{g(n)\in S} g\,(n) = O(\max S)$.

命题 2.5.1 如果 S 是一组定义在自然数上的有限正数值函数集合，并且 $|S| = O(1)$，那么 $\sum_{g(n)\in S} g\,(n) = O(\max S)$。

The proof is very easy and thus skipped, but do you see why we need S to be of a constant (i.e., $O(1)$) size in this proposition?

证明非常简单，因此略过，但你明白为什么在这个命题中我们需要 S 具有常数(即 $O(1)$)大小吗？

Example 2.5.3 Suppose that $a > 1$, and that $f(n) = a^{u(n)}, g(n) = a^{v(n)}$. Does $u(n) = O\big(v(n)\big)$ imply $f(n) = O\big(g(n)\big)$? Does $f(n) = O\big(g(n)\big)$ imply $u(n) = O\big(v(n)\big)$?

示例 2.5.3 假设 $a > 1$，并且 $f(n) = a^{u(n)}, g(n) = a^{v(n)}$。$u(n) = O\big(v(n)\big)$ 是否意味着 $f(n) = O\big(g(n)\big)$？$f(n) = O\big(g(n)\big)$ 是否意味着 $u(n) = O\big(v(n)\big)$？

Solution: No, $u(n) = O\big(v(n)\big)$ does not imply $f(n) = O\big(g(n)\big)$. Let $u(n) = 2\log n, v(n) = \log n$. Clearly, $u(n) = O\big(v(n)\big)$. However, $f(n) = 2^{u(n)} = n^2, g(n) = 2^{v(n)} = n$. It would be so wrong to say $n^2 = O(n)$.

解决方案:不，$u(n) = O\big(v(n)\big)$ 并不意味着 $f(n) = O\big(g(n)\big)$。令 $u(n) = 2\log n, v(n) = \log n$。显然，$u(n) = O\big(v(n)\big)$。然而，$f(n) = 2^{u(n)} = n^2, g(n) = 2^{v(n)} = n$。说 $n^2 = O(n)$ 是非常错误的。

Yes, $f(n) = O\big(g(n)\big)$ implies that $u(n) = O\big(v(n)\big)$. The proof is trivial and thus skipped.

是的， $f(n)=O\big(g(n)\big)$ 意味着 $u(n)=O\big(v(n)\big)$ 。证明很简单，因此略过。

In order to get familiar with the asymptotic notations, let's play with a couple of more challenging examples.

为了熟悉渐近符号，让我们来看几个更具挑战性的例子。

Example 2.5.4 Does there exist any positive-valued function $f(\ \ )$ defined on the set of positive integers that satisfies both of the following conditions? (a) There is no constant $c>0$ such that $f(x)=O(x^c)$. (b) There are no constants $c>0, a>1$ such that $a^{x^c}=O\big(f(x)\big)$. If yes, give an example. If no, prove your answer.

例 2.5.4 是否存在定义在正整数集上的正值函数 $f(\ \ )$ ，同时满足以下两个条件？(a) 不存在常数 $c>0$ 使得 $f(x)=O(x^c)$ 。(b) 不存在常数 $c>0, a>1$ 使得 $a^{x^c}=O\big(f(x)\big)$ 。如果存在，给出一个例子。如果不存在，请证明你的答案。

Solution: Yes. One example: $f(x)=x^{\log x}$. We show that $f(x)$ satisfies both requirements by contradiction. (a) Suppose that there is a constant $c>0$ such that $f(x)=O(x^c)$, which means $\exists N>0, M>0$ such that $\forall x>N, f(x)\le Mx^c$, i.e., $x^{\log x}\le Mx^c$. W. L. O. G., assume $x>1$. This is equivalent to $\log x\le \log_x M+c=\frac{\log M}{\log x}+c$. To obtain a contradiction, it suffices to choose x that is greater than both M and 2^{1+c} . (b) Suppose that there are constants $c>0, a>1$ such that $a^{x^c}=O\big(f(x)\big)$, which means $\exists N>0, M>0$ such that $\forall x>N, a^{x^c}\le Mf(x)$, i.e., $a^{x^c}\le Mx^{\log x}$. This implies that $x^c\log a\le \log M+\log^2 x<\left(\sqrt{\log M}+\log x\right)^2$, which, in turn, implies that $x^{\frac{c}{2}}\sqrt{\log a}<\sqrt{\log M}+\log x$. Since $\log x=O\left(x^{\frac{c}{4}}\sqrt{\log a}\right)$, when x is sufficiently large, there exists a constant $d>0$ such that $x^{\frac{c}{2}}\sqrt{\log a}<x^{\frac{c}{4}}\sqrt{\log a}d$, which implies that $x^{\frac{c}{4}}<d$. Contradiction.

解:存在。一个例子: $f(x)=x^{\log x}$ 。我们用反证法证明 $f(x)$ 满足这两个要求。(a) 假设存在一个常数 $c>0$ 使得 $f(x)=O(x^c)$ ，这意味着 $\exists N>0, M>0$ 使得 $\forall x>N, f(x)\le Mx^c$ ，即 $x^{\log x}\le Mx^c$ 。不失一般性，假设 $x>1$ 。这等价于 $\log x\le \log_x M+c=\frac{\log M}{\log x}+c$ 。为了得到矛盾，只需选择 x 大于 M 和 2^{1+c} 两者即可。(b) 假设存在常数 $c>0, a>1$ 使得 $a^{x^c}=O\big(f(x)\big)$ ，这意味着 $\exists N>0, M>0$ 使得 $\forall x>N, a^{x^c}\le Mf(x)$ ，即 $a^{x^c}\le Mx^{\log x}$ 。这意味着 $x^c\log a\le \log M+\log^2 x<\left(\sqrt{\log M}+\log x\right)^2$ ，进而意味着 $x^{\frac{c}{2}}\sqrt{\log a}<\sqrt{\log M}+\log x$ 。由于 $\log x=O\left(x^{\frac{c}{4}}\sqrt{\log a}\right)$ ，当 x 足够大时，存在一个常数 $d>0$ 使得 $x^{\frac{c}{2}}\sqrt{\log a}<x^{\frac{c}{4}}\sqrt{\log a}d$ ，这意味着 $x^{\frac{c}{4}}<d$ 。矛盾。

Example 2.5.5 In cryptography, we want the adversary to have a "minimum probability" of success. To quantify this "minimum probability", we define a negligible function [25]: A function $p(n)\colon \mathbb{N}\to[0,1]$ is negligible, if for all positive-valued polynomial $f(n)$, there exists $N>0$ such that for all $n>N, p(n)<\frac{1}{f(n)}$.

例 2.5.5 在密码学中，我们希望敌手成功的概率为“最小概率”。为了量化这个“最小概率”，我们定义一个可忽略函数[25]:函数 $p(n)\colon \mathbb{N}\to[0,1]$ 是可忽略的，如果对于所有正值多项式 $f(n)$ ，存在 $N>0$ 使得对于所有 $n>N, p(n)<\frac{1}{f(n)}$ 。

Now think of this question: Why can't we modify the definition by switching the order of the quantifiers? That is, why can't we require that there exists $N>0$ such that for all $n>N$, for all positive-valued polynomial $f(n), p(n)<\frac{1}{f(n)}$?

现在思考这个问题:为什么我们不能通过交换量词的顺序来修改定义呢？也就是说，为什么我们不能要求存在 $N > 0$ 使得对于所有 $n > N$ ，对于所有正值多项式 $f(n), p(n) < \frac{1}{f(n)}$ 呢？

Solution: The reason is that, if modified this way, the definition would become too restrictive. There is simply no function that could satisfy the modified requirement. In order to see this, consider an arbitrary function $p(n): \mathbb{N} \to [0,1]$. Below we show that for all $N > 0$, there exists $n > N$ and positive-valued polynomial $f(n)$, such that $p(n) \geq \frac{1}{f(n)}$. In fact, we can choose a constant $c = \frac{2}{p(N+1)}$, and define polynomial $f(n) = c$. Then $f(N+1)p(N+1) = 2 \Rightarrow p(N+1) \geq \frac{1}{f(N+1)}$.

解决方案:原因在于，如果以这种方式修改，定义会变得过于严格。根本不存在能满足修改后要求的函数。为了明白这一点，考虑任意函数 $p(n): \mathbb{N} \to [0,1]$ 。下面我们证明，对于所有 $N > 0$ ，存在 $n > N$ 和正值多项式 $f(n)$ ，使得 $p(n) \geq \frac{1}{f(n)}$ 。实际上，我们可以选择一个常数 $c = \frac{2}{p(N+1)}$ ，并定义多项式 $f(n) = c$ 。那么 $f(N+1)p(N+1) = 2 \Rightarrow p(N+1) \geq \frac{1}{f(N+1)}$ 。

Now we should be comfortable with asymptotic notations, and understand what is an asymptotic solution to a recurrence. The next question is how we can obtain an asymptotic solution. We demonstrate this by solving an example problem.

现在我们应该熟悉渐近记号了，并且理解什么是递推关系的渐近解。接下来的问题是我们如何能得到一个渐近解。我们通过解决一个示例问题来演示这一点。

Example 2.5.6 ([63], Exercise 2.74, slightly modified) Let f, g, h be functions from $\mathbb{Z}^+$ to $\mathbb{Z}^+$, such that $\forall n \geq 3, f(n) + g(n) + h(n) = n$. Solve the recurrence

示例 2.5.6([63]，练习 2.74，略有修改)设 f, g, h 是从 $\mathbb{Z}^+$ 到 $\mathbb{Z}^+$ 的函数，使得 $\forall n \geq 3, f(n) + g(n) + h(n) = n$ 。渐近地求解递推关系

$$T(n) = T\big(f(n)\big) + T\big(g(n)\big) + T\big(h(n)\big) + 1 \ (n \geq 3)$$

asymptotically, where $T(1) = T(2) = 1$.

其中 $T(1) = T(2) = 1$ 。

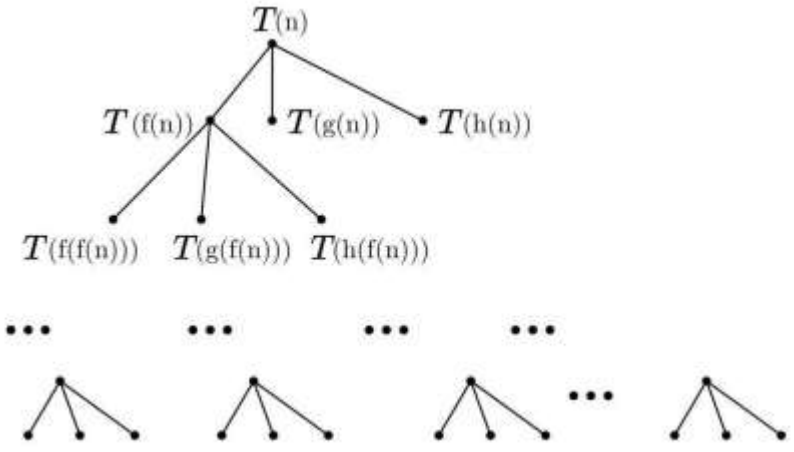

Figure 2.1: A Recursion Tree

图 2.1:递归树

To solve this example problem, we build a rooted tree. [10] which is called the recursion tree of the above recurrence. As illustrated in Figure 2.1, the root of the tree represents $T(n)$. Assuming $n \geq 3$, the root has three children, representing $T(f(n)), T(g(n)), T(h(n))$, respectively. Assuming $f(n) \geq 3$, the vertex $T(f(n))$ also has three children, representing $T\left(f(f(n))\right), T\left(g(f(n))\right), T\left(h(f(n))\right)$, respectively... The leaves of this tree represent $T(k)$ s for $k \leq 2$, and clearly they all equal 1.

为了解决这个示例问题，我们构建一棵有根树。[10] 它被称为上述递推关系的递归树。如图 2.1 所示，树的根表示 $T(n)$。假设 $n \geq 3$，根有三个子节点，分别表示 $T(f(n)), T(g(n)), T(h(n))$。假设 $f(n) \geq 3$，顶点 $T(f(n))$ 也有三个子节点，分别表示 $T\left(f(f(n))\right), T\left(g(f(n))\right)$， $T\left(h(f(n))\right)$ ..... 这棵树的叶子表示 $T(k)$ 对于 $k \leq 2$，并且显然它们都等于 1。

The recursion tree shows us how $T(n)$ is computed. Beginning from the leaves, we move upwards gradually, until reaching the root. Every non-leaf vertex is equal to the sum of its children, plus an extra cost of 1. So we immediately see that $T(n)$ equals the sum of all leaves, plus the sum of all extra costs. The sum of all leaves is equal to the total number of leaves, while the sum of all extra costs is equal to the total number of all non-leaf vertices. Therefore, $T(n)$ is equal to the total number of vertices in the recursion tree.

递归树向我们展示了 $T(n)$ 是如何计算的。从叶子开始，我们逐渐向上移动，直到到达根。每个非叶子顶点等于其所有子节点的和，再加上额外的成本 1。所以我们立刻看到 $T(n)$ 等于所有叶子的和，加上所有额外成本的和。所有叶子的和等于叶子的总数，而所有额外成本的和等于所有非叶子顶点的总数。因此， $T(n)$ 等于递归树中顶点的总数。

What is the total number of vertices in the recursion tree? We have no idea at this moment. But we know the total number of leaves is less than or equal to n. (Do you see why this is the case?) Denote by N_{leaf} the number of leaves, and by N_{non} the number of non-leaf vertices. Counting the total number of children of all non-leaf vertices in two ways, we get that $N_{\text{leaf}} + N_{\text{non}} - 1 = 3N_{\text{non}}$, which implies $N_{\text{non}} = \frac{N_{\text{leaf}} - 1}{2} \leq \frac{n-1}{2}$. So the total number of vertices in the recursion tree, or $T(n)$, is less than or equal to $\frac{3n-1}{2}$. On the other hand, a simple induction allows us to prove that $T(n) \geq \frac{n}{2}$. Consequently, $T(n) = \Theta(n)$.

递归树中的顶点总数是多少？目前我们还不清楚。但我们知道叶子节点的总数小于或等于 n。(你明白为什么是这种情况吗？)用 N_{leaf} 表示叶子节点的数量，用 N_{non} 表示非叶子节点的数量。通过两种方式计算所有非叶子节点的子节点总数，我们得到 $N_{\text{leaf}} + N_{\text{non}} - 1 = 3N_{\text{non}}$，这意味着 $N_{\text{non}} = \frac{N_{\text{leaf}} - 1}{2} \leq \frac{n-1}{2}$。所以递归树中的顶点总数，即 $T(n)$，小于或等于 $\frac{3n-1}{2}$。另一方面，通过简单的归纳法我们可以证明 $T(n) \geq \frac{n}{2}$。因此， $T(n) = \Theta(n)$。

7

Writing a complete solution to Example 2.5.6 is left as homework.

写出例 2.5.6 的完整解答留作作业。

---

[7] [10] We have not defined what is a rooted tree. Fortunately, most of the materials we present here do not require in-depth understanding of rooted trees. If you are interested in the formal definitions of trees, rooted trees, roots, children, leaves, etc., please see Section 5.3.

[10] 我们还没有定义什么是有根树。幸运的是，我们在这里介绍的大多数内容并不需要对有根树有深入的理解。如果你对树、有根树、根、子节点、叶子节点等的形式定义感兴趣，请参阅 5.3 节。

Finally, we present the Master Theorem, which provides asymptotic solutions to a large class of recurrences:

最后，我们给出主定理，它为一大类递归式提供渐近解:

Theorem 2.5.1 (Master Theorem for Divide-and-Conquer Recurrence, Proved by Bentley, Hak-en, and Saxe [11] [27]) Suppose $f(n)$ is a positive-valued function defined on the set of natural numbers. The recurrence relation $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ has the following solution 12

定理 2.5.1(分治递归的主定理，由 Bentley、Hak-en 和 Saxe [11] [27] 证明) 假设 $f(n)$ 是定义在自然数集上的正值函数。递归关系 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ 有以下解 12

- If $f(n) = \Theta(n^c)$ where $c < \log_b a$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
- 如果 $f(n) = \Theta(n^c)$ ，其中 $c < \log_b a$ ，那么 $T(n) = \Theta\left(n^{\log_b a}\right)$ 。
- If $f(n) = \Theta\left(n^{\log_b a}\log^k n\right)(k > -1)$, then $T(n) = \Theta\left(n^{\log_b a}\log^{k+1} n\right)$.
- 如果 $f(n) = \Theta\left(n^{\log_b a}\log^k n\right)(k > -1)$ ，那么 $T(n) = \Theta\left(n^{\log_b a}\log^{k+1} n\right)$ 。
- If $f(n) = \Theta(n^c)$ where $c > \log_b a$, and if $f(n)$ meets the regularity condition, i.e., $af\left(\frac{n}{b}\right) \le df(n)$ for some $d < 1$ and sufficiently large n , then $T(n) = \Theta(n^c)$.
- 如果 $f(n) = \Theta(n^c)$ ，其中 $c > \log_b a$ ，并且如果 $f(n)$ 满足正则条件，即对于某个 $d < 1$ 和足够大的 n 有 $af\left(\frac{n}{b}\right) \le df(n)$ ，那么 $T(n) = \Theta(n^c)$ 。

At a first look, Theorem 2.5.1 might be confusing. What do the conditions after those "if"s mean? How do you get the result? Actually, the intuition behind it is quite simple. Notice that there are only two terms on the right hand side of the recurrence, the first being $aT\left(\frac{n}{b}\right)$ and the second being $f(n)$. Which of them grows faster? This faster growing term dominates the sum.

乍一看，定理 2.5.1 可能令人困惑。那些“如果”后面的条件是什么意思？你是如何得到结果的？实际上，其背后的直觉相当简单。注意到递归式右边只有两项，第一项是 $aT\left(\frac{n}{b}\right)$ ，第二项是 $f(n)$ 。哪一项增长得更快？这个增长更快的项主导了求和。

The three cases considered in Theorem 2.5.1 correspond to three possibilities: that the first term grows significantly faster, that the two term grow at very roughly "similar" rates, and that the second term grows significantly faster, respectively. Imagine we are in the first case, and we are to do an induction on n for proof. Applying the induction hypothesis to $T\left(\frac{n}{b}\right)$, we get that $T(n) \le aM_1\left(\frac{n}{b}\right)^{\log_b a} + M_2 n^c = M_1 n^{\log_b a} + M_2 n^c$. Since $c < \log_b a$, for large n clearly we ignore the second term 13 and get that $T(n) \le M_1 n^{\log_b a}$, or, equivalently, $T(n) = O\left(n^{\log_b a}\right)$. On the

定理 2.5.1 中考虑的三种情况分别对应三种可能性:第一项增长显著更快、两项增长速度大致“相似”、第二项增长显著更快。假设我们处于第一种情况，要对 n 进行归纳证明。将归纳假设应用于 $T\left(\frac{n}{b}\right)$ ，我们得到 $T(n) \le aM_1\left(\frac{n}{b}\right)^{\log_b a} + M_2 n^c = M_1 n^{\log_b a} + M_2 n^c$ 。由于 $c < \log_b a$ ，对于较大的 n ，显然我们可以忽略第二项 13，得到 $T(n) \le M_1 n^{\log_b a}$ ，或者等价地， $T(n) = O\left(n^{\log_b a}\right)$ 。在……方面

hand, since $f(n)$ is positive valued, the solution to $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ grows no slower than the solution to $T(n) = aT\left(\frac{n}{b}\right)$. Given that the latter is $\Theta\left(n^{\log_b a}\right)$, the former has to be $\Theta\left(n^{\log_b a}\right)$, too.

另一方面，由于 $f(n)$ 是正值，$T(n) = aT\left(\frac{n}{b}\right) + f(n)$ 的解增长速度不慢于 $T(n) = aT\left(\frac{n}{b}\right)$ 的解。鉴于后者是 $\Theta\left(n^{\log_b a}\right)$，前者也必定是 $\Theta\left(n^{\log_b a}\right)$。

In the second case, also imagine that we are doing an induction. What we get is that

在第二种情况下，同样假设我们在进行归纳。我们得到的是

$$T(n) \leq aM_1\left(\frac{n}{b}\right)^{\log_b a} \log^{k+1}\frac{n}{b} + M_2 n^{\log_b a}\log^k n$$

---

[8] [11] Jon Bentley (1953-) is an American computer scientist. He received his PhD from UNC Chapel Hill. Besides proving the Master Theorem, he is also famous for proposing the k -d algorithm. Dorothea Blostein (Haken) (1959-) is a Canadian computer scientist, receiving her PhD from UIUC. She is also a daughter of Wolfgang Haken, who obtained a computer-assisted proof of the Four-Color Theorem. She helped her father check his proofs while in high school. James Saxe is an American computer scientist. He got a perfect score on the USAMO when in high school, and became a Putnam Fellow when in college. He received his PhD from CMU. He has a lot of contributions to various fields of computer science, including theory of computing, programming languages, and computer networks.

[11] 乔恩·本特利(Jon Bentley，1953 年 -)是一位美国计算机科学家。他在北卡罗来纳大学教堂山分校获得博士学位。除了证明主定理外，他还因提出 k -d 算法而闻名。多萝西娅·布洛斯坦(哈肯)(Dorothea Blostein (Haken)，1959 年 -)是一位加拿大计算机科学家，在伊利诺伊大学厄巴纳 - 香槟分校获得博士学位。她也是沃尔夫冈·哈肯(Wolfgang Haken)的女儿，沃尔夫冈·哈肯获得了四色定理的计算机辅助证明。她在高中时帮助父亲检查证明。詹姆斯·萨克塞(James Saxe)是一位美国计算机科学家。他高中时在美国数学奥林匹克竞赛中获得满分，大学时成为普特南学者。他在卡内基梅隆大学获得博士学位。他在计算机科学的各个领域都有很多贡献，包括计算理论、编程语言和计算机网络。

The Master Theorem was proved at CMU, where Bentley was a professor. Both Haken and Saxe were graduate students advised by Bentley. Haken moved back to UIUC after finishing her Master's degree, but Saxe remained there, until receiving his PhD. [88]

主定理是在卡内基梅隆大学证明的，本特利当时是该校教授。哈肯和萨克塞都是本特利指导的研究生。哈肯在完成硕士学位后回到了伊利诺伊大学厄巴纳 - 香槟分校，但萨克塞留在了那里，直到获得博士学位。[88]

[12] Here the constants a, b should be positive integers. However, for more general cases in which a, b are positive real numbers, Akra and Bazzi [1] have proved a similar result, which is only slightly weaker. A probably more important extension by Yap 83 covers the gap between cases.

[12] 这里的常数 a, b 应该是正整数。然而，对于 a, b 是正实数的更一般情况，阿克拉和巴齐 [1] 证明了一个类似的结果，只是稍微弱一些。叶普(Yap)[83] 可能更重要的一个扩展弥补了这些情况之间的差距。

[13] This is just an intuitive explanation, not a rigorous proof. When writing a proof, you can't do it this way.

[13] 这只是一个直观的解释，不是严格的证明。写证明时不能这样做。

$$= M_1 n^{\log_b a}(\log n - \log b)^{k+1} + M_2 n^{\log_b a}\log^k n$$

$$\leq M_1 n^{\log_b a}\log^{k+1} n + M_2 n^{\log_b a}\log^k n$$

Again, we are ignoring the second term because it is much smaller than the first term for large n. So we get that $T(n) \leq M_1 n^{\log_b a}\log^{k+1} n$, or, equivalently, $T(n) = O\left(n^{\log_b a}\log^{k+1} n\right)$. Note that the same approach would not produce $T(n) = O\left(n^{\log_b a}\log^k n\right)$ for us, because we would end up deriving $T(n) \leq (M_1 + M_2)n^{\log_b a}\log^k n$ from the induction hypothesis $T\left(\frac{n}{b}\right) \leq M_1\left(\frac{n}{b}\right)^{\log_b a}\log^k\left(\frac{n}{b}\right)$, which does not satisfy the induction step.

同样，我们忽略第二项，因为对于较大的 n，它比第一项小得多。所以我们得到 $T(n) \leq M_1 n^{\log_b a}\log^{k+1} n$，或者等价地，$T(n) = O\left(n^{\log_b a}\log^{k+1} n\right)$。注意，同样的方法对我们来说不会得出 $T(n) = O\left(n^{\log_b a}\log^k n\right)$，因为我们最终会从归纳假设 $T\left(\frac{n}{b}\right) \leq M_1\left(\frac{n}{b}\right)^{\log_b a}\log^k\left(\frac{n}{b}\right)$ 推导出 $T(n) \leq (M_1 + M_2)n^{\log_b a}\log^k n$，而这并不满足归纳步骤。

For a similar reason, if we do an induction in the third case, we get that

出于类似的原因，如果我们在第三种情况下进行归纳，我们会得到

$$T(n) \leq aM_1\left(\frac{n}{b}\right)^c + M_2 n^c = \left(\frac{aM_1}{b^c} + M_2\right)n^c,$$

which would not allow us to complete the induction. Therefore, we need an additional condition, the regularity condition. [14] In this case, roughly speaking,

这将使我们无法完成归纳。因此，我们需要一个额外的条件，即正则性条件。[14] 在这种情况下，粗略地说，

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) = a^2 T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n) = \cdots$$

$$\approx f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots \leq f(n) + df(n) + d^2 f(n) + \cdots = \frac{f(n)}{1-d}.$$

A detailed proof of Theorem 2.5.1 can be found, e.g., in [11.

定理 2.5.1 的详细证明可以在例如[11]中找到。

Example 2.5.7 Find the asymptotic solutions to the following recurrences.

例 2.5.7 求下列递推关系的渐近解。

1. $T(n) = 6T\left(\frac{n}{2}\right) + n^4$.

[9] [14] In fact, the regularity condition alone implies the third case condition $c > \log_b a$. Specifically, suppose "sufficiently large" n means $n \geq n_0$. Then we rewrite the regularity condition as $f(n) \geq \frac{a}{d}f\left(\frac{n}{b}\right)$, which implies that

[14] 事实上，仅正则性条件就意味着第三种情况条件 $c > \log_b a$。具体来说，假设“足够大” n 意味着 $n \geq n_0$。那么我们将正则性条件重写为 $f(n) \geq \frac{a}{d}f\left(\frac{n}{b}\right)$，这意味着

2. $T(n) = 10T\left(\frac{n}{2}\right) + n^2$.

3. $T(n) = 8T\left(\frac{n}{2}\right) + n^3$.

4. $T(n) = 10T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$.

Solution: We skip the first three and solve only the last recurrence. It is easy to see that $10T\left(\frac{n}{2}\right) + n < T(n) < 10T\left(\frac{n}{2}\right) + n^2$ for sufficiently large n . Using Theorem 2.5.1, we get that the solutions to both $T(n) = 10T\left(\frac{n}{2}\right) + n$ and $T(n) = 10T\left(\frac{n}{2}\right) + n^2$ are $\Theta\left(n^{\log_2 10}\right)$ asymptotically, and thus our solution is also $\Theta\left(n^{\log_2 10}\right)$.

解:我们跳过前三个，只求解最后一个递推关系。很容易看出，对于足够大的 n ，有 $10T\left(\frac{n}{2}\right) + n < T(n) < 10T\left(\frac{n}{2}\right) + n^2$ 。使用定理 2.5.1，我们得到 $T(n) = 10T\left(\frac{n}{2}\right) + n$ 和 $T(n) = 10T\left(\frac{n}{2}\right) + n^2$ 的解在渐近意义上都是 $\Theta\left(n^{\log_2 10}\right)$ ，因此我们的解也是 $\Theta\left(n^{\log_2 10}\right)$ 。

Problem Set 5

习题集 5

Problem 79 If $p(L)$ and $q(L)$ are polynomials in L , show that $p(L)q(L) = q(L)p(L)$.

问题 79 如果 $p(L)$ 和 $q(L)$ 是关于 L 的多项式，证明 $p(L)q(L) = q(L)p(L)$ 。

Problem 80 Write a complete solution to Example 2.5.6.

问题 80 写出例 2.5.6 的完整解。

Problem 81 [Difficulty Estimate $= 0.6$] Suppose that $p(L)$ is a polynomial in L , and that $p(L)$ annihilates function $T(n)$. Prove that $T(n) = \sum_{i=1}^{k} f_i(n) r_i^n$, where k is a constant, $f_1(n), \ldots, f_k(n)$ are polynomials in n , and $r_1, \ldots, r_k$ are all constants.

问题 81 [难度估计 $= 0.6$] 假设 $p(L)$ 是关于 L 的多项式，且 $p(L)$ 使函数 $T(n)$ 归零。证明 $T(n) = \sum_{i=1}^{k} f_i(n) r_i^n$ ，其中 k 是常数， $f_1(n), \ldots, f_k(n)$ 是关于 n 的多项式，且 $r_1, \ldots, r_k$ 均为常数。

Problem 82 [Difficulty Estimate $= 0.9$] Define a difference operator Δ: $\Delta = L - 1$, i.e., $\Delta T(n) = T(n+1) - T(n)$. Define a sum operator $\sum$: $\sum T = S + c$ where $\Delta S = T$ and c is a constant. Prove that for all functions f, g defined on positive integers, $\sum f \Delta g = fg - \sum Lg \Delta f$.

$$f(n) \geq \left(\frac{a}{d}\right)^{\log_b(n/n_0)} \min_{1 \leq x \leq n_0} f(x) = \left(\frac{n}{n_0}\right)^{\log_b(a/d)} \min_{1 \leq x \leq n_0} f(x) = \Theta\left(n^{\log_b a + \log_b \frac{1}{d}}\right).$$

Therefore, writing the third case condition $c > \log_b a$ here is actually redundant. We still write it, just to make the statement of Master Theorem more readable.

因此，在这里写出第三种情况条件 $c > \log_b a$ 实际上是多余的。我们仍然写它，只是为了使主定理的陈述更具可读性。

问题 82 [难度估计 = 0.9] 定义一个差分算子 Δ: $\Delta = L - 1$，即 $\Delta T(n) = T(n+1) - T(n)$。定义一个求和算子 $\sum$: $\sum T = S + c$，其中 $\Delta S = T$ 且 c 是常数。证明对于所有定义在正整数上的函数 f, g，有 $\sum f \Delta g = fg - \sum Lg \Delta f$。

Problem 83 [Difficulty Estimate=1.4] The first case of Theorem 2.5.1 uses the following condition: $f(n) = \Theta(n^c)$ where $c < \log_b a$. In the literature, you may find an alternative condition for this case: $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$. Prove using these two conditions in the theorem are equivalent.

问题 83 [难度估计=1.4] 定理 2.5.1 的第一种情况使用以下条件: $f(n) = \Theta(n^c)$，其中 $c < \log_b a$。在文献中，你可能会找到这种情况的另一个条件:对于某个 $\epsilon > 0$，有 $f(n) = O\left(n^{\log_b a - \epsilon}\right)$。证明定理中的这两个条件是等价的。

Problem 84 [Difficulty Estimate = 2.4] Find a recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ such that $f(n) = \Theta(n^c)$ where $c > \log_b a$, but $f(n)$ does not meet the regularity condition.

问题 84 [难度估计 = 2.4] 找到一个递推关系 $T(n) = aT\left(\frac{n}{b}\right) + f(n)$，使得 $f(n) = \Theta(n^c)$，其中 $c > \log_b a$，但 $f(n)$ 不满足正则性条件。

Problem 85 [Difficulty Estimate = 2.3] Suppose $T(\ \ )$ is a function defined on $\{0,1,2,\dots\}$, whose value is always a positive real number. It satisfies the recurrence $T(n) = \frac{\left(T(n-1)\right)^2 - 1}{T(n-2)}$ for all $n \geq 2$. Given that $T(0) > 1, |T(0) - T(1)| > 1$, show that $\log T(n) = O(n)$.

问题 85 [难度估计 = 2.3] 假设 $T(\ \ )$ 是定义在 $\{0,1,2,\dots\}$ 上的函数，其值始终为正实数。对于所有 $n \geq 2$，它满足递推关系 $T(n) = \frac{\left(T(n-1)\right)^2 - 1}{T(n-2)}$。已知 $T(0) > 1, |T(0) - T(1)| > 1$，证明 $\log T(n) = O(n)$。

Problem 86 [Difficulty Estimate = 2.8] For every natural number n, define $S(n) = n - m^2$, where m is the largest integer such that $n - m^2 \geq 0$. Let $T(n)$ be a recursively defined function: $T(n+1) = T(n) + S\left(T(n)\right)$ for every natural number n. Find a (reasonably good) asymptotic upper bound for $T(n)$.

问题 86 [难度估计 = 2.8] 对于每个自然数 n，定义 $S(n) = n - m^2$，其中 m 是使得 $n - m^2 \geq 0$ 成立的最大整数。设 $T(n)$ 是一个递归定义的函数:对于每个自然数 n，有 $T(n+1) = T(n) + S\left(T(n)\right)$。找到 $T(n)$ 的一个(相当好的)渐近上界。

Problem 87 (Putnam 1998-A4, modified) [Difficulty Estimate=1.1] Define a new binary operation $\sharp$ on positive integers: $a\sharp b$ is the positive integer formed by writing a followed by b in decimal. For example, $1224\sharp 25900 = 122425900, 33333\sharp 33 = 3333333, 7788\sharp 123 = 7788123$.

问题 87(1998 年普特南数学竞赛 A4 题，改编)[难度估计 = 1.1] 定义一种正整数上的新二元运算 $\sharp$: $a\sharp b$ 是通过将 a 后面接上 b 以十进制形式写出而形成的正整数。例如， $1224\sharp 25900 = 122425900, 33333\sharp 33 = 3333333, 7788\sharp 123 = 7788123$.

Now consider a function $A(\ \ )$ defined on $\{0,1,2,\dots\}$. Let $A(0) = 0, A(1) = 1$. For any $n \geq 2$, we always have $A(n) = A(n-1)\sharp A(n-2)$. For what values of n is $A(n)$ a multiple of 11 ? Prove your answer.

现在考虑一个定义在 $\{0,1,2,\dots\}$ 上的函数 $A(\ \ )$。设 $A(0) = 0, A(1) = 1$。对于任意 $n \geq 2$，我们总是有 $A(n) = A(n-1)\sharp A(n-2)$。 n 取何值时， $A(n)$ 是 11 的倍数？证明你的答案。

Problem 88 (UIUC Theory Qual-2003, Problem 1 (a-b)) [Difficulty Estimate=2.4] The main purpose of studying asymptotic complexity is to analyze algorithm complexity. In this problem, we need to analyze the complexity of a special sorting algorithm.

问题 88(伊利诺伊大学厄巴纳 - 香槟分校理论资格考试 - 2003 年，问题 1(a - b))[难度估计 = 2.4] 研究渐近复杂度的主要目的是分析算法复杂度。在这个问题中，我们需要分析一种特殊排序算法的复杂度。

Suppose that you are given an array $A[1..n]$ of length n to sort. However, you can't access the array $A[1..n]$, except through a particular algorithm S . While you are allowed to call S as many times as necessary, you can't see what's happening "inside" the algorithm S. In other words, you can only use it as a blackbox.

假设给你一个长度为 n 的数组 $A[1..n]$ 进行排序。然而，你无法直接访问数组 $A[1..n]$ ，只能通过特定算法 S 来操作。虽然你可以根据需要多次调用 S ，但你无法看到算法 S “内部” 的情况。换句话说，你只能将其当作一个黑盒来使用。

The algorithm S also does sorting, but in each execution it only sorts $\lceil\sqrt{n}\rceil$ elements, and its execution time is $\lceil\sqrt{n}\rceil$. Specifically, when you make a call $S(x)\left(0 \le x \le n - \lceil\sqrt{n}\rceil\right)$, the algorithm S sorts the $\lceil\sqrt{n}\rceil$ elements from $A[x+1]$ to $A\left[x + \lceil\sqrt{n}\rceil\right]$. Design an algorithm to sort the entire array, and prove that the worst-case time complexity of your algorithm is asymptotically optimal.

算法 S 也能进行排序，但每次执行时它只能对 $\lceil\sqrt{n}\rceil$ 个元素进行排序，其执行时间为 $\lceil\sqrt{n}\rceil$ 。具体而言，当你调用 $S(x)\left(0 \le x \le n - \lceil\sqrt{n}\rceil\right)$ 时，算法 S 会对从 $A[x+1]$ 到 $A\left[x + \lceil\sqrt{n}\rceil\right]$ 的 $\lceil\sqrt{n}\rceil$ 个元素进行排序。设计一种算法对整个数组进行排序，并证明你的算法在最坏情况下的时间复杂度是渐近最优的。

Problem 89 We say an operator A is bounded if for every function T defined on the set of positive integers, as long as

问题 89 如果对于定义在正整数集上的每个函数 T ，只要满足

$$\sum_{n=1}^{\infty}\left(T(n)\right)^2 = 1$$

we must have that

我们就一定有

$$\sum_{n=1}^{\infty}\left(AT(n)\right)^2$$

is bounded. We say an operator A is uniformly continuous if for every $\epsilon > 0$, there exists $\delta > 0$ such that for any functions T, S defined on the set of positive integers that satisfy

则称算子 A 是有界的。如果对于每个 $\epsilon > 0$ ，都存在 $\delta > 0$ ，使得对于定义在正整数集上的任意满足

$$\sum_{n=1}^{\infty}\left(T(n) - S(n)\right)^2 < \delta$$

we must have that

的函数 <b3></b3> ，我们一定有

$$\sum_{n=1}^{\infty}\left(AT(n) - AS(n)\right)^2 < \epsilon$$

Under what condition(s) are the two concepts defined above equivalent? In other words, under what conditions(s) is an operator bounded if and only if it is uniformly continuous? Prove your answer.

那么在什么条件下上述两个概念是等价的？换句话说，在什么条件下一个算子当且仅当它是一致连续时才是有界的？证明你的答案。

Problem 90 Solve the recurrence

问题 90 求解递归式

$$S(n) = 12S(n-1) - 32S(n-2),$$

where

其中

$$S(2) = 4(S(1) + 16).$$

While you might be able to find a precise solution, you are expected to provide an asymptotic solution only.

虽然你可能能够找到精确解，但只需要给出渐近解。

Programming Problems

编程问题

Problem 91 The Master Theorem does not cover the situation of $c > \log_b a$ with regularity condition unsatisfied. What kind of asymptotic solution would we have in this situation? Please experimentally explore it. Specifically, you need to design an experiment, carry out the experiment, and write a report to tell us your findings.

问题 91 主定理不涵盖正则条件不满足时的 $c > \log_b a$ 情况。在这种情况下我们会得到什么样的渐近解？请通过实验进行探索。具体来说，你需要设计一个实验，进行实验，并撰写一份报告来告知我们你的发现。

Writing Problems

写作问题

Problem 92 Annihilators are used well beyond algorithm analysis. Please write a brief essay to survey its usage in other areas.

问题 92 零化子的用途远不止于算法分析。请写一篇短文概述它在其他领域的用途。

4.5 Continued Fractions

4.5 连分数

We begin this section with a problem from elementary school:

我们从一个小学问题开始本节内容:

Example 4.5.1 Let A, B, C be positive integers. Suppose

例 4.5.1 设 A, B, C 为正整数。假设

$$\frac{24}{5} = A + \frac{1}{B + \frac{1}{C}}$$

Find A, B, C .

求 A, B, C 。

There is no doubt that $A = 4, B = 1, C = 4$. A fraction written this way is called a continued fraction. More formally, we have the following definition.

毫无疑问， $A = 4, B = 1, C = 4$ 。以这种方式书写的分数称为连分数。更正式地，我们有以下定义。

Definition 4.5.1 A finite continued fraction is a function defined on a sequence of real-valued variables $a_0, a_1, \ldots, a_N$:

定义 4.5.1 有限连分数是定义在实值变量序列 $a_0, a_1, \ldots, a_N$ 上的函数:

$$[a_0, a_1, \ldots, a_N] = a_0 + \frac{1}{} \ldots$$

$$a_1 + \cdots + \frac{1}{a_N}$$

Intuitively, we can see the sequence $[a_0], [a_0, a_1], \ldots [a_0, a_1, \ldots, a_N]$ as a gradual approach to the value of the finite continued fraction. Hence, each term of the sequence (the n th of which is $[a_0, a_1, \ldots, a_n]$) is called a convergent.

直观地，我们可以将序列 $[a_0], [a_0, a_1], \ldots [a_0, a_1, \ldots, a_N]$ 视为对有限连分数值的逐步逼近。因此，序列的每一项(其中第 n 项是 $[a_0, a_1, \ldots, a_n]$))称为一个渐进分数。

But is the sequence $[a_0], [a_0, a_1], \ldots [a_0, a_1, \ldots, a_N]$ really a gradual approach to the value of the finite continued fraction? In other words, does $[a_0, a_1, \ldots, a_n]$ get "closer" to $[a_0, a_1, \ldots, a_N]$ when n grows? In general, this may NOT be true. Consider, for example, the finite continued fraction $\left[1, \frac{1}{2}, -\frac{3}{2}, -1\right]$. 15 It is easy to calculate:

但是序列 $[a_0], [a_0, a_1], \ldots [a_0, a_1, \ldots, a_N]$ 真的是对有限连分数值的逐步逼近吗？换句话说，当 n 增大时，$[a_0, a_1, \ldots, a_n]$ 会更接近 $[a_0, a_1, \ldots, a_N]$ 吗？一般来说，这可能不成立。例如，考虑有限连分数 $\left[1, \frac{1}{2}, -\frac{3}{2}, -1\right]$ 。15 很容易计算:

$$[1] = 1; \left[1, \frac{1}{2}\right] = 3; \left[1, \frac{1}{2}, -\frac{3}{2}\right] = -5; \left[1, \frac{1}{2}, -\frac{3}{2}, -1\right] = 11 .$$

Therefore, the third convergent $\left[1, \frac{1}{2}, -\frac{3}{2}\right]$ is farther from the value of the finite continued fraction than the first two convergents.

因此，第三个渐进分数 $\left[1, \frac{1}{2}, -\frac{3}{2}\right]$ 比前两个渐进分数离有限连分数的值更远。

Nevertheless, if all a_i s are positive, we can actually guarantee convergents are getting "closer" to the value of the finite continued fraction, in the following sense.

然而，如果所有的 a_i 都是正数，我们实际上可以保证渐进分数在以下意义上越来越接近有限连分数的值。

Theorem 4.5.1 Suppose for all $i(0 \leq i \leq N), a_i > 0$. Let c_i be the ith convergent of $[a_0, a_1, \ldots, a_N]$, i.e., $c_i = [a_0, a_1, \ldots, a_i]$. Then we have $c_0 < c_2 < \cdots < c_N$ and $c_1 > c_3 > \cdots > c_N$. In other words, the even numbered convergents approach the value of the finite continued fraction from the left and the odd numbered convergents approach it from the right.

定理 4.5.1 假设对所有的 $i(0 \leq i \leq N), a_i > 0$ 。设 c_i 是 $[a_0, a_1, \ldots, a_N]$ 的第 i 个渐进分数，即 $c_i = [a_0, a_1, \ldots, a_i]$ 。那么我们有 $c_0 < c_2 < \cdots < c_N$ 和 $c_1 > c_3 > \cdots > c_N$ 。换句话说，偶数编号的渐进分数从左边逼近有限连分数的值，奇数编号的渐进分数从右边逼近它。

[10]

Proof: By induction on i , we can easily show that for all odd i and all $a_i < a_i'$, $[a_0, a_1, \ldots, a_i] > [a_0, a_1, \ldots, a_i']$; for all even i and all $a_i < a_i'$, $[a_0, a_1, \ldots, a_i] < [a_0, a_1, \ldots, a_i']$. (See Problem 162) Hence, for any odd $k \geq 1$,

证明:通过对 i 进行归纳，我们可以轻松证明，对于所有奇数 i 和所有 $a_i < a_i'$, $[a_0, a_1, \ldots, a_i] > [a_0, a_1, \ldots, a_i']$ ；对于所有偶数 i 和所有 $a_i < a_i'$, $[a_0, a_1, \ldots, a_i] < [a_0, a_1, \ldots, a_i']$ 。(见问题 162)因此，对于任何奇数 $k \geq 1$ ，

$$c_{k+2} = [a_0, a_1, \ldots, a_{k+2}] = \left[a_0, a_1, \ldots, a_{k-1}, a_k + \frac{1}{a_{k+1} + \frac{1}{a_{k+2}}}\right] < [a_0, a_1, \ldots, a_{k-1}, a_k] = c_k.$$

Similarly, for any even $k \geq 0$,

类似地，对于任何偶数 $k \geq 0$ ，

$$c_{k+2} = [a_0, a_1, \ldots, a_{k+2}] = \left[a_0, a_1, \ldots, a_{k-1}, a_k + \frac{1}{a_{k+1} + \frac{1}{a_{k+2}}}\right] > [a_0, a_1, \ldots, a_{k-1}, a_k] = c_k.$$

Now we see that odd numbered convergents are decreasing and even numbered convergents are increasing. The only thing remaining is to show the relationship between convergents and c_N , the value of the finite continued fraction.

---

[10] [15] How do we come up with this example? See Problem 161

[15] 我们是如何想出这个例子的？见问题 161

现在我们看到奇数项的渐进分数是递减的，偶数项的渐进分数是递增的。剩下唯一要证明的就是渐进分数与 c_N (有限连分数的值)之间的关系。

If N is odd, then we have shown $c_1 > c_3 > \cdots > c_N$. For any even $k \geq 0$,

如果 N 是奇数，那么我们已经证明了 $c_1 > c_3 > \cdots > c_N$ 。对于任何偶数 $k \geq 0$ ，

$$c_k = [a_0, a_1, \ldots, a_k] < \left[a_0, a_1, \ldots, a_{k-1}, a_k + \frac{1}{[a_{k+1}, \ldots, a_N]}\right] = c_N.$$

If N is even, the proof is similar.

如果 N 是偶数，证明过程类似。

If all a_i s are positive integers, then we say the finite continued fraction $[a_0, a_1, \ldots, a_N]$ is a simple finite continued fraction. When we study continued fractions, in most cases we focus on simple continued fractions only.

如果所有的 a_i s 都是正整数，那么我们称有限连分数 $[a_0, a_1, \ldots, a_N]$ 为简单有限连分数。当我们研究连分数时，在大多数情况下我们只关注简单连分数。

Obviously each simple finite continued fraction is equal to a positive rational number. Next, we show the converse is true as well.

显然，每个简单有限连分数都等于一个正有理数。接下来，我们证明其逆命题也成立。

Theorem 4.5.2 For all positive rational number $\frac{p}{q} \geq 1$ where p and q are positive integers relative prime to each other, there exists a simple finite continued fraction

定理 4.5.2 对于所有正有理数 $\frac{p}{q} \geq 1$ ，其中 p 和 q 是互质的正整数，存在一个简单有限连分数

$$[a_0, a_1, \ldots, a_N] = \frac{p}{q}.$$

Proof: We apply Euclidean algorithm to p and q : Let $a_0 = \left\lfloor \frac{p}{q} \right\rfloor$, and $a_1' = p - a_0 q$. (Note that $\frac{p}{q} = a_0 + \frac{a_1'}{q}$.) For $i = 1$, if $a_i' = 0$, then we are done; otherwise, let $a_i = \left\lfloor \frac{q}{a_i'} \right\rfloor$, and $a_{i+1}' = q - a_i a_i'$. (Note that $\frac{p}{q} = a_0 + \frac{a_1'}{q} = a_0 + \frac{1}{\frac{q}{a_1'}} = a_0 + \frac{1}{a_1 + \frac{a_2'}{a_1'}} = \left[a_0, a_1 + \frac{a_2'}{a_1'}\right]$.)

证明:我们对 p 和 q 应用欧几里得算法:设 $a_0 = \left\lfloor \frac{p}{q} \right\rfloor$ ，以及 $a_1' = p - a_0 q$ 。(注意 $\frac{p}{q} = a_0 + \frac{a_1'}{q}$ 。)对于 $i = 1$ ，如果 $a_i' = 0$ ，那么我们就完成了；否则，设 $a_i = \left\lfloor \frac{q}{a_i'} \right\rfloor$ ，以及 $a_{i+1}' = q - a_i a_i'$ 。(注意 $\frac{p}{q} = a_0 + \frac{a_1'}{q} = a_0 + \frac{1}{\frac{q}{a_1'}} = a_0 + \frac{1}{a_1 + \frac{a_2'}{a_1'}} = \left[a_0, a_1 + \frac{a_2'}{a_1'}\right]$ 。)

For each $i \geq 2$, if $a_i' = 0$, then we are done; otherwise, let $a_i = \left\lfloor \frac{a_{i-1}'}{a_i'} \right\rfloor$, and $a_{i+1}' = a_{i-1}' - a_i a_i'$. (Note that $\frac{p}{q} = \left[a_0, a_1, \ldots, a_{i-2}, a_{i-1} + \frac{a_i'}{a_{i-1}'}\right] = \left[a_0, a_1, \ldots, a_{i-1}, \frac{a_{i-1}'}{a_i'}\right] = \left[a_0, a_1, \ldots, a_{i-1}, a_i + \frac{a_{i+1}'}{a_i'}\right]$.)

对于每个 $i \geq 2$，如果 $a_i' = 0$，那么我们就完成了；否则，令 $a_i = \left\lfloor \frac{a_{i-1}'}{a_i'} \right\rfloor$，且 $a_{i+1}' = a_{i-1}' - a_i a_i'$。(注意 $\frac{p}{q} = \left[a_0, a_1, \ldots, a_{i-2}, a_{i-1} + \frac{a_i'}{a_{i-1}'}\right] = \left[a_0, a_1, \ldots, a_{i-1}, \frac{a_{i-1}'}{a_i'}\right] = \left[a_0, a_1, \ldots, a_{i-1}, a_i + \frac{a_{i+1}'}{a_i'}\right]$。)

This process must terminate, because $a_{i+1}' < a_{i-1}'$. When it terminates, we get the simple finite continued fraction we need.

这个过程必定会终止，因为 $a_{i+1}' < a_{i-1}'$。当它终止时，我们就得到了我们所需的简单有限连分数。

Given that each positive rational number greater than 1 can be written as a simple finite continued fraction, naturally we ask whether there is only one such continued fraction. The answer is no (and "yes"- see below).

鉴于每个大于 1 的正有理数都可以写成一个简单有限连分数，很自然地我们会问是否只有一个这样的连分数。答案是否定的(还有“肯定的”情况——见下文)。

Theorem 4.5.3 If $x = [a_0, a_1, \ldots, a_N]$ is a simple finite continued fraction where $N \geq 1$ is odd, then there exist an even number M and a simple finite continued fraction $[b_0, b_1, \ldots, b_M]$ such that $x = [b_0, b_1, \ldots, b_M]$. If $x = [a_0, a_1, \ldots, a_N] > 1$ is a simple finite continued fraction where $N \geq 0$ is even, then there exist an odd number M and a simple finite continued fraction $[b_0, b_1, \ldots, b_M]$ such that $x = [b_0, b_1, \ldots, b_M]$.

定理 4.5.3 如果 $x = [a_0, a_1, \ldots, a_N]$ 是一个简单有限连分数，其中 $N \geq 1$ 是奇数，那么存在一个偶数 M 和一个简单有限连分数 $[b_0, b_1, \ldots, b_M]$ 使得 $x = [b_0, b_1, \ldots, b_M]$。如果 $x = [a_0, a_1, \ldots, a_N] > 1$ 是一个简单有限连分数，其中 $N \geq 0$ 是偶数，那么存在一个奇数 M 和一个简单有限连分数 $[b_0, b_1, \ldots, b_M]$ 使得 $x = [b_0, b_1, \ldots, b_M]$。

Proof: We provide a proof for the first part only, because the second part can be proved similarly.

证明:我们仅对第一部分进行证明，因为第二部分的证明方式类似。

If $a_N = 1$, then we have $x = [a_0, a_1, \ldots, a_N] = [a_0, a_1, \ldots, a_{N-1}, 1] = [a_0, a_1, \ldots, a_{N-1} + 1]$.

如果 $a_N = 1$，那么我们有 $x = [a_0, a_1, \ldots, a_N] = [a_0, a_1, \ldots, a_{N-1}, 1] = [a_0, a_1, \ldots, a_{N-1} + 1]$。

If $a_N > 1$, then, defining $b_N = a_N - 1$, we have $x = [a_0, a_1, \ldots, a_N] = [a_0, a_1, \ldots, a_{N-1}, b_N + 1] = [a_0, a_1, \ldots, a_{N-1}, b_N, 1]$.

如果 $a_N > 1$，那么，定义 $b_N = a_N - 1$，我们有 $x = [a_0, a_1, \ldots, a_N] = [a_0, a_1, \ldots, a_{N-1}, b_N + 1] = [a_0, a_1, \ldots, a_{N-1}, b_N, 1]$。

If you take a closer look at the above proof, you see that we are converting a continued fraction with $a_N = 1$ to another one with $a_N > 1$, or the other way around. In fact, if we restrict our attention to those simple continued fractions with $a_N > 1$, we can guarantee that every one is unique.

如果你仔细研究上述证明，你会发现我们正在将一个具有 $a_N = 1$ 的连分数转换为另一个具有 $a_N > 1$ 的连分数，反之亦然。事实上，如果我们将注意力限制在那些具有 $a_N > 1$ 的简单连分数上，我们可以保证每个连分数都是唯一的。

Theorem 4.5.4 Suppose $[a_0, a_1, \ldots, a_N]$ and $[b_0, b_1, \ldots, b_M]$ are simple finite continued fractions with $a_N > 1, b_M > 1$. If $[a_0, a_1, \ldots, a_N] = [b_0, b_1, \ldots, b_M]$, then $N = M$ and for each $i (0 \leq i \leq N), a_i = b_i$.

定理 4.5.4 假设 $[a_0, a_1, \ldots, a_N]$ 和 $[b_0, b_1, \ldots, b_M]$ 是具有 $a_N > 1, b_M > 1$ 的简单有限连分数。如果 $[a_0, a_1, \ldots, a_N] = [b_0, b_1, \ldots, b_M]$，那么 $N = M$ 并且对于每个 $i (0 \leq i \leq N), a_i = b_i$。

The proof is not hard and can be found, e.g., in [42].

证明并不困难，例如可以在[42]中找到。

If we consider only those with $a_N = 1$, we can get a similar result.

如果我们只考虑那些具有 $a_N = 1$ 的连分数，我们可以得到类似的结果。

Example 4.5.2 Suppose $[a_0, a_1, \dots, a_N]$ and $[b_0, b_1, \dots, b_M]$ are simple finite continued fractions with $a_N = b_M = 1$. If $[a_0, a_1, \dots, a_N] = [b_0, b_1, \dots, b_M]$, prove that $N = M$ and that for each i $(0 \le i \le N)$, $a_i = b_i$.

例 4.5.2 假设 $[a_0, a_1, \dots, a_N]$ 和 $[b_0, b_1, \dots, b_M]$ 是具有 $a_N = b_M = 1$ 的简单有限连分数。如果 $[a_0, a_1, \dots, a_N] = [b_0, b_1, \dots, b_M]$，证明 $N = M$ 并且对于每个 i $(0 \le i \le N)$, $a_i = b_i$。

Solution: Since $a_N = b_M = 1$, it is easy to see that $[a_0, a_1, \dots, a_{N-2}, a_{N-1} + 1] = [a_0, a_1, \dots, a_N] = [b_0, b_1, \dots, b_M] = [b_0, b_1, \dots, b_{M-2}, b_{M-1} + 1]$. Applying Theorem 4.5.4 to this identity, we get exactly what we want.

解:由于 $a_N = b_M = 1$，很容易看出 $[a_0, a_1, \dots, a_{N-2}, a_{N-1} + 1] = [a_0, a_1, \dots, a_N] = [b_0, b_1, \dots, b_M] = [b_0, b_1, \dots, b_{M-2}, b_{M-1} + 1]$。将定理 4.5.4 应用于此恒等式，我们恰好得到了我们想要的结果。

Definition 4.5.2 An infinite continued fraction is the limit of the corresponding finite continued fractions, if it exists:

定义 4.5.2 一个无限连分数是相应有限连分数的极限(如果它存在的话):

$$[a_0, a_1, \dots] = \lim_{N\to\infty} [a_0, a_1, \dots, a_N]$$

It is called simple if all $a_i s$ are positive integers.

如果所有 $a_i s$ 都是正整数，则称其为简单连分数。

Theorem 4.5.5 Every infinite simple continued fraction $[a_0, a_1, \dots]$ converges, i.e., the limit $\lim_{N\to\infty} [a_0, a_1, \dots, a_N]$ exists.

定理 4.5.5 每个无限简单连分数 $[a_0, a_1, \dots]$ 都收敛，即极限 $\lim_{N\to\infty} [a_0, a_1, \dots, a_N]$ 存在。

Proof: Since the odd numbered convergents decrease [16] and they have a_0 as their lower bound, the limit

证明:由于奇数项收敛分数递减 [16] 并且它们以 a_0 为下界，所以极限

$\lim_{N\to\infty} [a_0, a_1, \dots, a_{2N+1}]$ exists. Similarly, we know that the limit $\lim_{N\to\infty} [a_0, a_1, \dots, a_{2N}]$ also exists. Applying the result of Problem 161, we get that

$\lim_{N\to\infty} [a_0, a_1, \dots, a_{2N+1}]$ 存在。类似地，我们知道极限 $\lim_{N\to\infty} [a_0, a_1, \dots, a_{2N}]$ 也存在。应用问题 161 的结果，我们得到

$$\lim_{N\to\infty} [a_0, a_1, \dots, a_{2N+1}] - \lim_{N\to\infty} [a_0, a_1, \dots, a_{2N}]$$

$$= \lim_{N\to\infty} ([a_0, a_1, \dots, a_{2N+1}] - [a_0, a_1, \dots, a_{2N}])$$

$$= \lim_{N\to\infty} \left(\frac{p_{2N+1}}{q_{2N+1}} - \frac{p_{2N}}{q_{2N}}\right) = \lim_{N\to\infty} \left(\frac{p_{2N+1}q_{2N} - p_{2N}q_{2N+1}}{q_{2N+1}q_{2N}}\right).$$

Applying induction on n to the result of Problem 161, we can easily prove that for all $N \geq 0$, $|p_{2N+1}q_{2N} - p_{2N}q_{2N+1}| = |p_1q_0 - p_0q_1| = 1$. Consequently, $\left|\frac{p_{2N+1}q_{2N} - p_{2N}q_{2N+1}}{q_{2N+1}q_{2N}}\right| \leq \frac{1}{2N(2N+1)}$, which implies its limit must be 0. Therefore, $\lim_{N\to\infty}[a_0, a_1, \dots, a_{2N+1}] = \lim_{N\to\infty}[a_0, a_1, \dots, a_{2N}]$, which means $\lim_{N\to\infty}[a_0, a_1, \dots, a_N]$ exists and is equal to them.

对问题 161 的结果对 n 应用归纳法，我们可以很容易地证明对于所有的 $N \geq 0$， $|p_{2N+1}q_{2N} - p_{2N}q_{2N+1}| = |p_1q_0 - p_0q_1| = 1$。因此，$\left|\frac{p_{2N+1}q_{2N} - p_{2N}q_{2N+1}}{q_{2N+1}q_{2N}}\right| \leq \frac{1}{2N(2N+1)}$，这意味着它的极限一定是 0。所以，$\lim_{N\to\infty}[a_0, a_1, \dots, a_{2N+1}] = \lim_{N\to\infty}[a_0, a_1, \dots, a_{2N}]$，这意味着 $\lim_{N\to\infty}[a_0, a_1, \dots, a_N]$ 存在且等于它们。

Just like rational numbers can be written as finite simple continued fractions, irrational numbers can be written as infinite simple continued fractions. See, e.g., [42], for a proof of the following theorem.

就像有理数可以写成有限简单连分数一样，无理数可以写成无限简单连分数。例如，关于以下定理的证明可参见[42]。

Theorem 4.5.6 The value of each infinite simple continued fraction is irrational. Each positive irrational number greater than 1 can be written as exactly one infinite simple continued fraction.

定理 4.5.6 每个无限简单连分数的值都是无理数。每个大于 1 的正无理数都可以恰好写成一个无限简单连分数。

There is an algorithm called continued fraction algorithm, which finds the infinite simple continued fraction for any given positive irrational number. (In fact, it can also be applied to positive rational numbers and so it works for any positive real number.) Specifically, for any input x, the algorithm takes the following steps:

有一种称为连分数算法的算法，它能为任何给定的正无理数找到无限简单连分数。(实际上，它也可以应用于正有理数，因此对任何正实数都有效。)具体来说，对于任何输入 x， 该算法采取以下步骤:

1. $x' \leftarrow x; a_0 \leftarrow \lfloor x' \rfloor$;

2. for each $i \geq 1$ do:

2. 对于每个 $i \geq 1$ 执行:

if $x' = a_{i-1}$ then terminate; otherwise $x' \leftarrow \frac{1}{x' - a_{i-1}}, a_i \leftarrow \lfloor x' \rfloor$;

如果 $x' = a_{i-1}$，则终止；否则 $x' \leftarrow \frac{1}{x' - a_{i-1}}, a_i \leftarrow \lfloor x' \rfloor$;

Example 4.5.3 An infinite continuous fraction of the form

例 4.5.3 形式为

$$[a_1, a_2, \dots, a_k, a_{k+1}, \dots, a_{k+\ell}, a_k, a_{k+1}, \dots, a_{k+\ell}, \dots, \dots, a_k, a_{k+1}, \dots, a_{k+\ell}, \dots, \dots]$$

[11] [16] Bear in mind that, although we are now studying infinite continued fractions, these convergents can be considered convergents of a finite simple continued fraction.

[16] 请记住，虽然我们现在研究的是无限连分数，但这些渐进分数可以被视为有限简单连分数的渐进分数。

is called periodic and often written as $[a_1, a_2, \dots, \overline{a_k, a_{k+1}, \dots, a_{k+\ell}}]$. Show that, if n is a positive integer, then $\sqrt{n^2+1}$ equals a periodic simple infinite continuous fraction.

的无限连分数称为周期连分数，通常写成 $[a_1, a_2, \dots, \overline{a_k, a_{k+1}, \dots, a_{k+\ell}}]$。证明，如果 n 是正整数，那么 $\sqrt{n^2+1}$ 等于一个周期简单无限连分数。

Solution: Consider $x = \left[n, \overline{2n}\right]$. Clearly we have

解:考虑 $x = \left[n, \overline{2n}\right]$。显然我们有

$$n + \frac{1}{x+n} = x$$

Solving this equation, we get that $x = \sqrt{n^2+1}$.

解这个方程，我们得到 $x = \sqrt{n^2+1}$。

The continued fraction algorithm terminates at a certain point if x is rational. It keeps going forever if x is irrational. In the latter case, if we force the algorithm to terminate at some point, the result is a finite simple continued fraction, which is an approximation of x. With a bit more analysis, we can show that the error of this approximation is less than $\frac{1}{q^2}$, where q is the denominator of the value of the finite simple continued fraction. This further implies the following theorem.

如果 x 是有理数，连分数算法会在某一点终止。如果 x 是无理数，它会一直进行下去。在后一种情况下，如果我们在某一点强制算法终止，结果是一个有限简单连分数，它是 x 的一个近似值。通过进一步分析，我们可以证明这个近似值的误差小于 $\frac{1}{q^2}$，其中 q 是有限简单连分数值的分母。这进一步暗示了以下定理。

Theorem 4.5.7 For any positive irrational number ζ, there are infinitely many rational numbers $\frac{p}{q}$ such that

定理 4.5.7 对于任何正无理数 ζ，存在无穷多个有理数 $\frac{p}{q}$ 使得

$$\left|\zeta - \frac{p}{q}\right| < \frac{1}{q^2}.$$

Proof: (Dirichlet Argument) Consider an arbitrary $Q > 1$. The $Q+1$ numbers $0, \zeta - \lfloor\zeta\rfloor, 2\zeta - \lfloor 2\zeta\rfloor, \dots, Q\zeta - \lfloor Q\zeta\rfloor$ are all distributed in the interval $[0,1)$. If we divide $[0,1)$ into Q smaller intervals $\left[0, \frac{1}{Q}\right), \left[\frac{1}{Q}, \frac{2}{Q}\right), \dots, \left[\frac{Q-1}{Q}, 1\right)$, then two of them must fall into the same small interval, i.e., there exist $p_1, p_2 (0 \le p_1 < p_2 \le Q)$ such that $|p_1\zeta - \lfloor p_1\zeta\rfloor - (p_2\zeta - \lfloor p_2\zeta\rfloor)| < \frac{1}{Q}$. The above inequality is equivalent to $\left|\zeta - \frac{|p_2\zeta| - |p_1\zeta|}{p_2 - p_1}\right| < \frac{1}{Q(p_2-p_1)} < \frac{1}{(p_2-p_1)^2}$. Defining $q = p_2 - p_1$ and $p = \lfloor p_2\zeta\rfloor - \lfloor p_1\zeta\rfloor$, we get that

证明:(狄利克雷论证)考虑任意一个 $Q > 1$。$Q+1$ 个数 $0, \zeta - \lfloor\zeta\rfloor$， $2\zeta - \lfloor 2\zeta\rfloor, \dots, Q\zeta - \lfloor Q\zeta\rfloor$ 都分布在区间 $[0,1)$ 内。如果我们将 $[0,1)$ 分成 Q 个更小的区间 $\left[0, \frac{1}{Q}\right), \left[\frac{1}{Q}, \frac{2}{Q}\right), \dots, \left[\frac{Q-1}{Q}, 1\right)$，那么其中两个必定落入同一个小区间，即，存在 $p_1, p_2 (0 \le p_1 < p_2 \le Q)$ 使得 $|p_1\zeta - \lfloor p_1\zeta\rfloor - (p_2\zeta - \lfloor p_2\zeta\rfloor)| < \frac{1}{Q}$。上述不等式等价于 $\left|\zeta - \frac{|p_2\zeta| - |p_1\zeta|}{p_2 - p_1}\right| < \frac{1}{Q(p_2-p_1)} < \frac{1}{(p_2-p_1)^2}$。定义 $q = p_2 - p_1$ 和 $p = \lfloor p_2\zeta\rfloor - \lfloor p_1\zeta\rfloor$，我们得到

$$\left|\zeta - \frac{p}{q}\right| < \frac{1}{Qq} < \frac{1}{q^2} \qquad (4.1)$$

This result looks like what we need, except that we need infinitely many such rational numbers $\frac{p}{q}$, not just one. (Some people may argue that Theorem 4.5.7 does not require p and q to be coprime to each other, and thus the existence of a pair(p, q)implies the existence of infinitely many pairs(kp, kq). Unfortunately, this is wrong, because the guarantee we need for(kp, kq) would be $\left|\zeta - \frac{kp}{kq}\right| < \frac{1}{k^2q^2}$, which we can't achieve.) See Example 4.5.4 below for a justification of why there are infinitely many of them.

这个结果看起来像是我们需要的，只是我们需要无穷多个这样的有理数 $\frac{p}{q}$，而不只是一个。(有些人可能会争辩说定理 4.5.7 并不要求 p 和 q 互质，因此一对 (p, q) 的存在意味着无穷多对 (kp, kq) 的存在。不幸的是，这是错误的，因为我们对 (kp, kq) 需要的保证是 $\left|\zeta - \frac{kp}{kq}\right| < \frac{1}{k^2q^2}$，而我们无法实现这一点。) 关于为什么会有无穷多个这样的数，请参见下面的例 4.5.4。

Example 4.5.4 Show that Dirichlet Argument provides infinitely many rational approximations $\frac{p}{q}$ to ζ.

例 4.5.4 证明狄利克雷论证为 ζ 提供了无穷多个有理近似值 $\frac{p}{q}$。

Solution: Suppose that there are only finitely many such rational numbers $\frac{p}{q}$. Then there exists a sufficiently large $Q_0 > 1$ such that for all these $\frac{p}{q}$, $Q_0 > \frac{1}{\left|\frac{p}{q}-\zeta\right|q}$. For any $Q > Q_0$, we must have

解:假设只有有限多个这样的有理数 $\frac{p}{q}$。那么存在一个足够大的 $Q_0 > 1$，使得对于所有这些 $\frac{p}{q}$, $Q_0 > \frac{1}{\left|\frac{p}{q}-\zeta\right|q}$ 都成立。对于任何 $Q > Q_0$，我们必定有

$$\left|\frac{p}{q} - \zeta\right| > \frac{1}{Q_0 q} > \frac{1}{Qq} \qquad (4.2)$$

However, recall that the Dirichlet argument began with choosing an arbitrary $Q > 1$. We can definitely choose a $Q > Q_0$, and expect the argument to remain valid. The rational number $\frac{p}{q}$ constructed based on such a Q must satisfy Equation (4.1), which contradicts Equation (4.2).

然而，回想一下，狄利克雷论证是从选择一个任意的 $Q > 1$ 开始的。我们肯定可以选择一个 $Q > Q_0$，并期望该论证仍然有效。基于这样一个 Q 构造的有理数 $\frac{p}{q}$ 必须满足方程 (4.1)，这与方程 (4.2) 相矛盾。

Given that we can approximate any irrational number with an error bound $\frac{1}{q^2}$, it is natural to ask whether we can improve the bound to $\frac{1}{q^3}, \frac{1}{q^4}, \ldots$ It turns out that we may not be able to improve the error bound. Our main difficulty comes from algebraic numbers, as defined below.

鉴于我们可以用误差界限 $\frac{1}{q^2}$ 来逼近任何无理数，很自然会问我们是否可以将界限改进到 $\frac{1}{q^3}, \frac{1}{q^4}, \ldots$。事实证明，我们可能无法改进误差界限。我们的主要困难来自于如下定义的代数数。

Definition 4.5.3 A number is called an algebraic number if it is a root of a polynomial with integral coefficients. Otherwise, it is called a transcendental number.

定义 4.5.3 如果一个数是具有整数系数的多项式的根，则称它为代数数。否则，称它为超越数。

Theorem 4.5.8 (Liouville) If an irrational number ζ is a root of a degree-n polynomial with integral coefficients, then there is a real number $c(\zeta) > 0$ such that for all rational numbers $\frac{p}{q}$, $\left|\zeta - \frac{p}{q}\right| > \frac{c(\zeta)}{q^n}$

定理 4.5.8(刘维尔)如果一个无理数 ζ 是一个次数为 n 的具有整数系数的多项式的根，那么存在一个实数 $c(\zeta) > 0$，使得对于所有有理数 $\frac{p}{q}$，$\left|\zeta - \frac{p}{q}\right| > \frac{c(\zeta)}{q^n}$

Problem Set 10

习题集 10

Problem 161 For a continued fraction $[a_0, a_1, \dots, a_N]$, define

问题 161 对于一个连分数 $[a_0, a_1, \dots, a_N]$，定义

$$p_0 = a_0; q_0 = 1;\ p_1 = a_0 a_1 + 1; q_1 = a_1;$$

$$p_{n+2} = a_{n+2} p_{n+1} + p_n;\ q_{n+2} = a_{n+2} q_{n+1} + q_n.$$

Show that for all $n \geq 0, \frac{p_n}{q_n} = [a_0, a_1, \dots, a_N]$.

证明对于所有 $n \geq 0, \frac{p_n}{q_n} = [a_0, a_1, \dots, a_N]$。

Problem 162 Suppose for all $i (0 \leq i \leq N), a_i > 0$. Show that, if N is odd and $a_N < a'_N$, then $[a_0, a_1, \dots, a_N] > [a_0, a_1, \dots, a'_N]$; if N is even and $a_N < a'_N$, then $[a_0, a_1, \dots, a_N] < [a_0, a_1, \dots, a'_N]$.

问题 162 假设对于所有 $i (0 \leq i \leq N), a_i > 0$。证明，如果 N 是奇数且 $a_N < a'_N$，那么 $[a_0, a_1, \dots, a_N] > [a_0, a_1, \dots, a'_N]$；如果 N 是偶数且 $a_N < a'_N$，那么 $[a_0, a_1, \dots, a_N] < [a_0, a_1, \dots, a'_N]$。

Problem 163 A continued fraction $[a_0, a_1, \dots, a_N]$ is called pseudo-simple if one of the a_i is a negative integer and all the others are positive integers. Is it true that for all positive rational number $\frac{p}{q}$ where p and q are positive integers relatively prime to each other, there exists a pseudo-

问题 163 如果连分数 $[a_0, a_1, \dots, a_N]$ 中有一个 a_i 是负整数而其他所有的都是正整数，则称该连分数为伪简单连分数。对于所有正有理数 $\frac{p}{q}$ (其中 p 和 q 是互质的正整数)，是否存在一个伪简单连分数

simple continued fraction

简单连分数

$$[a_0, a_1, \dots, a_N] = \frac{p}{q}?$$

If so, provide a proof. Otherwise, provide a counter example.

如果是这样，请给出证明。否则，请给出一个反例。

Problem 164 Suppose a continued fraction $[a_0, a_1, \dots]$ is cyclic. That is, there exist $k > 0, \ell > 0$ such that for all $i \geq k, a_{i+\ell} = a_i$. Prove there exist integers u, v, w, x such that $[a_0, a_1, \dots] = \frac{v + w\sqrt{x}}{u}$.

问题 164 假设一个连分数 $[a_0, a_1, \dots]$ 是循环的。也就是说，存在 $k > 0, \ell > 0$ 使得对于所有 $i \geq k, a_{i+\ell} = a_i$。证明存在整数 u, v, w, x 使得 $[a_0, a_1, \dots] = \frac{v + w\sqrt{x}}{u}$。

Problem 165 Prove that there is no injection from the set of transcendental numbers to the set of algebraic numbers.

问题 165 证明不存在从超越数集到代数数集的单射。

Problem 166 (USA TST Jan 2016-1) Let $\sqrt{3} = 1.b_1b_2b_3\ldots_{(2)}$ be the binary representation of $\sqrt{3}$. Prove that for any positive integer n, at least one of the digits $b_n, b_{n+1}, \ldots, b_{2n}$ equals 1.

问题 166(2016 年 1 月美国国家队选拔测试第 1 题)设 b_1 的二进制表示为 b_0。证明对于任意正整数 b_2，b_3 中的至少一位数字等于 1。

Problem 167 Prove the following lemma: If we have real numbers $\xi > 0$ and $\alpha > 1$, and there exist infinitely many positive integers a, b such that

问题 167 证明以下引理:如果我们有实数 b_0 和 b_1，并且存在无穷多个正整数 b_2 使得

$$0 < \left|\xi - \frac{a}{b}\right| < \frac{1}{b^{\alpha}},$$

then ξ is irrational.

那么 b_0 是无理数。

Problem 168 (IberoAmerica MO 2009-5) Let $a_1 = 1$. For $n \geq 1$, define

问题 168(2009 年伊比利亚美洲数学奥林匹克第 5 题)设 b_0。对于 b_1，定义

$$a_{2n} = a_n + 1;\ a_{2n+1} = \frac{1}{a_{2n}}.$$

Prove that each rational number appears in the sequence exactly once.

证明每个有理数在该序列中恰好出现一次。

Problem 169 ([87], Problem 7.4(4)) Suppose that ξ is a positive irrational number. Let $a_0, a_1, a_2, \ldots$ be an infinite sequence of positive integers such that $\xi = [a_0, a_1, a_2, \ldots]$. Let $b_1, b_2, b_3, \ldots$ be a finite or infinite sequence of positive integers. Show that

问题 169([87]，问题 7.4(4))假设 b_0 是一个正无理数。设 b_1 是一个正整数的无穷序列，使得 b_2。设 b_3 是一个正整数的有限或无穷序列。证明

$$\lim_{n\to\infty}[a_0, a_1, a_2, \ldots, a_n, b_1, b_2, b_3, \ldots] = \xi.$$

Chapter 6

第 6 章

Introduction to Theoretical Computer Science

理论计算机科学导论

6.1 Turing Machines and Halting

6.1 图灵机与停机问题

What is a computer? This fundamental question was answered by British mathematician Alan Turing, the father of computer science. In 1936, he created a model called Turing Machine to represent any conceivable computer in the world. Today, this model still represents the vast majority of computers in the world. There are only a handful of exceptions in laboratories, like quantum computers, that can't be represented by a Turing Machine.

什么是计算机？这个基本问题由计算机科学之父、英国数学家艾伦·图灵给出了答案。1936 年，他创建了一个名为图灵机的模型来表示世界上任何可想象的计算机。如今，这个模型仍然代表着世界上绝大多数的计算机。在实验室中只有少数例外，比如量子计算机，无法用图灵机来表示。

Informally, a Turing Machine is an automaton that has a single tape. The tape is infinitely long, and divided into infinitely many cells that can be both read and written. At the beginning, a bit string of finite length is placed on the tape as input, and a read-and-write head points at the leftmost bit of input. The automaton has a single register that stores its current state. In each step of computation, the automaton reads the content of the current cell from the tape, and then uses it together with the current state to decide the action to take. The action to take includes three parts:

非正式地说，图灵机是一种具有单个磁带的自动机。磁带是无限长的，并且被分成无限多个可以读写的单元。一开始，一个有限长度的位串被放在磁带上作为输入，一个读写头指向输入的最左边的位。自动机有一个单一的寄存器来存储其当前状态。在计算的每一步中，自动机从磁带上读取当前单元的内容，然后将其与当前状态一起用于决定要采取的动作。要采取的动作包括三个部分:

- Deciding the new state.
- 决定新状态。
- Deciding what to write (in the current cell).
- 决定要写什么(在当前单元中)。
- Deciding how to move the read-and-write head. There are three options: moving to the left cell, moving to the right cell, and staying in the current location.
- 决定如何移动读写头。有三种选择:向左移动一格、向右移动一格以及停留在当前位置。

There is one state called Termination. Once this state is reached, the Turing Machine terminates its computation and whatever left on the tape is the output.

有一种状态称为终止状态。一旦达到此状态，图灵机就会终止其计算，磁带上剩余的内容即为输出。

Formally, we have the following definition.

形式上，我们有以下定义。

Definition 6.1.1 A Turing Machine is a tuple (Γ, S, W, M) , where

定义 6.1.1 图灵机是一个元组 (Γ, S, W, M) ，其中

- Γ is a finite set such that Start, Termination $\in \Gamma$.
- Γ 是一个有限集，使得起始状态、终止状态 $\in \Gamma$ 属于该集合。
- $S: \Gamma \times \{0,1,\star\} \to \Gamma$ is a mapping that decides the new state, where $\star$ is a symbol representing an empty cell.
- $S: \Gamma \times \{0,1,\star\} \to \Gamma$ 是一个映射，用于决定新状态，其中 $\star$ 是表示空白单元格的符号。
- $W: \Gamma \times \{0,1,\star\} \to \{0,1,\star\}$ is a mapping that decides what to write.
- $W: \Gamma \times \{0,1,\star\} \to \{0,1,\star\}$ 是一个映射，用于决定写入什么内容。
- $M: \Gamma \times \{0,1,\star\} \to \{-1,0,1\}$ is a mapping that decides where to move the read-and-write head. Here -1 means moving to the left, 1 to the right, and 0 staying at the current location.
- $M: \Gamma \times \{0,1,\star\} \to \{-1,0,1\}$ 是一个映射，用于决定将读写头移动到何处。这里 -1 表示向左移动，1 表示向右移动，0 表示停留在当前位置。

For a modern student, a Turing Machine is nothing but a formal representation of an algorithm. (But why is it an algorithm? Isn't it intended to be a computer? You'll see why, once you have finished the study of this section.) Hence, we can construct a Turing Machine for whatever algorithm in mind.

对于现代学生而言，图灵机不过是算法的一种形式表示。(但它为何是一种算法呢？它难道不是旨在成为一台计算机吗？等你学完本节内容，就会明白原因。)因此，我们可以为任何想到的算法构造一台图灵机。

Example 6.1.1 Construct a Turing Machine that can reverse a 9-bit input.

示例 6.1.1 构造一台能够反转 9 位输入的图灵机。

This is a very easy exercise and thus we skip the solution. Just encode in the state what has been read/written. For example, use a state R_{010} to represent that 010 has just been read as input; similarly, use a state W_{010} to represent that 010 has just been written as output. Then we can easily define the three mappings needed. For example, $S(\text{ Start }, 0) = R_0, M(R_0, 1) = 1$, ... You just need to practice writing things formally.

这是一道非常简单的练习题，所以我们略过解答过程。只需在状态中编码已读取/写入的内容。例如，使用状态 R_{010} 表示刚刚读取到输入 010；类似地，使用状态 W_{010} 表示刚刚写入输出 010。然后我们可以轻松定义所需的三个映射。例如， $S(\text{Start}, 0) = R_0, M(R_0, 1) = 1$ ， ... 你只需练习形式化地书写内容。

Example 6.1.2 Construct a Turing Machine that can reverse any finitely long bitstring.

示例 6.1.2 构造一台能够反转任何有限长比特串的图灵机。

You might get seriously wrong if you are thinking along the lines of Example 6.1.1. Here the key point is that you can't encode in the state what has been read because you have no idea how long the input is - the set of states is finite and thus you can do the encoding only for input with a fixed upper bound of length.

如果你按照示例 6.1.1 的思路去思考，可能会大错特错。这里的关键在于，你无法在状态中编码已读取的内容，因为你不知道输入有多长——状态集是有限的，因此你只能对长度有固定上限的输入进行编码。

To solve this problem, we can use $\star$ as a landmark. For the purpose of illustration, suppose the input is 01 . Our Turing Machine takes the following steps:

为了解决这个问题，我们可以使用 $\star$ 作为一个标记。为了便于说明，假设输入是 01。我们的图灵机按以下步骤操作:

1. Move the read-and-write head to the rightmost bit of the input.

1. 将读写头移动到输入的最右位。

2. Read the rightmost bit of the input, 1, and notice that there is something (0) to its left. In other words, this bit is not the last bit to copy.

2. 读取输入的最右位，即 1，并注意到其左边有内容(0)。换句话说，这位不是要复制的最后一位。

3. Memorize 1, erase it from the tape (i.e., replace it with $\star$), and move to the right.

3. 记住 1，将其从磁带上擦除(即用 $\star$ 替换它)，然后向右移动。

4. When a $\star$ is seen, write 1, the bit we just memorized.

4. 当看到 $\star$ 时，写入 1，即我们刚刚记住的位。

5. Move back to the left, read the remaining bit 0, and notice there is nothing to its left. Now we realize that it is the last bit to copy. Once we finish copying it, we will never need to come back again.

5. 向左移动，读取剩余的位 0，并注意其左边没有其他内容。现在我们意识到它是最后一位要复制的。一旦我们完成复制它，就再也不需要回来了。

6. Memorize 0, erase it from the tape, and move to the right.

6. 记住 0，将其从磁带上擦除，然后向右移动。

7. Once we have passed the 1 we wrote earlier, and see a $\star$, write down 0 . Since this is the last bit 1 to copy, we are done.

7. 一旦我们经过之前写入的 1，并且看到一个 $\star$ ，就写下 0。由于这是最后一位要复制的 1，我们就完成了。

Do you see the above constructed Turing Machine is essentially an algorithm? Of course, nobody is interested in writing algorithms in such an awkward way. We are interested in Turing Machines only

because they are an abstract model of computers that enable theoretical studies. We never use them to solve algorithmic problems in practice.

你是否看到上面构造的图灵机本质上就是一个算法？当然，没有人会有兴趣以如此笨拙的方式编写算法。我们对图灵机感兴趣只是因为它们是计算机的一种抽象模型，能够进行理论研究。我们在实践中从不使用它们来解决算法问题。

In some literature, you see Turing Machines with more than one tape. By convention, we assume these machines always have both their input and their output on the first tape.

在一些文献中，你会看到具有不止一条磁带的图灵机。按照惯例，我们假设这些机器总是在第一条磁带上既有输入又有输出。

Definition 6.1.2 For $k > 1$, a k -tape Turing Machine is a tuple (Γ, S, W, M) , where

定义 6.1.2 对于 $k > 1$ ， 一个 k 带图灵机是一个元组 (Γ, S, W, M) ， 其中

- Γ is a finite set such that Start, Termination $\in \sum$.
- Γ 是一个有限集，使得开始、终止 $\in \sum$ 。
- $S: \Gamma \times \{0,1,\star\}^k \to \Gamma$ is a mapping that decides the new state, where $\star$ is a symbol representing an empty cell.
- $S: \Gamma \times \{0,1,\star\}^k \to \Gamma$ 是一个映射，它决定新的状态，其中 $\star$ 是表示空单元格的符号。
- $W: \Gamma \times \{0,1,\star\}^k \to \{0,1,\star\}^k$ is a mapping that decides what to write on each tape.
- $W: \Gamma \times \{0,1,\star\}^k \to \{0,1,\star\}^k$ 是一个映射，它决定在每条磁带上写什么。
- $M: \Gamma \times \{0,1,\star\}^k \to \{-1,0,1\}^k$ is a mapping that decides where to move the read-and-write heads. Here -1 means moving to the left, 1 to the right, and 0 staying at the current location.
- $M: \Gamma \times \{0,1,\star\}^k \to \{-1,0,1\}^k$ 是一个映射，它决定读写头向哪里移动。这里 -1 表示向左移动，1 表示向右移动，0 表示停留在当前位置。

It turns out that having additional tapes would not enable a Turing Machine to solve more problems.

事实证明，拥有额外的磁带并不能使图灵机解决更多问题。

Theorem 6.1.1 For any k -tape Turing Machine T , there is a (one-tape) Turing Machine T' such that T' halts on an input x if and only if T halts on x , and that in case of halting, T' and T generates exactly the same output on the same input. Why is this true? Because you can essentially simulate k tapes with a single tape. For example, you might want to use $2k$ consecutive cells to simulate a cell from each of the k tapes, and to indicate whether each of the k heads is at this location. Therefore, a simulated reading of all the k heads would involve the actual head moving along the real tape, finding the locations of all simulated heads, reading the contents of the simulated cells... A proof of (a variant of) Theorem 6.1.1 can be found in [50]. Interested readers are encouraged to read it.

定理 6.1.1 对于任何 k 带图灵机 T ， 存在一个(单带)图灵机 T' ， 使得 T' 在输入 x 上停机当且仅当 T 在 x 上停机，并且在停机的情况下， T' 和 T 在相同输入上产生完全相同的输出。为什么这是真的呢？因为你本质上可以用一条磁带模拟 k 条磁带。例如，你可能想用 $2k$ 个连续的单元格来模拟 k 条磁带中每条磁带的一个单元格，并指示每个 k 个头是否在这个位置。因此，对所有 k 个头的模拟读取将涉及实际

的头沿着真实磁带移动，找到所有模拟头的位置，读取模拟单元格的内容……定理 6.1.1(的一个变体)的证明可以在[50]中找到。鼓励感兴趣的读者去阅读。

12

Similarly, restricting tapes to be infinite in one direction only, or having some read-only tapes in addition to normal tapes, or using a larger set as the alphabet. 2 usually would not affect the computability of problems. Therefore, while you can see many different definitions of Turing Machines across different books, these definitions are usually equivalent to each other.

类似地，将磁带限制为仅在一个方向上无限，或者除了普通磁带之外还有一些只读磁带，或者使用更大的集合作为字母表。2 通常不会影响问题的可计算性。因此，虽然你可以在不同的书中看到图灵机的许多不同定义，但这些定义通常彼此等价。

Example 6.1.3 We say T is a ternary Turing Machine if the alphabet is $\{0,1,2,\star\}$ instead of $\{0,1,\star\}$. Show that for any ternary Turing Machine T , there is a Turing Machine S , such that T halts on an input x if and only if S halts on an equivalent input x' (where x' is obtained by writing each ternary digit in the binary form, e.g., $x = 0121 \rightarrow x' = 00011001$), and that in case of halting, the outputs of S and T are also equivalent in the above sense.

示例 6.1.3 如果字母表是 $\{0,1,2,\star\}$ 而非 $\{0,1,\star\}$，我们称 T 是一台三元图灵机。证明对于任何三元图灵机 T，存在一台图灵机 S，使得 T 在输入 x 上停机当且仅当 S 在等价输入 x' 上停机(其中 x' 通过将每个三元数字写成二进制形式获得，例如 $x = 0121 \rightarrow x' = 00011001$)，并且在停机的情况下， S 和 T 的输出在上述意义上也是等价的。

The key idea is to use a pair of cells of S 's tape to represent each cell of T 's tape.

关键思想是使用 S 的磁带中的一对单元来表示 T 的磁带中的每个单元。

Suppose $T = (\Gamma_T, S_T, W_T, M_T)$. We construct $S = (\Gamma_S, S_S, W_S, M_S)$. For each $s \in \Gamma_T$, there are a lot of corresponding states in Γ_S . For example, there should be a state representing that T is in state s and the reading of the current pair on S just starts. There should also be a state representing that T is in state s , the first cell of the current pair is 0, and we are reading the second cell. Yet another state represents that T is in state s , the current pair is(0,1)-which means the current cell on T is 1, and we just start writing ...

假设 $T = (\Gamma_T, S_T, W_T, M_T)$。我们构造 $S = (\Gamma_S, S_S, W_S, M_S)$。对于每个 $s \in \Gamma_T$，在 Γ_S 中有许多相应的状态。例如，应该有一个状态表示 T 处于状态 s 并且刚刚开始读取 S 上的当前对。还应该有一个状态表示 T 处于状态 s，当前对的第一个单元为 0，并且我们正在读取第二个单元。另一个状态表示 T 处于状态 s，当前对是(0,1)——这意味着 T 上的当前单元是 1，并且我们刚刚开始写入...

Essentially, S remembers T 's state, as well as the incomplete read/write T is doing, so that the work of S on a pair of cells is equivalent to that of T on a single cell.

本质上， S 记住 T 的状态以及 T 正在进行的未完成读/写操作，使得 S 在一对单元上的工作等同于 T 在单个单元上的工作。

---

12 [1] We must encode in the state whether the currently read bit is the last one-In order to get this information, every time the Turing Machine reads a bit, it must look at the cell to the left, and see whether it is empty. Thanks to TA Moyang Xie for bringing this issue to my attention in summer 2024.

[1] 我们必须在状态中编码当前读取的位是否是最后一位——为了获得这个信息，每次图灵机读取一位时，它必须查看左边的单元格，看它是否为空。感谢谢默洋提醒我在 2024 年夏天注意到这个问题。

The full formal specification of S is left for homework.

S 的完整形式规范留作作业。

While the above discussions of (variants of) Turing Machines might be interesting, the Turing Machines we studied so far differ from the computers we use today in an important aspect: Each of them solves a fixed algorithmic problem only. In other words, they do not support programming. Now we fix this issue.

虽然上述关于(各种变体的)图灵机的讨论可能很有趣，但到目前为止我们研究的图灵机与我们今天使用的计算机在一个重要方面有所不同:它们中的每一个都只解决一个固定的算法问题。换句话说，它们不支持编程。现在我们解决这个问题。

Definition 6.1.3 We say a Turing Machine T simulates the work of another Turing Machine S with program P if both of the following two requirements are met:

定义 6.1.3 如果满足以下两个要求，我们称图灵机 T 用程序 P 模拟另一台图灵机 S 的工作:

- For every finite-length bitstring x, T halts on input $x \star P$ if and only if S halts on x ; 6.1. TURING MACHINES AND HALTING
- 对于每个有限长度的位串 x, T ，在输入 $x \star P$ 上停机当且仅当 S 在 x 上停机；6.1. 图灵机与停机

- In case of halting, the output of T on input $x \star P$ is identical to that of S on input x .
- 在停机的情况下， T 在输入 $x \star P$ 上的输出与 S 在输入 x 上的输出相同。

Definition 6.1.4 A Turing Machine T is said to be Universal if for every Turing Machine S, there is a program P such that T simulates S with program P .

定义 6.1.4 一台图灵机 T 被称为通用图灵机，如果对于每台图灵机 S，存在一个程序 P ，使得 T 用程序 P 模拟 S 。

Apparently, today's computers are closer to Universal Turing Machines, rather than the single-task Turing Machines we studied earlier. But do Universal Turing Machines exist? The good news is that they do.

显然，当今的计算机更接近通用图灵机，而不是我们之前研究的单任务图灵机。但是通用图灵机存在吗？好消息是它们存在。

Theorem 6.1.2 There exists a Universal Turing Machine.

定理 6.1.2 存在一台通用图灵机。

A proof of Theorem 6.1.2 can be found in standard textbooks like [3]. The main idea is that we can construct a Universal Turing Machine by adding two tapes to a regular Turing Machine, one for storing the mappings S, W, M of the simulated Turing Machine, the other for storing the current state of the

---

[13] [2] The alphabet is the set of characters that are allowed to appear in the tape, including the space symbol. For example, in our definition of Turing Machine, the alphabet is $\{0,1,\star\}$.

[2] 字母表是允许出现在磁带上的字符集，包括空格符号。例如，在我们对图灵机的定义中，字母表是 $\{0,1,\star\}$ 。

simulated Turing Machine. Since adding extra tapes does not enhance Turing Machines, we can argue that without these extra tapes there is also a Universal Turing Machine.

定理 6.1.2 的证明可以在像[3]这样的标准教科书中找到。主要思想是我们可以通过给一台常规图灵机添加两条磁带来构造一台通用图灵机，一条用于存储被模拟图灵机的映射 S, W, M ，另一条用于存储被模拟图灵机的当前状态。由于添加额外的磁带不会增强图灵机的能力，我们可以论证没有这些额外的磁带也存在一台通用图灵机。

Given the existence of Universal Turing Machines, we need to study what problems these machines can solve. In reality, computational problems vary greatly. You can hardly find a general framework for all of them. However, in the theory of computation, fundamental computational problems are modeled as the decision of certain languages.

考虑到通用图灵机的存在，我们需要研究这些机器能够解决哪些问题。实际上，计算问题千差万别。你几乎找不到一个适用于所有问题的通用框架。然而，在计算理论中，基本的计算问题被建模为对某些语言的判定。

Suppose $\sum$ is an alphabet (a finite set with $\star \in \sum$). A finite sequence of symbols from $\sum_0 = \sum - \{\star\}$ is called a word. We define $\sum_0^\star$ to be the set of all words (including the empty word) over $\sum$. A subset of $\sum_0^\star$ is called a language. We can attempt to use a Turing Machine T to decide (the membership of) a language L : Put a word x on T 's tape as input, run T , and expect T 's output to be 1 when $x \in L$, to be 0 otherwise. Ideally, T does exactly what we expect it to do. In this case, the language L is said to be recursive.

假设 $\sum$ 是一个字母表(一个具有 $\star \in \sum$ 个元素的有限集)。来自 $\sum_0 = \sum - \{\star\}$ 的符号的有限序列称为一个单词。我们将 $\sum_0^\star$ 定义为 $\sum$ 上所有单词(包括空单词)的集合。 $\sum_0^\star$ 的一个子集称为一种语言。我们可以尝试使用图灵机 T 来判定一种语言 L 的(成员资格):将一个单词 x 放在 T 的磁带上作为输入，运行 T ，并期望当 $x \in L$ 时 T 的输出为 1，否则为 0。理想情况下， T 恰好按照我们期望的那样运行。在这种情况下，语言 L 被称为递归的。

Definition 6.1.5 A language $L \subseteq \sum_0^\star$ is said to be recursive if there is a Turing Machine that halts on every input x , outputting 1 when $x \in L$, outputting 0 otherwise. In this case, we also say that the Turing Machine accepts the language L , and that (the membership of) the language L is decidable. If there is no such Turing Machine, then we say that the language L is undecidable. A language $L \subseteq \sum_0^\star$ is said to be recursively enumerable if there is a Turing Machine that halts on every input $x \in L$; in case of halting, it outputs 1 when $x \in L$, and outputs 0 otherwise.

定义 6.1.5 如果存在一台图灵机，它对每个输入 x 都能停机，当 $x \in L$ 时输出 1，否则输出 0，那么一种语言 $L \subseteq \sum_0^\star$ 就被称为递归的。在这种情况下，我们也说该图灵机接受语言 L ，并且语言 L 的(成员资格)是可判定的。如果不存在这样的图灵机，那么我们就说语言 L 是不可判定的。如果存在一台图灵机，它对每个输入 $x \in L$ 都能停机，那么一种语言 $L \subseteq \sum_0^\star$ 就被称为递归可枚举的；在停机的情况下，当 $x \in L$ 时它输出 1，否则输出 0。

Clearly, a recursive language is recursively enumerable, but the converse is not true. Why a recursively enumerable language is called "recursively enumerable"? Maybe we have a way to "enumerate" all its words? The answer is yes. Please think of how we can enumerate all its words using a Turing Machine.

显然，递归语言是递归可枚举的，但反之则不成立。为什么递归可枚举语言被称为“递归可枚举”呢？也许我们有一种方法可以“枚举”它的所有单词？答案是肯定的。请思考一下我们如何使用图灵机来枚举它的所有单词。

Example 6.1.4 Show that, for any recursively enumerable language L, there is a Turing Machine T whose output is a list of all its words. If L is finite, T halts after outputting all its words. If L is infinite, T never halts.

例 6.1.4 证明，对于任何递归可枚举语言 L，存在一台图灵机 T，其输出是它所有单词的列表。如果 L 是有限的，T 在输出完所有单词后停机。如果 L 是无限的，T 永不停机。

Solution: First we realize that L is countable (do you see it?) and thus there is a list of all its words. Let this list be $w_1, w_2, \ldots, w_n, \ldots$ Since L is recursively enumerable, there is a Turing Machine T_0 that halts on every input $x \in L$; in case of halting, T_0 outputs 1 when $x \in L$, and outputs 0 otherwise. Now we construct a Turing Machine T that works in the following way: For each positive integer n, T simulates a single step of the executions of n copies of T_0, on inputs $w_1, \ldots, w_n$, respectively. In other words, T works in the following manner:

解:首先我们意识到 L 是可数的(你看出来了吗？)，因此存在它所有单词的一个列表。设这个列表为 $w_1, w_2, \ldots, w_n, \ldots$。由于 L 是递归可枚举的，所以存在一台图灵机 T_0，它对每个输入 $x \in L$ 都能停机；在停机的情况下，当 $x \in L$ 时 T_0 输出 1，否则输出 0。现在我们构造一台图灵机 T，它按如下方式工作:对于每个正整数 n, T，分别模拟 n 个 T_0 副本在输入 $w_1, \ldots, w_n$ 上执行的单步操作。换句话说，T 按如下方式工作:

- Simulates the first step of $T_0(w_1)$;
- 模拟 $T_0(w_1)$ 的第一步；
- Simulates the second step of $T_0(w_1)$, and the first step of $T_0(w_2)$;
- 模拟 $T_0(w_1)$ 的第二步和 $T_0(w_2)$ 的第一步；
- Simulates the third step of $T_0(w_1)$, the second step of $T_0(w_2)$, and the first step of $T_0(w_3)$;
- 模拟 $T_0(w_1)$ 的第三步、 $T_0(w_2)$ 的第二步和 $T_0(w_3)$ 的第一步；
- ...

Notice that, while T_0 may never halt on some inputs, the simulation of $T_0(w_n)$ will definitely terminate, as long as T_0 halts on input w_n. Once T finishes the simulation of $T_0(w_n)$, it outputs w_n. Easy to see the above process enumerates all words of L.

注意，虽然 T_0 在某些输入上可能永远不会停机，但只要 T_0 在输入 w_n 上停机，对 $T_0(w_n)$ 的模拟肯定会终止。一旦 T 完成对 $T_0(w_n)$ 的模拟，它就会输出 w_n。很容易看出，上述过程枚举了 L 的所有单词。

When you compare recursive languages with recursively enumerable languages, you see the difference is in whether the Turing Machine halts on all inputs. So we are interested in the following fundamental question (called the Turing Halting Problem): Given a Turing Machine T, and a word x, can we decide whether T halts on input x? In other words, is the language of Turing Halting Problem $\{(T, x) \mid T$ halts on input $x\}$ recursive? Unfortunately, the answer is negative.

当你将递归语言与递归可枚举语言进行比较时，你会发现区别在于图灵机是否在所有输入上都停机。所以我们对以下基本问题(称为图灵停机问题)感兴趣:给定一个图灵机 T 和一个单词 x，我们能否判定 T 在输入 x 上是否停机？换句话说，图灵停机问题“$\{(T, x) \mid T$ 在输入 $x\}$ 上停机”这个语言是递归的吗？不幸的是，答案是否定的。

Theorem 6.1.3 The language of Turing Halting Problem is not recursive.

定理 6.1.3 图灵停机问题的语言不是递归的。

A somewhat stronger result is Rice's Theorem. We say a property of language is trivial if either all languages have the property, or none of them has it. Now consider a nontrivial property P of (the language accepted by) a Turing Machine. Given the description of a Turing Machine, can we decide whether it [3] has the property P ? The answer is again negative.

一个稍微更强的结果是赖斯定理。如果所有语言都具有某个语言属性，或者所有语言都不具有该属性，我们就说这个语言属性是平凡的。现在考虑图灵机(所接受的语言)的一个非平凡属性 P 。给定一个图灵机的描述，我们能否判定它是否具有属性 P 呢？答案还是否定的。

Theorem 6.1.4 (Rice) The language of any nontrivial property of (the language accepted by a) Turing Machine is not recursive. 6.1. TURING MACHINES AND HALTING

定理 6.1.4(赖斯)图灵机(所接受的语言)的任何非平凡属性的语言都不是递归的。6.1. 图灵机与停机

You might wonder what these theorems mean in reality. Do they really matter? Yes, they do. They tell us that, even if we did not mind the time, space, and communications resources consumed by computation, our computers would still have very limited computational power. There are some problems (in fact, too many problems) that can never be solved by a computer, despite the widely existent over-optimistic belief in computers.

你可能想知道这些定理在现实中意味着什么。它们真的重要吗？是的，它们很重要。它们告诉我们，即使我们不介意计算所消耗的时间、空间和通信资源，我们的计算机的计算能力仍然非常有限。尽管人们普遍对计算机过度乐观，但仍有一些问题(事实上，有太多问题)计算机永远无法解决。

Problem Set 17

习题集 17

Problem 222 Construct a Turing Machine whose output is the length of its input. For instance, if the input is 0110100111, then the output is 1010 .

问题 222 构造一个图灵机，其输出是输入的长度。例如，如果输入是 0110100111，那么输出是 1010。

Problem 223 Write down the full formal specification of S for the solution of Example 6.1.3.

---

[14] [3] In fact, the language it accepts. Note that we do not consider nontrivial properties of the Turing Machine that does not belong to the language it accepts. For example, we do not consider which input leads to the shortest running time of the Turing Machine. While this is a non-trivial property of the Turing Machine, it is not a property of the language accepted.

[3] 实际上，是它所接受的语言。注意，我们不考虑不属于它所接受语言的图灵机的非平凡属性。例如，我们不考虑哪个输入会导致图灵机运行时间最短。虽然这是图灵机的一个非平凡属性，但它不是所接受语言的属性。

问题 223 写出示例 6.1.3 的解决方案中 S 的完整形式规范。

Problem 224 We say T is a Turing Machine with one-way tape if its tape is one-way infinite to the right, i.e., its tape has a left end. Show that for any Turing Machine S, there is a Turing Machine T with one-way tape, such that T halts on an input x if and only if S halts on x , and that in case of halting, T and S generate exactly the same output on the same input.

问题 224 如果一个图灵机的磁带向右是单向无限的，即它的磁带只有左端，我们就说它是具有单向磁带的图灵机。证明对于任何图灵机 S，存在一个具有单向磁带的图灵机 T ，使得 T 在输入 x 上停机当且仅当 S 在 x 上停机，并且在停机的情况下， T 和 S 在相同输入上产生完全相同的输出。

Problem 225 Show that a language $L \in \sum_0^\star$ is recursively enumerable if and only if there is a Turing Machine that halts on every input $x \in L$ and does not halt otherwise.

问题 225 证明一个语言 $L \in \sum_0^\star$ 是递归可枚举的，当且仅当存在一个图灵机，它在每个输入 $x \in L$ 上都停机，否则不停机。

Problem 226 Please decide whether the following proposition is true or false. Prove your answer.

问题 226 请判定以下命题是真还是假。证明你的答案。

Proposition 6.1.1 For any (one-tape) Turing machine T , there exists another Turing Machine T' such that T' halts on input x if and only if T does not halt on input x .

命题 6.1.1 对于任何(单带)图灵机 T ，存在另一个图灵机 T' ，使得 T' 在输入 x 上停机当且仅当 T 在输入 x 上不停机。

Problem 227 For any Turing Machine T , we define a function f_T : If T halts and outputs 1 on input x , then $f_T(x) = 1$; otherwise, $f_T(x) = 0$. Is the language $\{(T, x, y) \mid f_T(x) = f_T(y)\}$ recursive? Prove your answer.

问题 227 对于任何图灵机 T ，我们定义一个函数 f_T :如果 T 在输入 x 上停机并输出 1，那么 $f_T(x) = 1$ ；否则， $f_T(x) = 0$ 。语言 $\{(T, x, y) \mid f_T(x) = f_T(y)\}$ 是递归的吗？证明你的答案。

Problem 228 We say a Turing Machine T runs faster than another Turing Machine S on input x , if one of the following requirements is satisfied:

问题 228 我们说图灵机 T 在输入 x 上比另一个图灵机 S 运行得更快，如果满足以下要求之一:

- T halts on input x in t steps, S halts on input x in s steps, and $t < s$;
- T 在 t 步内对输入 x 停机， S 在 s 步内对输入 x 停机，且 $t < s$;
- T halts on input x , but S does not halt on input x .
- T 在输入 x 上停机，但 S 在输入 x 上不停机。

We say a Turing Machine T runs as fast as another Turing Machine S on input x , if one of the following requirements is satisfied:

我们说图灵机 T 在输入 x 上与另一个图灵机 S 运行得一样快，如果满足以下要求之一:

- T halts on input x in t steps, S halts on input x in s steps, and $t = s$;
- T 在 t 步内对输入 x 停机， S 在 s 步内对输入 x 停机，且 $t = s$;
- Neither T nor S halts on input x .

- T 和 S 在输入 x 上都不停机。

(a) Is the language $\{(T,S,x) \mid T$ runs faster than S on input $x\}$ recursively enumerable? Prove your answer.

(a) 语言 $\{(T,S,x) \mid T$ 在输入 $x\}$ 上比 S 运行得更快是递归可枚举的吗？证明你的答案。

(b) Is the language $\{(T,S,x) \mid T$ runs faster than, or as fast as S on input $x\}$ recursively enumerable? Prove your answer.

(b) 语言 $\{(T,S,x) \mid T$ 在输入 $x\}$ 上比 S 运行得更快或一样快是递归可枚举的吗？证明你的答案。

Problem 229 (UIUC CS373 Spring 2013 Homework 9-3) Disjoint languages A and B are said to be recursively separable if there is a recursive language L such that $A \subseteq L$ and $B \subseteq \bar{L}$. Prove that, for disjoint languages A and B, if both $\bar{A}$ and $\bar{B}$ are recursively enumerable, then A and B are recursively separable.

问题 229 (伊利诺伊大学厄巴纳 - 香槟分校计算机科学 373 课程 2013 年春季作业 9 - 3)不相交的语言 A 和 B 被称为递归可分离的，如果存在一个递归语言 L，使得 $A \subseteq L$ 且 $B \subseteq \bar{L}$。证明对于不相交的语言 A 和 B，如果 $\bar{A}$ 和 $\bar{B}$ 都是递归可枚举的，那么 A 和 B 是递归可分离的。

6.2 Two Randomized Algorithms

6.2 两种随机算法

You must have learned a good number of algorithms. They are all deterministic in the sense that for any given input, the execution of the algorithm is always the same. There is never any change in any step of the algorithm, from one execution to another. Traditional algorithms, e.g., traditional sorting algorithms, all fall into this category.

你肯定学过很多算法。它们都是确定性的，即对于任何给定的输入，算法的执行总是相同的。从一次执行到另一次执行，算法的任何步骤都不会有任何变化。传统算法，例如传统排序算法，都属于这一类。

Recall the quicksort algorithm you have learned.

回忆一下你学过的快速排序算法。

Algorithm quicksort $(x_1, x_2, \ldots, x_n)$

算法 快速排序 $(x_1, x_2, \ldots, x_n)$

pick a pivot x_i arbitrarily; let y, z be empty arrays;

任意选择一个枢轴 x_i；令 y, z 为空数组；

if $n > 1$

如果 $n > 1$

for $j = 1$ to $n (j \neq i)$

从 $j = 1$ 到 $n (j \neq i)$

if $x_j < x_i$

如果 $x_j < x_i$

append x_j to array y ;

将 x_j 追加到数组 y 中；

else

否则

append x_j to array z ;

将 x_j 追加到数组 z 中；

y = quicksort(y);

y = 对 y 进行快速排序；

z'=quicksort(z);

z' = 对 z 进行快速排序；

return the array $y' \parallel x_i \parallel z'$.

返回数组 $y' \parallel x_i \parallel z'$ 。

Let $T(n)$ be its running time on an n -array. If we were lucky enough to always pick the middle element of the array as pivot, then we would have $T(n) = \Theta(n) + 2T\left(\frac{n}{2}\right)$. By Theorem 2.5, we get that $T(n) = \Theta(n\log n)$, which seems good. However, in reality, we could end up picking the smallest or the biggest element as pivot. In such worst cases, we would have $T(n) = \Theta(n) + T(n-1)$, which implies that $T(n) = \Theta(n^2)$.

设 $T(n)$ 为其在一个 n 数组上的运行时间。如果我们足够幸运，总是选择数组的中间元素作为枢轴，那么我们会得到 $T(n) = \Theta(n) + 2T\left(\frac{n}{2}\right)$ 。根据定理 2.5，我们得到 $T(n) = \Theta(n\log n)$ ，这看起来不错。然而，在现实中，我们最终可能会选择最小或最大的元素作为枢轴。在这种最坏的情况下，我们会有 $T(n) = \Theta(n) + T(n-1)$ ，这意味着 $T(n) = \Theta(n^2)$ 。

Note that there is no way to guarantee you pick a "good" pivot. So, what can we do if we worry about the worst cases? Can we somehow avoid them? Well, you should notice that most elements are pretty good choices for pivot, and thus a random selection of pivot might help us avoid the worst cases (with a good probability). With this minor (but significant) modification, we get the randomized version of quicksort.

注意，没有办法保证你选择一个“好的”枢轴。那么，如果我们担心最坏的情况该怎么办呢？我们能以某种方式避免它们吗？嗯，你应该注意到大多数元素都是枢轴的相当不错的选择，因此随机选择枢轴可能会帮助我们避免最坏的情况(有很大的概率)。通过这个小(但很重要)的修改，我们得到了快速排序的随机化版本。

Algorithm randomquicksort $(x_1, x_2, \dots, x_n)$

算法 随机快速排序 $(x_1, x_2, \dots, x_n)$

pick a pivot x_i uniformly at random; let y, z be empty arrays;

随机均匀地选择一个轴点 x_i ； 令 y, z 为空数组；

if $n > 1$

如果 $n > 1$

for $j = 1$ to $n(j \neq i)$

从 $j = 1$ 到 $n(j \neq i)$

if $x_j < x_i$

如果 $x_j < x_i$

append x_j to array y ;

将 x_j 追加到数组 y 中；

else

否则

append x_j to array z ;

将 x_j 追加到数组 z 中；

y'=quicksort(y); z'=quicksort(z);

y' = 快速排序(y)；z' = 快速排序(z)；

return the array $y'|x_i|z'$.

返回数组 $y'|x_i|z'$ 。

Assuming each iteration in the for-loop takes one unit of time, then for the above randomized quicksort algorithm, we have 4

假设 for 循环中的每次迭代耗时一个单位时间，那么对于上述随机快速排序算法，我们有 4

$$T(n) = n + T(k) + T(n - k - 1),$$

where k is the number of elements smaller than the pivot. Considering the uniform distribution of k over $\{0,1, \dots, n-1\}$, we easily get that

其中 k 是小于轴点的元素数量。考虑到 k 在 $\{0,1, \dots, n-1\}$ 上的均匀分布，我们很容易得到

$$E[T(n)] = n + E[T(k)] + E[T(n - k - 1)] = n + 2E[T(k)] = n + \frac{2}{n}\sum_{k=0}^{n-1} E\,[T(k)],$$

which is a recurrence relation. Although we cannot apply Theorem 2.5.1 to it, we can use induction to show $E\big(T(n)\big) \leq 2n\ln n$. More precisely, we have the following lemma, which implies that the expected running time of randomized quicksort is $O(n\log n)$.

这是一个递归关系。虽然我们不能对其应用定理 2.5.1，但我们可以用归纳法证明 $E\big(T(n)\big) \leq 2n\ln n$ 。更确切地说，我们有以下引理，这意味着随机快速排序的期望运行时间是 $O(n\log n)$ 。

Lemma 6.2.1 Suppose $T(0) = T(1) = 0$ and that for all integer $n > 1, T(n) = n + \frac{2}{n}\sum_{k=0}^{n-1} T\,(k)$. Then for all positive integer $n, T(n) \leq 2n\ln n$.

引理 6.2.1 假设 $T(0)=T(1)=0$ 并且对于所有整数 $n>1, T(n)=n+\frac{2}{n}\sum_{k=0}^{n-1}T(k)$ 。那么对于所有正整数 $n, T(n)\leq 2n\ln n$ 。

Proof: By induction. The inequality clearly holds for $n=1$.

证明:通过归纳法。对于 $n=1$ ，该不等式显然成立。

Now assume it holds for all positive integer $n\leq N$. When $n=N+1$, we get that

现在假设它对所有正整数 $n\leq N$ 都成立。当 $n=N+1$ 时，我们得到

$$T(N+1)=N+1+\frac{2}{N+1}\sum_{k=0}^{N}T(k)\leq N+1+\frac{4}{N+1}\sum_{k=2}^{N}k\ln k$$

$$\leq N+1+\frac{4}{N+1}\int_{2}^{N+1}k\ln k dk=N+1+\frac{2}{N+1}\int_{2}^{N+1}\ln k dk^{2}$$

$$=N+1+\frac{2}{N+1}\left(k^{2}\ln k|_{2}^{N+1}-\int_{2}^{N+1}k^{2}\,d\ln k\right)$$

$$=N+1+\frac{2}{N+1}\left((N+1)^{2}\ln(N+1)-4\ln 2-\frac{(N+1)^{2}-4}{2}\right)$$

$$=2(N+1)\ln(N+1)+\frac{4-8\ln 2}{N+1}$$

$$<2(N+1)\ln(N+1),$$

which means the inequality holds for $n=N+1$ as well.

这意味着该不等式对 $n=N+1$ 也成立。

15

Since the result of $O(n\log n)$ gives us only an upper bound, naturally one would ask whether the time complexity is $\Theta(n\log n)$, or we might actually get better than that. It turns out that the former is true, because for an increasing and convex [5] function $T(\quad)$ as studied in Lemma 6.2.1, we always have

由于 $O(n\log n)$ 的结果只给了我们一个上界，自然而然地，人们会问时间复杂度是否是 $\Theta(n\log n)$ ，或者我们实际上是否能得到比这更好的结果。事实证明前者是正确的，因为对于引理 6.2.1 中研究的递增且凸的 [5] 函数 $T(\quad)$ ，我们总是有

$$T(n)=n+\frac{2}{n}\sum_{k=0}^{n-1}T(k)=n+\frac{2}{n}\sum_{k=0}^{n}T(k)-\frac{2T(n)}{n}$$

$$\geq n+\frac{2(n+1)}{n}T\left(\frac{n}{2}\right)-4\ln n\geq 2T\left(\frac{n}{2}\right)+n-4\ln n.$$

---

[15] [4] Here we assume that the elements to be sorted are distinct. Think of what might happen when the elements are not.

[4] 这里我们假设要排序的元素是互不相同的。想想当元素并非如此时会发生什么。

By the Master Theorem we know that $T(n) = 2T\left(\frac{n}{2}\right) + n - 4\ln n \Rightarrow T(n) = \Theta(n\log n)$. So our recurrence has a solution that grows no slower than that. (Besides the approach we discuss above, there are some other interesting ways to analyze the complexity of randomized quicksort algorithm. Interested readers can check out, e.g., [4, 58].)

根据主定理，我们知道 $T(n) = 2T\left(\frac{n}{2}\right) + n - 4\ln n \Rightarrow T(n) = \Theta(n\log n)$。所以我们的递归式有一个增长速度不慢于此的解。(除了我们上面讨论的方法，还有一些其他有趣的方法来分析随机快速排序算法的复杂度。感兴趣的读者可以查阅，例如，[4, 58]。)

Now let us look at another algorithm, min-cut algorithm. Suppose we get a simple, connected graph. Recall a cut edge in this graph is simply an edge that divides the graph into two components. A cut is a subset of edges that divide the graph into at least two components. A min-cut is a cut that consists of the smallest possible number of edges. Our problem is to use an algorithm to find one.

现在让我们来看另一种算法，最小割算法。假设我们有一个简单的连通图。回想一下，这个图中的割边就是将图分成两个部分的边。割是将图分成至少两个部分的边的子集。最小割是由尽可能少的边组成的割。我们的问题是使用一种算法来找到一个最小割。

A standard deterministic algorithm for this problem is to find a maximum flow in the graph, using, e.g., the Ford-Fulkerson Algorithm, but the time complexity would not be very satisfactory (see Problem 231). If you need a better algorithm, you may want to take the following observation into consideration: Since a min-cut is of the smallest size among all cuts, when you pick a random edge from the graph, it is quite likely to be outside of the min-cut. More precisely, you can use the following algorithm, designed by Karger [6

解决这个问题的一个标准确定性算法是在图中找到一个最大流，例如使用福特 - 富尔克森算法，但时间复杂度不会很令人满意(见问题 231)。如果你需要一个更好的算法，你可能需要考虑以下观察结果:由于最小割是所有割中规模最小的，当你从图中随机选择一条边时，它很可能不在最小割中。更准确地说，你可以使用以下由卡格尔设计的算法[6

Algorithm Karger-Min-cut $((V, E))$

算法 卡格尔 - 最小割 $((V, E))$

Repeat until $|V| \leq 2$

重复直到 $|V| \leq 2$

pick an edge $(u, v) \in E$ uniformly at random;

随机均匀地选择一条边 $(u, v) \in E$；

$$E \leftarrow E - \{(u, v)\}$$

merge u and v into a single vertex, and update V;

将 u 和 v 合并为一个顶点，并更新 V；

return E.

返回 E。

In each iteration, the probability of an edge in a particular min-cut being picked is no more than $\frac{k}{m}$, where k is the size of the min-cut and m is the total number of remaining edges. Since k is small, we hope to be so lucky that the algorithm never picks an edge from the min-cut before its termination. Will we really be so lucky? Let us proceed to formal analysis of the algorithm.

在每次迭代中，特定最小割中的一条边被选中的概率不超过 $\frac{k}{m}$，其中 k 是最小割的规模， m 是剩余边的总数。由于 k 很小，我们希望如此幸运，以至于算法在终止前从未从最小割中选择过边。我们真的会这么幸运吗？让我们继续对该算法进行形式化分析。

16

Lemma 6.2.2 The output of Karger's algorithm is a cut. It is a min-cut if the min-cut's edges were never picked during the execution of the algorithm.

引理 6.2.2 卡格尔算法的输出是一个割。如果在算法执行过程中最小割的边从未被选中，那么它就是一个最小割。

Lemma 6.2.3 The size of a min-cut is no more than $\frac{2m}{n}$, where n is the number of vertices and m is the number of edges.

引理 6.2.3 最小割的规模不超过 $\frac{2m}{n}$，其中 n 是顶点的数量， m 是边的数量。

Proof: Clearly, all edges incident with a vertex is a cut. Thus for each vertex v ,

证明:显然，与一个顶点相关联的所有边构成一个割。因此对于每个顶点 v ，

$$k \leq \deg(v),$$

where k is the size of a min-cut. On the other hand, we have

其中 k 是最小割的规模。另一方面，我们有

$$\sum_{v \in V} \deg(v) = 2m$$

Combining the above, we get that $kn \leq 2m \Rightarrow k \leq \frac{2m}{n}$.

综合上述内容，我们得到 $kn \leq 2m \Rightarrow k \leq \frac{2m}{n}$ 。

Theorem 6.2.1 During an execution of the Karger's algorithm, the probability of a min-cut's edges never being picked is no less than $\frac{2}{n(n-1)}$.

---

[16] [5] Why can we make this assumption? See Problem 230

[5] 我们为什么可以做这个假设？见问题 230

[6] David Karger (1967-) is an American computer scientist. He obtained his PhD from Stanford, where he met his wife, a novelist. He made numerous contributions to many areas of computer science.

[6] 大卫·卡格尔(1967 -)是一位美国计算机科学家。他在斯坦福大学获得博士学位，在那里他遇到了他的妻子，一位小说家。他在计算机科学的许多领域都做出了众多贡献。

定理 6.2.1 在执行卡格尔算法的过程中，最小割的边从未被选中的概率不小于 $\frac{2}{n(n-1)}$。

Proof: Lemma 6.2.3 tells us that during the first iteration, the probability of a particular min-cut's edge not being picked is $1-\frac{k}{m} \geq 1-\frac{2}{n}$. In the second iteration, since the number of vertices is less by 1, this probability lower bound shrinks to $1-\frac{2}{n-1}$ ... Putting together all iterations, we get that

证明:引理 6.2.3 告诉我们，在第一次迭代中，特定最小割的边未被选中的概率是 $1-\frac{k}{m} \geq 1-\frac{2}{n}$。在第二次迭代中，由于顶点数量减少了 1，这个概率下限缩小到 $1-\frac{2}{n-1}$ ...。综合所有迭代，我们得到

$$\Pr[\text{ edges in min-cut never picked }] \geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right) \ldots \left(1-\frac{2}{3}\right)=\frac{2}{n(n-1)} .$$

When n is large, the probability lower bound can be quite small, nearly zero. In fact, even when n is small (like 5), we are not happy with it as well (0.1 in the case of $n=5$). In reality, we definitely want a much better guarantee of success. In order to get such guarantee, we repeat Karger's algorithm for a number of times, and use the output with the smallest cardinality as the min-cut we find. For how many times do we have to repeat the algorithm? We leave it as homework (see Problem 232).

当 n 很大时，概率下限可能会非常小，几乎为零。实际上，即使 n 很小(比如 5)，我们对此也不满意(对于 $n=5$ 来说是 0.1)。实际上，我们肯定希望有更好的成功保证。为了得到这样的保证，我们多次重复卡格尔算法，并使用基数最小的输出作为我们找到的最小割。我们需要重复算法多少次呢？我们把它留作作业(见问题 232)。

Karger and Stein [35] noticed the issue of low success probability, and realized that the algorithm picked too many edges. For instance, during the last iteration, the probability of picking an edge outside of the minimum cut is only $\geq \frac{1}{3}$, and thus the lower of the success probability is reduced by two thirds. In contrast, during the first iteration, the probability of picking the edge outside of the minimum cut is $\geq 1-\frac{2}{n}$. Therefore, by restricting the number of picked edges to about $n-\frac{n}{\sqrt{2}}$, we can guarantee that the lower bound of success probability is about $\frac{1}{2}$. (Why?) If interested in the improved algorithm, please read [35] for more details.

卡格尔和斯坦因[35]注意到了成功概率低的问题，并意识到算法选择了太多的边。例如，在最后一次迭代中，选择最小割之外的边的概率只有 $\geq \frac{1}{3}$，因此成功概率下限降低了三分之二。相比之下，在第一次迭代中，选择最小割之外的边的概率是 $\geq 1-\frac{2}{n}$。因此，通过将选择的边的数量限制在大约 $n-\frac{n}{\sqrt{2}}$，我们可以保证成功概率下限约为 $\frac{1}{2}$。(为什么？)如果对改进后的算法感兴趣，请阅读[35]以获取更多细节。

So far we have presented two examples of randomized algorithms, namely the random quicksort and Karger's min-cut. If you compare them side by side, you can see a key difference: While the random quicksort algorithm always returns a correct output, Karger's min-cut algorithm has a non-zero probability of failure. Based on this fundamental difference, theoretical computer scientists have defined Las Vegas and Monte Carlo algorithms.

到目前为止，我们已经介绍了两个随机算法的例子，即随机快速排序和卡格尔的最小割算法。如果你将它们并排比较，你会看到一个关键区别:虽然随机快速排序算法总是返回正确的输出，但卡格尔的最小割算法有非零的失败概率。基于这个根本区别，理论计算机科学家定义了拉斯维加斯算法和蒙特卡罗算法。

Definition 6.2.1 A Las Vegas algorithm is a randomized algorithm that always produces a correct output. A Monte Carlo algorithm is a randomized algorithm whose output can be wrong with a non-zero probability.

定义 6.2.1 拉斯维加斯算法是一种总是产生正确输出的随机算法。蒙特卡罗算法是一种其输出可能以非零概率错误的随机算法。

It is tempting to say we want Las Vegas algorithm only, not the error-prone Monte Carlo algorithms. However, Las Vegas algorithms usually come with a cost-their running time usually varies with the input. For instance, while the expected running time of random quick sort is always $\Theta(n\log n)$, the coefficients hidden by the big- Θ notation are affected by the input. Different inputs can lead to very different coefficients.

很容易想说我们只想要拉斯维加斯算法，而不是容易出错的蒙特卡罗算法。然而，拉斯维加斯算法通常会有代价——它们的运行时间通常会随输入而变化。例如，虽然随机快速排序的期望运行时间总是 $\Theta(n\log n)$ ，但大 Θ 记号隐藏的系数会受到输入的影响。不同的输入可能导致非常不同的系数。

If you didn't care too much about the running time, you could turn a Monte Carlo algorithm into a Las Vegas algorithm by repeatedly executing the Monte Carlo algorithm and checking the correctness of the output, as long as the problem to be solved is in the complexity class NP.

如果你不太关心运行时间，只要要解决的问题属于 NP 复杂度类，你可以通过反复执行蒙特卡罗算法并检查输出的正确性，将蒙特卡罗算法转换为拉斯维加斯算法。

Example 6.2.1 Clearly, Karger's algorithm is a Monte Carlo algorithm. Repeating it multiple times and comparing the outputs improves our chances of finding a smaller cut, but there is no guarantee that the smallest cut obtained from multiple runs is indeed the minimum. How can we turn this algorithm into a Las Vegas algorithm?

例 6.2.1 显然，卡格尔算法是一种蒙特卡罗算法。多次重复它并比较输出可以提高我们找到更小割的机会，但不能保证从多次运行中得到的最小割确实是最小的。我们如何将这个算法转换为拉斯维加斯算法呢？

Solution: Run Karger's algorithm and compare its output to one computed using a deterministic maximum flow algorithm such as Ford-Fulkerson. If the two results match, we accept the output as correct for our Las Vegas algorithm. Otherwise, we repeat the above process.

解决方案:运行卡格尔算法，并将其输出与使用确定性最大流算法(如福特-富尔克森算法)计算的结果进行比较。如果两个结果匹配，我们接受该输出作为我们拉斯维加斯算法的正确结果。否则，我们重复上述过程。

While the above solution may seem impracticalafter all, why use a randomized algorithm if a deterministic one already yields the correct answer?—it is nevertheless technically valid. Can you think of a better example?

虽然上述解决方案可能看起来不切实际——毕竟，如果确定性算法已经能给出正确答案，为什么还要使用随机算法呢？——但它在技术上仍然是有效的。你能想出一个更好的例子吗？

Problem Set 18

问题集 18

Problem 230 Prove the function $T(\quad)$ studied in Lemma 6.2.1 is convex.

问题 230 证明引理 6.2.1 中研究的函数 $T(\quad)$ 是凸函数。

Problem 231 Suppose you are given a simple connected graph of n vertices as input. Design an algorithm that finds a min-cut, based on the famous Ford-Fulkerson algorithm for maximum flow. Assuming there are $\Theta(n^2)$ edges, what kind of worst-case time complexity can you get? You just need to present a time complexity analysis of your own algorithm. We are not asking you to prove your algorithm is optimal.

问题 231 假设输入给你的是一个具有 n 个顶点的简单连通图。基于著名的用于求最大流的福特-富尔克森算法，设计一种能找到最小割的算法。假设该图有 $\Theta(n^2)$ 条边，你能得到怎样的最坏情况时间复杂度？你只需给出你自己算法的时间复杂度分析。我们并不要求你证明你的算法是最优的。

Problem 232 Suppose we are given an upper bound $\epsilon(>0)$ of failure probability. In order to guarantee we get a min-cut with probability no less than $1-\epsilon$, for how many times should we repeat Karger's algorithm? Prove your answer is sufficient to guarantee the bound. You don't need to prove your answer is optimal.

问题 232 假设给定了一个失败概率的上界 $\epsilon(>0)$ 。为了保证我们以不小于 $1-\epsilon$ 的概率得到一个最小割，我们应该将卡格尔算法重复多少次？证明你的答案足以保证这个界限。你不需要证明你的答案是最优的。

Problem 233 Design a Monte Carlo algorithm for sorting. Analyze its error probability.

问题 233 设计一个用于排序的蒙特卡罗算法。分析其错误概率。

Problem 234 Prove that a simple graph of n vertices has at most $\binom{n}{2}$ min cuts.

问题 234 证明一个具有 n 个顶点的简单图最多有 $\binom{n}{2}$ 个最小割。

Problem 235 (Gatech CS7530 2004, Homework 1-1) Let n be a positive integer, not necessarily a power of 2. Describe an algorithm that generates a random permutation of $\{1,2,\ldots,n\}$ -the output of your algorithm should be distributed uniformly over all possible permutations. Remember that the only source of randomness your algorithm can use is a function rand( ), which returns a bit being equal to 0 with probability 0.5, being equal to 1 with probability 0.5 . The expected running time of your algorithm should be $O(n\log n)$.

问题 235 (佐治亚理工学院 CS7530 2004，作业 1 - 1)设 n 为一个正整数，不一定是 2 的幂。描述一种生成 $\{1,2,\ldots,n\}$ 的随机排列的算法 —— 你算法的输出应该在所有可能的排列上均匀分布。请记住，你的算法唯一能使用的随机源是一个函数 rand( )，它返回一个 0 或 1，返回 0 的概率为 0.5，返回 1 的概率为 0.5。你算法的期望运行时间应为 $O(n\log n)$ 。

Second, we show that, if three subtrees are pairwise intersecting, then there is a common vertex in all of them. Consider, e.g., (V_1,E_1), (V_2,E_2) , and (V_3,E_3) . Choose $u\in V_1\cap V_2, v\in V_1\cap V_3$, $w\in V_2\cap V_3$. Since there is no cycle, there is a single path connecting all these three vertices. W. L. O. G., assume that the path begins from u , goes through v , and ends at w . Since v is part of the only path connecting u and

w , it must be in V_2 -otherwise, u would be unreachable from w in (V_2, E_2) . Hence, v is in all of the three subtrees.

其次，我们证明，如果三个子树两两相交，那么它们有一个公共顶点。例如，考虑 $(V_1, E_1), (V_2, E_2)$ 、(V_3, E_3) 。选择 $u \in V_1 \cap V_2, v \in V_1 \cap V_3$ 、 $w \in V_2 \cap V_3$ 。由于不存在环，所以存在一条连接这三个顶点的唯一路径。不失一般性，假设该路径从 u 开始，经过 v ，并在 w 结束。由于 v 是连接 u 和 w 的唯一路径的一部分，它一定在 V_2 中 —— 否则，在 (V_2, E_2) 中 u 将无法从 w 到达。因此， v 在所有这三个子树中。

Third, we show a stronger proposition by induction: In the settings described by the problem, $(\cap_{1 \le i \le n} V_i, \cap_{1 \le i \le n} E_i)$ is a (non-empty) subtree.

第三，我们通过归纳法证明一个更强的命题:在问题所描述的设定中， $(\cap_{1 \le i \le n} V_i, \cap_{1 \le i \le n} E_i)$ 是一个(非空)子树。

The base case $(n = 3)$: we have shown that $(V_1 \cap V_2, E_1 \cap E_2)$ is a subtree. We have also shown that it intersects with (V_3, E_3) . So $(V_1 \cap V_2 \cap V_3, E_1 \cap E_2 \cap E_3)$ is a subtree.

基础情况 $(n = 3)$:我们已经证明 $(V_1 \cap V_2, E_1 \cap E_2)$ 是一个子树。我们也已经证明它与 (V_3, E_3) 相交。所以 $(V_1 \cap V_2 \cap V_3, E_1 \cap E_2 \cap E_3)$ 是一个子树。

Assuming the proposition is true for n subtrees, we now show that it is true for $n + 1$ subtrees. Let $V_n' = V_n \cap V_{n+1}, E_n' = E_n \cap E_{n+1}$. Clearly (V_n', E_n') is a (non-empty) subtree. Using the result of the base case, we also get that, for any $i (1 \le i \le n - 1), V_i \cap V_n' \ne \phi$. So we can apply the hypothesis to the n subtrees $(V_1, E_1), \cdots, (V_{n-1}, E_{n-1}), (V_n', E_n')$. The result is exactly what we need.

假设该命题对于 n 个子树成立，我们现在证明它对于 $n + 1$ 个子树也成立。设 $V_n' = V_n \cap V_{n+1}, E_n' = E_n \cap E_{n+1}$ 。显然 (V_n', E_n') 是一个(非空)子树。利用基础情形的结果，我们还得到，对于任意 $i (1 \le i \le n - 1), V_i \cap V_n' \ne \phi$ 。所以我们可以将假设应用于 n 个子树 $(V_1, E_1), \cdots, (V_{n-1}, E_{n-1}), (V_n', E_n')$ 。结果正是我们所需要的。

Solution: (Problem 197) First, observe that O_k contains a cycle $\{1,2, \cdots, k\}, \{k + 1, k + 2, \cdots, 2k\}, \{2k + 1,1, \cdots, k - 1\}, \cdots, \{2k^2 + 1,2k^2 + 2, \cdots, 2k^2 + k\}$, whose length is $2k + 1$. This implies that the chromatic number of O_k is ≥ 3 . On the other hand, we can color O_k using 3 colors only: First, assign color 1 to all vertices v such that $2k + 1 \in v$. All remaining vertices are subsets of cardinality k of $\{1,2, \cdots, 2k\}$. We pair every such subset v with its complement $\{1,2, \cdots, 2k\} \setminus v$. Then, for each pair, we assign colors 2 and 3 to the two subsets, respectively. Easy to see the result is a proper coloring.

解:(问题 197)首先，观察到 O_k 包含一个圈 $\{1,2, \cdots, k\}, \{k + 1, k + 2, \cdots, 2k\}, \{2k + 1,1, \cdots, k - 1\}, \cdots, \{2k^2 + 1,2k^2 + 2, \cdots, 2k^2 + k\}$ ，其长度为 $2k + 1$ 。这意味着 O_k 的色数为 ≥ 3 。另一方面，我们可以只用 3 种颜色给 O_k 着色:首先，给所有满足 v 使得 $2k + 1 \in v$ 的顶点分配颜色 1。所有剩余顶点都是 $\{1,2, \cdots, 2k\}$ 的基数为 k 的子集。我们将每个这样的子集 v 与其补集 $\{1,2, \cdots, 2k\} \setminus v$ 配对。然后，对于每一对，我们分别给这两个子集分配颜色 2 和 3。容易看出结果是一种恰当的着色。

Solution: (Problem 201) It is required that $\chi \ge \omega$. We show that there is no other restriction. The case of $\omega = 1$ or2is trivial and thus skipped. For the case of $\chi \ge \omega \ge 3$, we construct a graph G with chromatic number χ and clique number ω : By Tutte-Zykov-Mycielski Theorem, there is a graph H with chromatic number $\chi - \omega + 2$ that does not contain any triangle. Next, we add $\omega - 2$ new vertices to the graph, and make every new vertex adjacent to every vertex in H . The result is our graph G . Obviously, both the chromatic number and the clique number are increased by $\omega - 2$. So G is precisely what we need.

解:(问题 201)要求 $\chi \ge \omega$ 。我们证明没有其他限制。 $\omega = 1$ 为 1 或 2 的情形是平凡的，因此略过。对于 $\chi \ge \omega \ge 3$ 的情形，我们构造一个色数为 χ 且团数为 ω 的图 G :根据 Tutte-Zykov-Mycielski 定理，存

在一个色数为 $\chi-\omega+2$ 且不包含任何三角形的图 H 。接下来，我们向该图添加 $\omega-2$ 个新顶点，并使每个新顶点与 H 中的每个顶点相邻。结果就是我们的图 G 。显然，色数和团数都增加了 $\omega-2$ 。所以 G 正是我们所需要的。

Solution: (Problem 226) We show it's FALSE by contradiction. (A pseudo-proof by contradiction could be: If the proposition is true, then for every Turing Machine T and input x , we can decide whether T halts on x . To this end, we construct a new Turing Machine D , which simulates T and T' in turn, both on input x . Clearly one of the simulations will terminate. At that time, D also terminates. It outputs 1 if the first simulation terminates; it outputs 0 if the second terminates. The above is completely wrong, because, for an arbitrary Turing Machine T from the input, we would have no knowledge about the corresponding Turing Machine T' , even though we were confident that T' exists. There is no way for us to simulate the execution of T' .)

解决方案:(问题 226)我们用反证法证明其为假。(一个错误的反证法可能是:如果该命题为真，那么对于每一个图灵机 T 和输入 x ，我们都能判定 T 在 x 上是否停机。为此，我们构造一个新的图灵机 D ，它在输入 x 上依次模拟 T 和 T' 。显然，其中一个模拟会终止。此时， D 也会终止。如果第一个模拟终止，它输出 1；如果第二个模拟终止，它输出 0。以上完全错误，因为对于来自输入的任意一个图灵机 T ，我们对相应的图灵机 T' 一无所知，尽管我们确信 T' 存在。我们无法模拟 T' 的执行。)

Consider a Turing Machine T , which halts on input x if and only if $x=(t,y)$, where t is a Turing Machine and y is an input such that t halts on y . (Note that T definitely exists. All it does is to simulate t on input y and see whether it halts. When the simulated computation terminates, T halts. Otherwise, either the input x is not of the appropriate form, or the simulated computing does not terminate. In both cases, T does not halt.)

考虑一个图灵机 T ，当且仅当 $x=(t,y)$ 时，它在输入 x 上停机，其中 t 是一个图灵机， y 是一个输入，使得 t 在 y 上停机。(注意， T 肯定存在。它所做的就是在输入 y 上模拟 t ，并查看它是否停机。当模拟计算终止时， T 停机。否则，要么输入 x 不是合适的形式，要么模拟计算不会终止。在这两种情况下， T 都不停机。)

By the proposition, there exists T' that halts if and only if T does not halt. Equivalently, T' halts on input x if x is not of the appropriate form, or if $x=(t,y)$ where t does not halt on y . Given T' , we now construct a Turing Machine T'' that, on input $x=(t,y)$, decides whether Turing Machine t halts on input y. T'' simulates the execution of Turing Machine t on input y , and also simulates the execution of Turing Machine T' on input $(t,y)^{13}T''$ runs one simulation for one step, and then switches to the other for one step, and then switches back... Clearly, one of the two simulations will terminate. At that time, T'' also terminates. It outputs 1 if the first simulation terminates; it outputs 0 if the second terminates.

根据该命题，存在一个 T' ，当且仅当 T 不停机时它停机。等价地说，如果 x 不是合适的形式，或者如果 $x=(t,y)$ (其中 t 在 y 上不停机)，那么 T' 在输入 x 上停机。给定 T' ，我们现在构造一个图灵机 T'' ，它在输入 $x=(t,y)$ 上判定图灵机 t 在输入 $y.T''$ 上是否停机，模拟图灵机 t 在输入 y 上的执行，并且还模拟图灵机 T' 在输入 $(t,y)^{13}T''$ 上的执行，一步进行一个模拟，然后切换到另一个模拟一步，然后再切换回来……显然，两个模拟中的一个会终止。此时， T'' 也会终止。如果第一个模拟终止，它输出 1；如果第二个模拟终止，它输出 0。

We know T'' should not exist, due to the undecidability of halting. Contradiction.

由于停机问题的不可判定性，我们知道 T'' 不应该存在。矛盾。

Solution: (Problem 239) Let n be the length (i.e., the number of bits) of the description of an isomorphism. We construct a pseudo-ZK proof system(P, V)for proving G is isomorphic to G' :

解决方案:(问题 239)设 n 为同构描述的长度(即位数)。我们构造一个用于证明 G 与 G' 同构的伪零知识证明系统(P, V):

- The verifier V picks x from n -bit strings uniformly at random, and sends x to the prover P .
- 验证者 V 从 n 位字符串中均匀随机地选取 x ，并将 x 发送给证明者 P 。
- P computes $\theta' = \theta \oplus x$, and sends it to V , where θ is the isomorphism from G to G' .
- P 计算 $\theta' = \theta \oplus x$ ，并将其发送给 V ，其中 θ 是从 G 到 G' 的同构。
- P sends θ' to V .
- P 将 θ' 发送给 V 。
- V checks that $\theta' \oplus x$ is an isomorphism from G to G' . If that's the case, then V accepts; otherwise, V rejects.
- V 检查 $\theta' \oplus x$ 是否是从 G 到 G' 的同构。如果是这样，那么 V 接受；否则， V 拒绝。

Clearly, the proof is not PZK, because in fact, it completely reveals θ to V . However, it meets the requirement of pseudo-ZKness, because the only message received by V, θ' , is uniformly distributed over n -bit strings. The simulator has no difficulty in generating a random bit string that follows exactly the same distribution.

显然，该证明不是完美零知识证明(PZK)，因为实际上它完全向 θ 至 V 揭示了 θ 。然而，它满足了准零知识的要求，因为 V, θ' 接收到的唯一消息在 n 位字符串上是均匀分布的。模拟器可以毫无困难地生成一个遵循完全相同分布的随机位串。

17

6.3 Zero-Knowledge Proof: a Cryptographic Primitive

6.3 零知识证明:一种密码原语

A typical misunderstanding of cryptography is that it is all about encrypting and decrypting data, which is completely wrong. Modern cryptography is actually so powerful that it allows us to do something one can hardly believe, like proving a statement to you without letting you know anything. The latter is called Zero-Knowledge Proof, a very important primitive in modern cryptography.

对密码学的一种典型误解是，它只涉及对数据进行加密和解密，这是完全错误的。现代密码学实际上非常强大，它使我们能够做一些几乎令人难以置信的事情，比如在不让你知道任何信息的情况下向你证明一个陈述。后者被称为零知识证明，它是现代密码学中一个非常重要的原语。

---

17 13 Why can T'' simulate the execution of T' ? Because T is something we assume exists, and it does not depend on any input. Therefore we can construct T'' based on T' .

13 为什么 T'' 可以模拟 T' 的执行？因为 T 是我们假设存在的东西，并且它不依赖于任何输入。因此我们可以基于 T' 构造 T'' 。

Since Zero-Knowledge Proofs are part of so called interactive proofs, we first define interactive proof systems.

由于零知识证明是所谓交互式证明的一部分，我们首先定义交互式证明系统。

Definition 6.3.1 An interactive proof system(P, V)consists of a pair of probabilistic polynomial-time algorithms [7] , prover P and verifier V , that execute in turn with a common input x (the statement that needs to proved). Each time P (resp., V) stops, it sends a message to V (resp., P) before the latter resumes its execution. At the beginning, P starts execution first; at the end, V finishes its execution and generates an output, which is either "accept" (meaning V is convinced that x is true) or "reject" (meaning V is not convinced).

定义 6.3.1 一个交互式证明系统 (P, V) 由一对概率多项式时间算法 [7] 组成，即证明者 P 和验证者 V ，它们使用公共输入 x (需要证明的陈述)依次执行。每次 P (分别地， V)停止时，它会向 V (分别地，在 P)恢复执行之前)发送一条消息。开始时， P 首先开始执行；结束时， V 完成其执行并生成一个输出，该输出要么是 “接受”(意味着 V 确信 x 为真)，要么是 “拒绝”(意味着 V 不确信)。

The purpose of estabslishing an interactive proof system is to prove a proposition. But does the interactive proof system do its job well? We have the following standard for it.

建立交互式证明系统的目的是证明一个命题。但是交互式证明系统能很好地完成它的工作吗？对此我们有以下标准。

Definition 6.3.2 Let $P(n)$ be the probability of an event depending on a security parameter n . We say $P(n)$ is negligible if for all (positively valued) polynomial $f(n)$, we have $f(n) = O\left(\frac{1}{P(n)}\right)$. We say $P(n)$ is a high probability if $1 - P(n)$ is negligible.

定义 6.3.2 设 $P(n)$ 是一个依赖于安全参数 n 的事件的概率。如果对于所有(正值)多项式 $f(n)$ ，我们有 $f(n) = O\left(\frac{1}{P(n)}\right)$ ，那么我们说 $P(n)$ 是可忽略的。如果 $1 - P(n)$ 是可忽略的，那么我们说 $P(n)$ 是高概率的。

In cryptography, the security parameter n is often the length of key. We measure the running time of our algorithms using functions of n ; we also measure the running time of adversary algorithms using functions of n . Further, we measure the success probability of our algorithms using functions of n , hoping it to be a high probability, and measure the success probability of adversary algorithms using functions of n , hoping it to be negligible.

在密码学中，安全参数 n 通常是密钥的长度。我们使用 n 的函数来衡量我们算法的运行时间；我们也使用 n 的函数来衡量对手算法的运行时间。此外，我们使用 n 的函数来衡量我们算法的成功概率，希望它是高概率的，并使用 n 的函数来衡量对手算法的成功概率，希望它是可忽略的。

Definition 6.3.3 An interactive proof system(P, V)for a proposition S is complete if whenever S is true, V outputs "accept" with high probability; it is sound if whenever S is false, for all (cheating prover) $P^{\star}, V$ outputs "reject" with high probability.

定义 6.3.3 对于一个命题 S 的交互式证明系统 (P, V) 是完备的，如果每当 S 为真时， V 以高概率输出 “接受”；它是可靠的，如果每当 S 为假时，对于所有(作弊的证明者) $P^{\star}, V$ 以高概率输出 “拒绝”。

Please be aware of the subtle difference between the definitions of completeness and soundness. While for completeness, we only care about the situation in which both parties are honest, for soundness, we allow the prover to be arbitrarily cheating. The interactive proof system is considered sound only when V can defend it against all possible cheating strategies. (Please think of why there is such a difference.)

请注意完备性和可靠性定义之间的细微差别。对于完备性，我们只关心双方都诚实的情况，而对于可靠性，我们允许证明者任意作弊。只有当 V 能够抵御所有可能的作弊策略时，交互式证明系统才被认为是可靠的。(请思考为什么会有这样的差异。)

18

Example 6.3.1 Construct an interactive proof system(P, V)for proving input x is a prime number, in each of the cases below.

例 6.3.1 在以下每种情况下，构造一个用于证明输入 x 是质数的交互式证明系统 (P, V)。

1.(P, V)must be complete and sound.

1. (P, V) 必须是完备且可靠的。

2.(P, V)must be complete, but not sound.

2. (P, V) 必须是完备的，但不可靠。

3.(P, V)must be sound, but not complete.

3. (P, V) 必须是可靠的，但不完备。

4.(P, V)must be neither complete nor sound.

4.(P，V)既不能是完备的也不能是可靠的。

The solution is too easy and thus skipped.

这个解决方案太简单了，因此略过。

Example 6.3.1 is not the most interesting, because the involved proof is not a "proof" in the common sense-in your solution to Example 6.3.1, P does not really need to do anything to convince V . The reason is that, in this problem, there is no secret about x and thus V can decide whether x is a prime number by itself. Below we provide a further example in which there is a secret, and P needs to convince V about something related to the secret.

示例 6.3.1 不是最有趣的，因为其中涉及的证明并非通常意义上的“证明”——在你对示例 6.3.1 的解答中， P 实际上无需做任何事情就能让 V 信服。原因在于，在这个问题中，关于 x 没有秘密，因此 V 自身就能判定 x 是否为质数。下面我们给出一个更深入的示例，其中存在一个秘密，并且 P 需要让 V 相信与该秘密相关的某些事情。

Example 6.3.2 Construct a complete and sound interactive proof system(P, V)for proving input (G, G') is a pair of isomorphic graphs.

示例 6.3.2 构建一个完备且可靠的交互式证明系统(P，V)，用于证明输入 (G, G') 是一对同构图。

---

18 [7] This definition is semiformal at best, since "algorithm" is not a well defined mathematical term. A formal definition, which occupies a few pages in [25], is based on Turing machines, rather than on algorithms.

[7] 这个定义充其量只是半形式化的，因为“算法”不是一个定义明确的数学术语。一个基于图灵机而不是算法的形式化定义在 [25] 中占了几页篇幅。

Solution: In this example problem, the secret is the isomorphism between the two graphs. Bear in mind that no known algorithm can decide whether the two graphs are isomorphic or not in polynomial time. Therefore, V can't decide by itself whether G is isomorphic to G', which makes it necessary for P to convince V that the two graphs are isomorphic.

解决方案:在这个示例问题中，秘密是两个图之间的同构关系。请记住，目前还没有已知算法能够在多项式时间内判定两个图是否同构。因此，V 自身无法判定 G 是否与 G' 同构，这就使得 P 有必要让 V 相信这两个图是同构的。

Assume P, the prover,"knows" the isomorphic map θ. Hence, P simply sends θ to V, who checks $\theta(G) = G'$. If the identity holds, V outputs "accept"; otherwise "it outputs "reject."

假设证明者 P "知道"同构映射 θ。于是，P 只需将 θ 发送给验证者 V，由 V 进行检查。如果恒等式成立，V 输出"接受"；否则输出"拒绝"。

In the above example, we constructed an interactive proof that is both complete and sound. However, in order to prove the proposition, the prover has to reveal his knowledge of isomorphic map to the verifier V. In many applications, this is undesirable. Usually we wish that the verifier should learn as little as possible, ideally nothing, from the interactive proof.

在上述示例中，我们构建了一个既完备又可靠的交互式证明。然而，为了证明该命题，证明者必须向验证者 V 揭示他对同构映射的了解。在许多应用中，这是不可取的。通常我们希望验证者从交互式证明中了解到的信息越少越好，理想情况下是一无所知。

Definition 6.3.4 In an interactive proof system, the view of a party consists of its input, messages it has received, and all its coin flips.

定义 6.3.4 在交互式证明系统中，一方的视图由其输入、接收到的消息以及所有的抛硬币结果组成。

Everybody understands what is the input of a party. The received messages are also easy to understand- for P, they are the messages sent by V; for V, they are the messages sent by P. The only thing that might cause confusion is the coin flips. What are they?

每个人都明白一方的输入是什么。接收到的消息也很容易理解——对于 P 来说，它们是 V 发送的消息；对于 V 来说，它们是 P 发送的消息。唯一可能引起混淆的是抛硬币结果。它们是什么呢？

When a randomized algorithm executes, in some of the steps it must obtain random numbers. For instance, when the random quick sort algorithm executes, it must get a random number uniformly distributed in $\{1,2,\dots,n\}$. It does so through "flipping coins". Since each coin flip gives us only a single random bit, in order to choose a random number uniformly distributed in $\{1,2,\dots,2^k\}$, we need to flip k coins. The results of these coin flips are part of the view defined above.

---

[19] [8] A natural question is what if G is isomorphic to G' but P sends an incorrect isomorphism θ? Wouldn't that result in an output of "reject"? Well, that's possible in principle. However, notice that the definition of completeness requires both parties to be honest, i.e., to follow the protocol. So if P deviates from the protocol and sends a wrong isomorphism, even if V rejects, the interactive proof is still complete.

[8] 一个自然的问题是，如果 G 与 G' 同构，但 P 发送了一个错误的同构映射 θ 会怎样？这不会导致输出"拒绝"吗？嗯，原则上是有可能的。然而，请注意，完备性的定义要求双方都是诚实的，即遵循协议。所以，如果 P 偏离协议并发送一个错误的同构映射，即使 V 拒绝，交互式证明仍然是完备的。

当一个随机算法执行时，在某些步骤中它必须获取随机数。例如，当随机快速排序算法执行时，它必须获得一个在 $\{1,2,\dots,n\}$ 中均匀分布的随机数。它通过“抛硬币”来做到这一点。由于每次抛硬币只给我们一个随机位，为了选择一个在 $\{1,2,\dots,2^k\}$ 中均匀分布的随机数，我们需要抛 k 次硬币。这些抛硬币的结果是上述定义的视图的一部分。

Definition 6.3.5 An interactive proof system(P, V)is Perfectly Zero-Knowledge(PZK)if for all cheating verifier $V^\star$, there exists a probabilistic polynomial-time simulator σ that takes $V^\star$'s input as its own input and completes a simulation successfully with high probability. Under the condition that the simulation is deemed successful, σ's output must have a distribution identical to that of $V^\star$'s view.

定义 6.3.5 一个交互式证明系统(P，V)是完美零知识(PZK)的，如果对于所有作弊的验证者 $V^\star$，存在一个概率多项式时间模拟器 σ，它将 $V^\star$ 的输入作为自己的输入，并以高概率成功完成模拟。在模拟被视为成功的条件下，σ 的输出分布必须与 $V^\star$ 的视图分布相同。

The above definition is quite smart in that we have sidestepped the difficulty of defining an algorithm's knowledge. Instead, we observe that if the verifier's view (everything the verifier sees) can always be simulated by an algorithm that has never seen any secret of the prover, then we can safely say the interactive proof conveys zero knowledge to the verifier. This kind of definition is called the simulation paradigm, and is very frequently used in cryptography.

上述定义非常巧妙，因为我们避开了定义算法知识的困难。相反，我们观察到，如果验证者的视图(验证者所看到的一切)总是可以由一个从未见过证明者任何秘密的算法来模拟，那么我们就可以有把握地说交互式证明向验证者传达了零知识。这种定义被称为模拟范式，在密码学中非常常用。

In Definition 6.3.5, just like in the earlier definition of soundness, we have to consider an arbitrary cheating verifier $V^\star$, because the verifier might cheat, and if so, the cheating behavior may affect its view. Hence, we require that not just an honest verifier, but any cheating verifier, should get absolutely "zero" knowledge out of the interactive proof system.

在定义 6.3.5 中，就像在早期的可靠性定义中一样，我们必须考虑任意的作弊验证者 $V^\star$，因为验证者可能会作弊，如果是这样，作弊行为可能会影响其视图。因此，我们要求不仅诚实的验证者，而且任何作弊验证者，都应该从交互式证明系统中绝对获得“零”知识。

Example 6.3.3 Construct a complete, sound, and PZK interactive proof system(P, V)for proving input (G, G') is a pair of isomorphic graphs.

示例 6.3.3 构建一个完整、可靠且具有完美零知识的交互式证明系统(P, V)，用于证明输入 (G, G') 是一对同构图。

Solution: Suppose the input is (G, G') such that $\theta(G) = G'$. Repeat the following process for n times (where n is the security parameter): P generates a random isomorphic map θ' and sends $G'' = \theta'(G)$ to V. V challenges P by asking, with probability $\frac{1}{2}$, for the isomorphic map between G and G'', and with the remaining probability $\frac{1}{2}$, for the isomorphic map between G' and G''. P replies with either θ' or $\theta^{-1}\theta'$, depending on which isomorphic map V asked for. V proceeds to the next round if the map is correct, and rejects otherwise. If all rounds have been completed successfully, V accepts.

解决方案:假设输入为 (G, G') 使得 $\theta(G) = G'$。重复以下过程 n 次(其中 n 是安全参数): P 生成一个随机同构映射 θ' 并将 $G'' = \theta'(G)$ 发送给 V. V 以概率 $\frac{1}{2}$ 通过询问 P 关于 G 和 G'' 之间的同构映射来挑战，其余概率 $\frac{1}{2}$ 询问 G' 和 G''. P 之间的同构映射 V 根据所询问的同构映射回复 θ' 或 $\theta^{-1}\theta'$。如果映射正确，V 进入下一轮，否则拒绝。如果所有轮都成功完成，V 接受。

A somewhat relaxed version of PZK is Statistically Zero Knowledge (SZK), which is based on the statistical difference of two random variables.

完美零知识(PZK)的一个稍微宽松的版本是统计零知识(SZK)，它基于两个随机变量的统计差异。

Definition 6.3.6 Suppose X and Y are both random variables with range R . The statistical difference between X and Y is

定义 6.3.6 假设 X 和 Y 都是值域为 R 的随机变量。 X 和 Y 之间的统计差异为

$$\Delta(X,Y)=\frac{1}{2}\sum_{r\in R}|\Pr[X=r]-\Pr[Y=r]|.$$

We have to emphasize that the above definition came from statisticians. If the definition were given by a computer scientist, the constant factor of $\frac{1}{2}$ might not be included.

我们必须强调上述定义来自统计学家。如果这个定义由计算机科学家给出，可能不会包含 $\frac{1}{2}$ 这个常数因子。

Definition 6.3.7 An interactive proof system(P, V)is Statistically Zero-Knowledge(SZK)if for all cheating verifier $V^{\star}$, there exists a probabilistic polynomial-time simulator σ that takes $V^{\star}$'s input as its own input, such that the statistical difference between σ 's output and $V^{\star}$'s view is negligible.

定义 6.3.7 一个交互式证明系统(P, V) 是统计零知识(SZK)的，如果对于所有作弊验证者 $V^{\star}$ ，存在一个概率多项式时间模拟器 σ ，它将 $V^{\star}$ 的输入作为自己的输入，使得 σ 的输出与 $V^{\star}$ 的视图之间的统计差异可以忽略不计。

Example 6.3.4 Show that every PZK proof is also an SZK proof.

示例 6.3.4 证明每个 PZK 证明也是一个 SZK 证明。

Solution: There is no doubt that two identical probabilitic distributions have a zero (and thus negligible) statistical difference. However, the definition of PZK allows the simulator to fail with a negligible probability, while the definition of SZK does not. What can we do with this subtle issue?

解决方案:毫无疑问，两个相同的概率分布具有零(因此可以忽略不计)的统计差异。然而，PZK 的定义允许模拟器以可忽略的概率失败，而 SZK 的定义不允许。对于这个微妙的问题我们该怎么办？

Assume we have a PZK proof, together with its PZK simulator σ , which fails with a negligible, but not zero, probability. Now we are supposed to construct a simulator σ' that does not fail at all. The key observation is that we can allow σ' to make a random output whenever σ fails. In this way, σ' never needs to fail, but its output can be different with a negligible probability, as long as σ' behaves in exactly the same way as σ in all other cases. This clearly meets our requirement of SZK.

假设我们有一个 PZK 证明，以及它的 PZK 模拟器 σ ，它以可忽略但不为零的概率失败。现在我们要构造一个根本不会失败的模拟器 σ' 。关键的观察是，我们可以允许 σ' 在 σ 失败时进行随机输出。这样，σ' 永远不需要失败，但只要 σ' 在所有其他情况下与 σ 的行为完全相同，其输出就可以以可忽略的概率不同。这显然符合我们对 SZK 的要求。

When SZK is further relaxed, we get Computational Zero Knowledge (CZK), which is based on computational indistinguishability.

当 SZK 进一步放宽时，我们得到计算零知识(CZK)，它基于计算不可区分性。

Definition 6.3.8 Random variables X and Y are computationally indistinguishable from each other if for all probabilistic polynomial-time distinguisher 1. $|\Pr[D(X) = 1] - \Pr[D(Y) = 1]|$ is negligible. In this case, we usually write $X \overset{C}{=} Y$.

定义 6.3.8 如果对于所有概率多项式时间区分器 X 来说，可忽略不计，那么随机变量 X 和 Y 在计算上是不可区分的。在这种情况下，我们通常写作 $X \overset{C}{=} Y$ 。

The intuition behind this definition is that, if there is any probabilistic polynomial-time algorithm that behaves significantly differently on the two involved random variables, then there must be a probabilistic polynomial-time distinguisher that outputs 1 with significantly different probabilities on these two inputs. (Why?)

这个定义背后的直觉是，如果存在任何概率多项式时间算法，其在两个相关随机变量上的行为有显著差异，那么必然存在一个概率多项式时间区分器，它在这两个输入上输出 1 的概率有显著差异。(为什么？)

20

Example 6.3.5 (Final Exam 2024-4) Suppose that G is a large cyclic group of size N . The famous Decisional Diffie-Hellman Assumption states that, if g is picked uniformly at random from the set of all generators, and if x, y, z are three independent random variables uniformly distributed over $\{0,1,2,\dots,N-1\}$, then $(g^x, g^y, g^{xy}) \overset{C}{=} (g^x, g^y, g^z)$.

例 6.3.5(2024 - 4 期末考试)假设 G 是一个大小为 N 的大循环群。著名的判定性迪菲 - 赫尔曼假设(Decisional Diffie - Hellman Assumption)指出，如果 g 是从所有生成元集合中随机均匀选取的，并且如果 x, y, z 是在 $\{0,1,2,\dots,N-1\}$ 上均匀分布的三个独立随机变量，那么 $(g^x, g^y, g^{xy}) \overset{C}{=} (g^x, g^y, g^z)$ 。

Suppose m_1 and $m_2 (m_1 \neq m_2)$ are fixed elements of G . Prove, under the Decisional Diffie-Hellman Assumption, that $(g^x, g^y, g^{xy}m_1) \overset{C}{=} (g^x, g^y, g^{xy}m_2)$.

假设 m_1 和 $m_2 (m_1 \neq m_2)$ 是 G 中的固定元素。在判定性迪菲 - 赫尔曼假设下，证明 $(g^x, g^y, g^{xy}m_1) \overset{C}{=} (g^x, g^y, g^{xy}m_2)$ 。

Solution: Easy to see that $(g^x, g^y, g^{xy}) \overset{C}{=} (g^x, g^y, g^z)$ implies $(g^x, g^y, g^{xy}m_1) \overset{C}{=} (g^x, g^y, g^z m_1)$; $(g^x, g^y, g^{xy}m_2) \overset{C}{=} (g^x, g^y, g^z m_2)$. Next, observe that the distributions of $(g^x, g^y, g^z m_1)$ and $(g^x, g^y, g^z m_2)$ are exactly the same as (g^x, g^y, g^z) .

解:容易看出 $(g^x, g^y, g^{xy}) \overset{C}{=} (g^x, g^y, g^z)$ 意味着 $(g^x, g^y, g^{xy}m_1) \overset{C}{=} (g^x, g^y, g^z m_1)$; $(g^x, g^y, g^{xy}m_2) \overset{C}{=} (g^x, g^y, g^z m_2)$ 。接下来，注意到 $(g^x, g^y, g^z m_1)$ 和 $(g^x, g^y, g^z m_2)$ 的分布与 (g^x, g^y, g^z) 完全相同。

Definition 6.3.9 An interactive proof system(P, V)is Computationally Zero-Knowledge(CZK) if for all cheating verifier $V^\star$, there exists a probabilistic polynomial-time simulator σ that takes $V^\star$'s input as its own input, such that σ 's output and $V^\star$'s view are computationally indistinguish- $able$.

---

[20] [9] A distinguisher is an algorithm that outputs only 0 or 1 .

[9] 区分器是一种只输出 0 或 1 的算法。

定义 6.3.9 一个交互式证明系统 (P, V) 是计算零知识的(Computationally Zero - Knowledge，CZK)，如果对于所有作弊验证者 $V^{\star}$，存在一个概率多项式时间模拟器 σ，它将 $V^{\star}$ 的输入作为自己的输入，使得 σ 的输出和 $V^{\star}$ 的视图在计算上不可区分 - $able$.

Example 6.3.6 Show that every SZK proof is also a CZK proof.

例 6.3.6 证明每个 SZK 证明也是一个 CZK 证明。

Solution: For all probabilistic polynomial-time distinguisher D , we have

解:对于所有概率多项式时间区分器 D，我们有

$$|\Pr[D(X)=1]-\Pr[D(Y)=1]| = \left|\sum_r \Pr[X=r, D(X)=1] - \sum_r \Pr[Y=r, D(Y)=1]\right|$$

$$= \left|\sum_r (\Pr[X=r]\Pr[D(X)=1 \mid X=r] - \Pr[Y=r]\Pr[D(Y)=1 \mid Y=r])\right|$$

$$= \left|\sum_r (\Pr[X=r]-\Pr[Y=r])\right| \Pr[D(Z)=1 \mid Z=r]$$

$$\leq 2\Delta(X,Y) .$$

Since the right side is negligible, the left side is also negligible.

由于右边是可忽略不计的，左边也是可忽略不计的。

Example 6.3.7 Suppose p, q are prime numbers such that $p=2q+1; g, h$ are both quadratic residues with respect to modulus p , i.e., there exist α, β such that $g \equiv \alpha^2 (\bmod p), h \equiv \beta^2 (\bmod p)$. Assuming $g \neq 1 (\bmod p)$ and $h \neq 1 (\bmod p)$, we can show that there exists $e (1 \leq e \leq q-1)$ such that $g^e \equiv h (\bmod p)$. We call e the discrete logarithm of h to base g . (Notice the similarities to, and differences from, the logarithm we learned in middle school!)

例 6.3.7 假设 p, q 为素数，使得 $p=2q+1; g, h$ 关于模 p 均为二次剩余，即存在 α, β 使得 $g \equiv \alpha^2 (\bmod p), h \equiv \beta^2$ (模 p)。假设 $g \neq 1 (\bmod p)$ 且 $h \neq 1 (\bmod p)$，我们可以证明存在 $e (1 \leq e \leq q-1)$ 使得 $g^e \equiv h (\bmod p)$ 。我们称 e 是以 g 为底 h 的离散对数。(注意它与我们中学所学对数的异同！)

For large p, g, h , computing discrete logarithm is considered infeasible. In other words, given p, g, h , we can safely assume that the probability of any probabilistic polynomial-time algorithm computing e correctly is negligible.

对于大的 p, g, h，计算离散对数被认为是不可行的。换句话说，给定 p, g, h，我们可以安全地假设任何概率多项式时间算法正确计算出 e 的概率是可忽略不计的。

Now suppose that p, g, h are all public and e is a prover's secret. Construct a sound, complete, and CZK proof of knowing e .

现在假设 p, g, h 都是公开的，而 e 是证明者的秘密。构造一个可靠、完备且 CZK 的关于知道 e 的证明。

Solution: Repeat the following process for n times:

解决方案:对 n 次重复以下过程:

The prover picks $f \in \{1,2,\ldots,q-1\}$ uniformly at random, computes $j = g^f \mod p$, and sends j to the verifier. The verifier chooses a uniformly random bit as their challenge. If the challenge is 0, then the prover has to send f to the verifier; the verifier accepts after checking $j \equiv g^f \pmod p$, and rejects in case the equation does not hold. If the challenge is 1, then the prover has to send $f' = (e+f) \mod q$ to the verifier; the verifier accepts after checking $hj \equiv g^{f'} \pmod p$, and rejects in case the equation does not hold.

证明者随机均匀地选取 $f \in \{1,2,\ldots,q-1\}$，计算 $j = g^f \mod p$，并将 j 发送给验证者。验证者随机均匀地选择一个比特作为他们的挑战。如果挑战为 0，那么证明者必须将 f 发送给验证者；验证者在检查 $j \equiv g^f \pmod p$ 后接受，若等式不成立则拒绝。如果挑战为 1，那么证明者必须将 $f' = (e+f) \mod q$ 发送给验证者；验证者在检查 $hj \equiv g^{f'} \pmod p$ 后接受，若等式不成立则拒绝。

In addition to being CZK, the above interactive proof is also a Proof of Knowledge (POK), because its objective is to show the prover has knowledge of the secret discrete logarithm. The formal treatment of POK is well beyond the scope of this lecture, but can be found, e.g., in [25]. Roughly speaking, an interactive proof is POK if there is a so called knowledge extractor that can observe its execution and magically figure out the secret knowledge.

除了是 CZK 之外，上述交互式证明也是一个知识证明(POK)，因为其目的是表明证明者知道秘密离散对数。POK 的形式化处理远远超出了本讲座的范围，但可以在例如[25]中找到。粗略地说，如果存在一个所谓的知识提取器，它可以观察交互式证明的执行过程并神奇地找出秘密知识，那么这个交互式证明就是 POK。

Why isn't POK contradictory to CZK? Because the knowledge extractor is much more powerful than any realistic algorithm. It can even manipulate time to some extent-more precisely, it can arbitrarily stop the interactive proof system at any point, and replay (part of) it from any state. In reality, no algorithm can do that.

为什么 POK 与 CZK 不矛盾？因为知识提取器比任何现实的算法强大得多。它甚至可以在某种程度上操纵时间——更准确地说，它可以在任何点任意停止交互式证明系统，并从任何状态重新播放(部分)它。在现实中，没有算法能做到这一点。

Problem Set 19

习题集 19

Problem 236 Find a negligible function $p(n)$ such that for all $b > 1, \frac{1}{b^n} = O\big(p(n)\big)$.

问题 236 找到一个可忽略函数 $p(n)$ 使得对于所有 $b > 1, \frac{1}{b^n} = O\big(p(n)\big)$。

Problem 237 An alternative definition of a complete interactive proof system requires that whenever S is true, V outputs "accept" with probability no less than $\frac{2}{3}$. In what sense is this alternative definition equivalent to ours? Prove your answer.

问题 237 完备交互式证明系统的另一种定义要求，只要 S 为真， V 输出“接受”的概率不小于 $\frac{2}{3}$。在何种意义上这种替代定义与我们的定义等价？证明你的答案。

Problem 238 Prove the interactive proof we constructed for Example 6.3.3 is indeed complete, sound, and PZK.

问题 238 证明我们为例 6.3.3 构造的交互式证明确实是完备、可靠且零知识的。

Problem 239 An interactive proof system(P, V)is pseudo-zero-knowledge if for all cheating verifier $V^{\star}$, there exists a probabilistic polynomial-time simulator σ that takes $V^{\star}$'s input as its own input and completes a simulation successfully with high probability. Under the condition that the simulation is deemed successful, σ's output must have a distribution identical to that of $V^{\star}$'s received messages.

问题 239 如果对于所有作弊验证者 $V^{\star}$，都存在一个概率多项式时间模拟器 σ，该模拟器将 $V^{\star}$ 的输入作为自己的输入，并以高概率成功完成模拟，那么交互式证明系统(P, V)就是拟零知识的。在模拟被视为成功的条件下， σ 的输出分布必须与 $V^{\star}$ 接收到的消息的分布相同。

Construct an interactive proof system that is pseudo-zero-knowledge but not PZK.

构造一个拟零知识但不是 PZK 的交互式证明系统。

Problem 240 Prove the interactive proof we constructed in Example 6.3.7 is indeed CZK.

问题 240 证明我们在示例 6.3.7 中构造的交互式证明确实是计算性零知识的。

Problem 241 Construct a complete, sound, and SZK proof for graph isomorphism that is not PZK.

问题 241 为图同构构造一个完整、可靠且统计性零知识的证明，且不是 PZK 。

Problem 242 Recall that a distinguisher is an algorithm that outputs only 0 or 1. If we replace "distinguisher" with "algorithm", we get an alternative definition:

问题 242 回想一下，区分器是一种只输出 0 或 1 的算法。如果我们将“区分器”替换为“算法”，就会得到一个替代定义:

Definition 6.3.10 Random variables X and Y are computationally indistinguishable from each other if for all probabilistic polynomial-time algorithm $A, \sum_v |\Pr[A(X) = v] - \Pr[A(Y) = v]|$ is negligible.

定义 6.3.10 如果对于所有概率多项式时间算法 $A, \sum_v |\Pr[A(X) = v] - \Pr[A(Y) = v]|$，随机变量 X 和 Y 在计算上是不可区分的，那么 $A, \sum_v |\Pr[A(X) = v] - \Pr[A(Y) = v]|$ 是可忽略的。

Are these two definitions equivalent? If yes, please prove your answer. If no, provide an example that satisfies one of them, but not the other.

这两个定义等价吗？如果是，请证明你的答案。如果不是，请提供一个满足其中一个定义但不满足另一个定义的例子。

Problem 243 Suppose x, y, u, v are random variables and their range is the set of s -bit strings. Is each of the following propositions true? If so, provide a proof. Otherwise, give a counter example.

问题 243 假设 x, y, u, v 是随机变量，其值域是 s 位字符串的集合。以下每个命题都是真的吗？如果是，请提供证明。否则，给出一个反例。

(a) If $x \overset{C}{=} y, u \overset{C}{=} v$, then $(x, u) \overset{C}{=} (y, v)$.

(a) 如果 $x \overset{C}{=} y, u \overset{C}{=} v$ ，那么 $(x,u) \overset{C}{=} (y,v)$ 。

(b) If $(x,u) \overset{C}{=} (y,v)$, then $x \overset{C}{=} y, u \overset{C}{=} v$.

(b) 如果 $(x,u) \overset{C}{=} (y,v)$ ，那么 $x \overset{C}{=} y, u \overset{C}{=} v$ 。