矩阵

本文介绍线性代数中一个非常重要的内容——矩阵（Matrix），主要讲解矩阵的性质、运算，以及矩阵乘法的一些应用。

向量与矩阵

在线性代数中，向量分为列向量和行向量。

> **Warning**
>
> 在中国台湾地区关于「列」与「行」的翻译，恰好与中国大陆地区相反。在 **OI Wiki** 按照中国大陆地区的习惯，采用列（column）与行（row）的翻译。

线性代数的主要研究对象是列向量，约定使用粗体小写字母表示列向量。在用到大量向量与矩阵的线性代数中，不引起混淆的情况下，在手写时，字母上方的向量记号可以省略不写。

向量也是特殊的矩阵。如果想要表示行向量，需要在粗体小写字母右上方写转置记号。行向量在线性代数中一般表示方程。

引入

矩阵的引入来自于线性方程组。与向量类似，矩阵体现了一种对数据「打包处理」的思想。

例如，将线性方程组：

$$\begin{cases} 7x_1 + 8x_2 + 9x_3 = 13 \\ 4x_1 + 5x_2 + 6x_3 = 12 \\ x_1 + 2x_2 + 3x_3 = 11 \end{cases}$$

一般用圆括号或方括号表示矩阵。将上述系数抽出来，写成矩阵乘法的形式：

$$\begin{pmatrix} 7 & 8 & 9 \\ 4 & 5 & 6 \\ 1 & 2 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 13 \\ 12 \\ 11 \end{pmatrix}$$

简记为：

$$Ax = b$$

即未知数列向量 x，左乘一个矩阵 A，得到列向量 b。这个式子可以认为是线性代数的基本形式。

线性代数主要研究的运算模型是内积。内积是先相乘再相加，是行向量左乘列向量，得到一个数的过程。

矩阵乘法是内积的拓展。矩阵乘法等价于左边矩阵抽出一行，与右边矩阵抽出一列进行内积，得到结果矩阵的对应元素，口诀「左行右列」。

当研究对象是右边的列向量时，矩阵乘法相当于对列向量进行左乘。在左乘的观点下，矩阵就是对列向量的变换，将矩阵乘法中右边矩阵的每一个列向量进行变换，对应地得到结果矩阵中每一个列向量。

矩阵可以对一个列向量进行变换，也可以对一组列向量进行「打包」变换，甚至可以对整个空间——即全体列向量进行变换。当矩阵被视为对整个空间变换的时候，也就脱离了空间，成为了纯粹变换的存在。

定义

对于矩阵 A，主对角线是指 $A_{i,i}$ 的元素。

一般用 I 来表示单位矩阵，就是主对角线上为 1，其余位置为 0。

同型矩阵

两个矩阵，行数与列数对应相同，称为同型矩阵。

方阵

行数等于列数的矩阵称为方阵。方阵是一种特殊的矩阵。对于「n 阶矩阵」的习惯表述，实际上讲的是 n 阶方阵。阶数相同的方阵为同型矩阵。

研究方程组、向量组、矩阵的秩的时候，使用一般的矩阵。研究特征值和特征向量、二次型的时候，使用方阵。

主对角线

方阵中行数等于列数的元素构成主对角线。

对称矩阵

如果方阵的元素关于主对角线对称，即对于任意的 i 和 j，i 行 j 列的元素与 j 行 i 列的元素相等，则将方阵称为对称矩阵。

对角矩阵

主对角线之外的元素均为 0 的方阵称为对角矩阵，一般记作：

$$\mathrm{diag}\{\lambda_1, \cdots, \lambda_n\}$$

式中的 $\lambda_1, \cdots, \lambda_n$ 是主对角线上的元素。

对角矩阵是对称矩阵。

如果对角矩阵的元素均为 1，称为单位矩阵，记为 I。只要乘法可以进行，无论形状，任何矩阵乘单位矩阵仍然保持不变。

三角矩阵

如果方阵主对角线左下方的元素均为 0，称为上三角矩阵。如果方阵主对角线右上方的元素均为 0，称为下三角矩阵。

两个上（下）三角矩阵的乘积仍然是上（下）三角矩阵。如果对角线元素均非 0，则上（下）三角矩阵可逆，逆也是上（下）三角矩阵。

单位三角矩阵

如果上三角矩阵 A 的对角线全为 1，则称 A 是单位上三角矩阵。如果下三角矩阵 A 的对角线全为 1，则称 A 是单位下三角矩阵。

两个单位上（下）三角矩阵的乘积仍然是单位上（下）三角矩阵，单位上（下）三角矩阵的逆也是单位上（下）三角矩阵。

运算

矩阵的线性运算

矩阵的线性运算分为加减法与数乘，它们均为逐个元素进行。只有同型矩阵之间可以对应相加减。

矩阵的转置

矩阵的转置，就是在矩阵的右上角写上转置「T」记号，表示将矩阵的行与列互换。

对称矩阵转置前后保持不变。

矩阵乘法

矩阵的乘法是向量内积的推广。

矩阵相乘只有在第一个矩阵的列数和第二个矩阵的行数相同时才有意义。

设 A 为 $P \times M$ 的矩阵，B 为 $M \times Q$ 的矩阵，设矩阵 C 为矩阵 A 与 B 的乘积，

其中矩阵 C 中的第 i 行第 j 列元素可以表示为：

$$C_{i,j} = \sum_{k=1}^{M} A_{i,k} B_{k,j}$$

在矩阵乘法中，结果 C 矩阵的第 i 行第 j 列的数，就是由矩阵 A 第 i 行 M 个数与矩阵 B 第 j 列 M 个数分别 **相乘再相加** 得到的。这里的 **相乘再相加**，就是向量的内积。乘积矩阵中第 i 行第 j 列的数恰好是乘数矩阵 A 第 i 个行向量与乘数矩阵 B 第 j 个列向量的内积，口诀为 **左行右列**

线性代数研究的向量多为列向量，根据这样的对矩阵乘法的定义方法，经常研究对列向量左乘一个矩阵的左乘运算，同时也可以在这里看出「打包处理」的思想，同时处理很多个向量内积。

矩阵乘法满足结合律，不满足一般的交换律。

利用结合律，矩阵乘法可以利用 快速幂 的思想来优化。

在比赛中，由于线性递推式可以表示成矩阵乘法的形式，也通常用矩阵快速幂来求线性递推数列的某一项。

优化

首先对于比较小的矩阵，可以考虑直接手动展开循环以减小常数。

可以重新排列循环以提高空间局部性，这样的优化不会改变矩阵乘法的时间复杂度，但是会得到常数级别的提升。

```
// 以下文的参考代码为例
mat operator*(const mat& T) const {
  mat res;
  for (int i = 0; i < sz; ++i)
    for (int j = 0; j < sz; ++j)
      for (int k = 0; k < sz; ++k) {
        res.a[i][j] += mul(a[i][k], T.a[k][j]);
        res.a[i][j] %= MOD;
      }
  return res;
}

// 不如
mat operator*(const mat& T) const {
  mat res;
  int r;
  for (int i = 0; i < sz; ++i)
    for (int k = 0; k < sz; ++k) {
      r = a[i][k];
      for (int j = 0; j < sz; ++j)
        res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
    }
  return res;
}
```

方阵的逆

方阵 A 的逆矩阵 P 是使得 $A \times P = I$ 的矩阵。

逆矩阵不一定存在。如果存在，可以使用 高斯消元 进行求解。

方阵的行列式

行列式是方阵的一种运算。

参考代码

一般来说，可以用一个二维数组来模拟矩阵。

```
struct mat {
  LL a[sz][sz];

  mat() { memset(a, 0, sizeof a); }

  mat operator-(const mat& T) const {
    mat res;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) {
        res.a[i][j] = (a[i][j] - T.a[i][j]) % MOD;
      }
    return res;
  }

  mat operator+(const mat& T) const {
    mat res;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) {
        res.a[i][j] = (a[i][j] + T.a[i][j]) % MOD;
      }
    return res;
  }

  mat operator*(const mat& T) const {
    mat res;
    int r;
    for (int i = 0; i < sz; ++i)
      for (int k = 0; k < sz; ++k) {
        r = a[i][k];
        for (int j = 0; j < sz; ++j)
          res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
      }
    return res;
  }

  mat operator^(LL x) const {
    mat res, bas;
    for (int i = 0; i < sz; ++i) res.a[i][i] = 1;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) bas.a[i][j] = a[i][j] % MOD;
    while (x) {
```

```
        if (x & 1) res = res * bas;
        bas = bas * bas;
        x >>= 1;
      }
      return res;
    }
  };
```

看待线性方程组的两种视角

看待矩阵 A，或者变换 A，有两种视角。

第一种观点：按行看，观察 A 的每一行。这样一来把 A 看作方程组。于是就有了消元法解方程的过程。

第二种观点：按列看，观察 A 的每一列。A 本身也是由列向量构成的。此时相当于把变换 A 本身看成了列向量组，而 x 是未知数系数，思考 A 当中的这组列向量能不能配上未知数，凑出列向量 b。

例如，文章开头的例子变为：

$$\begin{pmatrix}7\\4\\1\end{pmatrix}x_1+\begin{pmatrix}8\\5\\2\end{pmatrix}x_2+\begin{pmatrix}9\\6\\3\end{pmatrix}x_3=\begin{pmatrix}13\\12\\11\end{pmatrix}$$

解方程变为研究，是否可以通过调整三个系数 x，使得给定的三个基向量能够凑出结果的向量。

按列看比按行看更新颖。在按列看的视角下，可以研究线性无关与线性相关。

矩阵乘法的应用

矩阵加速递推

以 斐波那契数列（Fibonacci Sequence） 为例。在斐波那契数列当中，$F_1=F_2=1$，$F_i=F_{i-1}+F_{i-2}(i\geq 3)$。

如果有一道题目让你求斐波那契数列第 n 项的值，最简单的方法莫过于直接递推了。但是如果 n 的范围达到了 10^{18} 级别，递推就不行了，此时我们可以考虑矩阵加速递推。

根据斐波那契数列 递推公式的矩阵形式:

$$[F_{n-1}\quad F_{n-2}]\begin{bmatrix}1&1\\1&0\end{bmatrix}=[F_n\quad F_{n-1}]$$

定义初始矩阵 $\text{ans}=[F_2\quad F_1]=[1\quad 1],\text{base}=\begin{bmatrix}1&1\\1&0\end{bmatrix}$。那么，$F_n$ 就等于 $\text{ans}\text{base}^{n-2}$ 这个矩阵的第一行第一列元素，也就是 $[1\quad 1]\begin{bmatrix}1&1\\1&0\end{bmatrix}^{n-2}$ 的第一行第一列元素。

注意

矩阵乘法不满足交换律，所以一定不能写成 $\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}^{n-2}[1 \quad 1]$ 的第一行第一列元素。另外，对于 $n \le 2$ 的情况，直接输出 1 即可，不需要执行矩阵快速幂。

为什么要乘上 base 矩阵的 $n-2$ 次方而不是 n 次方呢？因为 F_1, F_2 是不需要进行矩阵乘法就能求的。也就是说，如果只进行一次乘法，就已经求出 F_3 了。如果还不是很理解为什么幂是 $n-2$，建议手算一下。

下面是求斐波那契数列第 n 项对 10^9+7 取模的示例代码（核心部分）。

```
constexpr int mod = 1000000007;

struct Matrix {
  int a[3][3];

  Matrix() { memset(a, 0, sizeof a); }

  Matrix operator*(const Matrix &b) const {
    Matrix res;
    for (int i = 1; i <= 2; ++i)
      for (int j = 1; j <= 2; ++j)
        for (int k = 1; k <= 2; ++k)
          res.a[i][j] = (res.a[i][j] + a[i][k] * b.a[k][j]) % mod;
    return res;
  }
} ans, base;

void init() {
  base.a[1][1] = base.a[1][2] = base.a[2][1] = 1;
  ans.a[1][1] = ans.a[1][2] = 1;
}

void qpow(int b) {
  while (b) {
    if (b & 1) ans = ans * base;
    base = base * base;
    b >>= 1;
  }
}

int main() {
  int n = read();
  if (n <= 2) return puts("1"), 0;
  init();
  qpow(n - 2);
  println(ans.a[1][1] % mod);
}
```

这是一个稍微复杂一些的例子。

$$f_1 = f_2 = 0$$
$$f_n = 7f_{n-1} + 6f_{n-2} + 5n + 4 \times 3^n$$

我们发现，f_n 和 f_{n-1}, f_{n-2}, n 有关，于是考虑构造一个矩阵描述状态。

但是发现如果矩阵仅有这三个元素 $[f_n \quad f_{n-1} \quad n]$ 是难以构造出转移方程的，因为乘方运算和 $+1$ 无法用矩阵描述。

于是考虑构造一个更大的矩阵。

$$[f_n \quad f_{n-1} \quad n \quad 3^n \quad 1]$$

我们希望构造一个递推矩阵可以转移到

$$[f_{n+1} \quad f_n \quad n+1 \quad 3^{n+1} \quad 1]$$

转移矩阵即为

$$\begin{bmatrix} 7 & 1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 0 \\ 12 & 0 & 0 & 3 & 0 \\ 5 & 0 & 1 & 0 & 1 \end{bmatrix}$$

矩阵表达修改

「THUSCH 2017」大魔法师

大魔法师小 L 制作了 n 个魔力水晶球，每个水晶球有水、火、土三个属性的能量值。小 L 把这 n 个水晶球在地上从前向后排成一行，然后开始今天的魔法表演。

我们用 A_i, B_i, C_i 分别表示从前向后第 i 个水晶球（下标从 1 开始）的水、火、土的能量值。

小 L 计划施展 m 次魔法。每次，他会选择一个区间 $[l, r]$，然后施展以下 3 大类、7 种魔法之一：

1. 魔力激发：令区间里每个水晶球中 **特定属性** 的能量爆发，从而使另一个 **特定属性** 的能量增强。具体来说，有以下三种可能的表现形式：
   - 火元素激发水元素能量：令 $A_i = A_i + B_i$。
   - 土元素激发火元素能量：令 $B_i = B_i + C_i$。
   - 水元素激发土元素能量：令 $C_i = C_i + A_i$。

     需要注意的是，增强一种属性的能量并不会改变另一种属性的能量，例如 $A_i = A_i + B_i$ 并不会使 B_i 增加或减少。
2. 魔力增强：小 L 挥舞法杖，消耗自身 v 点法力值，来改变区间里每个水晶球的 **特定属性** 的能量。具体来说，有以下三种可能的表现形式：
   - 火元素能量定值增强：令 $A_i = A_i + v$。
   - 水元素能量翻倍增强：令 $B_i = B_i \cdot v$。
   - 土元素能量吸收融合：令 $C_i = v$。
3. 魔力释放：小 L 将区间里所有水晶球的能量聚集在一起，融合成一个新的水晶球，然后送给场外观众。生成的水晶球每种属性的能量值等于区间内所有水晶球对应能量值的代数和。**需要注意的是，魔力释放的过程不会真正改变区间内水晶球的能量**。

值得一提的是，小 L 制造和融合的水晶球的原材料都是定制版的 OI 工厂水晶，所以这些水晶球有一个能量阈值 998244353。当水晶球中某种属性的能量值大于等于这个阈值时，能量值会自动对阈值取模，从而避免水晶球爆炸。

小 W 为小 L（唯一的）观众，围观了整个表演，并且收到了小 L 在表演中融合的每个水晶球。小 W 想知道，这些水晶球蕴涵的三种属性的能量值分别是多少。

由于矩阵的结合律和分配律成立，单点修改可以自然地推广到区间，即推出矩阵后直接用线段树维护区间矩阵乘积即可。

下面将举几个例子。

$A_i = A_i + v$ 的转移

$$\begin{bmatrix} A & B & C & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ v & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} A+v & B & C & 1 \end{bmatrix}$$

$B_i = B_i \cdot v$ 的转移

$$\begin{bmatrix} A & B & C & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & v & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} A & B \cdot v & C & 1 \end{bmatrix}$$

「LibreOJ 6208」树上询问

有一棵 n 节点的树，根为 1 号节点。每个节点有两个权值 k_i, t_i，初始值均为 0。

给出三种操作：

1. $\mathrm{Add}(x, d)$ 操作：将 x 到根的路径上所有点的 $k_i \leftarrow k_i + d$
2. $\mathrm{Mul}(x, d)$ 操作：将 x 到根的路径上所有点的 $t_i \leftarrow t_i + d \times k_i$
3. $\mathrm{Query}(x)$ 操作：询问点 x 的权值 t_x

$n,\ m \leq 100000,\ -10 \leq d \leq 10$

若直接思考，下放操作和维护信息并不是很好想。但是矩阵可以轻松地表达。

$$\begin{bmatrix} k & t & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d & 0 & 1 \end{bmatrix} = \begin{bmatrix} k + d & t & 1 \end{bmatrix}$$

$$\begin{bmatrix} k & t & 1 \end{bmatrix} \begin{bmatrix} 1 & d & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} k & t + d \times k & 1 \end{bmatrix}$$

定长路径统计

问题描述

给一个 n 阶有向图，每条边的边权均为 1，然后给一个整数 k，你的任务是对于所有点对 (u, v) 求出从 u 到 v 长度为 k 的路径的数量（不一定是简单路径，即路径上的点或者边可能走多次）。

我们将这个图用邻接矩阵 G（对于图中的边 $(u \to v)$，令 $G[u, v] = 1$，其余为 0 的矩阵；如果有重边，则设 $G[u, v]$ 为重边的数量）表示这个有向图。下述算法同样适用于图有自环的情况。

显然，该邻接矩阵对应 $k = 1$ 时的答案。

假设我们知道长度为 k 的路径条数构成的矩阵，记为矩阵 C_k，我们想求 C_{k+1}。显然有 DP 转移方程

$$C_{k+1}[i, j] = \sum_{p=1}^{n} C_k[i, p] \cdot G[p, j]$$

我们可以把它看作矩阵乘法的运算，于是上述转移可以描述为

$$C_{k+1} = C_k \cdot G$$

那么把这个递推式展开可以得到

$$C_k = \underbrace{G \cdot G \cdots G}_{k\text{次}} = G^k$$

要计算这个矩阵幂，我们可以使用快速幂（二进制取幂）的思想，在 $O(n^3 \log k)$ 的复杂度内计算结果。

定长最短路

问题描述

给你一个 n 阶加权有向图和一个整数 k。对于每个点对 (u, v) 找到从 u 到 v 的恰好包含 k 条边的最短路的长度。（不一定是简单路径，即路径上的点或者边可能走多次）

我们仍构造这个图的邻接矩阵 G，$G[i, j]$ 表示从 i 到 j 的边权。如果 i, j 两点之间没有边，那么 $G[i, j] = \infty$。（有重边的情况取边权的最小值）

显然上述矩阵对应 $k = 1$ 时问题的答案。我们仍假设我们知道 k 的答案，记为矩阵 L_k。现在我们想求 $k + 1$ 的答案。显然有转移方程

$$L_{k+1}[i, j] = \min_{1 \le p \le n} \{L_k[i, p] + G[p, j]\}$$

事实上我们可以类比矩阵乘法，你发现上述转移只是把矩阵乘法的乘积求和变成相加取最小值，于是我们定义这个运算为 $\odot$，即

$$A \odot B = C \iff C[i, j] = \min_{1 \le p \le n} \{A[i, p] + B[p, j]\}$$

于是得到

$$L_{k+1} = L_k \odot G$$

展开递推式得到

$$L_k = \underbrace{G \odot \ldots \odot G}_{k\text{次}} = G^{\odot k}$$

我们仍然可以用矩阵快速幂的方法计算上式，因为它显然是具有结合律的。时间复杂度 $O(n^3 \log k)$。

限长路径计数/最短路

上述算法只适用于边数固定的情况。然而我们可以改进算法以解决边数小于等于 k 的情况。具体地，考虑以下问题：

> **问题描述**
>
> 给一个 n 阶有向图，边权为 1，然后给一个整数 k，你的任务是对于每个点对 (u,v) 找到从 u 到 v 长度小于等于 k 的路径的数量（不一定是简单路径，即路径上的点或者边可能走多次）。

我们对于每个点 v，建立一个虚点 v' 用于记录答案，并在图中加入 (v,v') 和 (v',v') 这两条边。那么对于点对 (u,v)，从 u 到 v 边数小于等于 k 的路径的数量，就和从 u 到 v' 边数恰好等于 $k+1$ 的路径的数量相等，这是因为对于任意一条边数为 $m(m \le k)$ 的路径 $(p_0=u) \to p_1 \to p_2 \to \cdots \to p_{m-1} \to (p_m=v)$，都存在一条边数为 $k+1$ 的路径 $(p_0=u) \to p_1 \to p_2 \to \cdots \to p_{m-1} \to (p_m=v) \to v' \to \cdots \to v'$ 与之一一对应。

对于求边数小于等于 k 的最短路，只需对每个点加一个边权为 0 的自环即可。

习题

- 洛谷 P1962 斐波那契数列，即上面的例题，同题 POJ3070
- 洛谷 P1349 广义斐波那契数列，base 矩阵需要变化一下
- 洛谷 P1939【模板】矩阵加速（数列），base 矩阵变成了 3×3 的矩阵，推导过程与上面差不多。