

ACM/ICPC at Wuhan University

Xioumu STL(文字)

Created by xioumu v1.00

目录

Graph	2
混合图欧拉路径	2
生成树的计数 (matrix-Tree)	2
关于中国邮递员问题和欧拉图应用	3
Prüfer 编码与 Cayley 公式 (完全图 K_n 有 $n^{(n-2)}$ 棵生成树)	6
DataStructure	7
线段树	7
Computational Geometry	7
数论	7
欧拉公式	7
欧拉函数	7
Pick 公式	8
卡特兰数	8
二维旋转	8
逆元	9
求和公式	10
DP	10
概率 DP	10
数位 DP	10
Java	10
JAVA 之 BigInteger	10

Graph

混合图欧拉路径

当一个有向图所有的度为 0 时, 这个有向图为欧拉图。

所以混合图的判断欧拉图只要判断原图的无向边往哪个方向即可, 所以可以先以任意形态连边, 然后有流量的为正向边, 无流量的为反向边。
入度 - 出度 > 0 的, 和源点连一条(入度 - 出度) / 2 的边, < 0 的, 和汇点连边。

最后如果最大流等于所有与源点连的流量, 即存在一种方法使原图变为欧拉图。

生成树的计数 (matrix-Tree)

Matrix-Tree 定理是解决生成树计数问题最有力的武器之一。它首先于 1847 年被 Kirchhoff 证明。在介绍定理之前, 我们首先明确几个概念:

1. G 的度数矩阵 $D[G]$ 是一个 $n \times n$ 的矩阵, 并且满足: 当 $i \neq j$ 时, $d_{ij} = 0$; 当 $i = j$ 时, d_{ij} 等于 v_i 的度数。
2. G 的邻接矩阵 $A[G]$ 也是一个 $n \times n$ 的矩阵, 并且满足: 如果 v_i, v_j 之间有边直接相连, 则 $a_{ij} = 1$, 否则为 0。

我们定义 G 的 Kirchhoff 矩阵(也称为拉普拉斯算子) $C[G]$ 为 $C[G] = D[G] - A[G]$, 则 *Matrix-Tree* 定理可以描述为: G 的所有不同的生成树的个数等于其 Kirchhoff 矩阵 $C[G]$ 任何一个 $n-1$ 阶主子式的行列式的绝对值。所谓 $n-1$ 阶主子式, 就是对于 $r (1 \leq r \leq n)$, 将 $C[G]$ 的第 r 行、第 r 列同时去掉后得到的新矩阵, 用 $C_r[G]$ 表示。

下面我们举一个例子来解释 *Matrix-Tree* 定理。如图 *a* 所示, G 是一个由 5 个点组成的无向图。

图 *a*

根据定义, 它的 Kirchhoff 矩阵 $C[G]$ 为

我们取 $r=2$, 则有:

$$|C_2[G]|$$

$$= \begin{vmatrix} 2 & -1 & 0 & 0 \\ -1 & 3 & 0 & -1 \\ 0 & 0 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{vmatrix}$$

$$= 2 \cdot \begin{vmatrix} 3 & 0 & -1 \\ 0 & 2 & -1 \\ -1 & -1 & 2 \end{vmatrix} - (-1) \cdot \begin{vmatrix} -1 & 0 & -1 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{vmatrix} + 0 \cdot \begin{vmatrix} -1 & 3 & -1 \\ 0 & 0 & -1 \\ 0 & -1 & 2 \end{vmatrix} - 0 \cdot \begin{vmatrix} -1 & 3 & 0 \\ 0 & 0 & 2 \\ 0 & -1 & -1 \end{vmatrix}$$

$$= 2 \left(3 \cdot \begin{vmatrix} 2 & -1 \\ -1 & 2 \end{vmatrix} - 0 \cdot \begin{vmatrix} 3 & -1 \\ -1 & 2 \end{vmatrix} + (-1) \cdot \begin{vmatrix} 0 & 2 \\ -1 & -1 \end{vmatrix} \right) - (-1) \left((-1) \cdot \begin{vmatrix} 2 & -1 \\ -1 & 2 \end{vmatrix} - 0 \cdot \begin{vmatrix} 3 & -1 \\ 0 & -1 \end{vmatrix} \right) + 0 - 0$$

$$= 2(3(4-1) - 0 + (-1)(0 - (-2))) - (-1)((-1)(4-1) - 0 + (-1)(0-0)) + 0 - 0$$

$$= 2(9-2) - (-1)(-1)(3) = 11$$

这 11 棵生成树如图 *b* 所示。

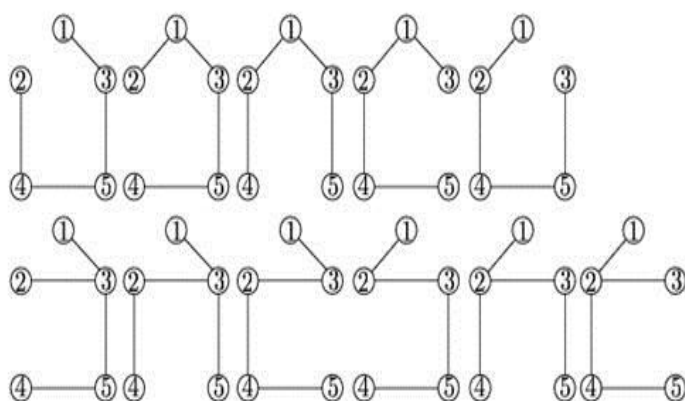


图 *b*

关于中国邮递员问题和欧拉图应用

中国邮递员问题:

1962 年有管梅谷先生提出中国邮递员问题（简称 CPP）。一个邮递员从邮局出发，要走完他所管辖的每一条街道，可重复走一条街道，然后返回邮局。任何选择一条尽可能短的路线。

这个问题可以转化为：给定一个具有非负权的赋权图 G ，

- (1) 用添加重复边的方法求 G 的一个 Euler 赋权母图 G^* ，使得尽可能小。
- (2) 求 G^* 的 Euler 环游。

人们也开始关注另一类问题，旅行商问题（简称 TSP）。TSP 是点路优化问题，它是 NPC 的。而 CPP 是弧路优化问题，该问题有几种变形，与加权图奇点的最小完全匹配或网络流等价，有多项式算法。[\[1\]](#)

欧拉图：

图 G 中经过每条边一次并且仅一次的回路称作欧拉回路。存在欧拉回路的图称为欧拉图。

无向图欧拉图判定：

无向图 G 为欧拉图，当且仅当 G 为连通图且所有顶点的度为偶数。

有向图欧拉图判定：

有向图 G 为欧拉图，当且仅当 G 的基图[\[2\]](#)连通，且所有顶点的入度等于出度。

欧拉回路性质：

性质 1 设 C 是欧拉图 G 中的一个简单回路，将 C 中的边从图 G 中删去得到一个新的图 G' ，则 G' 的每一个极大连通子图都有一条欧拉回路。

性质 2 设 C_1 、 C_2 是图 G 的两个没有公共边，但有至少一个公共顶点的简单回路，我们可以将它们合并成一个新的简单回路 C' 。

欧拉回路算法：

- 1 在图 G 中任意找一个回路 C ；
- 2 将图 G 中属于回路 C 的边删除；
- 3 在残留图的各极大连通子图中分别寻找欧拉回路；
- 4 将各极大连通子图的欧拉回路合并到 C 中得到图 G 的欧拉回路。

由于该算法执行过程中每条边最多访问两次，因此该算法的时间复杂度为 $O(|E|)$ 。

如果使用递归形式，得注意 $|E|$ 的问题。使用非递归形式防止栈溢出。

如果图 是有向图，我们仍然可以使用以上算法。

<http://acm.hdu.edu.cn/showproblem.php?pid=1116> 有向图欧拉图和半欧拉图判定

<http://acm.pku.edu.cn/JudgeOnline/problem?id=2337> 输出路径

中国邮递员问题①：

一个邮递员从邮局出发，要走完他所管辖的每一条街道，可重复走一条街道，然后返回邮局。所有街道都是双向通行的，且每条街道都有一个长度值。任何选择一条尽可能短的路线。

分析：

双向连通，即给定无向图 G 。

如果 G 不连通，则无解。

如果 G 是欧拉图，则显然欧拉回路就是最优路线。

如果 G 连通，但不是欧拉图，说明图中有奇点[3]。奇点都是成对出现的，证明从略。

对于最简单情况，即 2 个奇点，设 (u, v) 。我们可以在 G 中对 (u, v) 求最短路径 R ，构造出新图 $G' = G \cup R$ 。此时 G' 就是欧拉图。

证明： u 和 v 加上了一条边，度加一，改变了奇偶性。而 R 中其他点度加二，奇偶性不变。

由此可知，加一次 R ，能够减少两个奇点。推广到 k 个奇点的情况，加 $k/2$ 个 R 就能使度全为偶数。

接下的问题是求一个 k 个奇点的配对方案，使得 $k/2$ 个路径总长度最小。

这个就是无向完全图最小权匹配问题。有一种 Edmonds 算法，时间复杂度 $O(N^3)$ 。[4]

也可转换为二分图，用松弛优化的 KM 算法，时间复杂度也是 $O(N^3)$ 。

完整的算法流程如下：

- 1 如果 G 是连通图，转 2，否则返回无解并结束；
- 2 检查 G 中的奇点，构成图 H 的顶点集；
- 3 求出 G 中每对奇点之间的最短路径长度，作为图 H 对应顶点间的边权；
- 4 对 H 进行最小权匹配；
- 5 把最小权匹配里的每一条匹配边代表的路径，加入到图 G 中得到图 G' ；
- 6 在 G' 中求欧拉回路，即所求的最优路线。

中国邮递员问题②：

和①相似，只是所有街道都是单向通行的。

分析：

单向连通，即给定有向图 G 。

和①的分析一样，我们来讨论如何从 G 转换为欧拉图 G' 。

首先计算每个顶点 v 的入度与出度之差 $d'(v)$ 。如果 G 中所有的 v 都有 $d'(v) = 0$ ，那么 G 中已经存在欧拉回路。

$d'(v) > 0$ 说明得加上出度。 $d'(v) < 0$ 说明得加上入度。

而当 $d'(v) = 0$ ，则不能做任何新增路径的端点。

可以看出这个模型很像网络流模型。

顶点 $d'(v) > 0$ 对应于网络流模型中的源点，它发出 $d'(v)$ 个单位的流；顶点 $d'(v) < 0$ 对应于网络流模型中的汇点，它接收 $-d'(v)$ 个单位的流；而 $d'(v) = 0$ 的顶点，则对应于网络流模型中的中间结点，它接收的流量等于发出的流量。在原问题中还要求增加的路径总长度最小，我们可以给网络中每条边的费用值 设为图 中对应边的长度。这样，在网络中求最小费用最大流，即可使总费用最小。

这样构造网络 N :

- 1 其顶点集为图 G 的所有顶点, 以及附加的超级源 s 和超级汇 t ;
- 2 对于图 G 中每一条边 (u,v) , 在 N 中连边 (u,v) , 容量为 ∞ , 费用为该边的长度;
- 3 从源点 s 向所有 $d'(v) > 0$ 的顶点 v 连边 (s,v) , 容量为 $d'(v)$, 费用为 0;
- 4 从所有 $d'(v) < 0$ 的顶点 v 向汇点 t 连边 (v,t) , 容量为 $-d'(v)$, 费用为 0。

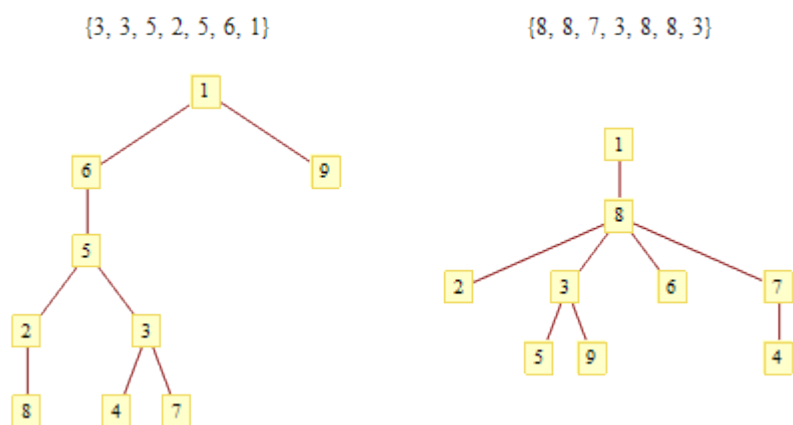
完整的算法流程如下:

- 1 如果 G 的基图连通且所有顶点的入、出度均不为 0, 转 2, 否则返回无解并结束;
- 2 计算所有顶点 v 的 $d'(v)$ 值;
- 3 构造网络 N ;
- 4 在网络 N 中求最小费用最大流;
- 5 对 N 中每一条流量 $f(u,v)$ 的边 (u,v) , 在图 G 中增加 $f(u,v)$ 次得到 G' ;
- 6 在 G' 中求欧拉回路, 即为所求的最优路线。

NPC 问题:

如果部分街道能够双向通行, 部分街道只能单向通行。这个问题已被证明是 NPC 的。[\[5\]](#)

Prüfer 编码与 Cayley 公式 (完全图 K_n 有 n^{n-2} 棵生成树)



Cayley 公式是说, 一个完全图 K_n 有 n^{n-2} 棵生成树, 换句话说 n 个节点的带标号的无根树有 n^{n-2} 个。今天我学到了 Cayley 公式的一个非常简单的证明, 证明依赖于 Prüfer 编码, 它是对带标号无根树的一种编码方式。

给定一棵带标号的无根树, 找出编号最小的叶子节点, 写下与它相邻的节点的编号, 然后删掉这个叶子节点。反复执行这个操作直到只剩两个节点为止。由于节点数 $n > 2$ 的树总存在叶子节点, 因此一棵 n 个节点的无根树唯一地对应了一个长度为 $n-2$ 的数列, 数列中的每个数都在 1 到 n 的范围内。下面我们只需要说明, 任何一个长为 $n-2$ 、取值范围在 1 到 n 之间的数列都唯一地对应了一棵 n 个节点的无根树, 这样我们的带标号无根树就和 Prüfer 编码之间形成一一对应的关系, Cayley 公式便不证自明了。

Wuhan University

注意到，如果一个节点 **A** 不是叶子节点，那么它至少有两条边；但在上述过程结束后，整个图只剩下一条边，因此节点 **A** 的至少一个相邻节点被去掉过，节点 **A** 的编号将会在这棵树对应的 Prüfer 编码中出现。反过来，在 Prüfer 编码中出现过的数字显然不可能是这棵树（初始时）的叶子。于是我们看到，没有在 Prüfer 编码中出现过的数字恰好就是这棵树（初始时）的叶子节点。找出没有出现过的数字中最小的那一个（比如④），它就是与 Prüfer 编码中第一个数所标识的节点（比如③）相邻的叶子。接下来，我们递归地考虑后面 $n-3$ 位编码（别忘了编码总长是 $n-2$ ）：找出除④以外不在后 $n-3$ 位编码中的最小的数（左图的例子中是⑦），将它连接到整个编码的第 2 个数所对应的节点上（例子中还是③）。再接下来，找出除④和⑦以外后 $n-4$ 位编码中最小的不被包含的数，做同样的处理……依次把③⑧②⑤⑥与编码中第 3、4、5、6、7 位所表示的节点相连。最后，我们还有①和⑨没处理过，直接把它们俩连接起来就行了。由于没处理过的节点数总比剩下的编码长度大 2，因此我们总能找到一个最小的没在剩余编码中出现的数，算法总能进行下去。这样，任何一个 Prüfer 编码都唯一地对应了一棵无根树，有多少个 $n-2$ 位的 Prüfer 编码就有多少个带标号的无根树。

一个有趣的推广是， n 个节点的度依次为 D_1, D_2, \dots, D_n 的无根树共有 $(n-2)! / [(D_1-1)!(D_2-1)!\dots(D_n-1)!]$ 个，因为此时 Prüfer 编码中的数字 i 恰好出现 D_i-1 次。

DataStructure

线段树

离散

假如离散线段的成一个点的话，记得两边端点和左边端点左边的点和右边端点右边的点各离散一个点，一条线段总共离散成 4 个点。否则会出问题。

lazy 函数

$\text{Lazy}[t] = \text{true}$ 表示需要向下传值，但节点 t 的值已经更新完。
也就是说 t 节点的值与 $\text{lazy}[t]$ 无关。

Computational Geometry

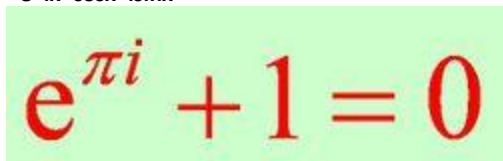
数论

欧拉公式

（欧拉公式）如果一个连通的平面图有 n 个点， m 条边和 f 个面，那么 $f=m-n+2$

欧拉函数

$$e^{ix} = \cos x + i \sin x$$

A green rectangular box containing the mathematical equation $e^{\pi i} + 1 = 0$ in a large, red, serif font.

自然对数的底 e ，圆周率 π ，两个单位：虚数单位 i 和自然数的单位 1，以及被称为人类伟大发现之一的 0。[数学家](#)们评价它是“上帝创造的公式”

设 $p_1^{e_1}, p_2^{e_2}, p_3^{e_3}, p_4^{e_4}, \dots, p_k^{e_k}$ 是 m 的所有互异素因数，那么

$$\phi(m) = m \prod_{i=1}^k \left(1 - \frac{1}{p_i}\right)$$

欧拉 ϕ 函数： $\phi(n)$ 是所有小于 n 的正整数里，和 n 互素的整数的个数。 n 是一个正整数。

欧拉证明了下面这个式子：

如果 n 的标准素因子分解式是 $p_1^{a_1} p_2^{a_2} \dots p_m^{a_m}$ ，其中众 $p_j (j=1, 2, \dots, m)$ 都是素数，而且两两不等。则有

$$\phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \dots (1 - \frac{1}{p_m})$$

利用容斥原理可以证明它。

Pick 公式

定理一：Pick 定理，1899 年

设 Γ 为平面上以格子点为顶点之单纯多边形，则其面积为

$$A = \frac{b}{2} + i - 1 \quad (8)$$

其中 b 为边界上的格子点数， i 为内部的格子点数。(8) 式叫做 Pick 公式。

定理二：(棱线定理, Edge Theorem)

对于任意连通的三角形化的平面图枝，其棱线的个数恒为

$$E = 2b + 3i - 3 \quad (12)$$

其中 b 与 i 分别表示图枝边界上的顶点数与内部的顶点数。

卡特兰数

卡特兰数：1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020, 91482563640, 343059613650, 1289904147324, 4861946401452,

令 $h(1)=1, h(0)=1$, catalan 数满足递归式：

$$h(n) = h(0) * h(n-1) + h(1) * h(n-2) + \dots + h(n-1) h(0) \quad (\text{其中 } n \geq 2)$$

另类递归式：

$$h(n) = h(n-1) * (4 * n - 2) / (n + 1);$$

该递推关系的解为：

$$h(n) = C(2n, n) / (n + 1) \quad (n = 1, 2, 3, \dots)$$

二维旋转

看下面的矩阵

$$\begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

这个是二维向量的旋转矩阵，它可以将一个向量逆时针旋转一个角度。

将其变形，变会得到二维向量的顺时针旋转的形式：

$$\begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix}$$

这里要注意一种特殊的情况，就是当角度为 90 度的时候，sin 和 cos 的结果只有 1, 0, -1 三种可能性。所以这个矩阵可以改写成特殊的形式，其意义在于用这样的旋转操作，不会产生精度问题。sin 和 cos 的运算的精度是比较低的，能少用则尽量少用。在计算几何中的向量旋转操作，大部分都可以通过变形，只用到旋转 90 度，从而避免精度问题。

===3 维===

围绕轴 $u = (ux, uy, uz)$ 来旋转的矩阵代码如下：

```
1. double a[3][3] = {
2.     {SQR(ux) + (1 - SQR(ux)) * c, ux * uy * (1 - c) - uz * s, ux * uz * (1 - c) + uy * s},
3.     {ux * uy * (1 - c) + uz * s, SQR(uy) + (1 - SQR(uy)) * c, uy * uz * (1 - c) - ux * s},
4.     {ux * uz * (1 - c) - uy * s, uy * uz * (1 - c) + ux * s, SQR(uz) + (1 - SQR(uz)) * c}
5. };
```

要注意这个 $ux * ux + uy * uy + uz * uz = 1$

有一个比较有意思的问题就是，知道旋转矩阵之后，怎么来确定向量 u 呢？

我们做如下变形之后，可以得到：

$$Ru = Iu \rightarrow (R - I)u = 0$$

也就是说我们要找到一个非 0 的向量，使得他和一个矩阵乘起来得到一个空矩阵。这个问题可以用高斯消元来解决。实际上我们就是要解一个线性方程组。

逆元

定义 如果 $ab \equiv 1 \pmod{m}$ ，则称 b 是 a 的模 m 逆，记作 a 的模 m 逆是方程 $ax \equiv 1 \pmod{m}$ 的解。

即 $\gcd(a, m) = 1$ ，即 (a, m) 互质

例：求 5 的模 7 逆

做辗转相除法，求得整数 b, k 使得 $5b + 7k = 1$ ，则 b 是 5 的模 7 逆。

计算如下：

$$7 = 5 + 2, \quad 5 = 2 \times 2 + 1.$$

$$\text{回代} \quad 1 = 5 - 2 \times 2 = 5 - 2 \times (7 - 5) = 3 \times 5 - 2 \times 7,$$

$$\text{得 } 5^{-1} \equiv 3 \pmod{7}.$$

例：求 21 的模 73 逆

做辗转相除法，求得整数 b, k 使得 $21b + 73k = 1$ ，则 b 是 21 的模 73 逆。

计算如下：

$$73 = 21 \times 3 + 10$$

$$21 = 10 \times 2 + 1$$

$$\text{回代} \quad 1 = 21 - 10 \times 2$$

$$1 = 21 - (73 - 21 \times 3) \times 2$$

$$= 21 - 73 \times 2 + 6 \times 21$$

$$= 7 \times 21 - 73 \times 2$$

Wuhan University

得 $21^{-1} \equiv 7 \pmod{73}$.

求和公式

$$1^4 + 2^4 + 3^4 + 4^4 + \dots + n^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$$

$$1^3 + 2^3 + 3^3 + 4^3 + \dots + n^3 = \left[\frac{n(n+1)}{2}\right]^2$$

$$1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$1 + 2 + 3 + 4 + \dots + n = \frac{n(n+1)}{2}$$

DP

概率 DP

要倒着 DP，有环可用高斯消元，或者设 x 直接把 x 给化简到一边。

数位 DP

1. 先预处理出来一些信息（如： $f(10), f(100), f(1000)$ 的值，一般用递推预处理），然后直接递归算。
2. 只需预付处理少量信息，然后直接记忆化搜索。

Java

JAVA 之 BigInteger

用 [Java](#) 来处理高精度问题，相信对很多 ACMer 来说都是一件很 happy 的事，简单易懂。用 Java 刷了一些题，感觉 Java 还不错，在处理高精度和进制转换中，调用库函数的来处理。下面是写的一些 Java 中一些基本的函数的及其.....

头文件：import java.io.*;

import java.util.*;

import java.math.*;

读入：Scanner cin = Scanner (System.in);

while(cin.hasNext())//等价于!=EOF

n=cin.nextInt();//读入一个 int 型的数

n=cin.nextBigInteger();//读入一个大整数

输出：System.out.print(n);//打印 n

System.out.println();//换行

System.out.printf("%d\n",n);//也可以类似 c++里的输出方式

定义：int i,j,k,a[];

a = new int[100];

Wuhan University

BigInteger n,m;

BigDecimal n;

String s;

数据类型:

数据类型 类型名 位长 取值范围 默认值

布尔型 boolean 1 true,false false

字节型 byte 8 -128-127 0

字符型 char 16 '\u000'-'uffff' '\u0000'

短整型 short 16 -32768-32767 0

整型 int 32 -2147483648,2147483647 0

长整型 long 64 -9.22E18,9.22E18 0

浮点型 float 32 1.4E-45-3.4028E+38 0.0

双精度型 double 64 4.9E-324,1.7977E+308 0.0

这里特别要提出出的两种类型:

BigInteger 任意大的整数, 原则上是, 只要你的计算机的内存足够大, 可以有无限位的

BigDecimal 任意大的实数, 可以处理小数精度问题。

BigInteger 中一些常见的函数:

A=BigInteger.ONE

B=BigInteger.TEN

C=BigInteger.ZERO

一些常见的数的赋初值。将 int 型的数赋值给 **BigInteger**, **BigInteger.valueOf(k)**;

基本的函数:

valueOf:赋初值

add:+ a.add(b);

subtract:-

multiply:*

divide:/

remainder: this % val

divideAndRemainder: a[0]=this / val; a[1]=this % val

pow: a.pow(b)=a^b

gcd,abs:公约数, 绝对值

negate: 取负数

signum: 符号函数

mod: a.mod(b)=a%b;

shiftLeft:左移, this << n , this*2^n;

shiftRight:右移, this >> n, this/2^n;

and:等同于 c++的&&,且;

or: ||, 或;

xor:异或, `BigInteger xor(BigInteger val),this^val`

not!:非;

bitLength: 返回该数的最小二进制补码表示的位的个数, 即 *不包括* 符号位 (`ceil(log2(this <0 ? -this : this + 1))`)。对正数来说, 这等价于普通二进制表示的位的个数。

bitCount: 返回该数的二进制补码表示中不包扩符号位在位的个数。该方法在 `BigIntegers` 之上实现位向量风格的集合时很有用。

isProbablePrime: 如果该 `BigInteger` 可能是素数, 则返回 `true`; 如果它很明确是一个合数, 则返回 `false`。参数 `certainty` 是对调用者愿意忍受的不确定性的度量: 如果该数是素数的概率超过了 `1 - 1/2**certainty` 方法, 则该方法返回 `true`。执行时间正比于参数确定性的值。

compareTo: 根据该数值是小于、等于、或大于 `val` 返回 `-1`、`0` 或 `1`;

equals: 判断两数是否相等, 也可以用 `compareTo` 来代替;

min, max: 取两个数的较小、大者;

intValue, longValue, floatValue, doubleValue: 把该数转换为该类型的数的值。

今天参考课本写了一个关于二进制与十进制转换的程序, 程序算法不难, 但写完后测试发现不论是二转十还是十转二, 对于大于 21 亿即超过整数范围的数不能很好的转换。都会变成 0。

参考书籍发现使用使用 `BigInteger` 可以解决这个问题。

于是查找了下 JDK,然后测试几次终于写成功了!

使用心得如下:

1, `BigInteger` 属于 `java.math.BigInteger`,因此在每次使用前都要 `import` 这个类。偶开始就忘记 `import` 了, 于是总提示找不到提示符。

2, 其构造方法有很多, 但现在偶用到的有: `BigInteger(String val)`

将 `BigInteger` 的十进制字符串表示形式转换为 `BigInteger`。

`BigInteger(String val, int radix)`

将指定基数的 `BigInteger` 的字符串表示形式转换为 `BigInteger`。

如要将 `int` 型的 2 转换为 `BigInteger` 型, 要写为 `BigInteger two=new BigInteger("2");` //注意 2 双引号不能省略

3, `BigInteger` 类模拟了所有的 `int` 型数学操作, 如 `add()`=="+",`divide()`=="-"等, 但注意其内容进行数学运算时不能直接使用数学运算符进行运算, 必须使用其内部方法。而且其操作数也必须为 `BigInteger` 型。

如: `two.add(2)`就是一种错误的操作, 因为 2 没有变为 `BigInteger` 型。

4, 当要把计算结果输出时应该使用 `toString` 方法将其转换为 10 进制的字符串, 详细说明如下:

`String toString()`

返回此 `BigInteger` 的十进制字符串表示形式。

输出方法: `System.out.print(two.toString());`

5,另外说明三个个用到的函数。 `BigInteger remainder(BigInteger val)`

返回其值为 `(this % val)` 的 `BigInteger`。

`BigInteger negate()`

返回其值是 `(-this)` 的 `BigInteger`。

`int compareTo(BigInteger val)`

将此 `BigInteger` 与指定的 `BigInteger` 进行比较。

`remainder` 用来求余数。

`negate` 将操作数变为相反数。

compare 的详解如下:

compareTo

`public int compareTo(BigInteger val)` 将此 `BigInteger` 与指定的 `BigInteger` 进行比较。对于针对六个布尔比较运算符 (`<`, `==`, `>`, `>=`, `!=`, `<=`) 中的每一个运算符的各个方法, 优先提供此方法。执行这些比较的建议语句是: `(x.compareTo(y) <op> 0)`, 其中 `<op>` 是六个比较运算符之一。

指定者:

接口 `Comparable<BigInteger>` 中的 `compareTo`

参数:

`val` - 将此 `BigInteger` 与之比较的 `BigInteger`。

返回:

将 `BigInteger` 的数转为 2 进制:

```
public class TestChange {  
    public static void main(String[] args) {  
        System.out.println(change("3",10,2));  
    }  
    //num 要转换的数 from 源数的进制 to 要转换成的进制  
    private static String change(String num,int from, int to){  
        return new java.math.BigInteger(num, from).toString(to);  
    }  
}
```