



ACM-ICPC Template Book

ACM-ICPC Template Book

ramay7

Soochow University
Mechanical and Electrical Engineering College

Oct. 2016

Contents

1 基础	1
1.1 常见错误	1
1.2 思考角度	1
1.3 奇巧淫技	2
1.3.1 memset 初始化极大极小值和较大较小值	2
1.3.2 二进制中 1 的个数	2
1.3.3 vector 去重	2
1.3.4 快速乘法	2
1.3.5 快速开方	3
1.3.6 高精度开根 $a^{\frac{1}{b}}$	3
1.3.7 把所有点(向量)从第三象限开始逆时针排序	3
1.3.8 求长度为 n 的有序数组 a 中 k 的个数	4
1.3.9 C++ 关闭同步	4
1.3.10 cout 浮点数控制精度输出	4
1.3.11 快读	4
1.4 结论	5
1.4.1 判断 n 个整数坐标能否构成正 n 边形	5
1.4.2 已知年月日计算星期几	5
1.4.3 方格染色	5
1.4.4 包含特定方格所有子矩阵面积和	6
1.4.5 n 以内 $gcd(a, b) = d$ 的无序对对数	6
1.4.6 $gcd(x^a - 1, x^b - 1) = x^{gcd(a,b)} - 1$	6
1.5 归并排序及利用归并排序求逆序对数	7
1.6 快排、利用快排查找第 k 大元素	8
1.7 矩阵快速幂	9
1.7.1 循环矩阵	10
1.8 高精度	11
1.8.1 利用 FFT 实现大数乘法	18
2 搜索	21
2.1 折半搜索	21
3 STL	23
3.1 bitset	23
3.1.1 用 <code>unsigned long</code> 值初始化 <code>bitset</code> 对象	23
3.1.2 用 <code>string</code> 对象初始化 <code>bitset</code> 对象	23
3.1.3 操作	23
3.1.4 测试	24
3.2 单调栈、单调队列	25
3.2.1 单调栈	25
3.2.2 单调队列	26
3.3 list	27
3.3.1 CF 350 E	27

4 计算几何	29
4.1 简单	29
4.1.1 定义	29
4.1.2 顺时针输出所有顶点	31
4.1.3 求半径 R 圆覆盖最多点数及由圆上两点和半径求圆心	32
4.1.4 计算两圆公共部分面积	34
4.1.5 判断两线段是否相交	35
4.1.6 判断点和多边形的关系	37
4.1.7 计算多边形的最小宽度	37
4.1.8 计算多边形的最长内直径	38
4.2 凸包	42
4.2.1 求凸包顶点:Graham 扫描法	42
4.2.2 判断稳定凸包	42
4.2.3 凸包直径	43
4.2.4 凸包周长及面积	44
4.2.5 凸包最大三角形面积	44
4.2.6 凸包最大四边形面积	45
4.2.7 最小覆盖矩形面积	45
4.2.8 凸包最小外接平行四边形面积	46
4.2.9 两相离凸包最短距离	47
4.3 三维几何	50
4.3.1 任意四面体的外接球半径公式	50
4.3.2 定义	50
4.3.3 给定四个点, 求四面体内切球球心和半径	52
4.3.4 给定四面体的六条边长求四面体体积	52
5 数论	55
5.1 一些理论	55
5.1.1 大整数取模	55
5.1.2 哥德巴赫猜想 (1 + 1 问题)	55
5.1.3 费马小定理	55
5.1.4 素数定理	55
5.1.5 算数基本定理	55
5.1.6 $ax + by = c$ 的任意整数解	56
5.1.7 n 是 m 的倍数, $[1, n]$ 中和 m 互素数个数	56
5.1.8 p 为奇素数, $1 \sim (p - 1)$ 模 p 的逆元对应全部 $1 \sim (p - 1)$ 中的所有数, 既是单射也是满射	56
5.1.9 调和级数求和	56
5.1.10 如果 $\gcd(a, m) = 1$, 那么 $\gcd(a + k * m, m) = 1, k \in \mathbb{Z}$	56
5.1.11 指数降幂公式	56
5.2 素数筛	57
5.2.1 区间素数筛	57
5.3 约数筛	58
5.3.1 51nod 1586	58
5.4 分解质因数	59
5.4.1 给定 n 求满足 $a^p = n$ 的最大 p	59
5.4.2 求满足 $\text{lcm}(i, j) = n (1 \leq i \leq j \leq n \leq 10^{14})$ 的 (i, j) 对数	60
5.5 逆元	60
5.5.1 介绍	60
5.5.2 模素数逆元连乘	60
5.6 欧拉函数	60
5.6.1 介绍	60
5.6.2 求单个数的欧拉函数	61
5.6.3 欧拉筛	62
5.7 扩展欧几里得	63
5.7.1 裴蜀定理	63
5.7.2 求解逆元 $ax \equiv 1 (\text{mod } n)$ 的最小非负数解	63
5.7.3 求所有解中 $C = x + y $ 最小值	64

5.7.4 求最小非负整数解 x 和此时的 y	64
5.8 模线性方程组	64
5.8.1 利用扩展欧几里德求解模线性同余方程 $ax \equiv b \pmod{n}$	64
5.9 中国剩余定理	66
5.9.1 模数互素	66
5.9.2 模数不互素	66
5.10 法雷级数	68
5.10.1 介绍	68
5.10.2 n 级法雷数列	68
5.10.3 法雷数列的构造	68
5.10.4 求 n 级法雷级数个数	68
5.10.5 性质	68
5.11 原根	69
5.11.1 介绍	69
5.11.2 求模素数 p 的所有原根	70
5.12 BSGS 算法	71
5.12.1 扩展 BSGS($\gcd(a, p) \neq 1$)	72
5.13 莫比乌斯反演	73
5.13.1 积性函数	73
5.13.2 狄利克雷卷积	73
5.13.3 莫比乌斯反演公式	73
5.13.4 莫比乌斯函数 μ	73
5.13.5 线性筛求解积性函数	74
5.14 反素数	77
5.14.1 介绍	77
5.14.2 求最小的 n 使得其约数个数为 x	78
5.15 卢卡斯定理	79
5.15.1 给定 n 求 C_n^m ($0 \leq m \leq n \leq 10^8$) 为奇数的 m 个数	79
5.16 特殊方法	81
5.16.1 n^k 的高三位	81
5.16.2 约数个数之和, 定义 $d(i)$ 为 i 的约数个数	81
5.16.3 给定 k , 求最小的 n 使得 n 的约数个数恰为 $n - k$ 个 ($k \leq 47777$)	83
5.16.4 $C_n^m \pmod{p}$ ($p = p_1 * p_2$, 且 p_1, p_2 为素数)	83
6 数学相关	85
6.1 Polya 原理、Burnside 引理	85
6.1.1 介绍	85
6.1.2 Burnside 引理	85
6.1.3 Polya 原理	86
6.1.4 $n * n$ 的方阵, 每个小格可涂 m 种颜色, 求在旋转操作下本质不同的解的总数	86
6.1.5 各有 a, b, c ($a, b, c \geq 0, a + b + c \leq 40$) 颗三种颜色, 问这些珠子能串成的项链有多少种? 考虑翻转和旋转。	88
6.1.6 给出 12 根等长的火柴棒, 每根火柴棒的颜色属于 1 – 6 中的一种, 问能拼成多少种不同的正方体? (考虑旋转)	90
6.1.7 [POJ 2888 Magic Bracelet]	91
6.2 容斥原理	94
6.2.1 求区间 $[A, B]$ 中和 n 互素的数个数? $1 \leq A \leq B \leq 10^{15}, 1 \leq n \leq 10^9$	94
6.2.2 给一个 n , 在 $p \in [1, n]$ 范围满足 $m^k = p$ ($m \geq 1, k > 1$) 的数字 p 的个数。 $1 \leq n \leq 10^{18}$	95
6.3 博弈论	96
6.3.1 威佐夫博弈 (<i>Wythoff Game</i>)	96
6.4 约瑟夫环问题	97
6.4.1 第一次第 m 个人出列, 以后每次第 k 个人出列	97
6.4.2 每次都是第 2 个人出列, 求 n 个人最终剩下的编号 (编号从 1 开始)	97
6.4.3 第 i 轮从上一轮出局的人的下一个人开始从 1 报数, 报到 i 就停止且报到 i 的这个人出局	97
6.4.4 获得每一轮出列的人的编号	98
6.5 康托展开	100

6.6 FFT	102
6.6.1 UVA 12633 超级车	103
6.6.2 SPOJ Triple Sum	103
6.6.3 HDU 5730 Shell Necklace	104
6.7 NTT	105
6.7.1 HDU 5322 Hope	106
6.7.2 HDU 5552 图计数	107
6.7.3 HDU 5829 卷积平移	108
7 动态规划	111
7.1 LIS	111
7.1.1 最长非降子序列	111
7.1.2 三维 LIS	111
7.2 区间 dp	114
7.2.1 LightOJ 1422	114
7.2.2 PKU 1142	114
7.2.3 CF 149 D	115
7.3 树型 dp	118
7.3.1 树的直径	118
7.3.2 求树直径的方法	118
7.3.3 HDU 4126	120
7.4 数位 dp	123
7.4.1 2014 GCJ Round 1B	123
7.4.2 区间最大上升子序列长度为 K 的数字个数	123
7.4.3 区间所有和 7 “无关” 数字平方和	124
7.4.4 SGU 390 Tickets	126
7.5 状压 dp	128
7.5.1 覆盖模型	128
7.5.2 图论模型	138
7.6 dp 优化	142
7.6.1 二进制优化	142
7.6.2 单调队列优化	142
7.6.3 斜率优化	145
7.6.4 四边形不等式优化	148
7.6.5 bitset 优化	151
8 数据结构	155
8.1 哈希	155
8.2 并查集	157
8.2.1 加权并查集	157
8.2.2 分层并查集	158
8.3 RMQ	164
8.4 线段树	165
8.4.1 矩形并的周长	165
8.4.2 矩形交的面积	166
8.4.3 矩形并面积	168
8.5 树状数组	171
8.5.1 单点更新，区间求和	171
8.5.2 区间更新，单点求值	171
8.5.3 一维区间更新和区间求和	171
8.5.4 二维区间更新和区间求和	172
8.6 整体二分	175
8.6.1 [POJ 2104: 区间第 k 小]	175
8.6.2 [BZOJ 3110 第 k 大数]	176
8.7 cdq 分治 (时间分治)	178
8.7.1 [NEU 1702 三维逆序对]	178
8.7.2 [UVALive 6776 三维 LIS]	180

8.7.3 [BZOJ 3295 动态逆序对]	181
8.7.4 [BZOJ 2244 导弹拦截]	183
8.7.5 [BZOJ 1176 Mokia]	185
8.7.6 [BZOJ 1429 货币兑换 Cash]	187
8.7.7 [BZOJ 2961 共点圆]	189
9 图论	193
9.1 最小生成树	193
9.1.1 Kruskal Algorithm	193
9.1.2 Prim Algorithm	193
9.2 拓扑排序	195
9.3 欧拉回路	196
9.3.1 判断	196
9.3.2 n 个点和 m 条无向边的图, 每条边仅可遍历一次, 求图中欧拉路径的条数	196
9.3.3 单词拼接, 字典序最小	197
9.3.4 正向反向两次遍历所有边, 路径输出	199
9.4 dfs 序	201
9.4.1 CF 396 C	201
9.5 LCA	203
9.5.1 暴力求解	203
9.5.2 二分搜索	203
9.5.3 基于 RMQ 的算法	204
9.5.4 Tarjan 算法	205
9.5.5 利用 LCA 获得树上任意两点距离	206
9.6 最小树形图	209
9.6.1 朱刘算法	209
9.6.2 有固定根	209
9.6.3 无固定根	211
9.6.4 最小树形图路径	213
9.6.5 其他	215
9.7 生成树计数	216
9.7.1 Matrix-Tree 定理	216
9.7.2 选择一些边连通使得任意两点之间恰好有一条路径, 求不同的选择方案数	216
9.7.3 最小生成树计数	217
9.8 最短路	220
9.8.1 Dijkstra + HeapNode	220
9.8.2 Bellman_Ford	220
9.8.3 SPFA	221
9.8.4 Floyd_Warshall	221
9.9 二分图匹配	222
9.9.1 概念	222
9.9.2 二分图的性质	223
9.9.3 二分图的判定	223
9.9.4 匈牙利算法 (Hungary Algorithm)	224
9.9.5 输出最小点覆盖的点	226
9.9.6 霍普克洛夫特 - 卡普算法	227
9.9.7 二分图带权匹配	229
9.9.8 二分图带权匹配拆点	232
9.9.9 稳定婚姻匹配	232
9.10 普吕弗序列	234
9.10.1 由无根树生成普吕弗序列	234
9.10.2 从普吕弗序列还原无根树	234
9.10.3 结论	234
9.10.4 BZOJ 1211	235
9.10.5 HDU 5629	235

10 字符串	237
10.1 Shift-And 算法	237
10.2 字典树	239
10.2.1 HDU 1857	239
10.2.2 HDU 5536	240

Chapter 1

基础

1.1 常见错误

1. 递归时隐藏的修改了全局变量，例如点分治重心，树型 dp → 每次复制一遍，开 vector
2. 测试数据时未将空间开到题目要求，隐藏的空间倍数关系，例如无向图 2 倍 (RE)
3. 除数是个减法式子: 整数: RE, 浮点数: WA → 特判
4. 乘法取模, $a * b \rightarrow a \% mod * (b \% mod) \% mod$, 必要时用快速乘法 (注意正负数)
5. *two pointers* 的时候，相等时移动指针 → 小心重复数据，死循环
6. 利用欧拉定理降幂的时候 $x \% mod \rightarrow$ 特判 $x \% mod == 0$, 此时答案是 0 啊
7. dp 时尽量滚动数组，防止 MLE
8. 几何题能有 int 写的时候不要用 double
9. C++ 关闭同步后不要 cin/cout 和 scanf/printf 混用，会 WA 的！
10. 如果出现斜率比较，使用叉积比较比使用斜率要安全！坐标要是 $1e9 * 1e9$ 或者 $1e9 * 1e18$ 级的开 double！
11. C++ 引用数组时，如果 void fun(int *arr) 虽然可以对数组 arr 进行赋值，改值操作，但是不能 memset(), 正确的方式应是: void fun(int (&arr)[N]), 其中 N 是数组长度。

1.2 思考角度

- random_shuffle()
- 二分思想
- 考虑问题反面
- 借助 bfs/dfs 状态预处理

1.3 奇巧淫技

1.3.1 memset 初始化极大极小值和较大较小值

```

1  typedef long long ll;
2  const ll INF = 0x3f3f3f3f3f3f3f3fll; // 末尾的 ll 必不可少
3
4  memset(arr, 0x3f, sizeof(arr)); // set int to 1061109567
5  memset(arr, 0xc0, sizeof(arr)); // set int to -1061109568
6  memset(arr, 0x7f, sizeof(arr)); // set int to 2139062143
7  memset(arr, 0x80, sizeof(arr)); // set int to -2139062144
8
9  memset(arr, 0x3f, sizeof(arr)); // set long long to 4.577e+18
10 memset(arr, 0xc0, sizeof(arr)); // set long long to -4.577e+18
11 memset(arr, 0x7f, sizeof(arr)); // set long long to 9.187e+18
12 memset(arr, 0x80, sizeof(arr)); // set long long to -9.187e+18
13
14 memset(arr, 0x43, sizeof(arr)); // set double to 1.08e+16
15 memset(arr, 0xc2, sizeof(arr)); // set double to -4.13e+13
16 memset(arr, 0x7f, sizeof(arr)); // set double to 1.38e+306
17 memset(arr, 0xfe, sizeof(arr)); // set double to -5.3e+303

```

1.3.2 二进制中 1 的个数

```

1  __builtin_popcount = int
2  __builtin_popcountl = long int
3  __builtin_popcountll = long long

```

1.3.3 vector 去重

```

1  vector<int>::iterator it = unique(vec.begin(), vec.end());
2  vec.erase(it, vec.end());

```

1.3.4 快速乘法

```

1  inline ll mulmod(ll x, ll y, ll mod) //正负数相乘均可
2  {
3      ll ret = 0;
4      __asm__ ("movq %1%%rax\n imulq %2\n idivq %3\n":
5          "=d"(ret) : "m"(x), "m"(y), "m"(mod) : "%rax");
6      return ret;
7  }
8
9  //非汇编版本
10 ll mulmod(ll x, ll y, ll mod) //只能处理正数相乘的情况
11 {
12     ll ret = 0, tmp = x;
13     while (y) {
14         if (y & 1) ret = (ret + tmp) % mod;
15         y >= 1;
16         tmp = (tmp + tmp) % mod;
17     }
18     return ret;
19 }

```

1.3.5 快速开方

```

1 float InvSqrt(float x)
2 {
3     int i;
4     float xhalf = 0.5 * x, y = x;
5     i = *(int *)&y;
6     i = 0x5f375a86 - (i >> 1);
7     y = *(float *)&i;
8     y = y * (1.5 - (xhalf * y * y));
9     y = y * (1.5 - (xhalf * y * y));
10    y = y * (1.5 - (xhalf * y * y));
11    return x * y;
12 }
```

1.3.6 高精度开根 $a^{\frac{1}{b}}$

由于是找最接近的，可以先大致确定一个数 r ，那么可以通过 $r = \text{pow}(a, 1/b)$ 来计算，然后我们分别计算 $(r - 1)^b, r^b, (r + 1)^b$ ，然后看这三个数哪个最接近 a 就行了。

```

1 const double eps = 1e-8;
2 const ll inf = (ll)(1e18) + 500;
3 const ll INF = (ll)1 << 31;
4 ll quick_pow(ll a, ll b) //快速幂求  $a^b$ ，考虑溢出
5 {
6     ll res = 1, tmp = a;
7     while(b) {
8         if(b & 1) {
9             double judge = 1.0 * inf / res;
10            if(tmp > judge) return -1;
11            res *= tmp;
12        }
13        b >>= 1;
14        if(tmp > INF && b > 0) return -1;
15        tmp = tmp * tmp;
16    }
17    return res;
18 }
19
20 ll find(ll a, ll b) { //比较精确地求  $a^{(1/b)}$ （向下取整）
21     ll r = (ll)(pow((double)(a), 1.0 / b) + eps);
22     ll p = quick_pow(r, b);
23     if(p == a) return r;
24     if(p == -1 || p > a) return r - 1;
25
26     ll t = quick_pow(r + 1, b);
27     if(t != -1 && t <= a) return r + 1;
28     else return r;
29 }
```

1.3.7 把所有点（向量）从第三象限开始逆时针排序

```

1 bool cmp(const Point& a, const Point& b)
2 {
3     if (1ll * a.y * b.y <= 0) {
4         if (a.y > 0 || b.y > 0) return a.y < b.y;
5         if (a.y == 0 && b.y == 0) return a.x < b.x;
6     }
7     return a.cross(b) > 0;
8 }
```

1.3.8 求长度为 n 的有序数组 a 中 k 的个数

`lower_bound()` 是在已排序数组中二分查找指向满足 $a[i] \geq k$ 的 $a[i]$ 的最小指针;
`upper_bound()` 是在已排序数组中二分查找指向满足 $a[i] > k$ 的 $a[i]$ 的最小指针。

```
1 cnt = upper_bound(a, a + n, k) - lower_bound(a, a + n, k);
```

1.3.9 C++ 关闭同步

```
1 ios_base::sync_with_stdio(0); cin.tie(0);
```

1.3.10 cout 浮点数控制精度输出

```
1 #include <iomanip>
2 cout << fixed << setprecision(6) << ans << endl;
```

1.3.11 快读

初级版:

```
1 inline int Read(){
2     int x = 0; char ch = getchar();
3     while(ch < '0' || ch > '9') ch = getchar();
4     while(ch >= '0' && ch <= '9') {x = x * 10 + ch - '0'; ch = getchar();}
5     return x;
6 }
```

强化版:

```
1 struct FastIO {
2     static const int S = 1000000;
3     int wpos, pos, len;
4     char wbuf[S];
5     FastIO():wpos(0){}
6     inline int xchar() {
7         static char buf[S];
8         if (pos == len) pos = 0, len = fread(buf, 1, S, stdin);
9         if (pos == len) return -1;
10        return buf[pos++];
11    }
12    inline int xint() {
13        int c = xchar(), x = 0;
14        while (c <= 32 && ~c) c = xchar();
15        if (c == -1) return -1;
16        for (; c >= '0' && c <= '9'; c = xchar()) x = x * 10 + (c - '0');
17        return x;
18    }
19 } io;
// 调用
21 n = io.xint();
// 读完标记
22 if (n == -1) break;
```

1.4 结论

1.4.1 判断 n 个整数坐标能否构成正 n 边形

给出 n 个二维坐标点且坐标都是整数，判断这 n 个整数点能否构成正 n 边形。因为坐标是整数点，所以当且仅当 $n = 4$ 时才有可能构成正四边形（正方形）。

判断正方形的方法：（共 6 条边）将这 n 个点的两两距离求出，从小到大排序，最短的四条边一定相等且是正方形的边长，而且要保证对角线也相等且是边长的根号 2 倍（去边长的平方的话就是 2 倍）。对角线是排序后的最长两条边。

```

1  scanf("%d", &n);
2  for (int i = 0; i < n; i++) {
3      scanf("%d%d", &x[i], &y[i]);
4  }
5  if (n != 4) {
6      printf("NO\n");
7      continue;
8  }
9  int total = 0;
10 for (int i = 0; i < n; i++) {
11     for (int j = i + 1; j < n; j++) {
12         edge[total++] = dis(i, j);
13     }
14 }
15 sort(edge, edge + total);
16 if (edge[0] == edge[1] && edge[1] == edge[2] &&
17     edge[2] == edge[3] && edge[4] == edge[5] &&
18     edge[4] == 2 * edge[3]) {
19     printf("YES\n");
20 } else {
21     printf("NO\n");
22 }
```

1.4.2 已知年月日计算星期几

```

1  inline int GetWeek(int year, int month, int day)
2  {//返回值:1--7
3      //把一月二月当作前一年的 13,14 月
4      if (month == 1 || month == 2) {
5          month += 12;
6          year--;
7      }
8      //判断是否在 1752 年 9 月 3 日之前
9      if ((year < 1752) || (year == 1752 && month < 9)
10         || (year == 1752 && month == 9 && day < 3)) {
11         return (day + month * 2 + 3 * (month + 1) / 5
12                 + year + year / 4 + 5) % 7 + 1;
13     } else {
14         return (day + month * 2 + 3 * (month + 1) / 5
15                 + year + year / 4 - year / 100 + year / 400) % 7 + 1;
16     }
17 }
```

1.4.3 方格染色

给一个 $n * m$ 的方格，可以选择从中选择一个矩形的两个顶点然后把这个矩形内的所有方格都染色，求坐标为 (i, j) 的方格被染色的概率？

```

1  double GetProbability(int i, int j)
2  {
3      double res = 1.0 * (i * j - 1) * ((n - i + 1) * (m - j + 1) - 1);
4      res += 1.0 * (i * (m - j + 1) - 1) * ((n - i + 1) * j - 1);
```

```

5     res == (i - 1) * (n - i);
6     res == (j - 1) * (m - j);
7     res = res * 2.0 + n * m * 2.0 - 1;
8     return res / n / n / m / m;
9 }
```

1.4.4 包含特定方格所有子矩阵面积和

一个位于 (x, y) 的方格，向左向右向上向下最多分别可以延伸: a, b, c, d 个单位，求包含这个方格的所有子矩阵的面积和。

$$\begin{aligned}
S &= \sum (L + R - 1) * (U + D - 1) \quad (1 \leq L \leq a, 1 \leq R \leq b, 1 \leq U \leq c, 1 \leq D \leq d) \\
&= \sum_{1 \leq L \leq a, 1 \leq R \leq b} (L + R - 1) * \sum_{1 \leq U \leq c, 1 \leq D \leq d} (U + D - 1) \\
&= \left(\sum_{L=1}^{L=a} L * b + \sum_{R=1}^{R=b} R * a - a * b \right) * \left(\sum_{U=1}^{U=c} U * d + \sum_{D=1}^{D=d} D * c - c * d \right) \\
&= \left(\frac{a * (a + 1)}{2} * b + \frac{b * (b + 1)}{2} * a - a * b \right) * \left(\frac{c * (c + 1)}{2} * d + \frac{d * (d + 1)}{2} * c - c * d \right) \\
&= \frac{a * b * c * d * (a + b) * (c + d)}{4}
\end{aligned} \tag{1.1}$$

1.4.5 n 以内 $\gcd(a, b) = d$ 的无序对对数

$$\begin{aligned}
num[d] &= \sum_{a=1}^{a=n} \sum_{b=1}^{b=n} [\gcd(a, b) == d] \\
&= 2 * (\phi(1) + \phi(2) + \phi(3) + \cdots + \phi(\frac{n}{d})) - 1 \\
&= 2 * sum[\frac{n}{d}] - 1
\end{aligned} \tag{1.2}$$

$sum[]$ 是欧拉函数的前缀和, $sum[1] = 1$

1.4.6 $\gcd(x^a - 1, x^b - 1) = x^{\gcd(a,b)} - 1$

1.5 归并排序及利用归并排序求逆序对数

因为合并操作是从小到大进行的，当右边的 $a[q]$ 复制到 T 中时，左边还没来得及复制到 T 中的那些数就是左边所有比 $a[q]$ 大的数。此时在累加器中加上左边的元素个数 $m - p$ 即可（左边剩余元素在区间 $[p, m)$ 中，因此元素个数为 $m - p$ ）。

```

1 int a[10] = { 2,1,7,9,13,11,12,20 }, T[10];
2 int cnt;
3
4 void MergeSort(int* a, int low, int high, int* T)
5 { //将数组 a[low...high-1] 从小到大排序, 时间复杂度为 O(nlogn)
6     if (low + 1 < high)
7     {
8         int mid = (low + high) / 2;
9         MergeSort(a, low, mid, T); //排序 a[low...(mid-1)]
10        MergeSort(a, mid, high, T); //排序 a[mid...(high-1)]
11        int p = low, q = mid, i = low;
12        while (p < mid || q < high) //只要有一个序列非空就继续合并
13        {
14            if (q >= high || (p < mid && a[p] < a[q]))
15                //复制第一个序列:
16                // 1 : 第二个序列空了
17                // 2 : 第二个序列非空并且第一个序列也非空且 a[p]<a[q]
18                T[i++] = a[p++];
19            else //复制第二个序列
20            {
21                T[i++] = a[q++];
22                cnt += mid - p;
23                for (int j = p; j < mid; j++) //输出逆序对
24                    printf("%d — %d\n", a[j], a[q - 1]);
25            }
26        }
27        for (int i = low; i < high; i++)
28            a[i] = T[i];
29    }
30}
31
32 int main()
33 {
34     printf("原来数组是: \n");
35     for (int i = 0; i < 8; i++)
36         printf("%d ", a[i]);
37     printf("\n");
38     cnt = 0; //cnt 是逆序对数
39     MergeSort(a, 0, 8, T);
40     printf("排序后的数组是: \n");
41     for (int i = 0; i < 8; i++)
42         printf("%d ", a[i]);
43     printf("\n");
44     printf("逆序对数是: %d\n", cnt);
45     return 0;
46 }
```

```

1 /******Binary Indexed Tree*****/
2 const int maxn = 500050;
3
4 int n, tot;
5 int bits[maxn];
6 long long data[maxn], tmp[maxn];
7
8 inline void update(int pos)
9 {
10     while (pos <= tot){
11         bits[pos]++;
12         pos += (pos & (-pos));
```

```

13     }
14 }
15
16 inline int solve(int pos)
17 {
18     int sum = 0;
19     while(pos > 0){
20         sum += bits[pos]; // bits[pos] 存储从 1--id 中数字的个数
21         pos -= (pos & (-pos));
22     }
23     return sum;
24 }
25
26 int main()
27 {
28     while (~scanf("%d",&n)&&n){
29         for (int i = 1; i <= n; i++) {
30             scanf("%lld",&data[i]);
31             tmp[i] = data[i];
32         }
33         sort(tmp + 1, tmp + n + 1);
34         tot = unique(tmp + 1, tmp + 1 + n) - tmp;
35         long long ans = 0;
36         memset(bits, 0, sizeof(bits));
37         /*
38         for (int i = n; i >= 1; i--) {
39             int id = lower_bound(tmp + 1, tmp + tot, data[i]) - tmp;
40             ans += solve(id);
41             // solve(id) 得到的是树状数组中比 data[i] 小的数字个数,
42             // 因为是倒着插入树状数组的, 那么这个 solve(id) 就是 data[i] 的逆序数了
43             update(id);
44         }
45         */
46         //两种写法都可以
47         for (int i = 1; i <= n; i++){
48             int id = lower_bound(tmp + 1,tmp + tot, data[i]) - tmp;
49             ans += (i - solve(id) - 1);
50             //正着插入的话, solve(id) 就是 1~i-1 中比 data[i] 小的数字个数
51             //一共是 i-1 个数去掉( data[i] , 所以由) data[i] 引起的逆序对数是 i-1-solve(id)
52             update(id);
53         }
54         printf("%lld\n",ans);
55     }
56     return 0;
57 }
```

1.6 快排、利用快排查找第 k 大元素

输入 $n \leq 10^7$ 个整数和一个正整数 k ($1 \leq k \leq n$)，输出这些整数从小到大排序后的第 k 个（例如， $k = 1$ 就是最小值）。

快速排序的时间复杂度为：最坏情况下： $O(n^2)$ ，平均情况下： $O(n \log n)$ 。

查找数组中第 k 大的元素的平均时间复杂度为： $O(n)$ 。

```

1 int a[10] = { 2,1,7,9,13,11,20,12 };
2
3 void QuickSort(int* a, int low, int high)//将数组 a[low...high] 从小到大排序
4 {
5     if (low < high) {
6         int i = low, j = high;
7         int pivot = a[low]; //选取 a[low] 为基准
8         while (i < j) {
9             while (i < j && a[j] > pivot) j--;// j 从右向左找比 pivot 小的数
10            if (i < j) a[i++] = a[j];
11        }
12        a[i] = pivot;
13        QuickSort(a, low, i - 1);
14        QuickSort(a, i + 1, high);
15    }
16 }
```

```

11     while ( i < j && a[ i ] < a[ j ]) i++; // i 从左向右找比 pivot 大的数
12     if ( i < j ) a[ j-- ] = a[ i ];
13   }
14   a[ i ] = pivot; // 将基准数插到“中间”
15   QuickSort( a, low, i - 1 ); // 将左边排序
16   QuickSort( a, i + 1, high ); // 将右边排序
17 }
18 }
19
20 int find_kth_smallest( int* a, int low, int high, int k )
21 { // 在数组 a[low...high] 中查找第 k 小的数
22   if ( low < high ) {
23     int i = low, j = high;
24     int pivot = a[ low ];
25     while ( i < j ) {
26       while ( i < j && a[ j ] > pivot ) j--;
27       if ( i < j ) a[ i++ ] = a[ j ];
28       while ( i < j && a[ i ] < pivot ) i++;
29       if ( i < j ) a[ j-- ] = a[ i ];
30     }
31     a[ i ] = pivot;
32     if ( i - low + 1 == k ) return pivot;
33     else if ( i - low + 1 < k )
34       return find_kth_smallest( a, i + 1, high, k - ( i - low + 1 ) );
35     // 第 k 小在右半部分, k 变为 k-(i-low+1)
36     // 因为左部分元素个数为 (i-1)-low+1=i-low 还有一个基准元素, pivot(a[i])
37     else return find_kth_smallest( a, low, i - 1, k );
38     // 第 k 小在左半部分, k 不变
39   }
40   return a[ low ];
41   // 当 low=high 时, k 必然也是 1, 这使要查询的数组中就一个元素, 直接返回就可以了
42 }
43
44 int main()
45 {
46   printf("排序之前的数组: \n");
47   for ( int i = 0; i < 8; i++ ) printf("%d ", a[ i ]);
48   printf("\n");
49   QuickSort( a, 0, 7 );
50   printf("排序之后的数组: \n");
51   for ( int i = 0; i < 8; i++ ) printf("%d ", a[ i ]);
52   printf("\n");
53   /*
54   for ( int k = 1; k < 8; k++ )
55   {
56     printf("第%d小的元素是%d\n", k, find_kth_smallest( a, 0, 7, k ) );
57     printf("查找之后的数组变为(:\n");
58     for ( int i = 0; i < 8; i++ )
59       printf("%d ", a[ i ]);
60     printf("\n");
61   }
62   */
63   return 0;
64 }
```

1.7 矩阵快速幂

```

1 const int MAX_SIZE = 20;
2 struct Matrix{
3   int row, col;
4   ll data[MAX_SIZE][MAX_SIZE];
5
6   Matrix operator * ( const Matrix& rhs ) const {
```

```

7 //矩阵相乘条件: col = rhs.row
8     Matrix res;
9     res.row = row, res.col = rhs.col;
10    for(int i = 1; i <= res.row; ++i) {
11        for(int j = 1; j <= res.col; ++j) {
12            res.data[i][j] = 0;
13            for(int k = 1; k <= row; ++k) {
14                res.data[i][j] += data[i][k] * rhs.data[k][j] % mod;
15                if(res.data[i][j] >= mod) res.data[i][j] -= mod;
16            }
17        }
18    }
19    return res;
20 }

21 Matrix operator ^ (const int m) const { //矩阵快速幂
22     Matrix res, tmp;
23     res.row = row, res.col = col; //row == col
24     memset(res.data, 0, sizeof(res.data));

25     tmp.row = row, tmp.col = col;
26     memcpy(tmp.data, data, sizeof(data));
27     for(int i = 1; i <= res.row; ++i) { res.data[i][i] = 1; }
28     int mm = m;
29     while(mm) {
30         if(mm & 1) res = res * tmp;
31         tmp = tmp * tmp;
32         mm >>= 1;
33     }
34     return res;
35 }
36
37 }
38 };

```

1.7.1 循环矩阵

[UVA 1386 Cellular Automaton]

给一个 $n \leq 500$ 个数组成的数环，每次取每个数左右范围 $0 \leq d < \frac{n}{2}$ 的所有数（包括本身和距离是 d 的数）相加和然后 $\bmod m$ 生成新的数，问操作 $k \leq 10^7$ 次后的数环是怎样的？

将二维矩阵压缩为一维每次只保留第一行数字，通过手动模拟原始矩阵的乘法可以到这个式子：

$$A[j] * B[(j - i + n) \% n]$$

其中下标都是从 0 开始到 $n - 1$ 。时间复杂度： $O(n^2 * \log k)$

```

1 const int MAX_N = 510;
2
3 int n, mod, d, K;
4 ll C[MAX_N], D[MAX_N], M[MAX_N], ret[MAX_N], ans[MAX_N];
5
6 inline void mul(ll *arr1, ll *arr2) { // arr2 = arr1 * arr2
7     memset(ret, 0, sizeof(ret));
8     for (int i = 0; i < n; ++i) {
9         for (int j = 0; j < n; ++j) {
10             ret[i] += arr1[j] * arr2[(j - i + n) % n] % mod;
11             if (ret[i] >= mod) ret[i] -= mod;
12         }
13     }
14     memcpy(arr2, ret, sizeof(ret));
15 }
16
17 void Qpower() {

```

```

18     memset (M, 0, sizeof (M));
19     M[0] = 1;
20     while (K) {
21         if (K & 1) mul(C, M);
22         mul(C, C);
23         K >= 1;
24     }
25 }
26
27 int main() {
28     while (~scanf ("%d%d%d%d", &n, &mod, &d, &K)) {
29         for (int i = 0; i < n; i++) {
30             ll x;
31             scanf ("%lld", &x);
32             D[i] = x % mod;
33         }
34         memset (C, 0, sizeof (C));
35         for (int i = 0; i < 2 * d + 1; i++) {
36             C[(n - d + i) % n] = 1;
37         }
38         Qpower();
39         memset (ans, 0, sizeof (ans));
40         for (int i = 0; i < n; i++) {
41             for (int j = 0; j < n; j++) {
42                 int ind = (j - i + n) % n;
43                 ans[i] += D[j] * M[(j - i + n) % n] % mod;
44                 if (ans[i] >= mod) ans[i] -= mod;
45             }
46         }
47         for (int i = 0; i < n; i++) {
48             if (i) printf (" ");
49             printf ("%lld", ans[i]);
50         }
51         printf ("\n");
52     }
53     return 0;
54 }
```

1.8 高精度

因为计算大数除法时需要用到乘法和减法，但是不指定字符串长度的乘法和减法不容易用字符数组表示，所以这里就没写用字符数组计算的大数除法。

```

1 /*大数加减乘仅限正整数***** */
2 //加法测试: HDU 1002
3 //减法测试: 百练OJ 2736
4 //乘法测试: 百练OJ 2980
5
6 const int MAX_N = 100010;
7
8 char a[MAX_N], b[MAX_N], ope[10], ans[MAX_N];
9 int data[MAX_N];
10
11 void Big_Plus()
12 {
13     int lena = strlen(a), lenb = strlen(b);
14     if (lena >= lenb){ //a 位数比 b 多
15         for (int i = lenb; i >= 0; i--) //遍历 b 中所有元素
16             b[i + (lena - lenb)] = b[i]; //数字后移, 使末位与 a 对齐
17         for (int i = 0; i < lena - lenb; i++)//将数字 b 中高位上多出来的部分遍历赋0
18             b[i] = '0';
19     } else {
20         for (int i = lena; i >= 0; i--)
21             a[i + (lenb - lena)] = a[i];
```

```

22     for (int i = 0; i < lenb - lena; i++)
23         a[i] = '0';
24     }
25     int carry = 0, lenans = max(lena, lenb);
26     ans[lenans] = '\0'; //添加结束符
27     int tmp = lenans - 1;
28     while (tmp >= 0) {//循环条件： a 和 b 对应位上数字加完
29         int x = a[tmp] - '0';
30         int y = b[tmp] - '0';
31         int z = x + y + carry;
32         if(z >= 10) carry = z / 10;
33         else carry = 0;
34         z %= 10;
35         ans[tmp] = z + '0';
36         tmp--;
37     }
38     if(carry) {//最高位计算完仍有进位
39         for (int i = lenans; i >= 0; i--) //lenans 是考虑到结束符
40             ans[i+1] = ans[i]; //后移
41         ans[0] = carry + '0'; //最高位为carry
42     }
43 }
44
45 void Big_Sub()
46 {
47     int lena = strlen(a); // ca 为 a 的长度
48     int lenb = strlen(b); // cb 为 b 的长度
49     if (lena > lenb || lena == lenb && strcmp(a, b) >= 0){
50         // a 的长度大于 b 或 a 的长度等于 b 且字符串 a>=b 结果为正，
51         int i, j;
52         for(i = lena - 1, j = lenb - 1; j >= 0; i--, j--)
53             a[i] -= (b[j] - '0');
54         for(i = lena - 1; i >= 0; i--) { //遍历 a 的所有下标
55             if (a[i] < '0') { //如果当前下标对应值小于 0，则借位操作
56                 a[i] += 10; //当前值加10
57                 a[i-1]--; //高位减1
58             }
59         }
60         i = 0;
61         //去除前导0
62         while (a[i] == '0' & & i < lena - 1) i++;
63         strcpy(ans, a + i); //将结果复制到 ans 数组
64     } else {//类似上面部分
65         int i, j;
66         for (i = lena - 1, j = lenb - 1; i >= 0; i--, j--)
67             b[j] -= (a[i] - '0');
68         for(j = lenb - 1; j >= 0; j--) {
69             if(b[j] < '0'){
70                 b[j] += 10;
71                 b[j - 1]--;
72             }
73         }
74         j = 0;
75         while(b[j] == '0' && j < lenb - 1) j++;
76         ans[0] = '-'; //运算结果为负
77         strcpy(ans + 1, b + j);
78     }
79 }
80
81 void Big_Mul()
82 {
83     int lena = strlen(a), lenb = strlen(b);
84     int lenans = lena + lenb - 1;
85     for(int i = 0; i <= (lena - 1) / 2; i++) swap(a[i], a[lena - 1 - i]); //数组逆置
86     for(int i = 0; i <= (lenb - 1) / 2; i++) swap(b[i], b[lenb - 1 - i]);

```

```

87     memset(data, 0, sizeof(data));
88     for (int i=0; i<lena; i++){
89         for (int j=0; j<lenb; j++){
90             data[i+j]=(a[i]-‘0’)*(b[j]-‘0’); //模拟乘法计算每一位,
91         }
92         int carry=0;
93         for (int i=0; i<lenans; i++){
94             int tmp=data[i]+carry;
95             carry=tmp/10;
96             data[i]=tmp%10;
97         }
98         while(carry){
99             data[lenans++]=carry%10;
100            carry/=10;
101        }
102        while(data[lenans-1]==0&&lenans>1) lenans--;
103        for (int i=0; i<=lenans-1; i++) ans[i]=data[lenans-1-i]+‘0’;
104        ans[lenans]=‘\0’; //添加结束符
105    }
106
107    int main()
108    {
109        //freopen("BigIntin.txt", "r", stdin);
110        //输入运算数和运算符之间有空格:
111        while(~scanf("%s%s%s", a, ope, b))
112        {
113            printf("%s%s%s=%", a, ope, b);
114            if(ope[0]=='+'){
115                Big_Plus();
116            } else if(ope[0]=='-'){
117                Big_Sub();
118            } else if(ope[0]=='*'){
119                Big_Mul();
120            }
121            puts(ans);
122        }
123        return 0;
124    }

```

```

1 /*大数加减乘除*****C++类实现string******/
2 //运算数仅限正整数
3
4 //加法测试: HDU 1002
5 //减法测试: 百练OJ 2736
6 //乘法测试: 百练OJ 2980
7 //除法测试: 百练OJ 2737
8
9 //string 比较函数相等返回: 0 , str1>str2 返回 1 , str1<str2 返回-1
10 int Compare(string str1, string str2)
11 {
12     if(str1.length() > str2.length()) return 1;
13     else if(str1.length() < str2.length()) return -1;
14     else return str1.compare(str2);
15 }
16
17 string Big_Plus(string str1, string str2)
18 {
19     string ans;
20     int len1=str1.length();
21     int len2=str2.length();
22     //将长度较小的前面补 0 , 使两个 string 长度相同
23     if(len1<len2){
24         for (int i=1; i<=len2-len1; i++){
25             str1="0"+str1;
26         }

```

```
27 } else {
28     for( int i=1;i<=len1-len2 ;i++){
29         str2="0"+str2;
30     }
31 }
32 int len=max(len1 ,len2 );
33 int carry=0;
34 for( int i=len -1;i>=0;i--){
35     int tmp=str1 [ i]- '0 '+str2 [ i]- '0 '+carry ;
36     carry=tmp/10;
37     tmp%==10;
38     ans=char(tmp+ '0 ')+ans ;
39 }
40 if(carry) ans=char(carry+ '0 ')+ans ;
41 return ans ;
42 }
43
44 //支持大数减小数
45 string Big_Sub(string str1 ,string str2 )
46 {
47     string ans ;
48     int carry=0;
49     int difference=str1 .length ()-str2 .length ()//长度差
50     for( int i=str2 .length ()-1;i>=0;i--){
51         if(str1 [ difference+i]<str2 [ i]+carry ){
52             ans=char(str1 [ difference+i]+10-str2 [ i]-carry+ '0 ')+ans ;
53             carry=1;
54         } else {
55             ans=char(str1 [ difference+i]-str2 [ i]-carry+ '0 ')+ans ;
56             carry=0;
57         }
58     }
59     for( int i=difference -1;i>=0;i--){
60         if(str1 [ i]-carry>='0 '){
61             ans=char(str1 [ i]-carry)+ans ;
62             carry=0;
63         } else {
64             ans=char(str1 [ i]-carry+10)+ans ;
65             carry=1;
66         }
67     }
68 //去除前导0
69 ans .erase(0 ,ans .find_first_not_of('0 '));
70 if(ans .empty ()) ans="0 ";
71 return ans ;
72 }
73
74 string Big_Mul(string str1 ,string str2 )
75 {
76     string ans ;
77     int len1=str1 .length ();
78     int len2=str2 .length ();
79     for( int i=len2 -1;i>=0;i--){
80         string tmpstr="";
81         int data=str2 [ i]- '0 ';
82         int carry=0;
83         if(data!=0){
84             for( int j=1;j<=len2 -1-i ;j++){
85                 tmpstr+="0 ";
86             }
87             for( int j=len1 -1;j>=0;j--){
88                 int t=data*(str1 [ j]- '0 ')+carry ;
89                 carry=t /10;
90                 t%==10;
91                 tmpstr=char(t+ '0 ')+tmpstr ;
92             }
93         }
94     }
95 }
```

```

92         }
93         if(carry!=0) tmpstr=char(carry+'0')+tmpstr;
94     }
95     ans=Big_Plus(ans,tmpstr);
96 }
97 ans.erase(0,ans.find_first_not_of('0'));
98 if(ans.empty()) ans="0";
99 return ans;
100 }

101 //正数相除，商为 quotient 余数为， residue
102
103 void Big_Div(string str1,string str2,string& quotient,string& residue)
104 {
105     quotient=residue=""; //商和余数清空
106     if(str2=="0"){//判断除数是否为0
107         quotient=residue="ERROR";
108         return;
109     }
110     if(str1=="0"){//判断被除数是否为0
111         quotient=residue="0";
112         return;
113     }
114     int res=Compare(str1,str2);
115     if(res<0){//被除数小于除数
116         quotient="0";
117         residue=str1;
118         return;
119     }else if(res==0){
120         quotient="1";
121         residue="0";
122         return ;
123     }else {
124         int len1=str1.length();
125         int len2=str2.length();
126         string tmpstr;
127         tmpstr.append(str1,0,len2-1); //将 str1 的前 len2 位赋给tmpstr
128         for(int i=len2-1;i<len1;i++){
129             tmpstr=tmpstr+str1[i]; //被除数新补充一位
130             tmpstr.erase(0,tmpstr.find_first_not_of('0'));//去除前导0
131             if(tmpstr.empty()) tmpstr="0";
132             for(char ch='9';ch>='0';ch--) { //试商
133                 string tmp,ans;
134                 tmp=tmp+ch;
135                 ans=Big_Mul(str2,tmp); //计算乘积
136                 if(Compare(ans,tmpstr)<=0){ //试商成功
137                     quotient=quotient+ch;
138                     tmpstr=Big_Sub(tmpstr,ans); //减掉乘积
139                     break;
140                 }
141             }
142         }
143     }
144     residue=tmpstr;
145 }
146     quotient.erase(0,quotient.find_first_not_of('0'));
147     if(quotient.empty()) quotient="0";
148 }

149 int main()
150 {
151     //freopen("BigIntin.txt","r",stdin);
152     string str1,str2,str3,str4;
153     while(cin>>str1>>str2){
154         cout<<Big_Plus(str1,str2)<<endl;
155         cout<<Big_Sub(str1,str2)<<endl;
156

```

```

157     cout<<Big_Mul(str1 ,str2)<<endl;
158     Big_Div(str1 ,str2 ,str3 ,str4 );
159     cout<< "商:" <<str3<<" " << "余数:" <<str4<<endl;
160 }
161 return 0;
162 }
```

```

1 const int SIZE = 11000;
2
3 struct BigInteger {
4     int len , s[SIZE + 5];
5
6     BigInteger () {
7         memset(s, 0, sizeof(s));
8         len = 1;
9     }
10    BigInteger operator = (const char *num) { //字符串赋值
11        memset(s, 0, sizeof(s));
12        len = strlen(num);
13        for(int i = 0; i < len; i++) s[i] = num[len - i - 1] - '0';
14        return *this;
15    }
16    BigInteger operator = (const int num) { //int 赋值
17        memset(s, 0, sizeof(s));
18        char ss[SIZE + 5];
19        sprintf(ss, "%d", num);
20        *this = ss;
21        return *this;
22    }
23    BigInteger (int num) {
24        *this = num;
25    }
26    BigInteger (char* num) {
27        *this = num;
28    }
29    string str() const { //转化成 string
30        string res = "";
31        for(int i = 0; i < len; i++) res = (char)(s[i] + '0') + res;
32        if(res == "") res = "0";
33        return res;
34    }
35    BigInteger clean() {
36        while(len > 1 && !s[len - 1]) len--;
37        return *this;
38    }
39
40    BigInteger operator + (const BigInteger& b) const {
41        BigInteger c;
42        c.len = 0;
43        for(int i = 0, g = 0; g || i < max(len , b.len); i++) {
44            int x = g;
45            if(i < len) x += s[i];
46            if(i < b.len) x += b.s[i];
47            c.s[c.len++] = x % 10;
48            g = x / 10;
49        }
50        return c.clean();
51    }
52
53    BigInteger operator - (const BigInteger& b) {
54        BigInteger c;
55        c.len = 0;
56        for(int i = 0, g = 0; i < len; i++) {
57            int x = s[i] - g;
58            if(i < b.len) x -= b.s[i];

```

```

59         if(x >= 0) g = 0;
60     else {
61         g = 1;
62         x += 10;
63     }
64     c.s[c.len++] = x;
65 }
66 return c.clean();
67 }

68 BigInteger operator * (const int num) const {
69     int c = 0, t;
70     BigInteger pro;
71     for(int i = 0; i < len; ++i) {
72         t = s[i] * num + c;
73         pro.s[i] = t % 10;
74         c = t / 10;
75     }
76     pro.len = len;
77     while(c != 0) {
78         pro.s[pro.len++] = c % 10;
79         c /= 10;
80     }
81     return pro.clean();
82 }
83 }

84 BigInteger operator * (const BigInteger& b) const {
85     BigInteger c;
86     for(int i = 0; i < len; i++) {
87         for(int j = 0; j < b.len; j++) {
88             c.s[i + j] += s[i] * b.s[j];
89             c.s[i + j + 1] += c.s[i + j] / 10;
90             c.s[i + j] %= 10;
91         }
92     }
93     c.len = len + b.len + 1;
94     return c.clean();
95 }
96 }

97 BigInteger operator / (const BigInteger &b) const {
98     BigInteger c, f;
99     for(int i = len - 1; i >= 0; --i) {
100        f = f * 10;
101        f.s[0] = s[i];
102        while(f >= b) {
103            f = f - b;
104            ++c.s[i];
105        }
106    }
107    c.len = len;
108    return c.clean();
109 }
110 //高精度取模
111 BigInteger operator % (const BigInteger &b) const{
112     BigInteger r;
113     for(int i = len - 1; i >= 0; --i) {
114         r = r * 10;
115         r.s[0] = s[i];
116         while(r >= b) r = r - b;
117     }
118     r.len = len;
119     return r.clean();
120 }
121 }

122 bool operator < (const BigInteger& b) const {

```

```

124     if(len != b.len) return len < b.len;
125     for(int i = len - 1; i >= 0; i--)
126         if(s[i] != b.s[i]) return s[i] < b.s[i];
127     return false;
128 }
129 bool operator > (const BigInteger& b) const {
130     return b < *this;
131 }
132 bool operator <= (const BigInteger& b) const {
133     return !(b < *this);
134 }
135 bool operator == (const BigInteger& b) const {
136     return !(b < *this) && !(*this < b);
137 }
138 bool operator != (const BigInteger &b) const {
139     return !(*this == b);
140 }
141 bool operator >= (const BigInteger &b) const {
142     return *this > b || *this == b;
143 }
144 friend istream & operator >> (istream &in, BigInteger& x) {
145     string s;
146     in >> s;
147     x = s.c_str();
148     return in;
149 }
150 friend ostream & operator << (ostream &out, const BigInteger& x) {
151     out << x.str();
152     return out;
153 }
154 };

```

1.8.1 利用 FFT 实现大数乘法

```

1 int main() {
2     while (~scanf("%s%s", sa, sb)) {
3         int lena = strlen(sa), lenb = strlen(sb);
4         int ka = 0, kb = 0;
5         init(max(lena, lenb) * 2 + 1);
6         for (int i = 0; i < N; ++i) A1[i] = A2[i] = CP(0, 0);
7         for (int i = 0; i < lena; ++i) A1[lena - 1 - i].a = sa[i] - '0';
8         for (int i = 0; i < lenb; ++i) A2[lenb - 1 - i].a = sb[i] - '0';
9
10        FFT(A1, N, 1); FFT(A2, N, 1);
11        for (int i = 0; i < N; ++i) A1[i] = A1[i] * A2[i];
12        FFT(A1, N, -1);
13        for (int i = 0; i < N; ++i) {
14            ans[i] = (int)(A1[i].a + 0.5);
15        }
16        int carry = 0;
17        for (int i = 0; i < N; ++i) {
18            int t = ans[i] + carry;
19            ans[i] = t % 10;
20            carry = t / 10;
21        }
22        while (carry) {
23            ans[N++] = carry % 10;
24            carry /= 10;
25        }
26        while (N > 1 && ans[N - 1] == 0) N--;
27        for (int i = N - 1; i >= 0; --i) {
28            printf("%c", ans[i] + '0');
29        }
30        printf("\n");

```

```
31     }  
32     return 0;  
33 }
```


Chapter 2

搜索

2.1 折半搜索

FZU 2178 礼物分配

给 $n(n \leq 30)$ 个物品和这些物品分别对 A 与 B 的价值，然后需要将这些物品给 A 和 B，A 和 B 分别拥有的数量 $numA$ 和 $numB$ 要满足： $numA + numB = n$ 且 $|numA - numB| \leq 1$ 。求两人获得的价值和之差绝对值最小值： $\min(|sumA - sumB|)$ 。

不妨假设第一人取了 $former = \frac{n+1}{2}$ 件物品，第二个人取了 $later = \frac{n}{2}$ 件物品。因为 n 最大为 30，如果直接枚举的话肯定会超时的。考虑第一个人在前 $former$ 个中取 i 个可以获得的价值的所有情况：这个可以状压 dp 递推出来，然后存在一个数组中。接着状压枚举第二个人在后 $later$ 件物品中取的情况，如果第二个人在后 $later$ 件物品中取的数量固定了，那么第一个人在前 $former$ 件物品中取的数量也固定了。只要二分查找这种情况的最优解即可。

时间复杂度： $O(\frac{n+1}{2} * 2^{\frac{n+1}{2}} * \log K)$

```
1 const int MAX_N = 35;
2 const int inf = 0x3f3f3f3f;
3
4 int T, n;
5 int v[MAX_N], w[MAX_N];
6 int store[16][10000], num[16];
7 // C[15][8] 大概 6400 左右
8
9 void solve() {
10     memset(num, 0, sizeof(num));
11     int former = (n + 1) / 2, later = n / 2;
12     for (int s = 0; s < (1 << former); ++s) {
13         int ret1 = 0, ret2 = 0, tmp = 0;
14         for (int i = 0; i < former; ++i) {
15             if (s & (1 << i)) ret1 += v[i], tmp++;
16             else ret2 += w[i];
17         }
18         store[tmp][num[tmp]++] = ret1 - ret2;
19     }
20     for (int i = 0; i <= former; ++i) { sort(store[i], store[i] + num[i]); }
21     int ans = inf;
22     for (int s = 0; s < (1 << later); ++s) {
23         int ret1 = 0, ret2 = 0, tmp = 0;
24         for (int i = 0; i < later; ++i) {
25             if (s & (1 << i)) ret2 += w[i + former];
26             else ret1 += v[i + former], tmp++;
27         }
28         int left = former - tmp, key = ret2 - ret1;
29         int pos = lower_bound(store[left], store[left] + num[left], key) - store[left];
30         if (pos >= num[left]) ans = min(ans, abs(store[left][num[left] - 1] - key));
31         else {
32             ans = min(ans, abs(store[left][pos] - key));
33             if (pos > 0) ans = min(ans, abs(store[left][pos - 1] - key));
34             if (pos < num[left] - 1) ans = min(ans, abs(store[left][pos + 1] - key));
35         }
36     }
37 }
```

```
35     }
36 }
37     printf("%d\n", ans);
38 }
39
40 int main() {
41     scanf("%d", &T);
42     while (T--) {
43         scanf("%d", &n);
44         for (int i = 0; i < n; ++i) { scanf("%d", &v[i]); }
45         for (int i = 0; i < n; ++i) { scanf("%d", &w[i]); }
46         solve();
47     }
48     return 0;
49 }
```

Chapter 3

STL

3.1 bitset

作用：判断每位状态 0 或者 1

定义：bitset<MAX_N> b; //MAX_N 是长度下标从 0 开始， 默认状态值都为 0

3.1.1 用 unsigned long 值初始化 bitset 对象

将 unsigned long 值转化为二进制的为模式，而 bitset 对象中的位集作为这种模式的副本。如果 bitset 类型大于 unsigned long 的二进制位数，则其余高阶位位置为 0，如果 bitset 类型长度小于 unsigned long 值的二进制位数，则只使用 unsigned long 值中的低阶位，超过 bitset 类型长度的高阶位将会被舍弃。例如：

```
bitset<32> bs(0xffff); // bits 0 ... 15 are set to 1; 16 ... 31 are 0
```

3.1.2 用 string 对象初始化 bitset 对象

string 对象只能为 01 串。如果 string 对象中的字符个数小于 bitset 类型的长度，则高位为 0。从 string 对象读入位集的顺序是从右向左（反向转化）：string 对象最右边的字符（即下标最大的那个字符）用来初始化低阶位（即下标为 0 的为）。可以指定某个子串作为初始值。例如：

```
1 string str("1111111000000011001101");
2 bitset<32> bs1(str); //str is fully used to initialize bs1
3 bitset<32> bs2(str, 5, 4); // 4 bits starting at str[5], 1100
4 bitset<32> bs3(str, str.size() - 4); // use last 4 characters
5 cout << "bs1 = " << bs1 << endl;
6 cout << "bs2 = " << bs2 << endl;
7 cout << "bs3 = " << bs3 << endl;
8 /*
9 Result:
10 bs1 = 0000000000111111000000011001101
11 bs2 = 000000000000000000000000000000001100
12 bs3 = 0000000000000000000000000000000000000000001101
13 */
```

3.1.3 操作

b.any()	b 中是否存在置为 1 的二进制位
b.none()	b 中不存在置为 1 的二进制位吗？
b.count()	b 中置为 1 的二进制位的个数
b.size()	b 中二进制位的个数
b[pos]	访问 b 中在 pos 处的二进制位
b.test(pos)	b 中在 pos 处的二进制位是否为 1 ?
b.set()	把 b 中所有二进制位都置为 1
b.set(pos)	把 b 中在 pos 处的二进制位置为 1
b.reset()	把 b 中所有二进制位都置为 0

b.reset(pos)	把 b 中在 pos 处的二进制位置为 0
b.flip()	把 b 中所有二进制位逐位取反
b.flip(pos)	把 b 中在 pos 处的二进制位取反
b.to_ulong()	用 b 中同样的二进制位返回一个 unsigned long 值

3.1.4 测试

注意：使用 string 初始化时从右向左处理，如下初始化的各个位的值将是 110，而非 011 string strVal("011");

```

1  bitset<3> bs1(strVal);
2  cout << "bs1[0] is " << bs1[0] << endl;
3  cout << "bs1[1] is " << bs1[1] << endl;
4  cout << "bs1[2] is " << bs1[2] << endl;
5  cout << bs1 << endl;
6 //any() 方法如果有一位为 1，则返回1
7  cout << "bs1.any() = " << bs1.any() << endl;
8 //none() 方法，如果有一个为 1 none 则返回 0，如果全为 0 则返回 1
9  bitset<3> bsNone;
10 cout << "bsNone.none() = " << bsNone.none() << endl;
11 //count() 返回几个位为1
12 cout << "bs1.count() = " << bs1.count() << endl;
13 //size() 返回位数
14 cout << "bs1.size() = " << bs1.size() << endl;
15 //flip() 诸位取反
16 bitset<3> bsFlip = bs1.flip();
17 cout << "bsFlip = " << bsFlip << endl;
18 //to_ulong: 用 bs1 中同样的二进制位返回一个 unsigned long 值
19 unsigned long val = bs1.to_ulong();
20 cout << val << endl;
21 /*
22 Result:
23 bs1[0] is 1
24 bs1[1] is 1
25 bs1[2] is 0
26 011
27 bs1.any() = 1
28 bsNone.none() = 1
29 bs1.count() = 2
30 bs1.size() = 3
31 bsFlip = 100
32 4
33 */

```

3.2 单调栈、单调队列

3.2.1 单调栈

单调队列和单调栈的时间复杂度都是: $O(n)$ 。

单调栈主要用于解决某个元素它向左向右为最大值或最小值的最大范围是什么。如果是最大值，那就要维护单调非递增栈（唯一最大就是单调递减栈），如果是最小值就要维护单调非递减栈（唯一最小就是单调递增栈）。

```

1 //求数组每个数以其为区间唯一最小值的最大区间左右端点
2 int top = 0, cur;
3 for (int i = 1; i <= n; ++i) {
4     while (top) {
5         cur = sta[top];
6         if (data[cur] <= data[i]) break; //非唯一时去掉等号
7         --top;
8     }
9     if (top == 0) left[i] = 1; //data[i] 的区间左端点
10    else left[i] = sta[top] + 1;
11    sta[++top] = i;
12}
13 top = 0;
14 for (int i = n; i >= 1; --i) {
15     while (top) {
16         cur = sta[top];
17         if (data[cur] <= data[i]) break;
18         --top;
19     }
20     if (top == 0) right[i] = n;
21     else right[i] = sta[top] - 1;
22     sta[++top] = i;
23}
24 //如果是求以其为最大值，只需要把 <= 换为 >=

```

[POJ 3494]: 给出一个 $n * m$ 的 01 矩阵，求出最大全 1 子矩阵面积。数据范围： $n, m \leq 2000$

我们把每一行单独处理，把从这行向上连续延伸全为 1 的最大长度看成是矩形的高，那么每行其实就是求个最大矩形面积。

$height[i][j]$: 第 i 行第 j 列元素往上最长的连续 1 长度

需要用 $O(n^2)$ 的复杂度预处理出 $height[]$ ，然后需要枚举每行，每行利用单调栈可以在 $O(n)$ 复杂度得到最大矩形面积。总的时间复杂度是: $O(n^2)$ 。

```

1 const int MAX_N = 2010;
2
3 int n, m, ans;
4 int mat[MAX_N][MAX_N];
5 int height[MAX_N][MAX_N], sta[MAX_N], L[MAX_N], R[MAX_N];
6
7 //height[i][j]: 第 i 行第 j 列元素往上最长的连续 1 长度
8 //维护单调非递减栈
9 void solve(int row)
{
10
11     int top = 0, cur;
12     height[row][m + 1] = 0;
13     for (int j = 1; j <= m + 1; ++j) {
14         while (1) {
15             cur = sta[top];
16             if (height[row][cur] <= height[row][j]) break;
17             R[cur] = j;
18             --top;
19         }
20         L[j] = cur;

```

```

21     sta[++top] = j;
22 }
23 for (int j = 1; j <= m; ++j) {
24     if (mat[row][j] == 0) continue;
25     int len = R[j] - L[j] - 1;
26     ans = max(ans, height[row][j] * len);
27 // printf("height[%d][%d] = %d len = %d\n", row, j, height[row][j], len);
28 }
29 }
30
31 int main()
32 {
33     while (~scanf("%d%d", &n, &m)) {
34         for (int i = 1; i <= n; ++i) {
35             for (int j = 1; j <= m; ++j) {
36                 scanf("%d", &mat[i][j]);
37             }
38         }
39         memset(height, 0, sizeof(height));
40         for (int j = 1; j <= m; ++j) {
41             for (int i = 1; i <= n; ++i) {
42                 if (mat[i][j] == 1) {
43                     height[i][j] = 1;
44                     while (mat[++i][j] == 1) {
45                         height[i][j] = height[i - 1][j] + 1;
46                     }
47                     --i;
48                 }
49             }
50         }
51         ans = 0;
52         for (int i = 1; i <= n; ++i) { solve(i); }
53         printf("%d\n", ans);
54     }
55     return 0;
56 }
```

3.2.2 单调队列

单调队列主要用于解决满足特定条件的区间问题（如：区间最大值不超过 k 的最大区间长度，所有区间长度为 k 的最大元素值，长度不超过 k 的最大连续子序列和，区间最值差 $\in [m, k]$ 的最大区间长度）。往往和前缀和结合在一起。需要判别队列应是何种单调性。

【HDU 3530 Subsequences】 给 $n(n \leq 10^5)$ 个数，求区间最值差 $\in [m, k]$ 的最大区间长度。

维护一个单调非递增队列和一个单调非递减队列，通过当前位置元素维护其单调性，然后调整队列首元素之差 $diff$ ，使 $diff \leq k$ ，判断调整后的 $diff$ 是否满足 $diff \geq m$ ，如果满足更新 ans 。

```

1 memset(dec, 0, sizeof(dec));
2 memset(inc, 0, sizeof(inc));
3 int ans = 0, diff, pre = 0; //额外注意 pre 赋初值为0
4 for (int i = 0; i < n; ++i) {
5     while (head_inc != tail_inc && data[i] < data[inc[tail_inc - 1]]) --tail_inc;
6     inc[tail_inc++] = i;
7
8     while (head_dec != tail_dec && data[i] > data[dec[tail_dec - 1]]) --tail_dec;
9     dec[tail_dec++] = i;
10
11    while(1) {
12        diff = data[dec[head_dec]] - data[inc[head_inc]];
13        if (diff > k) {
14            if (dec[head_dec] < inc[head_inc]) {
15                pre = dec[head_dec] + 1; //注意区间首的位置
16                head_dec++;
17            }
18        }
19    }
20 }
```

```

17     } else {
18         pre = inc[head_inc] + 1;
19         head_inc++;
20     }
21 } else break;
22 }
23 diff = data[dec[head_dec]] - data[inc[head_inc]];
24 if (diff >= m) ans = max(ans, i - pre + 1);
25 }
```

3.3 list

3.3.1 CF 350 E

给出一个长度为偶数的只含'(' 和')' 并且两者个数相等的字符串, 初始指针位置是 p , 下标从 1 开始. 有三种操作:

- R 指针位置右移, 即 $p++$
- L 指针位置左移, 即 $p--$
- D 删除 p 位置和相对应括号这个区间的所有括号

输出若干次操作后的字符串.

需要预处理出每个括号和相对应的括号的下标.

```

1 const int MAX_N = 500010;
2
3 int match[MAX_N];
4
5 int main()
6 {
7     int n, m, p;
8     while (cin >> n >> m >> p) {
9         string s1, s2;
10        cin >> s1;
11        cin >> s2;
12        stack<int> s;
13        for (int i = 0; i < n; i++) {
14            if (s1[i] == '(') {
15                s.push(i);
16            } else {
17                int t = s.top();
18                s.pop();
19                match[t] = i;
20                match[i] = t;
21            }
22        }
23        list<int> lis;
24        for (int i = 0; i < n; i++) { lis.push_back(i); }
25        list<int> ::iterator pos = lis.begin();
26        for (int i = 1; i < p; i++) pos++;
27        for (int i = 0; i < m; i++) {
28            if (s2[i] == 'L') pos--;
29            else if (s2[i] == 'R') pos++;
30            else {
31                list<int> ::iterator tmp = pos, it;
32                if (s1[*pos] == '(') {
33                    while ((*pos) != match[*tmp]) {
34                        pos++;
35                    }
36                } else {
37                    while ((*tmp) != match[*pos]) {
```

```
38             tmp--;
39         }
40     }
41     for (it = tmp; it != pos; ){
42         lis.erase(it++);
43     }
44     lis.erase(pos++);
45     if (pos == lis.end()) pos--;
46 }
47 }
48 for (it = lis.begin(); it != lis.end(); it++){
49     cout << s1[*it];
50 }
51 cout << endl;
52 }
53 return 0;
54 }
```

Chapter 4

计算几何

4.1 简单

4.1.1 定义

```
1 //判断浮点数的符号x
2 int sgn(double x)
3 {
4     if(fabs(x) < eps) return 0;
5     else if(x < 0) return -1;
6     else return 1;
7 }
8
9 struct Point { 点的定义:
10     double x, y;
11
12     Point() {}
13     Point(double _x, double _y) : x(_x), y(_y) {}
14     bool operator == (const Point& rhs) const {
15         return sgn(x - rhs.x) == 0 && sgn(y - rhs.y) == 0;
16     }
17     bool operator < (const Point& rhs) const {
18         return sgn(x - rhs.x) == 0 ? sgn(y - rhs.y) < 0 : x < rhs.x;
19     }
20     Point operator - (const Point& rhs) const {
21         return Point(x - rhs.x, y - rhs.y);
22     }
23     Point operator + (const Point& rhs) const {
24         return Point(x + rhs.x, y + rhs.y);
25     }
26     Point operator * (const double d) const {
27         return Point(x * d, y * d);
28     }
29     Point operator / (const double d) const {
30         return Point(x / d, y / d);
31     }
32     double dot(const Point& rhs) const { //点积
33         return x * rhs.x + y * rhs.y;
34     }
35     double cross(const Point& rhs) const { //叉积
36         // A.cross(B) < 0 说明向量 B 在向量 A 右侧
37         return x * rhs.y - y * rhs.x;
38     }
39     double dis(const Point& rhs) const { //两点距离
40         return hypot(x - rhs.x, y - rhs.y);
41     }
42     double len() const { //长度
43         return hypot(x, y);
44     }
```

```

45 //计算 pa 和 pb 的夹角, 即从这个点看 a,b 所成的夹角返回弧度值且是正值, ,
46 double rad(const Point& a, const Point& b) const {
47     Point p = *this;
48     return fabs(atan2(fabs((a - p).cross(b - p)), (a - p).dot(b - p)));
49 }
50 Point to_vector(double r) const { //返回长度为 r 的向量
51     double l = len();
52     if (!sgn(l)) return *this;
53     r /= l;
54     return Point(x * r, y * r);
55 }
56 Point rotleft(){ //逆时针旋转 90 度
57     return Point(-y, x);
58 }
59 Point rotright(){ //顺时针旋转 90 度
60     return Point(y, -x);
61 }
62 Point rotate(const Point& rhs, const double angle){ //绕着 rhs 点逆时针旋转 angle 度
63     Point v = (*this) - rhs;
64     double c = cos(angle), s = sin(angle);
65     return Point(rhs.x + v.x * c - v.y * s, rhs.y + v.x * s + v.y * c);
66 }
67 };

```

```

1 struct Line{ 线段的定义:
2     Point s, e;
3
4     Line() {}
5     Line(const Point& _s, const Point& _e) : s(_s), e(_e) {}
6     bool operator == (const Line& rhs) const {
7         return s == rhs.s && e == rhs.e;
8     }
9     //根据一个点和倾斜角 angle 确定直线,0<= angle < pi
10    Line(const Point& rhs, const double angle) const {
11        s = p;
12        if(sgn(angle - pi / 2) == 0){ //竖直直线
13            e = (s + Point(0, 1));
14        } else {
15            e = (s + Point(1, tan(angle)));
16        }
17    }
18    double length() const{ //线段长度
19        return s.dis(e);
20    }
21    double angle(){ //返回直线的倾斜角0<= angle < pi
22        double k = atan2(e.y - s.y, e.x - s.x);
23        if(sgn(k) < 0) k += pi;
24        if(sgn(k - pi) == 0) k -= pi;
25        return k;
26    }
27    //点和直线的关系: 1 在左侧, 2 在右侧, 3 在直线上
28    int relation(const Point& rhs) const{
29        int c = sgn((rhs - s).cross(e - s));
30        if(c < 0) return 1;
31        else if(c > 0) return 2;
32        else return 3;
33    }
34    bool point_on_seg(const Point& rhs) const { //点在线段上的判断
35        return sgn((rhs - s).cross(e - s)) == 0 && sgn((rhs - s).dot(rhs - e)) <= 0;
36    }
37    bool parallel(const Line v) const { //判断直线平行
38        return sgn((e - s).cross(v.e - v.s)) == 0;
39    }
40    double point_to_line_dis(const Point& rhs) const{ //点到直线的距离
41        return fabs((rhs - s).cross(e - s)) / length();
42    }

```

```

42     }
43     double point_to_seg_dis(const Point& rhs) const { //点到线段的距离
44         if(sgn((rhs - s).dot(e - s)) < 0 || sgn((rhs - e).dot(s - e)) < 0){
45             return min(rhs.dis(s), rhs.dis(e));
46         } else return point_to_line_dis(rhs);
47     }
48 }

```

4.1.2 顺时针输出所有顶点

将平面所有点都用上，构成一个多边形，顺时针（或逆时针）输出点的顺序。

多边形有可能是凸的也有可能是凹的。先找到最左下角的点（ x 值优先），然后对其余点以最左下角为基点极角排序。除去最左侧的一系列和 $point[0]$ 共线的点，其余的点的顺序即是逆时针的点的顺序。

```

1 const int MAX_N=2010;
2 const double eps=1e-10;
3
4 int T,n;
5 int ans[MAX_N];
6
7 Point point[MAX_N];
8
9 inline bool cmp(Point a,Point b)
10{//按照最左下角极角排序
11    double res = (a - point[0]).cross(b - point[0]);
12    if(res != 0.0) return res > 0;//res > 0 说明 b 在左侧，此时 a 的极角较小
13    else return a.dis(point[0]) < b.dis(point[0]); //共线时按照距离从小到大排序
14}
15
16 inline void solve()
17{
18    //找到最左下角顶点优先，x 最小，其次 y 最小
19    int k=0;
20    for( int i=0;i<n; i++){
21        if( point [ i ].x<point [ k ].x||( point [ i ].x==point [ k ].x&&point [ i ].y<point [ k ].y)){
22            k=i;
23        }
24    }
25    swap( point [ 0 ], point [ k ] );
26    sort( point+1,point+n,cmp); //排序从下标 1 开始，因为 point [ 0 ] 就是起点
27    int end=n-2;
28    //找到最后的顶点 end 使得， point [ end ] , point [ end+1 ] 和 point [ 0 ] 不共线
29    while(end>0){
30        double res=(point [ end ]-point [ 0 ]).cross( point [ end+1 ]-point [ 0 ]);
31        if( res!=0.0) break;
32        end--;
33    }
34    //从 point [ end+1 ] 到 point [ n-1 ] 都是和 point [ 0 ] 共线的点
35    //即 point [ end+1 ]..point [ n-1 ], point [ 0 ] 都在一条直线上
36    //因为极角排序时是按照到 point [ 0 ] 距离从小到大排序，
37    //所以这些点的逆时针顶点应该是按照距离从大到小考虑
38    //point [ end+1 ] 实际上应该是第 n-1 个点， point [ n-1 ] 实际上应该时第 end+1 个点
39    for( int i=end+1;i<n; i++){
40        ans [ i ]=point [ n+end-i ].index;
41    }
42    for( int i=0;i<=end ; i++){
43        ans [ i ]=point [ i ].index;
44    }
45    for( int i=0;i<n; i++){
46        printf("%d%c",ans [ i ], i==n-1?'\\n':',');
47    }
48 }
49
50 int main()

```

```

51 {
52     scanf("%d",&T);
53     while(T--){
54         scanf("%d",&n);
55         for(int i=0;i<n;i++){
56             scanf("%lf%lf",&point[i].x,&point[i].y);
57             point[i].index=i;
58         }
59         solve();
60     }
61     return 0;
62 }
```

4.1.3 求半径 R 圆覆盖最多点数及由圆上两点和半径求圆心

方法 1:

最优的情况一定是有两个点在圆弧上。先枚举两个点，计算两点在圆弧上的单位圆（一般会有两个）。但是可以统一取一个方向的（也就是 AB 取一个然后 BA 取另外一个）。然后枚举所有点，计算在这个单位圆内的点的个数。这样做的时间复杂度是 $O(n^3)$ 。

方法 2:

对每个点以 R 为半径画圆，对 N 个圆两两求交。这一步 $O(N^2)$ 。问题转化为求被覆盖次数最多的弧。因为如果最优圆覆盖 A 点那么最优圆一定在以 A 点为圆心的圆弧上。那么圆弧倍覆盖多少次也就意味着以这条圆弧为上任意一点为圆心花园能覆盖多少点。对每一个圆，求其上的每段弧重叠次数。

假如 A 圆与 B 圆相交。 A 上 $[PI/3, PI/2]$ 的区间被 B 覆盖 (PI 为圆周率)。那么对于 A 圆，我们在 $PI/3$ 处做一个 +1 标记，在 $PI/2$ 处做一个 -1 标记。

对于 $[PI * 5/3, PI * 7/3]$ 这样横跨 0 点的区间只要在 0 点处拆成两段即可。将一个圆上的所有标记排序，从头开始扫描。初始 $total = 0$ ，碰到 +1 标记给 $total++$ ，碰到 -1 标记 $total--$ 。扫描过程中 $total$ 的最大值就是圆上被覆盖最多的弧。求所有圆的 $total$ 的最大值就是答案。极角排序需要 $2 * n * \log n$ ，总复杂度 $O(N^2 * \log N)$

```

1 const int MAX_N=310;
2 const double eps=1e-6;
3 const double R=1.0;//定义覆盖圆半径为R
4
5 int T,n;
6
7 struct Point{
8     double x,y;
9 }point[MAX_N];
10
11 inline double dis(Point a,Point b)
12 {
13     return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
14 }
15
16 inline Point GetCenter(Point a,Point b)
17 {//获取 a,b 两点在圆周上的单元圆圆心单位圆圆心有两个,
18     struct Point mid,res;
19     mid.x=(a.x+b.x)/2,mid.y=(a.y+b.y)/2; //mid 是 a,b 中点坐标
20     double angle=atan2(b.y-a.y,b.x-a.x); //angle 是直线 ab 的倾斜角
21     double tmp=dis(a,b)/2; //tmp 是线段 ab 长度的一半
22     double d=sqrt(1.0-tmp*tmp); //d 是 ab 中点到圆心的距离
23     res.x=mid.x-d*sin(angle); //res 是直线 ab 左边的那个圆心
24     res.y=mid.y+d*cos(angle);
25     //下面的 res 是直线 ab 右边的那个圆心
26     //res.x=mid.x+d*sin(angle);
27     //res.y=mid.y-d*cos(angle);
28     return res;
29 }
30
31 int main()
32 {
33     scanf("%d",&T);
```

```

34     while(T--){
35         scanf("%d",&n);
36         for( int i=0;i<n; i++){
37             scanf("%lf%lf",&point[ i ].x,&point[ i ].y);
38         }
39         int ans=1;//初始化至少能覆盖一个点!!!
40         for( int i=0;i<n; i++){
41             for( int j=0;j<n; j++){
42                 if( i==j || dis(point[ i ], point[ j ]) - 2*R > eps) continue;
43                 int cnt=0;
44                 struct Point center=GetCenter(point[ i ], point[ j ]);
45                 for( int k=0;k<n; k++){
46                     if( dis(point[ k ], center) - R <= eps) cnt++;
47                 }
48                 ans=max( ans , cnt );
49             }
50         }
51         printf("%d\n",ans );
52     }
53     return 0;
54 }
```

```

1 const int MAX_N=310;
2 const double PI=acos(-1.0);
3 const double R=1.0;//定义覆盖圆半径为R
4
5 int T,n,total;
6
7 struct Point{
8     double x,y;
9 }point[MAX_N];
10
11 struct Angle{
12     double data;
13     int is;
14     bool operator < (const Angle& rhs) const {
15         return data<rhs.data;
16     }
17 }angle[MAX_N*2];
18
19 inline double Dis(Point a,Point b)//计算线段 ab 的长度
20 {
21     return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
22 }
23
24 inline double God(Point a,Point b)//计算向量 ab 的极角
25 {
26     double res=atan(fabs((b.y-a.y)/(b.x-a.x)));
27     if(b.y<a.y){
28         if(b.x<a.x) res+=PI;
29         else res=2*PI-res;
30     }else {
31         if(b.x<a.x) res=PI-res;
32     }
33     return res;
34 }
35
36 void solve()
37 {
38     int res=1;
39     for( int i=0;i<n; i++){
40         total=0;
41         for( int j=0;j<n; j++){
42             if( i==j ) continue;
43             double dist=Dis(point[ i ], point[ j ]);
```

```

44     if ( dist > 2*R) continue;
45     double base=God( point [ i ] , point [ j ] );
46     double extra=acos( dist / 2.0); //计算差角
47     angle [ total ]. data=base+extra;
48     angle [ total++]. is=-1;
49     angle [ total ]. data=base-extr a;
50     angle [ total++]. is=1;
51   }
52   if ( total <=res) continue;
53   sort ( angle , angle+total );
54   int tmp=1;
55   for ( int j =0;j<total ;j++){
56     tmp+=angle [ j ]. is ;
57     res=max( res ,tmp );
58   }
59 }
60 printf ( "%d\n" ,res );
61 }
62
63 int main()
64 {
65   scanf ( "%d" ,&T);
66   while ( T--){
67     scanf ( "%d" ,&n );
68     for ( int i =0;i<n ;i++){
69       scanf ( "%lf%lf" ,&point [ i ]. x,&point [ i ]. y );
70     }
71     solve ();
72   }
73   return 0;
74 }
```

4.1.4 计算两圆公共部分面积

处理两圆相交情况

分析交点及两圆圆心组成的四边形和交点与两圆圆心组成的两个扇形

假设 $c1$ 圆扇形角为 $2 * \alpha$ ，那么四边形的面积 s 等于: $0.5 * dis * c1.r * sin(\alpha) * 2 = dis * c1.r * sin(\alpha)$;

记 $c1, c2$ 扇形面积分别为 $s1, s2$. 那么 $s1 = (2 * \alpha / (2 * PI)) * PI * c1.r * c1.r = \alpha * c1.r * c1.r$;

而且 α 可以用余弦定理计算出来: $\cos(\alpha) = (d^2 + c1.r^2 - c2.r^2) / (2 * d * c1.r)$;

同理也可以计算出 $s2$ ，所以相交部分面积为: $s1 + s2 - s$.

```

1 const double eps=le-8;
2 const double PI=acos (-1.0);
3
4 struct Circle{
5   double x,y,r;
6 };
7
8 double GetDis( Circle c1 ,Circle c2)//计算圆心距
9 {
10   return sqrt ((c1.x-c2.x)*(c1.x-c2.x)+(c1.y-c2.y)*(c1.y-c2.y));
11 }
12
13 double CalcArea( Circle c1 ,Circle c2)
14 {
15   double dis=GetDis(c1,c2 );
16   if (dis-(c1.r+c2.r)>=eps){ //两圆相离
17     return 0;
18   }else if (c1.r>=c2.r+dis){ //c1 包含c2
19     return PI*c2.r*c2.r;
20   }else if (c2.r>=c1.r+dis){ //c2 包含c1
21     return PI*c1.r*c1.r;
22   }
23   double angle1=acos(( dis*dis+c1.r*c1.r-c2.r*c2.r)/(2*dis*c1.r));
```

```

24     double s1=angle1*c1.r*c1.r;      //c1 圆扇形面积
25
26     double s=dis*c1.r*sin(angle1);  //四边形面积
27
28     double angle2=acos((dis*dis+c2.r*c2.r-c1.r*c1.r)/(2*dis*c2.r)); //c2 圆扇形面积
29     double s2=angle2*c2.r*c2.r;
30
31     return s1+s2-s;
32 }
33
34 int main()
35 {
36     Circle a,b;
37     while(cin>>a.x>>a.y>>a.r>>b.x>>b.y>>b.r){
38         double ans=CalcArea(a,b);
39         printf("%.3lf\n",ans);
40     }
41     return 0;
42 }
```

4.1.5 判断两线段是否相交

已知 n 条木棍的起点和终点坐标，问第 i 条木棍和第 j 条木棍是否相连？

当两条木棍之间有公共点时，就认为他们时相连的。通过相连的木棍间接的连在一起的两根木棍也认为时相连的。

木棍就是二维平面上的线段，只要能判断线段是否相交，那么建图后可以通过 Floyd 算法或者并查集进行连接性判断。如何判断两条线段是否相交呢？首先会想到计算两条直线的交点，然后判断交点是否在线段上。那么两条直线的交点如何求得呢？虽然可以把直线表示成方程，通过建立方程组求解。但在几何问题中，运用向量的内积和外积进行计算是非常方便的。对于二维向量 $p_1 = (x_1, y_1)$ 和 $p_2 = (x_2, y_2)$ ，我们定义内积 $p_1 \cdot p_2 = x_1 * x_2 + y_1 * y_2$ ，外积 $p_1 * p_2 = x_1 * y_2 - x_2 * y_1$ 。要判断点 q 是否在线段 p_1-p_2 上，只要先利用外积根据是否有 $(p_1-q) \cdot (p_2-q) == 0$ 来判断点 q 是否在直线 p_1-p_2 上，再利用内积根据是否有 $(p_1-q) \cdot (p_2-q) <= 0$ 来判断点 q 是否落在 p_1-p_2 之间。而要求两直线的交点，通过变量 t 将直线 p_1-p_2 上的点表示为 $p_1 + t(p_2 - p_1)$ ，交点又在直线 q_1-q_2 上，所以有： $(q_2 - q_1) \cdot (p_1 + t(p_2 - p_1) - q_1) == 0$ 于是可以利用下式求得 t 的值：

$$p_1 + (p_2 - p_1) * ((q_2 - q_1) \cdot (q_1 - p_1) / (q_2 - q_1) \cdot (p_2 - p_1))$$

但是使用这个方法还要注意边界情况。也就是平行的线段也可能有公共点。这时我们可以选择通过检查端点是否在另一条线段上来判断。

```

1 double EPS=1e-10;
2 const int MAX_N=20;
3
4 int n;
5 bool connected[MAX_N][MAX_N];
6 int pre[MAX_N];
7
8 //考虑误差的加法运算
9 double add(double a,double b)
10 {
11     if(abs(a+b)<EPS*(abs(a)+abs(b))) return 0;
12     return a+b;
13 }
14
15 struct Point st[MAX_N],ed[MAX_N]; //st[i],ed[i] 分别是第 i 条线段的起点和终点
16
17 //判断点 q 是否在线段 p1-p2 上
18 bool on_seg(Point p1,Point p2,Point q)
19 {
20     return (p1-q).cross(p2-q)==0&&(p1-q).dot(p2-q)<=0;
21 }
22
23 //计算直线 p1-p2 与直线 q1-q2 的交点
```

```

24 Point intersection(Point p1,Point p2,Point q1,Point q2)
25 {
26     return p1+(p2-p1)*((q2-q1).cross(q1-p1)/(q2-q1).cross(p2-p1));
27 }
28
29 //Floyd_Warshall 算法判断任意两条木棍是否相连
30 void Floyd_Warshall()
31 {
32     for( int k=0;k<n;k++){
33         for( int i=0;i<n; i++){
34             for( int j=0;j<n; j++){
35                 connected[ i ][ j ] |= connected[ i ][ k ]&&connected[ k ][ j ];
36             }
37         }
38     }
39 }
40
41 int find( int x )
42 {
43     return pre[ x]==x?x:pre[ x ]=find( pre[ x ] );
44 }
45
46 //并查集判断任意两条木棍是否相连
47 void UnionFindSet()
48 {
49     for( int i=0;i<n; i++) pre[ i ]=i ;
50     for( int i=0;i<n; i++){
51         for( int j=i+1;j<n; j++){
52             if( connected[ i ][ j ]&&find( i )!=find( j )){
53                 pre[ j ]=i ;
54             }
55         }
56     }
57 }
58
59 void solve()
60 {
61     for( int i=0;i<n; i++){
62         connected[ i ][ i ]=true ;
63         for( int j=0;j<i ; j++){//判断木棍 i 和 j 是否有公共点
64             if( (st[ i ]-ed[ i ]).cross(st[ j ]-ed[ j ])==0){//木棍平行时
65                 connected[ i ][ j ]=connected[ j ][ i ]=on_seg(st[ i ],ed[ i ],st[ j ])
66                                         || on_seg(st[ i ],ed[ i ],ed[ j ])
67                                         || on_seg(st[ j ],ed[ j ],st[ i ])
68                                         || on_seg(st[ j ],ed[ j ],ed[ i ]);
69             } else { // 不平行时
70                 Point inter=intersection(st[ i ],ed[ i ],st[ j ],ed[ j ]);
71                 connected[ i ][ j ]=connected[ j ][ i ] =
72                     on_seg(st[ i ],ed[ i ],inter) && on_seg(st[ j ],ed[ j ],inter);
73             }
74         }
75     }
76     Floyd_Warshall();
77     UnionFindSet();
78 }
79
80 int main()
81 {
82     while(~scanf(" %d",&n)&&n){
83         for( int i=0;i<n; i++){
84             scanf(" %lf%lf%lf%lf ",&st[ i ].x,&st[ i ].y,&ed[ i ].x,&ed[ i ].y);
85         }
86         solve();
87         int a,b;
88         while(~scanf(" %d%d ",&a,&b)&&(a||b)){

```

```

89         // if( connected [a-1][b-1]) printf("CONNECTED\n");
90         if( find(a-1)==find(b-1)) printf("CONNECTED\n");
91         else printf("NOT CONNECTED\n");
92     }
93 }
94 return 0;
95 }
```

4.1.6 判断点和多边形的关系

判断多边形和点的关系: 3 点在顶点上,2 在边上,1 在内部,0 在外部

```

1 Point vertex[MAX_N]; //用来存储多边形的顶点
2 Lint line[MAX_N]; //用来存储多边形的边
3 int relation_with_point(Point q, int n)
{
4     for( int i = 0; i < n; i++){
5         if(q == vertex[i]) return 3;
6     }
7     vertex[n] = vertex[0];
8     for( int i = 0; i < n; ++i){
9         line[i] = Line(vertex[i], vertex[i + 1]);
10    }
11    for( int i = 0; i < n; i++){
12        if(line[i].point_on_seg(q)) return 2;
13    }
14    int cnt = 0;
15    for( int i = 0; i < n; i++){
16        int j = i + 1;
17        int k = sgn((q - vertex[j]).cross(vertex[i] - vertex[j]));
18        int u = sgn(vertex[i].y - q.y);
19        int v = sgn(vertex[j].y - q.y);
20        if(k > 0 && u < 0 && v >= 0) cnt++;
21        if(k < 0 && v < 0 && u >= 0) cnt--;
22    }
23    return cnt != 0;
24}
25
26
27 //点 p 在多边形顶点、边上、内部返回 true , 否则返回false
28 bool in_polygon(Point p)
{
29     int i, j, c = 0;
30     double testx = p.x, testy = p.y;
31     for (i = 0, j = n - 1; i < n; j = i++) {
32         if ( ((vertex[i].y > testy) != (vertex[j].y > testy)) &&
33             (testx < (vertex[j].x - vertex[i].x) *
34             (testy - vertex[i].y) / (vertex[j].y - vertex[i].y) + vertex[i].x) )
35             c = !c;
36     }
37     return c;
38 }
39 }
```

4.1.7 计算多边形的最小宽度

多边形的最小宽度是指以旋转多边形以某个方式垂直下落, 通过长度为 L 的狭缝, 狹缝长度的最小值即是多边形的最小宽度。例如长宽为 4, 5 的矩形的最小宽度为 4。

先将多边形求凸包, 枚举这个凸包的每一条边, 求出所有顶点距离这条边的距离的最大值, 这就是这条边的多对应的高度, 在所有边的高度中取最小值就是最小宽度 L。

```

1 inline bool cmp_x(const Point a, const Point b)
2 {
3     if(a.x == b.x) return a.y < b.y;
4     return a.x < b.x;
```

```

5 }
6
7 //求凸包
8 inline int Andrew()
9 {
10    sort(point, point + m, cmp_x);
11    int k = 0;
12    for(int i = 0; i < m; i++){
13        while(k > 1 && (vertex[k - 1] - vertex[k - 2]).cross
14        (point[i] - vertex[k - 1]) <= 0){
15            k--;
16        }
17        vertex[k++] = point[i];
18    }
19    int m = k;
20    for(int i = n - 2; i >= 0; i--){
21        while(k > m && (vertex[k - 1] - vertex[k - 2]).cross
22        (point[i] - vertex[k - 1]) <= 0){
23            k--;
24        }
25        vertex[k++] = point[i];
26    }
27    if(k > 1) k--;
28    return k;
29 }
30
31 //计算 c 到 a—b 的距离
32 inline double GetHight(Point a, Point b, Point c)
33 {
34     double d1 = a.dis(b);
35     double d2 = (b - a).cross(c - a);
36     return fabs(d2 / d1);
37 }
38
39 //计算凸包边相对顶点的最远距离在所有距离中取最小值,
40 inline double GetMInWidth(int k)
41 { //k 是凸包顶点数目
42     vertex[k] = vertex[0];
43     double res = 1e6;
44     for(int i = 0; i < k; i++){
45         double tmp = -1.0;
46         for(int j = 0; j < k; j++){
47             tmp = max(tmp, GetHight(vertex[i], vertex[i + 1], vertex[j]));
48         }
49         res = min(res, tmp);
50     }
51     return res;
52 }

```

4.1.8 计算多边形的最长内直径

多边形的最长内直径是指多边形顶点间的最大距离，并且这个最大距离的顶点连线不超出多边形内部。如果多边形是一个凸包的话，那很好办，直接暴力扫一遍顶点间距离然后取最大即可。但是多边形可能是凹多边形啊。首先最长内直径的一个端点一定是多边形的顶点，另一端点可能在某条边上（这时一定存在多边形顶点在这条内直径上）或者另一端点也是多边形的顶点，所以可以枚举多边形的顶点 a 和 b ，计算直线 ab 和多边形的所有交点（步骤 1），得到的所有交点都是在一条直线上的，但是将所有交点按照距离最左下角交点距离远近排序，相邻两个交点构成了一条小线段，并不是每条小线段都是在多边形的内部的，这时只需要判断线段的中点是否在多边形的内部或者是多边形边上的点即可，将所有在多边形内部的小线段打上标记，那么，连续最长的在多边形内部的小线段就构成了枚举顶点 ab 时得到的最大内直径。枚举所有顶点对，取最大即可。

步骤 1 的实现：考虑直线 ab 和多边形所有边的交点。如果边和直线 ab 平行，那么这条边的两个端点都要取，如果边不和直线 ab 平行，计算边所在直线和 ab 的交点，判断交点是否在边上即可。这样取完交点后需要去重。多边形的顶点需预先逆时针/顺时针排序。

```

1 //vertex1 是多边形的顶点, point1 是枚举顶点直线和多边形边的所有交点
2 int line_flag[MAX_N];
3
4 struct Line{
5     Point st, ed;
6
7     Line () {}
8     Line (Point _st, Point _ed) : st(_st), ed(_ed) {}
9 }line[MAX_N], line1[MAX_N];
10
11 //判断点 q 是否在线段 p1—p2 上
12 inline bool on_seg(Point p1, Point p2, Point q)
13 {
14     return (p1 - q).cross(p2 - q) == 0 && (p1 - q).dot(p2 - q) <= 0;
15 }
16
17 //判断 p 是否在直线 a 上
18 bool on_straight_line(Line a, Point p)
19 {
20     Point st = a.st, ed = a.ed;
21     if(p == st || p == ed) return true;
22     if((p.x - st.x) * (ed.y - st.y) == (p.y - st.y) * (ed.x - st.x)) return true;
23     else return false;
24 }
25
26 //获得直线 a 和直线 b 的交点
27 inline Point GetInter(Line a, Line b)
28 {
29     Point p1, p2, q1, q2;
30     p1 = a.st, p2 = a.ed;
31     q1 = b.st, q2 = b.ed;
32     return p1 + (p2 - p1) * ((q2 - q1).cross(q1 - p1) / (q2 - q1).cross(p2 - p1));
33 }
34
35 //判断直线 a 和 b 是否平行
36 inline bool Parallel(Line a, Line b)
37 {
38     Point p1, p2, q1, q2;
39     p1 = a.st, p2 = a.ed;
40     q1 = b.st, q2 = b.ed;
41     if((p2.y - p1.y) * (q2.x - q1.x) == (p2.x - p1.x) * (q2.y - q1.y)) return true;
42     else return false;
43 }
44
45 //获得直线 a 和多边形的所有交点
46 inline int GetPoint(Line a)
47 {
48     int total = 0;
49     for(int i = 0; i < n; i++){
50         if(Parallel(a, line[i])){
51             if(on_straight_line(a, line[i].st)){
52                 point1[total++] = line[i].st;
53                 point1[total++] = line[i].ed;
54             }
55         }else{
56             Point inter = GetInter(a, line[i]);
57             if(on_seg(line[i].st, line[i].ed, inter)){
58                 point1[total++] = inter;
59             }
60         }
61     }
62     return total;
63 }
64
65 Point zuoxiaojiao;

```

```

66 inline Point Getzuoxiajiao(int total)
67 {
68     Point ans = point1[0];
69     for(int i = 1; i < total; i++){
70         if(point1[i].x < ans.x ||
71             (point1[i].x == ans.x && point1[i].y < ans.y)){
72             ans = point1[i];
73         }
74     }
75     return ans;
76 }
77
78 //将所有交点按照距离最左下角点的距离从小到大排序
79 inline bool cmp_zuoxiajiao(const Point a, const Point b)
80 {
81     return a.dis(zuoxiajiao) < b.dis(zuoxiajiao);
82 }
83
84 inline bool in_polygon(Point p)
85 { //判断点 p 是否在多边形内部
86     int i, j, c = 0;
87     double testx = p.x, testy = p.y;
88     for (i = 0, j = n - 1; i < n; j = i++) {
89         if ( ((vertex1[i].y > testy) != (vertex1[j].y > testy)) &&
90             (testx < (vertex1[j].x - vertex1[i].x) *
91             (testy - vertex1[i].y) / (vertex1[j].y - vertex1[i].y) + vertex1[i].x) )
92             c = !c;
93     }
94     return c;
95 }
96
97 inline double GetDiameter() //获得多边形的最长内直径
98 {
99     double res = -1.0;
100    for(int i = 0; i < n; i++){
101        for(int j = i + 1; j < n; j++){
102            int total = GetPoint(Line(vertex1[i], vertex1[j]));
103            zuoxiajiao = Getzuoxiajiao(total);
104            sort(point1, point1 + total, cmp_zuoxiajiao);
105            int line_cnt = unique(point1, point1 + total) - point1 - 1; //去重
106            for(int k = 0; k < line_cnt; k++){
107                line1[k] = Line(point1[k], point1[k + 1]);
108            }
109            memset(line_flag, 0, sizeof(line_flag));
110            for(int k = 0; k < line_cnt; k++){
111                Point mid;
112                mid.x = (line1[k].st.x + line1[k].ed.x) / 2;
113                mid.y = (line1[k].st.y + line1[k].ed.y) / 2;
114                if(in_polygon(mid)){
115                    line_flag[k] = 1;
116                }
117            }
118            double tmp = -1.0;
119            int have = 0;
120            Point prest, nowed;
121            for(int k = 0; k < line_cnt; k++){
122                if(line_flag[k]){
123                    if(have == 0){
124                        have = 1;
125                        prest = line1[k].st;
126                    }
127                    nowed = line1[k].ed;
128                } else {
129                    if(have == 1){
130                        tmp = max(tmp, nowed.dis(prest));
131                    }
132                }
133            }
134        }
135    }
136 }
```

```
131                     have = 0;  
132                 }  
133             }  
134         } if (have == 1){  
135             tmp = max(tmp, nowed.dis(prest));  
136         }  
137         res = max(res, tmp);  
138     }  
139 }  
140 return res;  
141 }  
142 }
```

4.2 凸包

4.2.1 求凸包顶点:Graham 扫描法

向量的叉积: $A * B = |A| * |B| * \sin\alpha$, α 是向量 A 和向量 B 之间的夹角

向量的点积 $A \cdot B = |A| * |B| * \cos\alpha$

向量 A 和 B 的叉积小于 0, 说明向量 B 在向量 A 右侧

```

1 const int MAX_N=1010;
2 const double INF=1e90;
3 const double eps=1e-10;
4 const double PI=acos(-1.0);
5
6 int T,n,cases=0;
7
8 Point point[MAX_N],vertex[MAX_N];
9
10 inline bool cmp_x(const Point a,const Point b)
11 { //优先升序, 其次升序排序xy
12     if(a.x==b.x) return a.y<b.y;
13     return a.x<b.x;
14 }
15
16 //Graham 扫描法
17 inline int Andrew()
18 {
19     sort(point,point+n,cmp_x);
20     int k=0;
21     //构造下凸包
22     for(int i=0;i<n;i++){
23         while(k>1&&(vertex[k-1]-vertex[k-2]).cross(point[i]-vertex[k-1])<=0){
24             k--;
25         }
26         //因为存凸包的顶点是从 0 开始的, 而最左下的点一定是凸包顶点, 所以k>1
27         //因为如果 k=1 仍然继续执行的话, k-2 就小于 0 了
28         //当然如果凸包顶点是从 1 开始存储的话, 那么这里就应该是k>2
29         vertex[k++]=point[i];
30     }
31     //构造上凸包
32     int m=k; //m 是下凸包顶点数目, 且最后一个顶点是point[n-1]
33     for(int i=n-2;i>=0;i--) { //注意是从 point[n-2] 开始, 避免重复point[n-1]
34         while(k>m&&(vertex[k-1]-vertex[k-2]).cross(point[i]-vertex[k-1])<=0){
35             k--;
36         }
37         vertex[k++]=point[i];
38     }
39     if(k>1) k--; //point[0] 重复
40     return k;
41 }
```

4.2.2 判断稳定凸包

给出一个凸包, 判断凸包是否唯一确定 (稳定凸包), 即判断凸包每条边上是否存在至少三个点

```

1 inline bool JudgeStableConvexHull()
2 {
3     if(n<6) {
4         return false;
5     }
6     int total=Andrew();
7     vertex[total]=vertex[0];
8     /*将边上的点也存进凸包顶点里, 在判断的时候只需要判断凸包顶点中是否一定存在相邻向量叉积为
9      0 即可上面
10     Andrew() 函数中需要将
```

```

11     (vertex [k-1]-vertex [k-2]).cross (point [ i]-vertex [k-1])<=0改为
12     (vertex [k-1]-vertex [k-2]).cross (point [ i]-vertex [k-1])<0
13
14     for ( int  i=1;i<total ;i++){
15         if (( vertex [ i-1]-vertex [ i]).cross ( vertex [i+1]-vertex [ i]))!=0
16         &&(vertex [ i]-vertex [ i+1]).cross ( vertex [ i+2]-vertex [ i+1])!=0){
17             return  false ;
18         }
19     }
20     return  true ;
21 */
22
23     for ( int  i=0;i<total ;i++){ //遍历凸包边
24         Point  a=vertex [ i],b=vertex [ i+1];//该边端点是 a 和 b
25         int  cnt=0;
26         for ( int  j=0;j<n;j++){ //检查该边上点的个数
27             if (( point [ j]-a).cross (b-point [ j]))==0){// 在这条边上的点
28                 cnt++;
29                 if (cnt>=3)  break ;
30             }
31         }
32         if (cnt<3){ //少于三个点
33             return  false ;
34         }
35     }
36     return  true ;
37 }
```

4.2.3 凸包直径

旋转卡壳求凸包直径 (最远点距离)

```

1 inline  double  GetMostFarDistance()
2 {
3     int  total=Andrew ();
4     if (total==2){ //处理凸包退化的情况只有两个顶点 ,
5         return  vertex [0].dis (vertex [1]);
6     }
7
8     int  i=0,j=0;//在某个方向上的对踵点
9     //求出 x 轴方向上的对踵点对
10    for ( int  k=0;k<total ;k++){
11        if (!cmp_x (vertex [ i],vertex [k]))  i=k;
12        if (cmp_x (vertex [ j],vertex [k]))  j=k;
13    }
14    double  ans=0;
15    int  si=i ,sj=j ;
16    while (i!=sj || j!=si){ //将方向逐步旋转 180 度
17        ans=max (ans,vertex [ i].dis (vertex [ j]));
18        //判断先转到边 i-(i+1) 的法线方向还是边 j-(j+1) 的法线方向
19        int  nexti=(i+1)%total ,nextj=(j+1)%total ;
20        if (( vertex [nexti]-vertex [ i]).cross ( vertex [nextj]-vertex [ j])<0){
21            i=nexti; //先转到边 i-(i+1) 的法线方向
22        }else {
23            j=nextj; //先转到边 j-(j+1) 的法线方向
24        }
25    }
26    return  ans;
27 }
```

4.2.4 凸包周长及面积

计算凸包面积

将凸包看成一个个以凸包最左下顶点为顶点的凸包边为对边的三角形。那么依次扫个条边，计算三角形面
积累加即可。已知三角形三条边计算三角形面积，可用海伦 – 秦九韶公式

```

1 inline double GetConvexHullArea()
2 {
3     int total=Andrew();
4     if(total<=2){
5         return 0.0;
6     }
7     double area=0;
8     for( int i=2;i<total ; i++){
9         double a=vertex[ i-1 ]. dis( vertex[ 0 ] );
10        double b=vertex[ i ]. dis( vertex[ 0 ] );
11        double c=vertex[ i ]. dis( vertex[ i-1 ] );
12        double p=(a+b+c)/2;
13        area+=sqrt(p*(p-a)*(p-b)*(p-c));
14    }
15    return area;
16 }
```

计算凸包周长

```

1 inline double GetConvexHullPerimeter()
2 {
3     int total = Andrew();
4     vertex[ total ] = vertex[ 0 ];
5     double ans = 0;
6     for( int i = 0; i < total; i++){
7         ans+=vertex[ i ]. dis( vertex[ i+1 ] );
8     }
9     return ans;
10 }
11 //计算顶点为a,b,c的三角形面积c
12 inline double GetArea( Point a , Point b , Point c )
13 {
14     double len1=a. dis( b );
15     double len2=a. dis( c );
16     double len3=b. dis( c );
17     double p=(len1+len2+len3)/2;
18     return sqrt(p*(p-len1)*(p-len2)*(p-len3));
19 }
20 }
```

4.2.5 凸包最大三角形面积

旋转卡壳求凸包最大三角形面积，时间复杂度 $O(total^2)$, $total$ 是凸包顶点数

```

1 inline double GetBiggestTriangleAreaInConvexHull()
2 {
3     int total=Andrew();
4     vertex[ total ]=vertex[ 0 ];
5     double ans=0.0;
6     for( int i=0;i<total ; i++){
7         int j=(i+1)%total;
8         int k=(j+1)%total;
9         while( j!=i&&k!=i ){
10             ans=max(ans,fabs((vertex[ j ]-vertex[ i ]). cross( vertex[ k ]-vertex[ i ] )/2.0));
11             //已知三角形三顶点，利用叉积计算面积或者海伦秦九韶公式-
12             //ans=max(ans,GetArea( vertex[ i ],vertex[ j ],vertex[ k ]));
13             while( k!=i&&(vertex[ j ]-vertex[ i ]). cross((vertex[ k+1 ]-vertex[ k ]))<0){
14                 k=(k+1)%total;
15             }
16 }
```

```

16         j=(j+1)%total;
17     }
18 }
19 return ans;
20 }
```

4.2.6 凸包最大四边形面积

UESTC 371

给 $n \leq 1000$ 个点，从中选出不超过 4 个点，使得组成的多边形面积最大，输出最大面积。

求完凸包后，枚举对角线顶点，对另外两个顶点旋转卡壳。时间复杂度： $O(total^2)$

```

1 void solve()
2 { // P[]: 初始点, Q[]: 凸包顶点
3     int total = Andrew();
4     if (total <= 2) {
5         printf("0\n");
6         return ;
7     } else if (total == 3) {
8         printf("%.01f\n", fabs((Q[1] - Q[0]).cross(Q[2] - Q[0])));
9         return ;
10    }
11    double ans;
12    int first = 1;
13    for (int i = 0; i < total; ++i) {
14        int st = (i + 1) % total, ed = (i + 3) % total;
15        for (int j = i + 2; j < total - 1; ++j) {
16            Line line = Line(Q[i], Q[j]);
17            while (st != j && sgn(line.point_to_line_dis(Q[st]) -
18                line.point_to_line_dis(Q[(st + 1) % total])) <= 0) st = (st + 1) % total;
19            while (ed != i && sgn(line.point_to_line_dis(Q[ed]) -
20                line.point_to_line_dis(Q[(ed + 1) % total])) <= 0) ed = (ed + 1) % total;
21            double tmp1 = fabs((Q[i] - Q[st]).cross(Q[j] - Q[st]));
22            double tmp2 = fabs((Q[i] - Q[ed]).cross(Q[j] - Q[ed]));
23            if (first) ans = tmp1 + tmp2, first = 0;
24            else ans = max(ans, tmp1 + tmp2);
25        }
26    }
27    printf("%.01f\n", ans);
28 }
```

4.2.7 最小覆盖矩形面积

时间复杂度： $O(total)$, $total$ 是顶点数

```

1 inline double MinimalRectangleCover()
2 {
3     int total=Andrew();
4     if (total<=2){
5         return 0.0;
6     }
7     double ans=INF;
8     vertex [total]=vertex [0];
9     int r=1,p=1,q;
10    for (int i=0;i<total;i++){
11        double edge=vertex [ i ].dis( vertex [ i+1 ]);
12        double tmp1,tmp2;
13        while(1){ //卡出离边 vertex [ i ]--vertex [ i+1 ] 最远的点
14            tmp1=(vertex [ i+1 ]-vertex [ i ]).cross( vertex [ r+1 ]-vertex [ i ]); //叉积
15            tmp2=(vertex [ i+1 ]-vertex [ i ]).cross( vertex [ r ]-vertex [ i ]);
```

```

16     if (tmp2>tmp1) break;
17     r=(r+1)%total;
18   }
19   double height = tmp2 / edge;
20
21   while(1){ //卡出在 vertex[i]--vertex[i+1] 方向上正向最远的点
22     tmp1=(vertex[i+1]-vertex[i]).dot(vertex[p+1]-vertex[i]); //点积
23     tmp2=(vertex[i+1]-vertex[i]).dot(vertex[p]-vertex[i]);
24     if (tmp2>tmp1) break;
25     p=(p+1)%total;
26   }
27   double len1 = tmp2 / edge;
28   //len1 是从 vertex[i] 出发沿 vertex[i]--vertex[i+1] 正方向能达到的最大距离
29
30   if (i==0) q = p;
31   while(1){ //卡出在 vertex[i]--vertex[i+1] 方向上负方向最远的点
32     tmp1=(vertex[i+1]-vertex[i]).dot(vertex[q+1]-vertex[i]); //点积
33     tmp2=(vertex[i+1]-vertex[i]).dot(vertex[q]-vertex[i]);
34     if (tmp2<tmp1) break; //和上面的不一样
35     q=(q+1)%total;
36   }
37   double len2=tmp2/edge;
38   //len2 是从 vertex[i] 出发沿 vertex[i]--vertex[i+1] 负方向能达到的最大距离
39
40   double len=len1-len2;
41   ans=min(ans, len*height);
42 }
43
44 /* O(total^2) 算法
45 for (int i=0;i<total; i++){ 遍历每一条边//
46   Point A,B;
47   A=vertex[i],B=vertex[(i+1)%total];
48   double AB=A.dis(B);
49   double leftA ,rightA ,leftB ,rightB ,height ;
50   leftA=rightA=leftB=rightB=height=0.0;
51   for (int j=0;j<total; j++){ // 遍历凸包每一个顶点
52     Point C=vertex[j];
53     double d=(B-A).cross(C-A)/AB;
54     height=max(height ,d); // 更新高
55
56     double tmp=(B-A).dot(C-A)/AB; // 更新 A 点左右最大距离
57     if (tmp>0) rightA=max(rightA ,tmp);
58     else leftA=max(leftA ,-tmp);
59
60     tmp=(A-B).dot(C-B)/AB; // 更新 B 点左右最大距离
61     if (tmp<0) rightB=max(rightB ,-tmp);
62     else leftB=max(leftB ,tmp);
63     // printf(" i=%d j=%d rightA=%f rightB=%f\n",i,j,rightA,rightB );
64   }
65   double len1=max(leftA ,leftB -AB); // 左端最大超出
66   double len2=max(rightB ,rightA -AB); // 右端最大超出
67   double len=len1+len2+AB; //len 即是长
68   //printf(" len1=%f len2=%f len=%f height=%f\n",len1 ,len2 ,len ,height );
69   ans=min(ans, len*height); // 更新最小外接矩形
70 }
71 */
72 return ans;
73 }
```

4.2.8 凸包最小外接平行四边形面积

一定有两条边是凸包的边

¹ double height [MAX_N]; //用于存储凸包每条边上最远顶点到这条边的距离

```

2 double edge[MAX_N]; //存储边长度
3
4 inline double MinimalParallelogramCover()
5 {
6     int total=Andrew();
7     if (total<=2){
8         return 0.0;
9     }
10    vertex[total]=vertex[0];
11    //求每条边最远顶点距离, 即是这条边上对应的高
12    for (int i=0;i<total;i++){
13        height[i]=-1.0;
14        edge[i]=vertex[i].dis(vertex[i+1]);
15        for (int j=0;j<total;j++){
16            double tmp=((vertex[i+1]-vertex[i]).cross(vertex[j]-vertex[i]));
17            //edge; 顶点 vertex[j] 到边 i--(i+1) 的距离
18            height[i]=max(height[i],fabs(tmp/edge[i]));
19        }
20    }
21    double ans=INF;
22    //枚举两条边计算以这两条边为平行四边形边的平行四边形
23    for (int i=0;i<total;i++){
24        Point tmp1=vertex[i+1]-vertex[i];
25        for (int j=0;j<total;j++){
26            Point tmp2=vertex[j+1]-vertex[j];
27            double angle=(tmp1.cross(tmp2))/edge[i]/edge[j]; //计算向量夹角的正弦值
28            if (fabs(angle)<eps) continue;
29            ans=min(ans,height[i]*height[j]/fabs(angle));
30            //已知平行四边形两组对边上的高及一个内角的正弦值求平行四边形的面积
31        }
32    }
33    return ans;
34 }
35
36 int main()
37 {
38     scanf("%d",&T);
39     while(T--){
40         scanf("%d",&n);
41         for (int i=0;i<n;i++){
42             scanf("%lf%lf",&point[i].x,&point[i].y);
43         }
44         .....
45     }
46     return 0;
47 }
```

4.2.9 两相离凸包最短距离

工具:

- 点到线段最短距离
- 线段上点最短距离
- 将所有点顺时针排序

```

1 const int MAX_N=10010;
2 const double INF=1e90;
3 const double eps=1e-10;
4
5 int n,m;
6
7 Point P[MAX_N],Q[MAX_N],center;
```

```

8
9 inline bool cmp(Point a, Point b)
10 { // a 点在 b 点顺时针方向返回 true, 否则返回 false
11     double res=(a-center).cross(b-center);
12     if(res<eps) return true;
13     if(res>eps) return false;
14     //向量共线, 用距离判断
15     double d1=a.dis(center);
16     double d2=b.dis(center);
17     return d1>d2;
18 }
19
20 inline void Clockwise(Point point[], int num)
21 { //将所有点按照顺时针排序
22     //计算所有点重心
23     double x=0,y=0;
24     for(int i=0;i<num;i++){
25         x+=point[i].x;
26         y+=point[i].y;
27     }
28     center.x=x/num;
29     center.y=y/num;
30     sort(point, point+num, cmp);
31 }
32
33 inline double Get_Dis_Point(Point A, Point B, Point C)
34 { //计算 C 点到线段 AB 的最短距离
35     //AC.dot(AB)=|AC|*|AB|*cos(alpha), alpha 是角 BAC 的大小, 点积值小于 0
36     //说明 alpha>(PI/2), A 离 C 更近
37     if((C-A).dot(B-A)<-eps) return A.dis(C);
38     if((C-B).dot(A-B)<-eps) return B.dis(C);
39     //考虑 C 到 AB 的垂足落在线段 AB 上的情况
40     //AB.cross(AC)=|AB|*|AC|*sin(alpha其中), alpha 是角 BAC 的大小
41     return fabs((B-A).cross(C-A)/A.dis(B));
42 }
43
44 inline double Get_Dis_Segment(Point A, Point B, Point C, Point D)
45 { //求出线段 AB 和线段 CD 之间的最短距离只需计算端点最短距离,
46     double res=min(Get_Dis_Point(A,B,C),Get_Dis_Point(A,B,D));
47     res=min(res,Get_Dis_Point(C,D,A));
48     res=min(res,Get_Dis_Point(C,D,B));
49     return res;
50 }
51
52 inline double Get_Min_Dis(Point point1[], Point point2[], int num1, int num2)
53 { //获取两相离凸包的最短距离, point1[], point2[] 和 num1,num2 分别是两凸包顶点数组及顶点数量
54     int ymin=0,ymax=0;
55     // 获取一个凸包的 y 值最小点
56     for(int i=0;i<num1;i++){
57         if(point1[i].y<point1[ymin].y){
58             ymin=i;
59         }
60     }
61     // 获取另一个凸包的 y 值最大点
62     for(int i=0;i<num2;i++){
63         if(point2[i].y>point2[ymax].y){
64             ymax=i;
65         }
66     }
67     point1[num1]=point1[0], point2[num2]=point2[0];
68     double tmp, ans=point1[ymin].dis(point2[ymax]);
69     for(int i=0;i<num1;i++){
70         while(1){
71             tmp=(point2[ymax+1]-point1[ymin+1]).cross(point1[ymin]-point1[ymin+1])-
72                 (point2[ymax]-point1[ymin+1]).cross(point1[ymin]-point1[ymin+1]);

```

```
73         if (tmp>eps) break;
74         ymax=(ymax+1)%num2;
75     }
76     if (tmp<-eps){ // 只旋转到一条边上, 计算点到线段距离
77         ans=min(ans,Get_Dis_Point(point1[ymin],point1[ymin+1],point2[ymax]));
78     } else { // 同时旋转到两条边上
79         ans=min(ans,Get_Dis_Segment(point1[ymin],point1[ymin+1],
80                                         point2[ymax],point2[ymax+1]));
81     }
82     ymin=(ymin+1)%num1;
83 }
84 return ans;
85 }
86
87 int main()
88 {
89     while(~scanf("%d%d",&n,&m)&&(n|m)){
90         for( int i=0;i<n; i++){
91             scanf("%lf%lf",&P[i].x,&P[i].y);
92         }
93         for( int j=0;j<m; j++){
94             scanf("%lf%lf",&Q[j].x,&Q[j].y);
95         }
96         Clockwise(P,n); // 将顶点顺时针排序
97         Clockwise(Q,m);
98         double ans=min(Get_Min_Dis(P,Q,n,m),Get_Min_Dis(Q,P,m,n));
99         printf("%f\n",ans);
100    }
101    return 0;
102 }
```

4.3 三维几何

4.3.1 任意四面体的外接球半径公式

若四面体 ABCD 中, $\angle CAD = \alpha$, $\angle CBD = \beta$, 二面角 $A - CD - B$ 的大小为 θ , $CD = 2m$, 则其外接球半径:

$$R = \frac{m}{\sin \alpha \sin \beta \sin \theta} \cdot \sqrt{1 - (\cos \alpha \cos \beta + \sin \alpha \sin \beta \cos \theta)^2}$$

4.3.2 定义

```

1 struct Point { // 三维点定义
2     double x, y, z;
3
4     Point() {}
5     Point(double _x, double _y, double _z): x(_x), y(_y), z(_z) {}
6
7     void input() {
8         scanf("%lf%lf%lf", &x, &y, &z);
9     }
10    void output() {
11        printf("%.2lf %.2lf %.2lf\n", x, y, z);
12    }
13    bool operator == (const Point& rhs) const {
14        return sgn(x - rhs.x) == 0 && sgn(y - rhs.y) == 0 && sgn(z - rhs.z) == 0;
15    }
16    double len() {
17        return sqrt(x * x + y * y + z * z);
18    }
19    double len2() {
20        return x * x + y * y + z * z;
21    }
22    double dis(const Point& rhs) const {
23        return sqrt((x - rhs.x) * (x - rhs.x) +
24                    (y - rhs.y) * (y - rhs.y) + (z - rhs.z) * (z - rhs.z));
25    }
26    Point operator - (const Point& rhs) const {
27        return Point(x - rhs.x, y - rhs.y, z - rhs.z);
28    }
29    Point operator + (const Point& rhs) const {
30        return Point(x + rhs.x, y + rhs.y, z + rhs.z);
31    }
32    Point operator * (const double& d) const {
33        return Point(x * d, y * d, z * d);
34    }
35    Point operator / (const double& d) const {
36        return Point(x / d, y / d, z / d);
37    }
38    double dot(const Point& rhs) const { // 点乘
39        return x * rhs.x + y * rhs.y + z * rhs.z;
40    }
41    Point cross(const Point& rhs) const { // 叉乘
42        return Point(y * rhs.z - z * rhs.y,
43                     z * rhs.x - x * rhs.z, x * rhs.y - y * rhs.x);
44    }
45    double area(const Point& rhs1, const Point& rhs2) const { // 空间三点三角形面积
46        Point cur = *this;
47        Point ret = (rhs1 - cur).cross(rhs2 - cur);
48        return ret.len() / 2.0;
49    }
50    double point_to_plane(const Point& rhs1, const Point& rhs2,
51                         const Point& rhs3) const { // 点到平面距离
52        Point cur = *this;
53        Point ret = (rhs2 - rhs1).cross(rhs3 - rhs1);

```

```

54     return ret.dot(cur - rhs1) / ret.len();
55 }
56 double volume(const Point& rhs1, const Point& rhs2, const Point& rhs3) const{
57     // 四面体体积
58     Point cur = *this;
59     double s = rhs1.area(rhs2, rhs3);
60     double h = cur.point_to_plane(rhs1, rhs2, rhs3);
61     return fabs(s * h / 3.0);
62 }
63 };

```

```

1 struct Line { // 三维线段、直线定义
2     Point st, ed;
3
4     Line() {}
5     Line(Point _st, Point _ed): st(_st), ed(_ed) {}
6
7     void input() {
8         st.input();
9         ed.input();
10    }
11    bool operator == (const Line& rhs) const {
12        return st == rhs.st && ed == rhs.ed;
13    }
14    double length() {
15        return st.dis(ed);
16    }
17    double point_to_line(const Point& rhs) const { // 点到直线距离
18        return ((ed - st).cross(rhs - st)).len() / st.dis(ed);
19    }
20    double point_to_seg(const Point& rhs) const { // 点到线段距离
21        if (sgn((rhs - st).dot(ed - st)) < 0 || sgn((rhs - ed).dot(st - ed)) < 0) {
22            return min(rhs.dis(st), rhs.dis(ed));
23        }
24        return point_to_line(rhs);
25    }
26    Point projection(const Point& rhs) const { // 点到直线的投影
27        return st + ((ed - st) * ((ed - st).dot(rhs - st))) / ((ed - st).len2());
28    }
29    bool point_on_line(const Point& rhs) const { // 判断点是否在直线上
30        return sgn( (st - rhs).cross(ed - rhs)).len() == 0 &&
31                sgn( (st - rhs).dot(ed - rhs) ) == 0;
32    }
33};

```

```

1 struct Plane {
2     Point a, b, c, o; // 平面上的三个点以及法向量
3
4     Plane() {}
5     Plane (Point _a, Point _b, Point _c) {
6         a = _a;
7         b = _b;
8         c = _c;
9         o = pvec();
10    }
11    Point pvec() { // 法向量
12        return (b - a).cross(c - a);
13    }
14    bool point_on_plane(const Point& rhs) const { // 点在平面上的判断
15        return sgn((rhs - a).dot(o)) == 0;
16    }
17    double angleplane(const Plane& rhs) const { // 两平面夹角
18        return acos(o.dot(rhs.o)) / (o.len() * rhs.o.len());
19    }
20};

```

4.3.3 给定四个点，求四面体内切球球心和半径

[HDU 5733] 给定四个点，求四面体内切球球心和半径。

设四面体 $A_1A_2A_3A_4$ 的顶点 $A_i(i=1, 2, 3, 4)$ 所对的侧面 面积为 S_i ，顶点 A_i 坐标为 (x_i, y_i, z_i) ，四面体表面积之和为 sum ，则四面体内心 I 的坐标 (x, y, z) 为：

$$x = \frac{\sum_{i=1}^{i=4} s_i * x_i}{sum}$$

$$y = \frac{\sum_{i=1}^{i=4} s_i * y_i}{sum}$$

$$z = \frac{\sum_{i=1}^{i=4} s_i * z_i}{sum}$$

```

1 int main()
2 {
3     Point p[5];
4     double sum, s[5];
5     while (~scanf("%lf%lf%lf%lf", &p[0].x, &p[0].y, &p[0].z)) {
6         p[1].input(); p[2].input(); p[3].input();
7         Plane plane = Plane(p[0], p[1], p[2]);
8         if (plane.point_on_plane(p[3])) { // 四点共面
9             printf("O O O O\n");
10            continue;
11        }
12        sum = 0;
13        s[0] = p[1].area(p[2], p[3]);
14        s[1] = p[0].area(p[2], p[3]);
15        s[2] = p[0].area(p[1], p[3]);
16        s[3] = p[0].area(p[1], p[2]);
17        for (int i = 0; i < 4; ++i) sum += s[i];
18        double V = p[0].volume(p[1], p[2], p[3]);
19        double r = V * 3.0 / sum; // r * 四个面的总面积 / 3 = 体积
20        double ansx = 0, ansy = 0, ansz = 0;
21        for (int i = 0; i < 4; ++i) {
22            ansx += s[i] * p[i].x;
23            ansy += s[i] * p[i].y;
24            ansz += s[i] * p[i].z;
25        }
26        ansx /= sum, ansy /= sum, ansz /= sum;
27        printf("%.4lf %.4lf %.4lf %.4lf\n", ansx, ansy, ansz, r);
28    }
29    return 0;
30 }
```

4.3.4 给定四面体的六条边长求四面体体积

[POJ 2208] 给定四面体的六条边长求四面体体积

设六条边长依次为 l, m, n, p, q, r ，欧拉四面体体积公式：

$$V = \frac{1}{36} * \begin{vmatrix} p^2 & \frac{p^2+q^2-n^2}{2} & \frac{p^2+r^2-m^2}{2} \\ \frac{p^2+q^2-n^2}{2} & q^2 & \frac{q^2+r^2-l^2}{2} \\ \frac{p^2+r^2-m^2}{2} & \frac{q^2+r^2-l^2}{2} & r^2 \end{vmatrix}$$

```

1 int main()
2 {
3     double l, n, m, p, q, r;
4     while (~scanf("%lf%lf%lf%lf%lf%lf", &p, &q, &r, &n, &m, &l)) {
5         double tmp1 = (p * p + q * q - n * n) / 2.0;
```

```
6     double tmp2 = (p * p + r * r - m * m) / 2.0;
7     double tmp3 = (q * q + r * r - l * l) / 2.0;
8     double ans1 = p * p * (q * q * r * r - tmp3 * tmp3);
9     double ans2 = tmp1 * (tmp1 * r * r - tmp2 * tmp3);
10    double ans3 = tmp2 * (tmp1 * tmp3 - q * q * tmp2);
11    double ans = sqrt(ans1 - ans2 + ans3) / 6.0;
12    printf("%.4lf\n", ans);
13 }
14 return 0;
15 }
```


Chapter 5

数论

5.1 一些理论

5.1.1 大整数取模

把大整数写成“从左向右”的形式：如： $1234 = ((1 * 10 + 2) * 10 + 3) * 10 + 4$. 然后根据 $(n + m) \% p = ((n \% p) + (m \% p)) \% p$, 每步取模。

```
1 scanf("%s%d", &s, &p);
2 int len = strlen(s);
3 if(s[0] == '0' && len == 1){
4     printf("divisible\n");
5     continue;
6 }
7 if(p < 0) p = -p; //判断负数
8 int ans = 0, st = 0;
9 if(s[0] == '-') st = 1;
10 for(int i = st; i < len; i++){
11     ans = (int)((11) ans * 10 + s[i] - '0') % p); //注意爆int
12 }
13 if(ans == 0) printf("divisible\n");
14 else printf("not divisible\n");
```

5.1.2 哥德巴赫猜想（ $1+1$ 问题）

大于 2 的所有偶数都可以表示为两个素数的之和，大于 5 的所有奇数均可以表示为三个素数之和。
陈景润证明了：所有大于 2 的偶数均是一个素数和只有两个素数因数的合数之和（ $1+2$ 问题），称为陈氏定理。

5.1.3 费马小定理

p 是素数，且 $a \not\equiv 0 \pmod{p}$ ，则： $a^{p-1} \equiv 1 \pmod{p}$.

5.1.4 素数定理

$f(x) \approx \frac{x}{\ln(x)}$ ($f(x)$ 为不超过 x 的素数的个数)
 10^8 级别的素数筛选出的素数个数约为 $6 * 10^6$ 级别
 10^7 级别的素数筛选出的素数个数约为 $7 * 10^5$ 级别

5.1.5 算数基本定理

任何一个大于 1 的自然数 n ，若其不为素数则必可唯一的分解为有限个素数的乘积，即 $n = p_1^{a_1} * p_2^{a_2} * p_3^{a_3} * \dots * p_k^{a_k}$ ，那么同时 n 的因子个数为 $num = \prod(a_i + 1) (1 \leq i \leq k)$ 【LightOJ 1341】。

5.1.6 $ax + by = c$ 的任意整数解

a, b, c 为任意整数, 若 $ax + by = c$ 的一组整数解为 (x_0, y_0) , 则它的任意整数解为 $(x_0 + kb', y_0 - ka')$. 其中 $a' = \frac{a}{\gcd(a, b)}$, $b' = \frac{b}{\gcd(a, b)}$, k 取任意整数。

证明: 令 $g = \gcd(a, b)$. 设另一组解为 (x_1, y_1) , 则 $ax_0 + by_0 = ax_1 + by_1 (= c)$. 移项: $a(x_1 - x_0) = b(y_0 - y_1)$. 同除以 g 可得: $\frac{a}{g} * (x_1 - x_0) = \frac{b}{g} * (y_0 - y_1)$. 令 $a' = \frac{a}{g}$, $b' = \frac{b}{g}$. 则 $a'(x_1 - x_0) = b'(y_0 - y_1)$ 此时 a' 与 b' 互质. 所以 $x_1 - x_0$ 一定是 b' 的整数倍, 设倍数为 k . 则有: $x_1 - x_0 = kb'$, $y_0 - y_1 = ka'$, 可得: $x_1 = x_0 + kb'$, $y_1 = y_0 - ka'$. 结论加强: a, b, c 为任意整数, $g = \gcd(a, b)$, 方程 $ax + by = g$ 的一组解是 (x_0, y_0) . 则当 c 是 g 的倍数时, $ax + by = c$ 的一组解为 $(\frac{x_0*c}{g}, \frac{y_0*c}{g})$. 当 c 不为 g 的倍数时 $ax + by = c$ 无整数解。

5.1.7 n 是 m 的倍数, $[1, n]$ 中和 m 互素数个数

对于两个正整数 m 和 n , 如果 n 是 m 的倍数, 那么 $1 \sim n$ 中与 m 互素的数的个数为 $\frac{n}{m} * \phi(m)$. ($\phi(m)$ 为小于等于 m 的且与 m 互素的数的个数)

5.1.8 p 为奇素数, $1 \sim (p-1)$ 模 p 的逆元对应全部 $1 \sim (p-1)$ 中的所有数, 既是单射也是满射

也就是 $(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{(p-1)}) \bmod(p) = (1 + 2 + \dots + (p-1)) \bmod(p)$.

5.1.9 调和级数求和

令 $h[n]$ 表示 n 级调和级数的和, 即: $h[n] = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$. 则当 $n > 10^6$ 时可以用公式:

$$h[n] = \log(n * 1.0) + \frac{\log(n+1.0)}{2.0} + \frac{1.0}{(6.0 * n * (n+1))} - \frac{1.0}{(30.0 * n * n * (n+1) * (n+1))} + r$$

其中 r 为欧拉常数: $r = 0.57721566490153286060651209008240243104215933593992$. 上述公式误差不超过 10^{-8} .

5.1.10 如果 $\gcd(a, m) = 1$, 那么 $\gcd(a + k * m, m) = 1, k \in Z$

【POJ 2773】

证明: 只需要证明 $\gcd(a + m, m) = 1$. 令 $d = \gcd(a + m, m)$, 设 $a + m = d * p_1, m = d * p_2$, $p_1 > p_2$, 移项可得: $a = d * (p_1 - p_2)$, 如果 $d! = 1$, 那么 $\gcd(a, m)! = 1$, 与原条件不符, 故 $\gcd(a, m) = 1$, 同理可证 $\gcd(a + k * m, m) = 1, k \in Z$.

5.1.11 指数降幂公式

$$A^x \% p = A^{x \% \phi(p) + \phi(p)} \% p \quad (x \geq \phi(p))$$

其中 $\phi(p)$ 是 p 的欧拉函数值

5.2 素数筛

```

1 void GetPrime( int n ) //获得 n 以内的所有质数
2 {
3     prime_cnt = 0;
4     int m = sqrt(n + 0.5);
5     memset(vis, 0, sizeof(vis));
6     for( int i = 2; i <= m; i++){
7         if( vis[i] == 0){
8             prime[prime_cnt++] = i;
9             for( int j = i * i; j < n; j+= i){
10                 vis[j] = 1;
11             }
12         }
13     }
14 }
15
16 //线性时间素数筛: 利用每个合数必有一个最小素因子, 每个合数仅被它的最小素因子筛去
17 //代码核心: if( i % prime[j] == 0) break;
18 void GetPrime()
19 {
20     prime_cnt = 0;
21     memset(vis, 0, sizeof(vis));
22     for( int i = 2; i < MAX_N; i++){
23         if( vis[i] == 0) { prime[prime_cnt++] = i; }
24         for( int j = 0; j < prime_cnt && i * prime[j] < MAX_N; j++){
25             vis[i * prime[j]] = 1;
26             //将所有合数标记, prime[j] 是 i * prime[j] 的最小素因子
27             if( i % prime[j] == 0) break;
28         }
29     }
30 }
```

5.2.1 区间素数筛

```

1 const int MAX_N = 100010; // 最大区间长度
2
3 int prime_cnt;
4 int prime_list[MAX_N];
5 bool prime[MAX_N]; //prime[i] = true:i + L 是素数
6
7 void SegmentPrime( int L, int R)
8 {
9     int len = R - L + 1; // 区间长度
10    for( int i = 0; i < len; i++) prime[i] = true; // 初始全部为素数
11    int st = 0;
12    if( L % 2) st++; // L 是奇数
13    for( int i = st; i < len; i += 2) { prime[i] = false; }
14    // 相当于 i + L 是合数, 把 [L, R] 的偶数都筛掉
15    int m = (int)sqrt(R + 0.5);
16    for( int i = 3; i <= m; i += 2){ // 用 [3, m] 之间的数筛掉 [L, R] 之间的合数
17        if( i > L && prime[i - L] == false) { continue; }
18        // i 是 [L, R] 区间内的合数此时, [L, R] 区间中以 i 为基准的合数已经被筛掉了
19        int j = (L / i) * i; // 此时 j 是 i 的倍数中最接近 L 小于等于( L 的数)
20        if( j < L) j += i; // j >= L
21        if( j == i) j += i;
22        // 如果 j>=L 且 j==i 且 i 为素数, 则相当于 [L, R] 中的 i 也为素数, 所以要 j+=i
23        j -= L; // 把 j 调整为 [0, len) 之间
24        for( ; j < len; j += i) {
25            prime[j] = false;
26        }
27    }
28    if( L == 1) prime[0] = false;
```

```

29     if (L <= 2) prime[2 - L] = true;    // 特别注意 2 的情况!
30     prime_cnt = 0;
31     for (int i = 0; i < len; i++) {
32         if (prime[i]) prime_list[prime_cnt++] = i + L;
33     }
34 }
```

5.3 约数筛

定义 $e[i]$ 表示 i 的最小质因子的幂次， $div_num[i]$ 表示 i 的约数个数。
考虑在素数筛的过程中， $e[i]$ 的更新：

- i 为质数: $e[i]$
- $i \% prime[j] == 0 : e[i * prime[j]] = e[i] + 1$
- $i \% prime[j] != 0 : e[i * prime[j]] = 1$

$div_num[i]$ 的更新：

- i 为质数: $div_num[i] = 1$
- $i \% prime[j] == 0 : div_num[i * prime[j]] = \frac{div_num[i]}{e[i]+1} * (e[i] + 2)$
- $i \% prime[j] != 0 : div_num[i * prime[j]] = div_num[i] * div_num[prime[j]]$ (积性函数的性质)

```

1 const int MAX_N = 1000010;
2
3 int prime_cnt = 0;
4 int prime[MAX_N], e[MAX_N], div_num[MAX_N], vis[MAX_N];
5
6 void Sieve() {
7     e[1] = 0, div_num[1] = 1;
8     for (int i = 2; i < MAX_N; ++i) {
9         if (vis[i] == 0) {
10             prime[prime_cnt++] = i;
11             e[i] = 1;
12             div_num[i] = 2;
13         }
14         for (int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j) {
15             int now = i * prime[j];
16             vis[now] = 1;
17             if (i % prime[j] == 0) {
18                 e[now] = e[i] + 1;
19                 div_num[now] = div_num[i] / (e[i] + 1) * (e[i] + 2);
20                 break;
21             } else {
22                 e[now] = 1;
23                 div_num[now] = div_num[i] * div_num[prime[j]];
24             }
25         }
26     }
27 }
```

5.3.1 51nod 1586

有三个下标从 1 到 n 的数组 a, b, c 。 a 数组初始全为 0。

$$b[i] = \sum_{j|i} a[j] \quad c[i] = \sum_{j|i} b[j]$$

需要进行下列操作：

- 1 $x\ y$: 将 $a[x]$ 加上 y ($x \in [1, n], y \in [1, 10^6]$)
- 2 x : 询问当前 $c[x]$ ($x \in [1, n]$) 的值

对每次查询输出答案。

每次 $a[x]$ 添加 y , 那么 $b[r]_{i|r}$ 都会添加 y , 而 $c[s]_{i|s}$ 添加的数字是: $\text{div_num}[\frac{s}{i}] * x$ 。

```

1 int n, Q;
2 ll C[MAX_N];
3
4 int main() {
5     Sieve();
6     scanf("%d%d", &n, &Q);
7     while (Q--) {
8         int type, x, y;
9         scanf("%d", &type);
10        if (type == 1) {
11            scanf("%d%d", &x, &y);
12            for (int i = x; i <= n; i += x) {
13                C[i] += 111 * div_num[i / x] * y;
14            }
15        } else {
16            scanf("%d", &x);
17            printf("%lld\n", C[x]);
18        }
19    }
20    return 0;
21 }
```

5.4 分解质因数

```

1 void factor(int x)
2 { // factot: 存质因数, factor_cnt[i] : 存 factor[i] 的指数, factor_num : 质因数的个数
3     GetPrime();
4     factor_num = 0;
5     memset(factor_cnt, 0, sizeof(factor_cnt));
6     for (int i = 0; i < prime_cnt && prime[i] * prime[i] <= x; i++) {
7         if (x % prime[i] == 0) {
8             int tmp = 0;
9             while (x % prime[i] == 0) {
10                 x /= prime[i];
11                 tmp++;
12             }
13             factor[factor_num] = prime[i];
14             factor_cnt[factor_num++] = tmp;
15         }
16     }
17 }
```

5.4.1 给定 n 求满足 $a^p = n$ 的最大 p

【LightOJ 1220】

需要考虑 n 的正负。对于正数 n 只要对 n 进行质因子分解, 然后在所有质因子的幂中找 gcd 即可。对于负整数 n 需要考虑质因子的幂为偶数的情况, 因为得到的质因子实际上是它的相反数, 如果是偶数次幂的话得到的是正数, 所以需要将偶数次幂除以 2 找到把幂变为奇数。例如对于 -16 , 分解质因子得到 2 和幂次 4. 需要将 $\frac{4}{2}$ 得 2, 再除以 2 得 1, 所以答案就是 1. 然而对于 $n = -32$ 来说, 就不用了, 因为得到 2 的幂次是 5 是个奇数, 答案就是 5.

5.4.2 求满足 $\text{lcm}(i, j) = n (1 \leq i \leq j \leq n \leq 10^{14})$ 的 (i, j) 对数

【LightOJ 1236】

假设 n 质因子分解为: $n = p_1^{e_1} * p_2^{e_2} * p_3^{e_3} * \dots * p_k^{e_k}$ 。对于任意 p_i 的幂 e_i , 当对 $i j$ 进行质因数分解后, 相应的到的 p_i 的幂为 a_i 和 b_i , 那么一定满足 $\max(a_i, b_i) = e_i$. 如果 $a_i = e_i$, 那么 b_i 可以取 $[0, e_i]$, 共 $e_i + 1$ 种, 如果 $a_i < e_i$, 那么 b_i 只取 e_i , 这时 a_i 可以取 $[0, e_i]$, 共 e_i 种。合起来一共是 $2 * e_i + 1$ 种。所以对每个质因子这样考虑的话可以得到: $ans = (2 * e_i + 1)$. 但是要考虑到 $i \leq j$, 所以 $ans = \frac{ans}{2}$, 但是上面的考虑在所有的 p_i 当中 $a_i = e_i$ 同时 $b_i = e_i$ 只考虑了一次, 也就是 $i = j = n$ 只考虑了一次。所以还要 $ans = ans + 1$.

5.5 逆元

5.5.1 介绍

对于正整数 a, p 满足 $ax \equiv 1 (\% p)$ 的最小正整数 x 称为 a 的逆元。

乘法逆元的存在条件是: $\gcd(a, p) = 1$ 。

\equiv : 同余符号。 $a \equiv b (\text{mod } n)$ 的含义是” a 和 b 关于模 n 同余”, 即 $a \text{ mod } n = b \text{ mod } n$, 其充要条件是: $a - b$ 是 n 的整数倍. 当 $\gcd(a, n) = 1$ 时, 该方程有唯一解, 否则该方程无解。

特例:

1: 当 p 是素数且 $a \neq 0 (\text{mod } p)$ 时由费马小定理可得: $x \equiv a^{(p-2)} (\text{mod } p)$.

2: 已知 $b | a$, 则 $(\frac{a}{b}) \text{ mod } m = \frac{a \text{ mod } (mb)}{b}$ 。这个式子通常用于模数和分母不互素的情况, 这样就不能用费马小定理和扩展欧几里德求解, 必须将模数扩大为 $m * b$, 最后答案除以 b 。

证明: 假设 $\frac{a}{b} \text{ mod } (m) = x$,

即: $\frac{a}{b} = k * m + x (x < m)$

$\rightarrow a = k * b * m + b * x$

$\rightarrow a \text{ mod } (bm) = bx$

$\rightarrow \frac{a \text{ mod } (bm)}{b} = x$

$\rightarrow \frac{a}{b} \text{ mod } (m) = \frac{a \text{ mod } (bm)}{b} = x$

5.5.2 模素数逆元连乘

如果有的题目需要用到 $1 \sim n$ 模 M 的所有逆元 (M 为奇质数), 如果用快速幂求解时间复杂度为 $O(M * \log(M))$. 实际上有 $O(M)$ 的算法, 递推式如下:

$$\text{inv}[i] = (M - \frac{M}{i}) * \text{inv}[M \% i \% M]$$

推导过程如下:

设 $t = \frac{M}{i}, k = M \% i \rightarrow t * i + k = 0 (\text{mod } M) \rightarrow -t * i = k (\text{mod } M)$. 两边同时除以 $i * k$ 得:
 $\rightarrow -t * \text{inv}[k] = \text{inv}[i] (\text{mod } M)$. 再把 t 和 k 替换掉最终得到:

$$\text{inv}[i] = (M - \frac{M}{i}) * \text{inv}[M \% i \% M]$$

初始化 $\text{inv}[1] = 1$. 这样就可以通过递推法求出 $1 \sim n$ 模 M 的所有逆元了。

```

1 11 inv(11 a, 11 p) // 利用 inv[a] = (p - p / a) * inv[p % a] % p 求逆元
2 { // 需要保证 a < p 且 gcd(a, p) = 1
3     if(a == 1) return 1;
4     return inv(p % a, p) * (p - p / a) % p;
5 }
```

5.6 欧拉函数

5.6.1 介绍

定义: $\phi[n]$: 小于等于 n 且与其互素的正整数的个数

欧拉定理: $a^{\phi(n)} \equiv 1 (\text{mod } n) (\gcd(a, n) = 1)$

特例是费马小定理。利用这个定理求模意义下的乘法逆元: $a^{-1} \equiv a^{\phi(n)-1} \pmod{n}$

- $\phi(1) = 1$
- $\phi(N) = N \cdot \prod \frac{p-1}{p}$ (p 为 N 的所有素因数)
- $\phi(p^k) = p^k - p^{k-1} = (p-1) \cdot p^{k-1}$, 其中 p 为素数
- $\phi(m * n) = \phi(m) \cdot \phi(n)$, 其中 $\gcd(m, n) = 1$
- $\sum_{d|n} \phi(d) = n$
- 当 p 为素数时:

$$\phi(t * p) = \begin{cases} p * \phi(t) & t \% p = 0 \\ (p-1) * \phi(t) & t \% p! = 0 \end{cases}$$

- 当 $n > 1$ 时, $1 \dots n$ 中与 n 互质的整数和是 $\frac{n\phi(n)}{2}$
- 当 $n \geq 3$ 时, $\phi[n]$ 是偶数
- 两相邻质数之间合数的欧拉函数值小于较小的素数, 所以对于一个数 x 要找到最小的 t 使得 t 的欧拉函数值 $\phi[t] \geq x$ 则 t 必然是大于 x 的第一个素数。
- 若 $(n \% p == 0 \&\& \frac{n}{p} \% p == 0)$ 则有: $\phi(n) = \phi(\frac{n}{p}) * p$;
- 若 $(n \% p == 0 \&\& \frac{n}{p} \% p != 0)$ 则有: $\phi(n) = \phi(\frac{n}{p}) * (p-1)$;

其中 p 是 n 的质因数

5.6.2 求单个数的欧拉函数

求欧拉函数: 先令 $\phi[i] = i$, 根据性质 2, 遍历所有素数 p , 令 $\phi[kp] = \frac{\phi[kp]}{p} * (p-1)$

```

1 // 任何一个大于 1 的自然数 n , 若其不为素数则必可唯一的分解为有限个素数的乘积
2 int Euler(int n)
3 {
4     int ans = 1;
5     for (int i = 2; i * i <= n; i++){
6         if (n % i == 0){
7             n /= i;
8             ans *= (i - 1);
9             while (n % i == 0){
10                 n /= i;
11                 ans *= i;
12             }
13         }
14     }
15     if (n > 1) ans *= (n - 1);
16     return ans;
17 }
```

```

1 int Euler(int n)
2 {
3     int res = n, a = n;
4     for (int i = 2; i * i <= a; i++){
5         if (a % i == 0){
6             res = res / i * (i - 1); // 先进行除法防止数据溢出
7             while (a % i == 0) a /= i;
8         }
9     }
10    if (a > 1) res = res / a * (a - 1);
11    return res;
12 }
```

5.6.3 欧拉筛

```

1 void GetPhi() // 埃氏筛
2 {
3     phi[1] = 1;
4     for (int i = 2; i < MAX_N; i++) {
5         if (phi[i] == 0) {
6             for (int j = i; j < MAX_N; j += i) {
7                 if (!phi[j]) phi[j] = j;
8                 phi[j] = phi[j] / i * (i - 1);
9             }
10        }
11    }
12 }
```

```

1 bitset<MAX_N> bs;
2 int prime_cnt, prime[MAX_N], phi[MAX_N];
3 void GetPhi() // 欧拉筛同时获得素数表和欧拉表,
4 {
5     prime_cnt = 0;
6     bs.set();
7     for (int i = 2; i < MAX_N; ++i) {
8         if (bs[i] == 1) {
9             prime[prime_cnt++] = i;
10            phi[i] = i - 1;
11        }
12        for (int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j) {
13            bs[i * prime[j]] = 0;
14            if (i % prime[j]) {
15                phi[i * prime[j]] = (prime[j] - 1) * phi[i];
16            } else {
17                phi[i * prime[j]] = prime[j] * phi[i];
18                break;
19            }
20        }
21    }
22 }
```

[UVA 11428 GCD - Extreme (II): 给定 n 求: $G = \sum_{i=1}^{i < N} \sum_{j=i+1}^{j \leq N} \text{GCD}(i, j)$ ($n \leq 4000000$)

令 $\text{sum}[n]$ 为题式中答案。考虑递推 $\text{sum}[n] = \text{sum}[n - 1] + \text{gcd}(1, n) + \text{gcd}(2, n) + \text{gcd}(3, n) + \dots + \text{gcd}(n - 1, n)$ 。令 $f[n] = \text{gcd}(1, n) + \text{gcd}(2, n) + \text{gcd}(3, n) + \dots + \text{gcd}(n - 1, n)$ 。设满足 $\text{gcd}(x, n) = t \dots$ 的 $x (x < n)$ 的个数有 $h[t]$ 个，显然 x, t 均是 n 的约数，又因为不包含 $\text{gcd}(n, n)$ ，所以 $t < n$ 。将等式两边同除以 t 可得: $\text{gcd}(\frac{x}{t}, \frac{n}{t}) = 1$ 。所以 $h[t] = \phi[\frac{n}{t}]$ 。那么枚举 n 的约数可以得到: $f[n] = \sum(t * \phi[\frac{n}{t}])$ (t 为 n 的所有约数) $- n$ (相当于去掉 $\text{gcd}(n, n)$) 需要预处理 $f[n]$ 然后对于每个 n 有: $\text{sum}[i] = \text{sum}[i - 1] + f[i]$, $\text{sum}[1] = 0$; 递推即可。

```

1 typedef long long ll;
2 const int MAX_N = 4000010;
3
4 int n;
5 ll phi[MAX_N], f[MAX_N], sum[MAX_N];
6
7 void init()
8 {
9     GetPhi();
10    memset(f, 0, sizeof(f));
11    for (int i = 1; i < MAX_N; i++) {
12        for (int j = i; j < MAX_N; j += i) {
13            f[j] += i * phi[j / i];
14        }
15    }
16 }
```

```

18 int main()
19 {
20     init ();
21     while (~scanf ("%d", &n) && n){
22         sum[1] = 0;
23         for (int i = 2; i <= n; i++){
24             sum[i] = sum[i - 1] + f[i] - i;
25         }
26         printf ("%lld\n", sum[n]);
27     }
28     return 0;
29 }
```

5.7 扩展欧几里德

5.7.1 裴蜀定理

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

求解过程：因为 $ax + by = \gcd(a, b)$ 且 $\gcd(a, b) = \gcd(b, a \% b)$ 。那么有： $bx + (a \% b)y = \gcd(b, a \% b)$ ，即有： $bx + (a \% b)y = ax + by$ 。所以： $bx' + (a - \frac{a}{b} * b)y' = ax + by$ 移项： $ay' + b(x' - \frac{a}{b} * y') = ax + by$ 。得到： $x = y'$, $y = (x' - \frac{a}{b} * y')$ ，也就是 $x' = y - \frac{a}{b} * x$, $y' = x$ 。所以可以使用递归求解。

```

1 //需保证系数 a, b 应同为正数, 但是求解出来的 x, y 可正可负
2 int ex_gcd(int a, int b, int& x, int& y)
3 {
4     if (b == 0) {
5         x = 1, y = 0;
6         return a;
7     }
8     int r = ex_gcd(b, a % b, y, x);
9     y -= a / b * x;
10    return r;
11 }
```

解释 1: $y' = \frac{a}{b} * x$

由前面的推导可知： $x' = y - \frac{a}{b} * x$, $y' = x$, 其中 x, y 是真正解, x', y' 是递归调用时的下一层的解。所以需要将上一层的 x 传递给下一层的 y' , 另一方面因为 x 和 y 是未知的所以不能直接将上一层的 $y - a/b * x$ 传递给下一层的 x' , 但是可以先传递 y , 相当于人为的增加了 $\frac{a}{b} * x$, 所以当计算出真正的 x, y 时需要减去 $\frac{a}{b} * x$. 解释 2: $x = 1, y = 0$

递归的最后一层的方程式的是： $\gcd(a, b) * x + 0 * y = \gcd(a, b)$. 要使这个式子恒成立, 显然需要 $x = 1$ 递归的倒数第二层的方程式是： $k * \gcd(a, b) * x' + \gcd(a, b) * y' = \gcd(a, b)$. 其中 k 为任意整数。约掉 $\gcd(a, b)$ 可得： $k * x' + y' = 1$. 此式要恒成立显然 $x' = 0, y' = 1$ 。又因为此式中的 $x' = y, y' = x - \frac{a}{b} * y$. 所以 $y = 0$. 当然这时 $y' = x - \frac{a}{b} * y = 1 - \frac{a}{b} * 0 = 1$. 正好也是符合的。

5.7.2 求解逆元 $ax \equiv 1 \pmod{n}$ 的最小非负数解

先求解： $ax + ny = \gcd(a, n)$. 如果 $\gcd(a, n) = 1$, 则 $ans = (x \% n + n) \% n$. 否则无解。所以题目中经常是 n 为一个很大的素数, 这样保证了 $\gcd(a, n) = 1$.

这里得到的是最小非负数解 x , 如果要求 x 为正数还要在 $return$ 时特判： $if(x == 0)x = n;$

```

1 int Inv(int a, int n) // Module Inverse 模逆元
2 {
3     int d, x, y;
4     d = ex_gcd(a, n, x, y);
5     if(d == 1) return (x % n + n) % n;
6     // a 和 n 的最小公倍数是 1 此时解存在,
7     else return -1; //no solution
8 }
```

5.7.3 求所有解中 $C = |x| + |y|$ 最小值

假设基础解为 (x_0, y_0) , 那么通解可以表示为: $x = x_0 + k * b, y = y_0 - k * a$ ($k \in \mathbb{Z}$)。如果令 $x = 0$ 得: $k = -\frac{x_0}{b} \dots$, 令 $y = 0$ 得: $k = \frac{y_0}{a} \dots$ 当两者同时满足并根据分数的性质: $\frac{a}{b} = \frac{c}{d} = \frac{(a+c)}{(b+d)}$ 可得: $k = \frac{(y_0-x_0)}{(b+a)}$, 但是考虑到这个数的真实值可能为浮点数, 需要左右考虑。

```

1  ll solve(ll a, ll b, ll t) // 方程为 a * x + b * y = t
2  {
3      ll x, y, d, res;
4      d = ex_gcd(a, b, x, y);
5      if(t % d) return -1;
6      a /= d, b /= d, t /= d;
7      x *= t, y *= t; // 此时 x, y 为基础解
8      res = (ll)1e18;
9      ll tmpx, tmpy, c = (y - x) / (a + b);
10     for(int i = c - 1; i <= c + 1; i++){ // 左右考虑
11         tmpx = x + i * b, tmpy = y - i * a;
12         res = min(res, abs(tmpx) + abs(tmpy));
13     }
14     return res;
15 }
```

5.7.4 求最小非负整数解 x 和此时的 y

```

1  ll extra = 0;
2  if(x < 0) extra = 1;
3  ll t = x / b - extra;
4  printf("%lld %lld\n", x - t * b, y + t * a);
```

5.8 模线性方程组

输入正整数 a, b, n , 解方程 $ax \equiv b \pmod{n}$. $a, b, n \leq 10^9$

把 $ax \equiv b \pmod{n}$ 转化为 $ax - ny = b$, 当 $d = \gcd(a, n)$ 不是 b 的约数时无解, 否则两边同时除以 d , 得到 $a'x - n'y = b'$, 即 $a'x \equiv b' \pmod{n'}$ (这里 $a' = \frac{a}{d}, b' = \frac{b}{d}, n' = \frac{n}{d}$). 此时 a' 和 n' 已经互素, 因此只需要左乘 a' 在模 n' 下的逆, 则解为 $x \equiv (a')^{-1} * b' \pmod{n'}$, 这个解是模 n' 剩余系中的一个元素。还需要把解表示成模 n 剩余系中的元素。

令 $p = (a')^{-1} * b'$, 上述解相当于 $x = p, p + n', p + 2 * n', p + 3 * n' \dots$ 。对于模 n 来说, 假定 $p + i * n'$ 和 $p + j * n'$ 同余, 则 $(p + i * n') - (p + j * n') = (i - j) * n'$ 是 n 的倍数。因此 $(i - j)$ 必须是 d 的倍数。也就是说, 在模 n 剩余系下, $ax \equiv b \pmod{n}$ 恰好有 d 个解, 为 $p, p + n', p + 2 * n', \dots, p + (d - 1) * n'$.

5.8.1 利用扩展欧几里德求解模线性同余方程 $ax \equiv b \pmod{n}$

如果 $\gcd(a, b)$ 不能整除 c 则 $ax + by = c$ 无整数解。对于 $ax \equiv b \pmod{n}$ ($n > 0$), 上式等价于二元一次方程 $ax - ny = b$

```

1  bool ModularLinearEquation(int a, int b, int n)
2  {
3      int x, y, x0;
4      int d = ex_gcd(a, n, x, y); // d = gcd(a, n)
5      if(b % d) return false; // d 不能整除 b
6      x0 = x * (b / d) % n; // basic solution
7      for(int i = 1; i <= d; i++){
8          printf("%d\n", (x0 + i * (n / d) % n));
9      }
10     return true;
11 }
```

[SGU 106] 给出 $a, b, c, x_1, x_2, y_1, y_2$ 求满足 $ax + by + c = 0$ 且 $x \in [x_1, x_2], y \in [y_1, y_2]$ 的 x, y 有多少组。

相当于问在直线 $ax + by + c = 0$ 上有多少整点 (x, y) 满足 $x \in [x_1, x_2], y \in [y_1, y_2]$. 扩展欧几里德的应用。需要特别注意 $a = 0, b = 0, c = 0$ 的特判。

```

1  typedef long long ll;
2  double eps = 1e-8;
3
4  ll a, b, c, x1, yy1, x2, y2, ans;
5
6  ll ex_gcd(ll n, ll m, ll& x, ll& y) // 返回值是 gcd(a, b)
7  {
8      if (m == 0){
9          x = 1, y = 0;
10         return n;
11     }
12     ll d = ex_gcd(m, n % m, y, x);
13     y = n / m * x;
14     return d;
15 }
16
17 void ModularLinearEquation() // 模线性方程组
18 {
19     ll x, y, x0, y0;
20     ll d = ex_gcd(a, b, x, y); // 求解方程 ax+by=gcd(a, b)
21     if (c % d) {
22         printf("0\n");
23         return ;
24     }
25     // 此时的 x, y 是方程 ax + by = gcd(a, b) 的一组基础解
26     a /= d, b /= d, c /= d;
27     x0 = x * c, y0 = y * c; // x0, y0 是 ax + by = c 的一组基础解
28     // a * x + b * y = c 通解满足: a * (x0 + k * b) + b * (y0 - k * a) = c
29     // 所以: x' = x0 + k * b ..., y' = y0 - k * a ..., k ∈ Z
30     // 先求出满足 1:x1 <= x0 + k * b <= x2 的 k 的范围 [l1, r1]
31     // 再求出满足 2:yy1 <= y0 - k * a <= y2 的 k 的范围 [l2, r2] 两者取交集即可
32     // l1 = ceil((x1 - x0)/b), r1 = floor((x2 - x0)/b)
33     // l2 = ceil((y0 - y2)/a), r2 = floor((y0 - yy1)/a)
34     ll l1 = (ll)ceil((double)(x1 - x0) / b);
35     ll r1 = (ll)floor((double)(x2 - x0) / b);
36     ll l2 = (ll)ceil((double)(y0 - y2) / a);
37     ll r2 = (ll)floor((double)(y0 - yy1) / a);
38     ll l = max(l1, l2), r = min(r1, r2);
39     ans = (r - l + 1 < 0) ? 0 : (r - l + 1);
40     printf("%lld\n", ans);
41     return ;
42 }
43
44 int main()
45 {
46     while (~scanf("%lld%lld%lld", &a, &b, &c)){
47         scanf("%lld%lld%lld%lld", &x1, &x2, &yy1, &y2);
48         int flag = 0;
49         c = -c; // 移项, 得到方程 a * x + b * y = -c
50         // 下面将方程的所有系数 a, b, c 变为非负数
51         if (c < 0){
52             c = -c, a = -a, b = -b;
53         }
54         if (a < 0){
55             // 需要将 a 取相反数, 相当于把 x 也取了相反数, 所以需要将 [x1, x2] 的范围变为 [-x2, -x1]
56             ll tmpx1 = x1, tmpx2 = x2;
57             x1 = -tmpx2, x2 = -tmpx1;
58             a = -a;
59         }
60         if (b < 0){ // 同理
61             ll tmpyy1 = yy1, tmpy2 = y2;
62             yy1 = -tmpy2, y2 = -tmpyy1;
63             b = -b;
64         }
65     }
}

```

```

66     if(a == 0 && b == 0) { //0 * x + 0 * y = c
67         flag = 1;
68         if(c == 0) ans = (x2 - x1 + 1) * (y2 - yy1 + 1);
69         else ans = 0;
70     }else if(a == 0){ // b != 0
71         flag = 1;
72         if(c % b == 0){ // 0 * x + b * y = c --> b * y = c
73             ll tmpy = c / b;
74             if(yy1 <= tmpy && tmpy <= y2) ans = x2 - x1 + 1;
75             else ans = 0;
76         }else ans = 0;
77     }else if(b == 0) { //a != 0
78         flag = 1;
79         if(c % a == 0) { // a * x + 0 * y = c --> a * x = c
80             ll ttmpx = c / a;
81             if(x1 <= ttmpx && ttmpx <= x2) ans = y2 - yy1 + 1;
82             else ans = 0;
83         }else ans = 0;
84     }
85     // 以上处理完了 a = 0 或者 b = 0 的所有特例
86     if(flag) printf("%lld\n", ans);
87     else ModularLinearEquation();
88 }
89 return 0;
}

```

5.9 中国剩余定理

给定整数 n 和 n 个整数 $a[i], m[i]$, 求满足方程 $x \equiv a[i] (\text{mod } m[i]) (0 \leq i < n)$ 的最小正整数解。

例如求解: 满足 $n = 3, a[0] = 2, m[0] = 3, a[1] = 3, m[1] = 5, a[2] = 2, m[2] = 7$. 即:
 $n \% 3 = 2 \rightarrow 5 * 7 * a \% 3 = 1 \rightarrow a = 2, a[0] = 2 \therefore a = 4$ 得 $5 * 7 * 4$
 $n \% 5 = 3 \rightarrow 3 * 7 * b \% 5 = 1 \rightarrow b = 1, a[1] = 3 \therefore b = 3$ 得 $3 * 7 * 3$
 $n \% 7 = 2 \rightarrow 3 * 5 * c \% 7 = 1 \rightarrow c = 1, a[2] = 2 \therefore c = 2$ 得 $3 * 5 * 2$
累加得: $5 * 7 * 4 + 3 * 7 * 3 + 3 * 5 * 2 = 233$
取余得: $233 \% (3 * 5 * 7) = 23$, 所以最终答案就是 23

5.9.1 模数互素

```

1 //前提条件: m[i] > 0 且, m[i] 中任意两数互质
2 ll CRT( ll a[], ll m[], ll n)
3 { // Chinese Remainder Theorem
4     ll M = 1, ans = 0;
5     for( int i = 0; i < n; i++) { M *= m[i]; }
6     for( int i = 0; i < n; i++){
7         ll x, y, e;
8         e = M / m[i];
9         ex_gcd(e, m[i], x, y);
10        ans = (ans + e * x % M * a[i] % M) % M;
11    }
12    return (ans + M) % M; //最小正整数解
}

```

5.9.2 模数不互素

当 $m[i]$ 不满足两两互素时, 假设 $x \equiv a1(\text{mod } m1) \dots (1), x \equiv a2(\text{mod } m2) \dots (2)$.
令 $d = \gcd(m1, m2)$. 由 (1)(2) 得: $x = a1 + m1 * k1, x = a2 + m2 * k2$. 合并可得: $m1 * k1 = (a2 - a1) + m2 * k2$
两边同除以 d 得: $m1 / d * k1 = (a2 - a1) / d + m2 / d * k2$.
也就是: $m1 * k1 / d (a2 - a1) / d (\text{mod } m2 / d)$.
也就是: $m1 * k1 = (a2 - a1) (\text{mod } m2)$.

易知 k_1 有多个解, 假设 k' 为 k_1 的最小非负整数解则: $k_1 \equiv k' (\text{mod } m_2/d)$

即: $k_1 = k' + (m_2/d) * C$ (C 为某一整数). 将其带入 $x = a_1 + m_1 * k_1$

可得: $x = a_1 + m_1 * (k' + m_2/d * C)$

也就是: $x = a_1 + m_1 * k' + m_1 * m_2/d * C$.

也就是: $x = (a_1 + m_1 * k') (\text{mod } m_1 * m_2/d)$

也就是: $x = (a_1 + m_1 * k') (\text{mod } \text{lcm}(m_1, m_2)) \dots (3)$;

那么求解出 k' 后, 就可以将方程 (1)(2) 合并为一个方程 (3) 了, 依次迭代就能出解了。

[HDU 1573 X 问题]: 求解在 $(0, N]$ 区间满足 $x \equiv a[i] (\text{mod } m[i]) (0 \leq i < M)$ 的 x 的个数。

利用中国剩余定理求解出最小非负整数解 a_0 (如果有解) 和解的周期 m_0 , 假设解的个数为 k 个则: $a_0 + k * m_0 \leq N \rightarrow k \leq (N - a_0)/m_0$ ($N \geq a_0$)。当 $a_0 = 0$ 时解的个数是 $(N - a_0)/m_0$. 当 $a_0 > 0$ 时解的个数是 $(N - a_0)/m_0 + 1$.

```

1  typedef long long ll;
2  const int MAX_N = 15;
3
4  int T, M;
5  ll N, a0, m0, a[MAX_N], m[MAX_N];
6
7  ll ex_gcd(ll aa, ll bb, ll& xx, ll& yy)
8  {
9      if(bb == 0) {
10          xx = 1, yy = 0;
11          return aa;
12      }
13      ll dd = ex_gcd(bb, aa % bb, yy, xx);
14      yy -= aa / bb * xx;
15      return dd;
16  }
17
18 //求解: m0 * x = (aa - a0) (mod mm)
19 bool ModularLinearEquation(ll& m0, ll& a0, ll mm, ll aa)
20 {
21     ll x, y, d;
22     d = ex_gcd(m0, mm, x, y);
23     if(labs(aa - a0) % d != 0) return false;
24     mm /= d;
25     x = x * (aa - a0) / d % mm;
26     a0 += x * m0;
27     m0 *= mm;
28     a0 = (a0 % m0 + m0) % m0;
29     return true;
30 }
31
32 bool CRT(ll& m0, ll& a0)
33 {
34     bool flag = true;
35     m0 = 1, a0 = 0; //任意数 mod m0 = a0 恒成立
36     for(int i = 0; i < M; i++){
37         if(ModularLinearEquation(m0, a0, m[i], a[i]) == false){
38             flag = false;
39             break;
40         }
41     }
42     return flag;
43 }
44
45 int main()
46 {
47     scanf("%d", &T);
48     while(T--){
49         scanf("%lld%d", &N, &M);
50         for(int i = 0; i < M; i++){
51             scanf("%lld", &m[i]);
52         }
53     }
54 }
```

```

53     for (int i = 0; i < M; i++){
54         scanf("%lld", &a[i]);
55     }
56     if (CRT(m0, a0) == false || N < a0) printf("0\n");
57     else {
58         //printf("a0 = %lld m0 = %lld\n", a0, m0);
59         printf("%lld\n", (N - a0) / m0 + (a0 == 0 ? 0 : 1));
60     }
61 }
62 return 0;
63 }
```

5.10 法雷级数

5.10.1 介绍

真分数：若 p, q 是正整数， $0 < \frac{p}{q} < 1$ ，则称 $\frac{p}{q}$ 为真分数。
定理：若 $\frac{a}{d}, \frac{c}{d}$ 是最简真分数（也可以是 $\frac{0}{1}, \frac{1}{1}$ ），且 $\frac{a}{b} < \frac{c}{d}$ 则有：

- 数 $\frac{a+c}{b+d}$ 是一个最简真分数
- $\frac{a}{b} < \frac{a+c}{b+d} < \frac{c}{d}$

5.10.2 n 级法雷数列

对于任意给定的自然数 n ，将分母小于等于 n 的不可约的真分数按升序排列，并且在第一个分数之前加上 $\frac{0}{1}$ ，在最后一个分数之后加上 $\frac{1}{1}$ ，这个序列称为 n 级法雷数列，用 F_n 表示。例如：
 $F_5: \frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}$.

5.10.3 法雷数列的构造

应用上面的定理，如果 $\frac{a}{b}, \frac{c}{d}$ 是一个法雷数列，则在它们中间可以插入 $\frac{(a+c)}{(b+d)}$ ，这样一直二分构造，直到不能构造为止（分母大于 n ）。例如 F_5 的构造：

step1：在 $\frac{0}{1}$ 和 $\frac{1}{1}$ 之间插入 $\frac{1}{2}$ 可得： $\frac{0}{1}, \frac{1}{2}, \frac{1}{1}$
 step2：在每对相邻两个数之间插入 1 个数得： $\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}$
 step3：重复上述操作 $\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}$
 step4：重复上述操作需保证分母不大于 5 $\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}$
 构造结束。

5.10.4 求 n 级法雷级数个数

设 F_n 的个数为 $f[n]$ 个，则 F_n 比 $F(n-1)$ 增加的分母是 n ，所以增加的个数是分子比 n 小且与 n 互质的数个数，这就是欧拉函数 $\phi[n]$ ！

递推式： $f[n] = f[n-1] + \phi[n]$ 。所以有 $f[n] = 1 + \phi[1] + \phi[2] + \dots + \phi[n]$ 。

5.10.5 性质

- 因为 $n \geq 3$ 时，欧拉函数 $\phi[n]$ 是偶数，由此可得：除 $f[1] = 2$ 是偶数外，法雷级数其他各级的个数都是奇数。
- 如果 $\frac{a}{b}, \frac{c}{d}$ 是相邻的两项，则 $abs(a * d - b * c) = 1$
- 如果 $\frac{a}{b}, \frac{c}{d}, \frac{e}{f}$ 是相邻的三项，则 $\frac{a+c}{b+d} = \frac{c}{d}$ 。

```

1 // 求 MAX_N 以内每个数的法雷数
2 int f[MAX_N];
3 void GetFarey()
4 {
5     GetEuler(); //先筛得欧拉表
```

```

6     f[1] = 2;
7     for (int i = 2; i < MAX_N; i++){
8         f[i] = f[i - 1] + phi[i];
9     }
10    for (int i = 1; i < 10; i++){
11        printf("%d ", f[i]);
12    }
13    printf("\n");
14}

```

可以边筛素数边计算欧拉函数

```

1 void GetFarey()
2 {
3     memset(phi, 0, sizeof(phi));
4     phi[1] = 1;
5     for (int i = 2; i < MAX_N; i++){
6         if (phi[i] == 0){ // i 同时也是素数
7             for (int j = i; j < MAX_N; j += i){
8                 if (phi[j] == 0) phi[j] = j;
9                 phi[j] = phi[j] / i * (i - 1);
10            }
11        }
12    }
13    f[1] = 2;
14    for (int i = 2; i < MAX_N; i++){
15        f[i] = f[i - 1] + phi[i];
16    }
17}

```

构造 n 级法雷级数

```

1 int farey[MAX_N][2], total;
2 // faery[i][0], farey[i][1] 分别是第 i 个 farey 数列的分子和分母
3 void Make_Farey_Sequence(int a, int b, int c, int d)
4 {
5     if (a + c > n || b + d > n) return ;
6     Make_Farey_Sequence(a, b, a + c, b + d);
7     // 保证了法雷序列的有序性
8     farey[total][0] = a + c;
9     farey[total++][1] = b + d;
10    Make_Farey_Sequence(a + c, b + d, c, d);
11}
12
13 int main()
14 {
15     scanf("%d", &n);
16     farey[0][0] = 0, farey[0][1] = 1;
17     total = 1;
18     Make_Farey_Sequence(0, 1, 1, 1);
19     farey[total][0] = 1;
20     farey[total++][1] = 1;
21     for (int i = 0; i < total; i++){
22         printf("%d/%d\n", farey[i][0], farey[i][1]);
23     }
24     printf("\n");
25}

```

5.11 原根

5.11.1 介绍

定义：设 $m > 1, \gcd(a, m) = 1$, 使得 $a^r \equiv 1 \pmod{m}$ 成立的最小 r , 成为 a 对模 m 的阶。
定义 1：如果 a 模 m 的阶等于 $\phi(m)$, 则称 a 为模 m 的一个原根。

定义 2: 在模运算中, 若存在一个整数 g 使得式子 $g^k \equiv a \pmod{n}$, $1 \leq k < n$ 结果 a 各不相同, 则称 g 为模 n 的一个原根。

例如:

$$\begin{aligned} 3^1 &= 3 = 3^0 * 3 = 1 \times 3 = 3 \equiv 3 \pmod{7} \\ 3^2 &= 9 = 3^1 * 3 = 3 \times 3 = 9 \equiv 2 \pmod{7} \\ 3^3 &= 27 = 3^2 * 3 = 2 \times 3 = 6 \equiv 6 \pmod{7} \\ 3^4 &= 81 = 3^3 * 3 = 6 \times 3 = 18 \equiv 4 \pmod{7} \\ 3^5 &= 243 = 3^4 * 3 = 4 \times 3 = 12 \equiv 5 \pmod{7} \\ 3^6 &= 729 = 3^5 * 3 = 5 \times 3 = 15 \equiv 1 \pmod{7} \end{aligned}$$

所以 3 为模 7 的一个原根。

定理:

- 模 m 有原根的充要条件是: $m = 2, 4, p^a, 2 * p^a$ (p 为奇素数)
- 如果模 m 有原根, 那么一共有 $\phi(\phi(m))$ 个原根。 $\phi(m)$ 为欧拉函数值
- 如果 p 为素数, 那么 p 一定存在原根, 且模 p 的原根的个数为 $\phi(p - 1)$.

5.11.2 求模素数 p 的所有原根

对 $p - 1$ 素因子分解, 即 $p - 1 = p_1^{a_1} * p_2^{a_2} * p_3^{a_3} * \dots * p_k^{a_k}$ 是 $p - 1$ 的标准分解式, 若恒有 $g^{\frac{p-1}{p_i}} \equiv 1 \pmod{p}$ 成立则 g 就是 p 的原根。枚举 $g : 2 \sim p - 1$. 对于合数求原根只需把 $p - 1$ 换成 $\phi(p)$ 即可。

```

1 int factor[MAX_N], factor_cnt;
2 void Factor(int x) // 获得 x 的所有素因数
3 {
4     factor_cnt = 0;
5     int t = (int) sqrt(x + 0.5);
6     for (int i = 0; prime[i] <= t; i++) {
7         if (x % prime[i] == 0) {
8             factor[factor_cnt++] = prime[i];
9             while (x % prime[i] == 0) x /= prime[i];
10        }
11    }
12    if (x > 1) factor[factor_cnt++] = x;
13 }
14
15 ll quick_pow(ll a, ll b, ll m)
16 {
17     ll ans = 1, tmp = a % m;
18     while (b) {
19         if (b & 1) ans = ans * tmp % m;
20         tmp = tmp * tmp % m;
21         b >>= 1;
22     }
23     return ans;
24 }
25
26 int ans[MAX_N];
27 int main()
28 {
29     int p, total;
30     GetPrime();
31     while (~scanf("%d", &p)) {
32         Factor(p - 1);
33         total = 0;
34         for (int g = 2; g < p; g++) {
35             bool flag = true;
36             for (int i = 0; i < factor_cnt; i++) {
37                 int t = (p - 1) / factor[i];
38                 if (quick_pow(g, t, p) == 1) {
39                     flag = false;
40                     break;
41                 }
42             }
43             if (flag) ans[total] = g;
44             total++;
45         }
46     }
47 }
```

```

42     }
43     if( flag){
44         ans[ total++ ] = g;
45     }
46 }
47 for( int i = 0; i < total; i++){
48     printf("%d\n", ans[ i ]);
49 }
50 }
51 }
```

5.12 BSGS 算法

BSGS 算法用于求解: $a^x \equiv b \pmod{p}$ 在已知 $a \in [2, p)$, $b \in [1, p)$, (p 为质数) 的情况下的最小解 x 。
时间复杂度 $O(\sqrt{p})$ 。

令 $x = i * m + j$, 其中 $m = \text{ceil}(\sqrt{p})$ $0 \leq i < m$, $0 \leq j < m$, 那么就相当于求解: $a^{i*m+j} \equiv b \pmod{p}$ --> $a^j = b * a^{-i*m} \pmod{p}$, a^{-i*m} 是 a^{i*m} 模 p 的逆元。所以可以先处理出 $a^j \pmod{p}$ 的答案放入一个 hash 表中【BabyStep】，然后枚举 $i : 0 \sim m$ 【GaintStep】，查找 $b * a^{-i*m} \pmod{p}$ 是否在 hash 表中出现，如果出现，令出现的编号为 id , 则答案就是 $id + i * m$. 在时间允许的情况下，hash 表采用 map 也可以。为什么枚举 $i < m$ 就可以了呢？显然枚举 $i < m$ 就枚举完了 $x < p$ 的所有情况，当 $x \geq p$ 时可以令 $x = k * p + t$ ($t < p$)，则 $a^x \pmod{p} = a^{k*p+t} \pmod{p} = a^{k*p} * a^t \pmod{p} = a^{k*k} \pmod{p} * a^t \pmod{p}$ 。由费马小定理得: $a^p \equiv 1 \pmod{p}$ (p 为素数)，那么 $a^x \pmod{p} = 1^k \pmod{p} * a^t \pmod{p} = a^t \pmod{p}$ 。所以只需要枚举所有 x 小于 p 的情况就可以了，也就是枚举 $i < m$ 就可以了 (m 是向上取整的)。

```

1 const ll MOD = 100007;
2 ll hs[MOD + 100], id[MOD + 100];
3
4 ll find(ll x)
5 {
6     ll t = x % MOD;
7     while(hs[t] != x && hs[t] != -1) t = (t + 1) % MOD;
8     return t;
9 }
10
11 void insert(ll x, ll ii)
12 {
13     ll pos = find(x);
14     if(hs[pos] == -1){
15         hs[pos] = x;
16         id[pos] = ii;
17     }
18 }
19
20 ll get(ll x)
21 {
22     ll pos = find(x);
23     return hs[pos] == x ? id[pos] : -1;
24 }
25
26 ll inv(ll a, ll p) //求解: a * x ≡ 1 (mod p)
27 {
28     ll x, y, d;
29     d = ex_gcd(a, p, x, y); // ax + py = gcd(a, p) 本段代码省略了ex_gcd()
30     return d == 1 ? (x % p + p) % p : -1;
31 }
32
33 ll BSGS(ll a, ll b, ll p)
34 {//求解a^x ≡ b (mod p)
35     memset(hs, -1, sizeof(hs));
36     memset(id, -1, sizeof(id));
37     ll m = (ll)ceil(sqrt(p + 0.5));
38     ll tmp = 1;
39     for(ll i = 0; i < m; ++ i) {
```

```

40     insert(tmp, i);
41     tmp = tmp * a % p;
42 }
43 ll base = inv(tmp, p); //tmp = a ^ m % p
44 ll res = b;
45 for(ll i = 0; i < m; ++ i) {
46     if(get(res) != -1) return i * m + get(res);
47     res = res * base % p;
48 }
49 return -1;
50 }
```

5.12.1 扩展 BSGS($\gcd(a, p) \neq 1$)

初始化 $cnt = 0$ (消因子轮数), $d = 1$ (消掉的 \gcd 乘积). 令 $tmp = \gcd(a, p)$ 当 $tmp! = 1$ 时, 修改变量值: b / tmp (先判断 b 是否是 tmp 的倍数), p / tmp , $d = \frac{a}{tmp} * d \% p$; 通过若干轮消掉 a, p 的因子使得最终 $\gcd(a, p) = 1$. 这时再调用普通的 BSGS 得到解为 res , 则最终答案是 $res + cnt$. 但是这样求得的解是 $\geq cnt$ 的, 需要先判断下是否有 $< cnt$ 的解. 考虑 cnt 的最大值. 因为每次消去的最小因子是 2, 可以得到 cnt 的最大值是 $\log_2 p$, 先跑一遍 50 次遍历的循环是绰绰有余的。

```

1  ll gcd( ll x, ll y)
2  {
3      return y == 0 ? x : gcd(y, x % y);
4  }
5
6  ll solve( ll a, ll b, ll p)
7  {
8      ll tmp = 1;
9      for( int i = 0; i <= 50; ++ i) {
10         if(tmp == b) return i;
11         tmp = tmp * a % p;
12     }
13     ll cnt = 0, d = 1 % p;
14     while((tmp = gcd(a, p)) != 1) {
15         if(b % tmp) return -1;
16         b /= tmp;
17         p /= tmp;
18         d = a / tmp * d % p;
19         cnt++;
20     }
21     b = b * inv(d, p) % p;
22     ll ans = BSGS(a, b, p); // 这里就是调用普通的BSGS
23     if(ans == -1) return -1;
24     else return ans + cnt;
25 }
```

5.13 莫比乌斯反演

5.13.1 积性函数

定义域为 N^+ 的函数 f , 对于任意两个互质的正整数 $a, b : \gcd(a, b) = 1$, 均满足 $f(ab) = f(a) * f(b)$, 则函数 f 被称为积性函数。假如对于任意两个正整数 a, b 均有 $f(ab) = f(a) * f(b)$, 则称 f 为完全积性函数。

欧拉函数时积性函数, 但不是完全积性函数。

积性函数的性质:

- $f(1) = 1$
- 考虑一个大于 1 的正整数 N , 设 $N = \prod p_i^{a_i}$, 其中 p_i 为互不相同的质数, 那么对于一个积性函数 $f, f(N) = f(\prod p_i^{a_i}) = \prod f(p_i^{a_i})$, 如果 f 还满足完全积性, 则 $f(N) = \prod f(p_i)^{a_i}$
- 若 $f(n), g(n)$ 均为积性函数, 则函数 $h(n) = f(n)g(n)$ 也为积性函数。
- 若 $f(n)$ 为积性函数, 则函数 $F(n) = \sum_{d|n} f(d)$ 也是积性函数, 反之亦然。

5.13.2 狄利克雷卷积

对于函数 f, g , 定义它们的卷积为 $(f * g)(n) = \sum_{d|n} f(d)g(\frac{n}{d})$ 。

性质:

- $f*(g*h) = (f*g)*h$
- $f*(g + h) = f*g + f*h$
- $f*g = g*f$
- 两个积性函数的狄利克雷卷积仍是积性函数

5.13.3 莫比乌斯反演公式

$$F(n) = \sum_{d|n} f(d) \rightarrow f(n) = \sum_{d|n} \mu(d) F\left(\frac{n}{d}\right)$$

$$F(n) = \sum_{n|d} f(d) \rightarrow f(n) = \sum_{n|d} \mu\left(\frac{d}{n}\right) F(d)$$

5.13.4 莫比乌斯函数 μ

- $$\mu(d) = \begin{cases} 1 & n = 1 \\ (-1)^k & n = p_1 p_2 \dots p_k (p_i \text{ are all prime numbers}) \\ 0 & \text{other cases} \end{cases}$$
- $\sum_{d|n} \mu(d) = (n == 1 ? 1 : 0)$
- 对任意正整数 n 有: $\sum_{d|n} \frac{\mu(d)}{d} = \frac{\phi(n)}{n}$

设 $f(n) = \sum_{d|n} \phi(d)$, 又有 $\sum_{d|n} \phi(d) = n$, 所以 $f(n) = n$, 根据莫比乌斯反演可得:

$$\phi(n) = \sum_{d|n} \mu(d)f\left(\frac{n}{d}\right) = \sum_{d|n} \frac{\mu(d)n}{d}$$

5.13.5 线性筛求解积性函数

观察线性筛中的步骤，筛掉 n 的同时还得到了它的最小质因数 p ，我们希望知道 p 在 n 中的次数，这样就能利用 $f(n) = f(p^k)f(\frac{n}{p^k})$ 求出 $f(n)$ 。

令 $n = pm$ ，由于 p 是 n 的最小质因子，若 $p^2|n$ ，则 $p|m$ 并且 p 也是 m 的最小质因子，这样在筛的同时记录每个合数最小质因子的次数，就能算出新筛去合数最小质因子的次数。但是这样时不够的，我们需要快速求出 $f(p^k)$ ，这时就要结合 f 函数的性质考虑。

例如欧拉函数 $\phi(p^k) = (p-1)p^{k-1}$ 因此在进行筛时，如果 $p|m$ ，就乘上 p ，否则乘上 $p-1$ 。而对于莫比乌斯函数 μ ，只有当 $k=1$ 时 $\mu(p^k) = -1$ ， $\mu(p^k) = 0$ ，和欧拉函数一样根据 m 是否被 p 整除进行判断。

```

1 void GetMu()
2 {
3     memset(vis, 0, sizeof(vis));
4     mu[1] = 1;
5     prime_cnt = 0;
6     for(int i = 2; i < MAX_N; i++) {
7         if(vis[i] == 0) {
8             prime[prime_cnt++] = i;
9             mu[i] = -1;
10    }
11    for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; j++) {
12        vis[i * prime[j]] = 1;
13        if(i % prime[j]) mu[i * prime[j]] = -mu[i];
14        else{
15            mu[i * prime[j]] = 0;
16            break;
17        }
18    }
19 }
20 }
```

[ZOJ 3435]: $\sum_{i=0}^{i=a} \sum_{j=0}^{j=b} \sum_{k=0}^{k=c} [\gcd(i, j, k) == 1], a, b, c \in [1, 1000000]$

1. 当 $i = j = k = 0$ 时是不成立的。
2. 当 i, j, k 中有两个为 0 时，只有三种情况 $(0, 0, 1), (0, 1, 0), (1, 0, 0)$ 。
3. 当 i, j, k 中有一个为 0 时，相当于求 $\gcd(i, j) = 1, \gcd(i, k) = 1, \gcd(j, k) = 1$ 的对数。
4. 当 i, j, k 均大于 0 时，相当于求 $\gcd(i, j, k) = 1 (i \in [1, a], j \in [1, b], k \in [1, c])$ 的对数。

对于 3.4 两种情况用莫比乌斯反演即可。

```

1 GetMu();
2 int a, b, c;
3 while(~scanf("%d%d%d", &a, &b, &c)){
4     a--, b--, c--;
5     if(a > b) swap(a, b);
6     if(a > c) swap(a, c);
7     if(b > c) swap(b, c);
8     // a <= b <= c
9     ans = 3, tmp;
10    int last, x, y, z;
11    for(int i = 1; i <= b; i = last + 1) { // 注意枚举的范围
12        last = i;
13        x = a / i, y = b / i, z = c / i;
14        if(i <= a){
15            last = min(a / x, b / y);
16            last = min(last, c / z);
17        } else { // 防止出现除以0
18        }
19    }
20 }
```

```

18     last = min(b / y, c / z);
19 }
20 tmp = (11) * x * y * z + (11)x * y + (11)x * z + (11)y * z;
21 ans += tmp * (sum[last] - sum[i - 1]);
22 }
23 printf("%lld\n", ans);

```

$$\gcd(x, y) = p \quad (p \text{ 为质数}, x \in [1, n], y \in [1, m]), \text{ 有序对 } (x, y) \text{ 有多少对? } n, m \in [1, 10^7]$$

(2, 3) 和 (3, 2) 是不同的有序对

定义: $f(d)$ 为满足 $\gcd(x, y) = d$ ($x \in [1, n], y \in [1, m]$) 的 (x, y) 的对数。

定义: $F(d)$ 为满足 $d | \gcd(x, y)$ ($x \in [1, n], y \in [1, m]$) 的 (x, y) 的对数。

那么有: $F(n) = \sum_{d|n} f(d) = \frac{n}{d} * \frac{m}{d}$, 根据第二种形式的莫比乌斯反演有:

$$f(x) = \sum_{x|d} \frac{\mu(d)}{x} F(d) = \sum_{x|d} \mu\left(\frac{d}{x}\right) * \frac{n}{d} * \frac{m}{d}$$

题目要求是求 $\gcd(x, y)$ 为质数, 对于每个质数 p 相当于求 $x \in [1, \frac{n}{p}], y \in [1, \frac{m}{p}]$ 的 $\gcd(x, y) = 1$ 的有序对 (x, y) 的对数。我们枚举每个质数 p , 就有 $ans = \sum_p^{\min(n,m)} (\sum_d^{\min(n,m)} \mu(d) * \frac{n}{pd} * \frac{m}{pd})$, 直接枚举的话会 TLE, 所以继续优化。令 $T = pd$, 那么可得: $ans = \sum_p^{\min(n,m)} (\sum_d^{\min(n,m)} \mu(d) * \frac{n}{T} * \frac{m}{T}) = \sum_{T=1}^{\min(n,m)} \frac{n}{T} * \frac{m}{T} * (\sum_{p|T} \mu(\frac{T}{p}))$. 所以我们可以预处理出所有的 T 对应的 $\sum_{p|T} \mu(\frac{T}{p})$.

设 $sum(x) = \sum_{p|x} \mu(\frac{x}{p})$, 这里 p 为素数, 令 $g(x) = \mu(\frac{x}{p})$. 我们枚举每一个 k , 得到 $g(kx) = \mu(\frac{kx}{p})$, 分情况讨论有:

- $x \% k == 0$,

$$g(kx) = \begin{cases} \mu(x) & k = p \\ 0 & k! = p \end{cases}$$

- $x \% k! = 0$,

$$g(kx) = \begin{cases} \mu(x) & k = p \\ \mu(x) - g(x) & k! = p \end{cases}$$

$\lfloor \frac{N}{d} \rfloor$ 的取值只有 $2\lfloor \sqrt{N} \rfloor$ 种, 同理 $\lfloor \frac{M}{d} \rfloor$ 的取值也只有 $2\lfloor \sqrt{M} \rfloor$ 种, 并且相同取值对应的 d 是一个连续的区间, 因此 $\lfloor \frac{N}{d} \rfloor$ 和 $\lfloor \frac{M}{d} \rfloor$ 都相同的区间最多有 $2\lfloor \sqrt{N} \rfloor + 2\lfloor \sqrt{M} \rfloor$ 个, 这样 d 的枚举量就缩小为 $O(\sqrt{N} + \sqrt{M})$ 了。

```

1 typedef long long ll;
2 const int MAX_N = 10000010;
3
4 bitset<MAX_N> bs;
5 int prime_cnt, prime[MAX_N];
6 ll g[MAX_N], mu[MAX_N], sum[MAX_N];
7 //主函数中调用GetMu()
8 void GetMu()
9 {
10     bs.set();
11     mu[1] = 1;
12     prime_cnt = 0;
13     for(int i = 2; i < MAX_N; ++i) {
14         if(bs[i]) {
15             prime[prime_cnt++] = i;
16             mu[i] = -1;
17             g[i] = 1;
18         }
19         for(int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j) {
20             bs[i * prime[j]] = 0;
21             if(i % prime[j]) {
22                 mu[i * prime[j]] = -mu[i];
23                 g[i * prime[j]] = mu[i] - g[i];
24             } else {

```

```

25         mu[i * prime[j]] = 0;
26         g[i * prime[j]] = mu[i];
27         break;
28     }
29 }
30 for (int i = 1; i < MAX_N; ++i) {
31     sum[i] = sum[i - 1] + g[i];
32 }
33 }
34 }
35
36 inline ll solve(int n, int m)
37 {
38     int top = min(n, m), last;
39     ll ans = 0;
40     for (int i = 1; i <= top; i = last + 1) {
41         last = min(n / (n / i), m / (m / i));
42         ans += (ll)(n / i) * (m / i) * (sum[last] - sum[i - 1]);
43     }
44     return ans;
45 }
```

[BZOJ 2154]: $\sum_{i=1}^{i=n} \sum_{j=1}^{j=m} \text{lcm}(i, j) \% 100000009 \quad (n, m \leq 10^7)$

$$\begin{aligned}
ans &= \sum_{d=1}^{d=n} \sum_{i=1}^{i=n} \sum_{j=1}^{j=m} \frac{i * j}{d} (\gcd(i, j) = d) \\
&= \sum_{d=1}^{d=n} \sum_{i=1}^{\frac{n}{d}} \sum_{j=1}^{\frac{m}{d}} \frac{i * j * d^2}{d} (\gcd(i, j) = 1) \\
&= \sum_{d=1}^n d \sum_{i=1}^{\frac{n}{d}} \sum_{j=1}^{\frac{m}{d}} (i * j) (\gcd(i, j) = 1)
\end{aligned}$$

定义 $f(d) = \sum_{i=1}^a \sum_{j=1}^b (i * j) (\gcd(i, j) = d)$

定义 $F(d) = \sum_{i=1}^{i=a} \sum_{j=1}^{j=b} (i * j) (\gcd(i, j) \% d = 0)$

易得 $F(1) = \text{sum}(a, b) = \frac{a(a+1)}{2} * \frac{b(b+1)}{2}$

反演可得:

$$\begin{aligned}
f(1) &= \sum_{i=1}^{i=a} \sum_{j=1}^{j=b} (i * j) (\gcd(i, j) = 1, a \leq b) \\
&= \sum_{x=1}^a \mu(x) * x^2 \sum_{i=1}^{\frac{a}{x}} \sum_{j=1}^{\frac{b}{x}} (i * j) \\
&= \sum_{x=1}^a \mu(x) * x^2 \sum_{i=1}^{\frac{a}{x}} i * \sum_{j=1}^{\frac{b}{x}} j \\
&= \sum_{x=1}^a \mu(x) * x^2 * \text{sum}\left(\frac{a}{x}, \frac{b}{x}\right)
\end{aligned}$$

$$\therefore ans = \sum_{d=1}^n d \sum_{x=1}^{n'} \mu(x) * x^2 * \text{sum}\left(\frac{n'}{x}, \frac{m'}{x}\right) \quad (n' = \frac{n}{d}, m' = \frac{m}{d})$$

此时如果预处理出 $\mu(x) * x^2$ 的前缀和求 ans 的复杂度是 $O(\sqrt{n} * \sqrt{n}) = O(n)$, 对于本题而言是不够的。令 $T = d * x$ 可得: $ans = \sum_{T=1}^n \text{sum}\left(\frac{n}{T}, \frac{m}{T}\right) \sum_{x|T} \frac{T}{x} * x^2 * \mu(x)$. 令 $h[T] = \sum_{x|n} \frac{T}{x} * x^2 * \mu(x)$, 预处理出 $h[T]$ 【因为 $h(T)$ 是积性函数可以线性筛】，那么时间复杂度就变为 $O(\sqrt{n})$ 总的时间复杂度为 $O(T\sqrt{n})$ (T 是测试组数)

```

1  typedef long long ll;
2  const int MAX_N = 100000010;
3  const ll mod = 100000009;
4
5  bitset<MAX_N> bs;
6  int prime_cnt, prime[MAX_N / 100 * 7];
7  ll h[MAX_N], sum[MAX_N];
8  // 主函数中调用GetMu()
9  void GetMu()
10 {
11     bs.set();
12     prime_cnt = 0;
13     h[1] = sum[1] = 1;
14     for (int i = 2; i < MAX_N; ++i) {
15         if (bs[i]) {
16             prime[prime_cnt++] = i;
17             h[i] = (ll)i * (1 - i) % mod;
18         }
19         for (int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j) {
20             bs[i * prime[j]] = 0;
21             if (i % prime[j]) { // i 和 prime[j] 互质
22                 h[i * prime[j]] = h[i] * h[prime[j]] % mod;
23             } else {
24                 // 从原始式子  $(T) = \sigma(\mu(d) * d * d * (T/d))$ ，对 T 质因子分解只需要考虑前两项
25                 h[i * prime[j]] = h[i] * prime[j] % mod;
26                 break;
27             }
28         }
29     }
30     for (int i = 1; i < MAX_N; ++i) {
31         sum[i] = ((sum[i - 1] + h[i]) % mod + mod) % mod;
32     }
33 }
34
35 inline ll work(int n, int m)
36 {
37     ll res1 = (ll)n * (n + 1) / 2 % mod;
38     ll res2 = (ll)m * (m + 1) / 2 % mod;
39     return res1 * res2 % mod;
40 }
41
42 inline ll solve(int n, int m)
43 {
44     int top = min(n, m), last;
45     ll res = 0;
46     for (int i = 1; i <= top; i = last + 1) {
47         last = min(n / (n / i), m / (m / i));
48         res = (res + (sum[last] - sum[i - 1] + mod) % mod
49                 * work(n / i, m / i) % mod) % mod;
50     }
51     return res;
52 }
```

5.14 反素数

5.14.1 介绍

定义：对于任何正整数 n ，其约数个数记为 $f(n)$ ，例如 $f(6) = 4$ ，如果某个正整数满足 x ：对任意的正整 i ($0 < i < n$) 数，都有 $f(i) < f(n)$ ，那么称为 n 反素数。

性质：

1)：一个反素数的所有质因子必然是从 2 开始的连续若干个质数，因为反素数是保证约数个数为 x 的这个数 n 尽量小

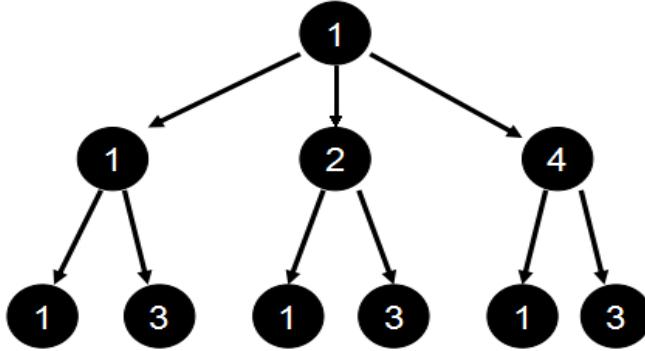
2): 同样的道理, 如果 $n = 2^{p_1} * 3^{p_2} * 5^{p_3} \dots$, 那么必有 $p_1 \geq p_2 \geq p_3 \dots$

5.14.2 求最小的 n 使得其约数个数为 x

由算术基本定理我们知道: 若一个数 $x = p_1^{a_1} p_2^{a_2} \dots p_s^{a_s}$, 那么 x 的约数个数

$$g(x) = \sum_{i=1}^{i=s} \prod (a_i + 1)$$

例如: $12 = 2^2 * 3$, 其约数个数为 6. 建立搜索树:



这棵树除了第一层外, 每一层对应着一个素数, 从上到下递增; 每一层的每一个节点对应着素数的幂, 从左到右递增, 每一条从根节点到叶节点的路径上数字相乘即为一个约数。我们要想获得 $g(n) = x$ 的最小正整数, 就要求 n 的质因子尽可能小, 且尽可能多, 换而言之就是幂次尽可能为 1, 所以就要对上面的树进行从上到下从左到右的 dfs 直到约数个数满足 x 。

【Codeforces 27E】 给定一个数 n , 求一个最小的正整数, 使得的约数个数为 n .

```

1  typedef unsigned long long lint;
2  const lint inf = ~0ull;
3  const int prime[] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53};
4
5  int n;
6  lint ans;
7
8  void dfs(int depth, int limit, lint tmp, int num)
9  {
10     if (num > n) return;
11     if (num == n && tmp < ans) ans = tmp;
12     for (int i = 1; i <= limit; ++i) { // i 相当于幂次
13         if ((double)tmp * prime[depth] > ans) break; // 不用扩展树的深度
14         tmp *= prime[depth];
15         if (n % (num * (i + 1)) == 0) {
16             dfs(depth + 1, i, tmp, num * (i + 1));
17         }
18     }
19 }
20
21 int main()
22 {
23     while (cin >> n){
24         ans = inf;
25         dfs(0, 63, 1, 1);
26         cout << ans << endl;
27     }
28     return 0;
29 }
```

【URAL 1748】: 给定一个数 n , 求 $[1, n]$ 内约数个数最多的且数值最小的数, 以及其约数个数 ($n \leq 10^{18}$)。

¹ //将搜索改为当前值 $tmp > n$ 时终止, 初始化 $ans = inf$, $cnt = 0$

```

2 //主函数里调用 dfs(0, 63, 1, 1) 初始化, prime[] 同例1
3 void dfs(int depth, int limit, lint tmp, int num)
4 {
5     if(tmp > n) return;
6     if(num > cnt || (num == cnt && tmp < ans)) {
7         ans = tmp;
8         cnt = num;
9     }
10    for(int i = 1; i <= limit; ++i) {
11        if((double)tmp * prime[depth] > n) break;
12        //需要用 double 强制类型转换, 否则爆数据
13        tmp *= prime[depth];
14        dfs(depth + 1, i, tmp, num * (i + 1));
15    }
16}
17

```

5.15 卢卡斯定理

用于解决组合数取模问题 $C_n^m \% p$ ($m \leq n \leq 10^{18}$, p 为素数)

当 $1 \leq m \leq n \leq 10^{18}$, $2 \leq p \leq 10^5$ 且 p 是素数时, 如果:

$$\begin{aligned} n &= n_k p^k + n_{k-1} p^{k-1} + \dots + n_1 p + n_0 \\ m &= m_k p^k + m_{k-1} p^{k-1} + \dots + m_1 p + m_0 \end{aligned}$$

那么:

$$C_n^m \% p = \prod_{i=0}^k C_{n_i}^{m_i} \pmod{p}$$

```

1 inline ll C(ll a, ll b)
2 { //计算组合数 C[a][b] , 如果是小数据并且模数固定可以预处理阶乘
3     if(b > a) return 0;
4     ll res = 1, x, y;
5     for(int i = 1; i <= b; ++i) {
6         x = (a + i - b) % mod;
7         y = i % mod;
8         res = res * x % mod * quick_pow(y, mod - 2) % mod; //模素数逆元
9     }
10    return res;
11}
12 inline ll Lucas(ll n, ll m)
13 {
14     if(n < 0 || m < 0 || m > n) return 0;
15     if(m == 0) return 1;
16     return C(n % mod, m % mod) * Lucas(n / mod, m / mod) % mod;
17}

```

5.15.1 给定 n 求 C_n^m ($0 \leq m \leq n \leq 10^8$) 为奇数的 m 个数

[HDU 4349]

即 $C_n^m \% 2 = 1$ 的 m 个数, 考虑将 n 和 m 都表示成 2 的幂次组合形式, 则任意的系数 n_i 和 m_i 非 0 即 1. 因为 $C_0^0 = C_1^0 = C_1^1 = 1$, 所以根据 Lucas 定理, 如果 $n_i = 0$, 则 $m_i = 0$, 如果 $n_i = 1$, 则 $m_i = 0$ 或 1, 根据乘法原理每个 $n_i = 1$, 对于 m_i 都有 2 种选择, 那么 $ans = 2^{cnt}$, 其中 cnt 是 n 分解为 2 的幂次形式系数为 1 的项个数

```

1 scanf("%d", &n);
2 cnt = 0;
3 while (n) {
4     if (n & 1) cnt++;

```

```
5     n >>= 1;  
6 }  
7 printf( "%d\n" , 1 << cnt );
```

5.16 特殊方法

5.16.1 n^k 的高三位

对于给定的一个数 n , 它可以写成 10^a , 其中 a 为浮点数, 则 $n^k = (10^a)^k = 10^{a*k} = 10^x * 10^y$; 其中 x, y 分别是 $a*k$ 的整数部分和小数部分. 对于 $t = n^k$ 这个数, 它的位数由 10^x 决定, 它的位数上的值则有 10^y 决定, 因此我们要求 t 的前三位, 只需要将 10^y 求出, 在乘以 100, 就得到了它的前三位. $fmod(x, 1)$ 可以求出 x 的小数部分。(或者使用 $floor$ 函数)

```

1 high_three_digits = (int)pow(10.0, 2.0 + fmod(k * 1.0 * log10(n * 1.0), 1)).或者
2 double t = 1.0 * k * log10(n * 1.0);
3 high_three_digits = (int)(pow(10.0, t - (int)floor(t) + 2.0));

```

5.16.2 约数个数之和, 定义 $d(i)$ 为 i 的约数个数

$$\sum_{i=1}^a d(i) = \sum \lfloor \frac{a}{i} \rfloor$$

$$\sum_{i=1}^a \sum_{j=1}^b d(i*j) = \sum_{\gcd(i,j)=1} \lfloor \frac{a}{i} \rfloor \lfloor \frac{b}{j} \rfloor$$

$$\sum_{i=1}^a \sum_{j=1}^b \sum_{k=1}^c d(i*j*k) = \sum_{\gcd(i,j)=\gcd(j,k)=\gcd(i,k)=1} \lfloor \frac{a}{i} \rfloor \lfloor \frac{b}{j} \rfloor \lfloor \frac{c}{k} \rfloor$$

这个性质可以推广到 n 维

[BZOJ 3994]: 求 $\sum_{i=1}^n \sum_{j=1}^m d(i*j)$, 定义 $d(i)$ 为 i 的约数个数. $n, m \in [1, 50000]$

$$ans = \sum_{\gcd(i,j)=1} \lfloor \frac{n}{i} \rfloor \lfloor \frac{m}{j} \rfloor = \sum_{i=1}^n \lfloor \frac{n}{i} \rfloor \sum_{j=1}^m \lfloor \frac{m}{j} \rfloor$$

$g(n) = \sum_{i=1}^n \lfloor \frac{n}{i} \rfloor = \sum_{i=1}^n d(i)$, 只需要预处理出 $g(n)$ 就可以在 $O(\sqrt{n})$ 时间范围内解决问题。如果选择分步加速的话, 预处理的复杂度是 $O(n\sqrt{n})$, 但是其实我们考虑每个数 i 的约数个数, 然后 $g(n)$ 就是前缀和了。

在线性筛时每个合数是被最小质因子筛掉的, 我们只需要记录这个每个数 m 最小质因子的幂次 $num[m], d[m]$ 记录 m 的约数个数, 显然 $d(m)$ 是积性函数. 对于 $m = i * prime[j]$, 如果 $i \% prime[j] = 0$, 根据积性函数性质 $num[m] = 1, d[m] = d[i] * d[prime[j]]$, 否则 $num[m] = num[i] + 1$, 因为 m 的约数个数是: $(e_1 + 1) * (e_2 + 1) * \dots * (e_k + 1)$, e_i 是质因子分解后各质因子的幂次, 我们考虑 m 从 i 的转移过程, m 只比 i 在 $prime[j]$ 的幂次上多 1, 所以 $d[m] = \frac{d[i]}{num[i]+1} * (num[i] + 2)$.

```

1 void GetMu() //线性时间预处理
2 {
3     bs.set();
4     prime_cnt = 0;
5     mu[1] = d[1] = 1;
6     for (int i = 2; i < MAX_N; ++i) {
7         if (bs[i]) {
8             prime[prime_cnt++] = i;
9             mu[i] = -1;
10            num[i] = 1;
11            d[i] = 2;
12        }
13        for (int j = 0; j < prime_cnt && i * prime[j] < MAX_N; ++j) {
14            bs[i * prime[j]] = 0;
15            if (i % prime[j]) {
16                mu[i * prime[j]] = -mu[i];
17                num[i * prime[j]] = 1;
18                d[i * prime[j]] = d[i] * d[prime[j]];
19            } else {
20                mu[i * prime[j]] = 0;

```

```

21         num[i * prime[j]] = num[i] + 1;
22         d[i * prime[j]] = d[i] / (num[i] + 1) * (num[i] + 2);
23         break;
24     }
25 }
26 for (int i = 1; i < MAX_N; ++i) {
27     sum[i] = sum[i - 1] + mu[i];
28     g[i] = g[i - 1] + d[i];
29 }
30 }
31 void Get_g() // O(n * sqrt(n)) 的预处理
32 {
33     int last;
34     for (int n = 1; n < MAX_N; ++n) {
35         for (int i = 1; i <= n; i = last + 1) {
36             last = n / (n / i);
37             g[n] += (ll)(last - i + 1) * (n / i);
38         }
39     }
40 }
41 }
42 inline ll solve(int n, int m)
43 {
44     ll res = 0;
45     int top = min(n, m), last;
46     for (int i = 1; i <= top; i = last + 1) {
47         last = min(n / (n / i), m / (m / i));
48         res += (sum[last] - sum[i - 1]) * g[n / i] * g[m / i];
49     }
50     return res;
51 }
52 }
```

对于【Codeforces 235 E Number Challenge】：

$$\sum_{i=1}^a \sum_{j=1}^b \sum_{k=1}^c d(ijk), a, b, c \in [1, 2000]$$

利用上述结论反演可得：

$$Ans = \sum_{i=1}^{i=a} \left\lfloor \frac{a}{i} \right\rfloor \sum_{d=1}^{\min(b,c)} (d) \sum_{d|j, (i,j)=1} \left\lfloor \frac{b}{j} \right\rfloor \sum_{d|k, (i,k)=1} \left\lfloor \frac{c}{k} \right\rfloor$$

记 $j = dj'$, $k = dk'$, 则：

$$Ans = \sum_{i=1}^{i=a} \left\lfloor \frac{a}{i} \right\rfloor \sum_{d=1}^{\min(b,c)} (d) \sum_{gcd(i,dj')=1} \left\lfloor \frac{b}{dj'} \right\rfloor \sum_{gcd(i,dk')=1} \left\lfloor \frac{c}{dk'} \right\rfloor$$

因为 $gcd(i, dj') = 1$, 我们先保证 $gcd(i, d) = 1$, 然后枚举 $j' : 1 \rightarrow \frac{b}{d}$, 保证 $gcd(i, j') = 1$, 这样就可以使得 $gcd(i, dj') = 1$, 累加即可。对于 $gcd(i, dk') = 1$ 同样处理。时间复杂度是: $O(a * b * log(b))$

```

1 //需要预处理 GetMu() 和 GetGcd()
2 inline ll work(int n, int x)
3 {
4     ll res = 0;
5     for (int i = 1; i <= n; ++i) {
6         if (gcd[i][x] == 1) {
7             res = (res + (n / i)) % mod;
8         }
9     }
10    return res;
11 }
```

```

13 inline ll solve(int a, int b, int c)
14 {
15     ll res = 0;
16     int top = min(b, c);
17     for (int i = 1; i <= a; ++i) {
18         for (int d = 1; d <= top; ++d) {
19             if (gcd[i][d] == 1) {
20                 ll tmp = 0;
21                 tmp = ((ll)(a / i) * mu[d] * work(b / d, i))
22                     % mod * work(c / d, i) % mod;
23                 res = ((res + tmp) % mod + mod) % mod;
24             }
25         }
26     }
27     return res;
28 }
```

5.16.3 给定 k , 求最小的 n 使得 n 的约数个数恰为 $n - k$ 个 ($k \leq 47777$)

【HDU 4542】

```

1 const int MAX_N = 50010;
2 int id[MAX_N];
3 void init()
4 {
5     for (int i = 1; i < MAX_N; ++i) id[i] = i;
6     // 初始化 id[i] 最多有 i 个数与其互质
7     for (int i = 1; i < MAX_N; ++i) {
8         for (int j = i; j < MAX_N; j += i) id[j]--;
9         // 根据 j 的约数有, iid[j]--
10        if (id[id[i]] == 0) id[id[i]] = i;
11        // 因为我们最终是要将 id[k] 表示成约数个数为 n-k 的最小数 n ,
12        // 如果此时 id[id[i]]=0 说明小于 i 的数中不存在数 x , 使得 x 的约数个数为 x-id[i]
13        // 那么实际上使得约数个数恰为 n-id[i] 的最小数 n 只能是 i 了
14        id[i] = 0;
15        // 显然小于等于 i 的数不存在数 x 使得 x 的约数个数恰为 x-i 个, 所以要赋 id[i]=0
16    }
17 } // 对于 id[i] 等于 0 的 i , 实际上就不存在数 x 使得 x 的约数个数恰为 x-i 个
```

5.16.4 $C_n^m \bmod p$ ($p = p_1 * p_2$, 且 p_1, p_2 为素数)

我们把 $C_n^m \% p$ 的值记为 x , 把 $C_n^m \% p_1$ 的值记为 x_1 , 把 $C_n^m \% p_2$ 的值记为 x_2 , 则有:

$$x \equiv x_1 (\% p_1) \quad x \equiv x_2 (\% p_2)$$

因为 p_1, p_2 都是素数, 所以 x_1 和 x_2 都可以用 Lucas 定理求解出来。利用中国剩余定理求解同余方程。定义:

inv_1 为: (p_2 在模 p_1 域下的逆元) * p_2
 inv_2 为: (p_1 在模 p_2 域下的逆元) * p_1

```

1 //quick_pow( a, b, c): (a ^ b) % c
2 inv1 = quick_pow(p2, p1 - 2, p1) * p2;
3 inv2 = quick_pow(p1, p2 - 2, p2) * p1;
```

那么答案就是:

$$x = (inv1 * x_1 + inv2 * x_2) \% p$$

Chapter 6

数学相关

6.1 Polya 原理、Burnside 引理

6.1.1 介绍

群:

给定一个集合 $G = \{a, b, c, \dots\}$ 和集合 G 上的二元运算 \star , 并满足

- 封闭性: $\forall a, b \in G, \exists c \in G, a \star b = c.$
- 结合律: $\forall a, b, c \in G, (a \star b) \star c = a \star (b \star c).$
- 单位元: $\exists e \in G, \forall a \in G, a \star e = e \star a = a.$
- 逆元: $\forall a \in G, \exists b \in G, a \star b = b \star a = e, b = a^{-1}$

则称集合 G 在运算 \star 之下是一个群, 简称 G 是群。一般 $a \star b$ 简写为 ab 。

置换:

n 个元素 $1, 2, \dots, n$ 之间的置换 $\begin{pmatrix} 1 & 2 & \cdots & n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$, 表示 1 被 1 到 n 中的某个数 a_1 取代, 2 被 1 到 n 中的某个数 a_2 取代, 直到 n 被 1 到 n 中的某个数 a_n 取代, 且 a_1, a_2, \dots, a_n 互不相同

置换群:

置换群的元素是置换, 运算是置换的连接。例如:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix} \begin{pmatrix} 3 & 1 & 2 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$$

$Z_k(K$ 不动置换类):

设 G 是 $1 \dots n$ 的置换群。若 K 是 $1 \dots n$ 中某个元素, G 中使 K 保持不变的置换的全体, 记以 Z_k , 叫做 G 中使 K 保持不动的置换类, 简称 K 不动置换类。

E_k (等价类):

设 G 是 $1 \dots n$ 的置换群。若 K 是 $1 \dots n$ 中某个元素, K 在 G 作用下的轨迹, 记作 E_k 。即 K 在 G 的作用下所能变化成的所有元素的集合。

$$|E_k| \cdot |Z_k| = |G| \quad k = 1 \dots n$$

$D(a_j)$ 表示在置换 a_j 下不变的元素的个数, s 表示置换种类数:

$$\sum_{i=1}^n |Z_i| = \sum_{i=1}^s D(a_i)$$

6.1.2 Burnside 引理

$$L = \frac{1}{|G|} \sum_{i=1}^n |Z_i| = \frac{1}{|G|} \sum_{i=1}^s D(a_i), L$$
 就是等价类数, 也就是互异的组合状态的个数。

证明: 不妨设 $N = 1 \dots n$ 中共有 L 个等价类, $N = E^1 + E^2 + \dots + E^L$, 则当 j 和 k 属于同一等价类时, 有 $|Z_j| = |Z_k|$ 。所以

$$\sum_{i=1}^n |Z_i| = \sum_{i=1}^L \sum_{k \in E_i} |Z_k| = \sum_{i=1}^L |E_i| \cdot |Z_i| = L \cdot |G|$$

循环:

记 $(a_1 a_2 \dots a_n) = \begin{pmatrix} a_1 & a_2 & \dots & a_{n-1} & a_n \\ a_2 & a_3 & \dots & a_n & a_1 \end{pmatrix}$ 称为 n 阶循环。每个置换都可以写成若干互不相交的循环的乘积，两个循环 $(a_1 a_2 \dots a_n)$ 和 $(b_1 b_2 \dots b_n)$ 互不相交是指 $a_i \neq b_j, i, j = 1, 2, \dots, n.$

例如: $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 4 & 2 \end{pmatrix} = (1\ 3)(2\ 5)(4)$. 这样的表示是唯一的。置换的循环节数是上述表示中循环的个数。

例如 $(1\ 3)(2\ 5)(4)$ 的循环节数是 3。特别的: 当 $n = 2$ 时, 两阶循环 $(i\ j)$ 叫做 i 和 j 的对换。任何一个循环, 都可以表达成若干换位之积。但是表达形式不尽统一, 甚至连换位个数都不相同。

例如 $(123)(12)(13)(13)(23)(21)(23)(12)(13)(31)(13)$ 。尽管如此, 有一个性质却是固有的, 它不依换位的个数不同而异, 那就是循环分解成换位的乘积时, 换位个数奇偶性是不变的, 或分解成奇数个换位之积或分解成偶数个换位之积。

若一个置换分解成奇数个换位之积, 叫做奇置换; 若分解成偶数个换位之积叫偶置换。单位置换为偶置换。

6.1.3 Polya 原理

设 G 是 p 个对象的一个置换群, 用 m 种颜色涂染 p 个对象, $G = \{g_1, g_2, \dots, g_s\}, g_i$ 是不同的置换选择, 令 g_i 的循环节数为 $c(g_i)$ ($i \in [1, s]$), 则不同的染色方案数为: $L = \frac{1}{|G|}(m^{c(g_1)} + m^{c(g_2)} + \dots + m^{c(g_s)})$

6.1.4 $n * n$ 的方阵, 每个小格可涂 m 种颜色, 求在旋转操作下本质不同的解的总数

我们可以在方阵中分出互不重叠的长为 $\lceil \frac{n+1}{2} \rceil$, 宽为 $\lceil \frac{n}{2} \rceil$ 的四个矩阵。当 n 为偶数时, 恰好分完; 当 n 为奇数时, 剩下中心的一个格子, 它在所有的旋转下都不动, 所以它涂任何颜色都对其它格子没有影响。令 m 种颜色为 $0 \sim m - 1$, 我们把矩阵中的每格的颜色所代表的数字顺次(左上角从左到右, 从上到下; 右上角从上到下, 从右到左; ...)排成 m 进制数, 然后就可以表示为一个十进制数, 其取值范围为 $0 \sim m^{\frac{n^2}{4}} - 1$ 。(因为 $\frac{n}{2} * \frac{n+1}{2} = \frac{n^2}{4}$)这样, 我们就把一个方阵简化为 4 个整数。我们只要找到每一个等价类中左上角的数最大的那个方案(如果左上角相同, 就顺时针方向顺次比较)这样, 在枚举的时候其它三个数一定不大于左上角的数, 效率应该是最高的。进一步考虑, 当左上角数为 i 时, ($i \in [0, R - 1]$). 令 $R = m^{\frac{n^2}{4}}$ 可分为下列的 4 类:

1. 其它三个整数均小于 i , 共 i^3 个。
2. 右上角为 i , 其它两个整数均小于 i , 共 i^2 个。
3. 右上角、右下角为 i , 左下角不大于 i , 共 $i + 1$ 个。
4. 右下角为 i , 其它两个整数均小于 i , 且右上角的数不小于左下角的, 共 $\frac{i * (i+1)}{2}$ 个。

因此:

$$\begin{aligned} L &= \sum_{i=0}^{R-1} (i^3 + i^2 + i + 1 + \frac{1}{2}i(i+1)) = \sum_{i=0}^{R-1} (i^3 + \frac{3}{2}i^2 + \frac{3}{2}i + 1) \\ &= \sum_{i=1}^R ((i-1)^3 + \frac{3}{2}(i-1)^2 + \frac{3}{2}(i-1) + 1) = \sum_{i=1}^R (i^3 - \frac{3}{2}i^2 + \frac{3}{2}i) \\ &= \frac{1}{4}R^2(R+1)^2 - \frac{3}{2} * \frac{1}{6}R(R+1)(2R+1) + \frac{3}{2} * \frac{1}{2}R(R+1) \\ &= \frac{1}{4}(R^4 + R^2 + 2R) \end{aligned}$$

当 n 为奇数时, 还要乘以一个 m .

采用 Polya 原理解决

确定置换群:

只有 4 个置换: $0^\circ, 90^\circ, 180^\circ, 270^\circ$, 所以置换群 $G = \{0^\circ, 90^\circ, 180^\circ, 270^\circ\}$

计算循环节个数:

首先, 给每个格子顺次编号 $1 \sim n^2$, 再开一个二维数组记录置换后的状态。最后通过搜索计算每个置换下的循环节个数, 效率为一次方级。

代入公式:

利用 Polya 定理得到最后结果。

$$L = \frac{1}{|G|}(m^{c(g_1)} + m^{c(g_2)} + \dots + m^{c(g_s)})$$

当然也可以直接思考循环节：

- 当 n 为偶数，在转 0° 时，循环节为 n^2 个，转 180° 时，循环节为 $\frac{n^2}{2}$ 个，转 90° 和 270° 时，循环节为 $\frac{n^2}{4}$ 个
- 当 n 为奇数，在转 0° 时，循环节为 n^2 个，转 180° 时，循环节为 $\frac{n^2-1}{2} + 1$ 个，转 90° 和 270° 时，循环节为 $\frac{n^2-1}{4} + 1$ 个

综合考虑可得： $L = \frac{1}{4}(m^{n^2} + m^{\frac{n^2+3}{4}} + m^{\frac{n^2+1}{2}} + m^{\frac{n^2+3}{4}})$ 。可以发现这个式子，其实是和数学推导的式子完全吻合的。

```

1  typedef long long ll;
2  const int MAX_N = 10;
3  const ll mod = 1000000007;
4
5  int a[MAX_N][MAX_N], b[MAX_N][MAX_N];
6  int m, n; // m: 颜色数, n: 方阵大小
7  ll ans;
8
9  void Rotate() //逆时针旋转 90 度
10 {
11     for (int i = 1; i <= n; ++i) {
12         for (int j = 1; j <= n; ++j) {
13             a[n - j + 1][i] = b[i][j];
14         }
15     }
16     for (int i = 1; i <= n; ++i) {
17         for (int j = 1; j <= n; ++j) {
18             b[i][j] = a[i][j];
19         }
20     }
21 }
22
23 void CircleSection() //计算当前状态循环节数
24 {
25     int num = 0; //记录循环节个数
26     for (int i = 1; i <= n; ++i) {
27         for (int j = 1; j <= n; ++j) {
28             if (a[i][j] > 0) { //搜索尚未被访问过的格子
29                 num++;
30                 int nexti, nextj, p;
31                 p = a[i][j];
32                 a[i][j] = 0;
33                 nexti = (p - 1) / n + 1;
34                 nextj = (p - 1) % n + 1; //得到这个循环的下一个格子
35                 while (a[nexti][nextj] > 0) {
36                     p = a[nexti][nextj];
37                     a[nexti][nextj] = 0; //已访问格子置零
38                     nexti = (p - 1) / n + 1;
39                     nextj = (p - 1) % n + 1;
40                 }
41             }
42         }
43     }
44     ll res = 1;
45     for (int i = 1; i <= num; ++i) {
46         res = res * m % mod;
47     }
48     ans = (ans + res) % mod;
49 }
50
51 void init()
52 {
53     for (int i = 1; i <= n; ++i) {

```

```

54     for( int j = 1; j <= n; ++j ) {
55         b[ i ][ j ] = a[ i ][ j ] = ( i - 1 ) * n + j ;
56     }
57 }
58
59 int main()
60 {
61     ll tt = quick_pow( 4, mod - 2 ); //求除以 4 模 mod 的逆元
62     while(~scanf("%d%d", &n, &m)){
63         ans = 0;
64         init(); //对方阵状态进行初始化
65         CircleSection(); //旋转 0 度状态下的循环节个数
66         Rotate();
67         CircleSection(); //逆时针 90 度
68         Rotate();
69         CircleSection(); //逆时针 180 度
70         Rotate();
71         CircleSection(); //逆时针 270 度
72         Rotate();
73         ans = ans * tt % mod;
74         printf("ans = %lld\n", ans); //根据 Polya 原理得到的结果
75
76
77         ll R = quick_pow(m, n * n / 4);
78         ll res = R * R % mod * R % mod * R % mod + R * R % mod + 2 * R % mod ;
79         if(n & 1) res = res * m % mod;
80         res = res * tt % mod;
81         printf("res = %lld\n", res); //根据数学推导得到的结果
82     }
83     return 0;
84 }
```

6.1.5 各有 a, b, c ($a, b, c \geq 0, a + b + c \leq 40$) 颗三种颜色，问这些珠子能串成的项链有多少种？考虑翻转和旋转。

[UVA 11255 Necklace]

令 $\sum_{i=1}^3 color[i] = n$, 即珠子总数。

考虑旋转置换

我们考虑旋转 i 颗珠子的间距，则 $0, i, 2i, \dots$ 构成一个循环，这个循环有 $\frac{n}{\gcd(n,i)}$ 。根据对称性，所有循环的长度均相同，因此一共有 $\gcd(i,n)$ 个循环。循环与循环之间是等价类，所以在一个循环内的某颗珠子的颜色确定了，那么在其余循环内的同样地位位置的珠子颜色也就确定了。我们得到的答案是每个循环的方案数并且在旋转置换下，当循环节确定了，每个循环的方案数都是一样的。我们假设循环节的长度（也就是循环内元素的个数）为 $x \in [0, n)$ ，第 i 种颜色的珠子的个数为 $color[i]$ 个，如果 $color[i] \% x! = 0$ ，那么第 i 种颜色的珠子在每个循环（等价类）内就不能均分，所以这种循环节就应该摒弃。当 $color[i] \% x == 0$ 时，令 $b[i] = \frac{color[i]}{x}$ ，则 $b[i]$ 就是每个循环（等价类）分得的这种颜色的珠子的个数。显然每个循环（等价类）内共有 $\sum_{i=1}^3 b[i] = \frac{1}{x} \sum_{i=1}^3 color[i]$ 颗珠子，记为 sum 。我们考虑单个循环（等价类）：有 sum 颗珠子，各个颜色分别有 $b[i]$ 颗，那么根据排列组合在循环节为 x 时，可得

$$Ans[x] = C_{sum}^{b[0]} * C_{sum-b[0]}^{b[1]} * C_{sum-b[0]-b[1]}^{b[2]}$$

考虑翻转置换

当 n 为奇数时，则对称轴上必有一点，对称轴有 n 条，每条对称轴形成 $\frac{n-1}{2}$ 个长度为 2 的循环和 1 个长度为 1 的循环。

当 n 为偶数时，有两种对称轴。穿过珠子的对称轴有 $\frac{n}{2}$ 条，各形成 $\frac{n}{2} - 1$ 个长度为 2 的循环和两个长度为 1 的循环。不穿过珠子的对称轴也有 $\frac{n}{2}$ 条，各形成 $\frac{n}{2}$ 个长度 2 的循环。

可以发现不论 n 为奇偶，对称轴的总个数都为 n ，同时旋转置换的个数也为 n ，所以根据 Burnside 引理最后需要除以 $2n$ 。

```

1 const int MAX_N = 42;
2
3 ll C[MAX_N][MAX_N];
4 int color[5], b[5], n;
5
6 void init() //组合数打表
7 {
8     C[0][0] = 1;
9     for(int i = 1; i < MAX_N; ++i) {
10         C[i][0] = C[i][i] = 1;
11         for(int j = 1; j < i; ++j) {
12             C[i][j] = C[i - 1][j] + C[i - 1][j - 1];
13         }
14     }
15 }
16
17 int gcd(int x, int y)
18 {
19     return y == 0 ? x : gcd(y, x % y);
20 }
21
22 ll CircleSection(int x)
23 {
24     int sum = 0;
25     for(int i = 1; i <= 3; ++i) {
26         if(b[i] % x) return 0;
27         b[i] /= x;
28         sum += b[i];
29     }
30     ll res = 1;
31     for(int i = 1; i <= 3; ++i) {
32         res *= C[sum][b[i]];
33         sum -= b[i];
34     }
35     return res;
36 }
37
38 ll Rotate() //旋转置换
39 {
40     ll res = 0;
41     for(int i = 0; i < n; ++i) {
42         int d = gcd(i, n);
43         memcpy(b, color, sizeof(color));
44         res += CircleSection(n / d); // n/d 是循环元素个数
45     }
46     return res;
47 }
48
49 ll Flip() //翻转置换
50 {
51     ll res = 0;
52     if(n & 1) {
53         for(int i = 1; i <= 3; ++i) {
54             memcpy(b, color, sizeof(color));
55             b[i]--;
56             if(b[i] < 0) continue;
57             res += CircleSection(2) * n; // n 条对称轴
58         }
59     } else {
60         //穿过珠子
61         for(int i = 1; i <= 3; ++i) {
62             for(int j = 1; j <= 3; ++j) {
63                 memcpy(b, color, sizeof(color));

```

```

64         b[i]--, b[j]--;
65         if(b[i] < 0 || b[j] < 0) continue;
66         res += CircleSection(2) * (n / 2);
67     }
68 //不穿过珠子
69     memcpy(b, color, sizeof(color));
70     res += CircleSection(2) * (n / 2);
71 }
72 return res;
73 }

74 int main()
75 {
76     init();
77     int T;
78     scanf("%d", &T);
79     while(T--) {
80         n = 0;
81         for(int i = 1; i <= 3; ++i) {
82             scanf("%d", color + i);
83             n += color[i];
84         }
85         ll ans = 0;
86         ans += Rotate();
87         ans += Flip();
88         printf("%lld\n", ans / (2 * n));
89     }
90     return 0;
91 }
92 }
```

6.1.6 给出 12 根等长的火柴棒，每根火柴棒的颜色属于 1 – 6 中的一种，问能拼成多少种不同的正方体？（考虑旋转）

[UVA 10601 Cubes] 首先正方体的旋转置换有 24 种。下面将每个循环内元素的个数称为循环的长度。

注意是棱边的置换循环，而不是面的置换循环。

1. 静止。只有一种置换。有 12 个循环，每个循环的长度为 1。
2. 以相对面的中心为轴旋转。可以旋转的角度是 $90^\circ, 180^\circ, 270^\circ$ 。选择轴有 3 种，每种轴下有 3 种旋转，所以有 $3 \times 3 = 9$ 种置换。
3. 旋转 $90^\circ, 270^\circ$ ，都是三个循环，每个循环的长度为 4。
4. 旋转 180° ，有 6 个循环，每个循环的长度为 2
5. 以对边中点为轴，只可以旋转 180° ，选择轴有 6 种，所以有 $6 \times 1 = 6$ 种置换。在这种旋转下有 5 个长度为 2 的循环和 2 个长度为 1 的循环
6. 以对顶点为轴，可以旋转 $120^\circ, 240^\circ$ 。选择轴有 4 种，所以有 $4 \times 2 = 8$ 种置换。
7. 旋转 $120^\circ, 240^\circ$ 都是有 4 个长度为 3 的循环。

找到了在每种置换下的循环个数和每个循环的长度就可以参考前面【例 2】处理旋转置换的方法解决。
需要特别关注的是，以对边中点为轴，只可以旋转 180° ，有 5 个长度为 2 的循环和 2 个长度为 1 的循环。
我们可以先枚举两个长度为 1 的循环选择的颜色，然后对于 5 个长度为 2 的循环就和上面的一样了。最后别忘了根据 Burnside 引理需要除以总的置换数 24。

```

1 11 work(int k)
2 { //每 k 条边必须相同，分成  $12/k$  组以对边中点为轴旋转（ $180^\circ$  是分成 5 组）
3     memcpy(b, a, sizeof(a));
4     int sum = 0;
5     for (int i = 1; i <= 6; ++i) {
```

```

6     if (b[i] % k) return 0;
7     b[i] /= k;
8     sum += b[i];
9 }
10 ll res = 1;
11 for (int i = 1; i <= 6; ++i) {
12     res *= C[sum][b[i]];
13     sum -= b[i];
14 }
15 return res;
16 }
17
18 ll solve()
19 {
20     ll res = 0;
21     // 静止
22     res += work(1);
23     // 以相对面中心为轴
24     res += (ll)3 * 2 * work(4); // 旋转 90° 和 270°
25     res += (ll)3 * work(2); // 旋转 180°
26     // 以对顶点为轴可以旋转, 120° 或 240°
27     res += (ll)4 * 2 * work(3);
28     // 以对边中点为轴, 只能旋转 180°
29     for (int i = 1; i <= 6; ++i) {
30         for (int j = 1; j <= 6; ++j) {
31             if (a[i] == 0 || a[j] == 0) continue;
32             a[i]--; a[j]--; // 将 a[i] 和 a[j] 设为选择的两条对边的颜色
33             res += (ll)6 * work(2);
34             // 剩下的是 5 个循环长度为 2 的循环, 6 代表对边选择情况
35             a[i]++; a[j]++;
36         }
37     }
38     return res / 24;
39 }
```

6.1.7 [POJ 2888 Magic Bracelet]

有一串 n 个珠子的项链，用 m 种颜色来染，有 k 个限制条件： $a[i]$ 和 $b[i]$ 不能相邻。问本质不同的项链有多少种？（考虑旋转，答案对 9973 取模，且 $\gcd(n, 9973) = 1$ ）。数据范围： $n \leq 10^9, 1 \leq m \leq 10, 0 \leq k \leq \frac{m*(m-1)}{2}$

假设有 k 个循环，用 $link[i][j]$ 表示第 i 种颜色能否和第 j 种颜色相邻：当 $link[i][j] = 1$ ，表示 i 和 j 不能相邻，否则可以相邻。无向边： $link[i][j] = link[j][i]$ 。

用 $dp[k][i][j]$ 表示经过 k 个循环从第 i 种颜色转移到第 j 种颜色的方案数：

$$dp[k][i][j] = \sum_{p \leq m} (1 - link[p][j]) * dp[k-1][i][p]$$

初始化： $dp[1][i][j] = 1 - link[i][j]$

初始矩阵： $A[i][j] = 1 - link[i][j]$ 来表示颜色间的连通性

由上面的递推式， $A^k[i][j]$ 代表的是：从第 i 种颜色经过 k 个循环后变为第 j 种颜色的方法数。考虑循环，需要知道从 i 出发回到 i 的方案数即： $A^k[i][i]$ 。 $solve(k)$ 表示循环个数为 k 时的方案数：

$$solve(k) = \sum_{i \leq m} A^k[i][i]$$

离散数学里有：如果用 0,1 矩阵 A 来表示无向图的连通情况的话， A^k 代表的就是一个点经过 k 条路后能到达的地方的方法数。

假设对于每个循环的步长为 i ，也就是 $0, i, 2i, \dots$ 构成一个循环。这个循环的周期为 $\frac{i*n}{\gcd(i,n)}$ ，所以这个循环有 $\frac{n}{\gcd(i,n)}$ 个元素，共有 $\gcd(i,n)$ 个循环。所以枚举的循环个数一定是 n 的因子，即： $k | n$ 。

满足循环个数为 k 的置换的旋转步长 i 满足 $\gcd(i,n) = k$ ，此种置换的个数也就是 $\gcd(\frac{i}{k}, \frac{n}{k}) = 1$ 的 i 的个

数, 即: $\phi(\frac{n}{k})$.

综上对于每个满足 $n \% k = 0$ 的 k 可以得到的方案数是

$$solve(k) * \phi\left(\frac{n}{k}\right)$$

枚举每个 k 然后求和最后还要除以 n (因为总的置换数为 n), 又因为有模数且模数为素数, 那么就相当于乘以 n^{mod-2} (费马小定理)。

$$Ans = \left\{ \sum_{i|n} solve(i) * \phi\left(\frac{n}{i}\right) \right\} * n^{p-2} \% p$$

```

1 const ll mod = 9973;
2
3 int link[15][15];
4
5 struct Matrix{
6     int row, col;
7     ll data[15][15];
8
9     Matrix operator * (const Matrix& rhs) const { // 矩阵相乘条件: col = rhs.row
10        Matrix res;
11        res.row = row, res.col = rhs.col;
12        for(int i = 1; i <= res.row; ++i) {
13            for(int j = 1; j <= res.col; ++j) {
14                res.data[i][j] = 0;
15                for(int k = 1; k <= row; ++k) {
16                    res.data[i][j] += data[i][k] * rhs.data[k][j];
17                }
18                res.data[i][j] %= mod;
19            }
20        }
21        return res;
22    }
23
24    Matrix operator ^ (const int m) const { // 矩阵快速幂
25        Matrix res, tmp;
26        res.row = row, res.col = col; //row == col
27        memset(res.data, 0, sizeof(res.data));
28
29        tmp.row = row, tmp.col = col;
30        memcpy(tmp.data, data, sizeof(data));
31        for(int i = 1; i <= res.row; ++i) { res.data[i][i] = 1; }
32        int mm = m;
33        while(mm) {
34            if(mm & 1) res = res * tmp;
35            tmp = tmp * tmp;
36            mm >>= 1;
37        }
38        return res;
39    }
40};
41
42 inline ll solve(int len, int m)
43 {
44     Matrix tmp;
45     tmp.row = tmp.col = m;
46     for(int i = 1; i <= m; ++i) {
47         for(int j = 1; j <= m; ++j) {
48             tmp.data[i][j] = 1 - link[i][j];
49         }
50     }
51     tmp = tmp ^ len;
52
53     ll ans = 0;
54     for(int i = 1; i <= m; ++i) {

```

```

55     ans = (ans + tmp.data[i][i]) % mod;
56 }
57 return ans;
58 }

59 int main()
60 {
61     int T, n, m, t;
62     scanf("%d", &T);
63     while (T--) {
64         memset(link, 0, sizeof(link));
65         scanf("%d%d%d", &n, &m, &t);
66         for (int i = 0; i < t; ++i) {
67             int former, later;
68             scanf("%d%d", &former, &later);
69             link[former][later] = link[later][former] = 1;
70         }
71     }
72
73     ll ans = 0;
74     for (int i = 1; i * i <= n; ++i) {
75         if (n % i) continue;
76         ans = (ans + phi(n / i) * solve(i, m)) % mod;
77         if (n / i == i) continue;
78         ans = (ans + phi(i) * solve(n / i, m)) % mod;
79     }
80     printf("%lld\n", ans * quick_pow(n % mod, mod - 2, mod) % mod);
81 }
82 return 0;
83 }
```

如果仅仅限制相邻的珠子颜色不能一样，总共有 m 种颜色。对于步长为 k 的循环，定义 $dp[i][0]$ 表示经历 i 个循环珠子颜色和循环起始珠子颜色一致的方案数，定义 $dp[i][1]$ 表示颜色不一样（初始化 $dp[1][0] = 1, dp[1][1] = 0$ ），那么可以得到转移方程：

$$\begin{aligned} dp[i][0] &= dp[i-1][1] \\ dp[i][1] &= dp[i-1][1] * (m-2) + dp[i][0] * (m-1) \end{aligned}$$

如果 m 很大的话，就用矩阵快速幂进行加速。定义矩阵： $A = \begin{pmatrix} 0 & 1 \\ m-1 & m-2 \end{pmatrix}, B = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, C = A^k * B$ 。
那么步长为 k 的循环总的方案数（考虑起始珠子有 m 种颜色可选）是：

$$\phi\left(\frac{n}{k}\right) * m * C[1][1]$$

条件是 $n \% k = 0$ ，最后还要乘上 n 模 mod 的逆元。

6.2 容斥原理

6.2.1 求区间 $[A, B]$ 中和 n 互素的数个数? $1 \leq A \leq B \leq 10^{15}, 1 \leq n \leq 10^9$

[HDU 4135]

我们将 n 进行素因子分解为 $p_1, p_2, p_3, \dots, p_k$, 先求解 $[1, r]$ 中和 n 不互素的数字个数。我们只需要考虑最大公约数是素因子的倍数的情况。选择素因子 num 个相乘, 得到 mul , 计算 $[1, r]$ 中和 n 不互素的数字个数是根据 num 的奇偶性: 奇加偶减, $\frac{r}{mul}$ 。

有位运算和 dfs 两种写法。

```

1  vector<ll> fac;
2
3  void GetFactor(int n)
4  {
5      fac.clear();
6      for(ll i = 2; i * i <= n; ++i) {
7          if(n % i == 0) {
8              fac.push_back(i);
9              while(n % i == 0) n /= i;
10         }
11     }
12     if(n > 1) fac.push_back(n);
13 }
14 /*
15 ll solve(ll A, int n)
16 {
17     ll ans = 0;
18     int total = fac.size();
19     for(ll i = 1; i < (1 << total); ++i) { // 用二进制位表示该位上对应编号的素因子是否选择
20         int bits = 0;
21         ll res = 1;
22         for(int j = 0; j < total; ++j) {
23             if(i & (1 << j)) { // 如果选择了第 j 个素因子
24                 bits++;
25                 res *= fac[j];
26             }
27         }
28         if(bits & 1) ans += A / res; // 选择的素因子个数为奇数个
29         else ans -= A / res;
30     }
31     return A - ans;
32 }
33 */
34 ll ans;
35
36 void dfs(int cur, int num, ll mul, ll A)
37 {
38     if(cur == fac.size()) {
39         if(num & 1) ans -= A / mul;
40         // A / mul 是最大公约数是 mul 的数字个数, ans 求的是互素数个数
41         else ans += A / mul;
42         return;
43     }
44     dfs(cur + 1, num, mul, A);
45     dfs(cur + 1, num + 1, mul * fac[cur], A);
46 }
47
48 int main()
49 {
50     int T, n, cases = 0;
51     ll A, B;
52     scanf("%d", &T);
53     while(T--) {
54         scanf("%lld%lld%d", &A, &B, &n);
55         GetFactor(n);
56     }
57 }
```

```

56     ans = 0;
57     dfs(0, 0, 1, B);
58     ll res = ans;
59     ans = 0;
60     dfs(0, 0, 1, A - 1);
61     printf("Case #%d: %lld\n", ++cases, res - ans);
62     //printf("Case #%d: %lld\n", ++cases, solve(B, n) - solve(A - 1, n));
63 }
64 return 0;
65 }
```

**6.2.2 给一个 n , 在 $p \in [1, n]$ 范围满足 $m^k = p$ ($m \geq 1, k > 1$) 的数字 p 的个数。
 $1 \leq n \leq 10^{18}$**

[HDU 2204] 我们可以枚举幂次 k , 考虑到 $2^{60} > 10^{18}$, 最多只需要枚举到 60 幂次。

同时对于一个数 p 的幂次 k 是个合数, 那么 k 一定可以表示成 $k = r * k'$, 其中 k' 是素数的形式, 那么:

$$p = m^k = m^{r*k'} = (m^r)^{k'}$$

所以我们只需要枚举素幂次 k 即可。

同时如果 $p^k \leq n$, 那么对于任意的 $p' < p$, 也一定满足 $p'^k \leq n$ 。所以对于每个 k 我们令 $p^k = n$, 即 $p = n^{\frac{1}{k}}$, 求出最大的 p , 同时也就是满足 $p^k \leq n$ 的所有 p 的个数。但是这样子会有重复。例如: $k = 2$ 时, 2^{2^3} 和 $k = 3$ 时, 2^{3^2} 就重复计数了。这时候需要用容斥原理: 加上奇数个素幂次相乘的个数, 减去偶数个素幂次相乘的个数。

又因为 $2 * 3 * 5 < 60, 2 * 3 * 5 * 7 > 60$, 那么最多只要考虑三个素幂次相乘情况。

时间复杂度: $O(3 * 2^{17})$ (60 以内共 17 个素数)

```

1 const ll prime[20] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59};
2 const int len = 17;
3 const double eps = 1e-8;
4
5 ll ans;
6 double n;
7
8 void dfs(int cur, int num, int total, ll k)
9 {
10    if(k > 60) return; // 素因子连乘最多不能超过 60 次幂, 因为  $2^{60} > 10^{18}$ 
11    if(num == total) {
12        ll p = (ll)(pow(n, 1.0 / (0.0 + k)) + eps) - 1; // 先把 1 去掉, 精度误差 eps
13        ans += p;
14        return ;
15    }
16    if(cur == len) return ;
17    dfs(cur + 1, num, total, k); // 第 i 个素数不选
18    dfs(cur + 1, num + 1, total, k * prime[cur]); // 第 i 个素数选择
19 }
20
21 int main()
22 {
23     while(~scanf("%lf", &n)) {
24         ll res = 0;
25         for(int i = 1; i <= 3; ++i) {
26             ans = 0;
27             dfs(0, 0, i, 1);
28             // 从下标 0 开始, 当前选择素数个数为 0 需要选择素数个数, i 个选择素数乘积为 1
29             if(i & 1) res += ans;
30             else res -= ans;
31         }
32         res += 1; // 在 dfs 时都没有统计
33         printf("%lld\n", res);
34     }
35     return 0;
36 }
```

6.3 博弈论

6.3.1 威佐夫博弈 (Wythoff Game)

有两堆石子各有 a, b 个，两个人轮流从某一堆或同时从两堆中取同样多的物品，规定每次至少取一个，多者不限，最后取光者得胜。判断先手是胜者还是负者。

我们用 $(a_k, b_k), a_k \leq b_k, k = 0, 1, 2, \dots, n$ 表示两堆物品的数量并称其为局势，如果甲面对 $(0, 0)$ ，那么甲已经输了，这种局势我们称为奇异局势。前几个奇异局势是： $(0, 0), (1, 2), (3, 5), (4, 7), (6, 10), (8, 13), (9, 15), (11, 18), (12, 20)$ 。可以看出， $a_0 = b_0 = 0$ ， a_k 是未在前面出现过的最小自然数，而 $b_k = a_k + k$ ，奇异局势有如下三条性质：

1. 任何自然数都包含在一个且仅有一个奇异局势中。由于 a_k 是未在前面出现过的最小自然数，所以有 $a_k > a_{k-1}$ ，而 $b_k = a_k + k > a_{k-1} + k - 1 = b_{k-1} > a_{k-1}$ 。所以性质 1 成立。
2. 任意操作都可将奇异局势变为非奇异局势。事实上，若只改变奇异局势 (a_k, b_k) 的某一个分量，那么另一个分量不可能在其他奇异局势中，所以必然是非奇异局势。如果使 (a_k, b_k) 的两个分量同时减少，则由于其差不变，且不可能是其他奇异局势的差，因此也是非奇异局势。
3. 采用适当的方法，可以将非奇异局势变为奇异局势。

从如上性质可知，两个人如果都采用正确操作，那么面对非奇异局势，先拿者必胜；反之，则后拿者取胜。任给一个局势 (a, b) ，判断它是不是奇异局势，有如下公式：

$$a_k = \lfloor k \times \frac{1 + \sqrt{5}}{2} \rfloor, b_k = a_k + k \quad (k = 0, 1, 2, \dots, n)$$

由于 $\frac{1+\sqrt{5}}{2} = \frac{2}{\sqrt{5}-1}$ ，可以先求出 $j = \lfloor a \times \frac{\sqrt{5}-1}{2} \rfloor$ ，则 $a_j = \lfloor j \times \frac{\sqrt{5}+1}{2} \rfloor$ ，

- 若 $a = \lfloor j \times \frac{\sqrt{5}+1}{2} \rfloor = a_j$ 那么判断 b 是否等于 $b_j = a_j + j = a + j$
- 若 a 不等于 a_j ，那么判断是否满足 $a = a_{j+1} = \lfloor (j+1) \times \frac{\sqrt{5}+1}{2} \rfloor, b = b_{j+1} = a_{j+1} + j + 1$
- 若都不是，那么就不是奇异局势。

```

1 double p = (sqrt(5.0) + 1.0) / 2.0;
2 if(a > b) swap(a, b);
3 int k = (int)(a / p);
4 if((a == (int)(k * p) && b == a + k) ||
5    (a == (int)((k + 1) * p) && b == a + k + 1)) {
6    printf("0\n"); // 先手输
7 } else {
8    printf("1\n"); // 先手胜
9 }
```

6.4 约瑟夫环问题

n 个人顺时针站成一圈，编号 $0 \sim n - 1$ ，初始位置为 0，每次让第 m 个人出列，问最后剩下的人的编号。

定义当有 n 个人围成一圈时最终剩下人的编号为 $dp[n]$ ，显然有 $dp[1] = 0$ 。考虑 $dp[n]$ 向 $dp[n - 1]$ 的转化。

当有 n 个人时，编号顺序为：

$$0 \ 1 \ 2 \ \dots \ m-2 \ m-1 \ m \dots \ n-2 \ n-1$$

起始位置是 0，第 m 个人（编号为 $m - 1$ ）出列后，变为：

$$0 \ 1 \ 2 \ \dots \ m-2 \ \quad m \ \dots \ n-2 \ n-1$$

现在只剩下了 $n - 1$ 个人。对比 $n - 1$ 个人的时候的初始情况，把 $n - 1$ 个人的 0 编号和 n 个人剔除 $m - 1$ 编号后的 m 编号对齐，把 $n - 1$ 个人的 1 编号和 n 个人剔除 $m - 1$ 编号后的 $m + 1$ 编号对齐。。。依此类推可以由当 $n - 1$ 个人围成一圈最终剩下的人编号 $dp[n - 1]$ 得到递推式：

$$dp[n] = (dp[n - 1] + m) \% n$$

6.4.1 第一次第 m 个人出列，以后每次第 k 个人出列

PKU 1781

注意到状态定义： $dp[n]$ 表示 n 个人围成一圈最终剩下的人的编号（编号从 0 开始）。因为第一次第 m 个人出列的时候是 n 个人围成一圈，所以有： $dp[n] = (dp[n - 1] + m) \% n$ 。而当 $i < n$ 个人围成一圈时，每次都是第 k 个人出列，所以： $dp[i] = (dp[i - 1] + k) \% i$ ($i < n$)。编号从 0 开始。

```

1 dp[1] = 0;
2 for (int i = 2; i < n; ++i) {
3     dp[i] = (dp[i - 1] + K) % i;
4 }
5 dp[n] = (dp[n - 1] + m) % n;
```

6.4.2 每次都是第 2 个人出列，求 n 个人最终剩下的编号（编号从 1 开始）

PKU 1781

$n \leq 10^{18}$ 。递推式：

```

1 dp[1] = 1
2 dp[i] = (dp[i - 1] + 2) % i   (i > 1)
3 if (dp[i] == 0) dp[i] = i;
```

打表找规律可以发现：

$$\begin{aligned} dp[1] &= 1 \\ dp[2 * i] &= 2 * dp[i] - 1 \\ dp[2 * i + 1] &= 2 * dp[i] + 1 \end{aligned}$$

递归解决即可。

6.4.3 第 i 轮从上一轮出局的人的下一个人开始从 1 报数，报到 i 就停止且报到 i 的这个人出局

HDU 5643

$n \leq 5000$ 围成一圈，编号依次为 $1 \sim 5000$ ，第一轮第一个人从 1 开始报数，报到 1 就停止且报到 1 的这个人出局。第二轮从上一轮出局的人的下一个人开始从 1 报数，报到 2 就停止且报到 2 的这个人出局。第三轮从上一轮出局的人的下一个人开始从 1 报数，报到 3 就停止且报到 3 的这个人出局。依次类推，求最终剩下的人的编号。

先把下标都从 0 开始，最后再统一加上 1。用 $dp[i][j]$ 表示当一共 i 个人围成一圈并且第一个人开始报数 j 得时候最终剩下的人的编号。初始化： $dp[1][] = 0$ 。状态转移：

$$\begin{aligned} dp[i][j] &= (dp[i-1][j+1] + j) \% i \\ ans[i] &= dp[i][1] + 1 \end{aligned}$$

$O(n^2)$ 预处理即可。

6.4.4 获得每一轮出列的人的编号

PKU 2886 和 HDU 5860

每个人有一个下一次出列的指向性： $data[i]$ 表示当第 i 个人出列时下一个出列的人是从他左起 ($data[i] > 0$) 或者右起 ($data[i] < 0$) 的第 $|data[i]|$ 个人，第一次第 k 个人出列。

借助线段树解决。叶子结点 0/1 表示当前编号的人是否已经出列。区间表示当前区间剩余未出列的人数。当一个人出列时，求出下一个出列的人是剩下人中的第几个。区间查询，单点更新。

时间复杂度： $O(n \log n)$ 。

[HDU 5860]: 现在有 $n \leq 3 * 10^6$ 个人，编号为 $1 \sim n$ ，依次排列成一列，给定一个 k ，每一轮依次为当前排列的第 1、第 $k+1$ 、第 $2k+1$ …直到列尾的人出局，现在有 $Q \leq 10^5$ 个询问，每个询问为 $ack[i]$ ，问第 $ask[i]$ 个出局的人的编号。

例如： $n = 7, k = 3$ 时出局的人的编号依次是：

1 4 7 2 5 3 6

```

1 const int MAX_N = 3000010;
2
3 int T, n, K, Q, Max;
4 int ans[MAX_N], stree[MAX_N << 2], ask[1000010];
5
6 void build(int left, int right, int cur)
7 {
8     if (left == right) {
9         stree[cur] = 1;
10    return;
11 }
12    int mid = (left + right) >> 1;
13    build(left, mid, lson(cur));
14    build(mid + 1, right, rson(cur));
15    stree[cur] = stree[lson(cur)] + stree[rson(cur)];
16 }
17
18 int query(int pos, int left, int right, int cur)
19 {
20     if (left == right) return left;
21     int mid = (left + right) >> 1;
22     if (pos <= stree[lson(cur)]) return query(pos, left, mid, lson(cur));
23     else return query(pos - stree[lson(cur)], mid + 1, right, rson(cur));
24 }
25
26 void update(int pos, int left, int right, int cur)
27 {
28     if (left == right) {
29         stree[cur] = 0;
30         return;
31     }
32     int mid = (left + right) >> 1;
33     if (stree[lson(cur)] >= pos) update(pos, left, mid, lson(cur));
34     else update(pos - stree[lson(cur)], mid + 1, right, rson(cur));
35     stree[cur] = stree[lson(cur)] + stree[rson(cur)];
36 }
37

```

```
38 void solve()
39 {
40     build(1, n, 1);
41     int total = 0, left = n;
42     while (left) {
43         int i = 1, cnt = 0;
44         while (i <= left) {
45             ans[++total] = query(i, 1, n, 1);
46             cnt++;
47             i += K;
48             if (total >= Max) return;
49         }
50         i -= K;
51         while (i >= 1) {
52             update(i, 1, n, 1);
53             i -= K;
54         }
55         left -= cnt;
56     }
57 }
58
59 int main()
60 {
61     scanf("%d", &T);
62     while (T--) {
63         scanf("%d%d%d", &n, &K, &Q);
64         Max = 0;
65         for (int i = 0; i < Q; ++i) {
66             scanf("%d", &ask[i]);
67             Max = max(Max, ask[i]);
68         }
69         solve();
70         for (int i = 0; i < Q; ++i) {
71             printf("%d\n", ans[ask[i]]);
72         }
73     }
74     return 0;
75 }
```

6.5 康托展开

[UVALive 6665 Dragonas Cruller]

以九宫格的形式给出 0~8 八个数字，然后通过移动 0 数字，使这个九宫格变成给定的状态，上下移动和左右移动的权值不一样，求最小移动路径值。

用康拓排序来去重。因为上下移动和左右移动的权值不一样，所以必须使用优先队列，这样才能保证解的优先性。

```

1 const int MAX_N = 400000;
2
3 int ch, cv, ans, st, ed, pos, npos;
4 int ast[15], aed[15], temp[15], fac[15] = {1, 1};
5 int vis[MAX_N];
6
7 struct Node{
8     int cost, pos, cantor;
9     // 当前状态下的路径值, 0 位置和康拓值
10    bool operator < (const Node& rhs) const {
11        return cost > rhs.cost;
12    }
13 }cur, nextnode;
14
15 void CalcFac()
16 {
17     for (int i = 2; i < 10; i++) {
18         fac[i] = fac[i - 1] * i; // 计算阶乘
19     }
20 }
21
22 int AToInt(int a[]) // 数组状态装换成相应康拓值
23 {
24     int x = 0;
25     for (int i = 0; i < 9; i++) {
26         int y = a[i];
27         for (int j = 0; j < i; j++) {
28             // 去掉已经用过的比 a[i] 小的值
29             if (a[j] < a[i]) y--;
30         }
31         x += fac[8 - i] * y; // 从右往左看是第 8-i 位
32     }
33     return x;
34 }
35
36 void IntToA(int x, int *a) // 康拓值转换成数组
37 {
38     int vvis[10];
39     memset (vvis, 0, sizeof (vvis));
40     for (int i = 0; i < 9; i++) {
41         int y = x / fac[8 - i];
42         // 第 8-i 位应该是第 y 小的数字
43         for (int j = 0; j < 9; j++) {
44             if (!vvis[j]) { // 没被用过的数字
45                 if (y == 0) { // 这时 j 就是没被用过的第 y 大的数字
46                     vvis[j] = 1, a[i] = j;
47                     break;
48                 }
49                 y--;
50             }
51         }
52         x %= fac[8 - i];
53     }
54 }
55
56 void change(int i, int flag)

```

```

57 // 需要在每次 i 之后将 temp 数组还原
58 // 如果是每次 i 都重新调用 IntToA 来计算 temp 会 TLE
59     if (flag == 1) {
60         if (i == 0){
61             npos = (pos + 3) % 9; // 向下移动
62             swap(temp[pos], temp[npos]);
63         } else if (i == 1) {
64             npos = (pos + 6) % 9; // 向上移动
65             swap(temp[pos], temp[npos]);
66         } else if (i == 2) {
67             npos = (pos + 8) % 9; // 向左移动
68             swap(temp[pos], temp[npos]);
69         } else if (i == 3) { // 向右移动
70             npos = (pos + 1) % 9;
71             swap(temp[pos], temp[npos]);
72         }
73     } else swap(temp[pos], temp[npos]); // 还原temp
74 }
75
76 int bfs()
77 {
78     int x;
79     priority_queue<Node> que;
80     memset(vis, 0, sizeof(vis));
81     cur.cost = 0;
82     cur.cantor = st;
83     que.push(cur);
84     vis[st] = 1;
85     int flag = 0;
86     while (!que.empty()) {
87         cur = que.top(); que.pop();
88         if (cur.cantor == ed) {
89             x = cur.cost, flag=1;
90             break;
91         }
92         pos = cur.pos;
93         IntToA(cur.cantor, temp);
94         for (int i = 0; i < 4; i++) {
95             change(i, 1);
96             nextnode.cost = cur.cost + ((i < 2) ? cv : ch);
97             nextnode.pos = npos;
98             nextnode.cantor = AToInt(temp);
99             if (!vis[nextnode.cantor] || vis[nextnode.cantor] > nextnode.cost) {
100                 vis[nextnode.cantor] = nextnode.cost;
101                 que.push(nextnode);
102             }
103             change(i, 0);
104         }
105     }
106     return x;
107 }
108
109 int main()
110 {
111     CalcFac();
112     while (~scanf("%d%d", &ch, &cv)) {
113         if (ch == 0 && cv == 0) break;
114         for (int i = 0; i < 9; i++) {
115             scanf("%d", &ast[i]);
116             if (ast[i] == 0) cur.pos = i;
117         }
118         for (int i = 0; i < 9; i++) scanf("%d", &aed[i]);
119         st = AToInt(ast);
120         ed = AToInt(aed);
121         ans = bfs();
}

```

```

122     printf("%d\n", ans);
123 }
124 return 0;
125 }
```

6.6 FFT

```

1 const int MAX_N = (1 << 18) + 10;
2 const double DPI = 2.0 * acos(-1.0);
3
4 struct CP {
5     double a, b;
6
7     CP() {}
8     CP(double _a, double _b): a(_a), b(_b) {}
9     CP operator * (const CP& rhs) const {
10         return CP(a * rhs.a - b * rhs.b, a * rhs.b + b * rhs.a);
11     }
12     CP operator + (const CP& rhs) const {
13         return CP(a + rhs.a, b + rhs.b);
14     }
15     CP operator - (const CP& rhs) const {
16         return CP(a - rhs.a, b - rhs.b);
17     }
18 } A1[MAX_N], A2[MAX_N];
19
20 int N, bit;
21 int rev[MAX_N];
22
23 // 初始化位逆序置换
24 void init(int Max) { // Max=( 变换前最高次项次数 )*2 + 1
25     // 二进制平摊翻转置换
26     for (N = 1, bit = 0; N < Max; N <= 1, bit++) ;
27     for (int i = 1; i < N; ++i) {
28         rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
29     }
30 }
31
32 void FFT(CP *A, int n, int ope) { // ope=1:DFT, ope=-1:IDFT
33     for (int i = 0; i < n; ++i) {
34         // 交换互为下标反转的元素
35         if (i < rev[i]) swap(A[i], A[rev[i]]);
36     }
37     for (int m = 2; m <= n; m <= 1) { // 控制层数
38         CP wm(cos(DPI / m), ope * sin(DPI / m)); // m 次单位复根
39         for (int k = 0; k < n; k += m) { // 控制起始下标
40             CP w(1, 0); // 初始化螺旋因子
41             for (int j = k; j < k + (m >> 1); ++j) { // 配对
42                 CP t = w * A[j + (m >> 1)];
43                 CP u = A[j];
44                 A[j] = u + t;
45                 A[j + (m >> 1)] = u - t;
46                 w = w * wm; // 更新螺旋因子
47             } // 上面的操作叫蝴蝶操作
48         }
49     }
50     if (ope == -1) { // IDFT
51         for (int i = 0; i < n; ++i) {
52             A[i].a /= n;
53         }
54     }
55 }
```

6.6.1 UVA 12633 超级车

在一个最大 $R * C (R, C \leq 50000)$ 的棋盘上放 $n \leq 50000$ 个超级车，每个超级车能攻击同行/同列/同主斜线上的格子（主斜线即：左上到右下的斜线）。问放完所有超级车后有多少个格子不会被攻击到？

一条主对角线上的点的同一形式： $C - y + x$ 。把所有的没被攻击到的格子都当成是主对角线上没被攻击来统计。对被攻击到的行，列，主对角线做标记，然后没被攻击到的行列做卷积。

```

1 void solve() {
2     init();
3     memset(A1, 0, sizeof(A1));
4     memset(A2, 0, sizeof(A2));
5     for (int i = 0; i < R; ++i) if (!row[i]) A1[i].a += 1;
6     for (int i = 0; i < C; ++i) if (!col[i]) A2[i].a += 1;
7     FFT(A1, N, 1); FFT(A2, N, 1);
8     for (int i = 0; i < N; ++i) A1[i] = A1[i] * A2[i];
9     FFT(A1, N, -1);
10    ll ans = 0;
11    for (int i = 0; i < Max; ++i) {
12        if (!dia[i])
13            ans += (ll)(A1[i].a + 0.5);
14    }
15    printf("Case %d: %lld\n", ++cases, ans);
16}
17
18
19 int main() {
20     scanf("%d", &T);
21     while (T--) {
22         memset(row, 0, sizeof(row));
23         memset(col, 0, sizeof(col));
24         memset(dia, 0, sizeof(dia));
25         scanf("%d%d%d", &R, &C, &n);
26         for (int i = 1; i <= n; ++i) {
27             int x, y;
28             scanf("%d%d", &x, &y);
29             x--, y = C - y;
30             row[x] = 1, col[y] = 1;
31             dia[x + y] = 1;
32         }
33         solve();
34     }
35     return 0;
36 }
```

6.6.2 SPOJ Triple Sum

给出 $n \leq 40000$ 个互不相同的数 $a_i \in [-20000, 20000]$ ，任取三个数（每个数不能重复取）求和，问所有可能组成的和，以及其组成方案数。

先把数域平移至 $[0, 40000]$ ，然后统计每个数出现的次数，作为系数多项式 $A[]$ 。定义 $f(x) = \sum x^{a_i}$, $f(2x) = \sum x^{2*a_i}$ 。

$$\begin{aligned} A &= f(x) \otimes f(x) \otimes f(x) = 6 * x^{a_1+a_2+a_3} + 3 * x^{2a_1+a_2} + x^{3a_1} \\ B &= f(2x) \otimes f(x) = x^{2a_1+a_2} + x^{3a_1} \\ C &= f(3x) = x^{3a_1} \end{aligned}$$

所以：

$$Ans = \frac{A - 3 * B + 2 * C}{6}$$

```

1 void solve() {
2     init(Max * 3 + 10);
3     FFT(A1, N, 1); FFT(A2, N, 1); FFT(B, N, 1);;
```

```

4   for (int i = 0; i < N; ++i) {
5     A1[i] = A1[i] * A2[i];
6     A1[i] = A1[i] * A2[i];
7     B[i] = B[i] * A2[i];
8   }
9   FFT(A1, N, -1); FFT(B, N, -1);
10  for (int i = 0; i < Max2; ++i) {
11    if (A1[i].a <= eps) continue;
12    double ans = (A1[i].a - B[i].a * 3 + C[i].a * 2) / 6.0;
13    if (ans > eps) {
14      printf("%d : %.01f\n", i - 60000, ans);
15    }
16  }
17}
18
19 int main() {
20   scanf("%d", &n);
21   Max = 0;
22   for (int i = 0; i < n; ++i) {
23     int t;
24     scanf("%d", &t);
25     t += 20000;
26     A1[t].a += 1;
27     A2[t].a += 1;
28     B[2 * t].a += 1;
29     C[3 * t].a += 1;
30     Max = max(Max, t);
31   }
32   solve();
33   return 0;
34 }
```

6.6.3 HDU 5730 Shell Necklace

给一排 $n \leq 10^5$ 颗珍珠（不考虑成环），然后用 a_i 表示可以把连续的 i 颗珍珠染色的染色方案数，求 n 颗珍珠的染色方案数，对 313 取模。

用 $dp[i]$ 表示将 i 颗珍珠染色的染色方案数，初始化 $dp[0] = 1$ ，枚举第一串连续的珍珠的染色方案数可得：

$$dp[i] = \sum_{j=1}^i a[j] * dp[i-j]$$

用分治 +FFT 卷积加速，先用 ‘double’ 存，最后再取模，时间复杂度： $O(n \log^2 n)$ 。

```

1 void cdq(int left, int right) {
2   if (left == right) return;
3   int mid = (left + right) >> 1;
4   cdq(left, mid);
5   int len = right - left + 1;
6   init(len);
7   for (int i = 0; i <= mid - left; ++i) A1[i].a = dp[i + left] % MOD, A1[i].b = 0;
8   for (int i = mid - left + 1; i < N; ++i) A1[i].a = A1[i].b = 0;
9   for (int i = 0; i < N; ++i) A2[i].a = data[i], A2[i].b = 0;
10  FFT(A1, N, 1); FFT(A2, N, 1);
11  for (int i = 0; i < N; ++i) A1[i] = A1[i] * A2[i];
12  FFT(A1, N, -1);
13  for (int i = mid - left + 1; i <= right - left; ++i) {
14    dp[i + left] = (dp[i + left] + (ll)(A1[i].a + 0.5) % MOD) % MOD;
15  }
16  cdq(mid + 1, right);
17}
18
19 int main() {
```

```

20     int n;
21     while (~scanf("%d", &n) && n) {
22         memset(data, 0, sizeof(data));
23         memset(dp, 0, sizeof(dp));
24         for (int i = 1; i <= n; ++i) {
25             scanf("%d", &data[i]);
26             data[i] %= MOD;
27         }
28         dp[0] = 1;
29         cdq(0, n);
30         printf("%d\n", dp[n]);
31     }
32     return 0;
33 }
```

6.7 NTT

模数需要满足形式: $a * 2^k + 1$, 并且 k 因为尽量大的数字, 然后求原根为 G , 例如: 998244353 的原根为 3, 152076289 的原根为 106。

比较好用的素数有: $2281701377 = 17 * 2^{27} + 1$, 平方刚好不会爆 long long; $1004535809 = 479 * 2^{21} + 1$ 加起来刚好不会爆 int, 并且这个数的原根是 3。

```

1 const int MAX_N = (1 << 18) + 10;
2 const int MOD = 152076289;
3 const int NUM = 20;
4 const int G = 106;
5 const double DPI = 2.0 * acos(-1.0);
6
7 int N, bit;
8 int rev[MAX_N], A1[MAX_N], A2[MAX_N], wn[2][NUM], inv[MAX_N];
9
10 int Qpow(int a, int b, int mod) {
11     int ret = 1; a %= mod;
12     while (b) {
13         if (b & 1) ret = 111 * ret * a % mod;
14         a = 111 * a * a % mod; b >>= 1;
15     }
16     return ret;
17 }
18
19 void init(int Max) {
20     for (N = 1, bit = 0; N < Max; N <= 1, ++bit) ;
21     for (int i = 1; i < N; ++i) {
22         rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
23     }
24 }
25
26 void NTT(int *A, int n, int ope) {
27     for (int i = 0; i < n; ++i) if (i < rev[i]) swap(A[i], A[rev[i]]);
28     int id = (ope == -1) ? 1 : 0, p = 1;
29     for (int m = 2; m <= n; m <= 1, ++p) { // m 次单位根
30         for (int k = 0; k < n; k += m) {
31             for (int j = k, w = 1; j < k + (m >> 1); ++j) { // 折半
32                 int t = 111 * w * A[j + (m >> 1)] % MOD; // 右项
33                 int u = A[j] % MOD; // 左项
34                 A[j] = u + t, A[j + (m >> 1)] = u - t;
35                 if (A[j] >= MOD) A[j] -= MOD;
36                 if (A[j + (m >> 1)] < 0) A[j + (m >> 1)] += MOD;
37                 w = 111 * w * wn[id][p] % MOD;
38             }
39         }
40     }
```

```

41     if (ope == -1) {
42         for (int i = 0; i < n; ++i) {
43             A[i] = 111 * A[i] * inv[n] % MOD;
44         }
45     }
46 }
47
48 void Pre() {
49     fac[0] = 1;
50     for (int i = 1; i < 10010 * 2; ++i) fac[i] = 111 * fac[i - 1] * i % MOD;
51     afac[10000 * 2] = Qpow(fac[10000 * 2], MOD - 2, MOD);
52     for (int i = 10000 * 2; i >= 1; --i) afac[i - 1] = 111 * afac[i] * i % MOD;
53     for (int i = 1; i < MAX_N; ++i) inv[i] = Qpow(i, MOD - 2, MOD);
54     for (int i = 0; i < NUM; ++i) {
55         int t = 1 << i;
56         wn[0][i] = Qpow(G, (MOD - 1) / t, MOD); // 预处理求值点
57         wn[1][i] = Qpow(wn[0][i], MOD - 2, MOD); // 求值点逆元
58     }
59 }
```

6.7.1 HDU 5322 Hope

给一个 $n \leq 10^5$, 对于 $1 \sim n$ 的一个排列, 对于 A_i , 如果存在最小的 $j > i$ 并且 $A_j > A_i$ 那么 i 和 j 之间有一条边, 那么一种排列就会形成若干连通块, 把每个连通块的大小求和为 P , 那么这个排列的贡献就是 $P * P$, 求 $1 \sim n$ 的所有排列的贡献和, 答案对 998244353 取模。

用 $dp[i]$ 表示 $1 \sim i$ 所有排列的贡献和。枚举第一个连通块的大小, 第一个连通块的最后一个数字必然是 i , 初始化 $dp[0] = 1$, 可得:

$$dp[i] = \sum_{j=1}^i C_{i-1}^{j-1} * (j-1)! * j^2 * dp[i-1] = (i-1)! * \sum_{j=1}^i j^2 * \frac{dp[i-j]}{(i-j)!}$$

分治预处理 +NTT, 时间复杂度: $O(n \log^2 n)$ 。

```

1 void cdq(int left, int right) {
2     if (left == right) return;
3     int mid = (left + right) >> 1;
4     cdq(left, mid);
5     int len = right - left + 1;
6     init(len);
7     for (int i = 0; i <= mid - left; ++i) {
8         A1[i] = 111 * dp[i + left] * afac[i + left] % MOD;
9     }
10    for (int i = mid - left + 1; i < N; ++i) A1[i] = 0;
11    for (int i = 0; i < N; ++i) A2[i] = sq[i];
12    NTT(A1, N, 1); NTT(A2, N, 1);
13    for (int i = 0; i < N; ++i) A1[i] = 111 * A1[i] * A2[i] % MOD;
14    NTT(A1, N, -1);
15    for (int i = mid - left + 1; i <= right - left; ++i) {
16        dp[i + left] += 111 * A1[i] * fac[i + left - 1] % MOD;
17        dp[i + left] %= MOD;
18    }
19    cdq(mid + 1, right);
20 }
21
22 void solve() {
23     fac[0] = 1;
24     for (int i = 1; i <= 100000; ++i) fac[i] = 111 * fac[i - 1] * i % MOD;
25     afac[100000] = Qpow(fac[100000], MOD - 2, MOD);
26     for (int i = 100000; i >= 1; --i) afac[i - 1] = 111 * afac[i] * i % MOD;
27     for (int i = 1; i <= 100000; ++i) sq[i] = 111 * i * i % MOD;
28     for (int i = 1; i < MAX_N; ++i) inv[i] = Qpow(i, MOD - 2, MOD);
29     dp[0] = 1;
```

```

30     cdq(0, 100000);
31 }
32
33 int main() {
34     solve();
35     int x;
36     while (~scanf("%d", &x)) {
37         printf("%d\n", dp[x]);
38     }
39     return 0;
40 }
```

6.7.2 HDU 5552 图计数

给一个 $n \leq 10000$ 和 $m < 2^{31}$, 表示一个 n 个点的无向图和每条边可以被染成 m 种颜色, 求满足下面条件的无向图的个数:

- 两个点最多只有一条边
- 至少要有一个环
- 图连通, 即图上任意两点可互达。

结果对 152076289 取模。

定义 $g[n]$ 表示 n 个点的生成图计数, 可以不连通, 每条边有 $m+1$ 种选择 (不取这条边和 m 种染色), 一共有 $\frac{n*(n-1)}{2}$ 条边, 那么:

$$g[n] = (m+1)^{\frac{n*(n-1)}{2}}$$

定义 $h[n]$ 表示 n 个点的生成树计数, 根据 Prüfer 序列和一共有 $n-1$ 条边可得:

$$h[n] = n^{n-2} * m^{n-1}$$

定义 $f[n]$ 表示 n 个点的连通图计数, 那么 $f[n] = \text{图的总数} - \text{非联通图的个数}$, 枚举 1 号点所在连通块的大小, 可得:

$$\begin{aligned} f[n] &= g[n] - \sum_{i=1}^{n-1} C_{n-1}^{i-1} * f[i] * g[n-i] \\ &= g[n] - (n-1)! * \sum_{i=1}^{n-1} \frac{f[i]}{(i-1)!} * \frac{g[n-i]}{(n-i)!} \end{aligned}$$

这是一个很明显的卷积形式, 分治 +NTT 搞下。最后 n 个点的有环图个数 $= f[n] - h[n]$ 。

时间复杂度: $O(n \log^2 n)$

```

1 int T, n, m, cases = 0;
2 int fac[10010 * 2], afac[10010 * 2], dp[10010 * 2], g[MAX_N];
3
4 void Pre() {
5     fac[0] = 1;
6     for (int i = 1; i < 10010 * 2; ++i) fac[i] = 111 * fac[i - 1] * i % MOD;
7     afac[10000 * 2] = Qpow(fac[10000 * 2], MOD - 2, MOD);
8     for (int i = 10000 * 2; i >= 1; --i) afac[i - 1] = 111 * afac[i] * i % MOD;
9     for (int i = 1; i < MAX_N; ++i) inv[i] = Qpow(i, MOD - 2, MOD);
10    for (int i = 0; i < NUM; ++i) {
11        int t = 1 << i;
12        wn[0][i] = Qpow(G, (MOD - 1) / t, MOD); // 预处理求值点
13        wn[1][i] = Qpow(wn[0][i], MOD - 2, MOD); // 求值点逆元
14    }
15 }
16
17 void cdq(int left, int right) {
18     if (left == right) return;
19     int mid = (left + right) >> 1, len = right - left + 1;
```

```

20 cdq(left, mid);
21 init(len);
22 for (int i = 0; i <= mid - left; ++i) A1[i] = 111 * dp[i + left] * afac[i + left - 1] % MOD;
23 for (int i = mid - left + 1; i < N; ++i) A1[i] = 0;
24 for (int i = 0; i < N; ++i) A2[i] = 111 * g[i] * afac[i] % MOD;
25
26 NTT(A1, N, 1); NTT(A2, N, 1);
27 for (int i = 0; i < N; ++i) A1[i] = 111 * A1[i] * A2[i] % MOD;
28 NTT(A1, N, -1);
29 for (int i = mid - left + 1; i <= right - left; ++i) {
30     dp[i + left] -= 111 * fac[i + left - 1] * A1[i] % MOD;
31     if (dp[i + left] < 0) dp[i + left] += MOD;
32 }
33 cdq(mid + 1, right);
34 }
35
36 int main() {
37     Pre();
38     scanf("%d", &T);
39     while (T--) {
40         scanf("%d%d", &n, &m);
41         for (int i = 0; i < n * 2; ++i) {
42             dp[i] = g[i] = Qpow(m + 1, i * (i - 1) / 2, MOD);
43         }
44         cdq(1, n);
45         int h = 111 * Qpow(n, n - 2, MOD) * Qpow(m, n - 1, MOD) % MOD;
46         int ans = (dp[n] - h + MOD) % MOD;
47         printf("Case #%d: %d\n", ++cases, ans);
48     }
49     return 0;
50 }
```

6.7.3 HDU 5829 卷积平移

给 $n \leq 10^5$ 个数, $a_i \in [0, 10^9]$, 对于这 n 个数的所有组合的非空子集 $s_i (2^n - 1)$ 个), 从中找到最大的 $\min(|s_i|, k)$ 个数求和, 对所有的 $k \in [1, n]$ 输出对 998244353 取模的结果。

先把所有的数按照从大到小排序, 考虑每个数的贡献:

$$\begin{aligned}
dp[k] &= \sum_{i=k}^n (C_{i-1}^0 + C_{i-1}^1 + \dots + C_{i-1}^{k-1}) * 2^{n-i} * a_i \\
&= dp[k-1] + \frac{1}{(k-1)!} \sum_{i=k}^n ((i-1)! * 2^{n-i} * a[i]) * \frac{1}{(i-k)!} \\
&= dp[k-1] + \frac{1}{(k-1)!} \sum_{i=k}^n A[i] * B[i-k]
\end{aligned}$$

其中:

$$A[i] = (i-1)! * 2^{n-i} * a[i], \quad B[i-k] = \frac{1}{(i-k)!}$$

如果把 $B[i]$ 改为 $B[i] = \frac{1}{(n-i)!}$, 那么可得:

$$\begin{aligned}
dp[n+k] &= \frac{1}{(k-1)!} \sum A[i] * B[n-i] \\
&= \frac{1}{(k-1)!} \sum A[i] * B[n-(n+k-i)] \\
&= \frac{1}{(k-1)!} \sum A[i] * B[i-k]
\end{aligned}$$

这样子就和初始的式子等价了。最后别忘了前缀和。这道题提交时还需要快读。

时间复杂度: $O(n \log n)$

```

1 int data[100010], pw[100010], fac[100010], afac[100010];
2
3 inline bool cmp(int a, int b) {
4     return a > b;
5 }
6
7 void Pre() {
8     pw[0] = fac[0] = 1;
9     for (int i = 1; i < 100010; ++i) {
10         pw[i] = 111 * pw[i - 1] * 2 % MOD;
11         fac[i] = 111 * fac[i - 1] * i % MOD;
12     }
13     afac[100000] = Qpow(fac[100000], MOD - 2, MOD);
14     for (int i = 100000; i >= 1; --i) afac[i - 1] = 111 * afac[i] * i % MOD;
15     for (int i = 1; i < MAX_N; ++i) inv[i] = Qpow(i, MOD - 2, MOD);
16     for (int i = 0; i < NUM; ++i) {
17         int t = 1 << i;
18         wn[0][i] = Qpow(G, (MOD - 1) / t, MOD); // 预处理求值点
19         wn[1][i] = Qpow(wn[0][i], MOD - 2, MOD); // 求值点逆元
20     }
21 }
22
23 struct FastIO {
24     static const int S = 1000000;
25     int wpos, pos, len;
26     char wbuf[S];
27     FastIO(): wpos(0) {}
28     inline int xchar() {
29         static char buf[S];
30         if (pos == len) pos = 0, len = fread(buf, 1, S, stdin);
31         if (pos == len) return -1;
32         return buf[pos++];
33     }
34     inline int xint() {
35         int c = xchar(), x = 0;
36         while (c <= 32 && ~c) c = xchar();
37         if (c == -1) return -1;
38         for (; c >= '0' && c <= '9'; c = xchar()) x = x * 10 + (c - '0');
39         return x;
40     }
41 } io;
42
43 int main() {
44     Pre();
45     int T, n;
46     T = io.xint();
47     //scanf("%d", &T);
48     while (T--) {
49         n = io.xint();
50         // n = Read();
51         for (int i = 1; i <= n; ++i) data[i] = io.xint();
52         sort(data + 1, data + n + 1, cmp);
53         init(2 * n + 1);
54         for (int i = 0; i < N; ++i) A1[i] = A2[i] = 0;
55         for (int i = 1; i <= n; ++i) {
56             A1[i] = 111 * fac[i - 1] * pw[n - i] % MOD * data[i] % MOD;
57             A2[i] = afac[n - i];
58         }
59         NTT(A1, N, 1); NTT(A2, N, 1);
60         for (int i = 0; i < N; ++i) A1[i] = 111 * A1[i] * A2[i] % MOD;
61         NTT(A1, N, -1);
62         int sum = 0;
63         for (int i = n + 1; i <= 2 * n; ++i) {
64             sum += 111 * A1[i] * afac[i - n - 1] % MOD;
65             if (sum >= MOD) sum -= MOD;
66         }
67     }
68 }
```

```
66         printf( "%d ", sum );
67     }
68     printf( "\n" );
69 }
70 return 0;
71 }
```

Chapter 7

动态规划

7.1 LIS

在已经得到的最大长度上升子序列 $extra$ 中查找第一个大于等于 $data[i]$ 的位置 pos ，那么对于 $data[i]$ 位置的最大上升子序列长度就为 $pos + 1$ ，然后用 $data[i]$ 替换 $extra[pos]$ ，这样做将最大长度子序列的“潜力”增大了，尽管最终得到的序列并不是真正的最大上升子序列，但是长度不变。

```
1 int len = 1;
2 extra[0] = data[0];
3 for (int i = 1; i < n; ++i) {
4     int pos = lower_bound(extra, extra + len, data[i]) - extra;
5     len = max(len, pos + 1);
6     extra[pos] = data[i];
7     // pos + 1 是每个位置可以得到的最大上升子序列长度
8 }
// 得到的 len 即是最大上升子序列长度
```

7.1.1 最长非降子序列

只需把上面的 $lower_bound()$ 改为 $upper_bound()$ 即可。

7.1.2 三维 LIS

UVALive 6667: 定义两个点 i 和 j ，如果满足 $x_i < x_j, y_i < y_j, z_i < z_j$ ，那么称： $i < j$ ，给 $n \leq 3 * 10^5$ 个点计算 LIS。

先把所有点的 Z 坐标离散化，然后按照 XYZ 的优先级排序，考虑 cdq 分治。对于区间 $[left, right]$ 内的点再按照 YZX 的优先级排序，记录下 $mid + 1$ 的横坐标，需要特殊处理，借用两颗树状数组。

```
1 int n, MaxZ;
2 int bit[2][MAX_N], dp[MAX_N], Z[MAX_N];
3
4 struct Point {
5     int x, y, z, id;
6 } P[MAX_N], Q[MAX_N];
7
8 bool xyz(Point a, Point b)
9 {
10     if (a.x != b.x) return a.x < b.x;
11     if (a.y != b.y) return a.y < b.y;
12     return a.z < b.z;
13 }
14
15 bool yzx(Point a, Point b)
16 {
17     if (a.y != b.y) return a.y < b.y;
18     if (a.z != b.z) return a.z > b.z;
```

```

19     return a.x > b.x;
20 }
21
22 inline int lowbit(int x) { return x & (-x); }
23
24 inline void update(int id, int x, int value)
25 {
26     for (int i = x; i <= MaxZ; i += lowbit(i)) {
27         bit[id][i] = max(bit[id][i], value);
28     }
29 }
30
31 inline int query(int id, int x)
32 {
33     int ret = 0;
34     for (int i = x; i > 0; i -= lowbit(i)) {
35         ret = max(ret, bit[id][i]);
36     }
37     return ret;
38 }
39
40 void Clear(int id, int x)
41 {
42     for (int i = x; i <= MaxZ; i += lowbit(i)) {
43         bit[id][i] = 0;
44     }
45 }
46
47 void cdq(int left, int right)
48 {
49     if (left == right) return;
50     int mid = (left + right) >> 1;
51     int midX = P[mid + 1].x;
52     cdq(left, mid);
53     int total = 0;
54     for (int i = left; i <= right; ++i) {
55         Q[total] = P[i];
56         Q[total++].id = i; // 重新编号, 以便下面的比较插入和查询
57     }
58     sort(Q, Q + total, yzx);
59     for (int i = 0; i < total; ++i) {
60         int pos = Q[i].id, tmp;
61         if (pos <= mid) {
62             update(0, Q[i].z, dp[pos]);
63             if (Q[i].x != midX) update(1, Q[i].z, dp[pos]); // 单独建树
64         } else {
65             if (Q[i].x != midX) tmp = query(0, Q[i].z - 1);
66             else tmp = query(1, Q[i].z - 1);
67             dp[pos] = max(dp[pos], tmp + 1);
68         }
69     }
70     for (int i = 0; i < total; ++i) {
71         int pos = Q[i].id;
72         if (pos <= mid) { // 清空树状数组
73             Clear(0, Q[i].z);
74             if (Q[i].x != midX) Clear(1, Q[i].z);
75         }
76     }
77     cdq(mid + 1, right);
78 }
79
80 int main()
81 {
82     scanf("%d", &n);
83     for (int i = 0; i < n; ++i) {

```

```
84     scanf( "%d%d%d", &P[ i ].x, &P[ i ].y, &P[ i ].z );
85     Z[ i ] = P[ i ].z, P[ i ].id = i ;
86     dp[ i ] = 1, way[ i ] = 1;
87 }
88 sort(Z, Z + n);
89 MaxZ = unique(Z, Z + n) - Z;
90 for (int i = 0; i < n; ++i) {
91     P[ i ].z = lower_bound(Z, Z + MaxZ, P[ i ].z) - Z + 1;
92 }
93 for (int i = 0; i <= MaxZ; ++i) {
94     bit[ 0 ][ i ] = bit[ 1 ][ i ] = 0;
95 }
96 sort(P, P + n, xyz);
97 cdq(0, n - 1);
98 int ans = 0;
99 for (int i = 0; i < n; ++i) {
100     if (dp[ i ] > ans) ans = dp[ i ];
101 }
102 printf("%d\n", ans);
103 return 0;
104 }
```

7.2 区间 dp

区间 dp 这种 dp 受制于状态转移过程一般的复杂度都是 $O(n^3)$ ，并且第一层循环一般都是倒着递推。其主要难点也许在于如何确定状态转移，更具体的说，就是确定循环枚举的第三层的 k 的含义。一般都是指区间中的第 k 个位置或者第 i 个人是第 k 个顺序等。

7.2.1 LightOJ 1422

需要去参加 $n \leq 100$ 个聚会，每个聚会可能需要穿不同的衣服，用数字编号表示 (1 – 100)。如果连续的聚会需要穿的衣服一样，那就不用换衣服，也可以选择在身上的衣服外面再套上新的衣服，但是脱下的衣服不能在用于剩下的聚会的了，问最少需要准备多少件衣服？

先解释下样例。 $n = 4 \quad 1\ 2\ 1\ 2$

意思是第一场和第三场聚会需要穿 1 号衣服，第二场和第四场聚会需要穿 2 号衣服，那么可以选择在第一场聚会时穿上 1 号衣服，在第二场聚会时在 1 号衣服外面套上 2 号衣服，在第三场聚会时脱下外面的 2 号衣服，然后第四场聚会时在套上一件新的 2 号衣服，那么总共至少 3 件衣服。

用 $dp[i][j]$ 表示从第 i 场聚会到第 j 场聚会最少需要的衣服数量。区间最优可由子区间最优递推得到。如果把子区间独立看的话，子区间是不存在联系的，但是当合并成父区间时需要考虑第一个首尾位置的数字编号是否相同，如果相同的话那么就可以节省一件衣服，因为可以选择将这件衣服一直保留在最里层，当在第二个区间需要最后一次需要穿上这件衣服时将外面所有的衣服脱掉即可。所以有状态转移方程：

$$dp[i][j] = \min_{i \leq k < j} \{dp[i][k] + dp[k + 1][j]\} \quad if \quad i != j \quad and \quad data[i] = data[j] \quad then \quad dp[i][j] --$$

因为在得到区间 $[i, j]$ 的状态时需要知道区间 $[k, j]$ 状态的最优解并且 $i \leq k$ ，所以关于 i 的循环应是 $i : n - 1 \rightarrow 0$ (下标从 0 开始)。

初始化： $dp[i][j] = inf$, $dp[i][i] = 1$ 。时间复杂度： $O(n^3)$

```

1  scanf("%d", &n);
2  for (int i = 0; i < n; ++i) {
3      scanf("%d", &data[i]);
4  }
5  for (int i = 0; i < n; ++i) {
6      for (int j = 0; j < n; ++j) {
7          dp[i][j] = inf;
8      }
9  }
10 for (int i = 0; i < n; ++i) { dp[i][i] = 1; }
11 for (int i = n - 1; i >= 0; --i) {
12     for (int j = i; j < n; ++j) {
13         for (int k = i; k < j; ++k) {
14             dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j]);
15         }
16         if (i != j && data[i] == data[j]) dp[i][j]--;
17     }
18 }
19 printf("Case %d: %d\n", ++cases, dp[0][n - 1]);

```

7.2.2 PKU 1142

给一个长度 ≤ 100 只含 "(", ")" ", "]" 四种括号的字符串，这个串可能不是恰好匹配的，输出最短的完美匹配串，多解输出任意解。例如：对于 (()) 应输出 ()()

区间 dp，记录路径，dfs 还原括号匹配。

```

1 void dfs(int left, int right)
{
    if (left > right) return;
    if (left == right) {
        if (s[left] == '(' || s[left] == ')') {
            printf("(");
        } else {
    }
}

```

```

8         printf("[]");
9     }
10    return ;
11}
12if (pre[left][right] == -1) {
13    printf("%c", s[left]);
14    dfs(left + 1, right - 1);
15    printf("%c", s[right]);
16} else {
17    dfs(left, pre[left][right]);
18    dfs(pre[left][right] + 1, right);
19}
20}
21
22int main()
23{
24    while (gets(s)) {
25        int n = strlen(s);
26        if (n == 0) {
27            printf("\n");
28            continue;
29        }
30        for (int i = 0; i < n; ++i) {
31            for (int j = 0; j < n; ++j) {
32                if (i > j) dp[i][j] = 0;
33                else if (i == j) dp[i][j] = 1;
34                else dp[i][j] = inf;
35            }
36        }
37        for (int i = n - 1; i >= 0; --i) {
38            for (int j = i; j < n; ++j) {
39                if ((s[i] == '(' && s[j] == ')') ||
40                    s[i] == '[' && s[j] == ']') {
41                    dp[i][j] = dp[i + 1][j - 1];
42                    pre[i][j] = -1;
43                }
44                for (int k = i; k < j; ++k) {
45                    if (dp[i][k] + dp[k + 1][j] < dp[i][j]) {
46                        dp[i][j] = dp[i][k] + dp[k + 1][j];
47                        pre[i][j] = k;
48                    }
49                }
50            }
51        }
52        dfs(0, n - 1);
53        printf("\n");
54    }
55    return 0;
56}

```

7.2.3 CF 149 D

给一个正确匹配的圆括号字符串 $s (2 \leq |s| \leq 700)$, 比如:(), (()), ((())), 要对括号进行染色:

- 对每个括号可以选择不染色, 染红色, 或者染蓝色
- 每对匹配的括号必须有且仅有一个染色
- 相邻染色的括号的颜色不能一样

求最终的所有染色方案数? 结果对 $1e9 + 7$ 取模。

标记状态和递归求解。利用栈的思想可以获取每个括号的匹配括号的位置。用 0, 1, 2 分别表示没被染色、被染成红色和被染成蓝色。

用 $dp[a][b][s][e]$ 表示区间 $[a, b]$ 且 $s[a]$ 状态为 s , $s[b]$ 状态为 e 的所有方案数。

- 当 $s[a]$ 和 $s[b]$ 是匹配括号时，只需要枚举 $s[a]$ 和 $s[b]$ 的所有匹配情况然后递归求解区间 $[a+1, b-1]$ ，但是要保证 $s[a+1]$ 和 $s[a]$ 以及 $s[b-1]$ 和 $s[b]$ 的颜色都不能一样。
- 当 $s[a]$ 和 $s[b]$ 不是匹配括号时，需要找到 $s[a]$ 的匹配括号位置 mid ，然后递归处理区间 $[a, mid], [mid+1, b]$ ，两者相乘即可。

时间复杂度： $O(9 * n^2)$

```

1 const int MAX_N = 710;
2 const ll mod = (ll)(1e9) + 7;
3
4 int n, top;
5 char s[MAX_N];
6 ll dp[MAX_N][MAX_N][3][3];
7 int good[3][3], bad[3][3], match[MAX_N], sta[MAX_N];
8
9 ll solve(int a, int b, int s, int e)
10 {
11     if (dp[a][b][s][e] != -1) return dp[a][b][s][e];
12     if (a + 1 == b) return dp[a][b][s][e] = (good[s][e] ? 1 : 0);
13     ll res = 0;
14     if (match[a] == b) {
15         if (!good[s][e]) return dp[a][b][s][e] = 0;
16         for (int i = 0; i < 3; ++i) {
17             if (bad[s][i]) continue;
18             for (int j = 0; j < 3; ++j) {
19                 if (bad[j][e]) continue;
20                 res = (res + solve(a + 1, b - 1, i, j)) % mod;
21             }
22         }
23     } else {
24         int mid = match[a];
25         for (int i = 0; i < 3; ++i) {
26             for (int j = 0; j < 3; ++j) {
27                 if (bad[i][j]) continue;
28                 res = (res + solve(a, mid, s, i) * solve(mid + 1, b, j, e)) % mod;
29             }
30         }
31     }
32     return dp[a][b][s][e] = res;
33 }
34
35 int main()
36 {
37     good[0][1] = good[1][0] = good[0][2] = good[2][0] = 1;
38     bad[1][1] = bad[2][2] = 1;
39     while (~scanf("%s", s)) {
40         n = strlen(s);
41         top = 0;
42         for (int i = 0; i < n; ++i) {
43             if (s[i] == '(') sta[++top] = i;
44             else {
45                 match[sta[top]] = i;
46                 match[i] = sta[top];
47                 --top;
48             }
49         }
50         memset(dp, -1, sizeof(dp));
51         ll ans = 0;
52         for (int i = 0; i < 3; ++i) {
53             for (int j = 0; j < 3; ++j) {
54                 ans = (ans + solve(0, n - 1, i, j)) % mod;
55             }
56         }
57         printf("%lld\n", ans);
58 }
```

```
59     return 0;  
60 }
```

7.3 树型 dp

7.3.1 树的直径

树中所有最短路径的最大值。

定理：

在一个连通的无项无环图中，以任意结点出发所能到达的最远结点，一定是该图直径的端点之一。

证明：假设直径是 $\delta(s, e)$ ，任意结点为 x ，其最远能到达的结点为 y 。分两种情况：

1. 如果 x 是直径 $\delta(s, e)$ 上的结点，如果 y 不是 s, e 之一，即： $xy > xs, xy > xe$ ，此时满足： $ys = yx + xs > xe + xs = se$ 和 $ye = yx + xe = xs + xe = se$ ，这与直径是 $\delta(s, e)$ 不符。所以 y 必然是 s, e 之一。
2. 如果 x 不是直径 $\delta(s, e)$ 上的结点，设 xy 路径上一结点 z 。如果 z 在直径 $\delta(s, e)$ 上，根据上面的证明可知 y 必然是 s, e 之一。如果 z 不在直径 $\delta(s, e)$ 上，即路径 xy 和直径 $\delta(s, e)$ 完全不相交，这时连接 sx, se ，易得： $sy = sx + xy > sx + xe = se$ ，这又与 $\delta(s, e)$ 是直径相违，所以 y 也必然是直径 s, e 端点之一。

树的直径等于以树直径上任意一点为根的有根树，其左子树的高度 +1，再加上其右子树高度 +1。

7.3.2 求树直径的方法

1. 根据定理 1 可以以任意结点出发找的其最远距离结点，这个结点必然是直径端点之一，再以这个结点出发找到求得其最远距离，可以使用 *dfs* 或 *bfs*。

```

1 void bfs(int u) // bfs 版本，采用链式前向星存边
2 {
3     queue<int> que;
4     vis[u] = 1;
5     que.push(u);
6     while (!que.empty()) {
7         int cur = que.front();
8         que.pop();
9         for (int i = head[cur]; i != -1; i = edge[i].next) {
10            int v = edge[i].to, w = edge[i].w;
11            if (vis[v]) continue;
12            dp[v] = dp[cur] + w;
13            vis[v] = 1;
14            que.push(v);
15        }
16    }
17 }
18 int solve()
19 {
20     memset(vis, 0, sizeof(vis));
21     memset(dp, 0, sizeof(dp));
22     bfs(1);
23     int tmp = 0, st; // st 是直径的一个端点
24     for (int i = 1; i <= n; ++i) {
25         if (dp[i] > tmp) {
26             tmp = dp[i];
27             st = i;
28         }
29     }
30     memset(dp, 0, sizeof(dp));
31     memset(vis, 0, sizeof(vis));
32     bfs(st);
33     tmp = 0;
34     for (int i = 1; i <= n; ++i) {
35         if (dp[i] > tmp) tmp = dp[i];
36     }
37     return tmp; // 树的直径
38 }
```

```

1 void dfs(int u) // dfs 版本
2 {
3     vis[u] = 1;
```

```

4     for (int i = head[u]; i != -1; i = edge[i].next) {
5         int v = edge[i].to, w = edge[i].w;
6         if (vis[v]) continue;
7         dp[v] = dp[u] + w;
8         dfs(v);
9     }
10 }
```

顺便提一下通过 dfs 的方式同时获得直径上的边，注意建双向边。

在 dfs 的第一行加一句： $pre[u] = p$ 。然后从直径的结尾处再次 dfs：调用 $dfs2(ed)$ 。

```

1 void dfs2(int u)
2 {
3     if (pre[u] == 0) return; // 注意把直径的开始处: pre[st]=0
4     for (int i = head[u]; i != -1; i = edge[i].next) {
5         int v = edge[i].v;
6         if (v == pre[u]) {
7             if (i & 1) edge[i - 1].flag = true; // 双向边, 标记从 st 到 ed 方向时的边
8             else edge[i + 1].flag = true;
9             num++; // 直径中边的数量
10            dfs2(v);
11        }
12    }
13 }
```

2. 另外一种求树的直径的方法。

我们可以想办法求出每个节点到其他节点的最远距离。这个最远距离可能来自节点的子树也可能来自节点的父亲节点。我们先用两个数组 $far[]$ 和 $ffar[]$ 保存节点通过子树可以得到的最远距离和次远距离，并用 $id[]$ 保存获得最远距离时该子树的根节点编号。假设节点 v 的父亲节点是 u ，对于 v 从 u 获得的最远距离 $father[v]$ ，如果 $id[u] = v$ ，那么只需要考虑 $father[v] = \max(father[u], ffar[u]) + w$ ， w 是 u 和 v 的距离。如果 $id[u] \neq v$ ，那么 $father[v] = \max(father[u], far[u]) + w$ 。两次处理都需要 dfs 。

```

1 //far[] 最远子树距离, ffar[] 次远子树距离, father[] 通过父亲节点获得的最远距离
2 // id[] 从子树获得最远距离时该子树的根节点编号
3
4 void dfs1(int u, int p) //p 是 u 的父亲节点
5 { //从子树获得最远距离和次远距离
6     if (far[u] != -1) return ;
7     far[u] = ffar[u] = 0;
8     // 找子树最远距离
9     for (int i = head[u]; i != -1; i = edge[i].next) {
10         int v = edge[i].to, w = edge[i].w;
11         if (v == p) continue; //只能从 u 的子树获得, 不能返回父亲节点
12         dfs1(v, u);
13         if (w + far[v] > far[u]) {
14             far[u] = far[v] + w;
15             id[u] = v;
16         }
17     }
18     // 找子树次远距离
19     for (int i = head[u]; i != -1; i = edge[i].next) {
20         int v = edge[i].to, w = edge[i].w;
21         if (v == p || v == id[u]) continue; //不能是父亲节点和最远距离节点
22         if (w + far[v] > ffar[u]) {
23             ffar[u] = far[v] + w;
24         }
25     }
26 }
27
28 void dfs2(int u, int p) //p 是 u 的父亲节点
29 { //从父亲节点获得最远距离
30     for (int i = head[u]; i != -1; i = edge[i].next) {
31         int v = edge[i].to, w = edge[i].w;
32         if (v == p) continue;
33         if (v == id[u]) { //如果 v 是 u 最远距离时的子树根节点
```

```

34         father[v] = max(father[u], ffar[u]) + w;
35     } else {
36         father[v] = max(father[u], far[u]) + w;
37     }
38     dfs2(v, u);
39 }
40 }
41 //主函数部分
42 memset(far, -1, sizeof(far));
43 memset(ffar, -1, sizeof(ffar));
44 memset(father, -1, sizeof(father));
45 dfs1(1, 0); //对每个点求其到子树上节点的最远距离和次远距离
46 father[1] = 0;
47 dfs2(1, 0); //对每个点求其经过父节点可到达的最远距离
48 int diameter = 0;
49 for (int i = 1; i <= n; ++i) {
50     longest[i] = max(father[i], far[i]); // longest[i] 时节点可以到达的最远距离 i
51     if (longest[i] > diameter) diameter = longest[i];
52     // printf("longest[%d] = %d\n", i, longest[i]);
53 }
54 printf("%d\n", diameter);
55

```

7.3.3 HDU 4126

给 n 个点和 m 条无向边，无重边。需要将这 n 个点连通。但是有 Q 次破坏，每次破坏会把 m 条边中的某条边的权值增大，求 Q 次破坏每次将 n 个点连通的代价的期望？（每次破坏后的最小生成树的代价累加除以 Q ）。 $n \leq 3000, Q \leq 10^4$ ，单边权 $\leq 10^7$ ，总边权 $\leq 10^9$

先求一遍最小生成树，设代价为 sum 。如果破坏的边不是最小生成树的边，那么这次的代价就是 sum 。如果破坏的边是最小生成树的边，根据这条边的两个端点把这棵树分成两个点集，设将这两个点集连通的最小代价为 tmp （也就是一个集合中的所有点到另一个集合中的所有点的最小边权），那么这次破坏的后生成树的代价为 $sum - dis[u][v] + min(tmp, w)$ ，其中 u 和 v 是边的端点， $dis[u][v]$ 是原有的边权， w 是破坏后的边权。

因为破坏的次数 Q 有 10^4 次，需要快速得得到 tmp 。记 $dp[u][v]$ 为破坏边 (u, v) 之后以 u 为根的子树和以 v 为根的子树两点最小权值。考虑每个节点 $root$ 对其他所有节点为根的所有子树与包含 u 的子树的 $dp[]$ 影响。例如当前考虑截断节点 u 和 u 的儿子 v 之间的边（最小生成树中的边），也就是考虑 $dp[u][v]$ 并且现在只考虑 $root$ 的影响，假设此时 $root$ 是在以 u 为根的子树中的，先递归解决 v 的所有儿子 vv ，然后在所有的 $dis[vv][root]$ 和 $dp[u][vv]$ 中取最小值即可。如果枚举每一个 $root$ ，那么就会把 u 和 v 中的所有点都用上了，而且每次枚举的复杂度都是 $O(n)$ 的，所以处理 $dp[]$ 数组的总的时间复杂度是： $O(n^2)$ ，每次破坏时只需要 $O(1)$ 的判断。

求最小生成树时如果用 *Kruskal*，就是 $O(m \log m)$ ，如果用 *Prim*，就是 $O(n^2)$ ，所以最终的时间复杂度是： $O(m \log m + n^2 + Q)$ 或者 $O(2 * n^2 + Q)$ 。

```

1 const int MAX_N = 3010;
2 const int inf = 0x3f3f3f3f;
3
4 int n, m, Q;
5 int dis[MAX_N][MAX_N], used[MAX_N][MAX_N], dp[MAX_N][MAX_N];
6 int way[MAX_N], vis[MAX_N], fa[MAX_N];
7 vector<int> vec[MAX_N];
8
9 void init()
10 {
11     for (int i = 0; i < n; ++i) {
12         vec[i].clear();
13         vis[i] = 0;
14         for (int j = 0; j < n; ++j) {
15             used[i][j] = 0;
16             dp[i][j] = dis[i][j] = inf;
17         }
18     }
}

```

```

19 }
20
21 int Prim()
22 {
23     int res = 0;
24     vis[0] = 1;
25     for (int i = 0; i < n; ++i) {
26         way[i] = dis[i][0];
27         fa[i] = 0;
28     }
29     vis[0] = 1, fa[0] = -1;
30     for (int i = 1; i < n; ++i) {
31         int Min = inf, id;
32         for (int j = 0; j < n; ++j) {
33             if (!vis[j] && way[j] < Min) {
34                 Min = way[j], id = j;
35             }
36         }
37         vis[id] = 1;
38         res += Min;
39         if (fa[id] != -1) {
40             vec[id].push_back(fa[id]);
41             vec[fa[id]].push_back(id);
42         }
43         for (int j = 0; j < n; ++j) {
44             if (!vis[j] && dis[id][j] < way[j]) {
45                 way[j] = dis[id][j];
46                 fa[j] = id;
47             }
48         }
49     }
50     return res;
51 }
52
53 int dfs(int root, int u, int p)
54 {
55     int Min = inf;
56     for (int i = 0; i < vec[u].size(); ++i) {
57         int v = vec[u][i];
58         if (v == p) continue;
59         int tmp = dfs(root, v, u);
60         Min = min(Min, tmp);
61         dp[u][v] = dp[v][u] = min(dp[u][v], tmp);
62     }
63     if (p != -1 && p != root) Min = min(Min, dis[root][u]);
64     return Min;
65 }
66
67 int main()
68 {
69     while (~scanf("%d%d", &n, &m) && (n || m)) {
70         init();
71         for (int i = 0; i < m; ++i) {
72             int u, v, w;
73             scanf("%d%d%d", &u, &v, &w);
74             dis[u][v] = dis[v][u] = w;
75         }
76         int sum = Prim();
77         for (int i = 0; i < n; ++i) dfs(i, i, -1);
78         scanf("%d", &Q);
79         double ans = 0;
80         for (int i = 0; i < Q; ++i) {
81             int u, v, w;
82             scanf("%d%d%d", &u, &v, &w);
83             if (fa[u] != v && fa[v] != u) ans += sum;

```

```
84         else ans += sum - dis[u][v] + min(w, dp[u][v]);  
85     }  
86     printf("%.4lf\n", ans / (1.0 * Q));  
87 }  
88 return 0;  
89 }
```

7.4 数位 dp

7.4.1 2014 GCJ Round 1B

求 $x \leq A, y \leq B$, 并且 $x \& y$ 值小于等于 K 的数字个数。数据范围: $A, B, K \leq 10^9$ 。

一般的数位 dp 在 dfs 的时候只记录一个 $limit$ 表示是否达到区间上限，这里需要记录三个，因为 x, y 和二进制与出来的值都有上限。

```

1 typedef long long ll;
2
3 int T, cases = 0;
4 ll A, B, K;
5 ll vis[35][3][3][3];
6
7 ll solve(int cur, int lessA, int lessB, int lessK)
8 {
9     if (cur == -1) return lessA && lessB && lessK;
10    if (vis[cur][lessA][lessB][lessK] != -1) return vis[cur][lessA][lessB][lessK];
11    int MaxA = lessA || ((A >> cur) & 1);
12    int MaxB = lessB || ((B >> cur) & 1);
13    int MaxK = lessK || ((K >> cur) & 1);
14    ll ret = solve(cur - 1, MaxA, MaxB, MaxK); // 0 & 0 = 0
15    if (MaxA) ret += solve(cur - 1, lessA, MaxB, MaxK); // 0 & 1 = 0
16    if (MaxB) ret += solve(cur - 1, MaxA, lessB, MaxK); // 1 & 0 = 0
17    if (MaxA && MaxB && MaxK) ret += solve(cur - 1, lessA, lessB, lessK); // 0 & 0 = 0
18    return vis[cur][lessA][lessB][lessK] = ret;
19 }
20
21 int main()
22 {
23     scanf("%d", &T);
24     while (T--) {
25         scanf("%lld%lld%lld", &A, &B, &K);
26         memset(vis, -1, sizeof(vis));
27         printf("Case # %d: %lld\n", ++cases, solve(31, 0, 0, 0));
28     }
29     return 0;
30 }
```

7.4.2 区间最大上升子序列长度为 K 的数字个数

HDU 4352 XHXJ's LIS

把一个数字从左到右（从高位到低位）看成一个序列，求区间 $[L, R]$ 内序列的最大上升子序列长度为 K 的数字个数。数据范围: $0 < L \leq R < 2^{63} - 1, 1 \leq K \leq 10$

区间减法。把高位数字状压成最多是 111111111(9 个 1) 的二进制数，然后按照求最大上升子序列的方法更新判断即可。

```

1 // 调用 dfs(len - 1, )
2 int T, cases = 0, K, digit[20];
3 ll L, R, dp[20][1100][10];
4
5 ll dfs(int pos, int state, int len, int first, int limit)
6 { // len: 已有长度, state: 状压, first: 前导0
7     if (pos == -1) return len == K;
8     if (len > K) return 0;
9     if (!limit && dp[pos][state][K] != -1) return dp[pos][state][K];
10    int last = limit ? digit[pos] : 9;
11    ll ret = 0;
12    for (int i = 0; i <= last; ++i) {
13        int next_state = state, next_len = len;
14        if ((i != 0 || (!first && i == 0)) && state < (1 << i)) { // i 是最大数字
```

```

15     next_state += (1 << i);
16     next_len++;
17 } else if (((state >> i) & 1) == 0){
18     int k = i + 1;
19     while (k != 10 && ((state >> k) & 1) == 0) ++k; // 使子序列潜力变大
20     if (k != 10) {
21         next_state = next_state - (1 << k) + (1 << i);
22     }
23 }
24     ret += dfs(pos - 1, next_state, next_len, first && (i == 0), limit && (i == last));
25 }
26 if (!limit) dp[pos][state][K] = ret;
27 return ret;
28 }

29 ll solve(ll x)
30 {
31     if (x == 0) return 0;
32     memset(digit, 0, sizeof(digit));
33     int len = 0;
34     while (x) {
35         digit[len++] = x % 10;
36         x /= 10;
37     }
38     return dfs(len - 1, 0, 0, 1, 1);
39 }
40

41 int main()
42 {
43     memset(dp, -1, sizeof(dp));
44     scanf("%d", &T);
45     while (T--) {
46         scanf("%lld%lld%d", &L, &R, &K);
47         printf("Case #%d: %lld\n", ++cases, solve(R) - solve(L - 1));
48     }
49     return 0;
50 }
51

```

7.4.3 区间所有和 7 “无关” 数字平方和

HDU 4570

定义和 7 有关的数字是满足下列条件之一的数字：

- 整数中某一位是 7；
- 整数的每一位加起来的和是 7 的整数倍；
- 这个整数是 7 的整数倍；

给一个区间 $[L, R]$ ($1 \leq L \leq R \leq 10^{18}$)，求区间内所有和 7 无关的数字的平方和。对 $1e9 + 7$ 取模。
满足区间减法。要对 ‘dfs’ 的返回结果（低位的情况已经解决）记录三个属性（用结构体保存）：

- cnt : 和 7 无关的数字个数
- sum : 和 7 无关的所有数字之和
- $sqsum$: 和 7 无关的所有数字的平方和

记当前答案为 ret ， dfs 返回答案为： nxt ，当前是 pos 位并且枚举的数字是 i ， $pw10[p]$ 表示 10^p 。考虑更新。

- $ret.cnt += nxt.cnt;$
- $ret.sum += nxt.sum + i * pw10[p] * nxt.cnt;$

- *ret.sqsum*

现在我们考虑 pos 位数: $i * 10^{pos} + x$, 平方得: $(i^2 * 10^{2*pos} + x^2 + 2 * 10^{pos} * i * x)$, 但是 x 可能有很多个。所以得:

$$\sum (i^2 * 10^{2*pos} + x^2 + 2 * 10^{pos} * i * x), \text{ 其中 } x \text{ 是所有低位符合条件的数, 个数是 } nxt.cnt \text{ 个}$$

$$ret.sqsum += \sum (i^2 * 10^{2*pos} * nxt.cnt)$$

$$ret.sqsum += \sum x^2 = nxt.sqsum$$

$$ret.sqsum += \sum 2 * 10^{pos} * i * x = 2 * 10^{pos} * i * \sum x = 2 * 10^{pos} * i * nxt.sum$$

对于 dfs 的函数参数我们需要存的是高位数字和和高位数字对 7 取模后的结果, 这样子在 dfs 的最后一层来判断是否和 7 有关。

```

1 int T, digit[20], vis[20][10][10];
2 ll L, R, pw10[20];
3
4 struct Node {
5     ll cnt, sum, sqsum;
6
7     Node() {}
8     Node(ll _cnt, ll _sum, ll _sqsum) : cnt(_cnt), sum(_sum), sqsum(_sqsum) {}
```

```
9 } dp[20][10][10];
```

```
10
11 Node dfs(int pos, int sum_rem, int num_rem, int limit)
12 { //: sum_rem 高位数字和对 7 取模余数 num_rem: 高位数字对 7 取模余数
13     if (pos == -1) {
14         if (sum_rem == 0 || num_rem == 0) return Node(0, 0, 0);
15         else return Node(1, 0, 0);
16     }
17     if (!limit && vis[pos][sum_rem][num_rem]) return dp[pos][sum_rem][num_rem];
18     int last = limit ? digit[pos] : 9;
19     Node ret = Node(0, 0, 0);
20     for (int i = 0; i <= last; ++i) {
21         if (i == 7) continue;
22         Node nxt = dfs(pos - 1, (sum_rem + i) % 7,
23                         (num_rem * 10 + i) % 7, limit && (i == last));
24         ret.cnt = (ret.cnt + nxt.cnt) % mod;
25         ret.sum = ((ret.sum + pw10[pos] * i % mod * nxt.cnt
26                     % mod) % mod + nxt.sum) % mod;
27         ret.sqsum = (ret.sqsum + nxt.sqsum) % mod;
28         ret.sqsum = (ret.sqsum + pw10[pos] * pw10[pos] % mod
29                         * i * i % mod * nxt.cnt % mod) % mod;
30         ret.sqsum = (ret.sqsum + pw10[pos] * 2 * i % mod
31                         * nxt.sum % mod) % mod;
32     }
33     if (!limit) {
34         vis[pos][sum_rem][num_rem] = 1;
35         dp[pos][sum_rem][num_rem] = ret;
36     }
37     return ret;
38 }
```

```
39
40 ll solve(ll x)
41 {
42     memset(digit, 0, sizeof(digit));
43     int len = 0;
44     while (x) {
45         digit[len++] = x % 10;
46         x /= 10;
47     }
48     return dfs(len - 1, 0, 0, 1).sqsum;
49 }
```

```

51 int main()
52 {
53     pw10[0] = 1;
54     for (int i = 1; i <= 18; ++i) { pw10[i] = pw10[i - 1] * 10 % mod; }
55     memset(vis, 0, sizeof(vis));
56     scanf("%d", &T);
57     while (T--) {
58         scanf("%lld%lld", &L, &R);
59         ll ans = (solve(R) - solve(L - 1)) % mod;
60         printf("%lld\n", (ans + mod) % mod);
61     }
62     return 0;
63 }
```

7.4.4 SGU 390 Tickets

有一位售票员给乘客售票。对于每位乘客，他会卖出多张连续的票，直到已卖出的票的编号的数位之和不小于给定的正数 k 。然后他会按照相同的规则给下一位乘客售票。初始时，售票员持有的票的编号是从 L 到 R 的连续整数。请你求出，售票员可以售票给多少位乘客。数据规模： $1 \leq L \leq R \leq 10^{18}$, $1 \leq k \leq 1000$ 。

这个不能区间减法做了，需要在每一步时判断上下限。

把一个十进制数的区间看成一个十叉树，每个分支代表一个数字。用 $dp[pos][left][cur]$ 表示枚举到第 pos 位，前一颗十叉树剩下数字和为 $left$ ，且当前的十叉树已得到的数字和为 cur 时低位数字随便选（非上下限情况）时构成数字和为 K 时可以卖出去的票数。

递归的终止 $pos = -1$ 时，判断 $left + cur$ 和 K 的大小关系来决定能否在当前数字卖出去票。

```

1 int K, n, m, digitL[20], digitR[20], vis[20][200][1010];
2 ll L, R;
3 pair<ll, int> dp[20][200][1010];
4
5 pair<ll, int> dfs(int pos, int left, int cur, int limitL, int limitR)
6 {
7     if (pos == -1) {
8         if (left + cur >= K) return make_pair(1, 0);
9         else return make_pair(0, left + cur);
10    }
11    if (!limitL && !limitR && vis[pos][cur][left]) return dp[pos][cur][left];
12    int low = limitL ? digitL[pos] : 0;
13    int high = limitR ? digitR[pos] : 9;
14    pair<ll, int> ret = make_pair(0, left), tmp;
15    for (int i = low; i <= high; ++i) {
16        tmp = dfs(pos - 1, ret.second, cur + i,
17                   limitL && (i == low), limitR && (i == high));
18        ret.first += tmp.first, ret.second = tmp.second;
19    }
20    if (!limitL && !limitR) {
21        vis[pos][cur][left] = 1;
22        dp[pos][cur][left] = ret;
23    }
24    return ret;
25 }
26
27 int main()
28 {
29     while (~scanf("%lld%lld%d", &L, &R, &K)) {
30         memset(vis, 0, sizeof(vis));
31         memset(digitL, 0, sizeof(digitL));
32         memset(digitR, 0, sizeof(digitR));
33         n = m = 0;
34         while (L) {
35             digitL[n++] = L % 10;
```

```
36     L /= 10;
37 }
38     while (R) {
39         digitR[m++] = R % 10;
40         R /= 10;
41     }
42     printf("%lld\n", dfs(m - 1, 0, 0, 1, 1).first);
43 }
44 return 0;
45 }
```

7.5 状压 dp

7.5.1 覆盖模型

[POJ 2411]

给出 $n * m$ ($1 \leq n, m \leq 11$) 的方格棋盘，用 $1 * 2$ 的长方形骨牌不重叠地覆盖这个棋盘，求覆盖满的方案数。

显然，如果 n, m 都是奇数则无解（由棋盘面积的奇偶性知），否则必然有至少一个解（很容易构造出），所以假设 n, m 至少有一个偶数，且 $m \leq n$ （否则交换）。把每行的放置方案 DFS 出来，逐行计算。用 $dp[i][s]$ 表示把前 $i - 1$ 行覆盖满、第 i 行覆盖状态为 s 的覆盖方案数。因为在第 i 行上放置的骨牌最多也只能影响到第 $i - 1$ 行，则容易得递推式：

$$\begin{aligned} dp[0][2^m - 1] &= 1 \\ dp[i][s_1] &= \sum dp[i - 1][s_2], (s_1, s_2) \text{ 整体作为放置方案} \end{aligned}$$

首先讨论 DFS 的一些细节。对于当前行每一个位置，我们有 3 种放置方法：

- 竖直覆盖，占据当前格和上一行同一列的格；
- 水平覆盖，占据当前格和该行下一格；
- 不放置骨牌，直接空格。

枚举出每个 (s_1, s_2) 的两种方法：

第一种：

DFS 共 5 个参数，分别为： p （当前列号）， s_1, s_2 （当前行和上一行的覆盖情况）， b_1, b_2 （上一列的放置对当前列两行的影响，影响为 1 否则为 0）。初始时 $s_1 = s_2 = b_1 = b_2 = 0$ 。

1. $p = p + 1, s_1 = s_1 * 2 + 1, s_2 = s_2 * 2 + 1, b_1 = b_2 = 0;$
2. $p = p + 1, s_1 = s_1 * 2 + 1, s_2 = s_2 * 2 + 1, b_1 = 1, b_2 = 0;$
3. $p = p + 1, s_1 = s_1 * 2, s_2 = s_2 * 2 + 1, b_1 = b_2 = 0.$

当 p 移出边界且 $b_1 = b_2 = 0$ 时记录此方案。

第二种：

观察第一种方法，发现 b_2 始终为 0，知这种方法有一定的冗余。换个更自然的方法，去掉参数 b_1, b_2 。

1. $p = p + 1, s_1 = s_1 * 2 + 1, s_2 = s_2 * 2;$
2. $p = p + 2, s_1 = s_1 * 4 + 3, s_2 = s_2 * 4 + 3;$
3. $p = p + 1, s_1 = s_1 * 2, s_2 = s_2 * 2 + 1.$

当 p 移出边界时，记录此方案。这样，我们通过改变 p 的移动距离成功简化了 DFS 过程，而且这种方法更自然。

每一位 0 和 1 的含义。对于枚举到第 i 行，第 i 行允许一些位置是空的，相当于保存一些状态，然后由第 $i + 1$ 行来填上。第 $i - 1$ 行的所有位置一定是要放置棋子的，因为要铺满整个棋盘。如果在第 i 行放置的棋子影响到了第 $i - 1$ 行，那么第 $i - 1$ 行的相应位（相当于 s_2 ）就为 0，如果没有影响到第 $i - 1$ 行，就为 1。就拿第二种方法的第三个方案来说，就是第 i 行的这一列不放置棋子，此时自然无法影响到第 $i - 1$ 行，所以有 $s_1 = s_1 * 2, s_2 = s_2 * 2 + 1$ ，而对于第一种方案，是第 i 行这一列放置了竖放的棋子，影响到了第 $i - 1$ 行，所以第 i 行的这一位上为 1，而第 $i - 1$ 行的这一位为 0，对于第二种方案也是同样的道理：第 i 行横放棋子，占据了两列，所以第 $i - 1$ 行的这两列都没有受到第 i 行的影响。

可以使用滚动数组优化空间消耗，

```

1 const int MAX_N = 13;
2 int n, m;
3 ll dp[MAX_N][1 << MAX_N], ans[MAX_N][MAX_N];
4
5 void dfs(int row, int col, int s1, int b1, int s2, int b2)
6 // void dfs(int row, int col, int s1, int s2)
7 {
8     if (col > m) return ;
9     if (col == m) {

```

```

10     if (b1 == 0 && b2 == 0) dp[row][s1] += dp[row - 1][s2];
11     // dp[row][s1] += dp[row - 1][s2];
12     return;
13 }
14
15 if (b1 == 0 && b2 == 0) {
16     dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1, 0);
17     dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1 | 1, 0);
18     dfs(row, col + 1, s1 << 1, 0, s2 << 1 | 1, 0);
19 } else if (b1 == 1 && b2 == 0) {
20     dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1 | 1, 0);
21 }
22 /*
23 dfs(row, col + 1, s1 << 1, s2 << 1 | 1);
24 dfs(row, col + 1, s1 << 1 | 1, s2 << 1);
25 dfs(row, col + 2, s1 << 2 | 3, s2 << 2 | 3);
26 */
27 }
28
29
30 ll solve()
31 {
32     memset(dp, 0, sizeof(dp));
33     dp[0][(1 << m) - 1] = 1;
34     for (int i = 1; i <= n; ++i) {
35         dfs(i, 0, 0, 0, 0, 0);
36         // dfs(i, 0, 0, 0);
37     }
38     return dp[n][(1 << m) - 1];
39 }
40
41 int main()
42 {
43     memset(ans, -1, sizeof(ans));
44     while (~scanf("%d%d", &n, &m) && (n || m)) {
45         // double begin = clock();
46         if (m > n) swap(n, m);
47         if (ans[n][m] != -1) printf("%lld\n", ans[n][m]);
48         else if (n * m % 2) printf("0\n");
49         else {
50             ans[n][m] = ans[m][n] = solve();
51             printf("%lld\n", ans[n][m]);
52         }
53         // printf("time used: %.3lf\n", (clock() - begin) / CLK_TCK);
54     }
55     return 0;
56 }
```

[PKU 3420]: 给一个 $4 * n (n \leq 10^9)$ 的棋盘, 用 $1 * 2$ 的棋子覆盖满整个棋盘, 求方案数, 对 $m \leq 10^5$ 取模。

此算法中应用到一个结论: 给出一个图的 0/1 邻接矩阵 G (允许有自环, 两点间允许有多条路径, 此时 $G_{i,j}$ 表示 i 到 j 的边的条数), 则从某点 i 走 k 步到某点 j 的路径数为 $G_{i,j}^k$ 。本结论实际上是通过递推得到的, 简单证明如下: 从 i 走 k 步到 j , 必然是从 i 走 $k-1$ 步到 m , 然后从 m 走 1 步到 j , 根据加法原理, 即 $G_{i,j}^k = \sum G_{i,m}^{k-1} * G_{m,j}$ 。它和矩阵乘法一模一样, 即: $G_k = G_{k-1} * G$ 。用矩阵快速幂加速就好了。

下面介绍这个算法。考虑一个有 2^m 个顶点的图, 每个顶点表示一行的覆盖状态, 即 SCR 算法中的 s 。如果 $(s1, s2)$ 为一个放置方案, 则在 $s2$ 和 $s1$ 之间连一条(有向)边, 则我们通过 DFS 一次可以得到一个邻接矩阵 G 。仍然按照逐行放置的思想来考虑, 则要求我们每行选择一个覆盖状态, 且相邻两行的覆盖状态 $(s1, s2)$ 应为一个放置方案, 一共有 n 行, 则要求选择 n 个状态, 在图中考虑, 则要求我们从初始(第 0 行)顶点 $(2^m - 1)$ 走 n 步到 $(2^m - 1)$ (顶点即是状态), 因为图的邻接矩阵是 DFS 得出来的, 每条边都对应一个放置方案, 所以可以保证走的每条边都合法。因此, 我们要求的就是顶点 $(2^m - 1)$ 走 n 步到达 $(2^m - 1)$ 的路径条数。由上面的结论知, 本题的答案就是 $G_{2^m - 1, 2^m - 1}^n$, 其中 $m = 4$ 。

下面的代码都是更具普遍性的写法, 也就是 $n * m$ 的棋盘用 $1 * 2$ 的棋子去覆盖的方案数。

```

1 const int MAX_N = 260;
2 const ll mod = (ll)(1e9 + 7);
3 const int MAX_M = 11;
4
5 struct Matrix {
6     int row, col;
7     int data[MAX_N][MAX_N];
8
9     Matrix operator * (const Matrix& rhs) const {
10         Matrix ret;
11         ret.row = row, ret.col = rhs.col;
12         memset(ret.data, 0, sizeof (ret.data));
13         for (int i = 0; i < ret.row; ++i) {
14             for (int j = 0; j < ret.col; ++j) {
15                 for (int k = 0; k < col; ++k) {
16                     ret.data[i][j] += 111 * data[i][k] * rhs.data[k][j] % mod;
17                     if (ret.data[i][j] >= mod) ret.data[i][j] -= mod;
18                 }
19             }
20         }
21         return ret;
22     }
23
24     Matrix operator ^ (const int& len) const {
25         Matrix ret, tmp;
26         int exp = len;
27         ret.row = ret.col = tmp.row = tmp.col = row;
28         memset(ret.data, 0, sizeof (ret.data));
29         for (int i = 0; i < row; ++i) { ret.data[i][i] = 1; }
30         for (int i = 0; i < row; ++i) {
31             for (int j = 0; j < col; ++j) {
32                 tmp.data[i][j] = data[i][j];
33             }
34         }
35         while (exp) {
36             if (exp & 1) ret = ret * tmp;
37             tmp = tmp * tmp;
38             exp >>= 1;
39         }
40         return ret;
41     }
42 };
43
44 int n, m;
45 int link[MAX_N][MAX_N];
46
47 void dfs(int col, int s1, int s2)
48 {
49     if (col > m) return;
50     if (col == m) {
51         link[s1][s2] = 1;
52         return;
53     }
54     dfs(col + 1, (s1 << 1) + 1, s2 << 1);
55     dfs(col + 1, s1 << 1, (s2 << 1) + 1);
56     dfs(col + 2, (s1 << 2) | 3, (s2 << 2) | 3);
57 }
58
59 int solve()
60 {
61     memset(link, 0, sizeof (link));
62     dfs(0, 0, 0);
63     Matrix ret;
64     ret.row = ret.col = (1 << m);
65     for (int i = 0; i < ret.row; ++i) {

```

```

66     for (int j = 0; j < ret.col; ++j) {
67         ret.data[i][j] = link[i][j];
68     }
69 }
70 ret = ret ^ n;
71 return ret.data[(1 << m) - 1][(1 << m) - 1];
72 }
73
74 int main()
75 {
76     while (~scanf ("%d%d", &n, &m)) {
77         if (m > n) swap(n, m);
78         double begin = clock();
79         if (n * m % 2) printf("0\n");
80         else printf("%d\n", solve());
81         printf("time used: %.3lfs\n", (clock() - begin) / CLK_TCK);
82     }
83     return 0;
84 }
```

这样对于 $m \leq 7$ 就可以很快出解了。但对于 $m = n = 8$, 上述算法都需要 1s 才能出解, 无法令人满意。此算法还有优化空间。

矩阵规模高达 $2^m * 2^m = 4^m$, 但是其中有用的(非 0 的)其实是很少的, 这是一个非常非常稀疏的矩阵, 使用三次方的矩阵乘法有点大材小用。我们改变矩阵的存储结构, 即第 p 行第 q 列的值为 value 的元素可以用一个三元组 $(p, q, value)$ 来表示, 采用一个线性表依行列顺序来存储这些非 0 元素。怎样对这样的矩阵进行乘法呢? 观察矩阵乘法的计算式 $c[i][j] = \sum a[i][k] * b[k][j]$ 当 $a[i][k]$ 或者 $b[k][j]$ 为 0 时, 结果为 0, 对结果没有影响, 完全可以略去这种没有意义的运算。则得到计算稀疏矩阵乘法的算法: 枚举 a 中的非 0 元素, 设为 $(p, q, v1)$ 在 b 中寻找所有行号为 q 的非 0 元素 $(q, r, v2)$, 并把 $v1 * v2$ 的值累加到 $c[p][r]$ 中。这个算法多次用到一个操作: 找出所有行号为 q 的元素。我是使用链式前向星存储的。比较难打, 可以看代码。

时间复杂度: $O(\log n * 4^m)$

```

1 const int MAX_M = 11;
2 const int LIMIT = 1 << 11;
3
4 struct Edge {
5     int nxt;
6 } edge[LIMIT * LIMIT];
7
8 int n, m, total_a, total_b, total_edge;
9 int head[LIMIT], link[LIMIT][LIMIT], r[LIMIT][LIMIT];
10
11 struct Value {
12     int x, y, v;
13 } value_a[LIMIT * LIMIT], value_b[LIMIT * LIMIT];
14
15 void AddEdge(int id)
16 {
17     edge[total_edge].nxt = head[id];
18     head[id] = total_edge++;
19 }
20
21 void Multiple(int (&a)[LIMIT][LIMIT], int b[LIMIT][LIMIT])
22 {
23     memset(head, -1, sizeof(head));
24     total_edge = total_a = total_b = 0;
25     for (int i = 0; i < (1 << m); ++i) {
26         for (int j = 0; j < (1 << m); ++j) {
27             if (a[i][j]) {
28                 value_a[total_a].x = i, value_a[total_a].y = j;
29                 value_a[total_a++].v = a[i][j];
30             }
31             if (b[i][j]) {
32                 value_b[total_b].x = i, value_b[total_b].y = j;
33                 value_b[total_b++].v = b[i][j];
34             }
35         }
36     }
37 }
```

```

34         AddEdge(i);
35     }
36 }
37 }
38 memset(a, 0, sizeof (a));
39 for (int i = 0; i < total_a; ++i) {
40     int xa = value_a[i].x, ya = value_a[i].y, va = value_a[i].v;
41     for (int j = head[ya]; j != -1; j = edge[j].nxt) {
42         int xb = value_b[j].x, yb = value_b[j].y, vb = value_b[j].v;
43         a[xa][yb] += (int)(111 * va * vb % mod);
44         if (a[xa][yb] >= mod) a[xa][yb] -= mod;
45     }
46 }
47 }
48
49 void dfs(int col, int s1, int s2)
50 {
51     if (col > m) return;
52     if (col == m) {
53         link[s1][s2] = 1;
54         return;
55     }
56     dfs(col + 1, (s1 << 1) | 1, s2 << 1);
57     dfs(col + 1, s1 << 1, (s2 << 1) | 1);
58     dfs(col + 2, (s1 << 2) | 3, (s2 << 2) | 3);
59 }
60
61 int solve()
62 {
63     memset(link, 0, sizeof (link));
64     dfs(0, 0, 0);
65     memset(r, 0, sizeof (r));
66     for (int i = 0; i < (1 << m); ++i) { r[i][i] = 1; }
67     while (n) {
68         if (n & 1) Multiple(r, link);
69         Multiple(link, link);
70         n >>= 1;
71     }
72     return r[(1 << m) - 1][(1 << m) - 1];
73 }
74
75 int main()
76 {
77     while (~scanf ("%d%d", &m, &m)) {
78         double begin = clock();
79         if (m > n) swap(n, m);
80         if (n * m % 2) printf("0\n");
81         else printf("%d\n", solve());
82         // printf("time used: %.3lf\n", (clock() - begin) / CLK_TCK);
83     }
84     return 0;
85 }

```

[SGU 131]

给出 $n * m (1 \leq n, m \leq 9)$ 的方格棋盘，用 $1 * 2$ 的矩形的骨牌和 L 形的 ($2 * 2$ 的去掉一个角) 骨牌不重叠地覆盖，求覆盖满的方案数。

例 1 中有两种 DFS 方案，其中第二种实现起来较第一种简单。但在本题中，新增的 L 形骨牌让第二种 DFS 难以实现，在例 1 中看起来有些笨拙的第一种 DFS 方案在本题却可以派上用场。回顾第一种 DFS，我们有 5 个参数，分别为： p (当前列号)， $s1, s2$ (当前行和对应的上一行的覆盖情况)， $b1, b2$ (上一列的放置对当前列两行的影响，影响为 1 否则为 0)。本题中可选择的方案增多。

```

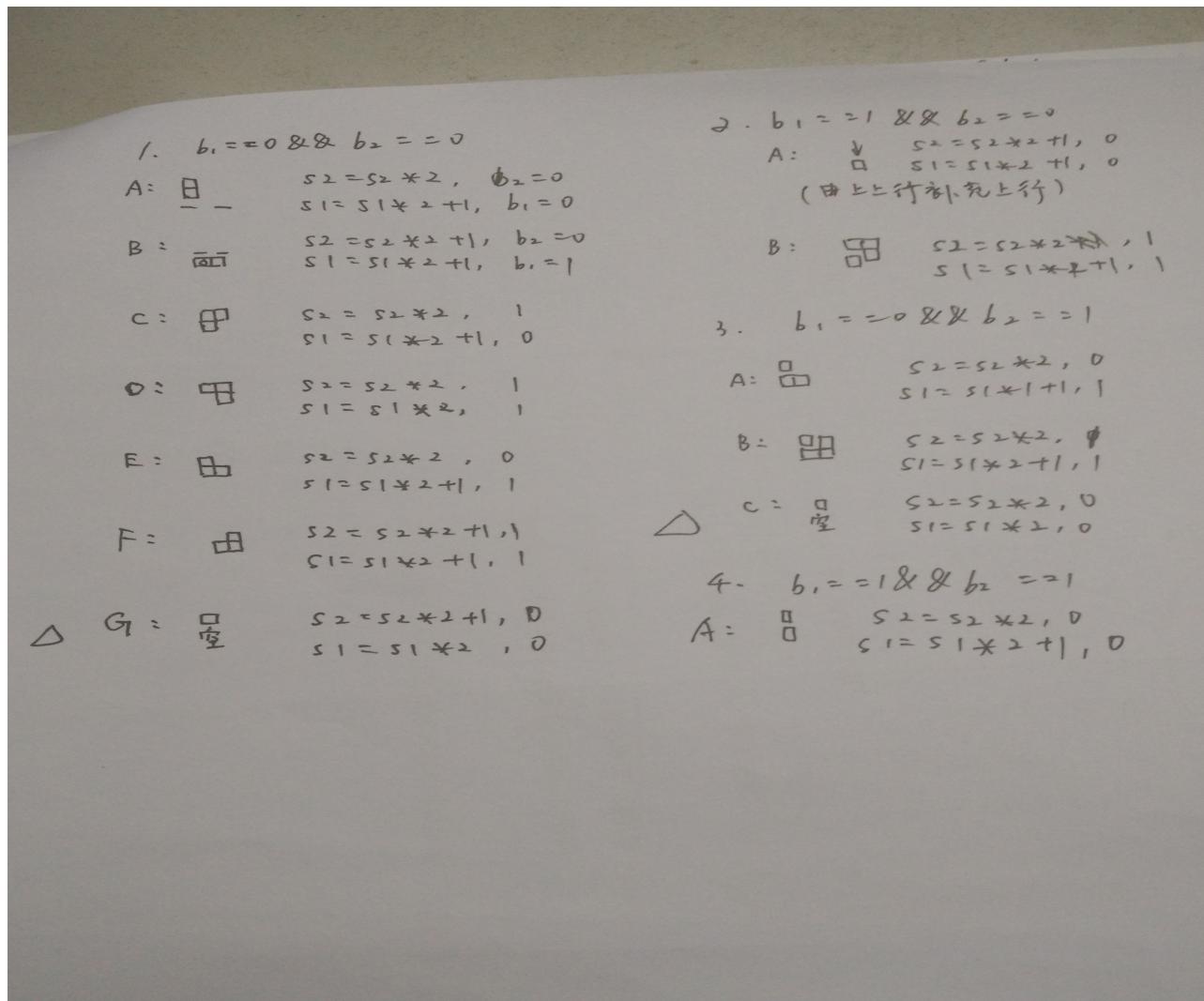
1 const int MAX_N = 10;
2

```

```

3 int n, m;
4 ll ans[MAX_N][MAX_N], dp[MAX_N][1 << MAX_N];
5
6 void dfs(int row, int col, int s1, int b1, int s2, int b2)
7 {
8     if (col > m) return;
9     if (col == m) {
10         if (b1 == 0 && b2 == 0) {
11             dp[row][s1] += dp[row - 1][s2];
12         }
13         return;
14     }
15     if (b1 == 0) {
16         if (b2 == 0) {
17             dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1, 0);
18             dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1, 0);
19             dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1, 1);
20         }
21         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1 | (1 - b2), 0);
22         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1 | (1 - b2), 1);
23     }
24     if (b2 == 0) dfs(row, col + 1, s1 << 1 | b1, 1, s2 << 1, 1);
25     dfs(row, col + 1, s1 << 1 | b1, 0, s2 << 1 | (1 - b2), 0);
26 }
27
28 ll solve()
29 {
30     memset(dp, 0, sizeof(dp));
31     dp[0][(1 << m) - 1] = 1;
32     for (int i = 1; i <= n; ++i) {
33         dfs(i, 0, 0, 0, 0, 0);
34     }
35     return dp[n][(1 << m) - 1];
36 }
37
38 int main()
39 {
40     memset(ans, -1, sizeof(ans));
41     while (~scanf("%d%d", &n, &m)) {
42         if (m > n) swap(n, m);
43         if (ans[n][m] != -1) printf("%lld\n", ans[n][m]);
44         else {
45             ans[n][m] = ans[m][n] = solve();
46             printf("%lld\n", ans[n][m]);
47         }
48     }
49     return 0;
50 }
```

如果一点点的分情况讨论的话，还是比较麻烦的，接下来的代码也是可以 AC 这道题的。



```

1 const int MAX_N = 12;
2
3 int n, m;
4 ll ans[MAX_N][MAX_N], dp[MAX_N][1 << MAX_N];
5
6 void dfs(int row, int col, int s1, int b1, int s2, int b2)
7 {
8     if (col > m) return;
9     if (col == m) {
10         if (b1 == 0 && b2 == 0) dp[row][s1] += dp[row - 1][s2];
11         return;
12     }
13     if (b1 == 0 && b2 == 0) {
14         dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1, 0); // 1A
15         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1 | 1, 0); // 1B
16         dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1, 1); // 1C
17         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1 | 1, 1); // 1D
18         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1, 0); // 1E
19         dfs(row, col + 1, s1 << 1, 1, s2 << 1, 1); // 1F
20         dfs(row, col + 1, s1 << 1, 0, s2 << 1 | 1, 0); // 1G
21     } else if (b1 == 1 && b2 == 0) {
22         dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1 | 1, 0); // 2A
23         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1, 1); // 2B
24     } else if (b1 == 0 && b2 == 1) {
25         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1, 0); // 3A
26         dfs(row, col + 1, s1 << 1 | 1, 1, s2 << 1, 1); // 3B
27         dfs(row, col + 1, s1 << 1, 0, s2 << 1, 0); // 3C
28     } else { // b1 == 1 && b2 == 1
29         dfs(row, col + 1, s1 << 1 | 1, 0, s2 << 1, 0); // 4A
    }
}

```

```

30     }
31 }
32
33 ll solve()
34 {
35     memset(dp, 0, sizeof(dp));
36     dp[0][(1 << m) - 1] = 1;
37     for (int i = 1; i <= n + 1; ++i) {
38         dfs(i, 0, 0, 0, 0, 0);
39     }
40     return dp[n][(1 << m) - 1];
41 }
42
43 int main()
44 {
45     memset(ans, -1, sizeof(ans));
46     while (~scanf("%d%d", &n, &m)) {
47         if (m > n) swap(n, m);
48         if (ans[n][m] != -1) printf("%lld\n", ans[n][m]);
49         else {
50             ans[n][m] = ans[m][n] = solve();
51             printf("%lld\n", ans[n][m]);
52         }
53     }
54     return 0;
55 }
```

给出 $n * m (n, m \leq 10)$ 的方格棋盘, 用 $1 * r$ 的长方形骨牌不重叠地覆盖这个棋盘, 求覆盖满的方案数。

先考虑 $r = 3$ 时的情况。首先, 此问题有解当且仅当 m 或 n 能被 3 整除。更一般的结论是: 用 $1 * r$ 的骨牌覆盖满 $m * n$ 的棋盘, 则问题有解当且仅当 m 或 n 能被 r 整除。当 $r = 2$ 时, 则对应于例 1 中 m, n 至少有一个是偶数的条件。

把每一列的状态表示成一个 r 进制数, 对于 $r = 3$, 我们用 $dp[i][s]$ 表示把前 $i - 1$ 行覆盖满、第 $i - 1$ 和第 i 行覆盖状态为 s 的覆盖方案数。对于第 p 列, $s_p = 0$ 表示 $s_{1p} = s_{2p} = 0$; $s_p = 1$ 表示 $s_{1p} = 0, s_{2p} = 1$; $s_p = 2$ 表示 $s_{1p} = s_{2p} = 1$ 。这样, 我们就只保留了必要的状态, 空间和时间上都有了改进。当 $r = 4$ 时, 可以类推, 用四进制表示三行的状态, $r = 5$ 时用五进制……分别写出 $r = 2, 3, 4, 5$ 的程序, 进行归纳, 统一 DFS 的形式, 可以把 $DFS(p, s1, s2)$ 分为两部分:

```

1. for i = 0 to r - 1 do
2     DFS(p + 1, s1 * r + i, s2 * r + (i + 1) mod r) ;
3. DFS(p + r, s1 * r^r + r^r - 1, s2 * r^r + r^r - 1);
```

问题解决。但 DFS 的这种分部方法是我们归纳猜想得到的, 并没有什么道理, 其正确性无法保证, 我们能否通过某种途径证明它的正确性呢? 仍以 $r = 3$ 为例。根据上面的讨论, s_p 取值 0 到 2, 表示两行第 p 位的状态, 但 s_p 并没有明确的定义。我们 ** 定义 s_p 为这两行的第 p 位从上面一行开始向下连续的 1 的个数 **, 这样的定义可以很容易地递推, 递推式同上两例没有任何改变, 却使得上述 DFS 方法变得很自然。

```

1 const int MAX_N = 11;
2 const int MAX_R = 11;
3 const int LIMIT = 9800000;
4
5 int n, m, r;
6 ll dp[MAX_N][LIMIT], ans[MAX_N][MAX_N][MAX_R], rr, rm;
7
8 void dfs(int row, int col, int s1, int s2)
9 {
10     if (col > m) return;
11     if (col == m) {
12         dp[row][s1] += dp[row - 1][s2];
13         return;
14     }
15     for (int i = 0; i < r; ++i) {
16         dfs(row, col + 1, s1 * r + i, s2 * r + (i + 1) % r);
17     }
18     dfs(row, col + r, s1 * rr + rr - 1, s2 * rr + rr - 1); // 横着放
```

```

19 }
20
21 ll solve()
22 {
23     memset(dp, 0, sizeof(dp));
24     rm = rr = 1;
25     for (int i = 0; i < r; ++i) { rr *= r; }
26     for (int i = 0; i < m; ++i) { rm *= r; }
27     dp[0][rm - 1] = 1;
28     for (int i = 1; i <= n; ++i) {
29         dfs(i, 0, 0, 0);
30     }
31     return dp[n][rm - 1];
32 }
33
34 int main()
35 {
36     memset(ans, -1, sizeof(ans));
37     while (~scanf("%d%d%d", &n, &m, &r)) {
38         if (m > n) swap(n, m);
39         if (ans[n][m][r] != -1) printf("%lld\n", ans[n][m][r]);
40         else if (n * m % r || r > n) printf("0\n");
41         else {
42             ans[n][m][r] = ans[m][n][r] = solve();
43             printf("%lld\n", ans[n][m][r]);
44         }
45     }
46     return 0;
47 }
```

[PKU 1038]

给出 $m (n \leq 150, m \leq 10)$ 的棋盘，要在上面放置 2×3 的骨牌，有一些方格无法放置，求最多能放置多少个。

根据上例的讨论，此题可以使用三进制来表示状态。 s_p 取值 0 到 2，表示两行第 p 位的状态，** 定义 s_p 为这两行的第 p 位从上面一行开始向下连续的 1 的个数 **。我们依旧用 s_1 表示当前行状态， s_2 表示上一行状态， p 表示位置， num 表示从状态 s_1 转移到 s_2 能增加的骨牌个数。

对于当前行第 p 位，如果放置骨牌，则有两种放置方式：

1. 竖放， s_1 当前位置 p 及后一个位置 $p+1$ 都为 2， s_2 要求 p 和 $p+1$ 列留空 2 格，即 $p, p+1$ 列都为 0，这样能刚好放置一块骨牌且不留空隙。对应的 DFS 调用方式： $DFS(p+2, s1 * 9 + 8, s2 * 9, num + 1)$ ；
2. 横放， s_1 的三个位置都要填满，即 $p, p+1, p+2$ 列均为 2， s_2 要求 $p, p+1, p+2$ 列均为 1，调用 $DFS(p+3, s1 * 27 + 26, s2 * 27 + 13, num + 1)$

如果第 p 位不放置骨牌，我们还要把 s_2 的状态转移到 s_1 中以备下次继续转移。形象地说就是虚填一些方块使得下次放置骨牌可以不留空隙（也就是填满所有的位置，但是 num 并不会增加）。对于某一位置，我们可以分 3 种情况 DFS 以达到目的。

1. s_1 的当前行空闲，上一行放满。这个状态可以由 s_2 的放满状态转移得到： $DFS(p+1, s1 * 3 + 1, s2 * 3 + 2, num)$
2. s_1 的当前行和上一行都空闲。这个状态可以由 s_2 的当前行空闲，上一行放满的状态转移得到： $DFS(p+1, s1 * 3, s2 * 3 + 1, num)$
3. s_1 的当前行放满。可以由 s_2 的放满状态转移得到： $DFS(x + 1, s1 * 3 + 2, s2 * 3 + 2, num)$ （放置了 1×1 的虚骨牌在 s_1 的当前位置）

剩下的是关于不能放置的点（黑点）的处理技巧。可以开两个数组 $vertical[i][j], horizontal[i][j]$ 来标志 (i, j) 位置是否能放置竖放和横放骨牌。每次读进一个不能放置的点，对于 $vertical[i][j]$ ，就把以这个位置为右上角的 2×3 的六个位置标记掉，对于 $horizontal[i][j]$ ，就把以这个位置为右上角的 3×2 的六个位置标记掉，然后在 DFS 的时候，判断一下就好了。

```

1 const int MAX_N = 155;
2 const int MAX_M = 15;
3 const int MAX_S = 60000;
4
5 int T, n, m, K, cur;
6 int vertical[MAX_N][MAX_M], horizontal[MAX_N][MAX_M];
7 int dp[MAX_S][2], pw3[MAX_M];
8
9 void dfs(int row, int col, int s1, int s2, int num)
10 {
11     if (col > m) return;
12     if (col == m) {
13         dp[s1][cur] = max(dp[s1][cur], dp[s2][cur ^ 1] + num);
14         return;
15     }
16     if (row >= 2 && vertical[row][col] == 0) { // 坚放
17         dfs(row, col + 2, s1 * 9 + 8, s2 * 9, num + 1);
18     }
19     if (row >= 1 && horizontal[row][col] == 0) { // 横放
20         dfs(row, col + 3, s1 * 27 + 26, s2 * 27 + 13, num + 1);
21     }
22     dfs(row, col + 1, s1 * 3, s2 * 3 + 1, num);
23     dfs(row, col + 1, s1 * 3 + 1, s2 * 3 + 2, num);
24     dfs(row, col + 1, s1 * 3 + 2, s2 * 3 + 2, num);
25 }
26
27 void solve()
28 {
29     memset(dp, 0, sizeof(dp));
30     cur = 0;
31     for (int i = 1; i < n; ++i) {
32         cur ^= 1;
33         dfs(i, 0, 0, 0, 0);
34     }
35     int ret = 0;
36     for (int i = 0; i < pw3[m]; ++i) {
37         ret = max(ret, max(dp[i][cur], dp[i][cur ^ 1]));
38     }
39     printf("%d\n", ret);
40 }
41
42 int main()
43 {
44     pw3[0] = 1;
45     for (int i = 1; i < 12; ++i) { pw3[i] = 3 * pw3[i - 1]; }
46     scanf("%d", &T);
47     while (T--) {
48         scanf("%d%d%d", &n, &m, &K);
49         memset(vertical, 0, sizeof(vertical));
50         memset(horizontal, 0, sizeof(horizontal));
51         for (int i = 0; i < K; ++i) {
52             int x, y;
53             scanf("%d%d", &x, &y);
54             x--;
55             y--;
56             vertical[x][y] = vertical[x + 1][y] = vertical[x + 2][y] = 1;
57             if (y > 0) vertical[x][y - 1] = vertical[x + 1][y - 1]
58                 = vertical[x + 2][y - 1] = 1;
59             horizontal[x][y] = horizontal[x + 1][y] = 1;
60             if (y > 0) horizontal[x][y - 1] = horizontal[x + 1][y - 1] = 1;
61             if (y > 1) horizontal[x][y - 2] = horizontal[x + 1][y - 2] = 1;
62         }
63         solve();
64     }
65     return 0;
}

```

7.5.2 图论模型

[HDU 4917]

给一个 $n(n \leq 40)$ 个顶点和 $m(m \leq 20)$ 个已知拓扑关系的图，求这个图合法的拓扑排序的个数。

已知的 m 个拓扑关系会将图分成若干个连通块，但是每个连通块的内部顶点数量不会超过 20(因为 $m \leq 20$)。我们把一个含有 num 个顶点的连通块内部可以形成的合法拓扑排序的方案数记为 $SCR(num)$ ，当前还剩下 $left$ 个顶点没被确定(包括连通块内的 num 个顶点)那么此时可得的方案数是： $SCR(num) * C[left][num]$ ，这个组合数的含义其实就是把这 num 个顶点按顺序放在 $left$ 个顶点的任意 num 个位置都是可以的，根据乘法原理，每个连通块的答案要相乘，同时 $left$ 要减掉 num 。

那么剩下来两个问题，如何确定每个连通块内部的顶点和求 $SCR(num)$? 对于第一个问题，我们可以建立一个 0/1 邻接矩阵： $child[u][v]$ ，当 $child[u][v] = 1$ 时表示 u 有一个儿子 v (根据读进来的 m 个关系确定)，否则 $child[u][v] = 0$ ，对于每个连通块内部点我们可以借鉴最短路中 Floyd 算法的思路， n^3 的复杂度把这个邻接矩阵“补全”，也就是所有点的所有儿子都标记得到。同一连通块可以通过并查集判断得到。同时要把所有的同一连通块中的顶点重新编号。

对于第二个问题，我们同样枚举状态 $s : 1 \rightarrow ((1 << s) - 1)$ ，把 s 二进制表示中的 1 看成已经排好序的点，只需要枚举最后一个点是谁，但是要满足这个点的所有父亲都已经排好序了，把 v 的父亲节点信息同样用一个二进制数 $pre[v]$ 保存就好了，只要状态 $s \& pre[v] = pre[v]$ ，那么就说明状态 s 中包含了所有 v 的父亲。这里是集合的思想。

时间复杂度是比较高的： $O(n * 2^{num} * num)$ ，时限给了 1000ms，下面的代码跑了 795ms。

```

1 const ll mod = (ll)(1e9 + 7);
2
3 int n, m;
4 int child[45][45], vis[45], fa[45], id[45], step[45], pre[45];
5 ll dp[1 << 21], C[45][45];
6
7 void Floyd()
8 {
9     for (int i = 1; i <= n; ++i) {
10         for (int j = 1; j <= n; ++j) {
11             if (child[i][j] == 0) continue;
12             for (int k = 1; k <= n; ++k) {
13                 if (child[j][k] == 0) continue;
14                 child[i][k] = 1;
15             }
16         }
17     }
18 }
19
20 int find(int x)
21 {
22     return fa[x] == x ? x : fa[x] = find(fa[x]);
23 }
24
25 int GetConnectedGraph(int st)
26 {
27     int ret = 0, ancestor = find(st);
28     memset(pre, 0, sizeof(pre));
29     for (int i = st; i <= n; ++i) {
30         if (vis[i]) continue;
31         int fi = find(i);
32         if (fi == ancestor) {
33             id[ret] = i;
34             step[i] = ret++;
35             vis[i] = 1;
36         }
37     }
38     for (int i = 0; i < ret; ++i) {
39         int v = id[i];
40         for (int u = 1; u <= n; ++u) {

```

```

41         if (child[u][v]) {
42             pre[v] += (1 << step[u]);
43         }
44     }
45 }
46 return ret;
47 }

48
49 ll SCR(int num)
50 {
51     memset(dp, 0, sizeof(dp));
52     dp[0] = 1;
53     for (int s = 1; s < (1 << num); ++s) {
54         for (int i = 0; i < num; ++i) {
55             if ((1 << i) > s) break;
56             if ((1 << i) & s) {
57                 int v = id[i], ss = s - (1 << i);
58                 int tmp = (pre[v] & ss);
59                 if (tmp == pre[v]) {
60                     dp[s] += dp[ss];
61                     if (dp[s] >= mod) dp[s] -= mod;
62                 }
63             }
64         }
65     }
66     return dp[(1 << num) - 1];
67 }

68
69 ll solve()
70 {
71     Floyd();
72     memset(vis, 0, sizeof(vis));
73     ll ret = 1;
74     int left = n;
75     for (int i = 1; i <= n; ++i) {
76         if (vis[i]) continue;
77         int num = GetConnectedGraph(i);
78         ll tmp = SCR(num);
79         ret = ret * C[left][num] % mod * tmp % mod;
80         left -= num;
81     }
82     return ret;
83 }

84
85 int main()
86 {
87     for (int i = 0; i < 45; ++i) {
88         C[i][0] = C[i][i] = 1;
89         for (int j = 1; j < i; ++j) {
90             C[i][j] = C[i - 1][j] + C[i - 1][j - 1];
91             if (C[i][j] >= mod) C[i][j] -= mod;
92         }
93     }
94     while (~scanf("%d%d", &n, &m)) {
95         memset(child, 0, sizeof(child));
96         for (int i = 0; i <= n; ++i) { fa[i] = i; }
97         for (int i = 0; i < m; ++i) {
98             int u, v, fu, fv;
99             scanf("%d%d", &u, &v);
100            child[u][v] = 1;
101            fu = find(u), fv = find(v);
102            if (fu != fv) { fa[fv] = fu; }
103        }
104        printf("%lld\n", solve());
105    }
}

```

```

106     return 0;
107 }
```

通过一种 DFS 的方式获得每个连通块内部的点，并打标记，效果比较好，只跑了 592ms。

同样是邻接矩阵 $link[i][j]$:

- $link[i][j] = 1$: i 是 j 的父亲
- $link[i][j] = -1$: j 是 i 的父亲
- $link[i][j] = 0$: i 和 j 不存在直接系

```

1 void dfs(int u, int& num)
2 {
3     id[num] = u;
4     vis[u] = 1;
5     int tmp = num++; // 暂时保存 u 的编号顺序
6     for (int i = 1; i <= n; ++i) {
7         if (link[u][i] != 0) { // 父亲方向, 儿子方向都遍历
8             if (!vis[i]) dfs(i, num); // 遍历 i 的所有儿子和父亲
9             if (link[u][i] == 1) { // 遍历 u 的所有儿子
10                 for (int j = 0; j < num; ++j) {
11                     if (id[j] == i) { // 找到 i
12                         pre[j] |= (1 << tmp);
13                         break;
14                     }
15                 }
16             }
17         }
18     }
19 }
```

[BZOJ 2784]

求 $n(n \leq 10^5)$ 以内所有数构成的满足下列条件的集合的个数：如果 x 在这个集合内，那么 $2x, 3x$ 都不能在这个集合内。例如： $n = 4$ ，符合条件的子集是：{1}, {1 4} {2} {2 3} {3} {3 4} {4}，一共 8 个。

需 (hen) 要 (nan) 想到这样一个数字矩阵：

x	3x	9x	27x	...
2x	6x	18x	54x	...
4x	12x	36x	108x	...
...

这样的数字矩阵最多会有：17 行 11 列。对于每行的数字显然不能取左右相邻的，对于相邻行的状态显然不能同一列都取。每种矩阵方案先构造矩阵，然后状压 dp+ 滚动数组递推，对于所有的可能矩阵方案乘法原理连乘即可。

虽然时限给了 10s，但是实际上 AC 只要 660MS。

```

1 int n;
2 int mat[18][13], vis[MAX_N], col[18];
3 ll dp[2][1 << 12];
4
5 ll wyr(int x)
6 {
7     int row;
8     // 构造矩阵
9     for (int i = 1; ; ++i) {
10         if (i == 1) mat[i][0] = x;
11         else mat[i][0] = 2 * mat[i - 1][0];
12         if (mat[i][0] > n) {
13             row = i - 1;
14             break;
15         }
16         vis[mat[i][0]] = 1;
17         for (int j = 1; ; ++j) {
18             mat[i][j] = mat[i][j - 1] * 3;
```

```
19     if (mat[i][j] > n) {
20         col[i] = j; // 每一行的列数
21         break;
22     }
23     vis[mat[i][j]] = 1;
24 }
25
26 memset(dp, 0, sizeof(dp));
27 dp[0][0] = 1, col[0] = 0;
28 int cur = 0;
29 ll ret = 0;
30 for (int i = 1; i <= row; ++i) {
31     cur ^= 1;
32     memset(dp[cur], 0, sizeof(dp[cur]));
33     for (int s = 0; s < (1 << col[i]); ++s) {
34         if (s & (s << 1)) continue; // 同一行不能取相邻
35         for (int pre = 0; pre < (1 << col[i - 1]); ++pre) {
36             if (dp[cur ^ 1][pre] == 0 || (pre & s)) continue;
37             // 上一行状态存在并且不能取相邻列
38             dp[cur][s] += dp[cur ^ 1][pre];
39             if (dp[cur][s] >= mod) dp[cur][s] -= mod;
40         }
41         if (i == row) {
42             ret += dp[cur][s];
43             if (ret >= mod) ret -= mod;
44         }
45     }
46 }
47 return ret;
48 }
49
50 int main()
{
51     scanf("%d", &n);
52     ll ans = 1;
53     for (int i = 1; i <= n; ++i) {
54         if (vis[i] == 0) {
55             ans = ans * wyr(i) % mod;
56         }
57     }
58     printf("%lld\n", ans);
59     return 0;
60 }
```

7.6 dp 优化

7.6.1 二进制优化

[HDU 1171]

$n \leq 1000$ 件商品，每件商品的数量和单价为 $num[i] \leq 100$ 和 $value[i]$ ，求将这些商品尽可能分成价值相等的两部分，输出两部分价值 A 和 $B(A \geq B)$ 。

把每种商品的数量二级制拆分成“另一种”商品。

将一部分 $O(\sum(num[i]))$ 的时间复杂度降为: $O(\sum \log(num[i]))$

```

1 const int MAX_N = 1010;
2
3 int sum, n, total;
4 int value[MAX_N], num[MAX_N], good[MAX_N * 15], dp[100010];
5
6 int main()
7 {
8     while (~scanf("%d", &n) && n >= 0) {
9         sum = total = 0;
10        for (int i = 0; i < n; ++i) {
11            scanf("%d%d", &value[i], &num[i]);
12            sum += value[i] * num[i];
13            for (int j = 1; j <= num[i]; j <= 1) {
14                good[total++] = j * value[i];
15                num[i] -= j;
16            }
17            if (num[i]) good[total++] = num[i] * value[i];
18        }
19        memset(dp, 0, sizeof(dp));
20        dp[0] = 1;
21        int half = sum / 2;
22        for (int i = 0; i < total; ++i) {
23            for (int j = half; j >= good[i]; --j) {
24                if (dp[j - good[i]]) {
25                    dp[j] = 1;
26                }
27            }
28        }
29        for (int i = half; i >= 0; --i) {
30            if (dp[i]) {
31                printf("%d %d\n", sum - i, i);
32                break;
33            }
34        }
35    }
36    return 0;
37 }
```

7.6.2 单调队列优化

[POJ 3401]

已知 $n \leq 2000$ 天每天买卖一张股票的单价和买卖上限，但是每天只能选择买或者卖或者不交易，而且两次交易日期中间至少要间隔 W 天，最多可以持有 Max 张股票。初始时有无限量的钱，求最终最多可以获利多少？

用 $dp[i][j]$ 表示在第 i 天拥有 j 张股票时的最多获利。

- 第 i 天不买不卖: $dp[i][j] = dp[i - 1][j]$
- 第 i 天买股票: $dp[i][j] = \max(dp[r][k] - cost[i] * (j - k)) \quad (r \leq i - W - 1, k < j)$

- 第 i 天卖股票: $dp[i][j] = \max(dp[r][k] + value[i] * (k - j)) \quad (r \leq i - W - 1, k > j)$

考虑实际意义上面式子中的 r 就应该是 $i - W - 1$ 。把第二种情况的状态转移改写一下:

$$dp[i][j] = (dp[i - W - 1][k] + buy[i] * k) - buy[i] * j \quad (k < j)$$

可以发现前面部分就是求前缀最大, 所以可以用单调队列来保存前缀最大。对于卖股票的行为同样处理, 但是因为 $k > j$, 所以要倒着处理。

```

1 const int MAX_N = 2010;
2 const int inf = 0x3f3f3f3f;
3
4 int T, n, Max, W, head, tail;
5 int cost[MAX_N], value[MAX_N], buy[MAX_N], sell[MAX_N];
6 // cost 和 value: 买卖单价, buy 和 sell: 买卖上限
7 int dp[MAX_N][MAX_N];
8
9 struct Que {
10     int num, sum;
11     // num: 股票数量 sum: 已经获利量
12     Que() {}
13     Que(int _num, int _sum): num(_num), sum(_sum) {}
14 } que[MAX_N];
15
16 void wyr()
17 {
18     for (int i = 0; i < n; ++i) memset(dp[i], -0x3f, sizeof(dp[i]));
19     // 预处理前 W 天
20     for (int j = 0; j <= buy[0]; ++j) { dp[0][j] = -j * cost[0]; }
21     for (int i = 1; i <= W; ++i) {
22         for (int j = 0; j <= Max; ++j) {
23             dp[i][j] = dp[i - 1][j];
24             if (j <= buy[i]) dp[i][j] = max(dp[i][j], -j * cost[i]);
25         }
26     }
27     for (int i = W + 1; i < n; ++i) {
28         head = tail = 0;
29         for (int j = 0; j <= Max; ++j) {
30             dp[i][j] = dp[i - 1][j]; // 不买不卖
31
32             // 买
33             while (head < tail && que[tail - 1].sum <
34                     dp[i - W - 1][j] + j * cost[i]) {
35                 tail--;
36             }
37             que[tail++] = Que(j, dp[i - W - 1][j] + j * cost[i]);
38             // 保证买的数量不超过上限
39             while (head < tail && j - que[head].num > buy[i]) head++;
40             dp[i][j] = max(dp[i][j], que[head].sum - j * cost[i]);
41         }
42
43         head = tail = 0; // 卖
44         for (int j = Max; j >= 0; --j) {
45             while (head < tail && que[tail - 1].sum <
46                     dp[i - W - 1][j] + j * value[i]) {
47                 tail--;
48             }
49             que[tail++] = Que(j, dp[i - W - 1][j] + j * value[i]);
50             while (head < tail && que[head].sum - j > sell[i]) head++;
51             dp[i][j] = max(dp[i][j], que[head].sum - j * value[i]);
52         }
53     }
54     int ans = 0;
55     for (int i = 0; i <= Max; ++i) {
56         ans = max(ans, dp[n - 1][i]);
57     }
}

```

```

58     printf("%d\n", ans);
59 }
60
61 int main()
62 {
63     scanf("%d", &T);
64     while (T--) {
65         scanf("%d%d%d", &n, &Max, &W);
66         for (int i = 0; i < n; ++i) {
67             scanf("%d%d%d%d", &cost[i], &value[i], &buy[i], &sell[i]);
68         }
69         wyr();
70     }
71     return 0;
72 }
```

POJ 3245

一个长度为 $n \leq 5 * 10^4$ 的序列（每个元素是 (a_i, b_i) 这样的数对），连续地分成若干组。每组左右边界是 $(l_1, r_1), (l_2, r_2), \dots, (l_p, r_p)$ ，满足 $l_i = r_{i-1} + 1, l_i \leq r_i, l_1 = 1, r_p = n$ 。分组必须满足两个条件：前面组的元素的 b 值比后面组元素的所有 a 值大；令 M_i 为第 i 个组内最大的 a ，所有 M_i 的和不超过 $limit$ 。令 S_i 为第 i 个组的元素的 b 的和，最小化 $\max\{S_i\}$ 。

只要有数列后面的 a_j 大于等于当前的 b_i ，那么 i 到 j 的所有元素必须在一个块，那么就把他们合并，合并就是 a 取最大值， b 取和。此时问题就变成了把这些点分成若干组，且各个组的 M_i 之和要满足限制条件，求 $\max\{S_i\}$ 最小是多少， S_i 表示第 i 组中各个点的 b 值之和。二分 $\max\{S_i\}$ 的最小值 x ，然后将问题转变成：将这些点分成若干组，且每组的 b 值之和不能超过 x ，求各个组的 M_i 之和也就各个组的 a 的最大值之和最小是多少，并且判断一下是否小于或等于 $limit$ 。

状态转移方程：

$$dp[i] = \max(dp[j] + \max(a[j+1], a[j+2], \dots, a[i])) \quad sum[i] - sum[j] \leq M$$

决策点具有单调性：在决策区间内维护一个单调下降的序列，缩小决策点的数目。需要使用 multiset。

时间复杂度： $O(n \log n)$

```

1 const int MAX_N = 50010;
2 const int inf = 0x3f3f3f3f;
3
4 int n, limit, head, tail;
5 int A[MAX_N], B[MAX_N], newA[MAX_N], newB[MAX_N], flag[MAX_N];
6 int pre[MAX_N], suf[MAX_N], sum[MAX_N], dp[MAX_N], Q[MAX_N];
7 multiset<int> mst;
8
9 int check(int x)
10 {
11     dp[0] = Q[0] = head = 0, tail = -1;
12     int st = 1;
13     mst.clear();
14     for (int i = 1; i <= n; ++i) {
15         if (newB[i] > x) return 0;
16         while (sum[i] - sum[st - 1] > x) st++; // st 是区间左端点，闭区间
17         while (head <= tail && newA[i] >= newA[Q[tail]]) { // 维护队列单调递减性
18             if (head < tail) mst.erase(dp[Q[tail - 1]] + newA[Q[tail]]); // 剔除值
19             --tail;
20         }
21         Q[++tail] = i;
22         if (head < tail) mst.insert(dp[Q[tail - 1]] + newA[i]); // 增加值
23         while (Q[head] < st) { // 将队列首元素位置调到区间内
24             if (head < tail) mst.erase(dp[Q[head]] + newA[Q[head + 1]]); // 剔除值
25             ++head;
26         }
27         dp[i] = dp[st - 1] + newA[Q[head]]; // 获得 dp 值
28         if (head < tail && dp[i] > (*mst.begin())) dp[i] = *mst.begin();
29         // mst 中所有值都是 dp[i] 的候选项
30         if (dp[i] > limit) return 0;
31 }
```

```

32     return 1;
33 }
34
35 void merge()
36 {
37     suf[n + 1] = 0, pre[0] = INT_MAX;
38     for (int i = 1; i <= n; ++i) {
39         pre[i] = min(pre[i - 1], B[i]);
40     }
41     for (int i = n; i >= 1; --i) {
42         suf[i] = max(suf[i + 1], A[i]);
43     }
44     memset(newA, 0, sizeof(newA));
45     memset(newB, 0, sizeof(newB));
46     memset(flag, 0, sizeof(flag));
47     for (int i = 1; i <= n; ++i) {
48         if (pre[i] > suf[i + 1]) flag[i] = 1;
49     }
50     int total = 1;
51     for (int i = 1; i <= n; ++i) {
52         newA[total] = max(newA[total], A[i]);
53         newB[total] += B[i];
54         if (flag[i]) total++;
55     }
56     n = total - 1;
57     sum[0] = 0;
58     for (int i = 1; i <= n; ++i) {
59         sum[i] = sum[i - 1] + newB[i];
60     }
61 }
62
63 void solve()
64 {
65     merge();
66     int low = 1, high = sum[n], mid;
67     while (low < high) {
68         mid = (low + high) >> 1;
69         if (check(mid)) high = mid;
70         else low = mid + 1;
71     }
72     printf("%d\n", high);
73 }
74
75 int main()
76 {
77     while (~scanf("%d%d", &n, &limit)) {
78         for (int i = 1; i <= n; ++i) {
79             scanf("%d%d", &A[i], &B[i]);
80         }
81         solve();
82     }
83     return 0;
84 }
```

7.6.3 斜率优化

[POJ 1180]

有 $n \leq 10^4$ 件商品需要加工，每件商品有两个属性： $O(i)$ 和 $F(i)$ ，可以把连续的一些商品看成一组一起加工，每加工一组需要 S 的时间启动机器，假设在时刻 t 开始加工从 $i \sim j$ 的商品，那么加工完这些商品的耗时是： $t + S + \sum_{r=i}^{r=j} O(r) = t + T$ ，需要的代价是： $(t + T) * \sum_{r=i}^{r=j} F(r)$ ，并且下一组的开始时刻是 $t + T$ ，初始时刻是 0。求加工完这些商品的最小代价？

可以发现每加工完一组商品的这部分时间: $T = S + \sum_{r=i}^{r=j} O(r)$ 对于后面的所有商品都是有影响的, 可以先把这部分影响计算累加, 影响是:

$$T * \sum_{r=i+1}^n F(r) = T * fsum(j)$$

于是用 $dp[i]$ 表示加工完 $i \sim n$ 商品的最小代价, $tsum[i]$ 和 $fsum[i]$ 表示分别 $O[]$ 和 $F[]$ 的后缀和。

$$dp[i] = \min(dp[j] + (S + tsum[i] - tsum[j]) * fsum[i]) \quad (i < j \leq n)$$

斜率方程:

$$\frac{dp[x] - dp[y]}{tsum[x] - tsum[y]} \leq fsum[i]$$

```

1 11 G(int x, int y)
2 {
3     return dp[x] - dp[y];
4 }
5
6 11 S(int x, int y)
7 {
8     return tsum[x] - tsum[y];
9 }
10
11 void solve()
12 {
13     head = tail = 0;
14     Q[0] = n + 1;
15     dp[n + 1] = 0;
16     for (int i = n; i >= 1; --i) {
17         while (head < tail && G(Q[head + 1], Q[head]) <=
18                 fsum[i] * S(Q[head + 1], Q[head])) ++head;
19         dp[i] = dp[Q[head]] + (tsum[i] - tsum[Q[head]] + s) * fsum[i];
20         while (head < tail && G(Q[tail], Q[tail - 1]) * S(i, Q[tail]) >
21                 G(i, Q[tail]) * S(Q[tail], Q[tail - 1])) --tail;
22         Q[++tail] = i;
23     }
24     printf("%lld\n", dp[1]);
25 }
```

[HDU 3669]

给出 $n \leq 5 * 10^4$ 个矩形的长和宽, 要求把这些矩形最多分成 K 组, 每组的代价是所有这组矩形的最大宽乘以最大长。求最小代价和?

先把矩形按照优先宽度: 从大到小, 其次高度: 从大到小的顺序排序, 其次要注意到一个事实: 如果一个矩形的宽和长都比另外一个矩形小, 那个这个矩形就是不用考虑的。因此我们根据排序后的矩形筛选出的矩形序列满足这样性质: 宽递减并且长递增。

用 $dp[i][k]$ 表示将前 i 个矩形分成 k 组的最小代价:

$$dp[i][k] = \min(dp[j][k - 1] + rec[j + 1].w * rec[i].h) \quad (j < i)$$

斜率方程:

$$\frac{dp[x][p - 1] - dp[y][p - 1]}{rec[y + 1].w - rec[x + 1].w} \leq rec[i].h$$

```

1 int n, K, head, tail;
2 int Q[MAX_N];
3 11 dp[MAX_N][MAX_K];
4
5 struct Rec {
6     int w, h;
7     bool operator < (const Rec& rhs) const {
8         if (w != rhs.w) return w > rhs.w;
```

```

9         else return h > rhs.h;
10    }
11 } read[MAX_N], rec[MAX_N];
12
13 ll G(int id, int x, int y)
14 {
15     return dp[x][id] - dp[y][id];
16 }
17
18 ll S(int x, int y)
19 {
20     return rec[y + 1].w - rec[x + 1].w;
21 }
22
23 void wyr()
24 {
25     for (int i = 1; i <= n; ++i) {
26         dp[i][1] = 1ll * rec[1].w * rec[i].h;
27     }
28     for (int k = 2; k <= K; ++k) {
29         head = tail = 0;
30         Q[++tail] = k - 1;
31         for (int i = k; i <= n; ++i) {
32             while (head < tail && G(k - 1, Q[head + 1], Q[head])
33                   <= S(Q[head + 1], Q[head]) * rec[i].h) ++head;
34             int t = Q[head];
35             dp[i][k] = dp[t][k - 1] + 1ll * rec[t + 1].w * rec[i].h;
36             while (head < tail && G(k - 1, Q[tail], Q[tail - 1]) * S(i, Q[tail])
37                   >= G(k - 1, i, Q[tail]) * S(Q[tail], Q[tail - 1])) --tail;
38             Q[++tail] = i;
39         }
40     ll ans = (ll)(1e18);
41     for (int i = 1; i <= K; ++i) {
42         ans = min(ans, dp[n][i]);
43     }
44     printf("%lld\n", ans);
45 }
46
47
48 int main()
49 {
50     while (~scanf("%d%d", &n, &K)) {
51         for (int i = 0; i < n; ++i) {
52             scanf("%d%d", &read[i].w, &read[i].h);
53         }
54         sort(read, read + n);
55         int MaxH = 0, total = 0;
56         for (int i = 0; i < n; ++i) {
57             if (read[i].h > MaxH) {
58                 rec[++total].w = read[i].w;
59                 rec[total].h = read[i].h;
60                 MaxH = read[i].h;
61             }
62         }
63         n = total;
64         wyr();
65     }
66     return 0;
67 }
```

7.6.4 四边形不等式优化

假如对于 $i < j$, 有:

$$w(i, j) + w(i+1, j+1) \leq w(i+1, j) + w(i, j+1)$$

称函数 w 满足四边形不等式。

将不等式变形得:

$$w(i+1, j+1) - w(i+1, j) \leq w(i, j+1) - w(i, j)$$

那么证明函数 w 是否满足四边形不等式, 即证明: 当 j 固定不变时是否有: $w(i, j+1) - w(i, j)$ 随 i 非递增。

- 定理 1

如果有状态转移方程:

$$m(i, j) = \min_{i < k \leq j} \{m(i, k-1) + m(k, j) + w(i, j)\} \quad (i < j, m(i, i) = 0)$$

那么函数 m 也满足四边形不等式:

$$m(i, j) + m(i+1, j+1) \leq m(i+1, j) + m(i, j+1), \quad i < j$$

- 定理 2

定义 $s(i, j)$ 为函数 $m(i, j)$ 对应的决策变量的最大值, 即:

$$s(i, j) = \max_{i < k \leq j} \{m(i, j) = w(i, j) + m(i, k-1) + m(k, j)\}$$

并且 $m(i, j)$ 满足四边形不等式, 那么 $s(i, j)$ 单调, 即:

$$s(i, j) \leq s(i, j+1) \leq s(i+1, j+1)$$

因此 $m(i, j)$ 的状态转移方程等价于:

$$m(i, j) = \min_{s(i, j-1) < k \leq s(i+1, j)} \{m(i, k-1) + m(k, j) + w(i, j)\} \quad (i < j, m(i, i) = 0)$$

这个转移的复杂度是: $O(n^2)$ 。

环形石子堆合并

[HDU 3506]: 给 $n \leq 1000$ 个围成一圈的石子堆, 每次可以合并相邻的两个石子堆成一个新的石子堆, 合并的代价是两堆石子数量之和, 求将 n 堆石子合并成一堆的最小代价?

先将 n 堆环形石子展开成一排 $2 * n$ 堆石子。用 $dp[i][j]$ 表示合并第 i 堆到第 j 堆石子的最小代价, 状态转移方程:

$$dp[i][j] = \min(dp[i][k] + dp[k+1][j]) + cost[i][j] \quad (i \leq k \leq j)$$

利用四边形不等式优化, 只需证明: 当 j 固定不变时是否有 $cost[i][j+1] - cost[i][j]$ 随 i 单调非递增。

$$cost[i][j] = \sum_{r=i}^{r=j} data[r]cost[i][j+1] - cost[i][j] = data[j+1]$$

因为 j 固定不变, 所以差值是个恒定值, 那么满足随 i 单调非递增 (不变)。

时间复杂度: $O(n^2)$

```

1 const int MAX_N = 1010 * 2;
2
3 int n;
4 int d[MAX_N], sum[MAX_N];
5 int dp[MAX_N][MAX_N], s[MAX_N][MAX_N];
6
7 void solve()
8 {
9     for (int i = 1; i <= 2 * n; ++i) {
10         sum[i] = sum[i - 1] + d[i];
11     }
12 }
```

```

11 }
12 d[2 * n + 1] = d[1];
13 for (int i = 1; i < 2 * n; ++i) {
14     dp[i][i + 1] = d[i] + d[i + 1];
15     s[i][i + 1] = i;
16 }
17 for (int len = 3; len <= n; ++len) {
18     for (int i = 1; i + len - 1 <= 2 * n; ++i) {
19         int j = i + len - 1;
20         dp[i][j] = INT_MAX;
21         int a = s[i][j - 1], b = s[i + 1][j];
22         for (int r = a; r <= b; ++r) {
23             int tmp = dp[i][r] + dp[r + 1][j] + sum[j] - sum[i - 1];
24             if (tmp < dp[i][j]) {
25                 dp[i][j] = tmp;
26                 s[i][j] = r;
27             }
28         }
29     }
30 }
31 int ans = INT_MAX;
32 for (int i = 1; i <= n; ++i) {
33     ans = min(ans, dp[i][i + n - 1]);
34 }
35 printf("%d\n", ans);
36 }
37
38 int main()
39 {
40     while (~scanf("%d", &n)) {
41         for (int i = 1; i <= n; ++i) {
42             scanf("%d", &d[i]);
43             d[i + n] = d[i];
44             dp[i][i] = 0;
45             s[i][i] = i;
46         }
47         solve();
48     }
49     return 0;
50 }

```

[HDU 2829]

给 $n \leq 1000$ 个正整数，定义下标 $i \sim j$ 的一串连续的没被挡板隔开的数的价值为： $\sum_{p=i}^{p=j} data[p] \sum_{q=p+1}^{q=j} data[q]$ ，只有一个数时价值为 0，选择合适的位置放置 $K \in [0, n)$ 个挡板，使得最终的总价值最小，输出最小总价值。

定义： $sum[i] = \sum_{r=1}^{r=i} data[r]$, $fsum[i] = \sum_{r=1}^{r=i} data[r] * sum[r]$

$$\begin{aligned}
cos[i][j] &= \sum_{p=i}^{p=j} data[p] \sum_{q=p+1}^{q=j} data[q] \\
&= \sum_{p=i}^{p=j} data[p] * (sum[j] - sum[p]) \\
&= sum[j] * (sum[j] - sum[i - 1]) - (fsum[j] - fsum[i - 1])
\end{aligned}$$

定义 $dp[i][k]$ 表示在前 i 个数放置 k 个挡板可获得价值，状态转移方程：

$$dp[i][k] = \min_{0 \leq j < i} \{dp[j][k - 1] + cost[j + 1][i]\}$$

考虑四边形不等式优化，只需要证明：当 j 固定时， $cost[i][j + 1] - cost[i][j]$ 随 i 单调非增。

$$\begin{aligned}
cost[i][j + 1] - cost[i][j] &= (sum[j + 1]^2 - fsum[j + 1]) - (sum[j]^2 - fsum[j]) \\
&\quad + (sum[j] - sum[j + 1]) * sum[i - 1]
\end{aligned}$$

因为 $sum[r]$ 是单调递增的，所以 $(sum[j] - sum[j + 1]) * sum[i]$ 在 j 固定的条件下随 i 单调递减，满足四边形不等式优化条件。

时间复杂度： $O(n * K)$ ，常数有点大

```

1 const int MAX_N = 1010;
2
3 int n, m;
4 int s[MAX_N][MAX_N];
5 ll data[MAX_N], sum[MAX_N], fsum[MAX_N];
6 ll cost[MAX_N][MAX_N], dp[MAX_N][MAX_N];
7
8 void wyr()
9 {
10     for (int i = 1; i <= n; ++i) {
11         for (int j = i; j <= n; ++j) {
12             cost[i][j] = cost[j][i] = sum[j] * (sum[j] - sum[i - 1])
13                             - (fsum[j] - fsum[i - 1]);
14         }
15     }
16     memset(dp, 0x3f, sizeof(dp));
17     memset(dp[0], 0, sizeof(dp[0]));
18     for (int i = 1; i <= n; ++i) {
19         dp[i][0] = cost[1][i];
20         s[i][0] = 0;
21     }
22     for (int k = 1; k <= m; ++k) {
23         for (int i = n; i >= 1; --i) {
24             if (k >= i) dp[i][k] = 0, s[i][k] = i;
25             else {
26                 int a, b;
27                 if (i == n) a = k - 1, b = n - 1;
28                 else a = s[i][k - 1], b = s[i + 1][k];
29                 for (int j = a; j <= b; ++j) {
30                     if (dp[j][k - 1] + cost[j + 1][i] < dp[i][k]) {
31                         dp[i][k] = dp[j][k - 1] + cost[j + 1][i];
32                         s[i][k] = j;
33                     }
34                 }
35             }
36         }
37     }
38     printf("%lld\n", dp[n][m]);
39 }
40
41 int main()
42 {
43     while (~scanf("%d%d", &n, &m) && (n + m)) {
44         for (int i = 1; i <= n; ++i) {
45             scanf("%lld", &data[i]);
46             sum[i] = sum[i - 1] + data[i];
47             fsum[i] = fsum[i - 1] + data[i] * sum[i];
48         }
49         wyr();
50     }
51     return 0;
52 }
```

考虑斜率优化。令 $k > j$ ，且：

$$dp[k][p - 1] + cost[k + 1][i] < dp[j][p - 1] + cost[j + 1][i]$$

即：

$$\begin{aligned} dp[k][p - 1] + sum[k] * (sum[k] - sum[i - 1]) - (fsum[k] - fsum[i - 1]) < \\ dp[j][p - 1] + sum[j] * (sum[j] - sum[i - 1]) - (fsum[j] - fsum[i - 1]) \end{aligned}$$

化简得斜率方程:

$$\frac{(dp[k][p-1] + fsum[k]) - (dp[j] + fsum[j])}{sum[k] - sum[j]} \leq sum[i]$$

时间复杂度: $O(n * K)$

```

1 const int MAX_N = 1010;
2
3 int n, K, head, tail;
4 int Q[MAX_N];
5 ll data[MAX_N], sum[MAX_N], fsum[MAX_N];
6 ll dp[MAX_N][MAX_N];
7
8 ll G(int id, int x, int y)
9 {
10     return dp[x][id] + fsum[x] - (dp[y][id] + fsum[y]);
11 }
12
13 ll S(int x, int y)
14 {
15     return sum[x] - sum[y];
16 }
17
18 void wyr()
19 {
20     for (int i = 1; i <= n; ++i) {
21         dp[i][0] = sum[i] * sum[i] - fsum[i];
22     }
23     dp[0][0] = 0;
24     for (int k = 1; k <= K; ++k) {
25         head = tail = 0;
26         Q[++tail] = k - 1;
27         for (int i = k; i <= n; ++i) {
28             while (head < tail && G(k - 1, Q[head + 1], Q[head]) <=
29                     S(Q[head + 1], Q[head]) * sum[i]) ++head;
30             int t = Q[head];
31             dp[i][k] = dp[t][k - 1] + sum[i] * (sum[i] - sum[t]) - fsum[i] + fsum[t];
32             while (head < tail && G(k - 1, i, Q[tail]) * S(Q[tail], Q[tail - 1]) <=
33                     G(k - 1, Q[tail], Q[tail - 1]) * S(i, Q[tail])) --tail;
34             Q[++tail] = i;
35         }
36     }
37     printf("%lld\n", dp[n][K]);
38 }
39
40 int main()
41 {
42     while (~scanf("%d%d", &n, &K) && (n + K)) {
43         for (int i = 1; i <= n; ++i) {
44             scanf("%lld", &data[i]);
45             sum[i] = sum[i - 1] + data[i];
46             fsum[i] = fsum[i - 1] + data[i] * sum[i];
47         }
48         wyr();
49     }
50     return 0;
51 }
```

7.6.5 bitset 优化

使用 bitset 优化, 利用 bitset 的位移特性和每一位 01 表示匹配状态。例如对于模式串: *abc* 和读入文本: *abcabcabc*。先根据读入文本得到每个字母向量表示 (从右往左看第 *i* 位为 1 表示读入文本的第 *i* 位 (从左往右看) 为该字母):

bs[a]:001001001

$bs[b]:010010010$

$bs[c]:100100100$

用 dp 表示匹配状态，初始时： $dp=111111111$ ，扫描模式串： abc 。

对于第一个字母 a ， $(dp \ll 1) \& bs[a]$ 可得： $dp = (111111110 \& 001001001) = 001001001$ ，这表示字母 a 可以在文本串中的哪些位置作为前缀。

对于第二个字母 b ， $(dp \ll 1) \& bs[b]$ 可得： $dp = (010010010 \& 010010010) = 010010010$ ，这表示 ab 可以在文本串中的哪些位置作为前缀。如果想要以当前 b 结尾作为前缀的话，那么必然需要以前一个字母 a 作为上一个字母的前缀，所以需要先 dp 左移一位，然后与上当前 b 可以匹配的位置。

对于第三个字母 c ， $(dp \ll 1) \& bs[c]$ 可得： $dp = (100100100 \& 100100100) = 100100100$ 。此时状态 1 的位置就表示可以和模式串匹配的结尾位置。

[HDU 5745]

给一个长度 $n \leq 10^5$ 的文本串和长度为 $m \leq 5000$ 的模式串，对于文本串的每个字母可以选择相邻位置字母交换但是不允许交叉交换。例如 $abcd$ 可以变换为 $bacd, abdc, acbd, badc$ ，但是不能变成 $bcad, bcda$ 等。对于文本串的每个位置判断以它为起始的子串能否变换为模式串。

其实就是限制了每个位置字母的交换位置只能是相邻的两个。用 $dp[i][j][\cdot]$ 表示文本串的第 i 个位置和模式串的第 j 个位置的匹配状态。第三维用 0,1,2 分别表示文本串的第 i 个字母和第 $i - 1$ 个字母交换，不动以及和第 $i + 1$ 个字母交换三种状态。状态转移：

$$\begin{aligned} dp[i][j][0] &= dp[i - 1][j - 1][2] \&& a[i] == b[j - 1] \\ dp[i][j][1] &= (dp[i - 1][j - 1][0] \mid dp[i - 1][j - 1][1]) \&& a[i] == b[j] \\ dp[i][j][2] &= (dp[i - 1][j - 1][0] \mid dp[i - 1][j - 1][1]) \&& a[i] == b[j + 1] \end{aligned}$$

先处理出文本串中每个字母出现的位置，相当于状压第一维，然后枚举模式串的每个位置借助 bitset 左移操作模拟匹配并且滚动数组。

时间复杂度： $O(\frac{n*m}{w})$, w 是机器字节数

```

1 const int MAX_N = 100010;
2 const int MAX_M = 5010;
3
4 int T, n, m;
5 int ans[MAX_N];
6 char str1[MAX_N], str2[MAX_M];
7 bitset<MAX_N> dp[2][3], bs[30];
8
9 void init() {
10     for (int i = 0; i < 2; ++i) {
11         for (int j = 0; j < 3; ++j) {
12             dp[i][j].reset();
13             dp[i][j][0] = 1;
14         }
15     }
16     for (int i = 0; i < 26; ++i) { bs[i].reset(); }
17     for (int i = 1; i <= n; ++i) {
18         bs[str1[i] - 'a'][i] = 1;
19     }
20 }
21
22 void solve() {
23     init();
24     int now = 0;
25     dp[0][1].set(); // 初始置为1
26     for (int i = 1; i <= m; ++i) {
27         now ^= 1;
28         if (i > 1) dp[now][0] = (dp[now ^ 1][2] << 1) & bs[str2[i - 1] - 'a'];
29         dp[now][1] = ((dp[now ^ 1][1] | dp[now ^ 1][0]) << 1) & bs[str2[i] - 'a'];
30         if (i <= m - 1)
31             dp[now][2] = ((dp[now ^ 1][0] | dp[now ^ 1][1]) << 1) & bs[str2[i + 1] - 'a'];
32         dp[now][0][0] = dp[now][1][0] = dp[now][2][0] = 1;
}

```

```

33 }
34     for (int i = 1; i <= n - m + 1; ++i) {
35         if (dp[now][0][i + m - 1] || dp[now][1][i + m - 1]) printf("1");
36         else printf("0");
37     }
38     for (int i = n - m + 2; i <= n; ++i) { // 最后的 m-1 个位置肯定不符
39         printf("0");
40     }
41     printf("\n");
42 }
43
44 int main() {
45     scanf("%d", &T);
46     while (T--) {
47         scanf("%d%d", &n, &m);
48         scanf("%s%s", str1 + 1, str2 + 1);
49         solve();
50     }
51     return 0;
52 }
```

[2016 大连 B]

给一个 $n \leq 1000$, 代表数字长度, 以及每位上候选数字集合, 再给一个数字字符串 $s(|s| \leq 5 * 10^6)$, 输出 s 中所有匹配的 n 位数字子串。

样例输入:

4 (一共四位)
 3 0 9 7 (第一位有三个候选数字分别为: 0 9 7)
 2 5 7 (第二位有两个候选数字分别为: 5 7)
 2 2 5 (第三位有两个候选数字分别为: 2 5)
 2 4 5 (第四位有两个候选数字分别为: 4 5)
 09755420524 (数字字符串 s)

样例输出: (所有匹配的四位数字子串)

9755
 7554
 0524

把 n 位数字看成模式串, 先处理处每个数字可以在模式串中的匹配位置, 然后扫描文本串。用 $dp[i][j]$ 表示文本串的第 i 个位置能否和模式串的第 j 个位置匹配 (前缀), 状态转移:

$$dp[i][j] = dp[i - 1][j - 1] \&& a[i] \in b[j]$$

时间复杂度: $O(\frac{n*m}{w})$, w 是机器字节数

```

1 const int MAX_M = 5000010;
2 const int MAX_N = 1010;
3
4 int n, len;
5 char str[MAX_M];
6 bitset<MAX_N> bs[10], dp[2];
7
8 void solve() {
9     len = strlen(str + 1);
10    dp[0].reset(), dp[1].reset();
11    dp[0][0] = 1;
12    int now = 0;
13    for (int i = 1; i <= len; ++i) {
14        now ^= 1;
15        dp[now] = (dp[now ^ 1] << 1) & bs[str[i] - '0'];
16        dp[now][0] = 1;
17        if (dp[now][n]) {
18            char ch = str[i + 1];
19            str[i + 1] = '\0';
20        }
21    }
22 }
```

```
20         printf("%s\n", str + (i - n + 1));
21         str[i + 1] = ch;
22     }
23 }
24
25 int main() {
26     while (~scanf("%d", &n)) {
27         for (int i = 0; i < 10; ++i) { bs[i].reset(); }
28         for (int i = 1; i <= n; ++i) {
29             int x, y;
30             scanf("%d", &x);
31             for (int j = 0; j < x; ++j) {
32                 scanf("%d", &y);
33                 bs[y][i] = 1;
34             }
35         }
36         scanf("%s", str + 1);
37         solve();
38     }
39     return 0;
40 }
41 }
```

Chapter 8

数据结构

8.1 哈希

[HDU 5918](#)

给定元素个数分别为 $n \leq 10^6$ 和 $m \leq 10^6$ 的数组 $A[]$ 和 $B[]$ (下标从 1 开始) 和一个常数 $p \leq 10^6$, 求在数组 $A[]$ 中满足 $A[q] = B[1], A[q + p] = B[2], A[q + 2 * p] = B[3]..., A[q + (m - 1) * p] = B[m]$ 的位置 $q(1 \leq q, q + (m - 1) * p \leq n)$ 的个数。

```
1 const ll mod1 = 1034128911111;
2 const ll mod2 = 1032728711111;
3 const ll prime1 = 95734711;
4 const ll prime2 = 132134911;
5 const int MAX_N = 1000010;
6
7 int T, n, m, p, cases = 0;
8 int A[MAX_N], B[MAX_N];
9
10 struct Hash {
11     ll a, b;
12 } hesh[MAX_N];
13
14 void solve()
15 {
16     ll ret1 = 0, ret2 = 0;
17     ll pw1 = 1, pw2 = 1;
18     for (int i = 1; i <= m; ++i) {
19         ret1 = (ret1 * prime1 % mod1 + B[i]) % mod1;
20         if (i > 1) pw1 = pw1 * prime1 % mod1;
21
22         ret2 = (ret2 * prime2 % mod2 + B[i]) % mod2;
23         if (i > 1) pw2 = pw2 * prime2 % mod2;
24     }
25     for (int i = 1; 111 * (m - 1) * p + i <= n; ++i) {
26         if (i <= p) {
27             hesh[i].a = hesh[i].b = 0;
28             for (int j = 0; j < m; ++j) {
29                 int pos = i + j * p;
30                 hesh[i].a = (hesh[i].a * prime1 % mod1 + A[pos]) % mod1;
31                 hesh[i].b = (hesh[i].b * prime2 % mod2 + A[pos]) % mod2;
32             }
33         } else {
34             hesh[i].a = (hesh[i - p].a - (11)A[i - p] * pw1 % mod1) * prime1 % mod1;
35             hesh[i].a = (hesh[i].a + A[i + (m - 1) * p]) % mod1;
36             if (hesh[i].a < 0) hesh[i].a += mod1;
37
38             hesh[i].b = (hesh[i - p].b - (11)A[i - p] * pw2 % mod2) * prime2 % mod2;
39             hesh[i].b = (hesh[i].b + A[i + (m - 1) * p]) % mod2;
40             if (hesh[i].b < 0) hesh[i].b += mod2;
41     }
}
```

```
42 }
43 int ans = 0;
44 for (int i = 1; 111 * (m - 1) * p + i <= n; ++i) {
45     if (hesh[i].a == ret1 && hesh[i].b == ret2) ans++;
46 }
47 printf("Case #%d: %d\n", ++cases, ans);
48 }

49
50 int main()
51 {
52     scanf("%d", &T);
53     while (T--) {
54         scanf("%d%d%d", &n, &m, &p);
55         for (int i = 1; i <= n; ++i) {
56             scanf("%d", &A[i]);
57         }
58         for (int i = 1; i <= m; ++i) {
59             scanf("%d", &B[i]);
60         }
61         solve();
62     }
63     return 0;
64 }
```

8.2 并查集

8.2.1 加权并查集

n 表示 n 个数字，编号 $1-n$ ，然后有 m 个区间 $[l, r]$ 和该区间和 s ，问在这 m 个区间中有多少个区间和是不正确的？如果不正确就忽略该区间和，否则将该区间和作为已知条件使用。

需要一个数组 $val, val[i]$ 表示 i 到根节点的距离，然后就是在查找根节点的过程更新路径上结点的 val ，和在 mix 函数里判断该区间和是否有效。1 到 r 之间的和为 s 可以理解为 l 到 r 的距离为 s 。

```

1 const int maxn = 200010;
2
3 int val[maxn], pre[maxn];
4 int n, m, u, v, w;
5 //val[i]: i 到根节点的距离; pre[i]: i 的父节点
6
7 int find(int x)
8 {
9     if (pre[x] == x) return x;
10    int tmp = find(pre[x]); // tmp 是 x 的根节点
11    val[x] = val[x] + val[pre[x]];
12    // 在递归时 x 还未连接到根节点上，只连接到父节点上，
13    // 所以这时的 val[x] 实际上是到父节点的距离
14    // 而 val[pre[x]] 是父节点到根节点的距离,
15    // 所以真正的 val[x]=val[x]+val[pre[x]]，左边的 val[x] 是到根节点距离，右边的是到父节点距离
16    // 可以从递归倒数第二层往前想
17    return pre[x] = tmp; //路径压缩
18 }
19
20 int mix(int x, int y, int z)
21 {
22     int fx = find(x), fy = find(y);
23     if (fx != fy) {
24         pre[fx] = fy; // 将 x 的根节点 fx 的父节点设为fy
25         val[fx] = val[y] - val[x] + z;
26         // x 到 y 的距离是 z， x 到 fx 的距离是 val[x]， y 到 fy 的距离是 val[y]
27         // 那么 fx 到 fy 的距离 val[fx]=x 到 fy 的距离 (z+val[y])- x 到 fx 的距离 val[x]
28         return 0;
29     } else {
30         if (abs(val[y] - val[x]) == z) return 0; // 必须是绝对值才行
31         // 因为尽管 y>x，但是到根节点的距离不确定大小（到根结点路径上个点的值大小不确定）
32         return 1;
33     }
34 }
35
36 int main()
37 {
38     while (~scanf("%d%d", &n, &m)){
39         //因为接下来由 u-- 的存在，所以要从开始0
40         for (int i = 0; i <= n; i++)
41             pre[i] = i;
42         memset(val, 0, sizeof(val));
43         int ans = 0;
44         for (int i = 1; i <= m; i++) {
45             scanf("%d%d%d", &u, &v, &w);
46             u--; // 这样做符合 val 的含义
47             ans += mix(u, v, w);
48         }
49         printf("%d\n", ans);
50     }
51     return 0;
52 }
```

8.2.2 分层并查集

有 n 个动物，编号 $1-n$ ，每个动物属于 A,B,C 三种中的一种，并且 A 种动物吃 B 种动物，B 种动物吃 C 种动物，C 种动物吃 A 种动物。有 K 条语句。格式是： d, x, y 。

- $d = 1$ 时表示 x 和 y 是属于同一种类
- $d = 2$ 时表示 x 吃 y

这条语句如果满足下列条件之一就是错误的语句：

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X ，就是假话。

问 K 条语句中一共有多少条语句是错误的？

在数组 $pre[3 * maxn]$ 中， $1-n$ 表示种类 A， $n+1-2*n$ 表示种类 B， $2*n+1-3*n$ 表示种类 C。
当 $d=1$ 时：先检查 x 和 y 是否属于不同的种类，如果是不同种类，则 $ans++$ ，否则就将 x, y 同属于 A,B,C 依次 mix；

当 $d=2$ 时：先检查 x 和 y 是否属于同一种类和 y 吃 x 的情况，如果是的，则 $ans++$ ，否则就将 x 属于 A, y 属于 B; x 属于 B, y 属于 C; x 属于 C, y 属于 A 三种情况 mix

```

1 const int maxn = 50010;
2
3 int n, k, d, x, y;
4 int pre[3 * maxn];
5
6 int find(int x)
7 {
8     return pre[x] == x ? x : pre[x] = find(pre[x]);
9 }
10
11 void mix(int x, int y)
12 {
13     int fx = find(x), fy = find(y);
14     if (fx != fy){
15         pre[fx] = fy;
16     }
17 }
18
19 int main()
20 {
21     scanf("%d%d", &n, &k);
22     for (int i = 1; i <= 3 * n; i++)
23         pre[i] = i;
24     int ans = 0;
25     for (int i = 0; i < k; i++){
26         scanf("%d%d%d", &d, &x, &y);
27         if (x > n || y > n || (d == 2 && x == y)){
28             ans++;
29             continue;
30         }
31         if (d == 1){
32             if (find(x) == find(y + n) || find(y) == find(x + n)) ans++;
33             // 因为当 x 和 y 属于不同种类时无非是 x 吃 y 或者 y 吃 x
34             // 而且每种捕食关系的三种种类归属情况同时入队列,
35             // 所以检查 x 和 y 属于同一种类就是检查 x 和 y+n 与 x+n 和 y 是否属于同一集合
36             // (前者是 x 吃 y, 后者是 y 吃 x)
37             else {
38                 mix(x, y); // 同属A
39                 mix(x + n, y + n); // 同属 B
40                 mix(x + 2 * n, y + 2 * n); // 同属 C
41             }
42         } else {

```

```

43     if ( find(x) == find(y) || find(y) == find(x + n) ) ans++;
44     //前者是检查 x 和 y 是否属于同一种类, 后者是检查是否是 y 吃 x
45     else {
46         mix(x, y + n); //x 属于 A , y 属于 B
47         mix(x + n, y + 2 * n); // x 属于 B , y 属于 C
48         mix(x + 2 * n, y); // 属于 x C , y 属于 A
49     }
50 }
51 }
52 printf("%d\n", ans);
53 return 0;
54 }
```

有 n 个数字, 每个数字非 0 即 1, 有 m 条语句, 每条语句: $l, r, even/odd$, 表示 l 到 r 区间上有奇/偶个 1. 问最多前多少条语句是正确的?

用 $val[i]$ 表示从 i 到根节点路径上含有 1 的数量的奇偶性。在寻找根节点的同时更新路径上的 val 。比较麻烦的是数据范围: $n <= 1000000000 m <= 5000$.

1. 可以用 map 来标记各个读入点的次序, 相当于有个代号, 在 map 里没读入的话就添加进 map 在 $find()$ 和 $mix()$ 函数里相当于是对代号的操作, 代号具有唯一性。

由于 $m <= 5000$, 所以最多会读入 10000 个相异的数, 那么对于 pre 数组和 val 数组都是可以接受的了。

2. hash 思路。同样是代号的想法, 把 l (或 r) mod $maxn$ 值相同的归在一类, 用 $head$ 数组的下标记录这类, $head$ 的数值表示最后一类的读入位置。和最短路里链式前向星的查找类似。

```

1 const int maxn=10010;
2
3 int pre[maxn], val[maxn], n, m, l, r, w;
4 char s[10];
5
6 int find(int x)
7 {
8     if (pre[x]==x) return x;
9     int tmp=find(pre[x]);
10    val[x]=(val[x]+val[pre[x]])%2;
11    return pre[x]=tmp;
12 }
13
14 int mix(int x, int y, int z)
15 { // 语句正确, mix 返回 1 , 否则返回0
16     int fx=find(x);
17     int fy=find(y);
18     if (fx!=fy)
19     {
20         pre[fx]=fy;
21         val[fx]=(val[y]-val[x]+z)%2;
22         return 1;
23     }
24     else
25     {
26         if (abs(val[x]-val[y])%2==z) return 1;
27         return 0;
28     }
29 }
30
31 int main()
32 {
33     while(~scanf("%d", &n)&&n){
34         for (int i=0; i<maxn; i++){
35             pre[i]=i;
36             val[i]=0;
37         }
38         scanf("%d", &m);
39         int ok=0;
40         int ans=0;
41         int index=0;
```

```

42     map<int , int> mp;
43     for ( int i=1;i<=m; i++ ) {
44         scanf( "%d%d%s",&l,&r , s );
45         if ( s[0]== 'e' ) w=0;
46         else w=1;
47         l--; // 这样做就可以合并相邻的区间, 例如: 读入 l=1,r=2 和 l=3,r=4
48         if ( mp. find (l)==mp. end () ) mp[ l]=index++;
49         // find() 函数返回一个迭代器指向键值为 key 的元素
50         // 如果没找到就返回指向 map 尾部的迭代器
51         if ( mp. find (r)==mp. end () ) mp[ r]=index++;
52         // 假设前 x 条语句是正确的, 第 x+1 条是错的
53         // 那么前 x 次读入 ok 都为 0 , 第 x+1 次读入由于 ok=0,mix() 返回0
54         // 所以执行 else 语句, ok 变为 1 , ans=i-1=(x+1)-1=x
55         // 从 x+2 开始由于 ok=1 , 不会执行 || 后面的判断, 恒 continue .
56         if (ok|| mix(mp[ l ],mp[ r ],w)) continue ;
57         else ok=1;
58         if (ok) ans=i-1;
59     }
60     if (ok==0) ans=m; // 所有的语句都是正确的
61     printf( "%d\n",ans );
62 }
63 return 0;
64 }
```

平面上有 n 个点, 每个点可以在东西南北四个方向上与另外一个点连接, (每个点最多和四个直接连通) 然后有 m 条语句用以表示这 n 个点之间的位置关系: a,b,d,S 表示 a 点在 b 点南方, 距离为 d (N,W,E 分别表示北, 西, 东) 接着有 k 条语句来查询: $a\ b\ t$: 由 m 条语句中的前 t 条能否得出 $a\ b$ 两点间的哈夫曼距离, 如果由 m 条语句中的前 t 条得出 $a\ b$ 两点不连通, 则结果为 -1.

先查询按照查询的 t 由小到大排序, 然后将读入数据读入并查集到相应次序, 两个点是否连通可通过 $find$ 来判断。 $x[i]\ y[i]$ 分别表示点 i 到根节点在横向和竖向的距离。那么如果 uu, vv 两点连通距离就是 $abs(x[uu] - x[vv]) + abs(y[uu] - y[vv])$; 但是输出需要按照查询的顺序输出, 所以可以把查询语句用结构体存储, 并且结构体中一个变量用于存储查询顺序, 那么计算每条语句的 ans 时, 把查询语句结构体数组按照 t 排序, 输出时, 再把查询语句按照查询顺序排序输出。

```

1 const int maxn=40010;
2 const int maxk=10010;
3
4 int pre [maxn] ,x [maxn] ,y [maxn] ,n,m,k ,a ,b ,d ,tx ,ty ;
5 char ss [10];
6
7 struct Query{
8     int index ,u,v,t ,ans;
9     // ans 是该条查询语句的答案
10 }query [maxk];
11
12 struct Read{
13     int u,v,xx ,yy;
14 }read [maxn];
15
16 bool cmp1(Query Q1,Query Q2)
17 { //按查询语句的 t 排序
18     if (Q1.t==Q2.t) return Q1.index<Q2.index ;
19     return Q1.t<Q2.t ;
20 }
21
22 bool cmp2(Query Q1,Query Q2)
23 { //按查询语句的查询顺序排序
24     return Q1.index<Q2.index ;
25 }
26
27 int find (int u)
28 {
29     if (pre [u]==u) return u;
30     int tmp=find (pre [u]);
```

```

31     x[u]=x[u]+x[pre[u]];
32     y[u]=y[u]+y[pre[u]];
33     return pre[u]=tmp;
34 }
35
36 void mix(int a,int b,int tx,int ty)
37 {
38     int fa=find(a);
39     int fb=find(b);
40     if(fa!=fb){
41         pre[fa]=fb;
42         x[fa]=x[b]-x[a]+tx;
43         y[fa]=y[b]-y[a]+ty;
44     }
45 }
46
47 int main()
48 {
49     while(~scanf("%d%d",&n,&m)&&n){
50         for(int i=0;i<=n;i++)
51             pre[i]=i;
52         memset(x,0,sizeof(x));
53         memset(y,0,sizeof(y));
54         for(int i=1;i<=m;i++){
55             scanf("%d%d%d%s",&a,&b,&d,ss);
56             if(ss[0]=='E'){tx=d;ty=0;} // 这里对方向做了统一规定
57             else if(ss[0]=='W'){tx=-d;ty=0;}
58             else if(ss[0]=='S'){tx=0;ty=d;}
59             else if(ss[0]=='N'){tx=0;ty=-d;}
60             read[i].u=a;
61             read[i].v=b;
62             read[i].xx=tx;
63             read[i].yy=ty;
64         }
65         scanf("%d",&k);
66         for(int i=1;i<=k;i++){
67             scanf("%d%d%d",&query[i].u,&query[i].v,&query[i].t);
68             query[i].ans=0;
69             query[i].index=i; //查询顺序标记
70         }
71         sort(query+1,query+k+1,cmp1);
72         int now=1;
73         for(int i=1;i<=k;i++){
74             for(int j=now;j<=query[i].t;j++){
75                 a=read[j].u;
76                 b=read[j].v;
77                 tx=read[j].xx;
78                 ty=read[j].yy;
79                 mix(a,b,tx,ty);
80             }
81             int uu=query[i].u;
82             int vv=query[i].v;
83             if(find(uu)==find(vv)) // uu 和 vv 连通{
84                 query[i].ans=abs(x[uu]-x[vv])+abs(y[uu]-y[vv]);
85             }
86             else query[i].ans=-1;
87
88             now=query[i].t+1;
89         }
90         sort(query+1,query+1+k,cmp2);
91         for(int i=1;i<=k;i++)
92             printf("%d\n",query[i].ans);
93     }
94     return 0;
95 }
```

有 n 个人玩石头剪刀布，有且只有一个裁判。除了裁判每个人的出拳形式都是一样的。

- $a < b$ 表示 b 打败 a
- $a = b$ 表示 a 和 b 出拳一样，平手
- $a > b$ 表示 a 打败 b

给出 m 个回合的游戏结果，问能否判断出谁是裁判？如果能还要输出是在哪个回合之后判断出谁是裁判。

枚举和加权并查集。

对于每个人假设其为裁判，然后去掉所有和他有关的匹配，判断是否会出现矛盾。

- $val[i] = 0$: i 和根节点属于同一集合
- $val[i] = 1$: 根节点打败 i
- $val[i] = 2$: i 打败根节点

在寻找根节点的 `find()` 函数中，`val` 的更新函数是： $val[x] = (val[x] + val[pre[x]]) \% 3$

举个例子：找到根节点之前 $val[x] = 1$, $val[pre[x]] = 2$: 表示父节点打败 x ，父节点也打败父节点的父节点。（注意此时 $val[x]$ 是 x 与父节点的关系）所以按照递归的思路，从递归倒数第二层开始 $pre[x]$ 就表示为根节点了，那么 $pre[x]$ 打败根节点。又因为 $pre[x]$ 也打败 x 所以 $val[x] = 0 = (1 + 2) \% 3$; 递归在往上一层时 $pre[x]$ 又表示为根节点了。

再来看看合并操作时的 `val` 关系。 fa, fb 分别为 aa, bb 的根节点， ww 是 aa 与 bb 的关系。

当 $fa = fb$ 时，令 $pre[fa] = fb$. 假设 $val[aa] = 1$, 即 fa 打败 $aa \dots (1), val[bb] = 2$, 即 bb 打败 $fb \dots (2)$, $ww = 1$, 即 bb 打败 $aa \dots (3)$. 则由 (1)(2) 的 aa 和 fb 是同一集合。再由 (1) 得 fa 打败 fb . 即 $val[fa] = 2$. 也就是 $val[fa] = (val[bb] - val[aa] + ww) \% 3$, 但是由于有可能 $val[bb] - val[aa] + ww < 0$, 所以正确的方程是： $val[fa] = (val[bb] - val[aa] + ww + 3) \% 3$. 当 $fa == fb$ 时，那么就要判断是否出现矛盾，如果出现矛盾那么说明 i 不能作为裁判。判断矛盾是： $(val[aa] - val[bb] + 3) \% 3$ 和 ww 是否相等。如果不矛盾，那么就接着读入输入到最后。

还要注意一点就是可能会出现多个裁判，那就是 Can not determine。

```

1 const int maxn = 510;
2 const int maxm = 2010;
3
4 int n,m,aa,bb,ww;
5 int pre[maxn], val[maxn];
6 int line ,tmpline , ans,cnt , flag ;
7 char s ;
8
9 struct Read {
10     int a, b, w;
11 }read[maxm];
12
13 void init()
14 {
15     for (int i = 0; i < maxn; i++){
16         pre[i] = i;
17         val[i]=0;
18     }
19 }
20
21 int find( int x)
22 {
23     if (pre[x]==x) return x;
24     int tmp=find( pre[x]);
25     val[x]=(val[x]+val[pre[x]])%3;
26     return pre[x]=tmp;
27 }
28
29 int main()
30 {
31     while (~scanf( "%d%d", &n, &m)){
32         if (m == 0){
33             //printf("case 1\n");

```

```
34     if (n==1)
35     printf("Player 0 can be determined to be the judge after 0 lines\n");
36     else printf("Can not determine\n");
37     continue;
38 }
39 for (int i = 0; i<m; i++){
40     scanf("%d%c%d", &aa, &s, &bb);
41     read[i].a=aa;
42     read[i].b=bb;
43     if (s == '=') read[i].w=0;
44     else if(s=='<') read[i].w=1;
45     else if(s=='>') read[i].w=2;
46 }
47 line=-1;
48 cnt=0;
49 for (int i=0;i<n;i++)//枚举每个人{
50     init();
51     tmpline=-1;
52     flag=0;
53     for (int j=0;j<m; j++) {
54         aa=read[j].a;
55         bb=read[j].b;
56         ww=read[j].w;
57         if (aa==i || bb==i) continue;
58         //去掉的影响看是否还会出现矛盾i
59         int fa=find(aa);
60         int fb=find(bb);
61         if (fa!=fb){
62             pre[fa]=fb;
63             val[fa]=(val[bb]-val[aa]+ww+3)%3;
64         } else{
65             if ((val[aa]-val[bb]+3)%3!=ww) //出现矛盾{
66                 tmpline=j+1;
67                 //出现矛盾所在行
68                 flag=1;
69                 break;
70             }
71         }
72         if (flag) break;
73     }
74     if (flag==0)//没出现矛盾{
75         ans=i;//可以是裁判i
76         cnt++;
77         if (cnt>=2) break;//裁判数量 >=2
78     }
79     else line=max(line ,tmpline);//
80 }
81 if (cnt==0){
82     //printf("case 2\n");
83     printf("Impossible\n");
84 } else if(cnt>=2){
85     //printf("case 3\n");
86     printf("Can not determine\n");
87 } else{
88     //printf("case 4\n");
89     printf("Player %d can be determined to be the judge after %d lines\n",ans ,line );
90 }
91 }
92 return 0;
93 }
```

8.3 RMQ

用于解决区间最值问题，需要 $O(n \log n)$ 的预处理。

```
1 int n;
2 int data[MAX_N], dp[MAX_N][20];
3
4 void RMQ()
5 {
6     memset(dp, 0, sizeof(dp));
7     int k = (int)log2(n * 1.0);
8     for (int i = 0; i < n; ++i) { dp[i][0] = data[i]; }
9     for (int j = 1; j <= k; ++j) { // 注意枚举顺序
10         for (int i = 0; i + (1 << j) - 1 < n; ++i) {
11             //一定要 -1，因为从 i 开始的第 1<<j 个元素下标是 i + (1 << j) - 1
12             int k = i + (1 << (j - 1));
13             dp[i][j] = max(dp[i][j - 1], dp[k][j - 1]);
14         }
15     }
16 }
17
18 // 查询区间 [left, right] 的最值
19 int len = right - left + 1;
20 int e = (int)log2(len * 1.0);
21 int k = right - (1 << e);
22 res = max(dp[left][e], dp[k][e]);
```

8.4 线段树

8.4.1 矩形并的周长

```

1 #define lson(x) ((x)<<1)
2 #define rson(x) (((x)<<1)|1)
3 const int maxn = 5050;
4 // maxn 是最大矩形数
5
6 int n, x1, x2, yy1, y2;
7 int xx[maxn << 2];
8
9 struct SegTree{
10     int left, right, len, num, flag;
11     // len 是有效长度, num 是有几条线段, flag 记录状态
12     bool lcover, rcover;
13     // 记录区间左右端点是否存在竖边
14 }segtree[maxn << 4];
15
16 struct Line{
17     int x1, x2, y, flag;
18     bool operator < (const Line a) const{
19         return y < a.y;
20     }
21 }line[maxn << 2];
22
23 void build(int left, int right, int cur)
24 {
25     segtree[cur].left = left;
26     segtree[cur].right = right;
27     segtree[cur].len = segtree[cur].num = segtree[cur].flag = 0;
28     segtree[cur].lcover = segtree[cur].rcover = false;
29     if(left + 1 == right) return ;
30     int mid = (left+right) >> 1;
31     build(left, mid, lson(cur));
32     build(mid, right, rson(cur));
33 }
34
35 void calc_len(int cur)
36 {
37     int left = segtree[cur].left;
38     int right = segtree[cur].right;
39     if(segtree[cur].flag > 0){
40         segtree[cur].len = xx[right] - xx[left];
41         segtree[cur].num = 1;
42         segtree[cur].lcover = segtree[cur].rcover = true;
43     }else if(left + 1 == right){
44         segtree[cur].len = segtree[cur].num = 0;
45         segtree[cur].lcover = segtree[cur].rcover = false;
46     }else{
47         segtree[cur].len = segtree[lson(cur)].len + segtree[rson(cur)].len;
48         segtree[cur].num = segtree[lson(cur)].num + segtree[rson(cur)].num;
49         segtree[cur].lcover = segtree[lson(cur)].lcover;
50         segtree[cur].rcover = segtree[rson(cur)].rcover;
51         if(segtree[lson(cur)].rcover && segtree[rson(cur)].lcover)
52             // 左右儿子的线段可以衔接
53             segtree[cur].num--;//合并成一个线段, 所以线段数量-1
54     }
55     return ;
56 }
57
58 void update(int a, int b, int flag, int cur)
59 {
60     int left = segtree[cur].left;
61     int right = segtree[cur].right;

```

```

62     if(left==a && right==b){
63         segtree[cur].flag += flag;
64         calc_len(cur);
65         return;
66     }
67     if(left + 1 == right) return;
68     int mid = (left + right) >> 1;
69     if(b <= mid) update(a, b, flag, lson(cur));
70     else if(a >= mid) update(a, b, flag, rson(cur));
71     else{
72         update(a, mid, flag, lson(cur));
73         update(mid, b, flag, rson(cur));
74     }
75     calc_len(cur);
76 }
77
78 int main()
79 {
80     while(~scanf("%d", &n)){
81         int tot = 0;
82         for(int i = 0; i < n; i++){
83             scanf("%d%d%d%d", &x1, &yy1, &x2, &y2);
84             line[tot].x1 = x1, line[tot].x2 = x2;
85             line[tot].y = yy1, line[tot].flag = 1;
86             line[tot+1].x1 = x1, line[tot+1].x2 = x2;
87             line[tot+1].y = y2, line[tot+1].flag = -1;
88             xx[tot] = x1, xx[tot+1] = x2;
89             tot += 2;
90         }
91         sort(xx, xx + tot);
92         int m = unique(xx, xx + tot) - xx;
93         build(0, m - 1, 1);
94         sort(line, line + tot);
95         int ans = 0, last = 0, a, b;
96         for(int i=0;i<tot-1;i++){
97             a=lower_bound(xx,xx+m,line[i].x1)-xx;
98             b=lower_bound(xx,xx+m,line[i].x2)-xx;
99             update(a,b,line[i].flag,1);
100            ans+=(line[i+1].y-line[i].y)*segtree[1].num*2;
101            //竖边数量是线段数量的两倍
102            ans+=abs(segtree[1].len-last);
103            // 因为加上边时/ flag 为 -1 , 这时线段树中这条边就不存在了
104            // 但是求周长时是应该加上的
105            // 而且在上一棵线段树中是有的, 所以要用abs
106            last=segtree[1].len;
107        }
108        a=lower_bound(xx,xx+m,line[tot-1].x1)-xx;
109        b=lower_bound(xx,xx+m,line[tot-1].x2)-xx;
110        update(a,b,line[tot-1].flag,1);
111        ans+=abs(segtree[1].len-last);
112        printf("%d\n",ans);
113    }
114    return 0;
115 }

```

8.4.2 矩形交的面积

```

1 #define lson(x) (x<<1)
2 #define rson(x) ((x<<1)|1)
3 using namespace std;
4
5 const int maxn=5050;
6
7 int T,n;

```

```

8 double x1,x2,yy1,y2,xx[maxn<<2];
9
10 struct SegTree{
11     int left,right,flag;
12     double len1,len2;
13     // len1 是覆盖一次的长度， len2 是覆盖不止一次的长度
14 }segtree[maxn<<4];
15
16 struct Line{
17     double x1,x2,y;
18     int flag;
19     bool operator < (const Line a) const{
20         return y<a.y;
21     }
22 }line[maxn<<2];
23
24 inline void build(int left,int right,int cur)
25 {
26     segtree[cur].left=left;
27     segtree[cur].right=right;
28     segtree[cur].flag=0;
29     segtree[cur].len1=segtree[cur].len2=0;
30     if(left+1==right) return ;
31     int mid=(left+right)>>1;
32     build(left,mid,lson(cur));
33     build(mid,right,rson(cur));
34 }
35
36 inline void calc_len(int cur)
37 {
38     int left=segtree[cur].left;
39     int right=segtree[cur].right;
40     if(segtree[cur].flag>0) segtree[cur].len1=xx[right]-xx[left];
41     else if(left+1==right) segtree[cur].len1=0;
42     else segtree[cur].len1=segtree[lson(cur)].len1+segtree[rson(cur)].len1;
43     // flag>1 说明整个区间有不止一次覆盖， flag=1 说明整个区间是完全覆盖，
44     // 但是在这个区间下可能有小区间之前已经被完全覆盖了，所以要加上子树中被覆盖一次的的区间长度
45     // flag<1 那么就是左右子树覆盖两次长度之和
46     if(segtree[cur].flag>1) segtree[cur].len2=xx[right]-xx[left];
47     else if(left+1==right) segtree[cur].len2=0;
48     // 叶子结点且 flag<=1，则覆盖两次的长度只能是0
49     else if(segtree[cur].flag==1)
50         segtree[cur].len2=segtree[lson(cur)].len1+segtree[rson(cur)].len1;
51     else segtree[cur].len2=segtree[lson(cur)].len2+segtree[rson(cur)].len2;
52
53     return ;
54 }
55
56 inline void update(int a,int b,int flag,int cur)
57 {
58     int left=segtree[cur].left;
59     int right=segtree[cur].right;
60     if(left==a&&right==b){
61         segtree[cur].flag+=flag;
62         calc_len(cur);
63         return ;
64     }
65     if(left+1==right) return ;
66     int mid=(left+right)>>1;
67     if(b<=mid) update(a,b,flag,lson(cur));
68     else if(a>=mid) update(a,b,flag,rson(cur));
69     else{
70         update(a,mid,flag,lson(cur));
71         update(mid,b,flag,rson(cur));
72     }
}

```

```

73     calc_len(cur);
74 }
75
76 int main()
77 {
78     scanf("%d",&T);
79     while(T--){
80         scanf("%d",&n);
81         int tot=0;
82         for(int i=0;i<n;i++){
83             scanf("%lf %lf %lf %lf",&x1,&yy1,&x2,&y2);
84             line[tot].x1=x1, line[tot].x2=x2;
85             line[tot].y=yy1, line[tot].flag=1;
86             line[tot+1].x1=x1, line[tot+1].x2=x2;
87             line[tot+1].y=y2, line[tot+1].flag=-1;
88             xx[tot]=x1, xx[tot+1]=x2;
89             tot+=2;
90         }
91         sort(xx,xx+tot);
92         int m=unique(xx,xx+tot)-xx;
93         build(0,m-1,1);
94         sort(line,line+tot);
95         double ans=0;
96         int a,b;
97         for(int i=0;i<tot-1;i++){
98             a=lower_bound(xx,xx+m,line[i].x1)-xx;
99             b=lower_bound(xx,xx+m,line[i].x2)-xx;
100            update(a,b,line[i].flag,1);
101            //printf(" i=%d len=%2f\n",i,segtree[1].len2);
102            ans+=segtree[1].len2*(line[i+1].y-line[i].y);
103        }
104        printf("%.2f\n",ans);
105    }
106    return 0;
107 }
```

8.4.3 矩形并面积

```

1 #define lson(x) (x<<1)
2 #define rson(x) ((x<<1)|1)
3 using namespace std;
4
5 const int maxn=110;
6
7 int n,cases=0;
8 double x1,x2,yy1,y2,xx[maxn<<2];
9
10 struct SegTree{
11     int left,right,flag;
12     // 只有 flag=1 时该区间才会完全是有效长度
13     double len;
14 } segtree[maxn<<4];
15
16 struct Line{
17     int flag;
18     double x1,x2,y;
19     bool operator < (const Line a) const{
20         return y<a.y;
21     } // 将横边按所在 y 值从小到大排序
22 } line[maxn<<2];
23
24 void build(int left,int right,int cur)
25 {
26     segtree[cur].left=left;
27 }
```

```

27     segtree[cur].right=right;
28     segtree[cur].flag=0;
29     segtree[cur].len=0;// len 是区间所存的有效横边长度
30     if(left+1==right) return;
31     int mid=(left+right)>>1;
32     build(left,mid,lson(cur));
33     build(mid,right,rson(cur));
34     // 如果是按照 [left,mid],[mid+1,right] 建树
35     // 那么会有一段横坐标 [mid,mid+1] 丢失
36 }
37
38 void calc_len(int cur)
39 {
40     int flag=segtree[cur].flag;
41     int left=segtree[cur].left;
42     int right=segtree[cur].right;
43     if(flag) segtree[cur].len=xx[right]-xx[left];
44     // 整个区间长度是区间左右端点所代表的横坐标之差
45     else if(left+1==right) segtree[cur].len=0; // 叶子结点且 flag<=0
46     else segtree[cur].len=segtree[lson(cur)].len+segtree[rson(cur)].len;
47 }
48
49 void update(int a,int b,int flag,int cur)
50 {
51     int left=segtree[cur].left;
52     int right=segtree[cur].right;
53     if(left==a&&right==b){
54         segtree[cur].flag+=flag;
55         calc_len(cur);
56         return;
57     }
58     if(left+1==right) return;
59     int mid=(left+right)>>1;
60     if(b<=mid) update(a,b,flag,lson(cur));
61     else if(a>=mid){
62         update(a,b,flag,rson(cur));
63     }else{
64         int mid=(left+right)>>1;
65         update(a,mid,flag,lson(cur));
66         update(mid,b,flag,rson(cur));
67     }
68     calc_len(cur);
69 }
70
71 int main()
72 {
73     while((~scanf("%d",&n))&&n){
74         int tot=0;
75         for(int i=0;i<n;i++){
76             scanf("%lf %lf %lf %lf",&x1,&yy1,&x2,&y2);
77             xx[tot]=x1,xx[tot+1]=x2;
78             line[tot].x1=x1,line[tot].x2=x2;
79             line[tot].y=yy1,line[tot].flag=1;
80             line[tot+1].x1=x1,line[tot+1].x2=x2;
81             line[tot+1].y=y2,line[tot+1].flag=-1;
82             tot+=2;
83         }
84         sort(xx,xx+tot);
85         int m=unique(xx,xx+tot)-xx; // 将横坐标去重
86         build(0,m-1,1);
87         double ans=0;
88         sort(line,line+tot);
89         //for(int i=0;i<tot;i++)
90         //printf("%lf %lf %lf\n",line[i].x1,line[i].x2,line[i].y);
91         for(int i=0;i<tot-1;i++){

```

```
92     int a=lower_bound(xx,xx+m,line[i].x1)-xx;
93     int b=lower_bound(xx,xx+m,line[i].x2)-xx;
94     update(a,b,line[i].flag,1);
95     ans+=segtree[1].len*(line[i+1].y-line[i].y);
96   }
97   printf("Test case #%-d\n",++cases);
98   printf("Total explored area: %.2f\n\n",ans);
99 }
100 return 0;
101 }
```

8.5 树状数组

```

1 C[i] = a[i - 2^k + 1] + a[i - 2^k] + ... + a[i];
2 // k 是 i 二进制表示中末尾连续的 0 的个数
3
4 int lowbit(int x)
5 {
6     return x & (-x); // 一个负数的二进制表示与其相反数的二进制之和等于2^32
7     // return x & (x ^ (x - 1));
8     // return x & (~(x - 1));
9 }
```

8.5.1 单点更新，区间求和

更新所有牵动的区间，求和时求所有区间的所有更新，每个区间覆盖若干子区间但是每个子区间的更新都会体现在父区间中。

定理： $A[k]$ 所牵动的序列为： $C[p_1], C[p_2], \dots$ ，其中 $p_1 = k, p_{i+1} = p_i + 2^{l_i}$ ($i \geq 1, l_i$ 指 p_i 二进制中末尾 0 的个数)。

典型应用：求逆序对数。

8.5.2 区间更新，单点求值

比如区间 $[L, R]$ 要加上 $value$ ，此时数组 $C[]$ 存的是 C 所覆盖的区间改变值之和，那么查询所有覆盖 x 的 C 。

区间更新和单点查询复杂度： $O(\log n)$ 。

8.5.3 一维区间更新和区间求和

C 中记录的是区间该变量， $query(x)$ 是 $data[x]$ 的该变量，设求 $[1, R]$ 区间和，原数据前缀和为 $pre[]$ ，则：

$$\begin{aligned}
Ans(R) &= pre[R] + \sum_{i=1}^R query(i) \\
&= pre[R] + \sum_{i=1}^R C[i] * (R - i + 1) \\
&= pre[R] + (R + 1) * \sum_{i=1}^R C[i] - \sum_{i=1}^R i * C[i]
\end{aligned}$$

另开一个数组 $B[i] = i * C[i]$ ，记录并更新即可。

```

1 // POJ 3468
2 int n, m;
3 ll pre[MAX_N], B[MAX_N], C[MAX_N];
4
5 struct BIT {
6     ll C[MAX_N], B[MAX_N];
7
8     void init() {
9         memset(B, 0, sizeof(B));
10        memset(C, 0, sizeof(C));
11    }
12    int lowbit(const int& x) const {
13        return x & (-x);
14    }
15    ll seg_query(const int& x) const {
```

```

16     ll ret1 = 0, ret2 = 0;
17     for (int i = x; i > 0; i -= lowbit(i)) {
18         ret1 += C[i], ret2 += B[i];
19     }
20     return ret1 * (x + 1) - ret2;
21 }
22 void seg_update(const int& x, const ll& value) {
23     for (int i = x; i <= n; i += lowbit(i)) {
24         C[i] += value, B[i] += value * x; // 注意这里更新 value * x
25     }
26 }
27 } bit;
28
29 int main()
30 {
31     while (~scanf("%d%d", &n, &m)) {
32         bit.init();
33         for (int i = 1; i <= n; ++i) {
34             ll t;
35             scanf("%lld", &t);
36             if (i == 1) pre[1] = t;
37             else pre[i] = pre[i - 1] + t;
38         }
39         for (int i = 0; i < m; ++i) {
40             char s[10];
41             int L, R;
42             ll t;
43             scanf("%s", s);
44             if (s[0] == 'Q') {
45                 scanf("%d%d", &L, &R);
46                 printf("%lld\n", bit.seg_query(R) -
47                         bit.seg_query(L - 1) + pre[R] - pre[L - 1]);
48             } else {
49                 scanf("%d%d%lld", &L, &R, &t);
50                 bit.seg_update(L, t);
51                 bit.seg_update(R + 1, -t);
52             }
53         }
54     }
55     return 0;
56 }
```

8.5.4 二维区间更新和区间求和

二维的区间更新和单点求值和一维的类似。

$$\begin{aligned}
 \sum_{i=1}^{i=x} \sum_{j=1}^{j=y} query(i, j) &= \sum_{i=1}^{i=x} \sum_{j=1}^{j=y} C[i][j] * (x - i + 1) * (y - j + 1) \\
 &= \sum_{i=1}^{i=x} \sum_{j=1}^{j=y} C[i][j] * (x + 1) * (y + 1) - (y + 1) * \sum_{i=1}^{i=x} \sum_{j=1}^{j=y} i * C[i][j] \\
 &\quad - (x + 1) * \sum_{i=1}^{i=x} \sum_{j=1}^{j=y} j * C[i][j] + \sum_{i=1}^{i=x} \sum_{j=1}^{j=y} i * j * C[i][j]
 \end{aligned}$$

```

1 // 支持区间更新和区间求和操作，测试BZOJ3132
2 struct BIT_2D { // 如果必要且空间允许注意开long long
3     int row, col; // 矩阵大小
4     int B[MAX_N][MAX_N], C[MAX_N][MAX_N], D[MAX_N][MAX_N], E[MAX_N][MAX_N];
5
6     void init() {
7         memset(B, 0, sizeof(B));
8         memset(C, 0, sizeof(C));
9     }
10 }
```

```

9     memset(D, 0, sizeof(D));
10    memset(E, 0, sizeof(E));
11}
12int lowbit(const int& x) const {
13    return x & (-x);
14}
15void seg_update(const int& x, const int& y, const int& value) {
16    for (int i = x; i <= row; i += lowbit(i)) {
17        for (int j = y; j <= col; j += lowbit(j)) {
18            C[i][j] += value;
19            B[i][j] += value * x;
20            D[i][j] += value * y;
21            E[i][j] += value * x * y;
22        }
23    }
24}
25int seg_query(const int& x, const int& y) const {
26    int ret1 = 0, ret2 = 0, ret3 = 0, ret4 = 0;
27    for (int i = x; i > 0; i -= lowbit(i)) {
28        for (int j = y; j > 0; j -= lowbit(j)) {
29            ret1 += C[i][j], ret2 += B[i][j];
30            ret3 += D[i][j], ret4 += E[i][j];
31        }
32    }
33    return ret1 * (x + 1) * (y + 1) - ret2 * (y + 1) - ret3 * (x + 1) + ret4;
34}
35} bit;
36
37// 矩阵左上角为 (a, b) 右下角为 (c, d) , 矩阵从上到下从左到右递增
38// 查询
39ans = bit.seg_query(c, d) + bit.seg_query(a - 1, b - 1);
40ans -= (bit.seg_query(a - 1, d) + bit.seg_query(c, b - 1));
41printf("%d\n", ans);
42// 更新
43scanf("%d", &value);
44bit.seg_update(a, b, value);
45bit.seg_update(c + 1, d + 1, value);
46bit.seg_update(a, d + 1, -value);
47bit.seg_update(c + 1, b, -value);

```

```

1 // CF341D: 子矩阵中每个元素都异或 value 和查询子矩阵所有元素异或和
2 int n, m;
3 ll C[4][MAX_N][MAX_N];
4
5 inline int lowbit(int x)
6 {
7     return x & (-x);
8 }
9
10 int GetId(int x, int y)
11 {
12     int ret = 0;
13     if (x & 1) ret += 1;
14     if (y & 1) ret += 2;
15     return ret;
16 }
17
18 void update(int x, int y, ll value)
19 {
20     int id = GetId(x, y);
21     for (int i = x; i <= n; i += lowbit(i)) {
22         for (int j = y; j <= n; j += lowbit(j)) {
23             C[id][i][j] ^= value;
24         }
25     }

```

```
26 }  
27  
28 ll query(int x, int y)  
29 {  
30     ll ret = 0;  
31     int id = GetId(x, y);  
32     for (int i = x; i > 0; i -= lowbit(i)) {  
33         for (int j = y; j > 0; j -= lowbit(j)) {  
34             ret ^= C[id][i][j];  
35         }  
36     }  
37     return ret;  
38 }  
39  
40 int main()  
41 {  
42     scanf("%d%d", &n, &m);  
43     for (int i = 0; i < m; ++i) {  
44         int id, a, b, c, d;  
45         ll value, ans;  
46         scanf("%d%d%d%d", &id, &a, &b, &c, &d);  
47         if (id == 1) {  
48             ans = query(c, d);  
49             ans ^= query(a - 1, d);  
50             ans ^= query(c, b - 1);  
51             ans ^= query(a - 1, b - 1);  
52             printf("%lld\n", ans);  
53         } else {  
54             scanf("%lld", &value);  
55             update(a, b, value);  
56             update(c + 1, d + 1, value);  
57             update(a, d + 1, value);  
58             update(c + 1, b, value);  
59         }  
60     }  
61     return 0;  
62 }
```

8.6 整体二分

8.6.1 [POJ 2104: 区间第 k 小]

$n \leq 10^5$ 个数和 $m \leq 5000$ 个询问，每次查询区间 $[left, right]$ 第 k_i 小的数是？

首先对于一次询问可以二分答案，然后通过验证比答案大的数有多少个来不断地缩小答案范围直至得到一个准确的答案。而对于多个询问也可以这么做，只不过对每一个询问都需要判定一下，以决定它被划分到哪一个答案的区间里。这个判定过程就是通过比较比二分的 mid 大的数的个数 num 和 k_i 的大小关系。如果比二分的 mid 小的数的个数 num 小于 k_i ，则需要去寻找的答案一定大于 mid ，这时 $k'_i = k_i - num$ 。如果 $num \geq k_i$ ，那么需要寻找的答案一定 $\leq mid$ 。直到递归结束。

时间复杂度： $O((n + m) * \log^2(n + m) * \log(\inf))$

```

1  typedef long long ll;
2  const int MAX_N = 100010;
3  const int MAX_M = 5010;
4  const int inf = (int)(1e9) + 10;
5
6  int n, m;
7  int ans[MAX_M], bit[MAX_N];
8
9  struct Query {
10      int type, x, y, k, id;
11  } Q[MAX_N + MAX_M], Q1[MAX_N + MAX_M], Q2[MAX_N + MAX_M];
12
13 inline int lowbit(int x) { return x & (-x); }
14
15 inline void update(int x, int value) {
16     for (int i = x; i <= n; i += lowbit(i)) {
17         bit[i] += value;
18     }
19 }
20
21 inline int query(int x) {
22     int ret = 0;
23     for (int i = x; i > 0; i -= lowbit(i)) {
24         ret += bit[i];
25     }
26     return ret;
27 }
28
29 void solve(int qL, int qR, int L, int R) {
30     if (qL > qR) return;
31     if (L == R) {
32         for (int i = qL; i <= qR; ++i) {
33             if (Q[i].type == 2) ans[Q[i].id] = L;
34         }
35         return;
36     }
37     int mid = (L + R) >> 1;
38     int tp1 = 0, tp2 = 0;
39     for (int i = qL; i <= qR; ++i) {
40         if (Q[i].type == 1) {
41             if (Q[i].x <= mid) {
42                 update(Q[i].id, Q[i].y); // Q[i].y = 1
43                 Q1[tp1++] = Q[i];
44             } else {
45                 Q2[tp2++] = Q[i];
46             }
47         } else {
48             int small = query(Q[i].y) - query(Q[i].x - 1);
49             if (small >= Q[i].k) Q1[tp1++] = Q[i];
50             else {
51                 Q[i].k -= small;
52                 Q2[tp2++] = Q[i];
53             }
54         }
55     }
56 }
```

```

53     }
54 }
55     }
56     for (int i = 0; i < tp1; ++i) {
57         if (Q1[i].type == 1) update(Q1[i].id, -Q1[i].y);
58     }
59     memcpy(Q + qL, Q1, tp1 * sizeof (Query));
60     memcpy(Q + qL + tp1, Q2, tp2 * sizeof (Query));
61     solve(qL, qL + tp1 - 1, L, mid);
62     solve(qL + tp1, qR, mid + 1, R);
63 }
64
65 int main() {
66     while (~scanf("%d%d", &n, &m)) {
67         for (int i = 1; i <= n; ++i) {
68             scanf("%d", &Q[i].x);
69             Q[i].type = 1, Q[i].y = 1;
70             Q[i].id = i;
71         }
72         for (int i = 1; i <= m; ++i) {
73             scanf("%d%d%d", &Q[n + i].x, &Q[n + i].y, &Q[n + i].k);
74             Q[n + i].type = 2, Q[n + i].id = i;
75         }
76         solve(1, n + m, -inf, inf);
77         for (int i = 1; i <= m; ++i) {
78             printf("%d\n", ans[i]);
79         }
80     }
81     return 0;
82 }
```

8.6.2 [BZOJ 3110 第 k 大数]

$n \leq 50000$ 个位置, $m \leq 50000$ 个操作。操作有两种:

- 1 $a b c$ 的形式表示在第 a 个位置到第 b 个位置, 每个位置加入一个数 $c (abs(c) \leq n)$
- 2 $a b c$ 形式, 表示询问从第 a 个位置到第 b 个位置, 第 $c (c \leq Maxlongint)$ 大的数是多少?(同一位可能有多个数)

整体二分, 对左区间操作 1 中的 $c \leq mid$ 进行区间每个数都加 1, 统计数字个数。然后对于右区间的每个操作 2 进行区间查询数字个数。比较划分。

```

1 typedef long long ll;
2 const int MAX_N = 50010;
3
4 int n, m;
5 ll B[MAX_N], C[MAX_N];
6
7 struct Query {
8     int id, type, a, b, ans;
9     ll c;
10    bool operator < (const Query& rhs) const {
11        return id < rhs.id;
12    }
13 } ques[MAX_N], newq1[MAX_N], newq2[MAX_N];
14
15 inline int lowbit(int x) { return x & -x; }
16
17 ll query(int x) {
18     ll ret1 = 0, ret2 = 0;
19     for (int i = x; i > 0; i -= lowbit(i)) {
20         ret1 += C[i], ret2 += B[i];
21     }
22     return ret1 * (x + 1) - ret2;
```

```

23 }
24
25 void update(int x, int value) {
26     for (int i = x; i <= n; i += lowbit(i)) {
27         C[i] += value, B[i] += value * x;
28     }
29 }
30
31 void solve(int qL, int qR, int left, int right) {
32     if (qL > qR) return;
33     if (left == right) {
34         for (int i = qL; i <= qR; ++i) {
35             if (ques[i].type == 2) ques[i].ans = left;
36         }
37         return;
38     }
39     int mid = (left + right) >> 1;
40     int tp1 = 0, tp2 = 0;
41     for (int i = qL; i <= qR; ++i) {
42         int L = ques[i].a, R = ques[i].b;
43         if (ques[i].type == 1) {
44             if (ques[i].c <= mid) newq1[tp1++] = ques[i];
45             else {
46                 update(L, 1);
47                 update(R + 1, -1);
48                 newq2[tp2++] = ques[i];
49             }
50         } else {
51             int cnt = query(R) - query(L - 1);
52             if (cnt < ques[i].c) {
53                 ques[i].c -= cnt;
54                 newq1[tp1++] = ques[i];
55             } else {
56                 newq2[tp2++] = ques[i];
57             }
58         }
59     }
60     for (int i = 0; i < tp2; ++i) {
61         if (newq2[i].type == 1) {
62             update(newq2[i].a, -1);
63             update(newq2[i].b + 1, 1);
64         }
65     }
66     memcpy(ques + qL, newq1, tp1 * sizeof(Query));
67     memcpy(ques + qL + tp1, newq2, tp2 * sizeof(Query));
68     solve(qL, qL + tp1 - 1, left, mid);
69     solve(qL + tp1, qR, mid + 1, right);
70 }
71
72 int main() {
73     scanf("%d%d", &n, &m);
74     for (int i = 1; i <= m; ++i) {
75         scanf("%d%d%d%lld", &ques[i].type, &ques[i].a, &ques[i].b, &ques[i].c);
76         ques[i].id = i;
77     }
78     solve(1, m, -n, n);
79     sort(ques, ques + m + 1);
80     for (int i = 1; i <= m; ++i) {
81         if (ques[i].type == 2) {
82             printf("%d\n", ques[i].ans);
83         }
84     }
85     return 0;
86 }

```

8.7 cdq 分治 (时间分治)

cdq 分治算法的核心在于：去掉时间的限制，将所有查询要求发生的时刻同化，化动态修改为静态查询。对于某些问题来说可以把某一维的限制通过排序看作时间限制然后运用 cdq 分治。解决三维偏序问题一般要第一维排序，第二维 cdq 分治，第三位借用数据结构：树状数组，线段树，平衡树等。

1. 使用 CDQ 分治的前提

- 修改操作对询问的贡献独立，修改操作互不影响
- 允许使用离线算法

2. 一般步骤

- 将整个操作序列分为两个长度相等的部分【分】
- 递归处理前一部分的子问题【治 1】
- 计算前一部分的子问题中的修改操作对后一部分子问题的影响【治 2】
- 递归处理后一部分子问题【治 3】

3. 分治的复杂度 (k 是一个和 n 无关的多项式)

- $T(n) = 2T(\frac{n}{2}) + O(kn) \rightarrow T(n) = O(kn \log n)$
- $T(n) = 2T(\frac{n}{2}) + O(kn \log n) \rightarrow T(n) = O(kn \log^2 n)$
- $T(n) = 2T(\frac{n}{2}) + O(k) \rightarrow T(n) = O(kn)$

8.7.1 [NEU 1702 三维逆序对]

定义点 $i \leq j$ 当且仅当 $x_i \leq x_j, y_i \leq y_j, z_i \leq z_j$ 。给 $n \leq 10^5$ 个点，对每个点求出小于等于它的点的数量。

```

1 const int MAX_N = 100010;
2
3 int T, n, MaxZ;
4 int bit[MAX_N], same[MAX_N], ans[MAX_N], Z[MAX_N];
5
6 struct Point {
7     int x, y, z, id;
8
9     bool operator == (const Point& rhs) const {
10         return x == rhs.x && y == rhs.y && z == rhs.z;
11     }
12 } P[MAX_N], Q[MAX_N];
13
14 bool xyz(Point a, Point b) {
15     if (a.x != b.x) return a.x < b.x;
16     if (a.y != b.y) return a.y < b.y;
17     return a.z < b.z;
18 }
19
20 bool yzx(Point a, Point b) {
21     if (a.y != b.y) return a.y < b.y;
22     if (a.z != b.z) return a.z < b.z;
23     return a.x < b.x;
24 }
25
26 inline int lowbit(int x) { return x & (-x); }
27
28 void update(int x, int value) {
29     for (int i = x; i <= MaxZ; i += lowbit(i)) {
30         bit[i] += value;
31     }
32 }
```

```

34 int query(int x) {
35     int ret = 0;
36     for (int i = x; i > 0; i -= lowbit(i)) {
37         ret += bit[i];
38     }
39     return ret;
40 }
41
42 void cdq(int left, int right) {
43     if (left == right) return;
44     int mid = (left + right) >> 1, xmid = P[mid].x;
45     cdq(left, mid);
46     int total = 0;
47     for (int i = left; i <= right; ++i) {
48         Q[total++] = P[i];
49     }
50     sort(Q, Q + total, yzx);
51     for (int i = 0; i < total; ++i) {
52         if (Q[i].x <= xmid) update(Q[i].z, 1);
53         else {
54             ans[Q[i].id] += query(Q[i].z);
55         }
56     }
57     for (int i = 0; i < total; ++i) {
58         if (Q[i].x <= xmid) update(Q[i].z, -1);
59     }
60     cdq(mid + 1, right);
61 }
62
63 int main() {
64     scanf("%d", &T);
65     while (T--) {
66         scanf("%d", &n);
67         for (int i = 0; i < n; ++i) {
68             scanf("%d%d%d", &P[i].x, &P[i].y, &P[i].z);
69             P[i].id = i, Z[i] = P[i].z;
70         }
71         sort(Z, Z + n);
72         MaxZ = unique(Z, Z + n) - Z;
73         for (int i = 0; i < n; ++i) {
74             P[i].z = lower_bound(Z, Z + MaxZ, P[i].z) - Z + 1;
75         }
76         sort(P, P + n, xyz);
77         for (int i = 0, j = 0; i < n; ) {
78             while (j < n && P[i] == P[j]) ++j;
79             while (i < j) {
80                 same[P[i].id] = P[j - 1].id;
81                 ++i;
82             }
83         }
84         for (int i = 0; i < n; ++i) {
85             P[i].x = i; // 这一步是必须的
86         }
87         for (int i = 0; i <= MaxZ; ++i) { bit[i] = 0; }
88         for (int i = 0; i < n; ++i) { ans[i] = 0; }
89         cdq(0, n - 1);
90         for (int i = 0; i < n; ++i) {
91             printf("%d\n", ans[same[i]]);
92         }
93     }
94     return 0;
95 }
```

8.7.2 [UVALive 6776 三维 LIS]

定义两个点 i 和 j , 如果满足 $x_i < x_j, y_i < y_j, z_i < z_j$, 那么称: $i < j$, 给 $n \leq 3 * 10^5$ 个点计算 LIS。

先把所有点的 Z 坐标离散化, 然后按照 XYZ 的优先级排序, 考虑 cdq 分治。对于区间 $[left, right]$ 内的点再按照 YZX 的优先级排序, 记录下 $mid + 1$ 的横坐标, 需要特殊处理, 借用两棵树状数组。

```

1 const int MAX_N = 300010;
2
3 int n, MaxZ;
4 int bit[2][MAX_N], dp[MAX_N], Z[MAX_N];
5
6 struct Point {
7     int x, y, z, id;
8 } P[MAX_N], Q[MAX_N];
9
10 bool xyz(Point a, Point b) {
11     if (a.x != b.x) return a.x < b.x;
12     if (a.y != b.y) return a.y < b.y;
13     return a.z < b.z;
14 }
15
16 bool yzx(Point a, Point b) {
17     if (a.y != b.y) return a.y < b.y;
18     if (a.z != b.z) return a.z > b.z;
19     return a.x > b.x;
20 }
21
22 inline int lowbit(int x) { return x & (-x); }
23
24 inline void update(int id, int x, int value) {
25     for (int i = x; i <= MaxZ; i += lowbit(i)) {
26         bit[id][i] = max(bit[id][i], value);
27     }
28 }
29
30 inline int query(int id, int x) {
31     int ret = 0;
32     for (int i = x; i > 0; i -= lowbit(i)) {
33         ret = max(ret, bit[id][i]);
34     }
35     return ret;
36 }
37
38 void Clear(int id, int x) {
39     for (int i = x; i <= MaxZ; i += lowbit(i)) {
40         bit[id][i] = 0;
41     }
42 }
43
44 void cdq(int left, int right) {
45     if (left == right) return;
46     int mid = (left + right) >> 1, midX = P[mid + 1].x;
47     cdq(left, mid);
48     int total = 0;
49     for (int i = left; i <= right; ++i) {
50         Q[total] = P[i];
51         Q[total++].id = i; // 重新编号, 以便下面的比较插入和查询
52     }
53     sort(Q, Q + total, yzx);
54     for (int i = 0; i < total; ++i) {
55         int pos = Q[i].id, tmp;
56         if (pos <= mid) {
57             update(0, Q[i].z, dp[pos]);
58             if (Q[i].x != midX) update(1, Q[i].z, dp[pos]); // 单独建树
59         } else {

```

```

60         if (Q[i].x != midX) tmp = query(0, Q[i].z - 1);
61     else tmp = query(1, Q[i].z - 1);
62     dp[pos] = max(dp[pos], tmp + 1);
63 }
64 for (int i = 0; i < total; ++i) {
65     int pos = Q[i].id;
66     if (pos <= mid) { // 清空树状数组
67         Clear(0, Q[i].z);
68         if (Q[i].x != midX) Clear(1, Q[i].z);
69     }
70 }
71 cdq(mid + 1, right);
72 }
73 }

74 int main() {
75     scanf("%d", &n);
76     for (int i = 0; i < n; ++i) {
77         scanf("%d%d%d", &P[i].x, &P[i].y, &P[i].z);
78         dp[i] = 1, P[i].id = i;
79         Z[i] = P[i].z;
80     }
81     sort(Z, Z + n);
82     MaxZ = unique(Z, Z + n) - Z;
83     for (int i = 0; i < n; ++i) {
84         P[i].z = lower_bound(Z, Z + MaxZ, P[i].z) - Z + 1;
85     }
86     sort(P, P + n, xyz);
87     for (int i = 0; i <= MaxZ; ++i) { bit[0][i] = bit[1][i] = 0; }
88     cdq(0, n - 1);
89     int ans = 0;
90     for (int i = 0; i < n; ++i) {
91         ans = max(ans, dp[i]);
92     }
93     printf("%d\n", ans);
94     return 0;
95 }
96 }
```

8.7.3 [BZOJ 3295 动态逆序对]

$n \leq 10^5$ 个元素依次删除 $m \leq 5 * 10^4$ 个元素，每个元素属于 $1 \sim n$ 中间的正整数，求删除元素之前序列有多少个逆序对。

首先把删除看成逆序插入。定义 x 表示插入时间， y 表示在数组中的初始位置， z 表示元素值的大小，每对逆序对即是三维偏序关系。如果最外层按照 y 排序，对于每个区间 $[left, right]$ 都是可以保证 y 单调递增，然后利用 x 和 mid 的大小关系划分区间。此时在区间 $[left, mid]$ 和 $[mid + 1, right]$ 内部都是可以保证 y 单调递增并且子区间 $[mid + 1, right]$ 的每一个 x 都是比左区间的每一个 x 大的考虑利用左子区间更新右子区间。

- 左边数值大的造成的逆序对

因为此时子区间内部都是 y 单调递增的，所以可以根据枚举的右子区间 $i : mid + 1 \sim right$ 的 y 大小来确定是否将左子区间的元素的 z 信息插入树状数组，此时完全不用管 x 信息，因为右子区间的每个元素的 x 都比左子区间的每个元素的 x 都大。在查询的时候，可以查比当前 i 的 z 小的个数，然后用插入的 z 的总个数减一下，就得到 z 大的元素个数了。第一阶段处理完了，需要根据插入“清空”树状数组信息。

- 右边数小造成的逆序对

遍历右子区间 $i : right \sim mid + 1$ ，将左子区间的 z 也从后往前插入树状数组，这样子可以保证左子区间 y 大的都插进了树状数组，正常查询即可。

时间复杂度： $O(n \log n)$

```

1 const int MAX_N = 100010;
2
```

```

3 int n, m;
4 int bit[MAX_N], vis[MAX_N], pos[MAX_N];
5 ll ans[MAX_N];
6
7 struct Node {
8     int x, y, z;
9 } P[MAX_N], Q[MAX_N];
10
11 inline int lowbit(int x) { return x & -x; }
12
13 void update(int x, int value) {
14     for (int i = x; i <= n; i += lowbit(i)) {
15         bit[i] += value;
16     }
17 }
18
19 int query(int x) {
20     int ret = 0;
21     for (int i = x; i > 0; i -= lowbit(i)) {
22         ret += bit[i];
23     }
24     return ret;
25 }
26
27 void cdq(int left, int right) {
28     if (left >= right) return;
29     int mid = (left + right) >> 1;
30     int tp1 = left, tp2 = mid + 1;
31     for (int i = left; i <= right; ++i) {
32         if (P[i].x <= mid) Q[tp1++] = P[i];
33         else Q[tp2++] = P[i];
34     }
35     tp1 = left;
36     for (int i = mid + 1; i <= right; ++i) {
37         while (tp1 <= mid && Q[tp1].y < Q[i].y) {
38             update(Q[tp1].z, 1);
39             tp1++;
40         }
41         ans[Q[i].x] += (tp1 - left) - query(Q[i].z - 1);
42     }
43     for (int i = left; i < tp1; ++i) update(Q[i].z, -1);
44
45     tp1 = mid;
46     for (int i = right; i >= mid + 1; --i) {
47         while (tp1 >= left && Q[tp1].y > Q[i].y) {
48             update(Q[tp1].z, 1);
49             tp1--;
50         }
51         ans[Q[i].x] += query(Q[i].z - 1);
52     }
53     for (int i = mid; i > tp1; --i) update(Q[i].z, -1);
54     memcpy(P + left, Q + left, (right - left + 1) * sizeof(Node));
55     cdq(left, mid);
56     cdq(mid + 1, right);
57 }
58
59 int main() {
60     scanf("%d%d", &n, &m);
61     for (int i = 1; i <= n; ++i) {
62         scanf("%d", &P[i].z);
63         P[i].x = 0, P[i].y = i, pos[P[i].z] = i;
64     }
65     int sz = n, t;
66     for (int i = 1; i <= m; ++i) {
67         scanf("%d", &t);
68     }
69 }
```

```

68     vis[t] = 1;
69     P[pos[t]].x = sz--;
70 }
71 for (int i = 1; i <= n; ++i) {
72     if (vis[i]) continue;
73     P[pos[i]].x = sz--;
74 }
75 cdq(1, n);
76 for (int i = 1; i <= n; ++i) ans[i] += ans[i - 1];
77 for (int i = n; i > n - m; --i) {
78     printf("%lld\n", ans[i]);
79 }
80 return 0;
81 }
```

8.7.4 [BZOJ 2244 导弹拦截]

拦截系统的工作方式：虽然它的第一发炮弹能够到达任意的高度、并且能够拦截任意速度的导弹，但是以后每一发炮弹都不能高于前一发的高度，其拦截的导弹的飞行速度也不能大于前一发。给出 $n \leq 5 \times 10^4$ 枚导弹的高度和速度，求每枚导弹被拦截到的概率。

实际上就是求每个元素是三维 LIS 上结点的概率。方案数可能很大，需要用 double 存。正反搞两次 cdq 分别求出以第 i 个元素结尾前半部分的 LIS 和以第 i 个元素起始后半部分的 LIS，利用 BIT 更新最大长度和方案数。

```

1 const int MAX_N = 50010;
2
3 int n, Max, m;
4 int Y[MAX_N], Z[MAX_N], sta[MAX_N];
5 double way[MAX_N];
6
7 struct Point {
8     int x, y, z, id;
9     bool operator < (const Point& rhs) const {
10         if (y != rhs.y) return y < rhs.y;
11         if (z != rhs.z) return z < rhs.z;
12         return x < rhs.x;
13     }
14 } P[MAX_N], Q[MAX_N];
15
16 struct Ans {
17     int dp[2]; // dp[0]: 以 i 结尾前半部分的 LIS
18     // dp[1]: 以 i 起始后半部分的 LIS
19     double way[2]; // way[]: 分别是方案数
20 } ans[MAX_N];
21
22 struct Bit {
23     int len;
24     double sum;
25
26     Bit() {}
27     Bit(int _len, double _sum): len(_len), sum(_sum) {}
28 } bit[MAX_N];
29
30 inline int lowbit(int x) { return x & -x; }
31
32 void update(int x, int len, double sum) {
33     for (int i = x; i <= Max; i += lowbit(i)) {
34         if (len > bit[i].len) {
35             if (bit[i].len == 0) sta[m++] = i;
36             bit[i] = Bit(len, sum);
37         } else if (len == bit[i].len) bit[i].sum += sum;
38     }
39 }
```

```

40
41 Bit query(int x) {
42     Bit ret = Bit(0, 0);
43     for (int i = x; i > 0; i -= lowbit(i)) {
44         if (bit[i].len > ret.len) ret = Bit(bit[i].len, bit[i].sum);
45         else if (bit[i].len == ret.len) ret.sum += bit[i].sum;
46     }
47     return ret;
48 }
49
50 void Clear(int x) {
51     for (int i = x; i <= Max; i += lowbit(i)) {
52         bit[i] = Bit(0, 0);
53     }
54 }
55
56 void cdq(int left, int right, int cur) {
57     if (left == right) return;
58     int mid = (left + right) >> 1;
59     int tp1 = left, tp2 = mid + 1;
60     for (int i = left; i <= right; ++i) {
61         if (P[i].x <= mid) Q[tp1++] = P[i];
62         else Q[tp2++] = P[i];
63     }
64     memcpy(P + left, Q + left, (right - left + 1) * sizeof(Point));
65     cdq(left, mid, cur);
66     tp1 = left;
67     m = 0;
68     for (int i = mid + 1; i <= right; ++i) {
69         while (tp1 <= mid && Q[tp1].y <= Q[i].y) {
70             update(Q[tp1].z, ans[Q[tp1].id].dp[cur], ans[Q[tp1].id].way[cur]);
71             tp1++;
72         }
73         int pos = Q[i].id;
74         Bit tmp = query(Q[i].z);
75         if (tmp.len + 1 > ans[pos].dp[cur]) {
76             ans[pos].dp[cur] = tmp.len + 1;
77             ans[pos].way[cur] = tmp.sum;
78         } else if (tmp.len + 1 == ans[pos].dp[cur]) {
79             ans[pos].way[cur] += tmp.sum;
80         }
81     }
82     for (int i = tp1 - 1; i >= left; --i) {
83         Clear(Q[i].z);
84     }
85     cdq(mid + 1, right, cur);
86
87     tp1 = left, tp2 = mid + 1;
88     for (int i = left; i <= right; ++i) {
89         if (tp1 <= mid && (tp2 > right ||
90             P[tp1] < P[tp2])) Q[i] = P[tp1++];
91         else Q[i] = P[tp2++];
92     }
93     memcpy(P + left, Q + left, (right - left + 1) * sizeof(Point));
94 }
95
96 void solve() {
97     sort(Z, Z + n); sort(Y, Y + n);
98     Max = unique(Z, Z + n) - Z;
99     int tot = unique(Y, Y + n) - Y;
100    for (int i = 1; i <= n; ++i) {
101        P[i].y = lower_bound(Y, Y + tot, P[i].y) - Y;
102        P[i].z = lower_bound(Z, Z + Max, P[i].z) - Z;
103        P[i].y = tot - P[i].y, P[i].z = Max - P[i].z;
104    }
105 }
```

```

105 }
106 sort(P + 1, P + n + 1); // yzx
107 cdq(1, n, 0);
108 for (int i = 1; i <= n; ++i) {
109     P[i].x = n - P[i].x + 1;
110     P[i].y = tot - P[i].y + 1;
111     P[i].z = Max - P[i].z + 1;
112 }
113 sort(P + 1, P + n + 1); // yzx
114 cdq(1, n, 1);
115 int len = 0;
116 double sum = 0;
117 for (int i = 1; i <= n; ++i) {
118     if (len < ans[i].dp[0]) {
119         len = ans[i].dp[0];
120         sum = ans[i].way[0];
121     } else if (len == ans[i].dp[0]) {
122         sum += ans[i].way[0];
123     }
124 }
125 printf("%d\n", len);
126 for (int i = 1; i <= n; ++i) {
127     if (i > 1) printf(" ");
128     int tmp = ans[i].dp[0] + ans[i].dp[1] - 1;
129     if (tmp < len) printf("0");
130     else printf("%.5lf", ans[i].way[0] * ans[i].way[1] / sum);
131 }
132 printf("\n");
133 }
134
135 int main() {
136     scanf("%d", &n);
137     for (int i = 1; i <= n; ++i) {
138         scanf("%d%d", &P[i].y, &P[i].z);
139         P[i].x = P[i].id = i;
140         Y[i - 1] = P[i].y, Z[i - 1] = P[i].z;
141         ans[i].dp[0] = ans[i].dp[1] = 1;
142         ans[i].way[0] = ans[i].way[1] = 1;
143     }
144     solve();
145     return 0;
146 }
```

8.7.5 [BZOJ 1176 Mokia]

一个 $W \times W$ 的棋盘，每个格子内有一个数，初始时全部为 0，现在要求维护两种操作：

- 1 $x \ y \ z$ 将格子 (x, y) 内的数加上 z
- 2 $x_1 \ y_1 \ x_2 \ y_2$ 询问矩阵 (x_1, y_1, x_2, y_2) 内所有格子的数的和。

操作 1 ≤ 160000 ，操作 2 ≤ 10000 ， $1 \leq W \leq 2 \times 10^6$ 。

对于询问矩阵 (x_1, y_1, x_2, y_2) ，根据容斥需要添加三个询问 $(x_1 - 1, y_1 - 1), (x_2, y_1 - 1), (x_1 - 1, y_2)$ ，和每个询问的符号 $flag$ ，属于哪个询问，每次查询时将答案连同符号累加到所属询问里。离散化后，建立三维偏序： $x_0 < x_1, y_0 < y_1, id_0 < id_1$ ，需要把所有 $< id$ 的值都加进树状数组，第一维按照优先 x 其次 y 排序，第二维根据 id 进行 cdq 分治，第三维按照 y 插入树状数组并统计。

```

1 const int MAX_N = 200100;
2
3 int S, W, n = 0, Max;
4 int X[MAX_N], Y[MAX_N];
5 ll bit[MAX_N];
6
7 struct Query {
```

```

8     int x, y, flag, value;
9     int id, be;
10    ll ans;
11
12    Query() {}
13    Query(int _x, int _y, int _flag, int _id, int _be, ll _ans):
14        x(_x), y(_y), flag(_flag), id(_id), be(_be), ans(_ans) {}
15    bool operator < (const Query& rhs) const {
16        if (x != rhs.x) return x < rhs.x;
17        if (y != rhs.y) return y < rhs.y;
18        return id < rhs.id;
19    }
20} ques[MAX_N], newq[MAX_N];
21
22 inline int lowbit(int x) { return x & -x; }
23
24 void update(int x, int value) {
25     for (int i = x; i <= Max; i += lowbit(i)) {
26         bit[i] += value;
27     }
28}
29
30 ll query(int x) {
31     ll ret = 0;
32     for (int i = x; i > 0; i -= lowbit(i)) {
33         ret += bit[i];
34     }
35     return ret;
36}
37
38 void cdq(int left, int right) {
39     if (left == right) return;
40     int mid = (left + right) >> 1;
41     int tp1 = left, tp2 = mid + 1;
42     for (int i = left; i <= right; ++i) {
43         if (ques[i].id <= mid) newq[tp1++] = ques[i];
44         else newq[tp2++] = ques[i];
45     }
46     memcpy(ques + left, newq + left, (right - left + 1) * sizeof(Query));
47     cdq(left, mid);
48     tp1 = left;
49     for (int i = mid + 1; i <= right; ++i) {
50         if (ques[i].flag != 0) { // not change operation
51             while (tp1 <= mid && ques[tp1].x <= ques[i].x){
52                 if (ques[tp1].flag == 0) { // change operation
53                     update(ques[tp1].y, ques[tp1].value);
54                 }
55                 tp1++;
56             }
57         }
58         ques[i].ans += query(ques[i].y);
59     }
60     for (int i = tp1 - 1; i >= left; --i) {
61         if (ques[i].flag == 0) {
62             update(ques[i].y, -ques[i].value);
63         }
64     }
65     cdq(mid + 1, right);
66     tp1 = left, tp2 = mid + 1;
67     for (int i = left; i <= right; ++i) {
68         if (tp1 <= mid && (tp2 > right ||
69             ques[tp1] < ques[tp2]))
70             newq[i] = ques[tp1++];
71         else newq[i] = ques[tp2++];
72 }

```

```

73     memcpy(ques + left, newq + left, (right - left + 1) * sizeof(Query));
74 }
75
76 bool cmp(const Query& a, const Query& b) {
77     return a.id < b.id;
78 }
79
80 int main() {
81     scanf("%d%d", &S, &W); // S 无关紧要
82     int tot1 = 0, tot2 = 0, type;
83     while (1) {
84         scanf("%d", &type);
85         if (type == 3) break;
86         ++n;
87         if (type == 1) {
88             scanf("%d%d%d", &ques[n].x, &ques[n].y, &ques[n].value);
89             ques[n].flag = 0, ques[n].id = n;
90             X[tot1++] = ques[n].x, Y[tot2++] = ques[n].y;
91         } else {
92             int x1, y1, x2, y2;
93             scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
94             ques[n] = Query(x1 - 1, y1 - 1, 1, n, n + 3, 0);
95             ques[n + 1] = Query(x2, y1 - 1, -1, n + 1, n + 3, 0);
96             ques[n + 2] = Query(x1 - 1, y2, -1, n + 2, n + 3, 0);
97             ques[n + 3] = Query(x2, y2, 2, n + 3, n + 3, 0);
98             n += 3;
99             X[tot1++] = x2, X[tot1++] = x1 - 1;
100            Y[tot2++] = y2, Y[tot2++] = y1 - 1;
101        }
102    }
103    sort(X, X + tot1); sort(Y, Y + tot2);
104    tot1 = unique(X, X + tot1) - X;
105    tot2 = unique(Y, Y + tot2) - Y;
106    for (int i = 1; i <= n; ++i) {
107        ques[i].x = lower_bound(X, X + tot1, ques[i].x) - X + 1;
108        ques[i].y = lower_bound(Y, Y + tot2, ques[i].y) - Y + 1;
109    }
110    sort(ques + 1, ques + n + 1);
111    Max = tot2;
112    cdq(1, n);
113    sort(ques + 1, ques + n + 1, cmp);
114    for (int i = 1; i <= n; ++i) {
115        if (ques[i].flag == 0) continue;
116        if (ques[i].flag == 2) {
117            printf("%lld\n", ques[i].ans);
118        } else {
119            int be = ques[i].be;
120            ques[be].ans += ques[i].ans * ques[i].flag;
121        }
122    }
123    return 0;
124 }
```

8.7.6 [BZOJ 1429 货币兑换 Cash]

给出 $n \leq 100000$ 天每天 A 和 B 两个股票的单价 a_i 和 b_i (买入和卖出价格), 购买时 A 和 B 两个股票的购买比例 $rate$, 初始时有 S 元, 求 n 天后的最大获利量?

用 $dp[i]$ 表示第 i 天可以的最大获利量, x_i, y_i 分别表示用当天的最大获利量可以获得的 A 和 B 两个股票的数量。

$$dp[i] = x_i * a_i + y_i * b_i$$

$$\frac{x_i}{y_i} = rate_i$$

可以解出对应的 x_i 和 y_i 。考虑 dp 状态转移方程：

$$dp[i] = \max(x_j * a_i + y_j * b_i) \quad (j < i)$$

转化成斜率， k 决策优于 j 决策，不妨设 $x_k < x_j$ 有：

$$x_k * a_i + y_k * b_i > x_j * a_i + y_j * b_i \quad (x_k > x_j) \rightarrow k_i = -\frac{a_i}{b_i} > \frac{y_k - y_j}{x_k - x_j}$$

因此需要以 x_k 为关键字维护一个凸线，维护一个点集 (x_k, y_k) ， x_k 是单调递增的，相邻两点的斜率是单调递减的，相当于一个上凸壳。用左半区间已经更新好的 dp 值求出上凸壳点集，然后用右半区的斜率去切左半区的点集更新 dp 值。左半区间只需要点集 (x, y) ，右半区间只需要斜率 $k_i = -\frac{a_i}{b_i}$ 。处理好左半边时需要保证点集按照坐标排好序，处理右半边时需要保证询问按照斜率排好序，相当于用一系列连续变化的直线去切一些连续点组成的凸壳。

```

1 const int MAX_N = 100100;
2 const double eps = 1e-9;
3 const double inf = 1e18;
4
5 int n, S;
6 int sta[MAX_N];
7 double ans[MAX_N];
8
9 struct Query {
10     int id;
11     double a, b, rate, k;
12     double x, y;
13
14     // 按照斜率 k 从大到小排序
15     bool operator < (const Query& rhs) const {
16         return k > rhs.k;
17     }
18 } Q[MAX_N], P[MAX_N];
19
20 double slope(int i, int j) {
21     if (j == 0) return -inf;
22     if (fabs(P[i].x - P[j].x) <= eps) return inf;
23     return (P[i].y - P[j].y) / (P[i].x - P[j].x);
24 }
25
26 void cdq(int left, int right) {
27     if (left == right) {
28         ans[left] = max(ans[left], ans[left - 1]);
29         P[left].x = ans[left] * P[left].rate / (P[left].a * P[left].rate + P[left].b);
30         P[left].y = P[left].x / P[left].rate;
31         return ;
32     }
33     int mid = (left + right) >> 1;
34     int tp1 = left, tp2 = mid + 1;
35     // 按照 id 划分区间
36     for (int i = left; i <= right; ++i) {
37         if (P[i].id <= mid) Q[tp1++] = P[i];
38         else Q[tp2++] = P[i];
39     }
40     memcpy(P + left, Q + left, (right - left + 1) * sizeof(Query));
41     int top = 0;
42     cdq(left, mid); // 递归处理左区间
43
44     // 从左区间获得上凸壳此时已经保证左区间点集按照坐标排好序,
45     for (int i = left; i <= mid; ++i) {
46         while (top > 1 && slope(sta[top], sta[top - 1])
47                 <= slope(i, sta[top - 1]) + eps) --top;
48         sta[++top] = i;
49     }
50     sta[++top] = 0; // 添加边界
51 }
```

```

52 // 用左区间的凸壳更新右区间，右区间保证斜率从大到小排序
53 for (int i = mid + 1, st = 1; i <= right; ++i) {
54     while (st < top && P[i].k - slope(sta[st], sta[st + 1]) <= eps) ++st;
55     int id = P[i].id;
56     ans[id] = max(ans[id], P[sta[st]].x * P[i].a + P[sta[st]].y * P[i].b);
57 }
58 cdq(mid + 1, right);

59 // 把当前区间 [left, right] 点集按坐标排序
60 tp1 = left, tp2 = mid + 1;
61 for (int i = left; i <= right; ++i) {
62     if (tp1 <= mid && (tp2 > right
63         || P[tp1].x < P[tp2].x
64         || (fabs(P[tp1].x - P[tp2].x) <= eps && P[tp1].y < P[tp2].y))) Q[i] = P[tp1++];
65     else Q[i] = P[tp2++];
66 }
67 memcpy(P + left, Q + left, (right - left + 1) * sizeof(Query));
68 }

69 int main() {
70     scanf("%d%d", &n, &S);
71     ans[0] = S;
72     for (int i = 1; i <= n; ++i) {
73         scanf("%lf%lf%lf", &P[i].a, &P[i].b, &P[i].rate);
74         P[i].k = -P[i].a / P[i].b, P[i].id = i;
75         ans[i] = 0;
76     }
77     sort(P + 1, P + n + 1);
78     cdq(1, n);
79     printf("%.3lf\n", ans[n]);
80     return 0;
81 }
82 }
```

8.7.7 [BZOJ 2961 共点圆]

有 $n \leq 5 * 10^5$ 次操作：

- 0 $x y$ 表示在坐标系中加入一个以 (x, y) 为圆心且过原点的圆
- 1 $x y$ 表示询问点 (x, y) 是否在所有已加入的圆的内部（含圆周）

对于每次询问输出 Yes 或者 No，输入保证圆心严格在 x 轴上方（纵坐标为正），且横坐标非零。

一个点 (x_0, y_0) 处于圆心 (x, y) 且过原点的圆内部等价于

$$(x_0 - x)^2 + (y_0 - y)^2 \leq x^2 + y^2 \quad 2x_0x + 2y_0y \geq x_0^2 + y_0^2$$

也就是符合条件的圆心必须位于半平面 $2x_0x + 2y_0y \geq x_0^2 + y_0^2$ 内部。当 $y_0 > 0$ 时，可得 $y \geq -\frac{x_0}{y_0}x + \frac{x_0^2 + y_0^2}{y_0}$ 。不妨考虑求出在与半平面垂直的方向上的投影最小的点，如果这个点也在半平面内那么必然所有的点都在半平面内，只有凸包上的点才会是想要的点。因此当 $y_0 > 0$ 时，对所有的圆心求一个下凸包，然后对于每个查询点找到在这个下凸包上面斜率最接近 $-\frac{x_0}{y_0}$ 的圆，只要这个圆心满足条件则所有的圆心都满足条件。当 $y_0 < 0$ 时需要维护上凸包。

```

1 const int MAX_N = 500010;
2 const double inf = 1e20;
3 const double eps = 1e-9;
4
5 int n;
6
7 struct Point {
8     int type, id, flag;
9     double x, y, k;
10
11     void input() {
```

```

12     scanf("%d%lf%lf", &type, &x, &y);
13 }
14 bool operator < (const Point& rhs) const {
15     return k < rhs.k;
16 }
17 double dis(const Point& rhs) const {
18     return hypot(x - rhs.x, y - rhs.y);
19 }
20 } P[MAX_N], Q[MAX_N], below[MAX_N], above[MAX_N];
21
22 inline double slope(Point a, Point b) {
23     if (fabs(a.x - b.x) < eps) return inf;
24     else return (a.y - b.y) / (a.x - b.x);
25 }
26
27 void cdq(int left, int right) {
28     if (left == right) return;
29     int mid = (left + right) >> 1;
30     int tp1 = left, tp2 = mid + 1;
31     for (int i = left; i <= right; ++i) {
32         if (P[i].id <= mid) Q[tp1++] = P[i];
33         else Q[tp2++] = P[i];
34     }
35     memcpy(P + left, Q + left, (right - left + 1) * sizeof (Point));
36     cdq(left, mid);
37     int tot1 = 0, tot2 = 0;
38     for (int i = left; i <= mid; ++i) {
39         if (P[i].type) continue;
40         while (tot1 > 1 && slope(above[tot1], above[tot1 - 1])
41                < slope(P[i], above[tot1])) tot1--;
42         above[++tot1] = P[i];
43         while (tot2 > 1 && slope(below[tot2], below[tot2 - 1])
44                > slope(P[i], below[tot2])) tot2--;
45         below[++tot2] = P[i];
46     }
47     int st1 = tot1, st2 = 1;
48     for (int i = mid + 1; i <= right; ++i) {
49         if (!P[i].type) continue;
50         if (P[i].y > 0) {
51             while (st2 < tot2 && slope(below[st2], below[st2 + 1]) < P[i].k) ++st2;
52             if (st2 <= tot2 && below[st2].dis(P[0]) < below[st2].dis(P[i])) P[i].flag = 0;
53         } else {
54             while (st1 > 1 && slope(above[st1], above[st1 - 1]) < P[i].k) --st1;
55             if (st1 >= 1 && above[st1].dis(P[0]) < above[st1].dis(P[i])) P[i].flag = 0;
56         }
57     }
58     cdq(mid + 1, right);
59     tp1 = left, tp2 = mid + 1;
60     for (int i = left; i <= right; ++i) {
61         if (tp1 <= mid && (tp2 > right
62                             || P[tp1].x < P[tp2].x
63                             || (fabs(P[tp1].x - P[tp2].x) <= eps && P[tp1].y < P[tp2].y)))
64             Q[i] = P[tp1++];
65         else Q[i] = P[tp2++];
66     }
67     memcpy(P + left, Q + left, (right - left + 1) * sizeof (Point));
68 }
69
70 bool cmp(Point a, Point b) {
71     return a.id < b.id;
72 }
73
74 int main() {
75     P[0].x = 0, P[0].y = 0;
76     scanf("%d", &n);

```

```
77     for (int i = 1; i <= n; ++i) {
78         P[i].input();
79         P[i].k = -P[i].x / P[i].y;
80         P[i].id = i, P[i].flag = 1;
81     }
82     sort(P + 1, P + n + 1);
83     cdq(1, n);
84     sort(P + 1, P + n + 1, cmp);
85     int first = 1; // 特殊处理询问之前没有圆的情况
86     for (int i = 1; i <= n; ++i) {
87         if (P[i].type) {
88             if (P[i].flag && !first) printf("Yes\n");
89             else printf("No\n");
90         } else if (first) first = 0;
91     }
92     return 0;
93 }
```


Chapter 9

图论

9.1 最小生成树

9.1.1 Kruskal Algorithm

时间复杂度: $O(m * \log m)$

```
1 int Kruskal()
2 { // 结构体存边
3     sort(edge, edge + m);
4     int res = 0;
5     for (int i = 0; i < m; ++i) {
6         int u = edge[i].u, v = edge[i].v, w = edge[i].w;
7         int fu = find(u), fv = find(v);
8         if (fu != fv) {
9             fa[fu] = fv;
10            res += w;
11        }
12    }
13    return res;
14 }
```

9.1.2 Prim Algorithm

时间复杂度: $O(n^2)$

```
1 int Prim()
2 {
3     int res = 0;
4     vis[0] = 1; // 以任意点为起点都可以
5     for (int i = 0; i < n; ++i) {
6         way[i] = dis[i][0]; // 二维数组存边
7         fa[i] = 0; // 用于标记每个点所用到的边的另一端点编号
8     }
9     vis[0] = 1, fa[0] = -1;
10    for (int i = 1; i < n; ++i) {
11        int Min = inf, id;
12        for (int j = 0; j < n; ++j) {
13            if (!vis[j] && way[j] < Min) { // 未连通点中找最近的点
14                Min = way[j], id = j;
15            }
16        }
17        vis[id] = 1; // 标记已连通
18        res += Min;
19        if (fa[id] != -1) { // 记录用到的边
20            vec[id].push_back(fa[id]);
21            vec[fa[id]].push_back(id);
22        }
23        for (int j = 0; j < n; ++j) { // 更新
```

```
24         if (!vis[j] && dis[id][j] < way[j]) {  
25             way[j] = dis[id][j];  
26             fa[j] = id;  
27         }  
28     }  
29     return res;  
30 }  
31 }
```

9.2 拓扑排序

拓扑排序 (*Topological Sorting*) 主要用来解决: $A \rightarrow B$ 表示活动 A 必须在活动 B 之前完成。请给出一个合理的活动顺序。也可以用拓扑排序来判断图中是否存在环。

过程:

- 记录每个点的入度。
- 将入度为 0 的顶点加入队列。
- 依次对入度为 0 的点进行删边操作, 同时将新得到的入度为零的点加入队列。
- 重复上述操作, 直至队列为空。

选择度为 0 的点不同, 一般拓扑排序也就不同。

但要注意字典序最小的拓扑排序和让编号为 1 的人尽量小, 然后让编号为 2 的人尽量小, 让编号为 3 的人尽量小……的区别。

前者是将队列中度为 0 的点, 利用优先队列小元素先出, 这样可以字典序最小。

而后者等价于

对于编号为 1: 除了强制有优先关系的, 在他前面不应该有其他的点

对于编号为 2: 在满足编号 1 的条件下和本身存在优先关系的, 在他前面不应该有其他的点

.... 举个例子:

$n = 4, m = 2$

$3 > 1$

$2 > 4$

符合前者的编号顺序应该是: 2, 3, 1, 4 (字典序最小)

符合后者的编号顺序应该是: 3, 1, 2, 4 (编号 1 尽量靠前)

我们来考虑这个排序在多解时的最终状态的性质: 编号大的靠后

那么我们可以反向建图, 先将编号大的排列出来, 然后将可能解锁的大编号也加进优先队列【并且优先队列是大元素先出】，依次排列就好了。

```

1 //普通的拓扑排序模板
2 int degree[MAX_N], ans[MAX_N];
3 vector<int> vec[MAX_N];
4 queue<int> que;
5
6 int TopoSort(int n)
7 {
8     while (!que.empty()) que.pop();
9     for (int i = 0; i < n; ++i) { // 编号 0--n
10         if (degree[i] == 0) que.push(i);
11     }
12     int total = 0, flag = 0;
13     while (!que.empty()) {
14         if (flag == 0 && que.size() > 1) flag = 1; // 有多个度为 0 的点
15         int cur = que.front();
16         que.pop();
17         ans[total++] = cur; //记录序列
18         for (int i = 0; i < vec[cur].size(); ++i) {
19             int to = vec[cur][i];
20             degree[to]--;
21             if (degree[to] == 0) { //度为 0 的点入队列
22                 que.push(to);
23             }
24         }
25     }
26     if (flag == 0 && total == n) return 1; // 唯一确定
27     for (int i = 0; i < n; ++i) {
28         if (degree[i] > 0) return -1; //有环, 冲突
29     }
30     return 0; //不能确定
31 }
```

9.3 欧拉回路

9.3.1 判断

欧拉回路：从图的某一个顶点出发，图中每条边走且仅走一次，最后回到出发点。

欧拉路径：从图的某一个顶点出发，图中每条边走且仅走一次，最后到达某一个点。

以下所有判断都还有一个限制条件：图连通 [并查集或者 dfs 或者 bfs 判断]。

无向图欧拉回路判断：所有顶点的度数都为偶数。

有向图欧拉回路判断：所有顶点的出度与入度相等。

无向图欧拉路径判断：至多有两个顶点的度数为奇数，其他顶点的度数为偶数。

有向图欧拉路径判断：至多有两个顶点的入度和出度绝对值差 1（若有两个这样的顶点，则必须其中一个出度大于入度，另一个入度大于出度），其他顶点的入度与出度相等。

9.3.2 n 个点和 m 条无向边的图，每条边仅可遍历一次，求图中欧拉路径的条数

[HDU 3081 $n \leq 10^5, m \leq 2 * 10^5$]

相当于每次可以从图中取走一个环或者环连环或者一个单链，求最少需要多少次能把所有边取完。考虑每个连通块（并查集实现）。因为是无向边，如果所有点的度数都是偶数，那么只需要一次就可以把这个连通块遍历完。如果连通块内有 k 个度数为奇数的点，那么需要 $\frac{k+1}{2}$ 次才能把这个连通块遍历完。注意排除孤立点的情况。

```

1 const int MAX_N = 100010;
2
3 int n, m;
4 int fa[MAX_N], degree[MAX_N], num[MAX_N];
5 set<int> s;
6 set<int>::iterator it;
7
8 void init()
9 {
10     for (int i = 0; i <= n; ++i) {
11         num[i] = degree[i] = 0;
12         fa[i] = i;
13     }
14 }
15
16 int find(int x)
17 {
18     return fa[x] == x ? x : fa[x] = find(fa[x]);
19 }
20
21 void wyr()
22 {
23     s.clear();
24     for (int i = 1; i <= n; ++i) {
25         int fi = find(i);
26         s.insert(fi);
27         if (degree[i] & 1) num[fi]++;
28     }
29     int ans = 0;
30     for (it = s.begin(); it != s.end(); ++it) {
31         int x = (*it);
32         if (num[x]) ans += (num[x] + 1) / 2;
33         else if (degree[x]) ans++; //排除孤立点
34     }
35     printf("%d\n", ans);
36 }
37
38 int main()
39 {
40     while (~scanf("%d%d", &n, &m)) {
41

```

```

42     init ();
43     for (int i = 0; i < m; ++i) {
44         int u, v;
45         scanf ("%d%d", &u, &v);
46         degree[u]++, degree[v]++;
47         int fu = find(u), fv = find(v);
48         fa[fu] = fv;
49     }
50     wyr ();
51 }
52 return 0;
53 }
```

9.3.3 单词拼接，字典序最小

[POJ 2337]

给 $n \leq 1000$ 个只含小写字母的单词，如果一个单词 A 的首字母是单词 B 的尾字母那么 A 可以拼接在 B 后面，问这 n 个单词能否拼成一个新的单词，如果可以输出字典序最小的答案。

```

1 const int MAX_N = 1010;
2
3 int T, n, total, m;
4 int in[30], out[30], fa[30], head[30], used[30];
5 int sta[MAX_N];
6
7 struct Dict {
8     char s[25];
9     bool operator < (const Dict& rhs) const {
10         return strcmp(s, rhs.s) >= 0;
11         // 保证链式前向遍历的时候先遍历“小”的单词
12     }
13 } dict[MAX_N];
14
15 struct Edge {
16     int v, next, vis;
17 } edge[MAX_N];
18
19 void AddEdge(int u, int v)
20 {
21     edge[total].v = v, edge[total].vis = 0;
22     edge[total].next = head[u];
23     head[u] = total++;
24 }
25
26 int find(int x)
27 {
28     return fa[x] == x ? x : fa[x] = find(fa[x]);
29 }
30
31 void BuildEdge()
32 {
33     total = 0;
34     sort(dict, dict + n);
35     memset(in, 0, sizeof(in));
36     memset(out, 0, sizeof(out));
37     memset(used, 0, sizeof(used));
38     memset(head, -1, sizeof(head));
39     for (int i = 0; i < 30; ++i) { fa[i] = i; }
40     for (int i = 0; i < n; ++i) {
41         int len = strlen(dict[i].s);
42         int u = dict[i].s[0] - 'a', v = dict[i].s[len - 1] - 'a';
43         AddEdge(u, v); // 每个单词看成一个有向边
44         used[u] = used[v] = 1;
```

```

45     out[u]++, in[v]++;
46     int fu = find(u), fv = find(v);
47     if (fu != fv) {
48         fa[fu] = fv;
49     }
50 }
51
52 int check()
53 {
54     int root = -1, cnt1 = 0, cnt2 = 0, st;
55     for (int i = 0; i < 26; ++i) {
56         if (used[i] == 0) continue;
57         if (root == -1) root = find(i); // 判断连通分量个数
58         else if (root != find(i)) return -1;
59
60         if (in[i] - out[i] == 1) cnt1++;
61         if (out[i] - in[i] == 1) cnt2++, st = i; // 出度恰比入度大1
62         if (abs(in[i] - out[i]) >= 2) return -1;
63     }
64     // 有向图欧拉回路判断
65     if ((cnt1 == 1 && cnt2 == 1) || (cnt1 == 0 && cnt2 == 0)) {
66         if (cnt1 == 1) return st;
67         for (int i = 0; i < 26; ++i) {
68             if (out[i] > 0) return i; // 找到字典序最小的起点
69         }
70     } else return -1;
71 }
72
73 void Euler(int u, int id)
74 {
75     for (int i = head[u]; i != -1; i = edge[i].next) {
76         if (edge[i].vis) continue;
77         edge[i].vis = 1;
78         Euler(edge[i].v, i);
79     }
80     if (id != -1) sta[m++] = id;
81 }
82
83 void wyr()
84 {
85     BuildEdge();
86     int st = check();
87     if (st == -1) {
88         printf("***\n");
89         return;
90     }
91     m = 0;
92     Euler(st, -1);
93     for (int i = n - 1; i >= 0; --i) { // 逆序输出
94         printf("%s", dict[sta[i]].s);
95         if (i) printf(".");
96         else printf("\n");
97     }
98 }
99
100 int main()
101 {
102     int T;
103     scanf("%d", &T);
104     while (T--) {
105         scanf("%d", &n);
106         for (int i = 0; i < n; ++i) {
107             scanf("%s", dict[i].s);
108         }
109     }

```

```

110     wyr();
111 }
112 return 0;
113 }
```

9.3.4 正向反向两次遍历所有边，路径输出

[POJ 2230]

$n \leq 10000$ 个点和 $m \leq 50000$ 条无向边的图，要求正向和反向两次遍历所有的边，输出 $2 * m + 1$ 个顶点编号表示遍历顺序，保证有解。

直接从任意起点遍历所有的边。必须先逆序保存遍历的顶点编号，也就是必须先 dfs 再保存顶点编号。
参考样例：

$$n = 3, m = 2 : (2, 3), (1, 2)$$

如果先保存顶点编号再 dfs 的话结果是：

$$1 \ 2 \ 1 \ 3 \ 2 \ (\text{实际上应是: } 1 \ 2 \ 3 \ 2 \ 1)$$

这和链式前向星的建边顺序有关。

```

1 const int MAX_N = 10010;
2 const int MAX_M = 50010;
3
4 int n, m, total, cnt;
5 int head[MAX_N], res[MAX_M * 2];
6
7 struct Edge {
8     int v, next, vis;
9 } edge[MAX_M * 2];
10
11 inline void AddEdge(int u, int v)
12 {
13     edge[total].v = v;
14     edge[total].next = head[u];
15     edge[total].vis = 0;
16     head[u] = total++;
17 }
18
19 void dfs(int u)
20 {
21     for (int i = head[u]; i != -1; i = edge[i].next) {
22         if (edge[i].vis == 0) {
23             edge[i].vis = 1;
24             dfs(edge[i].v);
25             res[cnt++] = edge[i].v;
26             // 必须先逆序保存结果
27         }
28     }
29 }
30
31 int main()
32 {
33     while (~scanf("%d%d", &n, &m)) {
34         memset(head, -1, sizeof(head));
35         total = cnt = 0;
36         for (int i = 0; i < m; ++i) {
37             int u, v;
38             scanf("%d%d", &u, &v);
39             AddEdge(u, v);
40             AddEdge(v, u);
41         }
42         dfs(1); // 选取任意起点
43         res[cnt++] = 1; // 起点
44     }
45 }
```

```
44     for (int i = cnt - 1; i >= 0; --i) { // 逆序输出
45         printf("%d\n", res[i]);
46     }
47 }
48 return 0;
49 }
```

9.4 dfs 序

9.4.1 CF 396 C

给一个 $n \leq 3 * 10^5$ 个结点，根为 1 的树。初始时每个节点的值均为 0，有 $Q \leq 3 * 10^5$ 种操作：

- 1 $v\ x\ k$ ：对于以 v 为根的子树，如果结点 i 到 v 的距离为 dis ，那么 i 的权值将增加 $x - dis * k$ ，距离就是路径上边的数量， v 结点的权值将增加 x 。
- 2 v ：查询 v 结点的权值

对于每次查询输出答案，结果模 $1e9 + 7$ 。

把以 v 为根的子树结点转化成 dfs 序，并且得到每个结点的深度 $depth[v]$ 。然后对于每次第 1 种操作，以 v 为根的子树上结点 i 相当于权值增加了

$$x - (depth[i] - depth[v]) * k = x + depth[v] * k - depth[i] * k$$

也就是对于 dfs 序上的区间 $[left[v], right[v]]$ 上的每一个结点 i 都需要增加 $x + depth[v] * k$ ，然后减掉 $depth[i] * k$ ，把加减的权值看成两棵树状数组，相当于区间更新，单点查询，维护一下即可。

时间复杂度： $O(Q * \log n)$

```

1 const int MAX_N = 300010;
2 const ll mod = (11)(1e9) + 7;
3
4 int n, d = 0, Q;
5 int depth[MAX_N], left[MAX_N], right[MAX_N];
6 ll add[MAX_N], sub[MAX_N];
7 vector<int> son[MAX_N];
8
9 void dfs(int u, int k) {
10     left[u] = ++d;
11     depth[u] = k;
12     int size = son[u].size();
13     for (int i = 0; i < size; ++i) {
14         dfs(son[u][i], k + 1);
15     }
16     right[u] = d;
17 }
18
19 inline int lowbit(int x) { return x & -x; }
20
21 void update(ll *bit, int x, ll value) {
22     value %= mod;
23     if (value < 0) value += mod;
24     for (int i = x; i <= n; i += lowbit(i)) {
25         bit[i] += value;
26         if (bit[i] >= mod) bit[i] -= mod;
27     }
28 }
29
30 ll query(ll *bit, int x) {
31     ll ret = 0;
32     for (int i = x; i > 0; i -= lowbit(i)) {
33         ret += bit[i];
34         if (ret >= mod) ret -= mod;
35     }
36     return ret;
37 }
38
39 int main() {
40     scanf("%d", &n);
41     for (int i = 2; i <= n; ++i) {
42         int t;
43         scanf("%d", &t);
44         son[t].push_back(i);
}

```

```
45 }
46 dfs(1, 1);
47 scanf("%d", &Q);
48 while (Q--) {
49     int type, id, x, k;
50     scanf("%d", &type);
51     if (type == 1) {
52         scanf("%d%d%d", &id, &x, &k);
53         ll value = 1ll * depth[id] * k % mod + x;
54         update(add, left[id], value);
55         update(add, right[id] + 1, -value);
56         update(sub, left[id], k);
57         update(sub, right[id] + 1, -k);
58     } else {
59         scanf("%d", &id);
60         ll ans = (query(add, left[id]) - query(sub, left[id])) * depth[id] % mod;
61         if (ans < 0) ans += mod;
62         printf("%lld\n", ans);
63     }
64 }
65 return 0;
66 }
```

9.5 LCA

9.5.1 暴力求解

记节点 v 到根的深度为 $depth[v]$ 。那么如果节点 w 是 u 和 v 的公共祖先的话，让 u 向上走 $depth[u] - depth[w]$ 步，让 v 向上走 $depth[v] - depth[w]$ 步，都将走到 w 。因此，首先让 u 和 v 中较深的一个向上走 $|depth[u] - depth[v]|$ 步，再一起一步步向上走，直到走到同一个节点即是 LCA 。

时间复杂度: $dfs: O(n)$, 单次查询: $O(n)$

```

1 // 链式前向星建边, 单向边
2 void dfs(int u, int p, int cur)
3 {
4     fa[u] = p, depth[u] = cur;
5     for (int i = head[u]; i != -1; i = edge[i].next) {
6         dfs(edge[i].v, u, cur + 1);
7     }
8 }
9
10 int LCA(int u, int v)
11 {
12     while (depth[u] > depth[v]) u = fa[u];
13     while (depth[v] > depth[u]) v = fa[v];
14     while (u != v) {
15         u = fa[u];
16         v = fa[v];
17     }
18     return u;
19 }主函数中:
20
21 for (int i = 1; i <= n; ++i) {
22     if (in[i] == 0) {
23         root = i; // 找到入度为 0 的根节点
24         break;
25     }
26 }
27 dfs(root, -1, 0);调用:
28 LCA(U, V);

```

9.5.2 二分搜索

首先对于任意节点 v ，利用其父节点 $parent[v][1]$ 信息，可以通过 $parent[v][2] = parent[parent[v][1]][1]$ 得到其向上走两步所到的节点，再利用这一信息，又可以通过 $parent[v][4] = parent[parent[v][2]][2]$ 得到其向上走四步所到的节点。依此类推，就能够得到其向上走 2^k 步所到的节点 $parent[v][k]$ 。有了 $k = \lfloor \log_2(n) \rfloor$ 以内的所有信息后，就可以二分搜索了。

预处理 $parent[v][k]$: $O(n \log n)$, 单次查询: $O(\log n)$

```

1 // 链式前向星建边, 单向边
2 void dfs(int u, int p, int d)
3 { // 获取每个节点的深度和直接父亲
4     depth[u] = d, fa[u][0] = p;
5     for (int i = head[u]; i != -1; i = edge[i].next) {
6         dfs(edge[i].v, u, d + 1);
7     }
8 }
9
10 void RMQ() // 时间复杂度: O(n log(n))
11 { // 倍增处理每个节点的祖先
12     for (int k = 0; k + 1 < MAX_LOG_N; ++k) {
13         for (int i = 1; i <= n; ++i) {
14             if (fa[i][k] == -1) fa[i][k + 1] = -1;
15             else fa[i][k + 1] = fa[fa[i][k]][k];
16         }
17     }
18 }

```

```

19 int LCA(int u, int v) // 时间复杂度: O(log(n))
20 {
21     // 先把两个节点提到同一深度
22     if (depth[u] > depth[v]) swap(u, v);
23     for (int k = 0; k < MAX_LOG_N; ++k) {
24         if (((depth[v] - depth[u]) >> k) & 1) {
25             v = fa[v][k];
26         }
27     }
28     if (u == v) return v;
29     // 二分搜索计算LCA
30     for (int k = MAX_LOG_N - 1; k >= 0; --k) {
31         if (fa[u][k] != fa[v][k]) {
32             u = fa[u][k];
33             v = fa[v][k];
34         }
35     }
36 }
37     return fa[u][0];
38 }
39 // 主函数中需要: memset(fa, -1, sizeof(fa)); fa[MAX_N][MAX_LOG_N]
40 for (int i = 1; i <= n; ++i) {
41     if (in[i] == 0) {
42         root = i;
43         break;
44     }
45 }
46 dfs(root, -1, 0);
47 RMQ();
48 printf("%d\n", LCA(u, v));

```

9.5.3 基于 RMQ 的算法

将树转为从根 DFS 标号后得到的序列处理。

首先按从根 DFS 访问的顺序得到顶点序列 $vis[i]$ 和对应的深度 $depth[i]$ 。对于每个顶点 v , 记其在 $vis[]$ 中首次出现的下标为 $id[v]$ 。而 $LCA(u, v)$ 就是访问 u 之后到访问 v 之前所经过的节点中离根最近的那个, 假设 $id[u] \leq id[v]$, 那么有:

$$LCA(u, v) = vis[k], \text{ 其中 } k \text{ 是所有 } id[u] \leq i \leq id[v] \text{ 中 } depth[i] \text{ 最小的 } i$$

预处理: $O(n \log n)$, 单次查询: $O(1)$

```

1 int vis[MAX_N * 2], depth[MAX_N * 2], dp[MAX_N * 2][20];
2 // 需要开两倍空间
3 int head[MAX_N], in[MAX_N], id[MAX_N];
4 // 链式前向星建边, 单向边
5 void dfs(int u, int p, int d, int& k)
6 {
7     vis[k] = u; // dfs 访问顺序
8     id[u] = k; // 节点在 vis 中首次出现的下标
9     depth[k++] = d; // 节点对应的深度
10    for (int i = head[u]; i != -1; i = edge[i].next) {
11        int v = edge[i].v;
12        if (v == p) continue;
13        dfs(v, u, d + 1, k); // 递归访问子节点
14        vis[k] = u; // 再次访问
15        depth[k++] = d; // 标记 vis 的深度
16    }
17 }
18
19 void RMQ(int root) // 处理区间深度最小值保存最小值的下标,
20 { // 就是区间左右端点最近公共祖先的下标, 即: vs[Min] = LCA
21     int k = 0;
22     dfs(root, -1, 0, k);
23     int m = k; // m = 2 * n - 1

```

```

24     int e = (int)(log2(m + 1.0)); // 区间长度 m + 1
25     for (int i = 0; i < m; ++i) dp[i][0] = i;
26     for (int j = 1; j <= e; ++j) {
27         for (int i = 0; i + (1 << j) - 1 < m; ++i) {
28             int nxt = i + (1 << (j - 1));
29             if (depth[dp[i][j - 1]] < depth[dp[nxt][j - 1]]) {
30                 dp[i][j] = dp[i][j - 1];
31             } else {
32                 dp[i][j] = dp[nxt][j - 1];
33             }
34         }
35     }
36 }
37
38 int LCA(int u, int v)
39 {
40     int left = min(id[u], id[v]), right = max(id[u], id[v]);
41     int k = (int)(log2(right - left + 1.0)); // 区间长度, 注意用! log2
42     int pos, nxt = right - (1 << k) + 1; // nxt 分界点
43     if (depth[dp[left][k]] < depth[dp[nxt][k]]) {
44         pos = dp[left][k];
45     } else {
46         pos = dp[nxt][k];
47     }
48     return vis[pos];
49 }
50 // 主函数中:
51 RMQ(root);
52 printf("%d\n", LCA(u, v));

```

9.5.4 Tarjan 算法

算法从根节点 $root$ 开始搜索，每次递归搜索所有的子树，然后处理跟当前根节点相关的所有查询。用集合表示一类节点，这些节点跟集合外的点的 LCA 都一样，并把这个 LCA 设为这个集合的祖先。当搜索到节点 x 时，创建一个由 x 本身组成的集合，这个集合的祖先为 x 自己。然后递归搜索 x 的所有儿子节点。当一个子节点搜索完毕时，把子节点的集合与 x 节点的集合合并，并把合并后的集合的祖先设为 x 。因为这棵子树内的查询已经处理完， x 的其他子树节点跟这棵子树节点的 LCA 都是一样的，都为当前根节点 x 。所有子树处理完毕之后，处理当前根节点 x 相关的查询。遍历 x 的所有查询，如果查询的另一个节点 v 已经访问过了，那么 x 和 v 的 LCA 即为 v 所在集合的祖先。

其中关于集合的操作都是使用并查集高效完成。

算法的复杂度为， $O(n)$ 搜索所有节点，搜索每个节点时会遍历这个节点相关的所有查询。如果总的查询个数为 Q ，则总的复杂度为 $O(n + Q)$ 。

```

1 int in[MAX_N], fa[MAX_N], cnt[MAX_N], vis[MAX_N], ancestor[MAX_N];
2 // fa []: 并查集操作, cnt [i]: i 作为 LCA 的次数
3 // vis []: 标记是否访问, ancestor []: 存集合的LCA
4 vector<int> vec[MAX_N], query[MAX_N];
5 // vec [i]: 存 i 指向节点(儿子)信息, query [i]: 存需要查 i 和谁的 LCA
6
7 //采用邻接表建边, 双向边
8 int find(int x)
9 {
10     return fa[x] == x ? x : fa[x] = find(fa[x]);
11 }
12
13 void Union(int x, int y)
14 {
15     int fx = find(x), fy = find(y);
16     if (fx != fy) {
17         fa[fy] = fx;
18     }
19 }
20
21 void Tarjan(int x)

```

```

22 {
23     for (int i = 0; i < vec[x].size(); ++i) {
24         Tarjan(vec[x][i]);
25         Union(x, vec[x][i]); // 注意传参和并查集合并顺序
26     }
27     vis[x] = 1; // 标记已经被访问
28     for (int i = 0; i < query[x].size(); ++i) {
29         if (vis[query[x][i]]) {
30             // x 和 query[x][i] 的 LCA 是 find(query[x][i])
31             cnt[find(query[x][i])]++;
32         }
33     }
34 }
35
36 void init()
37 {
38     for (int i = 0; i <= n; ++i) {
39         fa[i] = ancetor[i] = i;
40         vec[i].clear();
41         query[i].clear();
42     }
43     memset(vis, 0, sizeof(vis));
44     memset(in, 0, sizeof(in));
45     memset(cnt, 0, sizeof(cnt));
46 }
47 // 主函数中:
48 // 找到根节点
49 int root;
50 for (int i = 1; i <= n; ++i) {
51     if (in[i] == 0) {
52         root = i;
53         break;
54     }
55 }
56 Tarjan(root);
57 for (int i = 1; i <= n; ++i) {
58     if (cnt[i]) printf("%d:%d\n", i, cnt[i]);
59     // 输出每个节点作为查询的 LCA 的次数
60 }

```

9.5.5 利用 LCA 获得树上任意两点距离

先用基于 RMQ 算法的求 LCA 的方法求出 LCA。记 $dis[i]$ 为根节点到 i 节点的距离，那么 u 和 v 之间的距离就是：

$$Ans = dis[u] + dis[v] - 2 * dis[LCA(u, v)]$$

时间复杂度：预处理: $O(n \log n)$ ，查询: $O(1)$

```

1 int n, m, total;
2 int head[MAX_N], in[MAX_N], id[MAX_N], dis[MAX_N];
3 int vis[MAX_N * 2], depth[MAX_N * 2], dp[MAX_N * 2][20];
4
5 // 链式前向星建边
6 void dfs(int u, int p, int d, int& k)
7 {
8     vis[k] = u, id[u] = k;
9     depth[k++] = d;
10    for (int i = head[u]; i != -1; i = edge[i].next) {
11        int v = edge[i].v, w = edge[i].w;
12        if (v == p) continue;
13        dis[v] = dis[u] + w;
14        dfs(v, u, d + 1, k);
15        vis[k] = u;
16        depth[k++] = d;
17    }

```

```

18 }
19
20 void RMQ(int root)
21 {
22     int k = 0;
23     dfs(root, -1, 0, k);
24     int mm = k;
25     int e = (int)log2(mm + 1.0);
26     for (int i = 0; i < mm; ++i) dp[i][0] = i;
27     for (int j = 1; j <= e; ++j) {
28         for (int i = 0; i + (1 << j) - 1 < mm; ++i) {
29             int nxt = i + (1 << (j - 1));
30             if (depth[dp[i][j - 1]] < depth[dp[nxt][j - 1]]) {
31                 dp[i][j] = dp[i][j - 1];
32             } else {
33                 dp[i][j] = dp[nxt][j - 1];
34             }
35         }
36     }
37 }
38
39 int LCA(int u, int v)
40 {
41     int left = min(id[u], id[v]), right = max(id[u], id[v]);
42     int k = (int)log2(right - left + 1.0);
43     int pos, nxt = right - (1 << k) + 1;
44     if (depth[dp[left][k]] < depth[dp[nxt][k]]) {
45         pos = dp[left][k];
46     } else {
47         pos = dp[nxt][k];
48     }
49     return dis[u] + dis[v] - 2 * dis[vis[pos]];
50 }
51
52 void init()
53 {
54     total = 0;
55     memset(head, -1, sizeof(head));
56     memset(vis, 0, sizeof(head));
57     memset(in, 0, sizeof(in));
58     memset(dis, 0, sizeof(dis));
59     for (int i = 0; i <= n; ++i) { fa[i] = i; }
60 }
61
62 int main()
63 {
64     while (~scanf("%d%d", &n, &m)) {
65         init();
66         for (int i = 0; i < m; ++i) {
67             int u, v, w;
68             char s[10];
69             scanf("%d%d%d%s", &u, &v, &w, s);
70             AddEdge(u, v, w); // 理论上单向边也可以
71             // AddEdge(v, u, w);
72             in[v]++;
73         }
74         int root;
75         for (int i = 1; i <= n; ++i) {
76             if (in[i] == 0) {
77                 root = i;
78                 break;
79             }
80         }
81         RMQ(root);
82         scanf("%d", &m);

```

```
83     while (m--) {
84         int u, v;
85         scanf("%d%d", &u, &v);
86         printf("%d\n", LCA(u, v));
87     }
88 }
89 return 0;
90 }
```

9.6 最小树形图

有向图的最小生成树。

9.6.1 朱刘算法

朱 - 刘 Edmonds 算法步骤:

1. 找到除了 $root$ 以为其他点的权值最小的入边。用 $In[i]$ 记录
2. 如果出现除了 $root$ 以为存在其他孤立的点，则不存在最小树形图
3. 找到图中所有的环，并对环进行缩点，重新编号
4. 更新其他点到环上的点的距离，如：环中的点有 $Vk1\ Vk2 \dots Vki$ 总共 i 个，用缩成的点叫 Vk 替代，则在压缩后的图中，其他所有不在环中点 v 到 Vk 的距离定义如下：

$$gh[v][Vk] = \min_{j=1}^i gh[v][Vkj] - \min_{j=1}^i cost[Vkj] \quad (1 \leq j \leq i)$$

即所有到环上点的权减去环上点的最小入边权。

而 Vk 到 v 的距离为

$$gh[Vk][v] = \min_{j=1}^i gh[Vkj][v] \quad (1 \leq j \leq i)$$

从环上点出发的边的边权为环上所有点到终点的边权的最小值

5. 重复 3, 4 直到没有环为止。

时间复杂度: $O(nm)$

9.6.2 有固定根

POJ 3164 Command Network

给出 n 个点 (下标从 $1 \sim n$) 的坐标和 m 条单向边 $[i, j]$ 表示可以从 i 点建一条边到 j , 权值是两点距离, 求将这 n 个点连通的最小边权和。如果无法连通输出 "poor snoopy", 否则输出最小边权和。

```

1 const int MAX_N = 110;
2 const int MAX_M = 10010;
3 const double INF = 1e20;
4 const double eps = 1e-8;
5
6 int n, m;
7 int NV, NE;
8 // NV 和 NE 分别是结点和边个数结点下标从( 0—NV - 1, 边下标从 0— NE - 1)
9 int ID[MAX_N], vis[MAX_N], pre[MAX_N];
10 //结点重新编号后的编号、寻找环时是否访问、结点的前驱结点
11 double In[MAX_N]; //结点入度最小值
12
13 struct Point{
14     double x, y;
15
16     Point() {}
17     Point(double _x, double _y) : x(_x), y(_y) {}
18
19     double dis(const Point& rhs) const {
20         return hypot(x - rhs.x, y - rhs.y);
21     }
22 } point[MAX_N];
23
24 struct Edge{
25     int u, v;
26     double w;
27
28     Edge() {}
29     Edge(int _u, int _v, double _w) : u(_u), v(_v), w(_w) {}

```

```

30 } edge[MAX_M];
31
32 //ZLEdmonds Algorithm
33 double Directed_MST( int root )
34 {
35     double res = 0, w;
36     int u, v;
37     while(1){
38         // 1. 找最小入边
39         for( int i = 0; i < NV; i++) In[ i ] = INF;
40         //将所有结点的入度设为-1
41         for( int i = 0; i < NE; i++){
42             u = edge[ i ].u;
43             v = edge[ i ].v;
44             w = edge[ i ].w;
45             if( u != v && w < In[ v ]) { //消除自环影响
46                 In[ v ] = w;
47                 pre[ v ] = u;
48             }
49         }
50         for( int i = 0; i < NV; i++){
51             //检查是否存在除 root 独立点若存在则不存在最小树形图,
52             if( i == root ) continue;
53             if( In[ i ] == INF ) return -1; // i 是独立点
54         }
55         // 2. 找环
56         int cnt = 0; //记录所有结点重新标号后新结点数量
57         memset( vis , -1, sizeof( vis ) );
58         memset( ID , -1, sizeof( ID ) );
59         In[ root ] = 0; // 将根节点入度重设为0
60         for( int i = 0; i < NV; i++){ // 遍历所有结点找环
61             res += In[ i ]; // 结果加上当前结点的最少入度
62             v = i;
63             while( vis[ v ] != i && ID[ v ] == -1 && v != root ){
64                 //将 v 所在有向环或有向单链上的所有结点收缩为() i 结点
65                 vis[ v ] = i;
66                 v = pre[ v ];
67             }
68             if( v != root && ID[ v ] == -1){ //只有有向环时重新标号
69                 for( int u = pre[ v ]; u != v; u = pre[ u ]){
70                     //将 i 所在有向环上所有结点收缩为结点 cnt
71                     ID[ u ] = cnt;
72                 }
73                 ID[ v ] = cnt++;
74             }
75         }
76         if( cnt == 0 ) break;
77         //只剩下了 root 所在的有向环
78         for( int i = 0; i < NV; ++i){
79             if( ID[ i ] == -1) ID[ i ] = cnt++;
80             // root 所在有向环上结点或者其他有向环外接单链上点重新标号
81         }
82         // 3. 建新图
83         for( int i = 0; i < NE; i++){ // 更新其他点到环上的距离
84             u = edge[ i ].u;
85             v = edge[ i ].v;
86             edge[ i ].u = ID[ u ];
87             edge[ i ].v = ID[ v ];
88             if( edge[ i ].u != edge[ i ].v){ // 收缩点后不是同一结点
89                 edge[ i ].w -= In[ v ];
90             }
91         }
92         NV = cnt; // 更新新图的结点数量和根节点编号
93         root = ID[ root ];
94     }

```

```

95     return res;
96 }
97
98 int main()
99 {
100    while(~scanf("%d%d", &n ,&m)){
101        NV = n, NE = m;
102        for( int i = 1; i <= n; ++i){
103            scanf("%lf%lf", &point [ i ].x, &point [ i ].y);
104        }
105        for( int i = 0; i < m; ++i){
106            int u, v;
107            scanf("%d%d", &u, &v);
108            if( u != v) edge[ i ].w = point [ u ].dis( point [ v ]);
109            else edge[ i ].w = INF; // 消除自环影响
110            edge[ i ].u = u - 1, edge[ i ].v = v - 1; // 保证结点下标从 0--NV-1
111        }
112        double ans = Directed_MST(0);
113        if(fabs(ans + 1) <= eps) printf("poor snoopy\n");
114        else printf("%.2f\n", ans);
115    }
116    return 0;
117 }
```

9.6.3 无固定根

HDU 2121 Ice_cream's world II

有 n 个城市 (编号 $0 \sim n-1$) 和 m 条边需要在 n 个城市中选择一个城市作为首都, 使得从首都到其他城市的都能连通而且路径权值和要最小。如果不能找到这样的首都输出 -1, 否则输出最小的路径权值和相应的首都, 如果在同一最小路径权值下有多个首都选择, 输出最小编号的那个。

如果原图是可以生成最小树形图, 因为只添加了一个新的结点 n , 且该结点 n 到 $0 \sim n-1$ 的边权为 $(sum + 1)(sum$ 原图边权和), 设新图求出的最小树形图的权值和为 ans , 新图的最小树形图的权值只比原图多了连通 n 结点的边权即 sum , 那么 $ans - sum$ 后即应为原图的最小树形图的权值和, 如果大于 sum , 说明 n 结点不止和原图中的一个结点相连, 所以原图不存在最小树形图。当然如果新图不存在最小树形图, 原图更不可能生成最小树形图。

寻找原图的最小树形图的根, 如果有多解, 输出最小编号根。

原图的最小树形图的根节点即是和 n 结点相连的那个节点所在边的编号设为 $real_root$, 新图的 $root = n$, 在寻找结点最小入弧时如果弧的起点是 $root$, 那么 $real_root =$ 这条入弧的终点, 这只是必要条件。考虑充分条件: 如果原图的最小树形图的根只有一个解, 那么这样找肯定成立。如果有多个根, 那么这些根必然存在一个环上, 而且环上的每个点都可以作为原图的根节点, 又因为将环缩点时是将环上的所有点缩成环上最小编号点, 所以找到的必然是最小编号点。还需要注意因为 $real_root$ 是边的编号而且边的顶点信息在 ZLEdmonds 算法中是会发生变化的, 所以真正的根节点是 $real_root - m.m$ 是给出的边数, 因为初始建边时当 $i \geq m$ 时, $edge[i].m = i - m$, 而且求出的 $real_root$ 一定是 $\geq m$

```

1 typedef long long ll;
2 const int MAX_N = 1010;
3 const int MAX_M = 11000;
4 const ll INF = LONG_LONG_MAX;
5
6 int n, m, NV, NE, root, real_root;
7 int vis[MAX_N], pre[MAX_N], ID[MAX_N];
8 ll In[MAX_N];
9
10 struct Edge{
11     int u, v;
12     ll w;
13
14     Edge() {}
15     Edge(int _u, int _v, ll _w) : u(_u), v(_v), w(_w) {}
16 } edge[MAX_M];
17 }
```

```

18 11 ZLEdmonds(int root, int NV, int NE)
19 {
20     ll res = 0, w;
21     int u, v;
22     while(1){
23         for(int i = 0; i < NV; i++) {
24             In[i] = INF;
25         }
26         for(int i = 0; i < NE; i++){
27             u = edge[i].u, v = edge[i].v, w = edge[i].w;
28             if(u != v && w < In[v]){
29                 In[v] = w;
30                 pre[v] = u;
31                 if(u == root) {
32                     real_root = i;
33                 }
34             }
35         }
36         for(int i = 0; i < NV; i++){
37             if(i != root && In[i] == INF) return -1;
38         }
39         int cnt = 0;
40         memset(vis, -1, sizeof(vis));
41         memset(ID, -1, sizeof(ID));
42         In[root] = 0;
43         for(int i = 0; i < NV; i++){
44             res += In[i];
45             v = i;
46             while(v != root && vis[v] != i && ID[v] == -1){
47                 vis[v] = i;
48                 v = pre[v];
49             }
50             if(v != root && ID[v] == -1){
51                 for(u = pre[v]; u != v; u = pre[u]){
52                     ID[u] = cnt;
53                 }
54                 ID[v] = cnt++;
55             }
56             if(cnt == 0) break;
57         }
58         for(int i = 0; i < NV; i++){
59             if(ID[i] == -1) ID[i] = cnt++;
60         }
61         for(int i = 0; i < NE; i++){
62             u = edge[i].u, v = edge[i].v;
63             edge[i].u = ID[u], edge[i].v = ID[v];
64             if(edge[i].u != edge[i].v){
65                 edge[i].w -= In[v];
66             }
67         }
68         NV = cnt;
69         root = ID[root];
70     }
71     return res;
72 }
73
74 int main()
75 {
76     while(~scanf("%d%d", &n, &m)){
77         NE = 0;
78         ll sum = 0;
79         for(int i = 0; i < m; i++){
80             int u, v;
81             ll w;
82             scanf("%d%d%lld", &u, &v, &w);

```

```

83         edge[NE++] = Edge(u, v, w);
84         sum += w;
85     }
86     sum++;
87     for (int i = 0; i < n; i++){
88         edge[NE++] = Edge(n, i, sum);
89     }
90     NV = n + 1, root = n;
91     ll ans = ZLEdmonds(root, NV, NE);
92     if (ans == -1 || ans - sum >= sum) printf("impossible\n\n");
93     else printf("%lld %d\n\n", ans - sum, real_root - m);
94 }
95 return 0;
96 }
```

9.6.4 最小树形图路径

Codeforces 240E Road Repairs

有 n 个城市，编号 $1 \sim n$ ，首都编号为 1，有 m 条有向边 $u[i], v[i], w[i], w[i] = 0$ 表示这条边是完好的， $w[i] = 1$ 表示这条边需要修理，问从首都出发能到达任意城市最少需要修多少条边？如果不能到达任意城市输出 -1，否则输出需要修复的最少边数和边的编号。如果有多个组答案输出任意一组。

```

1 typedef long long ll;
2 const int MAX_N = 100010;
3 const int MAX_M = 2000010;
4 //尽可能大点，因为记录路径需要不断扩展和统计边的编号MAX_M
5 const int INF = 0x7fffffff;
6
7 int n, m, NV, NE;
8 int pre[MAX_N], vis[MAX_N], In[MAX_N], ID[MAX_N];
9 int usedEdge[MAX_M], preEdge[MAX_N];
10
11 struct Edge{
12     int u, v, w, ww, id;
13
14     Edge() {}
15     Edge(int _u, int _v, int _w, int _ww, int _id) :
16         u(_u), v(_v), w(_w), ww(_ww), id(_id) {}
17 }edge[MAX_M];
18
19 struct Used{
20     int pre, id;
21 }cancel[MAX_M];
22
23
24 int ZLEdmonds( int root )
25 {
26     memset(usedEdge, 0, sizeof(usedEdge));
27     int res = 0, u, v, w;
28     int total = NE;
29     while(1){
30         for(int i = 0; i < NV; i++) { In[i] = INF; }
31         for(int i = 0; i < NE; i++){
32             u = edge[i].u, v = edge[i].v, w = edge[i].w;
33             if(u != v && w < In[v]){
34                 In[v] = w;
35                 pre[v] = u;
36                 //记录这个顶点所在的边编号
37                 preEdge[v] = edge[i].id;
38             }
39         }
40         for(int i = 0; i < NV; i++){
41             if(i != root && In[i] == INF) return -1;
42         }
43     }
44 }
```

```

42 }
43 int cnt = 0;
44 memset(vis, -1, sizeof(vis));
45 memset(ID, -1, sizeof(ID));
46 In[root] = 0;
47 for(int i = 0; i < NV; i++){
48     res += In[i];
49     v = i;
50     if(i != root){ //非根节点
51         usedEdge[preEdge[i]]++; //这个顶点所在的边被使用
52     }
53     while(v != root && vis[v] != i && ID[v] == -1){
54         vis[v] = i;
55         v = pre[v];
56     }
57     if(v != root && ID[v] == -1){
58         for(u = pre[v]; u != v; u = pre[u]){
59             ID[u] = cnt;
60         }
61         ID[v] = cnt++;
62     }
63     if(cnt == 0) break;
64     for(int i = 0; i < NV; i++){
65         if(ID[i] == -1) ID[i] = cnt++;
66     }
67     for(int i = 0; i < NE; i++){
68         u = edge[i].u, v = edge[i].v;
69         edge[i].u = ID[u], edge[i].v = ID[v];
70         //将原先的边的 id 重新编号
71         if(edge[i].u != edge[i].v){
72             //edge[i].u 和 edge[i].v 是新图的点编号，两者不相等说明两者在原图中不属于同一有向环
73             edge[i].w -= In[v];
74             //如果在新图中用到该边那么上一次图中用到这条边就要被取消
75             cancle[total].id = edge[i].id;
76             cancle[total].pre = preEdge[v];
77             //重新编排新图的边的编号
78             edge[i].id = total++;
79         }
80     }
81     NV = cnt;
82     root = ID[root];
83 }
84 }
85 for(int i = total - 1; i >= NE; i--){ //从后往前扫新建边编号
86     if(usedEdge[i]){ //如果这条边被使用
87         usedEdge[cancle[i].id]++; //这条边的使用情况+1
88         usedEdge[cancle[i].pre]--; //上一次用到这条边被取消
89         //相当于在上一次图中的有向环中的这个指向 v 顶点的有向边被取消
90     }
91 }
92 return res;
93 }
94
95 int main()
96 {
97     while(~scanf("%d%d", &n, &m)){
98         for(int i = 0; i < m; i++){
99             int u, v, w;
100            scanf("%d%d%d", &u, &v, &w);
101            u--, v--;
102            edge[i] = Edge(u, v, w, w, i);
103        }
104        NV = n, NE = m;
105        int ans = ZLEdmonds(0);
106        if(ans == -1 || ans == 0) printf("%d\n", ans);
}

```

```

107     else {
108         printf("%d\n", ans);
109         for (int i = 0; i < m; i++) { // ww 是最初边权
110             if (edge[i].ww == 1 && usedEdge[i]) printf("%d ", i + 1);
111         }
112         printf("\n");
113     }
114 }
115 return 0;
116 }
```

9.6.5 其他

[UVA 11865 Stream My Contest]

有 n 个点编号为 $0 \sim n - 1$, 0 为根节点, m 条有向边, u, v, w, val 表示从 u 到 v 的花费是 w , 可以传输数据的带宽是 val , 每个点只可以从指向它的边中选择一个带宽 (带宽不可以叠加), 一张连通图的带宽是所有连通边中的最小带宽。问在总花费不超过 C 的条件下, 将 n 个点连通的最大带宽?

找出所有边中的最大带宽 $high$ 和最小带宽 low , 先跑一遍最小树形图判断能否在花费 C 内将图连通。如果可以连通那么二分枚举最大带宽为 $limit$, 只需要在判断点的最小入边时加上限制 $edge[i].val \geq limit$ 即可。`ZLEdmonds()` 函数的接口可设置为:

```

1 int ZLEdmonds( int root , int NV, int NE, int limit )
```

第一次跑时 $limit$ 为 low , 判断返回值 res 是否为 -1 和 $res \leq C$ 即可。

会不会二分出来的答案不是出现在原图中的某条边的带宽呢? 不会的。因为找最小入边的判断条件是和原图边的带宽比较的, 所以最终二分的答案还会落到边上的带宽。

[HDU 3072 Intelligence System]

n 个点编号从 $0 \sim n - 1$, 根节点编号为 0 , m 条有向边, 属于同一有向环中的边权为 0 , 求从根节点能到达其余所有点的最小花费?

属于同一强连通分量中边权为 0 . 先用 *Tanjan* 算法将同一有向环中的点重新编号, 然后将所有边的顶点重新编号, 并判断是否属于同一强连通分量, 最后再跑一遍最小树形图即可。

9.7 生成树计数

9.7.1 Matrix-Tree 定理

Matrix - Tree 定理 (*Kirchhoff* 矩阵 - 树定理)

- G 的度数矩阵 $D[G]$ 是一个 $n * n$ 的矩阵，并且满足：当 $i \neq j$ 时， $d[i][j] = 0$ ；当 $i = j$ 时， $d[i][j]$ 等于 v_i 的度数。
- G 的邻接矩阵 $A[G]$ 也是一个 $n * n$ 的矩阵，并且满足：如果 $v_i v_j$ 之间有边直接相连，则 $a[i][j] = 1$ ，否则为 0 ($a[i][i] = 0$)。

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。

时间复杂度： $O(n^3)$ 。有向图、无向图没区别的。

9.7.2 选择一些边连通使得任意两点之间恰好有一条路径，求不同的选择方案数

[SPOJ 104 Highways]

给 $n \leq 12$ 个点， m 条无向边，无重边和自环，选择一些边连通使得任意两点之间恰好有一条路径。求不同的选择方案数？

```

1 const int MAX_N = 20;
2 const double eps = 1e-6;
3
4 int degree[MAX_N];
5 double C[MAX_N][MAX_N];
6
7 int sgn(double x)
8 {
9     if(fabs(x) <= eps) return 0;
10    else if(x > 0) return 1;
11    else return -1;
12 }
13
14 double det(double mat[][MAX_N], int n)
15 {
16     double res = 1;
17     int cnt = 0;
18     for(int i = 0; i < n; ++i) {
19         if(sgn(mat[i][i]) == 0) { // 正对角线上元素为 0
20             for(int j = i + 1; j < n; ++j) {
21                 // 从后面的行中找到第 i 列不为 0 的行，并交换
22                 if(sgn(mat[j][i]) != 0) {
23                     for(int k = i; k < n; ++k) {
24                         swap(mat[i][k], mat[j][k]);
25                     }
26                     cnt++; // 交换行次数
27                     break;
28                 }
29             }
30         }
31         if(sgn(mat[i][i]) == 0) return 0;
32         for(int j = i + 1; j < n; ++j) {
33             mat[j][i] /= mat[i][i];
34             // 消去倍数，double 可以防止整数相除误差
35             // 必要时使用 long double + cin/cout 输入输出
36         }
37         for(int j = i + 1; j < n; ++j) {
38             for(int k = i + 1; k < n; ++k) {
39                 mat[j][k] -= mat[j][i] * mat[i][k];
40             }
41         }
42     }
43 }
```

```

42     res *= mat[i][i];
43 }
44 if (cnt & 1) res = -res;
45 return res;
46 }

47
48 int main()
49 {
50     int T, n, m;
51     scanf("%d", &T);
52     while (T--) {
53         memset(degree, 0, sizeof(degree));
54         memset(C, 0, sizeof(C));
55         scanf("%d%d", &n, &m);
56         for (int i = 0; i < m; ++i) {
57             int a, b;
58             scanf("%d%d", &a, &b);
59             a--; b--;
60             C[a][b] = C[b][a] = -1;
61             degree[a]++; degree[b]++;
62         }
63         for (int i = 0; i < n; ++i) {
64             C[i][i] = degree[i];
65         }
66         printf("%.0lf\n", det(C, n - 1));
67         // 计算 n-1 阶相当于去除了最后一行和最后一列,
68     }
69     return 0;
70 }
```

9.7.3 最小生成树计数

[HDU 4408 Minimum Spanning Tree]

给 $n \leq 100$ 个点和 $m \leq 1000$ 条边, 求生成最小生成树的方案数? 答案模 $p \leq 10^{10}$ 。

在用 *Kruskal* 算法求最小生成树时, 我们的做法是: 将图 $G = V, E$ 中的所有边按照权值由小到大进行排序, 等长的边可以按照任意顺序; 然后从小到大扫描每一条边, 将未连通的点连通, 权值累加, 最后得到的图 G' 就是图 G 的最小生成树。

将所有权值相同的边看成一个阶段整体处理, 这一阶段生成树个数和下一阶段是独立的。

我们将求最小生成树时所用的祖先存进 *father* 数组, 将连通块的祖先存进 *U* 数组 (相当于连通块缩成点)。用 *vis*[*i*] = 1 标记连通块的祖先。用 *link*[*i*][*j*] 表示两个原本独立的连通块 (*祖先分别为* *i* *j*) 的连通分量的度 (连通的边数)。

对于加入相同权值 *same* 后的新图可能会形成多个连通块, 这是需要对每个连通块计数。找到每个连通块的祖先, 将属于这个祖先的点计算。

确定祖先和该连通块的所有点

首先寻找的连通块应含有新加进来的边, 否则如果该连通块和未加边的连通块 (上一阶段的) 完全一样, 那就重复计数了。所以在加边时用 *vis* 数组记录 *father* 数组中的祖先是否被访问, 如果被访问那么将属于这个连通块的所有点看成一个点, 添加进新图的连通块, 将新图的连通块中的每一个点都存在祖先的数组 *vec* 中。

确定连通块内的点与点是否直接连通和每个点的度数

枚举每个新图的连通块的祖先, 在 *Matrix-Tree* 中, 图 G 的邻接矩阵 $A[G]$ 被定义为: 当 $v[i] v[j]$ 直接相连时, $A[i][j] = 1$, 否则 $A[i][j] = 0$ 。但是由于这里我们将旧图的连通块缩点了, 那么新图的点与点应看成是旧图的连通块与连通块, 所以新图的邻接矩阵应是点与点的边数, 而不是非 0 即 1 的关系 (而且 $link[i][j] = link[j][i]$)。

我们对新图的每个连通块求生成树的个数, 然后累乘, 将所有连通块求完后需要清空 *vec* 数组。同时还要把 *U* 数组和 *father* 数组更新, 因为这时所有的新连通块和旧图的连通块都要合并成一个新的连通块。最后还要检查图是否连通了 (所有点的公共祖先是否一样)。

```

1 const int MAX_N = 110;
```

```

2 const int MAX_M = 1010;
3
4 ll mod;
5 int vis[MAX_N], fa[MAX_N], U[MAX_N], link[MAX_N][MAX_N];
6 vector<int> vec[MAX_N];
7 ll C[MAX_N][MAX_N];
8
9 struct Edge{
10     int u, v, w;
11
12     Edge () {}
13     Edge (int _u, int _v, int _w): u(_u), v(_v), w(_w) {}
14     bool operator < (const Edge& rhs) const {
15         return w < rhs.w;
16     }
17 } edge[MAX_M];
18
19 void init(int n)
20 {
21     memset(link, 0, sizeof(link));
22     memset(vis, 0, sizeof(vis));
23     for(int i = 0; i < n; ++i) {
24         fa[i] = i;
25     }
26 }
27
28 inline int find(int x, int arr[])
29 {
30     return arr[x] == x ? x : arr[x] = find(arr[x], arr);
31 }
32
33 ll det (ll mat[][MAX_N], int n)
34 {
35     for(int i = 0; i < n; ++i) {
36         for(int j = 0; j < n; ++j) {
37             mat[i][j] = (mat[i][j] % mod + mod) % mod;
38         }
39     }
40     ll res = 1;
41     int cnt = 0;
42     for(int i = 0; i < n; ++i) {
43         for(int j = i + 1; j < n; ++j) {
44             while(mat[j][i]) { //这种方式避免了求逆元
45                 ll t = mat[i][i] / mat[j][i];
46                 for(int k = i; k < n; ++k) {
47                     mat[i][k] = (mat[i][k] - mat[j][k] * t) % mod;
48                     swap(mat[i][k], mat[j][k]);
49                 }
50                 cnt++;
51             }
52             if(mat[i][i] == 0) return 0;
53             res = res * mat[i][i] % mod;
54         }
55     }
56     if(cnt & 1) res = -res;
57     return (res + mod) % mod;
58 }
59
60 ll solve(int n, int m)
61 {
62     sort(edge, edge + m);
63     int same = -1;
64     ll ans = 1;
65     for(int i = 0; i <= m; ++i) {
66         if(edge[i].w != same || i == m) {

```

```

67     for( int j = 0; j < n; ++j) {
68         if( vis[j] ) { //旧图以为祖先的连通块被访问j
69             int fj = find(j, U); //找到连通块所在连通块
70             vec[fj].push_back(j);
71             fa[j] = fj;
72             vis[j] = 0;
73         }
74     }
75     for( int j = 0; j < n; ++j) {
76         int size = vec[j].size(); //扫描新图的每一个连通块
77         if( size <= 1) continue;
78         memset(C, 0, sizeof(C));
79         for( int k = 0; k < size; ++k) {
80             for( int h = k + 1; h < size; ++h) {
81                 int u = vec[j][k];
82                 int v = vec[j][h]; // u 和 v 是 j 这个连通块内的两个点
83                 C[k][h] -= link[u][v]; // link[u][v] 表示 u 和 v 的连通边数
84                 C[h][k] = C[k][h];
85                 C[k][k] += link[u][v];
86                 C[h][h] += link[v][u]; //对对角线元素添加连通度连通的边数()
87             }
88         }
89         ans = ans * det(C, size - 1) % mod;
90     }
91     for( int j = 0; j < n; ++j) {
92         U[j] = fa[j] = find(j, fa);
93         vec[j].clear();
94     }
95
96     if( i == m) break;
97     same = edge[i].w;
98 }
99     int u = edge[i].u, v = edge[i].v;
100    int fu = find(u, fa), fv = find(v, fa);
101    if( fu == fv) continue;
102    vis[fu] = vis[fv] = 1;
103    U[find(fv, U)] = find(fu, U);
104    link[fu][fv]++, link[fv][fu]++;
105 }
106    int flag = 1, com = find(0, fa);
107    for( int i = 1; i < n; ++i) { // 检验是否所有点都连通
108        if( com != find(i, fa)) {
109            flag = 0;
110            break;
111        }
112    }
113    if( flag == 0) ans = 0;
114    return (ans + mod) % mod;
115 }
116
117 int main()
118 {
119     int n, m;
120     while(~scanf("%d%d%lld", &n, &m, &mod) && (n || m || mod)) {
121         init(n);
122         for( int i = 0; i < m; ++i) {
123             int u, v, w;
124             scanf("%d%d%d", &u, &v, &w);
125             edge[i] = Edge(u - 1, v - 1, w);
126         }
127         printf("%lld\n", solve(n, m));
128     }
129     return 0;
130 }
```

9.8 最短路

运行限制及时间：

- Dijkstra: 不含负权。运行时间依赖于优先队列的实现，如 $O(n \log n + m)$
- Bellman-Ford: 无限制。运行时间 $O(n * m)$
- SPFA: 无限制。运行时间 $O(k * m)$ (k 远小于 n)
- Floyd-Warshall: 无限制。运行时间 $O(n^3)$

其中 1 ~ 3 均为单源最短路径 (Single Source Shortest Paths) 算法；4 为全源最短路径 (All Pairs Shortest Paths) 算法

9.8.1 Dijkstra + HeapNode

```

1 struct HeapNode{
2     int d, u;
3
4     HeapNode() {}
5     HeapNode(int _d, int _u): d(_d), u(_u) {}
6     bool operator < (const HeapNode& rhs) const {
7         return d > rhs.d;
8     }
9 };
10
11 void Dijkstra(int s)
12 { // 链式前向星建立双向边
13     priority_queue<HeapNode> que;
14     for (int i = 0; i <= n; ++i) { dis[i] = inf; }
15     dis[s] = 0;
16     memset(done, 0, sizeof(done));
17     que.push(HeapNode(0, s));
18     while (!que.empty()) {
19         HeapNode cur = que.top();
20         que.pop();
21         int u = cur.u;
22         if (done[u]) continue;
23         done[u] = 1;
24         for (int i = head[u]; i != -1; i = edge[i].next) {
25             int v = edge[i].v, w = edge[i].w;
26             if (dis[v] > dis[u] + w) {
27                 dis[v] = dis[u] + w;
28                 pre[v] = u; // 记录路径
29                 que.push(HeapNode(dis[v], v));
30             }
31         }
32     }
33 }
```

9.8.2 Bellman_Ford

判断负值圈存在的方法是：在 $n - 1$ 次松弛操作之后再执行一次循环，如果还有更新操作发生，则说明存在负值圈。

```

1 struct Edge {
2     int u, v, w;
3 } edge[MAX_M];
4
5 int Bellman_Ford(int st, int ed)
6 { // 边的编号从 0--total - 1
7     for (int i = 1; i <= n; ++i) { dis[i] = inf; }
8     dis[st] = 0;
```

```

9   for (int i = 1; i < n; ++i) { // n - 1 次松弛操作
10  for (int j = 0; j < total; ++j) {
11      int u = edge[j].u, v = edge[j].v, w = edge[j].w;
12      dis[v] = min(dis[v], dis[u] + w);
13  }
14 }
15 return dis[ed];
16 }
```

9.8.3 SPFA

```

1 bool SPFA(int st)
2 { // 也可以 stack 实现
3     queue<int> que;
4     for (int i = 1; i <= n; ++i) { dis[i] = inf; }
5     dis[st] = 0;
6     memset(inque, 0, sizeof(inque)); // 是否在队列内
7     memset(cnt, 0, sizeof(cnt)); // 进队次数
8     vis[st] = 1;
9     que.push(st);
10    while (!que.empty()) {
11        int u = q.front();
12        q.pop();
13        vis[u] = 0;
14        for (int i = head[u]; i != -1; i = edge[i].next) { // 链式前向星存边
15            int v = edge[i].v, w = edge[i].w;
16            if (dis[u] < inf && dis[v] > dis[u] + w) {
17                dis[v] = dis[u] + w;
18                if (!vis[v]) {
19                    vis[v] = 1;
20                    que.push(v);
21                    if (++cnt[v] > n) return false; // 存在负环
22                }
23            }
24        }
25    }
26    return true;
27 }
```

9.8.4 Floyd_Warshall

与前面四种不同, Floyd-Warshall 算法是所谓的”全源最短路径”, 也就是任意两点间的最短路径。它并不是对单源最短路径 $|v|$ 次迭代的一种渐进改进, 但是对非常稠密的图却可能更快, 因为它的循环更加紧凑。而且, 这个算法支持负的权值。

```

1 for (int k = 1; k <= n; ++k) {
2     for (int i = 1; i <= n; ++i) {
3         for (int j = 1; j <= n; ++j) {
4             dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
5         }
6     }
7 }
```

其中 dis 数组应初始化为邻接矩阵。需要提醒的是, $dist[i][i]$ 实际上表示”从顶点 i 绕一圈再回来的最短路径”, 因此图存在负环当且仅当 $dist[i][i] < 0$ 。

初始化时, $dist[i][i]$ 可以初始化为 0, 也可以初始化为无穷大。

9.9 二分图匹配

9.9.1 概念

点覆盖、最小点覆盖

点覆盖集即一个点集，使得所有边至少有一个端点在集合里。

极小点覆盖：本身为点覆盖，其真子集都不是。

最小点覆盖：假如选了一个点就相当于覆盖以它为端点的所有边，选择最少的点来覆盖所有的边。

点覆盖数：最小点覆盖的点数。

边覆盖、极小边覆盖

边覆盖集即一个边集，使得所有点都与集合里的边邻接。

极小边覆盖：本身是边覆盖，其真子集都不是。

最小边覆盖：边最少的边覆盖。

边覆盖数：最小边覆盖的边数。

独立集、极大独立集

独立集即一个点集，集合中任两个结点不相邻。

极大独立集：本身为独立集，再加入任何点都不是。

最大独立集：点最多的独立集。

独立数：最大独立集的点数。

团

团即一个点集，集合中任两个结点相邻。

极大团：本身为团，再加入任何点都不是。

最大团：点最多的团。

团数：最大团的点数。

边独立集、极大边独立集

边独立集即一个边集，满足边集中的任两边不邻接。（一个顶点最多只有一条边经过）

极大边独立集：本身为边独立集，再加入任何边都不是。

最大边独立集：边最多的边独立集。

边独立数：最大边独立集的边数。

边独立集又称匹配，相应的有极大匹配，最大匹配，匹配数)。

支配集、极小支配集

支配集即一个点集，使得所有其他点至少有一个相邻点在集合里。

极小支配集：本身为支配集，其真子集都不是。

最小支配集：点最少的支配集。

支配数：最小支配集的点数。

边支配集、极小边支配集

边支配集即一个边集，使得所有边至少有一条邻接边在集合里。

极小边支配集：本身是边支配集，其真子集都不是。

最小边支配集：边最少的边支配集。

边支配数：最小边支配集的边数。

最小路径覆盖

最小路径覆盖：是“路径”覆盖“点”，即用尽量少的不相交简单路径覆盖有向无环图 G 的所有顶点，即每个顶点严格属于一条路径。路径的长度可能为 0(单个点)。

最小路径覆盖数 = G 的点数 - 最小路径覆盖中的边数。应该使得最小路径覆盖中的边数尽量多，但是又不能让两条边在同一个顶点相交。

拆点

将每一个顶点 i 拆成两个顶点 X_i 和 Y_i 。然后根据原图中边的信息，从 X 部往 Y 部引边。所有边的方向都是由 X 部到 Y 部。因此，所转化出的二分图的最大匹配数则是原图 G 中最小路径覆盖上的边数。因此由最小路径覆盖数 = 原图 G 的顶点数 - 二分图的最大匹配数。

如果原图允许路径交叉，即同一顶点多条路径经过【例如 POJ 2594】，那么可以用 Floyd 算法将所有间接连通的点对变为直接连通，这样以来如果原图种的最小路径覆盖在一个顶点有多条路径，新图可以使得一些路径直接跳过这个顶点而直接到达终点，这样就可以使用匈牙利算法求出最大匹配数。

匹配

匹配是一个边集，满足边集中的边两两不邻接。匹配又称边独立集。

在匹配中的点称为匹配点或饱和点；反之，称为未匹配点或未饱和点。

交错轨是图的一条简单路径，满足任意相邻的两条边，一条在匹配内，一条不在匹配内。

增广轨：是一个始点与终点都为未匹配点的交错轨。

最大匹配是具有最多边的匹配。

匹配数是最大匹配的大小。

完美匹配是匹配了所有点的匹配。（判断完美匹配可以求出最大匹配数然后和顶点数比较）

完备匹配是匹配了二分图较小集合（二分图 X, Y 中小的那个）的所有点的匹配。

增广轨定理：一个匹配是最大匹配当且仅当没有增广轨。

所有匹配算法都是基于增广轨定理：一个匹配是最大匹配当且仅当没有增广轨。这个定理适用于任意图。

9.9.2 二分图的性质

二分图中，点覆盖数是匹配数。

1. 二分图的最小点覆盖数等于最大匹配数：最少的点使得每条边都至少和其中的一个点相关联。并且最小点覆盖的顶点一定在最大匹配的边的端点中产生。原因如下。只需要考虑边不是最大匹配边能否被最大匹配边的端点覆盖。如果这条边的两个端点都不是最大匹配边端点集中的点，那么这条边就是增广路，这和最大匹配性质相违背（求完最大匹配后不应该还有增广路），所以这条边一定有一个端点是最大匹配边端点集中的点，那么只需要选择这个点就能覆盖这条边了。
2. 二分图的独立数（最大独立集的点数）等于顶点数减去最大匹配数。把最大匹配两端的点都从顶点集中去掉这个时候剩余的点是独立集，这是 $|V| - 2 * |M|$ ，同时必然可以从每条匹配边的两端取一个点加入独立集并且保持其独立集性质，所以一共是 $|V| - |M|$ 。
3. 最小边覆盖 = 图中点的个数 - 最大匹配数 = 最大独立集

9.9.3 二分图的判定

二分图：有两顶点集且图中每条边的两个顶点分别位于两个顶点集中，每个顶点集中没有边直接相连接！

无向图 G 为二分图的充分必要条件是，G 至少有两个顶点，且其所有回路的长度均为偶数。

判断二分图的常见方法是染色法：

开始对任意一未染色的顶点染色，之后判断其相邻的顶点中，若未染色则将其染上和相邻顶点不同的颜色，若已经染色且颜色和相邻顶点的颜色相同则说明不是二分图，若颜色不同则继续判断，bfs 和 dfs 可以搞定！

易知：任何无回路的图均是二分图

用 BFS+ 链式前向星判断二分图的代码：

```

1 const int MAX_N=210;
2 const int MAX_M=40010;
3
4 int n,m,total;
5 int head[MAX_N],color[MAX_N];
6
7 struct Edge{
8     int to,next;
9
10    Edge() {}
11    Edge(int to,int next) : to(to),next(next) {
12    }
13 }edge[MAX_M];
14
15 inline void AddEdge(int from,int to)
16 {
17     edge[from].to=to;
18     edge[from].next=head[from];
19     head[from]=total++;
20 }
```

```

21 inline bool IsBipartiteGraph()
22 {
23     int st=1;
24     memset(color,-1,sizeof(color));
25     while(1){
26         int find=0;
27         for( int i=st ; i<=n ; i++){
28             if( color[ i ] == -1 ){
29                 st=i ;
30                 find=1;
31                 break;
32             }
33         }
34         if( find==0) break ;
35         queue<int> que;
36         color[ st ]=1;
37         // color[ i ]=0 和 color[ j ]=0 是不同的两组
38         que.push( st );
39         st++;
40         while( !que.empty() ){
41             int cur=que.front();
42             que.pop();
43             for( int i=head[ cur ] ; i!= -1 ; i=edge[ i ].next ){
44                 int v=edge[ i ].to;
45                 if( color[ v ] == -1 ){
46                     color[ v ]=1-color[ cur ];
47                     que.push( v );
48                 } else if( color[ v ] == color[ cur ] ){
49                     return false ;
50                 }
51             }
52         }
53     }
54     return true;
55 }
56
57 int main()
58 {
59     while(~scanf(" %d %d ",&n,&m)){
60         total=0;
61         memset(head,-1,sizeof(head));
62         for( int i=0 ; i<m ; i++ ){
63             int from,to;
64             scanf(" %d %d ",&from,&to);
65             AddEdge( from , to );
66         }
67         if( IsBipartiteGraph() == false ){
68             printf(" No\n ");
69         } else {
70             printf(" Yes\n ");
71         }
72     }
73     return 0;
74 }
```

9.9.4 匈牙利算法 (Hungary Algorithm)

根据一个匹配是最大匹配当且仅当没有增广路, 求最大匹配就是找增广轨, 直到找不到增广轨, 就找到了最大匹配。遍历每个点, 查找增广路, 若找到增广路, 则修改匹配集和匹配数, 否则, 终止算法, 返回最大匹配数。

时间复杂度是: $O(n * m)$

```
1 const int MAX_N=510;
2
3 int T,n,m,total;
4 int vis[MAX_N],match[MAX_N],head[MAX_N];
5
6 struct Edge{
7     int to,next;
8
9     Edge () {}
10    Edge(int to,int next) : to(to),next(next) {
11    }
12 }edge[MAX_N*MAX_N];
13
14 inline void AddEdge(int from,int to)
15 {
16     edge[total].to=to;
17     edge[total].next=head[from];
18     head[from]=total++;
19 }
20
21 inline bool dfs(int u)
22 {
23     for(int i=head[u]; i!=-1; i=edge[i].next){
24         int v=edge[i].to;
25         if(!vis[v]){
26             vis[v]=1;
27             if(match[v]==-1 || dfs(match[v])){
28                 match[v]=u;
29                 return true;
30             }
31         }
32     }
33     return false;
34 }
35
36 inline void solve()
37 {
38     memset(match,-1,sizeof(match));
39     int res=0;
40     for(int i=1;i<=n;i++){
41         memset(vis,0,sizeof(vis));
42         if(dfs(i)){
43             res++;
44         }
45     }
46     printf("%d\n",res); // res 即是最大匹配数
47 }
48
49 int main()
50 {
51     scanf("%d",&T);
52     while(T--){
53         total=0;
54         memset(head,-1,sizeof(head));
55         scanf("%d%d",&n,&m);
56         for(int i=1;i<=m;i++){
57             int from,to;
58             scanf("%d%d",&from,&to);
59             AddEdge(from,to);
60         }
61         solve();
62     }
63     return 0;
64 }
```

9.9.5 输出最小点覆盖的点

选择最少的点, 使得每条边至少有一个端点被选中. 最小覆盖数等于最大匹配数。

先求出最大匹配数, 并将每个 x 和 y 对应的匹配记录下来. 然后从 X 中的所有未匹配点出发扩展匈牙利树, 标记树中的所有点, 则 X 中的未标记点和 Y 中的已标记点组成了所求的最小覆盖.

在一个 $R * C (R, C \leq 1000)$ 的网格上放了 $n \leq 10^5$ 目标. 可以在网格外发射子弹, 子弹会沿着垂直或者水平方向飞行, 并打掉飞行路径上的所有目标. 计算最少需要多少子弹, 各从哪些位置发射, 才能把所有目标全部打掉.

建图: 将每一行看作一个 X 结点, 每一列看作一个 Y 结点, 每个目标对应一条边. 这样, 子弹打掉所有目标意味着每条边至少有一个结点被选中. 需要特别注意的是本题: 各行从上到下编号为 $1 \sim R$, 各列从左到右编号为 $1 \sim C$!

```

1 const int MAX_N = 1010;
2
3 int n, m, k, total;
4 int head[MAX_N], visx[MAX_N], visy[MAX_N];
5 int matchx[MAX_N], matchy[MAX_N];
6
7 struct Edge{
8     int to, next;
9 }edge[MAX_N*MAX_N];
10
11
12 inline void AddEdge(int from, int to)
13 {
14     edge[total].to = to;
15     edge[total].next = head[from];
16     head[from] = total++;
17 }
18
19 inline bool dfs(int u)
20 {
21     visx[u] = 1;
22     for(int i = head[u]; i != -1; i = edge[i].next){
23         int v = edge[i].to;
24         if(visy[v]) continue;
25         visy[v] = 1;
26         if(matchy[v] == -1 || dfs(matchy[v])){
27             matchx[u] = v;
28             matchy[v] = u;
29             return true;
30         }
31     }
32     return false;
33 }
34
35 inline int Hungary()
36 { // 匈牙利算法求最大匹配
37     int res = 0;
38     memset(matchy, -1, sizeof(matchy));
39     memset(matchx, -1, sizeof(matchx));
40     for(int i = 1; i <= n; i++){
41         memset(visx, 0, sizeof(visx));
42         memset(visy, 0, sizeof(visy));
43         if(dfs(i)) res++;
44     }
45     return res;
46 }
47
48 int main()
49 {
50     while(cin >> n >> m >> k && (n || m || k)){
51         total = 0;
52     }
53 }
```

```

52     memset(head, -1, sizeof(head));
53     for(int i = 0; i < k; i++){
54         int tpx, tpy;
55         cin >> tpx >> tpy;
56         AddEdge(tpx, tpy);
57     }
58     int ans = Hungary();
59     // 将所有的 X 顶点和 Y 顶点标记状态清0
60     memset(visx, 0, sizeof(visx));
61     memset(visy, 0, sizeof(visy));
62     for(int i = 1; i <= n; i++){
63         if(matchx[i] == -1){
64             //对所有不在最大匹配中的 X 顶点扩展匈牙利树标记树中顶点,
65             dfs(i);
66         }
67     }
68     cout << ans ;
69     for(int i = 1; i <= n; i++){ // X 顶点中所有没被标记的顶点
70         if(visx[i] == 0) cout << " r " << i ;
71     }
72     for(int i = 1; i <= m; i++){ // Y 顶点中所有被标记的顶点
73         if(visy[i] == 1) cout << " c " << i ;
74     }
75     cout << endl;
76 }
77 return 0;
78 }
```

9.9.6 霍普克洛夫特 - 卡普算法

Hopcroft-Carp 算法先使用 BFS 查找多条增广路，然后使用 DFS 遍历增广路（累加匹配数，修改匹配点集），循环执行，直到没有增广路为止。

BFS 遍历只对点进行分层（不标记是匹配点和未匹配点），然后用 DFS 遍历看上面的层次哪些是增广路径（最后一个点是未匹配的）。

BFS 过程可以看做是图像树结构一样逐层向下遍历，还要防止出现相交的增广路径。

每次使用调用 BFS 查找到多条增广路的路径长度都是相等的，而且都以第一次得到的 dis 为该次查找增广路径的最大长度。

时间复杂度: $O(n * \sqrt{m})$

```

1 int matchx[MAX_N], matchy[MAX_N], head[MAX_N];
2 // matchx[] 记录 x 点集的匹配
3 int disx[MAX_N], disy[MAX_N], vis[MAX_N];
4 // disx[] 记录 x 点集的点的层次
5
6 struct Edge{
7     int to, next;
8
9     Edge() {}
10    Edge(int _to, int _next) : to(_to), next(_next) {
11    }
12} edge[MAX_N*MAX_N];
13
14 inline void init()
15 {
16     total=0;
17     memset(matchx,-1,sizeof(matchx));
18     memset(matchy,-1,sizeof(matchy));
19     memset(head,-1,sizeof(head));
20 }
21
22 inline void AddEdge(int from, int to)
23 {
24     edge[total].to = to;
25     edge[total].next = head[from];
26 }
```

```

26     head[from] = total++;
27 }
28
29 inline bool bfs()
30 {
31     dis = INT_MAX;
32     memset(disx, -1, sizeof(disx));
33     memset(disy, -1, sizeof(disy));
34     queue<int> que;
35     //从 x 中找到所有未匹配点, 组成第 0 层
36     for (int i = 0; i < n; i++){
37         if (matchx[i] == -1){
38             que.push(i);
39             disx[i] = 0;
40         }
41     }
42     while (!que.empty()){
43         int u = que.front();
44         que.pop();
45         if (disx[u] > dis) break;
46         //新的 x 点集中的点增广路径长度大于 dis
47         for (int i = head[u]; i != -1; i = edge[i].next){
48             int v = edge[i].to;
49             if (disy[v] == -1){ // v 是未匹配点
50                 disy[v] = disx[u] + 1;
51
52                 if (matchy[v] == -1) {//得到本次 BFS 遍历的最大层
53                     dis = disy[v];
54                 } else {
55                     disx[matchy[v]] = disy[v] + 1;
56                     // v 是匹配点, 继续延伸
57                     que.push(matchy[v]);
58                 }
59             }
60         }
61     }
62     return dis != INT_MAX;
63 }
64
65 inline bool dfs(int u)
66 {
67     for (int i = head[u]; i != -1; i = edge[i].next){
68         int v = edge[i].to;
69         if (!vis[v]){
70             vis[v] = 1;
71             if (matchy[v] != -1 && disy[v] == dis) continue;
72             if (matchy[v] == -1 || dfs(matchy[v])){
73                 matchy[v] = u;
74                 matchx[u] = v;
75                 return true;
76             }
77         }
78     }
79     return false;
80 }
81
82 //返回最大匹配数
83 inline int Hopcroft_Carp()
84 {
85     int ans = 0;
86     while( bfs() ){
87         memset(vis, 0, sizeof(vis));
88         for (int i = 0; i < n; i++){
89             if (matchx[i] == -1 && dfs(i)) ans++;
90         }
91     }
92 }
```

```

91     }
92     return ans;
93 }
```

匈牙利算法和 Hopcroft-Carp 算法细节的对比:

匈牙利算法每次都以一个点查找增广路径, Hopcroft-Carp 算法是每次都查找多条增广路径;

匈牙利算法每次查找的增广路径的长度是随机的, Hopcroft-Carp 算法每趟查找的增广路径的长度只会在原来查找到增广路径的长度增加偶数倍(除了第一趟, 第一趟得到增广路径长度都是 1)。

9.9.7 二分图带权匹配

Kuhn-Munkers 算法用来解决最大权匹配问题: 在一个二分图内, 左顶点为 X , 右顶点为 Y , 现对于每组左右连接 $X[i], Y[j]$ 有权 $w[i][j]$, 求一种匹配使得所有 $w[i][j]$ 的和最大。

最大权匹配一定是完备匹配。如果两边的点数相等则是完美匹配。

如果点数不相等, 其实可以虚拟一些点, 使得点数相等, 也成为了完美匹配。

算法描述:

Kuhn-Munkers 算法是通过给每个顶点一个标号(叫做顶标)来把求最大权匹配的问题转化为求完备匹配的问题的。设顶点 X_i 的顶标为 $A[i]$, 顶点 Y_j 的顶标为 $B[j]$, 顶点 X_i 与 Y_j 之间的边权为 $w[i][j]$ 。在算法执行过程中的任一时刻, 对于任一条边 (i, j) , $A[i] + B[j] \geq w[i][j]$ 始终成立, 初始 $A[i]$ 为与 x_i 相连的边的最大边权, $B[j] = 0$ 。

Kuhn-Munkers 算法的正确性基于以下定理:

若由二分图中所有满足 $A[i] + B[j] = w[i][j]$ 的边 (i, j) 构成的子图(称做相等子图)有完备匹配, 那么这个完备匹配就是二分图的最大权匹配。因为对于二分图的任意一个匹配, 如果它包含于相等子图, 那么它的边权和等于所有顶点的顶标和; 如果它有的边不包含于相等子图, 那么它的边权和小于所有顶点的顶标和。所以相等子图的完备匹配一定是二分图的最大权匹配。

Kuhn-Munkers 算法思路:

初始时为了使 $A[i] + B[j] \geq w[i][j]$ 恒成立, 令 $A[i]$ 为所有与顶点 X_i 关联的边的最大权, $B[j] = 0$ 。如果当前的相等子图没有完备匹配, 就按下面的方法修改顶标以使扩大相等子图, 直到相等子图具有完备匹配为止。

我们求当前相等子图的完备匹配失败了, 是因为对于某个 X 顶点, 我们找不到一条从它出发的交错路。这时我们获得了一棵交错树, 它的叶子结点全部是 X 顶点。现在我们把交错树中 X 顶点的顶标全都减小某个值 d , Y 顶点的顶标全都增加同一个值 d , 那么我们会发现:

1. 两端都在交错树中的边 (i, j) , $A[i] + B[j]$ 的值没有变化。也就是说, 它原来属于相等子图, 现在仍属于相等子图。
2. 两端都不在交错树中的边 (i, j) , $A[i]$ 和 $B[j]$ 都没有变化。也就是说, 它原来属于(或不属于)相等子图, 现在仍属于(或不属于)相等子图。
3. X 端不在交错树中, Y 端在交错树中的边 (i, j) , 它的 $A[i] + B[j]$ 的值有所增大。它原来不属于相等子图, 现在仍不属于相等子图。
4. X 端在交错树中, Y 端不在交错树中的边 (i, j) , 它的 $A[i] + B[j]$ 的值有所减小。也就是说, 它原来不属于相等子图, 现在可能进入了相等子图, 因而使相等子图得到了扩大。

现在的问题就是求 d 值了。

为了使 $A[i] + B[j] \geq w[i][j]$ 始终成立, 且至少有一条边进入相等子图, d 应该等于: $\min(A[i] + B[j] - w[i][j])$, X_i 在交错树中, Y_i 不在交错树中。

Kuhn-Munkers 算法实现:

朴素的实现方法, 时间复杂度为 $O(n^4)$: 需要找 $O(n)$ 次增广路, 每次增广最多需要修改 $O(n)$ 次顶标, 每次修改顶标时由于要枚举边来求 d 值, 复杂度为 $O(n^2)$, 总的复杂度为 $O(n^4)$ 。

实际上 KM 算法的复杂度是可以做到 $O(n^3)$ 的。我们给每个 Y 顶点一个“松弛量”函数 slack, 每次开始找增广路时初始化为无穷大。在寻找增广路的过程中, 检查边 (i, j) 时, 如果它不在相等子图中, 则让 $slack[j]$ 变成原值与 $A[i] + B[j] - w[i][j]$ 的较小值。这样, 在修改顶标时, 取所有不在交错树中的 Y 顶点的 $slack$ 值中的最小值作为 d 值即可。但还要注意一点: 修改顶标后, 要把所有的不在交错树中的 Y 顶点的 $slack$ 值都减去 d 。

Kuhn – Munkras 算法流程:

1. 初始化可行顶标的值
2. 用匈牙利算法寻找完备匹配
3. 若未找到完备匹配则修改可行顶标的值
4. 重复 (2)(3) 直到找到相等子图的完备匹配为止

一些技巧:

- 最小权完备匹配: 将所有的边权值取其相反数, 求最大权完备匹配, 匹配的值再取相反数即可。
- KM 算法的运行要求是必须存在一个完备匹配, 求一个最大权匹配(不一定完备): 把不存在的边权值赋为 0。
- 边权之积最大: 每条边权取自然对数, 然后求最大和权匹配, 求得的结果 a 再算出 e^a 就是最大积匹配。需要注意精度问题。
- 当算法结束之后, 所有顶标之和最小. 即 $\sum (lx[i] + ly[i])(1 \leq i \leq n)$ 最小:
例如 [UVALive 11383]: 有一个 $n * n$ 的矩阵, 需要定义行值 $row[i]$ 和列值 $col[j]$ 使得对于矩阵中的任意元素 $data[i][j] \leq row[i] + col[j]$, 求最小的 $\sum (row[i] + col[i])$. 即所有行列值和最小. 那么就可以定义 $lx[i] = \max(w[i][j]), ly[i] = 0$ 分别代表 i 行的行值和 i 列的列值, 其中 $w[i][j]$ 是矩阵中的元素值. 显然这样取一定可以满足 $w[i][j] \leq lx[i] + ly[j]$, 但是这样不能保证所有行列值和最小, 需要用 KM() 算法对行列值进行松弛. 当跑完 KM() 算法后的顶点坐标值(即行列值) $lx[i]$ 和 $ly[i]$ 就是最优的了.

最小权匹配。以 POJ 2195 为例。

给出一个 $r * c$ 的矩阵, 字母 H 代表房屋, 字母 m 代表客人, 房屋的数量和客人的数量相同。每间房只能住一个人。求这些客人全部住进客房的最少移动步数?

```

1 const int MAX_N = 400;
2
3 //求最大小权匹配时图的级别一般为() 10^2 , 所以用邻接矩阵存图
4 int r, c, n, m;
5 char s[MAX_N][MAX_N];
6 int match[MAX_N], visx[MAX_N], visy[MAX_N];
7 int lx[MAX_N], ly[MAX_N], w[MAX_N][MAX_N], slack[MAX_N];
8
9 struct Pos{
10     int x,y;
11 }house[MAX_N * MAX_N], host[MAX_N * MAX_N];
12
13 inline bool dfs(int x)
14 {
15     visx[x] = 1;
16     for(int y = 0; y < m; y++){
17         if(visy[y]) continue;
18         int tmp = lx[x] + ly[y] - w[x][y];
19         if(tmp == 0){
20             visy[y] = 1;
21             if(match[y] == -1 || dfs(match[y])){
22                 match[y] = x;
23                 return true; // 找到增广轨
24             }
25         } else {
26             slack[y] = min(slack[y], tmp);
27         }
28     }
29     return false;
30     // 没有找到增广轨, 说明顶点 X 没有对应的匹配
31     // 与完备匹配(相等子图的完备匹配)不符
32 }
33
34 inline int KM()
35 {
36     memset(match, -1, sizeof(match));

```

```

37     memset(lx, 0, sizeof(lx));
38     for(int i = 0; i < n; i++){
39         lx[i] = INT_MIN;
40         for(int j = 0; j < m; j++){
41             lx[i] = max(lx[i], w[i][j]);
42         }
43     }
44     for(int i = 0; i < n; i++){
45         //初始边的松弛值为最大
46         for(int j = 0; j < m; j++){ slack[j] = INT_MAX; }
47         while(1){
48             memset(visx, 0, sizeof(visx));
49             memset(visy, 0, sizeof(visy));
50             if(dfs(i)) break; //找到增广轨，则该点增广完成，进入下一点增广
51             //没有找到增广轨需要改变顶标使图中可行边数量增加
52             int d = INT_MAX;
53             for(int j = 0; j < m; j++){
54                 if(!visy[j]) d = min(d, slack[j]);
55             }
56             //增广轨（增广过程中遍历到）中 X 方顶标全部减去常数d
57             for(int j = 0; j < n; j++) { if(visx[j]) lx[j] -= d; }
58             //增广轨中 Y 方顶标全部增加d
59             for(int j = 0; j < m; j++) {
60                 if(visy[j]) ly[j] += d;
61                 else slack[j] -= d; //不在增广轨中的顶点Y
62             }
63         }
64         int res = 0;
65         for(int j = 0; j < m; j++) {
66             if(match[j] != -1) res += w[match[j]][j];
67         }
68     }
69     return res;
70 }
71
72 int main()
73 {
74     while(~scanf("%d%d", &r, &c) && (r || c)){
75         n = m = 0;
76         for(int i = 0; i < r; i++) {
77             scanf("%s", s[i]);
78             for(int j = 0; j < c; j++){
79                 if(s[i][j] == 'H'){
80                     house[n].x = i;
81                     house[n++].y = j;
82                 } else if(s[i][j] == 'm') {
83                     host[m].x = i;
84                     host[m++].y = j;
85                 }
86             }
87         }
88         for(int i = 0; i < n; i++){
89             for(int j = 0; j < m; j++){
90                 int tmp = abs(house[i].x - host[j].x) + abs(house[i].y - host[j].y);
91                 w[i][j] = -tmp; // 求最小权匹配，将边权取反，然后求最大权匹配
92             }
93         }
94         int ans = KM();
95         printf("%d\n",-ans); // 结果取相反数
96     }
97     return 0;
98 }
```

9.9.8 二分图带权匹配拆点

有 m 个作坊和 n 件物品，给出每件商品在每个作坊加工完成的时间，求出加工完 n 件商品的最少平均时间。每件商品只能在一个作坊完成，每个作坊同一时间只能加工一件商品，每件商品的完成时间需要加上它的等待时间。

假设一个作坊最终加工 k 件商品，加工顺序依次是从 1 到 k ，则在该作坊加工的商品的总耗时是：

$$\text{cost}[1] + (\text{cost}[1] + \text{cost}[2]) + (\text{cost}[1] + \text{cost}[2] + \text{cost}[3]) + \dots + (\text{cost}[1] + \dots + \text{cost}[k])$$

则第 i 个商品对总耗时的贡献是: $\text{cost}[i] * (k - i + 1)$ ，那么倒数第 i 个商品对总耗时的贡献是: $\text{cost}[i] * i$ 。

需要将 m 个作坊拆成 $n * m$ 个作坊，用 $w[i][j * n + p]$ 表示第 i 个商品在第 j 个作坊倒数第 p 个加工完成的权值（对总耗时的贡献）。剩下的就是常规的将最小权匹配转换成最大权匹配做法。

```

1 for (int i = 0; i < n; i++){
2     for (int j = 0; j < m; j++){
3         int tmp;
4         cin >> tmp;
5         for (int k = 0; k < n; k++){
6             w[i][j * n + k] = -(k + 1) * tmp;
7         }
8     }
9 }
10 m = n * m;

```

9.9.9 稳定婚姻匹配

GaleShapley Algorithm: 男子将一轮一轮地去追求他中意的女子，女子可以选择接受或者拒绝他的追求者。第一轮，每个男子都选择自己名单上排在首位的女子，并向她表白。此时，一个女子可能面对的情况有三种：没有人跟她表白，只有一个人跟她表白，有不止一个人跟她表白。在第一种情况下，这个女子什么都不用做，只需要继续等待；在第二种情况下，接受那个人的表白，答应暂时和他匹配；在第三种情况下，从所有追求者中选择自己最中意的那一位，答应和他暂时匹配，并拒绝所有其他追求者。

第一轮结束后，有些男子已经匹配，有些男子仍然是单身。在第二轮追女行动中，每个单身男子都从所有还没拒绝过他的女子中选出自己最中意的那一个，并向她表白，不管她现在是否是单身。和第一轮一样，女子们需要从表白者中选择最中意的一位，拒绝其他追求者。注意，如果这个女子已经有匹配了，当她遇到了更好的追求者时，她必须放弃现有匹配，和更好的追求者形成新的匹配。这样，一些单身男子将会得到匹配，那些已经有了匹配的男子也可能重新变为单身。在以后的每一轮中，单身男子继续追求列表中的下一个女子，女子则从包括现男友在内的所有追求者中选择最好的一个，并对其他人说不。这样一轮一轮地进行下去，直到某个时候所有人都不再单身，下一轮将不会有新的表白发生，整个过程自动结束。此时的婚姻搭配就一定是稳定的了。

判断算法有穷性：

随着轮数的增加，总有一个时候所有人都能配对。由于在每一轮中，至少会有一个男子向某个女子告白，因此总的告白次数将随着轮数的增加而增加。倘若整个流程一直没有因所有人都配上对了而结束，最终必然会出现某个男子追遍了所有女子的情况。而一个女子只要被人追过一次，以后就不可能再单身了。既然所有女子都被这个男子追过，就说明所有女子现在都不是单身，也就是说此时所有人都已配对。

判断匹配稳定性

首先注意到，随着轮数的增加，一个男子追求的对象总是越来越糟，而一个女子的男友只可能变得越来越好。假设男 A 和女 1 各自有各自的对象，但比起现在的对象，男 A 更喜欢女 1。因此，男 A 之前肯定已经跟女 1 表白过。既然女 1 最后没有跟男 A 在一起，说明女 1 拒绝了男 A，也就是说她有了比男 A 更好的男孩儿。这就证明了，两个人虽然不是一对，但都觉得对方比自己现在的伴侣好，这样的情况绝不可能发生。

性质

这种男追女，女拒男的方案对男性更有利。事实上，稳定婚姻搭配往往不止一种，然而上述算法的结果可以保证，每一位男性得到的伴侣都是所有可能的稳定婚姻搭配方案中最理想的，同时每一位女性得到的伴侣都是所有可能的稳定婚姻搭配方案中最差的。

时间复杂度: $O(n^2)$

有 n 对男女, 先给出每个女生对 n 位男生的选择意向, 排在前面的优先选择, 然后给出 n 位男生的选择意向, 排在前面的优先选择. 输出每位女生的匹配, 使得每位女生都是稳定的最佳选择.

下面算法中存储女生的选择意向时有两种选择: 队列和数组. 把注释部分去掉//即是队列写法.

```

1 const int MAX_N = 1010;
2
3 int T, n;
4 int x[MAX_N][MAX_N], y[MAX_N][MAX_N]; //x means girls, y means boys
5 //x[i][j] = k means ith girl's kth preference is jth boy
6 int matchx[MAX_N], matchy[MAX_N];
7 //matchx[i] is used to store ith girl's final match
8 int order[MAX_N];
9 //order[i] means the ith girl's choose order; initialized as 1
10 queue<int> girl[MAX_N]; // girl[i] is used to store ith girl's preference order
11
12 void GaleShapley()
13 {
14     memset(matchy, -1, sizeof(matchy));
15     // all boy's states are initialized as -1 to imply not match
16     queue<int> single; //store all single girls' index
17     for(int i = 1; i <= n; i++) { single.push(i); }
18     while(!single.empty()){
19         int ttmpx = single.front(); //single girl
20         single.pop();
21         //int ttmpy = girl[ttmpx].front();
22         // ttmpx's current priority preference
23         //girl[ttmpx].pop();
24         int ttmpy = x[ttmpx][order[ttmpx]++];
25         int cur = matchy[ttmpy]; //The boy ttmpy's current match
26         if(cur == -1) {
27             matchx[ttmpx] = ttmpy;
28             matchy[ttmpy] = ttmpx;
29         } else if(y[ttmpy][ttmpx] < y[ttmpy][cur]){
30             //The girl ttmpx's preference is priority to the girl cur
31             matchx[ttmpx] = ttmpy;
32             matchy[ttmpy] = ttmpx;
33             single.push(cur);
34         } else { //The girl ttmpx doesn't find match
35             single.push(ttmpx);
36         }
37     }
38 }
39
40 int main()
41 {
42     cin >> T;
43     while(T-- > 0){
44         cin >> n;
45         for(int i = 1; i <= n; i++) { order[i] = 1; }
46         for(int i = 1; i <= n; i++){
47             //while(!girl[i].empty()) { girl[i].pop(); }
48             for(int j = 1; j <= n; j++){
49                 cin >> x[i][j];
50                 //int t;
51                 //cin >> t;
52                 //girl[i].push(t);
53             }
54         }
55         for(int i = 1; i <= n; i++){
56             for(int j = 1; j <= n; j++){
57                 int t;
58                 cin >> t;
59                 y[i][t] = j;
60             }
61         }
62     }
63 }
```

```

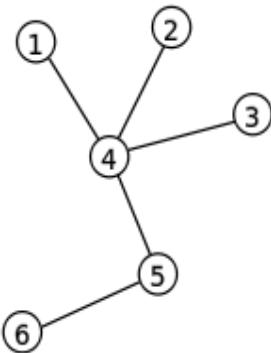
62     GaleShapley();
63     for( int i = 1; i <= n; i++){
64         //output each girl's matching result
65         cout << matchx[ i ] << endl;
66     }
67     if(T > 0) cout << endl;
68 }
69 return 0;
70 }
```

9.10 普吕弗序列

在图论中，标号树的普吕弗序列是由树唯一地产生的序列。 n 顶点的标号树有长 $n-2$ 的普吕弗序列。

9.10.1 由无根树生成普吕弗序列

每次（第 i 次）去掉树上编号最小的的叶子结点，并把普吕弗序列的第 i 项设为这个叶子的邻顶点的标号。一棵树的普吕弗序列是唯一的。



上面的树结构可得普吕弗序列：4,4,4,5。

9.10.2 从普吕弗序列还原无根树

设这普吕弗序列长 $n-2$ 。首先写出数 1 至 n 。第一步，找出 1 至 n 中没有在序列中出现的最小数。把标号为这数的顶点和标号为序列首项的顶点连起来，并把这数从 1 至 n 中删去，序列的首项也删去。接着每一步以 1 至 n 中剩下的数和余下序列重复以上步骤。最后当序列用完，把 1 至 n 中最后剩下的两数的顶点连起来。

9.10.3 结论

- 一个长为 $n-2$ 且每项都在 1 至 n 之间的序列 S ，有唯一的标号树以 S 为普吕弗序列。
- 长 $n-2$ 的序列共有 n^{n-2} 个，从而证明凯莱公式： n 顶点的标号无根树共有 n^{n-2} 棵。
- 一棵标号树实际上是标号完全图的一棵生成树。对普吕弗序列加以限制。类似的方法可以得到标号完全二分图的生成树总数。若 G 是完全二分图，一部分的顶点标号 1 到 n_1 ，另一部分的顶点标号 n_1+1 到 n 。 G 的标号生成树总数为， $n_1^{n_2-1} * n_2^{n_1-1}$ 其中 $n_2 = n - n_1$ 。
- 普吕弗序列中某个编号 i 出现的次数 $= d(i) - 1$, $d(i)$ 是这个编号的节点在无根树中的度数
- n 个节点的度依次为 d_1, d_2, \dots, d_n 的无根树共有 $\frac{(n-2)!}{[(d_1-1)*(d_2-1)*\dots*(d_{n-2}-1)]}$ 个，因为此时 Prüfer 编码中的数字 i 恰好出现 $d_i - 1$ 次。
- n 个节点的度依次为 d_1, d_2, \dots, d_n ，另有 m 个节点度数未知，求有多少种生成树？[BZOJ1005 明明的烦恼] 令每个已知度数的节点的度数为 $d_i (i : 1 \sim k)$ ，有 n 个节点， m 个节点未知度数， $sum = \sum_{i=1}^k (d(i) - 1)$ 。

$$Ans = C_{n-2}^{sum} * \frac{sum!}{\prod_{i=1}^k (d(i) - 1)!} * (n - k)^{n - sum - 2}$$

- n 个点的有标号有根树的计数： $n^{n-2} * n = n^{n-1}$

9.10.4 BZOJ 1211

给定一棵 $n \leq 150$ 个结点的树中所有点的度数，求有多少种可能的树？直接乘会爆 long long，所以先把每个数分解质因数，把质因数的次数相加相减，然后再乘起来。无解需要输出 0。当 $n! = 1 \& \& d[i] == 0$ 时或者当 $\sum(d[i] - 1)! = n - 2$ 时无解。

```

1 int n, sum = 0;
2 int num[200];
3
4 ll Qpow(int a, int b) {
5     ll ret = 1, t = a;
6     while (b) {
7         if (b & 1) ret *= t;
8         t *= t;
9         b >= 1;
10    }
11    return ret;
12}
13
14 void update(int x, int flag) {
15     for (int i = 2; i * i <= x; ++i) {
16         while (x % i == 0) num[i] += flag, x /= i;
17     }
18     if (x > 1) num[x] += flag;
19 }
20
21 int main() {
22     scanf("%d", &n);
23     for (int i = 2; i <= n - 2; ++i) update(i, 1);
24     for (int i = 1; i <= n; ++i) {
25         int d;
26         scanf("%d", &d);
27         if (d == 0 && n > 1) {
28             printf("0\n");
29             return 0;
30         }
31         sum += d;
32         for (int j = 2; j <= d - 1; ++j) update(j, -1);
33     }
34     if (sum != 2 * n - 2) printf("0\n");
35     else {
36         ll ans = 1;
37         for (int i = 2; i <= n; ++i) {
38             if (num[i]) ans *= Qpow(i, num[i]);
39         }
40         printf("%lld\n", ans);
41     }
42     return 0;
43 }
```

9.10.5 HDU 5629

给 $n \leq 50$ 个点，每个结点的度数不超过 $a_i \in [1, n]$ ，求构造树大小（结点个数）为 $s \in [1, n]$ 的方案数，答案对 $1e9 + 7$ 取模。

用 $dp[i][j][k]$ 表示处理到第 i 个节点，从中选择 j 个节点，且可重集大小（普吕弗序列长度）为 k 的排列组合数。初始化 $dp[0][0][0] = 1$ ，考虑用 $dp[i-1][j][k]$ 来更新 $dp[i][j][k]$ 的状态：不用第 i 个结点和枚举第 i 个结点的度数。滚动数组实现。

时间复杂度： $O(n^4)$ ，可以使用 FFT 优化到 $O(n^3 \log n)$ ，但是因为常数原因，差别不大。

```

1 const int MAX_N = 55;
2 const ll mod = (ll)(1e9) + 7;
3
```

```

4 int T, n;
5 int a[MAX_N];
6 ll dp[2][MAX_N][MAX_N], C[MAX_N][MAX_N];
7
8 int main() {
9     for (int i = 0; i < MAX_N; ++i) {
10         C[i][0] = C[i][i] = 1;
11         for (int j = 1; j < i; ++j) {
12             C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % mod;
13         }
14     }
15     scanf("%d", &T);
16     while (T--) {
17         scanf("%d", &n);
18         for (int i = 1; i <= n; ++i) scanf("%d", &a[i]);
19         memset(dp, 0, sizeof(dp));
20         dp[0][0][0] = 1;
21         int now = 0;
22         for (int i = 1; i <= n; ++i) {
23             now ^= 1;
24             memset(dp[now], 0, sizeof(dp[now]));
25             for (int j = 0; j <= i; ++j) {
26                 for (int k = 0; k <= n - 2; ++k) {
27                     dp[now][j][k] += dp[now ^ 1][j][k];
28                     if (dp[now][j][k] >= mod) dp[now][j][k] -= mod;
29                     for (int p = 1; p <= a[i] && k + p - 1 <= n - 2; ++p) {
30                         dp[now][j + 1][p + k - 1] += dp[now ^ 1][j][k] * C[k + p - 1][p - 1] % mod;
31                         if (dp[now][j + 1][p + k - 1] >= mod) dp[now][j + 1][p + k - 1] -= mod;
32                     }
33                 }
34             }
35             printf("%d", n);
36             for (int j = 2; j <= n; ++j) printf(" %lld", dp[now][j][j - 2]);
37             printf("\n");
38         }
39     }
40     return 0;
41 }
```

Chapter 10

字符串

10.1 Shift-And 算法

基于位并行的算法，如 Shift-And 算法的基本思想是：将模式串集合与文本串的匹配状态用位向量存储，匹配过程就是用位操作更新位向量的过程。它的优点是所需存储空间小，匹配速度快；缺点是算法性能会随模式串个数的增多而下降，只适合中小规模的模式串集合。

Shift-And 算法维护一个字符串的集合，集合中的每个字符串既是模式串 p 的前缀，同时也是已读入文本的后缀。每读入一个新的文本字符，该算法采用位并行的方法更新该集合，该集合用一个位掩码 $D = d_m \dots d_1$ 来表示。 D 的第 j 位被置为 1，当且仅当 $p_1 \dots p_j$ 是 $t_1 \dots t_i$ 的后缀。

Shift-And 算法首先构造一个 m 位 (m 是模式串的长度) 的向量表 $B[]$ ，用来记录字符在模式串的出现位置。如果 p_j 为字符 c ，掩码 $B[c]$ 的第 j 位被置为 1，否则为 0。首先置 $D = 0^m$ ，对于每个新读入的文本字符 t_{i+1} ，用如下公式对 D 进行更新： $D[i+1] = ((D[i] << 1) | 0^{m-1} 1) \& B[t_{i+1}]$ 。在匹配时，逐个扫描文本字符并更新向量 D ，当 $D[i] \& 10^{m-1} 0^m$ 时，在文本位置 i 处匹配成功。

Shift-And 算法扩展到多模式串时，将所有模式串的位向量 D 包装到一个机器字里，用位并行技术同时对 r 个位向量进行更新，初始化字和匹配掩码分别是所有初始化字和所有匹配掩码的连接。

设机器字的长度为 w ，文本串的长度为 n ，模式串的个数为 r ，最短模式串长度为 m ，那么 Shift-And 算法的时间复杂度为 $O(n \lceil \frac{m * r}{w} \rceil)$ 。由于采用了位并行技术，Shift-And 算法的匹配速度是很快的。但一旦模式串的长度和超出机器字的长度，算法的性能都会发生明显下降。

[2016 大连 B]

给一个 $n \leq 1000$ ，代表数字长度，以及每位上候选数字集合，再给一个数字字符串 $s(|s| \leq 5 * 10^6)$ ，输出 s 中所有匹配的 n 位数字子串。

样例输入：

4 (一共四位)

3 0 9 7 (第一位有三个候选数字分别为：0 9 7)

2 5 7 (第二位有两个候选数字分别为：5 7)

2 2 5 (第三位有两个候选数字分别为：2 5)

2 4 5 (第四位有两个候选数字分别为：4 5)

09755420524 (数字字符串 s)

样例输出：(所有匹配的四位数字子串)

9755

7554

0524

```
1 #include <iinttypes.h>
2
3 const int MAX_N = 1005;
4 const int MAX_LEN = 10000005;
5 const int MAX_ARR_LEN = ((MAX_N >> 6) + 5);
6 const int MASK = 63;
7
8 int n;
```

```

9 11 num[10][MAX_ARR_LEN], ret_n[MAX_ARR_LEN];
10 11 ind_x_arr[MAX_N], ind_y_arr[MAX_N];
11 char str[MAX_LEN];
12
13 void init() {
14     memset(num, 0, sizeof(num));
15     memset(ret_n, 0, sizeof(ret_n));
16     for (int i = 0; i < n; ++i) {
17         ind_x_arr[i] = (i >> 6) + 1; // i 位置属于哪一段
18         ind_y_arr[i] = 111 << (i & MASK); // 二进制中对应的位置
19     }
20 }
21
22 inline void set_one(ll *arr, int pos) {
23     arr[ind_x_arr[pos]] |= ind_y_arr[pos];
24 }
25
26 inline bool seek_one(ll *arr, int pos) {
27     return arr[ind_x_arr[pos]] & ind_y_arr[pos];
28 }
29
30 inline void left_move_one(ll *arr, int pos) {
31     for (int i = pos; i >= 1; --i) {
32         arr[i] <= 1;
33         arr[i] |= (!!(arr[i - 1] & 0x8000000000000000ll));
34     }
35     set_one(arr, 0);
36 }
37
38 inline void and_opt(ll *arr1, ll *arr2, int pos) {
39     for (int i = 1; i <= pos; ++i) {
40         arr1[i] &= arr2[i];
41     }
42 }
43
44 int main() {
45     scanf("%d", &n);
46     init();
47     for (int i = 0; i < n; ++i) {
48         int x, y;
49         scanf("%d", &x);
50         for (int j = 0; j < x; ++j) {
51             scanf("%d", &y);
52             set_one(num[y], i);
53         }
54     }
55     scanf("%s", str);
56     int len = strlen(str);
57     int L = n >> 6;
58     if (n & MASK) L++; // 分成若干段数
59     for (int i = 0; i < len; ++i) {
60         left_move_one(ret_n, L);
61         and_opt(ret_n, num[str[i] - '0'], L);
62         if (seek_one(ret_n, n - 1)) {
63             char ch = str[i + 1];
64             str[i + 1] = '\0';
65             printf("%os\n", str + i - n + 1);
66             str[i + 1] = ch;
67         }
68     }
69     return 0;
70 }
```

10.2 字典树

10.2.1 HDU 1857

给出一个 $n * m(n, m \in [20, 500])$ 的字母表，对每个询问串输出它在字母表中的位置。可以从某个位置一直向右，一直向右下查找或者一直向下，输出最左上的起始位置。查找不到输出 $(-1, -1)$ 。

先把所有的询问串创建字典树，然后扫描字母表，看字母表中的串是否有在字典树中出现。

```

1 const int MAX = 1000010;
2 const int NUM = 26;
3 const int dir[4][2] = {{0, 1}, {1, 1}, {1, 0}};
4
5 int n, m;
6 int child[MAX][NUM], bel[MAX];
7 char mat[510][510];
8 pair<int, int> ans[10010];
9
10 struct Trie {
11     int tot, root;
12
13     void init() {
14         memset(child[1], 0, sizeof(child[1]));
15         tot = root = 1; bel[1] = 0;
16     }
17     void insert(const char* str, const int id) {
18         int* cur = &root;
19         for (const char* p = str; *p; ++p) {
20             cur = &child[*cur][*p - 'A'];
21             if (*cur == 0) {
22                 *cur = ++tot;
23                 memset(child[tot], 0, sizeof(child[tot]));
24                 bel[tot] = 0;
25             }
26         }
27         bel[*cur] = id;
28     }
29     void query(const int x, const int y, const int id) {
30         int step = 0, ttmpx = x, ttmpy = y;
31         int* cur = &root;
32         while (1) {
33             if (step > 20 || ttmpx >= n || ttmpy >= m) return;
34             char ch = mat[ttmpx][ttmpy];
35             cur = &child[*cur][ch - 'A'];
36             if (*cur == 0) return;
37             int pos = bel[*cur];
38             if (pos != 0 && ans[pos].first == -1) ans[pos] = make_pair(x, y);
39             ttmpx += dir[id][0], ttmpy += dir[id][1];
40             step++;
41         }
42     }
43 } dic;
44
45 int main() {
46     scanf("%d%d", &n, &m);
47     for (int i = 0; i < n; ++i) scanf("%s", mat[i]);
48     char str[20];
49     getchar(); getchar();
50     int num = 0;
51     dic.init();
52     while (gets(str) && str[0] != '-') {
53         dic.insert(str, ++num);
54         ans[num] = make_pair(-1, -1);
55     }
56     for (int i = 0; i < n; ++i) {
57         for (int j = 0; j < m; ++j) {

```

```

58         for (int k = 0; k < 3; ++k) {
59             dic.query(i, j, k);
60         }
61     }
62     for (int i = 1; i <= num; ++i) printf("%d %d\n", ans[i].first, ans[i].second);
63     return 0;
64 }

```

10.2.2 HDU 5536

给 $n \leq 1000$ 个 $\leq 1e9$ 的正整数 a_i , 从中找到三个互不相同的 a_i, a_j, a_k 使得 $(a_i + a_j) \otimes a_k$ 最大。输出最大值。 \otimes 表示异或。

把所有 a_i 从高位开始插进字典树, 贪心查找。再支持一个删除操作就可以了, 因为要保证互不相同。

时间复杂度: $O(n^2 * 30)$, 再乘上一个微小的常数。

```

1 const int MAX = 1000010;
2 const int NUM = 2;
3 const int MAX_N = 1010;
4
5 int child[MAX][NUM], cnt[MAX];
6
7 struct Trie {
8     int root, tot;
9
10    void init() {
11        root = tot = 1;
12        child[1][0] = child[1][1] = 0;
13        cnt[1] = 1;
14    }
15    void insert(const int x) {
16        int* cur = &root;
17        for (int i = 30; i >= 0; --i) {
18            cur = &child[*cur][(x >> i) & 1];
19            if (*cur == 0) {
20                *cur = ++tot;
21                child[tot][0] = child[tot][1] = 0;
22                cnt[tot] = 0;
23            }
24            cnt[*cur]++;
25        }
26    }
27    void remove(const int x) {
28        int* cur = &root;
29        for (int i = 30; i >= 0; --i) {
30            cur = &child[*cur][(x >> i) & 1];
31            cnt[*cur]--;
32        }
33    }
34    int query(const int x) {
35        int ret = 0;
36        int* cur = &root;
37        for (int i = 30; i >= 0; --i) {
38            int now = (x >> i) & 1, store = *cur;
39            if (now == 0) {
40                cur = &child[*cur][1];
41                if (cnt[*cur]) ret += (1 << i);
42                else cur = &child[store][0];
43            } else {
44                cur = &child[*cur][0];
45                if (cnt[*cur]) ret += (1 << i);
46                else cur = &child[store][1];
47            }
48        }

```

```
49         return ret;
50     }
51 } dic;
52
53 int T, n;
54 int a[MAX_N];
55
56 int main() {
57     scanf("%d", &T);
58     while (T--) {
59         scanf("%d", &n);
60         dic.init();
61         for (int i = 1; i <= n; ++i) {
62             scanf("%d", &a[i]);
63             dic.insert(a[i]);
64         }
65         int ans = 0;
66         for (int i = 1; i <= n; ++i) {
67             dic.remove(a[i]);
68             for (int j = i + 1; j <= n; ++j) {
69                 dic.remove(a[j]);
70                 ans = max(ans, dic.query(a[i] + a[j]));
71                 dic.insert(a[j]);
72             }
73             dic.insert(a[i]);
74         }
75         printf("%d\n", ans);
76     }
77     return 0;
78 }
```