



# ACM Algorithm Template

吟梦的 ACM 算法模板

Yume Akiyama

<kirainmoe@gmail.com>

<https://kirainmoe.com/>

XMU ACM Team

February 24, 2021



# Contents

1	数据结构	2
1.1	树状数组	2
1.1.1	单点修改, 区间求和	2
1.1.2	区间修改, 区间求和	3
1.1.3	二维树状数组 (矩阵求和与修改)	3
1.1.4	树状数组维护 MEX	4
1.2	线段树	6
1.2.1	单点/区间修改, 区间查询	6
1.2.2	二维线段树 (线段树套线段树, 矩阵区间最值)	8
1.2.3	可持久化线段树 (主席树)	10
1.2.4	动态开点线段树	11
1.2.5	Segment Tree Beats! (区间取 min)	12
1.3	堆	14
1.3.1	普通二叉堆	14
1.3.2	左偏树 (可并堆)	14
1.4	并查集	16
1.4.1	路径压缩	16
1.4.2	按秩合并	16
1.4.3	可持久化数组 & 可持久化并查集	17
1.5	ST 表	18
1.6	单调队列	19
1.7	平衡二叉树	20
1.7.1	替罪羊树	20
1.7.2	Treap (指针版)	23
1.7.3	FHQTreap (无旋 Treap)	29
1.7.4	伸展树 Splay (指针版)	31
1.7.5	平衡树测试数据	37
1.8	Link-Cut Tree	38
1.9	点分治	41
1.10	树链剖分	43
1.11	树套树	45
1.11.1	线段树套平衡树 (Treap)	45

1.11.2 树状数组套动态开点权值线段树	46
1.12 虚树	49
1.13 K-D Tree	51
1.13.1 求第 k 近点对距离	51
1.13.2 动态维护二维空间信息	53
1.14 启发式合并 (DST on Tree)	55
<b>2 图论</b>	<b>57</b>
2.1 最短路	57
2.1.1 Dijkstra	57
2.1.2 SPFA 算法	57
2.2 负环判定	60
2.2.1 BFS 判负环	60
2.2.2 DFS 判负环	60
2.3 最小生成树	61
2.3.1 Kruskal 算法	61
2.3.2 堆优化的 Prim 算法	62
2.3.3 最小瓶颈路	63
2.3.4 最小直径生成树	63
2.3.5 最小曼哈顿距离生成树	64
2.4 Tarjan	66
2.4.1 求割边 (桥)	66
2.4.2 求割点	66
2.4.3 求无向图点双连通分量 & 缩点	67
2.4.4 求无向图的边双连通分量 & 缩点	68
2.4.5 求有向图强连通分量 & 缩点	70
2.5 拓扑排序	72
2.6 欧拉回路	72
2.7 哈密尔顿路径	73
2.8 最近公共祖先 (LCA)	75
2.8.1 倍增法求 LCA	75
2.8.2 Tarjan 求 LCA	75
2.9 K 短路	76
2.10 树 & 子树的重心	77
2.11 树的直径	79
2.12 Kirchoff 矩阵树定理	80
2.12.1 无向图中的矩阵树定理	80
2.12.2 有向图中的矩阵树定理	80
2.13 2-SAT 问题	82
2.13.1 写法一 (tarjan 缩点)	82
2.13.2 写法二 (暴力)	83

2.14	图的度序列判断 . . . . .	85
2.14.1	Erdos 定理 . . . . .	85
2.14.2	Havel 定理 . . . . .	85
<b>3</b>	<b>网络流/二分图/匹配</b>	<b>86</b>
3.1	二分图 . . . . .	86
3.1.1	二分图匹配-匈牙利算法 . . . . .	86
3.1.2	二分图判定-染色算法 . . . . .	87
3.2	最大流 . . . . .	88
3.2.1	Edmonds-Karp 增广路算法 . . . . .	88
3.2.2	Dinic 算法 (带当前弧优化) . . . . .	89
3.3	最小费用最大流 . . . . .	91
3.4	带花树算法 . . . . .	92
3.5	KM 算法 . . . . .	94
<b>4</b>	<b>字符串和回文算法</b>	<b>97</b>
4.1	字典树 (Trie) . . . . .	97
4.2	KMP 算法 . . . . .	98
4.3	扩展 KMP (Z 函数) . . . . .	99
4.4	Manacher 算法 . . . . .	100
4.5	AC 自动机 . . . . .	101
4.6	后缀数组 . . . . .	103
4.7	后缀自动机 . . . . .	107
4.8	广义后缀自动机 . . . . .	112
4.9	回文树/回文自动机 . . . . .	115
4.10	字符串哈希 . . . . .	116
4.11	字符串循环同构的最小表示法 . . . . .	116
4.12	Lyndon 分解 . . . . .	117
<b>5</b>	<b>数学专题</b>	<b>119</b>
5.1	素数、欧拉函数、莫比乌斯函数 . . . . .	119
5.1.1	线性筛 . . . . .	119
5.1.2	求单值的欧拉函数 . . . . .	120
5.2	杜教筛和积性函数 . . . . .	121
5.2.1	积性函数 . . . . .	121
5.2.2	狄利克雷卷积 . . . . .	121
5.2.3	杜教筛 . . . . .	122
5.3	Min25 筛 . . . . .	123
5.3.1	求区间素数之和 . . . . .	123
5.3.2	求区间素数个数 . . . . .	123
5.4	GCD 和 LCM . . . . .	124
5.4.1	欧几里得算法 . . . . .	124

5.4.2 拓展欧几里得算法 (exgcd) . . . . .	124
5.5 快速幂 / 快速乘 . . . . .	125
5.5.1 取模快速幂 . . . . .	125
5.5.2 矩阵乘法和快速幂 . . . . .	125
5.6 快速乘 . . . . .	126
5.6.1 $O(\log n)$ 的快速乘 . . . . .	126
5.6.2 $O(1)$ 的快速乘 . . . . .	126
5.7 乘法逆元 . . . . .	127
5.7.1 exgcd 求逆元 . . . . .	127
5.7.2 费马小定理求逆元 . . . . .	127
5.7.3 线性预处理逆元 . . . . .	128
5.8 高斯消元 . . . . .	128
5.8.1 高斯消元求解方程 . . . . .	128
5.8.2 高斯消元求矩阵行列式 . . . . .	129
5.9 离散对数 / BSGS 算法 . . . . .	129
5.10 欧拉降幂 . . . . .	130
5.11 自适应辛普森积分 . . . . .	131
5.12 二次剩余 . . . . .	132
5.13 中国剩余定理 CRT . . . . .	133
5.14 素数判定 & 大数质因子分解 . . . . .	134
5.15 欧拉五边形数 & 整数拆分 . . . . .	136
5.16 线性规划的单纯形算法 . . . . .	138
<b>6 多项式</b> . . . . .	<b>141</b>
6.1 拉格朗日插值法 . . . . .	141
6.2 快速傅里叶变换 (FFT) . . . . .	141
6.3 快速数论变换 (NTT) . . . . .	144
6.3.1 写法一 (int) . . . . .	144
6.3.2 写法二 (ll) . . . . .	145
6.3.3 分治 NTT . . . . .	147
6.3.4 任意模数 NTT . . . . .	147
6.4 多项式求逆 . . . . .	149
6.5 多项式求导 & 指对数运算 . . . . .	150
6.6 多项式除法、多点求值、快速插值 . . . . .	152
<b>7 组合数学</b> . . . . .	<b>159</b>
7.1 常见公式和经典问题 . . . . .	159
7.1.1 经典恒等式 . . . . .	159
7.1.2 容斥原理 . . . . .	159
7.2 询问排列数、组合数 . . . . .	160
7.3 卡特兰数 . . . . .	160

7.4	斯特林数 . . . . .	161
7.4.1	第一类斯特林数 . . . . .	161
7.4.2	第二类斯特林数 . . . . .	161
7.5	Lucas 定理和扩展 Lucas . . . . .	162
7.6	Polya 定理 . . . . .	163
8	<b>动态规划</b> . . . . .	165
8.1	背包问题 . . . . .	165
8.1.1	0-1 背包 (每种物品只有一个) . . . . .	165
8.1.2	完全背包 (每种物品无限多个) . . . . .	165
8.1.3	多重背包 (每种物品有有限多个) . . . . .	165
8.1.4	混合背包 (有的物品有限, 有的物品无限) . . . . .	166
8.1.5	二维费用背包问题 . . . . .	166
8.1.6	分组背包 . . . . .	166
8.2	最长公共子序列 (LCS) . . . . .	167
8.3	最长上升子序列 . . . . .	167
8.4	最小编辑距离 . . . . .	168
8.5	最大连续子段和 . . . . .	169
8.6	区间 DP . . . . .	169
8.7	状态压缩 DP . . . . .	169
8.8	数位 DP . . . . .	170
8.9	斜率优化 . . . . .	171
9	<b>计算几何</b> . . . . .	173
9.1	二维几何基本定义 . . . . .	173
9.2	基本运算 . . . . .	174
9.2.1	求向量的极角 . . . . .	174
9.2.2	点积、向量模、两向量夹角 . . . . .	174
9.2.3	叉积, 三角形有向面积 . . . . .	175
9.2.4	向量位置关系 . . . . .	175
9.2.5	向量旋转、求向量的单位法向量 . . . . .	175
9.3	点与线 . . . . .	176
9.3.1	直线的表示方法 . . . . .	176
9.3.2	两条直线的交点 . . . . .	176
9.3.3	点到直线的距离 . . . . .	177
9.3.4	点到线段的距离 . . . . .	177
9.3.5	点在直线上的投影 . . . . .	177
9.3.6	线段相交判定 . . . . .	177
9.4	多边形 . . . . .	178
9.4.1	多边形的面积 . . . . .	178
9.4.2	判断点在多边形内外 . . . . .	179

9.4.3	三角形的重心 . . . . .	179
9.4.4	求质点组的重心 . . . . .	179
9.5	圆 . . . . .	180
9.5.1	定义 . . . . .	180
9.5.2	直线 AB 与圆 C 的交点 . . . . .	180
9.5.3	两圆相交 . . . . .	181
9.5.4	过定点到圆的切线 . . . . .	181
9.5.5	两圆的公切线 . . . . .	182
9.5.6	两点/三点定圆 . . . . .	183
9.5.7	最小覆盖圆 . . . . .	184
9.5.8	最小覆盖球的模拟退火算法 . . . . .	184
9.6	凸包 (Convex Hull) . . . . .	185
9.6.1	Andrew 算法求凸包 . . . . .	185
9.6.2	Graham 法求凸包 . . . . .	186
9.7	旋转卡壳 . . . . .	186
9.8	半平面交 . . . . .	188
9.8.1	半平面的表示 . . . . .	188
9.8.2	半平面交 . . . . .	188
9.9	扫描线 . . . . .	190
9.10	平面最近点对 (分治算法) . . . . .	192
9.11	给定半径任意圆能覆盖的最多的点的数量 (Angle-Sweep) . . . . .	193
9.12	三维计算几何 . . . . .	194
9.12.1	基本定义 . . . . .	194
9.12.2	求两向量夹角 . . . . .	196
9.12.3	点、线、平面 . . . . .	196
9.12.4	叉积应用 . . . . .	196
9.12.5	向量旋转 . . . . .	197
9.12.6	点沿直线移动 . . . . .	197
9.12.7	三维空间点到线段的距离 . . . . .	197
9.12.8	计算四面体的体积 . . . . .	198
9.13	常用公式 . . . . .	198
9.13.1	三角形 . . . . .	198
9.13.2	四边形 . . . . .	198
9.13.3	正 n 边形 . . . . .	198
9.13.4	圆 . . . . .	199
9.13.5	棱柱 . . . . .	199
9.13.6	棱锥 . . . . .	199
9.13.7	圆锥 . . . . .	199
9.13.8	圆台 . . . . .	200

<b>10 博弈论</b>	<b>201</b>
10.1 Nim 游戏 . . . . .	201
10.1.1 定义 . . . . .	201
10.1.2 概念和性质 . . . . .	201
10.1.3 结论 . . . . .	201
10.1.4 SG 函数打表模板 . . . . .	202
10.2 常见博弈模型 . . . . .	202
<b>11 杂项</b>	<b>203</b>
11.1 分治 . . . . .	203
11.1.1 二分 . . . . .	203
11.1.2 三分法 . . . . .	203
11.2 CDQ 分治求三维偏序 . . . . .	204
11.3 逆序对 . . . . .	205
11.3.1 归并排序求逆序对 . . . . .	205
11.3.2 树状数组求逆序对 . . . . .	206
11.4 莫队算法 (带修改) . . . . .	206
11.5 表达式处理 . . . . .	208
11.6 C++ 大整数模板 . . . . .	210
11.7 快速读入 . . . . .	212
11.8 __int128 输出函数 . . . . .	213
11.9 开栈 . . . . .	213
11.10 随机数生成 . . . . .	213
<b>12 语言 &amp; 库参考</b>	<b>215</b>
12.1 ACM Java 速成 . . . . .	215
12.2 ACM Python 速成 . . . . .	216
12.3 字符串处理函数 . . . . .	217
12.3.1 <cstring> 头文件函数 . . . . .	217
12.3.2 STL string . . . . .	217
12.4 GNU pbds Reference . . . . .	219
12.4.1 头文件和命名空间定义 . . . . .	219
12.4.2 Hash Table . . . . .	219
12.4.3 Priority Queue . . . . .	219
12.4.4 Tree . . . . .	220
12.5 重载哈希函数 . . . . .	221
<b>附录</b>	<b>222</b>
常用积分表 & 级数表 . . . . .	222
现场赛自用 vim 配置 . . . . .	229
对拍脚本 . . . . .	229

# Chapter 1

## 数据结构

### 1.1 树状数组

$$C[i] = A[i - 2^k + 1] + A[i - 2^k + 2] + \dots + A[i]$$
$$\text{sum}[i] = C[i] + C[i - 2^{k1}] + C[(i - 2^{k1}) - 2^{k2}] + \dots$$

#### 1.1.1 单点修改，区间求和

Tips: lowbit(x) => (x & (-x)) 的含义：当 x 为 0 时，结果为 0。当 x 为奇数时，结果为 1。当 x 为偶数时，结果为 x 中 2 的最大次方的因子。

```
1 class BinaryIndexedTree {
2     public:
3         int n, c[MAXN];
4         int lowbit(int x) {
5             return x & (-x);
6         }
7         void add(int i, int value) {
8             while (i <= n)
9                 c[i] += value, i += lowbit(i);
10        }
11        int sum(int x) {
12            int res = 0;
13            while (x > 0)
14                res += c[x], x -= lowbit(x);
15            return res;
16        }
17        void scale(int c) {
18            for (int i = 1; i < n; i++)
19                sum[i] *= c;
20        }
21    };
```

### 1.1.2 区间修改，区间求和

引入差分序列，试树状数组支持区间修改。树状数组中维护的是差分序列，需要有一个数组维护原数据。

```
1 class IntervalBinaryIndexedTree {
2     public:
3         int n;
4         ll a[MAXN], data[MAXN];
5         int lowbit(int x) {
6             return x & (-x);
7         }
8         void add(int pos, LL addVal) {
9             while (pos <= n)
10                 a[pos] += addVal, pos += lowbit(pos);
11         }
12         ll sum(int pos) {
13             ll res = 0;
14             while (pos > 0)
15                 res += a[pos], pos -= lowbit(pos);
16             return res;
17         }
18
19         /* 向区间 [x,y] 加上 val, 在点 x 加上 val, y+1 加上 -val */
20         void update(int x, int y, ll val) {
21             add(x, val), add(y + 1, -val);
22         }
23         /* 单点查询时, 需要加上该点的原数据 */
24         ll query(int x) {
25             return data[x] + sum(x);
26         }
27 } T;
```

### 1.1.3 二维树状数组（矩阵求和与修改）

```
1 class BinaryIndexedTree2D {
2     public:
3         int a[MAXN][MAXN], a1[MAXN][MAXN], a2[MAXN][MAXN], a3[MAXN][MAXN];
4         int lowbit(int x) {
5             return x & (-x);
6         }
7         void add(int x, int y, int val) {
8             for (int i = x; i <= n; i += lowbit(i))
9                 for (int j = y; j <= m; j += lowbit(j)) {
10                     a[i][j] += val;
11                     a1[i][j] += val * (x - 1);
```

```

12         a2[i][j] += val * (y - 1);
13         a3[i][j] += val * (x - 1) * (y - 1);
14     }
15 }
16 int query(int x, int y) {
17     int t1 = 0, t2 = 0, t3 = 0, t4 = 0;
18     for (int i = x; i > 0; i -= lowbit(i))
19         for (int j = y; j > 0; j -= lowbit(j))
20             t1 += a[i][j], t2 += a1[i][j], t3 += a2[i][j], t4 += a3[i][j];
21     return t1 * x * y + t4 - t2 * y - t3 * x;
22 }
23 /* 在矩形 (x1, y1), (x2, y2) 上加上 val */
24 void update_matrix(int x1, int y1, int x2, int y2, int val) {
25     add(x2 + 1, y2 + 1, tmp), add(x1, y1, tmp);
26     add(x2 + 1, y1, -tmp), add(x1, y2 + 1, -tmp);
27 }
28 /* 查询矩形 (x1, y1) (x2, y2) 所有元素的和 */
29 int sum_matrix(int x1, int y1, int x2, int y2) {
30     return query(x2, y2) + query(x1 - 1, y1 - 1)
31         - query(x2, y1 - 1) - query(x1 - 1, y2);
32 }
33 };

```

#### 1.1.4 树状数组维护 MEX

```

1 class MEX {
2 public:
3     int n, a[maxn], b[maxn];
4     inline int lowbit(int x) {
5         return x & (-x);
6     }
7     void init() {
8         for (int i = 0; i <= n; i++)
9             a[i] = 0, b[i] = 0;
10    }
11    void update(int x, int val) {
12        if (x > n)
13            return;
14        b[x] += val;
15        if ((b[x] == 1 && val == 1) || (b[x] == 0 && val == -1))
16            while (x <= n)
17                a[x] += val, x += lowbit(x);
18    }
19    int query(int x) {
20        int ans = 0;
21        while (x > 0)

```

```
22         ans += a[x], x -= lowbit(x);
23     return ans;
24 }
25 int mex() {
26     int l = 1, r = n, mid, cnt, ans;
27     while (l <= r) {
28         mid = (l + r) >> 1;
29         cnt = tree[idx[x]].query(mid);
30         if (cnt == mid)
31             ans = mid, l = mid + 1;
32         else
33             r = mid - 1;
34     }
35     return ans;
36 }
37 };
```

## 1.2 线段树

### 1.2.1 单点/区间修改，区间查询

以维护区间和、区间积为例，根据情况修改 lazytag 和传递过程，还可以维护区间最值、区间异或等等：

```
1 class SegTree {
2     public:
3         #define lson (rt << 1)
4         #define rson (rt << 1) + 1
5         long long sum[MAXN << 2], lmul[MAXN << 2], lplus[MAXN << 2], mod;
6         void pushUp(int rt) {
7             sum[rt] = (sum[lson] + sum[rson]) % mod;
8         }
9         void pushDown(int rt, int l, int r) {
10            int mid = (l + r) >> 1;
11            sum[lson] = (sum[lson] * lmul[rt] % mod + lplus[rt] * (mid - l + 1)) % mod;
12            sum[rson] = (sum[rson] * lmul[rt] % mod + lplus[rt] * (r - mid)) % mod;
13            lmul[lson] = (lmul[lson] * lmul[rt]) % mod;
14            lmul[rson] = (lmul[rson] * lmul[rt]) % mod;
15            lplus[lson] = (lplus[lson] * lmul[rt] % mod + lplus[rt]) % mod;
16            lplus[rson] = (lplus[rson] * lmul[rt] % mod + lplus[rt]) % mod;
17            lplus[rt] = 0, lmul[rt] = 1;
18        }
19        void build(int rt, int l, int r) {
20            lmul[rt] = 1, lplus[rt] = 0;
21            if (l == r) {
22                read(sum[rt]);
23                sum[rt] %= mod;
24                return;
25            }
26            int mid = (l + r) >> 1;
27            build(lson, l, mid);
28            build(rson, mid + 1, r);
29            pushUp(rt);
30        }
31        void multiply(int rt, int l, int r, int ul, int ur, long long val) {
32            if (ur < l || ul > r)
33                return;
34            if (ul <= l && r <= ur) {
35                sum[rt] = (sum[rt] * val) % mod;
36                lmul[rt] = (lmul[rt] * val) % mod;
37                lplus[rt] = (lplus[rt] * val) % mod;
38                return;
39            }
40            pushDown(rt, l, r);
41            int mid = (l + r) >> 1;
42            if (ul <= mid)
```

```

43         multiply(lson, l, mid, ul, ur, val);
44     if (mid < ur)
45         multiply(rson, mid + 1, r, ul, ur, val);
46     pushUp(rt);
47 }
48 void add(int rt, int l, int r, int ul, int ur, long long val) {
49     if (ur < l || ul > r)
50         return;
51     if (ul <= l && r <= ur) {
52         sum[rt] = (sum[rt] + val * (r - l + 1)) % mod;
53         lplus[rt] = (lplus[rt] + val) % mod;
54         return;
55     }
56     pushDown(rt, l, r);
57     int mid = (l + r) >> 1;
58     if (ul <= mid)
59         add(lson, l, mid, ul, ur, val);
60     if (mid < ur)
61         add(rson, mid + 1, r, ul, ur, val);
62     pushUp(rt);
63 }
64 void singleAdd(int rt, int l, int r, int pos, long long val) {
65     if (l == r) {
66         sum[rt] += val;
67         return;
68     }
69     pushDown(rt);
70     int mid = (l + r) >> 1;
71     if (pos <= mid)
72         singleAdd(lson, l, mid, pos, val);
73     else
74         singleAdd(rson, mid + 1, r, pos, val);
75     pushUp(rt);
76 }
77 long long query(int rt, int l, int r, int ql, int qr) {
78     if (qr < l || ql > r)
79         return OLL;
80     if (ql <= l && r <= qr)
81         return sum[rt] % mod;
82     pushDown(rt, l, r);
83     int mid = (l + r) >> 1;
84     return (
85         query(lson, l, mid, ql, qr) + query(rson, mid + 1, r, ql, qr)
86     ) % mod;
87 }
88 } T;

```

## 1.2.2 二维线段树（线段树套线段树，矩阵区间最值）

维护矩阵区间最大值/最小值。

```
1 class SegTree2D {
2     public:
3         int maxm[MAXN << 1][MAXN << 1], minm[MAXN << 1][MAXN << 1];
4         void pushUp(int x, int y, int dir) {
5             if (dir == 1)           // x-axis direction
6                 maxm[x][y] = max(maxm[x << 1][y], maxm[x << 1 | 1][y]),
7                 minm[x][y] = min(minm[x << 1][y], minm[x << 1 | 1][y]);
8             if (dir == 2)           // y-axis direction
9                 maxm[x][y] = max(maxm[x][y << 1], maxm[x][y << 1 | 1]),
10                minm[x][y] = min(minm[x][y << 1], minm[x][y << 1 | 1]);
11         }
12         void build2D(int rtx, int rty, int l, int r, int u) {
13             if (l == r) {
14                 if (u > 0)
15                     maxm[rtx][rty] = minm[rtx][rty] = a[u][l];
16                 else
17                     pushUp(rtx, rty, 1);      // update 1D
18             } else {
19                 int mid = (l + r) >> 1;
20                 build2D(rtx, rty << 1, l, mid, u);
21                 build2D(rtx, rty << 1 | 1, mid + 1, r, u);
22                 pushUp(rtx, rty, 2);      // update 2D
23             }
24         }
25         void build(int rt, int u, int d) {          // u:up, d:down
26             if (u == d)
27                 build2D(rt, 1, 1, n, u);
28             else {
29                 int mid = (u + d) >> 1;
30                 build(rt << 1, u, mid), build(rt << 1 | 1, mid + 1, d);
31                 build2D(rt, 1, 1, n, -1);        // -1: more than one line
32             }
33         }
34         void update2D(int rtx, int rty, int l, int r, int ql, int qr, int u, int val)
35             {
36                 if (ql <= l && qr >= r) {
37                     if (u > 0)
38                         maxm[rtx][rty] = minm[rtx][rty] = val;
39                     else
40                         pushUp(rtx, rty, 1);
41                 } else {
42                     int mid = (l + r) >> 1;
43                     if (ql <= mid)
44                         update2D(rtx, rty << 1, l, mid, ql, qr, u, val);
45                     if (mid + 1 <= qr)
```

```

45         update2D(rtx, rty << 1 | 1, mid + 1, r, ql, qr, u, val);
46         pushUp(rtx, rty, 2);
47     }
48 }
49 void update(int rt, int u, int d, int qu, int qd, int ql, int qr, int val) {
50     if (u == d)
51         update2D(rt, 1, 1, n, ql, qr, u, val);
52     else {
53         int mid = (u + d) >> 1;
54         if (qu <= mid)
55             update(rt << 1, u, mid, qu, qd, ql, qr, val);
56         if (mid + 1 <= qd)
57             update(rt << 1 | 1, mid + 1, d, qu, qd, ql, qr, val);
58         update2D(rt, 1, 1, n, ql, qr, -1, val);
59     }
60 }
61 pair<int, int> query2D(int rtx, int rty, int l, int r, int ql, int qr, int u) {
62     if (ql <= l && r <= qr)
63         return make_pair(maxm[rtx][rty], minm[rtx][rty]);
64     else {
65         int mid = (l + r) >> 1;
66         pair<int, int> res = make_pair(-INF, INF), tmp;
67         if (ql <= mid) {
68             tmp = query2D(rtx, rty << 1, l, mid, ql, qr, u);
69             res.first = max(res.first, tmp.first), res.second = min(res.second,
70                           tmp.second);
71         }
72         if (mid + 1 <= qr) {
73             tmp = query2D(rtx, rty << 1 | 1, mid + 1, r, ql, qr, u);
74             res.first = max(res.first, tmp.first), res.second = min(res.second,
75                           tmp.second);
76         }
77         return res;
78     }
79 }
80 pair<int, int> query(int rt, int u, int d, int qu, int qd, int ql, int qr) {
81     if (qu <= u && d <= qd)
82         return query2D(rt, 1, 1, n, ql, qr, u == d ? u : -1);
83     else {
84         int mid = (u + d) >> 1;
85         pair<int, int> res = make_pair(-INF, INF), tmp;
86         if (qu <= mid) {
87             tmp = query(rt << 1, u, mid, qu, qd, ql, qr);
88             res.first = max(res.first, tmp.first), res.second = min(res.second,
89                           tmp.second);
90         }
91         if (mid + 1 <= qd) {
92             tmp = query(rt << 1 | 1, mid + 1, d, qu, qd, ql, qr);
93         }
94     }
95 }
```

```

90             res.first = max(res.first, tmp.first), res.second = min(res.second,
91                         ↵ tmp.second);
92         }
93     }
94 }
95 } T;

```

### 1.2.3 可持久化线段树（主席树）

可持久化线段树的主要思想：保存每次插入操作的历史版本，实际上是在动态开点线段树的基础上，通过复用某些未修改的节点，创建  $n$  棵线段树。每进行一次修改时，产生新的节点数 = 树的高度，是  $O(n \log n)$  级别的。

```

1  /**
2  * 求静态区间 k 大值
3  * 如果要求区间 [1, r] 区间 k 小值，那么只需要找到插入 r 时的版本，通过权值线段树可以求 k
4  ↵ 大值。
5  * 求 [l, r] 区间最小值呢？前缀和呗，用 [1, r] 的信息减去 [1, l - 1] 的信息即可。
6 */
7 class PersistSegTree {
8 private:
9     static const int MAXN = 2 * 1e5 + 50,
10                MAXM = 2 * 1e5 + 50,
11                MAX_NLOGM = MAXN << 5;
12 public:
13     struct Node {
14         int val;
15         Node *lc, *rc;
16     } t[MAX_NLOGM];
17     Node* root[MAXM];
18
19     int cnt;
20
21     PersistSegTree(): cnt(0) {}
22     void pushUp(Node *x) {
23         x->val = x->lc->val + x->rc->val;
24     }
25     Node* build(int l, int r, int a[]) {
26         Node *cur = &t[++cnt];
27         if (l == r) {
28             cur->val = a[l];
29             return cur;
30         }
31         int m = (l + r) >> 1;
32         cur->lc = build(l, m, a), cur->rc = build(m + 1, r, a), pushUp(cur);

```

```

32     return cur;
33 }
34 Node* update(Node *now, int l, int r, int pos, int val) {
35     t[++cnt] = *now;
36     Node *cur = &t[cnt];
37     if (l == r) {
38         cur->val += val;
39         return cur;
40     }
41     int m = (l + r) >> 1;
42     if (pos <= m)
43         cur->lc = update(now->lc, l, m, pos, val);
44     else
45         cur->rc = update(now->rc, m + 1, r, pos, val);
46     pushUp(cur);
47     return cur;
48 }
49 int query(Node *p, Node *q, int l, int r, int k) {
50     if (l == r)
51         return l;
52     int m = (l + r) >> 1, lcnt = p->lc->val - q->lc->val;
53     if (k <= lcnt)
54         return query(p->lc, q->lc, l, m, k);
55     else
56         return query(p->rc, q->rc, m + 1, r, k - lc);
57 }
58 } T;
59
60 // 区间查询: anspos = T.query(T.root[r], T.root[l-1], 1, t, k);

```

## 1.2.4 动态开点线段树

有的时候，线段树需要维护的区间很大很大，但是实际用到的节点很少；那么干脆就不要开这么多的节点，用到的时候再向内存要。

例如要建立一个权值线段树，但是在线操作不让离散化，值域又是  $\infty$  级别的。即使这个区间的范围很大，但是如果询问  $q$  比较少的话，我们只需要  $q \log \inf$  个节点，就可以办到。

由于是动态开点的，左右子节点的下标代数关系就不再成立，因此节点中需要记录当前节点的左子树和右子树根节点位置。

```

1 class DynamicSegTree {
2 public:
3     struct Node {
4         int val, lc, rc;
5     } a[maxn << 2];
6     int cnt = 0;
7

```

```

8 // 修改操作, 若 rt 不存在, 则首先动态开点; 否则直接更新
9 void update(int &rt, int l, int r, int pos, int val) {
10    if (!rt)
11        rt = ++cnt;
12    // ...
13 }
14 // 查询操作, 若 rt 不存在直接返回空值 (可能是 0 或者反向最值)
15 int query(int rt, int l, int r, int ql, int qr) {
16    if (!rt)
17        return 0;
18    // ...
19 }
20 };

```

### 1.2.5 Segment Tree Beats! (区间取 min)

```

1 class SegmentTreeBeats {
2 private:
3     static const int maxn = 1000050;
4     struct Node {
5         int maxv, secmaxv, maxcnt;
6         ll sum;
7     } t[maxn << 2];
8 public:
9     #define lson (rt << 1)
10    #define rson (rt << 1) | 1
11    void pushUp(int rt) {
12        t[rt].maxcnt = 0, t[rt].sum = t[lson].sum + t[rson].sum;
13        t[rt].maxv = max(t[lson].maxv, t[rson].maxv);
14        t[rt].secmaxv = max(t[lson].secmaxv, t[rson].secmaxv);
15        if (t[lson].maxv == t[rson].maxv)
16            t[rt].secmaxv = max(t[rt].secmaxv, min(t[lson].maxv, t[rson].maxv));
17        if (t[lson].maxv == t[rt].maxv)
18            t[rt].maxcnt += t[lson].maxcnt;
19        if (t[rson].maxv == t[rt].maxv)
20            t[rt].maxcnt += t[rson].maxcnt;
21    }
22    void build(int rt, int l, int r, int a[]) {
23        if (l == r) {
24            t[rt].sum = t[rt].maxv = a[l];
25            t[rt].secmaxv = -1, t[rt].maxcnt = 1;
26            return;
27        }
28        int mid = (l + r) >> 1;
29        build(lson, l, mid, a), build(rson, mid + 1, r, a);
30        pushUp(rt);

```

```

31 }
32 void pushMin(int rt, int v) {
33     if (v >= t[rt].maxv)
34         return;
35     t[rt].sum += 1ll * (v - t[rt].maxv) * t[rt].maxcnt;
36     t[rt].maxv = v;
37 }
38 void pushDown(int rt) {
39     pushMin(lson, t[rt].maxv), pushMin(rson, t[rt].maxv);
40 }
41 void setMin(int rt, int l, int r, int ul, int ur, int val) {
42     if (val >= t[rt].maxv)
43         return;
44     if (ul <= l && r <= ur && t[rt].secmaxv < val) {
45         pushMin(rt, val);
46         return;
47     }
48     int mid = (l + r) >> 1;
49     pushDown(rt);
50     if (ul <= mid)
51         setMin(lson, l, mid, ul, ur, val);
52     if (mid < ur)
53         setMin(rson, mid + 1, r, ul, ur, val);
54     pushUp(rt);
55 }
56 ll query(int rt, int l, int r, int ql, int qr, int op) {
57     if (ql <= l && r <= qr)
58         return op == 2 ? t[rt].sum : t[rt].maxv;
59     int mid = (l + r) >> 1;
60     pushDown(rt);
61     ll ans = 0;
62     if (op == 2) {
63         ans = 0;
64         if (ql <= mid)
65             ans += query(lson, l, mid, ql, qr, op);
66         if (mid < qr)
67             ans += query(rson, mid + 1, r, ql, qr, op);
68     } else {
69         ans = -1;
70         if (ql <= mid)
71             ans = max(ans, query(lson, l, mid, ql, qr, op));
72         if (mid < qr)
73             ans = max(ans, query(rson, mid + 1, r, ql, qr, op));
74     }
75     pushUp(rt);
76     return ans;
77 }
78 } T;

```

## 1.3 堆

### 1.3.1 普通二叉堆

```
1 class Heap {    // priority_queue
2 public:
3     long long val[MAXN];
4     int cnt, tmp, rt;
5     void insert(int x) {
6         val[++cnt] = x, tmp = cnt;
7         while (tmp != 0) {
8             int par = tmp / 2;
9             if (val[par] > x)
10                 std::swap(val[par], val[tmp]);
11             else
12                 break;
13             tmp = par;
14         }
15     }
16     int top() { return val[1]; }
17     void pop() {
18         if (cnt == 0)    // empty
19             return;
20         rt = 1, val[rt] = val[cnt];
21         while (2 * rt < cnt) {
22             int lc = rt << 1, rc = (rt << 1) + 1;
23             if (rc >= cnt || val[lc] < val[rc])
24                 if (val[rt] > val[lc])
25                     std::swap(val[rt], val[lc]), rt = lc;
26                 else
27                     break;
28             else
29                 if (val[rt] > val[rc])
30                     std::swap(val[rt], val[rc]), rt = rc;
31                 else
32                     break;
33         }
34         cnt--;
35     }
36     bool empty() { return cnt == 0; }
37 };
```

### 1.3.2 左偏树（可并堆）

左偏树与配对堆一样，是一种可并堆，具有堆的性质，并且可以快速合并。左偏树每个节点的左儿子的  $dist$  都大于等于右儿子的  $dist$ .

对于一棵二叉树，我们定义**外节点**为左儿子或右儿子为空的节点，定义一个外节点的  $dist$  为 1 一个不是外节点的节点  $dist$  为其到子树中**最近**的外节点的距离加一。空节点的  $dist$  为 0。一棵  $n$  个节点的二叉树，根的  $dist$  不超过  $\lceil \log(n + 1) \rceil$ 。  
可并堆也可以用 pb\_ds 实现。

```
1 namespace Leftist {
2     int fa[maxn], val[maxn], dis[maxn], ch[maxn][2];
3     int findrt(int x) {
4         while (fa[x])
5             x = fa[x];
6         return x;
7     }
8     int merge(int x, int y) {
9         if (!x || !y)
10            return x | y;
11         if (val[x] > val[y] || (val[x] == val[y] && x > y))
12             swap(x, y);
13         ch[x][1] = merge(ch[x][1], y), fa[ch[x][1]] = x;
14         if (dis[ch[x][0]] < dis[ch[x][1]])
15             swap(ch[x][0], ch[x][1]);
16         dis[x] = dis[ch[x][1]] + 1;
17         return x;
18     }
19     void pop(int x) {
20         val[x] = -1, fa[ch[x][0]] = fa[ch[x][1]] = 0;
21         merge(ch[x][0], ch[x][1]);
22     }
23     // 合并 (x, y) 所在堆: merge(findrt(x), findrt(y))
24     // 查询 x 所在堆的最小数: val[findrt(x)]
25     // 插入元素: 令 val[cnt] = x, fa[cnt] = 0, dist[cnt] = 0
26     // 删除堆顶: pop(x)
27 }
```

## 1.4 并查集

### 1.4.1 路径压缩

```
1 namespace UnionFindSet {
2     int p[MAXN], n;
3     void init() {
4         for (int i = 0; i <= n; i++)
5             p[i] = i;
6     }
7     int find(int x) {
8         return x == p[x] ? x : p[x] = find(p[x]);
9     }
10    void unions(int x, int y) {
11        int px = find(x), py = find(y);
12        if (px != py)
13            p[py] = px;
14    }
15 }
```

### 1.4.2 按秩合并

把子树小的集合合并到子树大的集合上，不会破坏路径结构。

```
1 namespace UnionFindSet {
2     int p[MAXN], rank[MAXN], n;           // rank 表示集合的秩 (大小)
3     void init() {
4         for (int i = 0; i <= n; i++)
5             p[i] = i, rank[i] = 1;
6     }
7     int find(int x) {
8         return x == p[x] ? x : find(p[x]);
9     }
10    void unions(int x, int y) {
11        int px = find(x), py = find(y);
12        if (px == py)
13            return;
14        else
15            if (rank[px] > rank[py])
16                p[py] = px;
17            else if (rank[px] == rank[py])
18                p[py] = px, rank[px]++;
19            else
20                p[px] = py;
21    }
22 }
```

### 1.4.3 可持久化数组 & 可持久化并查集

```
1 namespace PersistUnionSet {
2     const int maxn = 2e5 * 20;
3     // root 保存版本, dep 用于按秩合并
4     int n, tot, lc[maxn], rc[maxn], fa[maxn], dep[maxn], root[maxn];
5     // 初始化并查集为初始状态
6     void build(int &rt, int l, int r) {
7         rt = ++tot;
8         if (l == r) {
9             fa[rt] = l;
10            return;
11        }
12        int mid = (l + r) >> 1;
13        build(lc[rt], l, mid), build(rc[rt], mid + 1, r);
14    }
15    // 可持久化数组
16    void update(int rt, int l, int r, int pos) {
17        if (l == r) {
18            dep[rt]++;
19            return;
20        }
21        int mid = (l + r) >> 1;
22        if (pos <= mid)
23            update(lc[rt], l, mid, pos);
24        else
25            update(rc[rt], mid + 1, r, pos);
26    }
27    int query(int rt, int l, int r, int pos) {
28        if (l == r)
29            return rt;
30        int mid = (l + r) >> 1;
31        if (pos <= mid)
32            return query(lc[rt], l, mid, pos);
33        else
34            return query(rc[rt], mid + 1, r, pos);
35    }
36    // 合并函数
37    void merge(int last, int &rt, int l, int r, int pos, int p) {
38        rt = ++tot, lc[rt] = lc[last], rc[rt] = rc[last];
39        if (l == r) {
40            fa[rt] = p, dep[rt] = dep[last];
41            return;
42        }
43        int mid = (l + r) >> 1;
```

```

44     if (pos <= mid)
45         merge(lc[last], lc[rt], l, mid, pos, p);
46     else
47         merge(rc[last], rc[rt], mid + 1, r, pos, p);
48 }
49 // 在以 rt 为根的版本上查询
50 int find(int rt, int x) {
51     int now = query(rt, 1, n, x);
52     if (fa[now] == x)
53         return now;
54     return find(rt, fa[now]);
55 }
56 // 在第 i 个版本上合并集合
57 void merge(int i, int x, int y) {
58     int px = find(root[i], x), py = find(root[i], y);
59     if (fa[px] != fa[py]) {
60         if (dep[px] > dep[py])
61             swap(px, py);
62         merge(root[i-1], root[i], 1, n, fa[px], fa[py]);
63         if (dep[px] == dep[py])
64             update(root[i], 1, n, fa[py]);
65     }
66 }
67 }

```

## 1.5 ST 表

ST 表可用于查询区间最值，需要  $O(n \log n)$  时间预处理，查询可以在  $O(1)$  时间内完成。不支持修改：

```

1 namespace STTable {
2     int log2[MAXN], st[MAXN][MAXLOG];
3     void preprocess() {
4         for (int i = 0; i < n; i++) {
5             scanf("%d", &st[i][0]);
6         }
7         // 预处理 log2
8         log2[0] = -1;
9         for (int i = 1; i <= n; i++)
10            log2[i] = (i & (i - 1)) ? log2[i-1] : log2[i-1] + 1 ;
11         // 预处理
12         for (int i = 1; (1 << i) <= n; i++)
13             for (int j = 0; j + (1 << i) - 1 < n; j++)
14                 st[j][i] = max(st[j][i - 1], st[j + (1 << (i - 1))][i-1]);
15     }
16     int query(int x, int y) {

```

```

17     // 如果数组从 0 开始, 要注意 x, y 的起点
18     int k = log2(y - x + 1);
19     return max(st[x][k], st[y - (1 << k) + 1][k]);
20 }
21 };

```

## 1.6 单调队列

```

1 class MonotonousQueue {
2 public:
3     int q[MAXN], head, tail, cmpflag;
4     void init(int flag) {
5         memset(q, 0, sizeof q);
6         head = 0, tail = 0, cmpflag = flag;
7     }
8     bool empty() { return head == tail; }
9     int size() { return tail - head; }
10    int front() { return q[head]; }
11    void pop() { head++; }
12    void push(int x) {
13        if (head == tail) {
14            q[tail++] = x;
15            return;
16        }
17        if (cmpflag) { // 单调递增列
18            while (q[tail - 1] >= x)
19                tail--;
20            q[tail++] = x;
21        } else { // 单调递减列
22            while (q[tail - 1] <= x)
23                tail--;
24            q[tail++] = x;
25        }
26    }
27 };

```

## 1.7 平衡二叉树

### 1.7.1 替罪羊树

```
1 class ScapeGoat {
2     public:
3         struct Node {
4             /* p: 父节点编号; size: 以该节点为根的子树节点个数 (包括自身)
5                 num: 当前节点维护的数; ch[0]: 左儿子, ch[1]: 右儿子 */
6             int p, num, size, ch[2];
7         };
8
9         const double alpha;      /* 旋转因子 0.5 <= alpha <= 1 */
10        Node t[MAXN];          /* 节点池 */
11        int cnt, root,           /* cnt: 记录节点个数, root: 记录根节点编号 */
12        sum, idx[MAXN];         /* idx[] 和 sum 用于重构时储存链状数据在节点池中的下标和
13        → 节点数 */
14 #define lson(x) t[x].ch[0]
15 #define rson(x) t[x].ch[1]
16
17     /* 构造函数, 默认旋转因子为 0.75, 树根为 1, 需要构建两个 -inf 和 inf 的虚拟节点便
18     → 于后序操作 */
19     ScapeGoat(): alpha(0.75), cnt(2), root(1) {
20         t[1].num = -INF, t[1].size = 2, t[1].ch[1] = 2;
21         t[2].num = INF, t[2].size = 1, t[2].p = 1;
22     }
23     /**
24      * 检查以 x 为根的子树是否平衡
25      * 替罪羊树的平衡机制: 当 0.5 <= alpha(旋转因子) <= 1 时, 对于某个结点 x, 如果
26      → 该结点满足:
27          * 1. size(lch(x)) <= alpha * size(x) --- x 的左子树节点数小于等于 (旋转因子
28          → * x 的子节点数)
29          * 2. size(rch(x)) <= alpha * size(x) --- x 的右子树节点数小于等于 (旋转因子
30          → * x 的子节点数)
31          * 即: 这个节点两棵子树的 size 都不超过以该节点为根的子树的 size, 那么就称该节
32          → 点及其子树是平衡的。
33          * 如果子树不是平衡的, 那么直接重构成完全二叉树即可。
34          */
35         bool balance(int x) {
36             return (
37                 ((double) t[x].size * alpha) >= ((double) t[lson(x)].size)
38                 && ((double) t[x].size * alpha) >= ((double) t[rson(x)].size)
39             );
40         }
41
42         /* 将平衡树的元素变成链存在 idx[] 数组中 */
43         void recycle(int x) {
44             if (lson(x))
```

```

38         recycle(lson(x));
39         idx[++sum] = x;
40         if (rson(x))
41             recycle(rson(x));
42     }
43     /* 根据 idx[] 数组建立平衡树 */
44     int build(int l, int r) {
45         if (l > r)
46             return 0;
47         int mid = (l + r) >> 1, x = idx[mid];
48         // 递归建立左右子树并为左右儿子写入父节点信息，然后更新子树根节点的 size
49         lson(x) = build(l, mid - 1), rson(x) = build(mid + 1, r);
50         t[lson(x)].p = t[rson(x)].p = x;
51         t[x].size = t[lson(x)].size + t[rson(x)].size + 1;
52         return x;
53     }
54     /* 重建替罪羊树以保证平衡结构 */
55     void rebuild(int x) {
56         sum = 0;           // 重置 sum，重新写入信息到 idx[] 数组中
57         recycle(x);      // 拍扁成链
58         int fa = t[x].p,          // 保存 x 当前的父节点
59             son = t[fa].ch[1] == x, // 如果 son = 1，更新右儿子；否则更新左儿子
60             cur = build(1, sum);
61         t[fa].ch[son] = cur, t[cur].p = fa; // 更新父节点和当前节点数据
62         if (x == root)
63             root = cur;
64     }
65     /* 将数字 x 插入到当前树中 */
66     void insert(int x) {
67         int u = root, cur = ++cnt;
68         t[cur].size = 1, t[cur].num = x;           // 插入数 x 到新分配的节点中
69         while (1) {
70             t[u].size++;                         // 从根节点向下更新节点的数量
71             int son = (x >= t[u].num);
72             if (t[u].ch[son])
73                 u = t[u].ch[son];
74             else {
75                 t[u].ch[son] = cur, t[cur].p = u;
76                 break;
77             }
78         }
79         // 检查平衡性
80         int flag = 0;
81         for (int i = cur; i; i = t[i].p)
82             if (!balance(i))
83                 flag = i;
84         // 如果不平衡，则暴力重构
85         if (flag)

```

```

86             rebuild(flag);
87     }
88     /* 删除编号为 x 的数 */
89     void erase(int x) {
90         if (lson(x) && rson(x)) {
91             int cur = lson(x);
92             while (rson(cur))
93                 cur = rson(cur);
94             t[x].num = t[cur].num, x = cur;
95         }
96         int son = lson(x) ? lson(x) : rson(x),
97             k = rson(t[x].p) == x;
98         t[t[x].p].ch[k] = son, t[son].p = t[x].p;
99
100        for (int i = t[x].p; i; i = t[i].p)
101            t[i].size--; // 更新节点数目
102        if (x == root)
103            root = son; // 特殊处理删除的节点为根的情况
104    }
105    /* 查找数 x 在树节点池 t[] 数组中的下标 */
106    int getPos(int x) {
107        int u = root;
108        while (1) {
109            if (t[u].num == x)
110                return u;
111            else
112                u = t[u].num < x ? rson(u) : lson(u);
113        }
114    }
115    /* 询问 x 在树中的排名 */
116    int getRank(int x) {
117        int u = root, ans = 0;
118        while (u)
119            if (t[u].num < x) {
120                ans += t[lson(u)].size + 1;
121                u = rson(u);
122            } else
123                u = lson(u);
124        return ans;
125    }
126    /* 询问第 k 个数的值 */
127    int getKth(int k) {
128        int u = root;
129        k++;
130        while (1)
131            if (t[lson(u)].size == k - 1)
132                return t[u].num;
133            else if (t[lson(u)].size >= k)

```

```

134         u = lson(u);
135     else
136         k -= t[lson(u)].size + 1, u = rson(u);
137     return t[u].num;
138 }
139 /* 获取 x 的前驱结点 */
140 int getForward(int x) {
141     int u = root, ans = -INF;
142     while (u) {
143         if (t[u].num < x)
144             ans = max(ans, t[u].num), u = rson(u);
145         else
146             u = lson(u);
147     }
148     return ans;
149 }
150 /* 获取 x 的后继结点 */
151 int getBackword(int x) {
152     int u = root, ans = INF;
153     while (u) {
154         if (t[u].num > x)
155             ans = min(ans, t[u].num), u = lson(u);
156         else
157             u = rson(u);
158     }
159     return ans;
160 }
161 } T;

```

### 1.7.2 Treap (指针版)

```

1 class Treap {
2 public:
3     struct Node {
4         int val /* 当前结点维护的数值 */, size /* 子树结点和本身的数个数 */,
5             cnt /* 值为 val 的数个数 */, rd /* 优先级, 通常是随机值 */;
6         Node *ch[2]; /* ch[0] 左儿子, ch[1] 右儿子 */
7     } t[MAXN], nil; /* 结点池 & 空指针 */
8
9     int cnt; /* 结点池节点总数 */
10    Node *root, *NIL; /* 根节点 & 空指针 */
11
12    #define lson(p) p->ch[0]
13    #define rson(p) p->ch[1]
14
15    Treap(): cnt(0), root(NULL) {

```

```

16     srand(time(0));
17     nil = (Node) { 0, 0, 0, 0, 0, 0 }, NIL = &nil;
18     build();
19 }
20
21 /**
22 * 初始化操作，首先插入两个初始结点 (inf 和 -inf) 并构造其关系
23 * 令树根为无穷小结点 (1)，不要忘记 pushUp
24 */
25 void build() {
26     alloc(-INF), alloc(INF);
27     root = &t[1], rson(root) = &t[2];
28     pushUp(root);
29 }
30
31 /**
32 * 在结点池中为新结点分配空间
33 */
34 Node* alloc(int val) {
35     t[++cnt].val = val, t[cnt].cnt = t[cnt].size = 1, t[cnt].rd = rand();
36     t[cnt].ch[0] = t[cnt].ch[1] = NIL;
37     return &t[cnt];
38 }
39
40 /**
41 * 利用左右儿子的信息更新当前结点的 size, size[p] = size[lson] + size[rson] +
42 * cnt[p]
43 */
44 void pushUp(Node *p) {
45     p->size = lson(p)->size + rson(p)->size + p->cnt;
46 }
47
48 /**
49 * 右旋操作，注意 p 传入的是指针引用
50 *      P           L
51 *    /   \
52 *  L     R == zig(P) ==>      P
53 *    \           / \
54 *     Q           Q   R
55 */
56 void zig(Node *&p) {
57     Node *ls = lson(p);
58     lson(p) = rson(ls), rson(ls) = p;
59     p = ls;
60     pushUp(p), pushUp(rson(p));
61 }
62
63 /**

```

```

63 * 左旋操作同理
64 *      P          L
65 *      / \          \
66 *      L   R  <== zag(L) ==  P
67 *      \           / \
68 *      Q           Q   R
69 */
70 void zag(Node *&p) {
71     Node *rs = rson(p);
72     rson(p) = lson(rs), lson(rs) = p;
73     p = rs;
74     pushUp(p), pushUp(lson(p));
75 }
76 /**
77 * 插入一个新的结点, rt 为当前操作子树的指针, val 是待插入的数
78 * 1. 如果 rt 是空, 说明当前 rt 是叶结点, 可以直接分配空间
79 * 2. 如果 rt 指向结点的 val 等于待插入数, 说明有相同的数, 直接让 cnt 自增
80 * 3. 如果 val < 当前 val, 那么向左子树插入, 完成后判断优先级关系, 不满足则右旋
81 * 4. 如果 val > 当前 val, 那么向右子树插入, 完成后判断优先级关系, 不满足则左旋
82 * 5. 最后调用 pushUp 更新 size 和 cnt
83 * 无论何时, 根节点的 rd 值应该始终大于等于左右儿子的 rd 值
84 */
85 void insert(Node *&rt, int val) {
86     if (rt == NIL) {
87         rt = alloc(val);
88         return;
89     } else if (rt->val == val) {
90         rt->cnt++;
91     } else if (val < rt->val) {
92         insert(lson(rt), val);
93         if (rt->rd < lson(rt)->rd)
94             zig(rt);
95     } else {
96         insert(rson(rt), val);
97         if (rt->rd < rson(rt)->rd)
98             zag(rt);
99     }
100    pushUp(rt);
101 }
102 /**
103 * 删除结点, rt 为当前操作的子树的指针, val 是待删除的数值, 每次只删除一次
104 * 1. 如果 rt 为空, 说明代表该值的结点不存在
105 * 2. 如果 rt 指向的当前结点的值等于待删除的数值, 则分两种情况处理:
106 *     (a) 如果当前结点维护的数的数量不止一个 (rt->cnt > 1), 那么让其减一并 pushUp
107 *         更新
108 *     (b) 如果当前结点维护的数只有一个, 则判断其是否存在左右儿子:
109 */

```

```

110     * (i) 如果不存在左右儿子，说明这是一个叶结点，直接将其删除 (rt = NIL)
111     * (ii) 如果存在左右儿子，则判断
112         * (1) 如果右儿子为空或左儿子的优先级大于右儿子，右旋并从右子树查找删除
113         * (2) 反之则左旋并从左子树查找删除
114     * 3. 其他情况则根据 val 和 rt->val 的大小情况进入左右子树查找删除
115     * 4. 最后不要忘记 pushUp 更新结点信息
116 */
117 void erase(Node *&rt, int val) {
118     if (rt == NIL) {
119         return;
120     } else if (rt->val == val) {
121         if (rt->cnt > 1) {
122             rt->cnt--;
123         } else if (lson(rt) != NIL || rson(rt) != NIL) {
124             if (rson(rt) == NIL || lson(rt)->rd > rson(rt)->rd)
125                 zig(rt), erase(rson(rt), val);
126             else
127                 zag(rt), erase(lson(rt), val);
128         } else {
129             rt = NIL;
130             return;
131         }
132     } else if (val < rt->val)
133         erase(lson(rt), val);
134     else
135         erase(rson(rt), val);
136     pushUp(rt);
137 }
138 /**
139 * 查询某个数 val 在所给的以 rt 为根的子树中的排名，注意 Treap 满足二叉搜索树的性
140 质
→ * 如果 rt 为空，说明是叶子结点，那么返回 0
141 * 如果 val == rt->val，那么直接返回在左子树中 val 所在节点的左子树的 size + 1
142 * 如果 val 在左子树中，则递归进入左子树中继续查询；
143 * 如果在右子树中，排名即为 rt 左子树的 size + rt 的 cnt + 右子树中的查询结果
144 */
145 int rank(Node *rt, int val) {
146     if (rt == NIL)
147         return 0;
148     else if (rt->val == val)
149         return lson(rt)->size + 1;
150     else if (val < rt->val)
151         return rank(lson(rt), val);
152     else
153         return lson(rt)->size + rt->cnt + rank(rson(rt), val);
154 }
155

```

```

157 /**
158 * 查询在以 rt 为根的子树中排名为第 k 大的数
159 * 如果 rt 为空, 说明到了叶子结点都没有找到, 随便返回啥都行
160 * 如果 rt 的左儿子的 size 大于 k, 说明第 k 大的数在左子树中
161 * 否则如果 rt 左儿子的 size + rt->cnt >= k, 说明 rt 指向的数就是第 k 大的数
162 * 其他情况就在右子树中寻找 k - lson(rt)->size - rt->cnt 大的数
163 */
164 int kth(Node *rt, int rk) {
165     if (rt == NIL)
166         return INF;
167     else if (lson(rt)->size >= rk)
168         return kth(lson(rt), rk);
169     else if (lson(rt)->size + rt->cnt >= rk)
170         return rt->val;
171     else
172         return kth(rson(rt), rk - lson(rt)->size - rt->cnt);
173 }
174
175 /**
176 * 求 val 的前驱 (前驱定义为比 val 小的第一个数)
177 */
178 int prev(int val) {
179     int ans = -INF;
180     Node *rt = root;
181     while (rt != NIL) {
182         if (val == rt->val) {
183             if (lson(rt) != NIL) {
184                 rt = lson(rt);
185                 while (rson(rt) != NIL)
186                     rt = rson(rt);
187                 ans = rt->val;
188             }
189             break;
190         }
191         if (rt->val < val && rt->val > ans)
192             ans = rt->val;
193         rt = val < rt->val ? lson(rt) : rson(rt);
194     }
195     return ans;
196 }
197
198 /**
199 * 求 val 的后继 (后继定义为比 val 大的第一个数)
200 */
201 int next(int val) {
202     int ans = INF;
203     Node *rt = root;
204     while (rt != NIL) {

```

```
205     if (val == rt->val) {
206         if (rson(rt) != NIL) {
207             rt = rson(rt);
208             while (lson(rt) != NIL)
209                 rt = lson(rt);
210             ans = rt->val;
211         }
212         break;
213     }
214     if (rt->val > val && rt->val < ans)
215         ans = rt->val;
216     rt = val < rt->val ? lson(rt) : rson(rt);
217 }
218 return ans;
219 }
220 } T;
```

### 1.7.3 FHQTreap (无旋 Treap)

```
1 class FHQTreap {
2     public:
3         struct Node {
4             int val, size, cnt, rd;
5             Node *ch[2];
6         } t[MAXN], nil, *root, *NIL;
7         int cnt;
8
9 #define lson(x) x->ch[0]
10 #define rson(x) x->ch[1]
11
12     FHQTreap(): cnt(0) {
13         srand((int)time(NULL));
14         nil = (Node) { 0, 0, 0, 0, 0, 0 }, root = NIL = &nil;
15     }
16     void pushUp(Node *p) {
17         p->size = lson(p)->size + rson(p)->size + p->cnt;
18     }
19     void split(Node *now, int k, Node *&x, Node *&y) {
20         if (now == NIL)
21             x = y = NIL;
22         else if (now->val <= k)
23             x = now, split(rson(now), k, rson(now), y);
24         else
25             y = now, split(lson(now), k, x, lson(now));
26         pushUp(now);
27     }
28     void rankSplit(Node *now, int k, Node *&x, Node *&y) {
29         if (now == NIL)
30             x = y = NIL;
31         else {
32             if (k <= lson(now)->size)
33                 y = now, split(lson(now), k, x, lson(now));
34             else
35                 x = now, split(rson(now), k - lson(now)->size - 1, rson(now), y);
36             pushUp(now);
37         }
38     }
39     Node* merge(Node *a, Node *b) {
40         if (a == NIL || b == NIL)
41             return a == NIL ? b : a;
42         if (a->rd < b->rd) {
43             rson(a) = merge(rson(a), b), pushUp(a);
44             return a;
45         } else {
46             lson(b) = merge(a, lson(b)), pushUp(b);
47         }
48     }
49 }
```

```

47         return b;
48     }
49 }
50 Node* alloc(int val) {
51     t[++cnt] = (Node) { val, 1, 1, rand(), NIL, NIL };
52     return &t[cnt];
53 }
54 void insert(int val) {
55     Node *x, *y;
56     split(root, val, x, y), root = merge(merge(x, alloc(val)), y);
57 }
58 void erase(int val) {
59     Node *x, *y, *z;
60     split(root, val-1, x, y), split(x, val-1, x, y);
61     y = merge(lson(y), rson(y)), root = merge(merge(x, y), z);
62 }
63 int rank(int val) {
64     Node *x, *y;
65     split(root, val-1, x, y);
66     int res = x->size - 1;
67     root = merge(x, y);
68     return res;
69 }
70 Node *kth(Node *cur, int k) {
71     while (1)
72         if (k <= lson(cur)->size)
73             cur = lson(cur);
74         else
75             if (k <= lson(cur)->size + cur->cnt)
76                 return cur;
77             else
78                 k = k - (lson(cur)->size + cur->cnt), cur = rson(cur);
79 }
80 int prev(int t) {
81     Node *x, *y;
82     split(root, t-1, x, y);
83     int ans = kth(x, x->size)->val;
84     root = merge(x, y);
85     return ans;
86 }
87 int next(int t) {
88     Node *x, *y;
89     split(root, t, x, y);
90     int ans = kth(y, 1)->val;
91     root = merge(x, y);
92     return ans;
93 }
94 };

```

#### 1.7.4 伸展树 Splay (指针版)

```
1 class Splay {
2     private:
3         static const int MAXN = 200050,
4             INF = 0x3f3f3f3f;
5     public:
6         struct Node {
7             int val, /* 结点维护的数值 */
8                 size, /* 子树结点和本身的数个数 */
9                 cnt; /* 值为 val 的数个数 */
10            Node *ch[2], *p; /* 左右儿子和父亲指针 */
11        } t[MAXN] /* 结点池 */, nil /* 逻辑空结点 */;
12        Node *root /* 根节点指针 */, *NIL /* 逻辑空结点指针 */;
13
14        int cnt; /* 结点池中结点总数 */
15
16 #define lson(x) x->ch[0]
17 #define rson(x) x->ch[1]
18 #define son(x, d) x->ch[d]
19 #define p(x) x->p
20
21     /* 构造函数 */
22     Splay(): cnt(0) {
23         nil = (Node) { 0, 0, 0, 0, 0, 0 }, root = NIL = &nil;
24     }
25     /**
26     * 利用左右儿子信息更新当前结点的 size
27     */
28     void pushUp(Node *p) {
29         p->size = lson(p)->size + rson(p)->size + p->cnt;
30     }
31     /**
32     * 根据所给数组, 直接建立平衡树
33     */
34     Node* build(int l, int r, Node *p, int a[], int d, int offset) {
35         if (l > r)
36             return NIL;
37         int m = (l + r) >> 1;
38         Node *mid = alloc(m + offset, a[m], p);
39         if (p != NIL)
40             son(p, d) = mid;
41         else if (!offset)
42             root = mid;
43         build(l, m - 1, mid, a, 0, offset);
44         build(m + 1, r, mid, a, 1, offset);
45         pushUp(mid);
46         return mid;
47     }
```

```

47 }
48 /**
49 * 单旋操作函数，x 为待操作的结点，dir 是方向 (0 左旋,1 右旋)
50 * splay 每个结点都记录了它的父亲，因此可以直接传入操作结点，并得到它的父亲
51 * 也因此，在操作修改父子关系的时候，不要忘记修改子结点的父亲
52 * 下面的注释以 zig(右旋) 为例，zag(左旋) 是同理的
53 *      P           L
54 *    /   \
55 *  L     R == rotate(L, 1) ==>      P
56 *    \           /
57 *     Q         Q   R
58 */
59 void rotate(Node *x, bool dir) {
60     /* step1: 获得操作结点的父结点 */
61     Node *y = p(x);                      // P
62     // pushDown(x), pushDown(y);
63     /* step2: 将当前结点的右儿子赋给父结点的左儿子 */
64     son(y, !dir) = son(x, dir);        // lson(P) <= rson(L): Q
65     if (son(x, dir) != NIL)
66         son(x, dir)->p = y;          // parent(Q) <= P
67     /* step3: 将操作结点接到其父结点的父结点 */
68     p(x) = p(y);                      // parent(L) <= parent(P)
69     /* step4: 判断父结点的父结点是否为空 (如果为空说明当前父结点是根)
70         否则就断开祖先结点与父结点的联系，建立祖先结点与操作结点的联系。
71         更新父结点的父结点的左/右子树信息；两个节点随意旋转不会改变相对祖
72     先结点的方向 */
73     if (p(y) != NIL) {
74         if (y == son(p(y), dir))      // 这一步在图中就没法体现了
75             son(p(y), dir) = x;
76         else
77             son(p(y), !dir) = x;
78     }
79     /* step5: 最后将操作结点旋转方向上的儿子设为原先的父结点 */
80     son(x, dir) = y, p(y) = x;        // rson(L) <= P, parent(P) <= L
81     /* step6: 不要忘记更新结点的信息 */
82     pushUp(y), pushUp(x);           // 先 y 后 x, 顺序不能变
83 }
84 /**
85 * 伸展 (splay) 操作，将结点 x 伸展到 y 的子节点处
86 * 如果要将 x 旋转到根，那么 y 可以传入 Splay::NIL
87 * 分三种情况：
88 * (1) 如果当前结点的父结点即为 (子树的) 根节点，单旋即可
89 * (2) 设 x, y=p(x), z=p(y)，如果 x, y 都是对应父亲相同方向上的子结点，则同时对两
90 * 个结点右旋或左旋
91 * (3) 如果 x, y 是对应父亲不同方向上的子结点，那么对两者进行反方向的旋转，可以直接
92 * 用 while 完成
93 */
94 void splay(Node *x, Node *y) {

```

```

92     Node *tmp;
93     while (p(x) != y) {           // 循环进行旋转操作
94         // pushDown(x);
95         // if (y != NIL)      pushDown(y);
96         tmp = x->p;
97         if (x == lson(tmp)) {    // 如果 x 是左儿子
98             // 当前 tmp 的父结点不是 y, 并且 tmp 也是祖先的左儿子
99             // 都在左侧一条链上的情况, 进行 zig-zig
100            if (p(tmp) != y && tmp == lson(tmp->p))
101                rotate(tmp, 1);
102                rotate(x, 1);
103            } else {                  // x 是右儿子
104                // 都在右侧一条链上的情况, 进行 zag-zag
105                if (p(tmp) != y && tmp == rson(tmp->p))
106                    rotate(tmp, 0);
107                    rotate(x, 0);
108            }
109        }
110        if (y == NIL) {
111            root = x;    // 如果 y 是 NIL, 说明旋转后 x 是根
112        }
113    }
114    Node* alloc(int val, Node *p) {
115        t[++cnt] = (Node) {
116            val, 1, 1, NIL, NIL, p      // val, size, cnt, lson, rson, p
117        };
118        return &t[cnt];
119    }
120 /**
121 * 插入一个新数到 splay 中
122 */
123 void insert(int val) {
124     if (root == NIL) {          // root 为 NIL 说明树为空
125         cnt = 0, root = alloc(val, NIL);
126         return;
127     }
128     Node *x = root, *y;
129     while (1) {
130         /* 插入新节点的时候, 从根节点向下更新以 i 为根的子树结点数 */
131         x->size++;
132         if (val == x->val) {
133             x->cnt++;
134             pushUp(x);
135             y = x;
136             break;
137         } else if (val < x->val) {
138             if (lson(x) != NIL)
139                 x = lson(x);

```

```

140             else {          // 在左儿子处插入新节点
141                 lson(x) = y = alloc(val, x);
142                 break;
143             }
144         } else {
145             if (rson(x) != NIL)
146                 x = rson(x);
147             else {
148                 rson(x) = y = alloc(val, x);
149                 break;
150             }
151         }
152     }
153
154     splay(y, NIL);      // 完成插入后将新节点旋转到根
155 }
156 /**
157 * 查找一个数字所在的结点并返回指针
158 * 操作后，被查找的数会被旋转到根节点
159 */
160 Node *search(int val) {
161     if (root == NIL)
162         return NIL;           // 不存在的结点
163     Node *x = root, *y = NIL;
164     while (1) {
165         if (val == x->val) {
166             y = x;
167             break;
168         } else if (val < x->val) {
169             if (lson(x) != NIL)
170                 x = lson(x);
171             else
172                 break;           // 404
173         } else {
174             if (rson(x) != NIL)
175                 x = rson(x);
176             else
177                 break;           // 404
178         }
179     }
180     splay(x, NIL);      // 完成查找后将离 val 最近的结点伸展到根部
181     return y;
182 }
183 /**
184 * 查询在以 rt 为根的子树中的最小值，只需要一直往左儿子走，走到底就可以
185 */
186 Node *minNode(Node *rt) {
187     Node *y = p(rt);

```

```

188     while (lson(rt) != NIL)
189         rt = lson(rt);
190     splay(rt, y);           // 记录原先 rt 的根节点, 将最小值旋到子树的根部
191     return rt;
192 }
193 /**
194 * 根据值删除结点 (如果一个数有多个, 只删除一个)
195 */
196 void erase(int val) {
197     if (root == NIL)
198         return;           // splay 为空
199     Node *x = search(val), // search 之后, 值为 val 的结点会被旋转到根部
200             *y;
201     if (x == NIL)        // 不存在值为 val 的结点
202         return;
203     if (x->cnt > 1) {   // 值为 val 的数有两个以上, 直接减掉一个
204         x->cnt--;
205         pushUp(x);
206         return;
207     } else if (lson(x) == NIL && rson(x) == NIL) { // x 没有左右儿子, 直接删除
208         root = NIL, cnt = 0;    // 因为 search(val) 将值为 val 的数旋到了根, 所
209         ↳ 以如果没有左右子结点说明树为空
210         return;
211     } else if (lson(x) == NIL) {
212         root = rson(x);
213         p(rson(x)) = NIL;
214         return;
215     } else if (rson(x) == NIL) {
216         root = lson(x);
217         p(lson(x)) = NIL;
218         return;
219     } else {            // 左右子树都为空, 从右子树中找最小替换根
220         y = minNode(rson(x));      // 完成后最小值会被旋到 x 的右儿子
221         p(y) = NIL, p(lson(x)) = y, lson(y) = lson(x);
222         pushUp(y);
223         root = y;
224     }
225 /**
226 * 求某个数是第几大
227 * 直接旋到根, 然后返回左儿子的 size + 1
228 */
229 int rank(int val) {
230     Node *x = search(val);
231     return x == NIL ? 0 /* 404 */ : lson(x)->size + 1;
232 }
233 /**
234 * 求第 k 大的数是多少

```

```

235  /*
236  Node* kth(int k) {
237      if (root == NIL || k > root->size)      // 不存在或比当前数的数量大
238          return NIL;
239      Node *x = root;
240      while (1) {
241          // pushDown(x);
242          if (lson(x)->size + 1 <= k && k <= lson(x)->size + x->cnt)
243              break;        // 当前 x 指向即为目标
244          else if (k <= lson(x)->size)
245              x = lson(x);    // 到左子树中找
246          else
247              k -= lson(x)->size + x->cnt, x = rson(x);
248      }
249      splay(x, NIL);        // 将 x 旋到根部
250      return x;
251  }
252  /* 查询前驱结点 (第一个小于 num 的结点) */
253  int prev(int val) {
254      Node *u = root;
255      int res = -INF;
256      while (u != NIL) {
257          if (u->val < val && u->val > res)
258              res = u->val;
259          if (val > u->val)
260              u = rson(u);
261          else
262              u = lson(u);
263      }
264      return res;
265  }
266  /* 查询后继结点 (第一个大于 num 的结点) */
267  int next(int val) {
268      Node *u = root;
269      int res = INF;
270      while (u != NIL) {
271          if (u->val > val && u->val < res)
272              res = u->val;
273          if (val < u->val)
274              u = lson(u);
275          else
276              u = rson(u);
277      }
278      return res;
279  }
280 } T;

```

Splay 的基本区间操作 (例题: NOI2005 维护数列):

- 操作区间  $[l, r]$ : 首先先把  $(l-1)$  旋到根节点, 再把  $(r+1)$  旋到根节点右儿子上, 那么现在根节点的右儿子的左儿子 (即  $r+1$  的左儿子) 即为所求区间。
- Splay 维护区间最大子序列: 每个结点记录  $lmx, rmx, mmx$ ; 分别表示: 从区间左端点  $l$ /右端点  $r$  开始的连续的前缀最大子序列, 以及该区间的最大子序列。
- 转移方程  $lmx = \max(lson->lmx, lson->sum + x->val + rson->lmx)$ ,  $rmx$  同理对偶;  $mmx = \max(lson->mmx, rson->mmx, lson->rmx + x->val + rson->lmx)$ .
- 将区间  $[l, r]$  变为同一个数: 打标记并修改值,  $sum = size \times val$ , 如果待修改的值  $val > 0$ , 那么  $lmx = rmx = mmx = sum$ ; 否则  $lmx = rmx = 0, mmx = val$ .
- 将区间  $[l, r]$  翻转: 打上修改标记, 并交换左右儿子, 然后交换当前结点的  $lmx$  和  $rmx$ .

### 1.7.5 平衡树测试数据

1. 插入, 2. 删除, 3. 查询某数排名, 4. 按排名查数, 5. 查某个数前驱, 6. 求某个数后继。

Input :	Output :
20	964673
1 964673	964673
5 968705	1
4 1	964673
3 964673	3
5 965257	1
1 915269	1
1 53283	964673
3 964673	964673
3 53283	
3 53283	
1 162641	
5 973984	
1 948119	
2 915269	
2 53283	
6 959161	
1 531821	
1 967521	
2 531821	
1 343410	

## 1.8 Link-Cut Tree

**动态树问题** 要求维护一个**有根树森林**, 支持树的分割、合并等操作, 即支持在线维护树的连通性, 并且在一条链上或路径上做修改。

**实链剖分** 动态树 / LCT 维护的是一个森林, 希望通过实链剖分, 使得动态树的链成为我们指定的、便于求解的链; 对于节点  $u$  及  $u$  的所有子树与  $u$  的连边  $(u, v_i) \in E$ , 选择一条边  $(u, v^*)$ , 以该边进行剖分, 则称  $(u, v^*)$  为**实边**,  $(u, v_i), v_i \neq v^*$  为**虚边**。

### 辅助树

- 若干棵 splay 构成了辅助树, 每棵 splay 维护原树中的一条路径
- 每棵“辅助树”维护的是“一棵原树”
- 若干棵辅助树构成了 LCT, 维护整个森林
- 辅助树中 splay 的根节点的父节点, 指向**原树中当前链的父节点**, 即当前链最顶端的节点它的父节点。
- 每个连通块中恰好有一个节点的父节点为空。

```
1 class LinkCutTree {
2 public:
3     #define lson(x) ch[x][0]
4     #define rson(x) ch[x][1]
5     int ch[maxn][2], fa[maxn], size[maxn], stk[maxn], rev[maxn];
6     inline bool isRoot(int x) {
7         return lson(fa[x]) != x && rson(fa[x]) != x;
8     }
9     inline bool get(int x) {
10        return rson(fa[x]) == x;
11    }
12    inline void pushUp(int x) {
13        size[x] = size[lson(x)] + size[rson(x)] + 1;
14    }
15    inline void reverse(int x) {
16        swap(lson(x), rson(x));
17        rev[x] ^= 1;
18    }
19    inline void pushDown(int x) {
20        if (!rev[x])
21            return;
22        if (lson(x))
23            reverse(lson(x));
24        if (rson(x))
25            reverse(rson(x));
```

```

26     rev[x] = 0;
27 }
28
29 void rotate(int x) {
30     int y = fa[x], z = fa[y], k = get(x), w = ch[x][!k];
31     if (!isRoot(y))
32         ch[z][get(y)] = x;
33     ch[x][!k] = y, ch[y][k] = w;
34     if (w)
35         fa[w] = y;
36     fa[y] = x, fa[x] = z;
37     pushUp(y);
38 }
39
40 void splay(int x) {
41     int top = 0;
42     stk[++top] = x;
43     for (int i = x; !isRoot(i); i = fa[i])
44         stk[++top] = fa[i];
45     while (top)
46         pushDown(stk[top--]);
47     int y, z;
48     while (!isRoot(x)) {
49         y = fa[x], z = fa[y];
50         if (!isRoot(y))
51             rotate((get(x) ^ get(y)) ? x : y);
52         rotate(x);
53     }
54     pushUp(x);
55 }
56
57 // 访问节点 x, 使从根节点到 x 的路径成为实链
58 void access(int x) {
59     for (int y = 0; x; x = fa[y = x])
60         splay(x), rson(x) = y, pushUp(x);
61 }
62 // 将节点 x 变成 Splay 中的根
63 void makeRoot(int x) {
64     access(x), splay(x), reverse(x);
65 }
66 // 判断连通性, 如果 find(x) == find(y) 则 x, y 在同一棵树中
67 int find(int x) {
68     access(x), splay(x);
69     while (lson(x))
70         pushDown(x), x = lson(x);
71     splay(x);
72     return x;
73 }

```

```

74 // 拉出 x=>y 的路径形成一个 splay
75 void split(int x, int y) {
76     makeRoot(x), access(y), splay(y);
77 }
78 // 连边 (x, y)
79 void link(int x, int y) {
80     makeRoot(x);
81     if (find(y) != x)
82         fa[x] = y;
83 }
84 // 断开边 (x, y)
85 void cut(int x, int y) {
86     makeRoot(x);
87     if (find(y) == x && fa[y] == x && !rson(y)) {
88         fa[y] = rson(x) = 0;
89         pushUp(x);
90     }
91 }
92
93 void init() {
94     memset(ch, 0, sizeof ch), memset(fa, 0, sizeof fa);
95     memset(size, 0, sizeof size), memset(rev, 0, sizeof rev);
96 }
97 } T;

```

## 1.9 点分治

### 问题场景

点分治是大规模处理树上路径问题的工具。大意是找到一个点，递归统计其所有子树的答案，然后利用容斥原理或其它方式合并答案，最后得到整棵树的答案。

### 步骤

- 找到整棵树的重心点  $rt$ ，由  $rt$  向下递归求解。
- 统计以  $rt$  为根的子树的答案  $ans'_{rt}$ ，并使用容斥原理、染色法等去除不合法的答案，得到  $rt$  的最终答案  $ans_{rt}$
- 对  $rt$  的子树  $ch_i$  求解，同样先找到以  $ch_i$  为根的子树的重心  $r'$ ，然后从重心  $r'$  向下递归求解，回到步骤 (1)。

例：计算一棵树上距离为  $k$  的点是否存在

```
1 struct Edge {
2     int u, v, next;
3     ll w;
4 };
5 struct Dist {
6     ll w;
7     int par;
8     bool operator < (const Dist &b) const {
9         return w < b.w;
10    }
11 };
12
13 const int MAXN = 10050, MAXM = 150, INF = 0x3f3f3f3f;
14
15 int n, m, cnt = 1, minv = INF, rt = 0, tot = 0, dcnt = 0;
16
17 bool vis[MAXN];
18 int head[MAXN], qry[MAXM], ans[MAXM], size[MAXN], son[MAXN];
19 Edge e[MAXN << 1];
20 Dist d[MAXN];
21
22 void findRoot(int u, int p) {
23     size[u] = 1, son[u] = 0;
24     for (int i = head[u], v; i; i = e[i].next) {
25         if (vis[(v = e[i].v)] || v == p)
26             continue;
27         findRoot(v, u);
28         size[u] += size[v];
29         son[u] = max(son[u], size[v]);
30     }
31 }
```

```

31     son[u] = max(son[u], tot - size[u]);
32     if (son[u] < minv)
33         minv = son[u], rt = u;
34 }
35 void getDist(int u, int p, int par, ll dt) {
36     d[dcnt++] = (Dist) { dt, par };
37     for (int i = head[u], v; i; i = e[i].next) {
38         if (vis[(v = e[i].v)] || p == v)
39             continue;
40         getDist(v, u, par, dt + e[i].w);
41     }
42 }
43 void solve(int cur) {
44     dcnt = 0;
45     for (int i = head[cur], v; i; i = e[i].next) {
46         if (vis[(v = e[i].v)])
47             continue;
48         getDist(v, cur, v, e[i].w);
49     }
50     d[dcnt++] = (Dist){ 0ll, 0 };
51     sort(d, d + dcnt);

52     for (int i = 0; i < m; i++) {
53         if (ans[i])
54             continue;
55         int l = 0;
56         while (l < dcnt && d[l].w + d[dcnt-1].w < qry[i])
57             l++;
58         while (l < dcnt && !ans[i]) {
59             if (qry[i] - d[l].w < d[l].w)
60                 break;
61             int pos = lower_bound(d, d + dcnt, (Dist){ qry[i] - d[l].w, 0 }) - d;
62             while (pos < dcnt && d[pos].par == d[l].par)
63                 pos++;
64             if (pos < dcnt && d[pos].w + d[l].w == qry[i])
65                 ans[i] = 1;
66             l++;
67         }
68     }
69 }
70 }
71 void work(int x) {
72     vis[x] = 1;
73     solve(x);
74     for (int i = head[x], v; i; i = e[i].next) {
75         if (vis[(v = e[i].v)])
76             continue;
77         rt = 0, minv = INF, tot = size[v];
78         findRoot(v, 0);

```

```

79     work(rt);
80 }
81 }
82 int main() {
83     // ...
84     tot = n;
85     findRoot(1, 0);
86     work(rt);
87     // ...
88     return 0;
89 }
```

## 1.10 树链剖分

**问题场景** 链上求和，链上求最值，链上修改，子树修改，子树求和。如：修改和查询点  $x$  到  $y$  的路径的信息；修改和查询以  $x$  为根的子树的信息。

```

1 int size[MAXN], top[MAXN], son[MAXN], dfn[MAXN], dep[MAXN], father[MAXN], dcnt = 1;
2 ll val[MAXN], raw[MAXN];
3 /**
4 * 第一次 DFS, 需要:
5 * 1. 标记每个结点的深度 dep[]
6 * 2. 标记每个结点的父亲 father[]
7 * 3. 标记每个非叶子结点的子树大小 size[]
8 * 4. 标记每个非叶子结点的重儿子编号 son[]
9 */
10 void dfs_calc(int rt, int p, int d) {
11     father[rt] = p, dep[rt] = d, size[rt] = 1;
12     for (int i = head[rt]; i; i = e[i].next)
13         if (e[i].v != father[rt]) {
14             dfs_calc(e[i].v, rt, d + 1);
15             size[rt] += size[e[i].v];
16             if (son[rt] == -1 || size[e[i].v] > size[son[rt]])
17                 son[rt] = e[i].v;
18         }
19 }
20 /**
21 * 第二次 DFS, 需要:
22 * 1. 标记每个点的新编号/DFS 序: dfn[cur]
23 * 2. 根据新编号将值赋到数组中
24 * 3. 处理每个点所在链的顶端 top[cur]
25 * 4. 先处理重儿子, 然后递归处理轻儿子
26 */
27 void dfs_decomposition(int cur, int hnode) {
28     top[cur] = hnode, dfn[cur] = dcnt;
```

```

29     raw[dcnt++] = val[cur];
30     if (son[cur] == -1)           // 没有儿子
31         return;
32     dfs_decomposition(son[cur], hnode);    // 先处理重儿子
33     // 处理轻儿子
34     for (int i = head[cur]; i; i = e[i].next) {
35         int v = e[i].v;
36         if (!dfn[v])
37             dfs_decomposition(v, v);    // 轻儿子单独成新链
38     }
39 }
40 /**
41 * 完成后可以用 raw[] 建立线段树，然后下面是查询和修改操作
42 * 修改和查询 x 到 y 的路径，直接调用就可以；如果要处理以 x 为根的子树，
43 * 因为我们记录了每个非叶结点的子树大小，并且每个子树的新编号都是连续的，
44 * 所以直接线段树区间操作 [dfn[x], dfn[x] + size[x] - 1] 即可
45 */
46 ll query_path(int x, int y) {
47     ll ans = 0;
48     while (top[x] != top[y]) {
49         if (dep[top[x]] < dep[top[y]])
50             std::swap(x, y);
51         ans = (ans + t.query(1, 1, n, dfn[top[x]], dfn[x])) % t.mod;
52         x = father[top[x]];
53     }
54     if (dep[x] > dep[y])
55         std::swap(x, y);
56     ans = (ans + t.query(1, 1, n, dfn[x], dfn[y])) % t.mod;
57     return ans;
58 }
59 /* 更新从 (x, y) 的路径 */
60 void update_path(int x, int y, ll val) {
61     while (top[x] != top[y]) {
62         if (dep[top[x]] < dep[top[y]])
63             std::swap(x, y);
64         t.update(1, 1, n, dfn[top[x]], dfn[x], val);
65         x = father[top[x]];
66     }
67     if (dep[x] > dep[y])
68         std::swap(x, y);
69     t.update(1, 1, n, dfn[x], dfn[y], val);
70 }

```

## 1.11 树套树

### 1.11.1 线段树套平衡树 (Treap)

**应用场景** 维护多维度的信息，例如查询某个数在区间内的排名、查询区间第  $k$  大，单点修改，查询区间内  $k$  的前驱和后继等等。

```
1 class SegTree {
2 public:
3     Treap::Node* t[MAXN << 2];
4     SegTree() {
5         fill(t, t + (MAXN << 2), T.NIL);
6     }
7     void build(int rt, int l, int r) {
8         for (int i = l; i <= r; i++)
9             T.insert(t[rt], a[i]);
10        if (l != r) {
11            int mid = (l + r) >> 1;
12            build(rt << 1, l, mid);
13            build(rt << 1 | 1, mid + 1, r);
14        }
15    }
16    void update(int rt, int l, int r, int x, int y) {
17        T.erase(t[rt], a[x]);
18        T.insert(t[rt], y);
19        if (l == r)
20            return;
21        int mid = (l + r) >> 1;
22        if (x <= mid)
23            update(rt << 1, l, mid, x, y);
24        else
25            update(rt << 1 | 1, mid + 1, r, x, y);
26    }
27    int rank(int rt, int l, int r, int ql, int qr, int k) {
28        if (qr < l || ql > r)
29            return 0;
30        if (ql <= l && r <= qr)
31            return T.rank(t[rt], k);
32        else {
33            int mid = (l + r) >> 1;
34            return rank(rt << 1, l, mid, ql, qr, k) + rank(rt << 1 | 1, mid + 1, r,
35                ql, qr, k);
36        }
37    }
38    int kth(int x, int y, int k) {
39        int minx = 0, maxx = 1e8, mid, tmp;
40        while (minx < maxx) {
41            mid = (minx + maxx + 1) >> 1;
```

```

41     if ((tmp = rank(1, 1, n, x, y, mid)) < k)
42         minx = mid;
43     else
44         maxx = mid - 1;
45 }
46 return maxx;
47 }
48 int prev(int rt, int l, int r, int ql, int qr, int k) {
49     if (qr < l || ql > r)
50         return -INT_MAX;
51     if (ql <= l && r <= qr)
52         return T.prev(t[rt], k);
53     else {
54         int mid = (l + r) >> 1;
55         return max(prev(rt << 1, l, mid, ql, qr, k), prev(rt << 1 | 1, mid + 1,
56             → r, ql, qr, k));
57     }
58 }
59 int next(int rt, int l, int r, int ql, int qr, int k) {
60     if (qr < l || ql > r)
61         return INT_MAX;
62     if (ql <= l && r <= qr)
63         return T.next(t[rt], k);
64     else {
65         int mid = (l + r) >> 1;
66         return min(next(rt << 1, l, mid, ql, qr, k), next(rt << 1 | 1, mid + 1,
67             → r, ql, qr, k));
68     }
69 }
70 };

```

## 1.11.2 树状数组套动态开点权值线段树

**应用场景** 查询一段区间内数的个数 + 单点修改。

```

1 const int maxn = 2e5 + 10, lim = 2e5;
2
3 // 动态开点权值线段树
4 class SegTree {
5 public:
6     int tot;
7     struct Node {
8         ll sum;
9         int lc, rc;
10    } t[maxn * 80];
11
12    SegTree(): tot(0) {}

```

```

13     void pushUp(int x) {
14         t[x].sum = t[t[x].lc].sum + t[t[x].rc].sum;
15     }
16     void update(int &root, int l, int r, int x, int val) {
17         if (!root)
18             root = ++tot;
19         if (l == r) {
20             t[root].sum += val;
21             return;
22         }
23         int mid = (l + r) >> 1;
24         if (x <= mid)
25             update(t[root].lc, l, mid, x, val);
26         else
27             update(t[root].rc, mid + 1, r, x, val);
28         pushUp(root);
29     }
30     int query(int root, int l, int r, int ql, int qr) {
31         if (!root)
32             root = ++tot;
33         if (l >= ql && r <= qr)
34             return t[root].sum;
35         int mid = (l + r) >> 1, ans = 0;
36         if (ql <= mid)
37             ans += query(t[root].lc, l, mid, ql, qr);
38         if (qr > mid)
39             ans += query(t[root].rc, mid + 1, r, ql, qr);
40         return ans;
41     }
42 } T;
43
44 /* 树状数组 */
45 // rt[i] 保存每个单点对应的权值线段树根节点, rt1, rt2 分别表示区间 [l, r] 的树根
46 int rt[maxn], rt1[maxn], rt2[maxn], cnt1, cnt2, n, m;
47 int lowbit(int x) {
48     return x & (-x);
49 }
50 // 单点修改: update_BIT(i, a[i], ± a[i])
51 void update_BIT(int pos, int x, int val) {
52     for (int i = pos; i <= n; i += lowbit(i))
53         T.update(rt[i], 1, lim, x, val);
54 }
55 // 求区间 [l, r] 的端点在两端的根
56 void locate(int l, int r) {
57     cnt1 = cnt2 = 0;
58     for (int i = l-1; i > 0; i -= lowbit(i))
59         rt1[cnt1++] = rt[i];
60     for (int i = r; i > 0; i -= lowbit(i))

```

```

61         rt2[cnt2++] = rt[i];
62     }
63 // 求区间 [l, r] 有多少个数字 k
64 ll ask(int l, int r, int k) {
65     ll ans = 0;
66     if (r == k) {
67         for (int i = 0; i < cnt1; i++)
68             ans -= T.t[rt1[i]].sum;
69         for (int i = 0; i < cnt2; i++)
70             ans += T.t[rt2[i]].sum;
71         return ans;
72     }
73     int mid = (l + r) >> 1;
74     if (k <= mid) {
75         for (int i = 0; i < cnt1; i++)
76             rt1[i] = T.t[rt1[i]].lc;
77         for (int i = 0; i < cnt2; i++)
78             rt2[i] = T.t[rt2[i]].lc;
79         return ask(l, mid, k);
80     } else {
81         for (int i = 0; i < cnt1; i++)
82             ans -= T.t[T.t[rt1[i]].lc].sum;
83         for (int i = 0; i < cnt2; i++)
84             ans += T.t[T.t[rt2[i]].lc].sum;
85         for (int i = 0; i < cnt1; i++)
86             rt1[i] = T.t[rt1[i]].rc;
87         for (int i = 0; i < cnt2; i++)
88             rt2[i] = T.t[rt2[i]].rc;
89         return ans + ask(mid + 1, r, k);
90     }
91 }
```

## 1.12 虚树

**问题场景** 给定一棵树，有多次询问，每组询问对树上的一些关键点，询问它们的某些信息，满足所有询问中的关键点数量**总和**与树的大小同阶。此时可以对原树建立一棵只包含关键点和 LCA 的虚树，可以将问题的复杂度压缩到  $O(Q \log_2 N + f(Q))$  的等级。

对一棵有根树而言，虚树中包含了所有询问中的关键点和它们的 LCA，尽可能地压缩了树的大小。同时，虚树当中对于连接两个点  $u, v$ ，边  $(u, v)$  的边权定义为：原树中  $u$  到  $v$  的最短路径。

**关键性质** 在一棵有根树中，任选  $k$  个不重复的点，这  $k$  个点两两之间的不同 LCA 数量不超过  $k-1$  个（即  $k$  个点两两之间的 LCA 数量不超过自身的阶数）。

### 构造虚树

- 首先对树上所有点进行 DFS，预处理处各个点的 DFS 序和 LCA。
- 对所有的关键点按照 DFS 序排序，然后顺序将关键点插入；新增一个超级根（0 号节点）指向原来的树根。
- 用栈  $S$  来维护虚树的右链，初始时  $S$  内只有 0 号节点。如果此时要加入节点  $u$ ：
  - 首先求出  $u$  与栈顶元素  $v = S[top]$  的 LCA 点  $p$ 。
  - 如果  $p = v$ ，说明  $(p, v)$  是一条链，此时将  $v$  入栈。
  - 如果  $p \neq v$ ，那就说明  $u, v$  在  $p$  的不同子树中，此时不断退出栈顶元素  $S[top]$ ，直到栈顶的 \*\*第二个元素\*\*（P.S. 加入超级节点的原因）的深度  $depth[S[top - 1]] < depth[p]$ 。
  - 在退栈的过程中，将栈顶元素  $v_0$  和第二个元素  $v_1$  连边  $(v_1, v_0)$ 。
  - 重复该过程直到所有的点都被加入。

```
1 namespace VirtualTree {
2     int anc[maxn][maxlog + 5], dfn[maxn], depth[maxn], vdepth[maxn];
3     bool vis[maxn];
4     vector<int> VT[maxn];
5
6     void dfs(int rt, int dep, int p) {
7         anc[rt][0] = p, dfn[rt] = dcnt++, depth[rt] = dep, vis[rt] = 1;
8         for (int i = 1; (1 << i) <= dep; i++)
9             anc[rt][i] = anc[anc[rt][i-1]][i-1];
10        for (int i = head[rt]; i; i = e[i].next) {
11            if (e[i].v == p)
12                continue;
13            if (!vis[e[i].v])
14                dfs(e[i].v, dep + 1, rt);
15        }
16    }
17 }
```

```

18     int lca(int a, int b) {
19         if (depth[a] < depth[b])
20             swap(a, b);
21         int d = depth[a] - depth[b];
22         for (int i = 0; (1 << i) <= d; i++)
23             if ((1 << i) & d)
24                 a = anc[a][i];
25         if (a == b)
26             return a;
27         for (int i = maxlog ; i >= 0; i--) {
28             if (anc[a][i] != anc[b][i])
29                 a = anc[a][i], b = anc[b][i];
30         }
31         return anc[a][0];
32     }
33
34     inline bool cmp(const int &i, const int &j) {
35         return dfn[i] < dfn[j];
36     }
37
38     void build_virtual_tree(int node[], int k) {
39         static int st[maxn];
40         sort(node, node + k, cmp), sz = 0;
41         for (int i = 0; i < k; i++) {
42             if (i == 0)
43                 assert(node[i] == 1);
44             if (sz <= 1) {
45                 st[++sz] = node[i];
46                 continue;
47             }
48             int u = node[i], p = lca(u, st[sz]);
49             if (p == st[sz]) {
50                 st[++sz] = u;
51                 continue;
52             }
53             while (sz > 1 && dfn[p] <= dfn[st[sz-1]])
54                 VT[st[sz-1]].emplace_back(st[sz]), sz--;
55             if (p != st[sz])
56                 VT[p].emplace_back(st[sz]), st[sz] = p;
57             st[++sz] = u;
58         }
59         for (int i = 1; i < sz; i++)
60             VT[st[i]].emplace_back(st[i+1]);
61     }
62
63     void calc_vt_depth(int x, int p) {
64         fa[x] = p, vdepth[x] = vdepth[p] + 1;
65         for (int i : VT[x])

```

```

66         calc_vt_depth(i, x);
67     }
68 }

69 // 处理路径 (u, v) 上的更新和询问
70 ll a[maxn], w[maxn];
71 // 更新
72 while (u != v) {
73     if (rdp[u] < rdp[v])
74         swap(u, v);
75     w[u] += x, a[u] += x, u = fa[u];
76 }
77 a[u] += x;
78 // 询问
79 ll ans = 0;
80 while (u != v) {
81     if (rdp[u] < rdp[v])
82         swap(u, v);
83     if (depth[u] - depth[fa[u]] - 1)
84         ans += w[u] * (depth[u] - depth[fa[u]] - 1); // 用 depth 统计
85     ans += a[u];
86 }
87 ans += a[u];
88

```

## 1.13 K-D Tree

### 1.13.1 求第 k 近点对距离

```

1 namespace KDTree {
2     int n, k;
3     priority_queue<int, vector<int>, greater<int> > q;
4     struct node {
5         int x, y;
6     } s[maxn];
7     bool cmp1(node a, node b) { return a.x < b.x; }
8     bool cmp2(node a, node b) { return a.y < b.y; }
9     int lc[maxn], rc[maxn], L[maxn], R[maxn], D[maxn], U[maxn];
10    void maintain(int x) {
11        L[x] = R[x] = s[x].x;
12        D[x] = U[x] = s[x].y;
13        if (lc[x])
14            L[x] = min(L[x], L[lc[x]]), R[x] = max(R[x], R[lc[x]]),
15            D[x] = min(D[x], D[lc[x]]), U[x] = max(U[x], U[lc[x]]);
16        if (rc[x])

```

```

17     L[x] = min(L[x], L[rc[x]]), R[x] = max(R[x], R[rc[x]]),
18     D[x] = min(D[x], D[rc[x]]), U[x] = max(U[x], U[rc[x]]);
19 }
20 int build(int l, int r) {
21     if (l > r) return 0;
22     int mid = (l + r) >> 1;
23     double av1 = 0, av2 = 0, va1 = 0, va2 = 0; // average variance
24     for (int i = l; i <= r; i++) av1 += s[i].x, av2 += s[i].y;
25     av1 /= (r - l + 1);
26     av2 /= (r - l + 1);
27     for (int i = l; i <= r; i++)
28         va1 += (av1 - s[i].x) * (av1 - s[i].x),
29         va2 += (av2 - s[i].y) * (av2 - s[i].y);
30     if (va1 > va2)
31         nth_element(s + l, s + mid, s + r + 1, cmp1);
32     else
33         nth_element(s + l, s + mid, s + r + 1, cmp2);
34     lc[mid] = build(l, mid - 1);
35     rc[mid] = build(mid + 1, r);
36     maintain(mid);
37     return mid;
38 }
39 int sq(int x) { return x * x; }
40 int dist(int a, int b) {
41     return max(sq(s[a].x - L[b]), sq(s[a].x - R[b])) +
42            max(sq(s[a].y - D[b]), sq(s[a].y - U[b]));
43 }
44 void query(int l, int r, int x) {
45     if (l > r) return;
46     int mid = (l + r) >> 1, t = sq(s[mid].x - s[x].x) + sq(s[mid].y - s[x].y);
47     if (t > q.top()) q.pop(), q.push(t);
48     int distl = dist(x, lc[mid]), distr = dist(x, rc[mid]);
49     if (distl > q.top() && distr > q.top()) {
50         if (distl > distr) {
51             query(l, mid - 1, x);
52             if (distr > q.top()) query(mid + 1, r, x);
53         } else {
54             query(mid + 1, r, x);
55             if (distl > q.top()) query(l, mid - 1, x);
56         }
57     } else {
58         if (distl > q.top()) query(l, mid - 1, x);
59         if (distr > q.top()) query(mid + 1, r, x);
60     }
61 }
62 ll solve(int _n, int _k) {
63     n = _n, k = _k * 2;
64     for (int i = 1; i <= k; i++)

```

```

65         q.push(0);
66     for (int i = 1; i <= n; i++)
67         read(s[i].x), read(s[i].y);
68     build(1, n);
69     for (int i = 1; i <= n; i++)
70         query(1, n, i);
71     return q.top();
72 }
73 }
```

### 1.13.2 动态维护二维空间信息

```

1  namespace KDTree {
2      int n, op, xl, xr, yl, yr, lstans;
3      struct node {
4          int x, y, v;
5      } s[maxn];
6      bool cmp1(int a, int b) { return s[a].x < s[b].x; }
7      bool cmp2(int a, int b) { return s[a].y < s[b].y; }
8      double a = 0.725;
9      int rt, cur, d[maxn], lc[maxn], rc[maxn], L[maxn], R[maxn], D[maxn], U[maxn],
10         siz[maxn], sum[maxn];
11     int g[maxn], t;
12     void print(int x) {
13         if (!x) return;
14         print(lc[x]);
15         g[++t] = x;
16         print(rc[x]);
17     }
18     void maintain(int x) {
19         siz[x] = siz[lc[x]] + siz[rc[x]] + 1;
20         sum[x] = sum[lc[x]] + sum[rc[x]] + s[x].v;
21         L[x] = R[x] = s[x].x;
22         D[x] = U[x] = s[x].y;
23         if (lc[x])
24             L[x] = min(L[x], L[lc[x]]), R[x] = max(R[x], R[lc[x]]),
25             D[x] = min(D[x], D[lc[x]]), U[x] = max(U[x], U[lc[x]]);
26         if (rc[x])
27             L[x] = min(L[x], L[rc[x]]), R[x] = max(R[x], R[rc[x]]),
28             D[x] = min(D[x], D[rc[x]]), U[x] = max(U[x], U[rc[x]]);
29     }
30     int build(int l, int r) {
31         if (l > r) return 0;
32         int mid = (l + r) >> 1;
33         double av1 = 0, av2 = 0, va1 = 0, va2 = 0;
34         for (int i = l; i <= r; i++) av1 += s[g[i]].x, av2 += s[g[i]].y;
```

```

35     av1 /= (r - l + 1);
36     av2 /= (r - l + 1);
37     for (int i = l; i <= r; i++)
38         va1 += (av1 - s[g[i]].x) * (av1 - s[g[i]].x),
39                 va2 += (av2 - s[g[i]].y) * (av2 - s[g[i]].y);
40     if (va1 > va2)
41         nth_element(g + l, g + mid, g + r + 1, cmp1), d[g[mid]] = 1;
42     else
43         nth_element(g + l, g + mid, g + r + 1, cmp2), d[g[mid]] = 2;
44     lc[g[mid]] = build(l, mid - 1);
45     rc[g[mid]] = build(mid + 1, r);
46     maintain(g[mid]);
47     return g[mid];
48 }
49 void rebuild(int& x) {
50     t = 0;
51     print(x);
52     x = build(1, t);
53 }
54 bool bad(int x) { return a * siz[x] <= (double)max(siz[lc[x]], siz[rc[x]]); }
55 void insert(int& x, int v) {
56     if (!x) {
57         x = v;
58         maintain(x);
59         return;
60     }
61     if (d[x] == 1) {
62         if (s[v].x <= s[x].x)
63             insert(lc[x], v);
64         else
65             insert(rc[x], v);
66     } else {
67         if (s[v].y <= s[x].y)
68             insert(lc[x], v);
69         else
70             insert(rc[x], v);
71     }
72     maintain(x);
73     if (bad(x)) rebuild(x);
74 }
75 int query(int x) {
76     if (!x || xr < L[x] || xl > R[x] || yr < D[x] || yl > U[x]) return 0;
77     if (xl <= L[x] && R[x] <= xr && yl <= D[x] && U[x] <= yr) return sum[x];
78     int ret = 0;
79     if (xl <= s[x].x && s[x].x <= xr && yl <= s[x].y && s[x].y <= yr)
80         ret += s[x].v;
81     return query(lc[x]) + query(rc[x]) + ret;
82 }

```

```

83     void add(int x, int y, int v) {
84         cur++, s[cur].x = x, s[cur].y = y, s[cur].v = v;
85         insert(rt, cur);
86     }
87     ll query(int _xl, int _yl, int _xr, int _yr) {
88         xl = _xl, yl = _yl, xr = _xr, yr = _yr;
89         return query(rt);
90     }
91 }
```

## 1.14 启发式合并 (DST on Tree)

**问题场景** 给定一棵树，要求在  $O(N \log N)$  的时间内，计算所有子树中含有某种性质的点；只包含查询操作，不包含修改。

```

1 // CCPC2020 长春 - E. Strange Memory
2 int sz[maxn], in[maxn], out[maxn], ver[maxn << 2];
3
4 vector<int> G[maxn];
5
6 void getsz(int u, int p)
7 {
8     sz[u] = 1;
9     in[u] = ++dfn;
10    ver[dfn] = u;
11    for (auto v : G[u]) {
12        if (v == p)
13            continue;
14        getsz(v, u);
15        sz[u] += sz[v];
16    }
17    out[u] = ++dfn;
18}
19
20 void dsu_on_tree(int u, int p, bool keep)
21 {
22     int mx = -1, heavy = -1;
23     for (auto v : G[u])
24         if (v != p && sz[v] > mx)
25             mx = sz[v], heavy = v;
26     for (auto v : G[u])
27         if (v != p && v != heavy)
28             dsu_on_tree(v, u, 0);
29     if (heavy != -1)
30         dsu_on_tree(heavy, u, 1);
```

```

31     for (auto v : G[u]) {
32         if (v == p || v == heavy)
33             continue;
34         for (int i = in[v]; i < out[v]; i++) {
35             if (ver[i] == 0)
36                 continue;
37             // 统计轻子节点的贡献
38             int bitwise = (a[u] ^ a[ver[i]]);
39             for (int j = 0; j < maxlog; j++) {
40                 int bt = ((ver[i] >> j) & 1);
41                 ans[j] += tot[bitwise][j][!bt];
42             }
43         }
44         // 把轻子节点加进影响中
45         for (int i = in[v]; i < out[v]; i++) {
46             if (ver[i] == 0)
47                 continue;
48             for (int j = 0; j < maxlog; j++) {
49                 int bt = (ver[i] >> j) & 1;
50                 tot[a[ver[i]]][j][bt]++;
51             }
52         }
53     }
54     // 统计自己的贡献，把自己加进影响中
55     for (int j = 0; j < maxlog; j++) {
56         int bt = (u >> j) & 1;
57         tot[a[u]][j][bt]++;
58     }
59     // 如果当前节点不是重儿子，则消除影响
60     if (!keep) {
61         for (int i = in[u]; i < out[u]; i++) {
62             if (ver[i] == 0)
63                 continue;
64             for (int j = 0; j < maxlog; j++) {
65                 int bt = (ver[i] >> j) & 1;
66                 tot[a[ver[i]]][j][bt]--;
67             }
68         }
69     }
70 }

```

# Chapter 2

## 图论

### 2.1 最短路

#### 2.1.1 Dijkstra

```
1 int dist[MAXN], vis[MAXN];
2 int dijkstra(int s, int t) {
3     priority_queue<pair<int, int> > pq;
4     memset(vis, 0, sizeof vis), fill(dist, dist + n, inf);
5     dist[s] = 0, pq.push(make_pair(0, s));
6     while (!pq.empty()) {
7         int cur = pq.top().second;
8         pq.pop();
9         if (vis[cur])
10            continue;
11         vis[cur] = 1;
12         for (int i = head[cur]; i; i = e[i].next) {
13             int y = e[i].v;
14             if (dist[y] > dist[cur] + e[i].w) {
15                 dist[y] = dist[cur] + e[i].w;
16                 pq.push(make_pair(-dist[y], y));
17             }
18         }
19     }
20     return dist[t];
21 }
```

#### 2.1.2 SPFA 算法

**原始版本** 适合有负权边的图或边数较少的图。

```

1 int dist[MAXN], vis[MAXN];
2 int SPFA(int s, int t) {
3     memset(vis, 0, sizeof vis), fill(dist, dist + n, inf);
4     queue<int> Q;
5     Q.push(s), vis[s] = 1, dist[s] = 0;
6     while (!Q.empty()) {
7         int cur = Q.front();
8         Q.pop(), vis[cur] = 0;
9         for (int i = head[cur], v, w; i; i = e[i].next) {
10            v = e[i].v, w = e[i].w;
11            if (dist[v] > dist[cur] + w) {
12                dist[v] = dist[cur] + w;
13                if (!vis[v])
14                    Q.push(v), vis[v] = 1;
15            }
16        }
17    }
18    return dist[t];
19 }
```

**SLF 优化**：使用双端队列优化，设从  $u$  拓展出了  $v$ 、队首元素  $k$ ，如果  $dist[v] < dist[k]$ ，则将  $v$  加入队首，否则加入队尾。

```

1 int dist[MAXN], vis[MAXN];
2 int SPFA(int s, int t) {
3     memset(vis, 0, sizeof vis), fill(dist, dist + n, inf);
4     deque<int> Q;
5     Q.push_back(s), vis[s] = 1, dist[s] = 0;
6     while (!Q.empty()) {
7         int cur = Q.front();
8         Q.pop_front(), vis[cur] = 0;
9         for (int i = head[cur], v, w; i; i = e[i].next) {
10            v = e[i].v, w = e[i].w;
11            if (dist[v] > dist[cur] + w) {
12                dist[v] = dist[cur] + w;
13                if (!vis[v])
14                    if (Q.size() && dist[v] >= dist[Q.front()])
15                        Q.push_back(v), vis[v] = 1;
16                    else
17                        Q.push_front(v), vis[v] = 1;
18            }
19        }
20    }
21    return dist[t];
22 }
```

**LLL 优化**：设队首元素为  $k$ , 每次松弛时加入判断, 设队列中所有  $dist$  的平均值为  $x$ , 若  $dist[k] > x$ , 则将  $k$  移到队尾, 查找下一个元素; 直到使得  $dist[k] \leq x$  再进行松弛。

```
1 int dist[MAXN], vis[MAXN];
2 int SPFA(int s, int t) {
3     int u, v, num = 0;
4     ll x = 0;
5     deque<int> q;
6     fill(dist, dist + n, inf), memset(vis, 0, sizeof vis);
7     dist[s] = 0, vis[s] = 1, q.push_back(s), num++;
8     while (q.size()) {
9         u = q.front(), q.pop_front(), num--, x -= dist[u];
10        while (num && dist[u] > x / num)
11            q.push_back(u), u = q.front(), q.pop_front();
12        vis[u] = 0;
13        for (int i = head[u]; i; i = e[i].next) {
14            v = e[i].v;
15            if (dist[v] > dist[u] + e[i].w) {
16                dist[v] = dist[u] + e[i].w;
17                if (!vis[v]) {
18                    vis[v] = 1, num++, x += dist[v];
19                    if (q.size() && dist[v] < dist[q.front()])
20                        q.push_front(v);
21                    else
22                        q.push_back(v);
23                }
24            }
25        }
26    }
27    return dist[t];
28 }
```

## 2.2 负环判定

### 2.2.1 BFS 判负环

基于 SPFA 的负环判定，使用  $incnt[v]$  记录节点  $v$  的入队次数；在松弛的时候有状态转移  $incnt[u] = incnt[v] + 1$ ，如果有一个点的  $incnt[v] \geq n$ ，说明存在负环。

```
1 | bool spfa(int s) {
2 |     memset(vis, 0, sizeof vis);
3 |     dist[s] = 0;
4 |     vis[s] = 1;
5 |     queue<int> q;
6 |     q.push(s);
7 |     while (q.size()) {
8 |         int cur = q.front();
9 |         q.pop();
10 |        vis[cur] = 0;
11 |        for (int i = head[cur]; i; i = e[i].next) {
12 |            int v = e[i].v;
13 |            if (dist[v] > dist[cur] + e[i].w) {
14 |                dist[v] = dist[cur] + e[i].w;
15 |                incnt[v] = incnt[cur] + 1;
16 |                if (incnt[v] >= n)
17 |                    return true;
18 |                if (!vis[v])
19 |                    vis[v] = 1, q.push(v);
20 |            }
21 |        }
22 |    }
23 |    return false;
24 | }
```

### 2.2.2 DFS 判负环

```
1 | int dist[maxn];
2 | bool vis[maxn];
3 | bool SPFA(int x) {
4 |     vis[x] = 1;
5 |     for (int i = head[x], v; i; i = e[i].next) {
6 |         v = e[i].v;
7 |         if (dist[v] < dist[x] + e[i].w) {
8 |             dist[v] = dist[x] + e[i].w;
9 |             if (vis[v])
10 |                 return 0;
11 |             if (!SPFA(v))
12 |                 return 0;
```

```
13     }
14 }
15 vis[x] = 0;
16 return 1;
17 }
```

## 2.3 最小生成树

### 2.3.1 Kruskal 算法

适用于一般图。

```
1 int p[MAXN];
2 int find(int x) {
3     return x == p[x] ? x : p[x] = find(p[x]);
4 }
5 int main() {
6     // ...
7     for (int i = 1; i <= n; i++)
8         p[i] = i;
9     sort(e, e + m);
10    int cnt = 0, ans = 0;
11    for (int i = 1; i <= ecnt && cnt < n; i++) {
12        int px = find(e[i].u), py = find(e[i].v);
13        if (px != py) {
14            p[px] = py;
15            ans += e[i].w;
16            cnt++;
17        }
18    }
19    if (cnt != n)
20        printf("No solution\n");
21    else
22        printf("%d", ans);
23    return 0;
24 }
```

### 2.3.2 堆优化的 Prim 算法

适用于边多点少的稠密图。

```
1 struct Edge {
2     int u, v, w, next;
3 } e[MAXM << 1];
4 struct Node {
5     int v, w;
6     bool operator < (const Node b) const {
7         return w > b.w;
8     }
9 };
10 priority_queue<Node> pq;
11 int n, m, head[MAXN], cnt = 1;
12 bool vis[MAXN];
13 void add_edge(int u, int v, int w) {
14     e[cnt] = (Edge) { u, v, w, head[u] };
15     head[u] = cnt++;
16 }
17
18 int main() {
19     // ...
20     vis[0] = 1;
21     pq.push((Node) {1, 0});
22     LL ans = 0;
23     int cnt = 0;
24     while (!pq.empty() && cnt <= n) {
25         Node cur = pq.top();
26         pq.pop();
27         if (vis[cur.v])
28             continue;
29         vis[cur.v] = 1;
30         ans += cur.w;
31         cnt++;
32         for (int i = head[cur.v]; i; i = e[i].next) {
33             if (!vis[e[i].v])
34                 pq.push((Node) {e[i].v, e[i].w});
35         }
36     }
37     if (cnt != n) {
38         printf("No solution!");
39     } else {
40         printf("%lld", ans);
41     }
42 }
```

### 2.3.3 最小瓶颈路

**问题描述** 给定一个加权无向图，并给定无向图中两个结点  $u$  和  $v$ ，求  $u$  到  $v$  的一条路径，使得路径上边的最大权值最小。

可以证明这个“最大权值最小”的边一定在最小生成树上。对于询问两个点  $u, v$  的最小瓶颈路的问题，我们对求完的最小生成树用 Tarjan 或者倍增求一遍最近公共祖先 LCA，两者到达 LCA 的路径中的最大边就是最小瓶颈路了。

```
1 int query(int x, int y) {
2     int d = 0;
3     if (depth[x] < depth[y])
4         swap(x, y);
5     for (d = 0; (1 << (d + 1)) <= depth[x]; d++);
6     int ans = -1;
7     for (int i = d; i >= 0; i--)
8         if (depth[x] - (1 << i) >= depth[y]) {
9             ans = max(ans, cost[x][i]);
10            x = ancestor[x][i];
11        }
12        if (x == y)
13            return ans;
14        for (int i = d; i >= 0; i--)
15            if (ancestor[x][i] > 0 && ancestor[x][i] != ancestor[y][i]) {
16                ans = max(ans, max(cost[x][i], cost[y][i]));
17                x = ancestor[x][i], y = ancestor[y][i];
18            }
19        ans = max(ans, max(cost[x][0], cost[y][0]));
20    return ans;
21 }
```

### 2.3.4 最小直径生成树

**定义** 在无向图的所有生成树中，直径长度最小的一棵生成树。

```
1 bool cmp(int a, int b) {
2     return val[a] < val[b];
3 }
4 void floyd() {
5     for (int k = 1; k <= n; k++)
6         for (int i = 1; i <= n; i++)
7             for (int j = 1; j <= n; j++)
8                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
9 }
10 int solve() {
11     floyd();
```

```

12     for (int i = 1; i <= n; i++) {
13         for (int j = 1; j <= n; j++)
14             rk[i][j] = j, val[j] = d[i][j];
15         sort(rk[i] + 1, rk[i] + 1 + n, cmp);
16     }
17     int ans = INF;
18     // 图的绝对中心可能在结点上
19     for (int i = 1; i <= n; i++)
20         ans = min(ans, d[i][rk[i][n]] * 2);
21     // 图的绝对中心可能在边上
22     for (int i = 1; i <= m; i++) {
23         int u = a[i].u, v = a[i].v, w = a[i].w;
24         for (int p = n, i = n - 1; i >= 1; i--)
25             if (d[v][rk[u][i]] > d[v][rk[u][p]])
26                 ans = min(ans, d[u][rk[u][i]] + d[v][rk[u][p]] + w),
27                 p = i;
28     }
29     return ans;
30 }
```

### 2.3.5 最小曼哈顿距离生成树

```

1 struct P {
2     int x, y, id;
3     bool operator < (const P &rhs) const {
4         return x != rhs.x ? x < rhs.x : y < rhs.y;
5     }
6 }p[MAXN];
7
8 namespace ManhattanMST {
9     int tot, fa[MAXN], a[MAXN], b[MAXN];
10    pair<int, int> bit[MAXN];
11    struct Edge {
12        int u, v, w;
13        bool operator < (const Edge &rhs) const {
14            return w < rhs.w;
15        }
16    } edge[MAXN << 2];
17    int getfather(int x) { return x == fa[x] ? x : (fa[x] = getfather(fa[x])); }
18    int dis(P a, P b) { return abs(a.x - b.x) + abs(a.y - b.y); }
19    int lowbit(int x) { return x & (-x); }
20    void insert(int x, int val, int pos) {
21        for(int i = x; i; i -= lowbit(i))
22            if(val < bit[i].first)
23                bit[i] = {val, pos};
24    }
}
```

```

25     int query(int x, int lmt) {
26         int ret1 = INT_MAX, ret2 = -1;
27         for(int i = x; i <= lmt; i += lowbit(i))
28             if(ret1 > bit[i].first)
29                 ret1 = bit[i].first, ret2 = bit[i].second;
30         return ret2;
31     }
32     int solve() {
33         tot = 0;
34         for(int dir = 0; dir <= 3; ++ dir) {
35             if (dir & 1)
36                 for(int i = 1; i <= n; ++ i)
37                     swap(p[i].x, p[i].y);
38             else if (dir)
39                 for(int i = 1; i <= n; ++ i)
40                     p[i].x = - p[i].x;
41             sort(p + 1, n + p + 1);
42             for(int i = 1; i <= n; ++ i)
43                 a[i] = b[i] = p[i].y - p[i].x;
44             sort(b + 1, n + b + 1);
45             int lmt = unique(b + 1, n + b + 1) - b - 1;
46             for(int i = 1; i <= lmt; ++ i) bit[i] = {INT_MAX, -1};
47             for(int i = n; i; -- i) {
48                 int pos = lower_bound(b + 1, lmt + b + 1, a[i]) - b;
49                 int ans = query(pos, lmt);
50                 if(ans != -1) {
51                     edge[++ tot].u = p[i].id;
52                     edge[tot].v = p[ans].id;
53                     edge[tot].w = dis(p[i], p[ans]);
54                 }
55                 insert(pos, p[i].x + p[i].y, i);
56             }
57         }
58         sort(edge + 1, tot + edge + 1);
59         for(int i = 1; i <= n; ++ i) fa[i] = i;
60         int cnt = 0, ret = 0;
61         for(int i = 1; i <= tot; ++ i) {
62             int u = getfather(edge[i].u);
63             int v = getfather(edge[i].v);
64             if(u == v) continue;
65             fa[u] = v;
66             ++ cnt; ret += edge[i].w;
67             if(cnt == n - 1) break;
68         }
69         return ret;
70     }
71 }
```

## 2.4 Tarjan

### 2.4.1 求割边（桥）

```
1  namespace CutEdge {
2      int cnt = 0;
3      vector<pair<int, int>> G;
4      void tarjan(int x, int in_edge) {
5          dfn[x] = low[x] = ++cnt;
6          for (int i = head[x]; i; i = e[i].next) {
7              int y = e[i].v;
8              if (!dfn[y]) {
9                  tarjan(y, i), low[x] = min(low[x], low[y]);
10                 if (low[y] > dfn[x])
11                     G.emplace_back(make_pair(e[i].u, e[i].v));
12             }
13             else if (i != (in_edge ^ 1))
14                 low[x] = min(low[x], dfn[y]);
15         }
16     }
17     vector<pair<int, int>> solve() {
18         for (int i = 1; i <= n; i++)
19             if (!dfn[i])
20                 tarjan(i, 0);
21         return G;
22     }
23 }
```

### 2.4.2 求割点

```
1  namespace CutPoint {
2      int cnt = 0, root = 0;
3      vector<int> P;
4      void tarjan(int x) {
5          dfn[x] = low[x] = ++cnt;
6          int child_count = 0;
7          for (int i = head[x]; i; i = e[i].next) {
8              int y = e[i].v;
9              if (!dfn[y]) {
10                  tarjan(y);
11                  low[x] = min(low[x], low[y]);
12                  if (low[y] >= dfn[x]) {
```

```

13         child_count++;
14         if (x != root || child_count > 1)
15             P.emplace_back(x);
16     }
17 } else
18     low[x] = min(low[x], dfn[y]);
19 }
20 }
21 vector<int> solve() {
22     for (int i = 1; i <= n; i++)
23         if (!dfn[i])
24             root = i, tarjan(i);
25     return P;
26 }
27 }
```

### 2.4.3 求无向图点双连通分量 & 缩点

**点双连通** 在一个无向图中，若任意两点间至少存在两条“点不重复”的路径，则说这个图是点双连通的。

**点双连通分量** 在一个无向图中，点双连通的极大子图称为点双连通分量。

**等价定义** 点双连通图定义等价于：任意两条边都在同一个简单环中。

```

1 namespace vDCC {
2     int dfn[MAXN], low[MAXN], dcnt = 0, root = 0, ans = 0;
3     bool cut[MAXN];           // 记录点 i 是不是割点
4     stack<int> s;
5     vector<int> dcc[MAXN];
6     void tarjan(int x) {
7         dfn[x] = low[x] = dcnt++;
8         s.push(x);
9         if (x == root && head[x] == -1) {          // 孤立点，单独成图
10             dcc[ans++].push_back(x);
11             return;
12         }
13         for (int i = head[x]; i; i = e[i].next) {
14             int y = e[i].v;
15             if (!dfn[y]) {
16                 tarjan(y), low[x] = min(low[x], low[y]);
17                 if (low[y] >= dfn[x]) {        // x 是割点
18                     cut[x] = 1;
19                     int tmp;
20                     do {
21                         tmp = s.top(), s.pop();
22                         if (tmp == y) break;
23                         low[tmp] = dfn[tmp];
24                     }
```

```

22             dcc[ans].push_back(tmp);
23         } while (tmp != y);
24         dcc[ans].push_back(x);
25         ans++;
26     }
27 } else
28     low[x] = std::min(low[x], dfn[y]);
29 }
30 }
31 int solve(int n) {
32     for (int i = 1; i <= n; i++)
33         if (!dfn[i])
34             root = i, tarjan(i);
35     return ans; // 返回点双连通分量个数
36 }
37
38 /* 缩点 */
39 Edge ec[MAXN << 1];
40 int hc[MAXN], cdcnt = 2, new_id[MAXN], num;
41 void add_vdcc_edge(int u, int v) {
42     ec[cdcnt] = (Edge) {u, v, hc[u]};
43     hc[u] = cdcnt++;
44 }
45 void shrink() {
46     num = ans;
47     // 给割点编号
48     for (int i = 1; i <= n; i++)
49         if (cut[i])
50             new_id[i] = ++num;
51     for (int i = 0; i < ans; i++) {           // ans 是 v-DCC 数 - 1
52         for (int j = 0; j < dcc[i].size(); j++) {
53             int x = dcc[i][j];
54             if (cut[x])
55                 add_vdcc_edge(i, new_id[x]), add_vdcc_edge(new_id[x].i);
56             else
57                 c[x] = i;           // 除了割点之外, 标记其它的点只属于一个 v-DCC
58         }
59     }
60     // 节点数: ans, 边数: cdcnt / 2
61     // for (int i = 2; i < tc; i += 2)
62     //     printf("%d %d\n", ec[i].u, ec[i].v);
63 }
64 }

```

## 2.4.4 求无向图的边双连通分量 & 缩点

**边双连通** 如果任意两点至少存在两条边不重复路径, 则称该图为边双连通的。

**等价定义** 边双连通图的定义等价于：任意一条边至少在一个简单环中。

```
1  namespace eDCC {
2      // 为了进行异或运算找到边，必须设定前向星 cnt 起点为 2
3      int dfn[MAXN], low[MAXN], dcnt = 0;
4      int c[MAXN], dcc = 0;
5      bool is_bridge[MAXN];
6      void tarjan(int x, int in_edge) {
7          dfn[x] = low[x] = dcnt++;
8          for (int i = head[x]; i; i = e[i].next) {
9              int y = e[i].v;
10             if (!dfn[y]) {
11                 tarjan(y, i), low[x] = min(low[x], low[y]);
12                 if (low[y] > dfn[x])           // x 是割边
13                     is_bridge[i] = is_bridge[i ^ 1] = 1;    // 标记该边及其反向边为
14                     ← 桥
15             } else if (i != (in_edge ^ 1))
16                 low[x] = std::min(low[x], dfn[y]);
17         }
18     }
19     void dfs(int x) {
20         c[x] = dcc;
21         for (int i = head[x]; i; i = e[i].next) {
22             int y = e[i].v;
23             // 节点 y 已被访问或者 (x,y) 是桥
24             if (c[y] || is_bridge[i])
25                 continue;
26             dfs(y);
27         }
28     }
29     void solve(int n) {
30         for (int i = 1; i <= n; i++)
31             if (!dfn[i])
32                 tarjan(i, 0);
33         for (int i = 1; i <= n; i++)
34             if (!c[i])
35                 dcc++, dfs(i);
36         printf("There are %d e-DCCs.\n", dcc);
37         for (int i = 1; i <= n; i++)
38             printf("node %d belongs to e-dcc %d\n", i, c[i]);
39     }
40
41     // 缩点
42     Edge ec[MAXN << 1];
43     int hc[MAXN], cecnt = 2;
44     void add_cut_edge(int u, int v) {
45         ec[cecnt] = (Edge) {u, v, hc[u]};
        hc[u] = cecnt++;
    }
```

```

46 }
47 void shrink() {
48     for (int i = 2; i <= cnt; i++) { // cnt 是原图的边数
49         int x = e[i].x, y = e[i].y;
50         if (c[x] == c[y])
51             continue; // x, y 同属一个 e-DCC, 无事可做
52         add_cut_edge(x, y); // 否则将 x, y 加入新图中
53     }
54     // 新图有 dcc 个点, cecnt / 2 条边
55     // for (int i = 2; i < cecnt; i += 2)
56     //     printf("%d %d\n", ec[i].u, ec[i].v);
57 }
58 }
```

## 2.4.5 求有向图强连通分量 & 缩点

```

1 namespace SCC {
2     int dfn[MAXN], low[MAXN], c[MAXN]; // c[i] 表示节点 i 所属的 scc 编号
3     int scccnt = 0, order = 1;
4     bool instack[MAXN];
5     vector<int> scc[MAXN];
6     stack<int> s;
7     void tarjan(int x) {
8         dfn[x] = low[x] = order++;
9         s.push(x), instack[x] = 1;
10        for (int i = head[x]; i; i = e[i].next) {
11            int y = e[i].v;
12            if (!dfn[y])
13                tarjan(y), low[x] = min(low[x], low[y]);
14            else if (instack[y])
15                low[x] = min(low[x], dfn[y]);
16        }
17        if (dfn[x] == low[x]) {
18            int tmp;
19            do {
20                tmp = s.top(), s.pop();
21                c[tmp] = scccnt, instack[tmp] = 0;
22                scc[scccnt].push_back(tmp);
23            } while (tmp != x);
24            sccnt++;
25        }
26    }
27
28    // 缩点
29    Edge ec[MAXN];
30    int hc[MAXN], ccnt = 1;
```

```
31     void add_scc_edge(int u, int v) {
32         ec[ccnt] = (Edge) { u, v, hc[u] };
33         hc[u] = ccnt++;
34     }
35     void shrink() {
36         for (int x = 1; x <= n; x++) {
37             for (int i = head[x]; i; i = e[i].next) {
38                 int y = e[i].v;
39                 if (c[x] == c[y])
40                     continue;
41                 add_scc_edge(c[x], c[y]);
42             }
43         }
44     }
45 }
```

## 2.5 拓扑排序

```
1 struct Edge {
2     int u, v, next;
3 } e[MAXM];
4 int head[MAXN], in[MAXN], ans[MAXN], cnt = 0;
5 queue<int> q;
6 for (int i = 0; i < ecnt; i++)      // 统计入度
7     in[e[i].v]++;
8 for (int i = 0; i < n; i++)          // 将入度为 0 的节点加入队列中
9     if (in[i] == 0)
10         q.push(in[i]);
11 while (!q.empty()) {
12     int next = q.front();
13     q.pop();
14     ans[cnt++] = next;
15     for (int i = head[next]; i != 0; i = e[i].next) {
16         in[e[i].v]--;
17         if (in[e[i].v] == 0)
18             q.push(e[i].v);
19     }
20 }
21 if (cnt == n - 1)      // 判断是否存在拓扑序列
22     for (int i = 0; i < cnt; i++)
23         printf("%d", ans[i]);
24 else
25     printf("No Answer!");
```

## 2.6 欧拉回路

```
1 stack<int> ans;
2 multiset<int> e[MAXN];
3 void dfs(int x) {
4     // 遍历与 x 相连的每条无向边
5     for (multiset<int>::iterator itor = e[x].begin(); itor != e[x].end(); itor =
6         → e[x].begin()) {
7         // 删除边 (注意是无向图, 要删两次) 并对下一个节点遍历
8         int tmp = *itor;
9         to[x].erase(itor), to[u].erase(to[u].find(x));
10        dfs(tmp);
11    }
12    // DFS 完成后, 将当前点加入队列
13    ans.push(x);
```

```

13 }
14 while (!ans.empty()) { // 倒序输出栈中的内容即为答案
15     printf("%d", ans.top());
16     ans.pop();
17 }

```

此外也有一种删边的做法是每次搜索边  $e_{is}$  的时候，标记  $used[e_{ij}] = 1$  来表示删边，适用于用前向星存储图时的情况。

## 2.7 哈密尔顿路径

```

1 int map[MAXN][MAXN], ans[MAXN];
2 bool vis[MAXN];
3 void hamilton() {
4     int s = 1, t, ansi = 2;      // 任选一个节点为起点
5     memset(vis, 0, sizeof vis);
6     for (int i = 1; i <= n; i++)
7         if (map[s][i]) {        // 找到一个与起点相连的点为终点
8             t = i;
9             break;
10        }
11    ans[0] = s, ans[1] = t;
12    vis[s] = 1, vis[t] = 1;      // 设置起点、终点的访问状态
13    for (;;) {
14        for (;;) {
15            int i;
16            for (i = 1; i <= n; i++)
17                if (map[t][i] && !vis[i]) {
18                    ans[ansi++] = i;
19                    vis[i] = true;
20                    t = i;
21                    break;
22                }
23                if (i > n)
24                    break;
25        }
26        std::reverse(ans, ans + ansi);
27        std::swap(s, t);
28        for (;;) {
29            int i;
30            for (i = 1; i <= n; i++)
31                if (map[t][i] && !vis[i]) {
32                    ans[ansi++] = i;
33                    vis[i] = true;
34                    t = i;
35                    break;
36                }
37        }
38    }
39 }

```

```

36     }
37     if (i > n)
38         break;
39 }
40 int mid = 0;
41 if (!map[s][t]) {
42     for (int i = 1; i < ansi - 2; i++)
43         if (map[s][ans[i + 1]] && map[ans[i]][t]) {
44             mid = i + 1;
45             break;
46         }
47     std::reverse(ans + mid, ans + ansi);
48     t = ans[ansi - 1];
49 }

50 if (ansi == n)
51     return;
52 for (int i = 1; i <= n; i++)
53     if (!vis[i]) {
54         int j;
55         for (j = 1; j < ansi - 1; j++)
56             if (map[ans[j]][i]) {
57                 mid = j;
58                 break;
59             }
60         if (map[ans[mid]][i]) {
61             t = i, mid = j;
62             break;
63         }
64     }
65 }
66 s = ans[mid-1];
67 std::reverse(ans, ans + mid);
68 std::reverse(ans + mid, ans + ansi);
69 ans[ansi++] = t;
70 vis[t] = 1;
71 }
72 }
```

## 2.8 最近公共祖先 (LCA)

### 2.8.1 倍增法求 LCA

```
1 int anc[MAXN][MAXLOG + 5], depth[MAXN];
2 bool vis[MAXN];
3 void dfs(int rt, int dep, int p) {
4     anc[rt][0] = p, depth[rt] = dep, vis[rt] = 1;
5     for (int i = 1; (1 << i) <= dep; i++)
6         anc[rt][i] = anc[anc[rt][i-1]][i-1];
7     for (int i = head[rt]; i; i = e[i].next)
8         if (!vis[e[i].v])
9             dfs(e[i].v, dep + 1, rt);
10 }
11 int lca(int a, int b) {
12     if (depth[a] < depth[b])
13         swap(a, b);
14     int delta = depth[a] - depth[b];
15     for (int i = 0; i <= MAXLOG; i++)
16         if (delta & (1 << i))
17             a = anc[a][i];
18     if (a == b)
19         return b;
20     for (int i = MAXLOG; i >= 0; i--)
21         if (anc[a][i] != anc[b][i])
22             a = anc[a][i], b = anc[b][i];
23     return anc[a][0];
24 }
```

### 2.8.2 Tarjan 求 LCA

```
1 struct Edge {
2     int v, next, res;
3 } e[MAXN << 1], query[MAXM << 1];
4 int head[MAXN], qhead[MAXN], q[MAXN], cnt = 1, qcnt = 2;
5 bool vis[MAXN];
6 void add_query(int x, int y) {
7     query[qcnt] = (Edge) {y, qhead[x], 0};
8     qhead[x] = qcmt++;
9 }
10 int find(int x) {
11     return x == p[x] ? x : p[x] = find(p[x]);
12 }
13 void tarjan(int t) {
14     vis[t] = 1;
```

```

15     for (int i = head[t]; i; i = e[i].next)
16         if (!vis[e[i].v]) {
17             tarjan(e[i].v);
18             p[e[i].v] = t;
19         }
20     for (int i = qhead[t]; i; i = query[i].next)
21         if (vis[query[i].v]) {
22             query[i].res = find(query[i].v);
23             query[i^1].res = query[i].res;
24         }
25 }
26 void solve(int n, int m) {
27     for (int i = 0; i < m; i++) {
28         read(a), read(b);
29         add_query(a, b);
30         add_query(b, a);
31     }
32     for (int i = 0; i < n; i++)
33         p[i] = i;
34     tarjan(s);
35     for (int i = 2; i < qcnt; i += 2) {
36         printf("%d\n", query[i].res);
37     }
38 }
```

## 2.9 K 短路

```

1 namespace KShortestPath {
2     // 启发式搜索求 K 短路, 来源: CCPC2019 网络赛 1004
3     struct HeapNode {
4         int cur, eid;
5         ll dist;
6         HeapNode(int cur, int eid, ll dist): cur(cur), eid(eid), dist(dist) {}
7         bool operator < (const HeapNode &b) const {
8             return dist > b.dist;
9         }
10    };
11    priority_queue<HeapNode> pq;
12
13    void solve(int n, int q) {
14        for (rint i = 1; i <= n; i++)
15            sort(e[i].begin(), e[i].end());
16        int maxk = -1, cnt = 0;
17        for (rint i = 0; i < q; i++) {
18            cin >> k[i];
19        }
20    }
21 }
```

```

19         maxk = max(maxk, k[i]);
20     }
21     while (pq.size())
22     {
23         pq.pop();
24         for (int i = 1; i <= n; i++)
25             if (e[i].size())
26                 pq.push(HeapNode(i, 0, e[i][0].w));
27         while (1)
28         {
29             int cur = pq.top().cur, eid = pq.top().eid;
30             ll dist = pq.top().dist;
31             pq.pop();
32             ans[++cnt] = dist;
33             if (cnt == maxk)
34                 break;
35             if (eid + 1 < (int)e[cur].size())
36                 pq.push(HeapNode(cur, eid + 1, dist - e[cur][eid].w +
37                                 e[cur][eid+1].w));
38             if (e[e[cur][eid].v].size())
39                 pq.push(HeapNode(e[cur][eid].v, 0, dist + e[e[cur][eid].v][0].w));
40         }
41         for (int i = 0; i < q; i++)
42             cout << ans[k[i]] << endl;
43     }
44 }
```

## 2.10 树 & 子树的重心

**定义** 找到一个点, 其所有的子树中最大的子树节点数最少, 那么这个点就是这棵树的重心, 删去重心后, 生成的多棵树尽可能平衡.

### 性质

- 树中所有点到某个点的距离和中, 到重心的距离和是最小的; 如果有两个重心, 那么他们的距离和一样.
- 把两个树通过一条边相连得到一个新的树, 那么新的树的重心在连接原来两个树的重心的路径上.
- 把一个树添加或删除一个叶子, 那么它的重心最多只移动一条边的距离.

**应用** 树的重心在树的点分治中有重要的作用, 可以避免  $O(n^2)$  的极端复杂度 (从退化链的一端出发), 保证  $N \log N$  的复杂度.

```

1 int sz[MAXN], maxcnt[MAXN], rt;           // 每个结点的儿子所在子树的最大结点数, rt 为
2   ↳ 所求重心
3 void getCenter(int u, int fa) {
4     maxcnt[u] = 0;
5     for (int i = head[u]; i; i = e[i].next) {
6         if (e[i].v == fa)
7             continue;
8         getCenter(e[i].v, u);
9         maxcnt[u] = max(maxcnt[u], sz[e[i].v]);
10    }
11    maxcnt[u] = max(maxcnt[u]. n - sz[u]);
12    if (maxcnt[u] < maxcnt[rt])
13        rt = u;
}

```

**原理** 利用重心的性质：把两个树通过一条边相连得到一个新的树，那么新的树的重心在连接原来两个树的重心的路径上。由于一棵树的重心可能不只有一个，因此默认保存深度更深的重心，然后在深度更深的重心与根节点之间查找另一个重心。

**性质**  $a$  可能是以  $rt$  为根的子树的重心，当且仅当满足  $size[rt] - size[a] > size[a]$ 。

```

1 // 子树大小 & 父节点 & 深度 & 以 i 为根的子树较深的重心
2 int size[maxn], fa[maxn], depth[maxn], c[maxn];
3
4 // 在 (a, b) 之间寻找以 rt 为根的子树重心
5 void update(int rt, int a, int b) {
6     while (depth[a] > depth[rt] && size[rt] - size[a] > size[a])
7         a = fa[a];
8     while (depth[b] > depth[rt] && size[rt] - size[b] > size[b])
9         b = fa[b];
10    c[rt] = depth[a] > depth[b] ? a : b;
11}
12
13 void dfs(int u, int p) {
14     fa[u] = p, size[u] = 1, c[u] = u;
15     for (int i = head[u]; i; i = e[i].next) {
16         int v = e[i].v;
17         if (v == p)
18             continue;
19         depth[v] = depth[u] + 1, dfs(v, u);
20         size[u] += size[v];
21         update(u, c[u], c[v]);
22     }
23 }
24
25 void solve(int n) {

```

```

26     depth[1] = 1, dfs(1, 0);
27     int ans[10] = {0};
28     for (int i = 1; i <= n; i++) {
29         int cnt = 0;
30         ans[cnt++] = c[i];
31         if (fa[c[i]] && size[i] - size[c[i]] == size[c[i]])
32             ans[cnt++] = fa[c[i]];
33         sort(ans, ans + cnt);
34         for (int j = 0; j < cnt; j++)
35             printf("%d%c", ans[j], " \n"[j == cnt-1]);
36     }
37 }
```

## 2.11 树的直径

任选一个点找到最远的点，然后再从那个最远点 s 找离 s 最远的点 t， $\text{dist}(s, t)$  即为树的直径。

```

1 int diameter() {
2     queue<pair<int, int> > q;
3     q.push(make_pair(1, 0)), vis[1] = 1;
4     int idx = 0, maxl = 0;
5     while (!q.empty()) {
6         int cur = q.front().first, len = q.front().second;
7         q.pop();
8         if (len > maxl)
9             idx = cur, maxl = len;
10        for (int i = head[cur], v; i; i = e[i].next) {
11            v = e[i].v;
12            if (!vis[v])
13                q.push(make_pair(v, len + 1)), vis[v] = 1;
14        }
15    }
16    memset(vis, 0, sizeof vis);
17    q.push(make_pair(idx, 0)), maxl = 0, vis[idx] = 1;
18    while (!q.empty()) {
19        int cur = q.front().first, len = q.front().second;
20        q.pop();
21        maxl = max(maxl, len);
22        for (int i = head[cur], v; i; i = e[i].next) {
23            v = e[i].v;
24            if (!vis[v])
25                q.push(make_pair(v, len + 1)), vis[v] = 1;
26        }
27    }
28    return maxl;
29 }
```

## 2.12 Kirchoff 矩阵树定理

- 问题场景：解决一张图（有向图、无向图）的生成树个数计数问题。
- 使用前提：计数的图都允许重边，但是不允许自环。

### 2.12.1 无向图中的矩阵树定理

设  $G$  是一个有  $n$  个顶点的无向图，定义：

- 度数矩阵  $D(G) = \begin{cases} d_{ij} = \deg(i), i = j \\ d_{ij} = 0, i \neq j \end{cases}$
- 邻接矩阵  $A(G) = \begin{cases} a_{ij} = a_{ji} = c(i, j), i \neq j \\ a_{ij} = a_{ji} = 0, i = j \end{cases}$ ，其中  $c(i, j)$  为点  $i$  与  $j$  相连的边的数量。
- 拉普拉斯 (Laplace) 矩阵 (或 Kirchoff 基尔霍夫矩阵)  $L(G) = D(G) - A(G)$
- 图  $G$  的所有生成树个数  $t(G)$

则无向图的矩阵树定理描述为：

- 行列式描述：对于  $\forall i$ ，有  $t(G) = \det[L(G, i)]$ ，其中  $L(G, i)$  表示 Laplace 矩阵中去掉第  $i$  行和第  $i$  列后构成的子矩阵， $\det(L)$  表示取  $L$  的行列式。
- 特征值描述：设  $\lambda_1, \lambda_2, \dots, \lambda_{n-1}$  为  $L(G)$  的  $n-1$  个**非零特征值**，则  $t(G) = \frac{1}{n}(\lambda_1, \lambda_2, \dots, \lambda_{n-1})$
- 性质：对于无向图的 Laplace 矩阵，它的  $n-1$  阶主子式都相等。

### 2.12.2 有向图中的矩阵树定理

设  $G$  是一个有  $n$  个顶点的有向图，定义：

- 出度矩阵  $D^{out}(G) = \begin{cases} d_{ij} = \deg^{out}(i), i = j \\ d_{ij} = 0, i \neq j \end{cases}$
- 入度矩阵  $D^{in}(G) = \begin{cases} d_{ij} = \deg^{in}(i), i = j \\ d_{ij} = 0, i \neq j \end{cases}$
- 邻接矩阵  $A(G) = \begin{cases} a_{ij} = a_{ji} = c(i, j) \\ a_{ij} = a_{ji} = 0, i = j \end{cases}$ ，其中  $c(i, j)$  为点  $i$  与点  $j$  相连的边的数量。
- 出度 Laplace 矩阵  $L^{out}(G) = D^{out}(G) - A(G)$

- 入度 Laplace 矩阵  $L^{in}(G) = D^{in}(G) - A(G)$
- 图 G 所有以 r 为根的所有根向树形图（基图为树，所有边指向父亲）的个数为  $t^{root}(G, r)$
- 图 G 所有以 r 为根的所有叶向树形图（基图为树，所有边指向儿子）的个数为  $t^{leaf}(G, r)$

则有向图的矩阵树定理描述为：

- 根向形式：对于  $\forall k$ , 有  $t^{root}(G, k) = \det [L^{out}(G, k)]$
- 叶向形式：对于  $\forall k$ , 有  $t^{leaf}(G, k) = \det [L^{in}(G, k)]$
- 如果要统计一张有向图中所有的根向/叶向树形图，只要枚举所有的根  $k$  并对  $t^{root}(G, r)$  或  $t^{leaf}(G, r)$  求和即可

BEST 定理：设  $G$  是有向欧拉图，那么  $G$  的不同欧拉回路总数  $ec(G) = t^{root}(G, k) \prod_{v \in V} (\deg(v) - 1)!$  对于欧拉图  $G$  的任意两个节点  $k, k'$ , 都有  $t^{root}(G, k) = t^{root}(G, k')$ , 且欧拉图  $G$  所有节点的出度和入度相等。

```

1  namespace MatrixTree {
2      ll L[maxn][maxn];
3
4      ll mod_pow(ll a, ll b) {
5          ll ans = 1ll;
6          while (b) {
7              if (b & 1)
8                  ans = ans * a % mod;
9              a = a * a % mod, b >= 1;
10         }
11         return ans;
12     }
13
14     void add_edge(int u, int v, ll val, bool dir) {
15         if (!dir)
16             L[u][u] = (L[u][u] + val) % mod,
17             L[v][v] = (L[v][v] + val) % mod,
18             L[u][v] = (L[u][v] - val + mod) % mod,
19             L[v][u] = (L[v][u] - val + mod) % mod;
20         else
21             L[v][v] = (L[v][v] + val + mod) % mod,
22             L[u][v] = (L[u][v] - val + mod) % mod;
23     }
24
25     ll det(int n) {
26         ll inv, tmp, ans = 1;
27         for (int i = 2; i <= n; i++) {
28             for (int j = i + 1; j <= n; j++)
29                 if (!L[i][i] && L[j][i]) {
30                     ans *= -1, swap(L[i], L[j]);
31                 }
32         }
33         for (int i = 1; i <= n; i++)
34             ans *= mod_pow(L[i][i], n - 1);
35         return ans;
36     }
37 }
```

```

31             break;
32         }
33         inv = mod_pow(L[i][i], mod - 2);
34         for (int j = i + 1; j <= n; j++) {
35             tmp = L[j][i] * inv % mod;
36             for (int k = i; k <= n; k++)
37                 L[j][k] = (L[j][k] - L[i][k] * tmp % mod + mod) % mod;
38         }
39     }
40     for (int i = 2; i <= n; i++)
41         ans = ans * L[i][i] % mod;
42     return ans;
43 }
44 }
```

## 2.13 2-SAT 问题

**定义** 简单的说就是给出  $n$  个集合，每个集合有两个元素，已知若干个  $\langle a, b \rangle$ ，表示  $a$  与  $b$  矛盾（其中  $a$  与  $b$  属于不同的集合）。然后从每个集合选择一个元素，判断能否一共选  $n$  个两两不矛盾的元素。

**原理** 首先建图：假设两个集合  $\{a_1, b_1\}$  和  $\{a_2, b_2\}$ ，如果  $a_1, b_2$  冲突，那么连有向边  $(a_1, b_1)$  和  $(b_2, a_2)$ ，然后跑一遍 Tarjan 有向图缩点，判断是否有一个集合的两个元素同时在同一个 SCC 中，如果有则无解，否则有解。

**输出方案** Tarjan 算法求强连通分量时用了栈，求得各点所在的 SCC 编号相当于反拓扑序。对于任意集合可以表示为  $\{x, \neg x\}$ ；如果变量  $\neg x$  的拓扑序在  $x$  之后（即  $\text{topo}(\neg x) \geq \text{topo}(x)$ ），则取  $x$  为真。

### 2.13.1 写法一 (tarjan 缩点)

```

1  namespace TwoSAT {
2      #define clear(x) memset(x, 0, sizeof(x))
3      int head[maxn], dfn[maxn], low[maxn], c[maxn], stk[maxn];
4      int top = 0, scccnt = 0, order = 1, cnt = 1;
5      bool instack[maxn];
6      struct Edge {
7          int u, v, next;
8          Edge(int u = 0, int v = 0, int next = 0): u(u), v(v), next(next) {}
9      } e[maxm];
10     void add_edge(int u, int v) {
11         e[cnt] = Edge(u, v, head[u]);
12         head[u] = cnt++;
13     }
14 }
```

```

13 }
14 void init() {
15     clear(dfn), clear(low), clear(c), clear(instack), clear(head);
16     scccnt = 0, order = 1, cnt = 1, top = 0;
17 }
18 void tarjan(int x) {
19     dfn[x] = low[x] = order++;
20     stk[++top] = x, instack[x] = 1;
21     for (int i = head[x]; i; i = e[i].next) {
22         int y = e[i].v;
23         if (!dfn[y])
24             tarjan(y), low[x] = min(low[x], low[y]);
25         else if (instack[y])
26             low[x] = min(low[x], dfn[y]);
27     }
28     if (dfn[x] == low[x]) {
29         int tmp;
30         do {
31             tmp = stk[top--];
32             c[tmp] = sccnt, instack[tmp] = 0;
33         } while (tmp != x);
34         sccnt++;
35     }
36 }
37 void shrink(int n) {
38     for (int x = 0; x <= n; x++)
39         if (!dfn[x])
40             tarjan(x);
41 }
42 bool solve(int n) {
43     shrink(n);
44     for (int i = 0; i <= n; i += 2)
45         if (c[i] == c[i+1])
46             return false;
47     return true;
48 }
49 }
```

## 2.13.2 写法二 (暴力)

```

1 struct Twosat {
2     int n;
3     vector<int> g[maxn * 2];
4     bool mark[maxn * 2];
5     int s[maxn * 2], c;
6     bool dfs(int x) {
```

```

7     if (mark[x ^ 1]) return false;
8     if (mark[x]) return true;
9     mark[x] = true, s[c++] = x;
10    for (int i = 0; i < (int)g[x].size(); i++)
11        if (!dfs(g[x][i])) return false;
12    return true;
13 }
14 void init(int n) {
15     this->n = n;
16     for (int i = 0; i < n * 2; i++) g[i].clear();
17     memset(mark, 0, sizeof(mark));
18 }
19 void add_clause(int x, int y) { // 这个函数随题意变化
20     g[x].push_back(y ^ 1);           // 选了 x 就必须选 y^1
21     g[y].push_back(x ^ 1);
22 }
23 bool solve() {
24     for (int i = 0; i < n * 2; i += 2)
25         if (!mark[i] && !mark[i + 1]) {
26             c = 0;
27             if (!dfs(i)) {
28                 while (c > 0) mark[s[--c]] = false;
29                 if (!dfs(i + 1)) return false;
30             }
31         }
32     return true;
33 }
34 };

```

## 2.14 图的度序列判断

### 2.14.1 Erdos 定理

非负整数序列  $(d_1, d_2, \dots, d_p)$  ( $d_1 \geq d_2 \geq \dots \geq d_p$ ) 是图的度序列，当且仅当：

- $\sum(di) (1 \leq i \leq p)$  为偶数 (即满足图的度序列的条件)
- 并且对于一切整数  $k, 1 \leq k \leq p-1$ , 有  $\sum(di) (1 \leq i \leq k) \leq k * (k-1) + \sum \min(k, di) (k+1 \leq i \leq p)$ .

### 2.14.2 Havel 定理

- 把序列排成不增序，即  $d_1 \geq d_2 \geq \dots \geq d_n$ ，则  $d$  可简单图化当且仅当  $d' = (d_2-1, d_3-1, \dots, d(d_1+1)-1, d(d_1+2), d(d_1+3), \dots, d_n)$  可简单图化。
- 实际上就是说，我们把  $d$  排序以后，找出度最大的点 (设度为  $d_1$ )，把它和度次大的  $d_1$  个点之间连边，然后这个点就可以不管了，一直继续这个过程，直到建出完整的图，或出现负度等明显不合理的情况。

# Chapter 3

## 网络流/二分图/匹配

### 3.1 二分图

#### 3.1.1 二分图匹配-匈牙利算法

```
1 int match[MAXN], vis[MAXN];
2 bool dfs(int x) {
3     // 遍历 x 的每条出边
4     for (int i = head[x], y; i; i = e[i].next) {
5         y = e[i].v;
6         // 如果在当前递归 DFS 的过程中, y 没有被访问过
7         if (!vis[y]) {
8             vis[y] = 1;      // 先将 y 分配给 x, 标记 y 被访问
9             // 如果 y 没有被匹配, 那就让它与 x 匹配
10            // 否则, 尝试对 y 已匹配的边匹配其他的节点, 然后再让 y 与 x 匹配
11            if (!match[y] || dfs(match[y])) {
12                match[y] = x;
13                return true;
14            }
15        }
16    }
17    // 如果走到了这里, 说明该点匹配失败
18    return false;
19 }
20 int main() {
21     /* ... */
22     int ans = 0;
23     for (int i = 1; i <= n; i++) {
24         // 重设 vis 数组, 表示所有点都不在增广路中
25         memset(vis, 0, sizeof vis);
26         // 尝试为每个点匹配 (找增广路), 如果匹配成功则 ans++
27         if (dfs(i))
28             ans++;
29     }
}
```

```
30     /* ... */  
31 }
```

### 3.1.2 二分图判定-染色算法

```
1 int vis[MAXN], flag = 1;  
2 void dfs(int x, int cur) {  
3     vis[x] = cur;  
4     for (int i = head[x]; i; i = e[i].next) {  
5         if (vis[e[i].v] == 0)  
6             dfs(vis[e[i].v], 3 - cur);           // 注意这里的小技巧, 3-1=2, 3-2=1.  
7         else if (vis[e[i].v] == cur) {  
8             flag = 0;  
9             return;  
10        }  
11    }  
12 }  
13 for (int i = 0; i < n; i++) {  
14     if (!vis[i] && flag)  
15         dfs(i, 1);  
16     if (!flag)  
17         break;  
18 }  
19 printf(flag ? "Yes" : "No");
```

## 3.2 最大流

### 3.2.1 Edmonds-Karp 增广路算法

算法思想：不断地使用 BFS 寻找增广路，直至网络上不存在增广路位置。BFS 时只考虑  $f(x, y) < c(x, y)$  的边并找到一条  $S -> T$  的路径，同时计算出路径上各边的剩余容量最小值  $minf$ ，则网络的流量可以增加  $minf$ 。根据斜对称性质，该算法增广时需要考虑正向边和反向边。找到增广路之后就回溯更新增广路及其反向边的剩余容量。

```
1  namespace EdmondsKarp {
2      Edge e[MAXN << 1];
3      int cnt, head[MAXN], flow[MAXN], prev[MAXN], s, t, maxflow;
4      bool vis[MAXN];
5
6      void init(int ts, int tt) {
7          memset(head, 0, sizeof head);
8          memset(flow, 0, sizeof flow);
9          memset(prev, 0, sizeof prev);
10         cnt = 2, s = ts, t = tt, maxflow = 0;
11     }
12
13     void add_edge(int x, int y, int z) {
14         e[cnt] = (Edge) {x, y, z, head[x]};
15         head[x] = cnt++;
16         e[cnt] = (Edge) {y, x, 0, head[y]}; // 容量为 0 的反向边
17         head[y] = cnt++;
18     }
19
20     bool bfs() { // 寻找增广路
21         memset(vis, 0, sizeof vis);
22         queue<int> q;
23         q.push(s);
24         vis[s] = 1;
25         flow[s] = INF;
26         while (q.size()) {
27             int cur = q.front();
28             q.pop();
29             for (int i = head[cur]; i; i = e[i].next) {
30                 if (e[i].c) { // 存在这条边并有剩余容量
31                     int y = e[i].v;
32                     if (vis[y])
33                         continue;
34                     flow[y] = min(flow[cur], e[i].c);
35                     prev[y] = i; // 记录前驱便于更新
36                     q.push(y);
37                     vis[y] = 1;
38
39                     if (y == t) { // 到达汇点，返回增广路存在
40                         return true;
41                     }
42                 }
43             }
44         }
45         return false;
46     }
47
48     void print() {
49         cout << "MaxFlow: " << maxflow << endl;
50         cout << "Flow: ";
51         for (int i = 0; i < MAXN; i++)
52             cout << flow[i] << " ";
53         cout << endl;
54     }
55 }
```

```

40             return true;
41         }
42     } // if
43     } // for
44 } // while
45     return false;      // 增广路不存在
46 } // bool dfs
47
48 void update() {    // 当增广路存在，更新增广路及其反向边的剩余容量
49     int x = t, i;
50     while (x != s) {
51         i = prev[x], e[i].c -= flow[t];
52         e[i ^ 1].c += flow[t];
53         x = e[i].u;
54     }
55     maxflow += flow[t];
56 }
57
58 int work() {        // 主算法
59     while (bfs())
60         update();
61     return maxflow;
62 }
63 }
```

### 3.2.2 Dinic 算法 (带当前弧优化)

不断在残量图上使用 BFS 求出结点的层次，构建分层图（即给结点标注  $d[i]$ ）；然后在分层图上 DFS 寻找增广路，回溯时实时更新剩余容量。每个点可以流向多条出边。时间复杂度为  $O(n^2m)$ 。

```

1 namespace Dinic {
2     struct Edge {
3         int v, next, c;
4         Edge(int v = 0, int c = 0, int nxt = 0): v(v), next(nxt), c(c) {}
5     } e[maxm << 1];
6
7     int head[maxm], cur[maxm], d[maxm], q[maxm], s, t, cnt = 2, first, tail;
8     void add_edge(int u, int v, int c) {
9         e[cnt] = Edge(v, c, head[u]);
10        head[u] = cnt++;
11        e[cnt] = Edge(u, 0, head[v]);
12        head[v] = cnt++;
13    }
14    bool bfs() {
15        for (int i = 0; i <= t; i++)
16            d[i] = 0, cur[i] = head[i];
17        first = tail = 0;
```

```

18     q[tail++] = s, d[s] = 1;
19     while (first < tail) {
20         int cur = q[first++];
21         for (int i = head[cur]; i; i = e[i].next) {
22             if (e[i].c && !d[e[i].v]) {
23                 q[tail++] = e[i].v;
24                 d[e[i].v] = d[cur] + 1;
25                 if (e[i].v == t)
26                     return true;
27             }
28         }
29     }
30     return false;
31 }
32 int dinic(int x, int flow) {
33     if (x == t)
34         return flow;
35     int rest = flow, k;
36     for (int i = cur[x]; i && rest; i = e[i].next) {
37         cur[x] = i;
38         if (e[i].c && d[e[i].v] == d[x] + 1) {
39             k = dinic(e[i].v, min(rest, e[i].c));
40             if (!k)
41                 d[e[i].v] = 0;
42             e[i].c -= k, e[i ^ 1].c += k;
43             rest -= k;
44         }
45     }
46     return flow - rest;
47 }
48 int work() {
49     int maxflow = 0, flow = 0;
50     while (bfs())
51         while (flow = dinic(s, inf))
52             maxflow += flow;
53     return maxflow;
54 }
55 }
```

### 3.3 最小费用最大流

给定一个网络  $G = (V, E)$ , 每条边  $(u, v) \in E$  带有属性  $c(u, v)$ , 表示从当前边流过 1 个单位流量时的费用。在最大化流量的基础上, 求最小的费用。

```
1 namespace MCMF {
2     int head[maxn], flow[maxn], cost[maxn], pre[maxn], from[maxn], ecnt, s, t;
3     bool vis[maxn];
4     struct Edge {
5         int u, v, f, c, next;
6         Edge(int u = 0, int v = 0, int f = 0, int c = 0, int next = 0)
7             : u(u), v(v), f(f), c(c), next(next) {}
8     } e[maxn];
9     void init() {
10        memset(head, 0, sizeof head);
11        memset(pre, 0, sizeof pre);
12        ecnt = 2;
13    }
14    void add_edge(int u, int v, int f, int c) {
15        e[ecnt] = Edge(u, v, f, c, head[u]);
16        head[u] = ecnt++;
17    }
18    bool spfa() {
19        memset(vis, 0, sizeof vis);
20        memset(flow, 0x3f, sizeof flow);
21        memset(cost, 0x3f, sizeof cost);
22        queue<int> q;
23        q.push(s), vis[s] = 1, cost[s] = 0, pre[t] = -1;
24        while (q.size()) {
25            int cur = q.front();
26            q.pop(), vis[cur] = 0;
27            for (int i = head[cur], v; i; i = e[i].next) {
28                v = e[i].v;
29                if (e[i].f > 0 && cost[v] > cost[cur] + e[i].c) {
30                    cost[v] = cost[cur] + e[i].c;
31                    pre[v] = cur, from[v] = i;
32                    flow[v] = min(flow[cur], e[i].f);
33                    if (!vis[v])
34                        vis[v] = 1, q.push(v);
35                }
36            }
37        }
38        return pre[t] != -1;
39    }
40    pair<int, int> solve(int _s, int _t) {
41        s = _s, t = _t;
42        int maxflow = 0, mincost = 0;
43        while (spfa()) {
```

```

44     int cur = t;
45     maxflow += flow[t], mincost += flow[t] * cost[t];
46     while (cur != s) {
47         e[from[cur]].f -= flow[t];
48         e[from[cur] ^ 1].f += flow[t];
49         cur = pre[cur];
50     }
51 }
52 return make_pair(maxflow, mincost);
53 }
54 }
```

## 3.4 带花树算法

**问题场景** 用于解决一般图最大匹配问题。

```

1 namespace Blossom {
2     int head[MAXN], match[MAXN], fa[MAXN], pre[MAXN], top[MAXN], in[MAXN];
3     int q[MAXN * MAXN * 2], first, tail, lcatime = 0, ecnt = 1;
4     bool odd[MAXN], vis[MAXN];
5     Edge e[MAXN * MAXN * 2];
6
7     void add_edge(int u, int v) {
8         e[ecnt] = Edge(u, v, head[u]), head[u] = ecnt++;
9     }
10
11    int find(int x) {
12        return fa[x] == x ? x : fa[x] = find(fa[x]);
13    }
14
15    int lca(int x, int y) {
16        lcatime++;
17        while (x)
18            in[x] = lcatime, x = find(top[x]);
19        x = y;
20        while (in[x] != lcatime)
21            x = find(top[x]);
22        return x;
23    }
24
25    void blossom(int x, int y, int k) {
26        while (find(x) != k) {
27            pre[x] = y, y = match[x];
28            odd[y] && (odd[y] = 0, q[tail++] = y);
29            find(x) == x && (fa[x] = k);
```

```

30         find(y) == y && (fa[y] = k);
31         x = pre[y];
32     }
33 }
34
35 bool augment(int s) {
36     for (int i = 0; i <= n; i++)
37         fa[i] = i, top[i] = pre[i] = odd[i] = vis[i] = 0;
38     vis[s] = 1, first = tail = 0, q[tail++] = s;
39     while (first < tail) {
40         int now = q[first++];
41         for (int i = head[now], v, x, y, j, k; i; i = e[i].next) {
42             v = e[i].v;
43             if (!vis[v]) {
44                 top[v] = now, pre[v] = now,
45                 odd[v] = 1, vis[v] = 1;
46                 if (!match[v]) {
47                     j = v;
48                     while (j)
49                         x = pre[j], y = match[x], match[j] = x, match[x] = j, j
50                         = y;
51                     return true;
52                 }
53                 vis[match[v]] = 1, top[match[v]] = v, q[tail++] = match[v];
54             }
55             else if (find(now) != find(v) && !odd[v])
56                 k = lca(now, v), blossom(now, v, k), blossom(v, now, k);
57         }
58     }
59     return false;
60 }
61
62 int solve() {
63     int ans = 0;
64     for (int i = 1; i <= n; i++)
65         if (!match[i])
66             ans += augment(i);
67     printf("%d\n", ans);
68     for (int i = 1; i <= n; i++)
69         printf("%d ", match[i]);
70     return ans;
71 }

```

## 3.5 KM 算法

问题场景 解决二分图最大带权匹配，时间复杂度  $O(n^3)$ .

```
1 template <typename T> struct KM { // km
2     int n, org_n, org_m;
3     vector<int> matchx, matchy; // 左/右集合对应的匹配点
4     vector<int> pre; // 连接右集合的左点
5     vector<bool> visx, visy; // 拜访数组 左/右
6     vector<T> lx, ly, slack;
7     vector<vector<T>> g;
8     queue<int> q;
9     T inf, res;
10
11    KM(int _n, int _m) {
12        org_n = _n, org_m = _m, n = max(_n, _m);
13        inf = numeric_limits<T>::max(), res = 0;
14        g = vector<vector<T>>(n, vector<T>(n));
15        for (int i = 0; i < _n; i++)
16            for (int j = 0; j < _m; j++)
17                g[i][j] = -inf;
18        matchx = vector<int>(n, -1), matchy = vector<int>(n, -1);
19        pre = vector<int>(n);
20        visx = vector<bool>(n), visy = vector<bool>(n);
21        lx = vector<T>(n, -inf), ly = vector<T>(n);
22        slack = vector<T>(n);
23    }
24
25    void addEdge(int u, int v, int w) {
26        g[u][v] = max(w, 0); // 负值还不如不匹配 因此设为 0 不影响
27    }
28
29    bool check(int v) {
30        visy[v] = true;
31        if (matchy[v] != -1) {
32            q.push(matchy[v]), visx[matchy[v]] = true; // in S
33            return false;
34        }
35        // 找到新的未匹配点 更新匹配点 pre 数组记录着"非匹配边"上与之相连的点
36        while (v != -1)
37            matchy[v] = pre[v], swap(v, matchx[pre[v]]);
38        return true;
39    }
40
41    void bfs(int i) {
42        while (!q.empty())
43            q.pop();
44        q.push(i);
```

```

45 visx[i] = true;
46 while (true) {
47     while (!q.empty()) {
48         int u = q.front();
49         q.pop();
50         for (int v = 0; v < n; v++)
51             if (!visy[v]) {
52                 T delta = lx[u] + ly[v] - g[u][v];
53                 if (slack[v] >= delta) {
54                     pre[v] = u;
55                     if (delta)
56                         slack[v] = delta;
57                     else if (check(v))
58                         return;
59                 }
60             }
61     }
62     // 没有增广路 修改顶标
63     T a = inf;
64     for (int j = 0; j < n; j++)
65         if (!visy[j])
66             a = min(a, slack[j]);
67     for (int j = 0; j < n; j++) {
68         if (visx[j]) // S
69             lx[j] -= a;
70         if (visy[j]) // T
71             ly[j] += a;
72         else // T'
73             slack[j] -= a;
74     }
75     for (int j = 0; j < n; j++)
76         if (!visy[j] && slack[j] == 0 && check(j))
77             return;
78 }
79 }

80
81 ll solve() {
82     for (int i = 0; i < n; i++)
83         for (int j = 0; j < n; j++)
84             lx[i] = max(lx[i], g[i][j]);    // 初始顶标
85
86     for (int i = 0; i < n; i++) {
87         fill(slack.begin(), slack.end(), inf);
88         fill(visx.begin(), visx.end(), false);
89         fill(visy.begin(), visy.end(), false);
90         bfs(i);
91     }
92 }

```

```
93     // custom
94     for (int i = 0; i < n; i++) {
95         if (g[i][matchx[i]] > 0)
96             res += g[i][matchx[i]];
97         else
98             matchx[i] = -1;
99     }
100    return res;
101 }
102 };
```

# Chapter 4

## 字符串和回文算法

### 4.1 字典树 (Trie)

```
1 class Trie {
2     public:
3         int node[MAXN][30];
4         bool has[MAXN], vis[MAXN];
5         int cnt;
6         void init() {
7             memset(node, 0, sizeof node);
8             memset(has, 0, sizeof has);
9             memset(vis, 0, sizeof vis);
10            cnt = 1;
11        }
12        int idx(char s) {
13            return s - 'a';
14        }
15        void insert(char *s, int n) {
16            int u = 0;
17            for (int i = 0; i < n; i++) {
18                int c = idx(s[i]);
19                if (!node[u][c])
20                    node[u][c] = ++cnt;
21                u = node[u][c];
22            }
23            has[u] = 1;
24        }
25        int query(char *s, int n) {
26            int u = 0;
27            for (int i = 0; i < n; i++) {
28                int c = idx(s[i]);
29                if (!node[u][c])
30                    return -1;
31                u = node[u][c];
```

```

32     }
33     if (!has[u])
34         return -1;
35     if (!vis[u]) {
36         vis[u] = 1;
37         return 0;
38     } else
39         return 1;
40 }
41 };

```

对于 0-1 字典树：

- 找异或最大值：当前位是 1 就走 0，是 0 就走 1；走不通再走另一个；
- 找与/或的最大值：以与运算为例，如果当前位是 1，那么肯定优先走 1；如果当前位是 0，那么当前位和 0 或 1 运算的结果都是 0，我们无法确定走哪条支路才是最优解。于是我们可以将两条路合并成一条，把 1 的树自底向上合并到 0 的树

## 4.2 KMP 算法

```

1 namespace KMP {
2     int next[MAXN];
3     void calcNextArr() {
4         next[0] = -1;
5         int j = 0, k = -1, len = strlen(pattern) - 1;
6         while (j < len) {
7             if (k == -1 || pattern[j] == pattern[k]) {
8                 next[++j] = ++k;
9                 if (pattern[j] == pattern[k])
10                     next[j] = next[k];
11             } else
12                 k = next[k];
13         }
14     }
15
16     int KMP() {
17         int i = 0, j = 0;
18         int tlen = strlen(target), plen = strlen(pattern);
19         while (i < tlen && j < plen)
20             if (j == -1 || target[i] == pattern[j])
21                 i++, j++;
22             else
23                 j = next[j];
24             if (j == plen)
25                 return i - j;

```

```

26     else
27         return -1;
28     }
29 }
```

**KMP 求最小循环节** 假设  $S$  的长度为  $len$ , 则  $S$  存在最小循环节, 循环节的长度  $L$  为  $len - next[len]$ , 子串为  $S[0...len - next[len] - 1]$ 。

- 如果  $len$  可以被  $len - next[len]$  整除, 则表明字符串  $S$  可以完全由循环节循环组成, 循环周期  $T = len/L$ 。
- 如果不能, 说明还需要再添加几个字母才能补全。需要补的个数是循环个数  $L - len \bmod L = L - (len - L) \bmod L = L - next[len] \bmod L$   $L = len - next[len]$ 。

### 4.3 扩展 KMP (Z 函数)

长度为  $n$  的字符串  $s$ , 它的  $Z$  函数定义为:  $Z[i] =$  从位置  $i$  开始且为  $s$  的前缀的字符串的最大长度, 即:

$$\begin{cases} z[0] = 0 \\ z[i] = \max(l), s[0...l-1] = s[i, i+l-1] \end{cases}$$

应用:

- 求  $s$  的每个后缀与  $t$  的最长公共前缀: 构造  $r = t + s$  后计算  $Z$  函数即可。
- 在  $t$  中查找模式串  $p$  的所有出现次数: 构造  $s = p + \$ + t$  后计算  $s$  的  $Z$  函数, 对于区间  $[0, |t| - 1]$  的任意  $i$  考虑  $k = z[i + |p| + 1]$ , 如果  $k = |p|$  则有一个  $p$  出现在  $t[i]$  处。

```

1 namespace ZFunction {
2     int z[maxn << 1];
3     int* calc(char s[]) {
4         int len = strlen(s);
5         for (int i = 1, l = 0, r = 0; i < len; i++) {
6             if (i <= r)
7                 z[i] = min(r - i + 1, z[i-1]);
8             while (i + z[i] < len && s[z[i]] == s[i + z[i]])
9                 z[i]++;
10            if (i + z[i] - 1 > r)
11                l = i, r = i + z[i] - 1;
12        }
13        return z;
14    }
15 }
```

## 4.4 Manacher 算法

- $r[i]$ : 以  $i$  为回文中心的回文串半径
- $pre[i]$ : 以  $i$  为起点的回文串数量
- $suf[i]$ : 以  $i$  为终点的回文串数量

```
1 namespace Manacher {
2     int r[maxn << 1], pre[maxn << 1], suf[maxn << 1], l, len;
3     char str[maxn << 1];
4
5     int init(char *s) {
6         str[0] = '$', str[1] = '#', len = 2, l = strlen(s);
7         for (int i = 0; i < l; i++)
8             str[len++] = s[i], str[len++] = '#';
9         str[len] = 0;
10        return len;           // 返回构造的字符串长度
11    }
12
13    int solve() {
14        int ans = -1, id = 0, mx = 0;
15        for (int i = 1; i < len; i++) {
16            r[i] = (i < mx) ? min(r[2 * id - i], mx - i) : 1;
17            while (str[i - r[i]] == str[i + r[i]])
18                r[i]++;
19            if (mx < i + r[i])
20                mx = i + r[i], id = i;
21            ans = max(ans, r[i] - 1);
22        }
23        return ans;           // 返回最长回文半径
24    }
25
26    void calc() {
27        for (int i = l * 2, x; i >= 2; i--)
28            x = i / 2, pre[x]++, pre[x - (r[i] / 2)]--;
29        for (int i = l; i >= 1; i--)
30            pre[i] += pre[i+1];
31        for (int i = 2, x; i <= l * 2; i++)
32            x = (i + 1) / 2, suf[x]++, suf[x + (r[i] / 2)]--;
33        for (int i = 1; i <= l; i++)
34            suf[i] += suf[i-1];
35    }
36 }
```

## 4.5 AC 自动机

在 Trie 树的基础上，为节点增加 fail 指针；当前节点失配的时候，将匹配指针转移到 fail 指针指向的节点。

### 建树

- 根节点指向的所有节点的 *fail* 指针都指向根节点
- 不存在的节点，*fail* 指针指向根节点
- 普通节点，字符为 *s* 的 *fail* 指针，指向它的父节点的 *fail* 指针指向节点 *fail[p]* 沿 *s* 走到的节点。

### 匹配

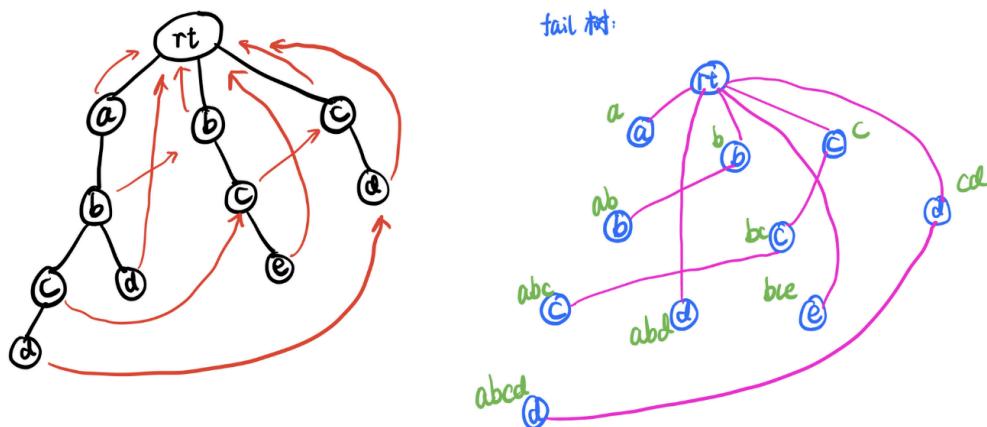
- 如果走到了不存在的节点，则将匹配指针移到 *fail* 指针指向的节点
- 从根节点开始匹配，原理与 Trie 树相同，匹配指针沿着 *p[i]* 所在的字母向下走
- 如果失配，则沿着 *fail* 指针移动，若匹配上则继续匹配，否则不断沿着 *fail* 指针走。

### Fail 指针

- 每个节点 *s* 有一个失配指针 *p*，所有的 *s* 和它们的 *p* 构成的树形结构称为 fail 树。
- fail 树上每个节点所代表的字符串，是其所有子树所代表的字符串的后缀。一个节点所有祖先代表的字符串都是这个节点代表的字符串的后缀，如下图所示。
- 重要性质：每个节点的 *fail* 指针，都指向当前节点代表的字符串的最长后缀（如果存在）。

### Fail 指针的用法

- 统计每个模式串 *p* 在文本串 *t* 当中出现的次数：将 *t* 在 AC 自动机的上匹配同时建立 fail 树，当经过某个节点时，对答案的贡献为：这个节点所有祖先的权值之和。利用树上差分将经过的所有节点计数 + 1。
- 一个模式串 *p<sub>i</sub>* 在其它模式串中出现的次数统计：如果 *p<sub>i</sub>* 在其它的模式串中出现，那么其它模式串的链上一定有一个节点的 *fail* 指针指向该节点，直接统计该节点在 fail 树上的子节点个数即可。



```

1  namespace AhoCorasickAutomaton {
2      #define clear(x) memset(x, 0, sizeof x)
3      int trie[maxn][26], has[maxn], fail[maxn], cnt;
4      int sz[maxn], pos[maxn], q[maxn];
5      void init() {
6          clear(trie), clear(has), clear(fail), cnt = 0;
7      }
8      inline int idx(char s) {
9          return s - 'a';
10     }
11     int insert(char* s, int l) {
12         int u = 0;
13         for (int i = 0, c; i < l; i++) {
14             c = idx(s[i]);
15             if (!trie[u][c])
16                 trie[u][c] = ++cnt;
17             u = trie[u][c];
18             sz[u]++;
19         }
20         has[u]++;
21         return u;
22     }
23     void get_fail() {
24         int head = 0, tail = 0;
25         for (int i = 0; i < 26; i++)
26             if (trie[0][i])
27                 fail[trie[0][i]] = 0, q[tail++] = trie[0][i];
28         while (head < tail) {
29             int u = q[head++];
30             for (int i = 0; i < 26; i++)
31                 if (trie[u][i])
32                     fail[trie[u][i]] = trie[fail[u]][i],
33                     q[tail++] = trie[u][i];
34                 else
35                     trie[u][i] = trie[fail[u]][i];
36         }
37     }
38     void solve() {
39         for (int i = cnt; i >= 0; i--)
40             sz[fail[q[i]]] += sz[q[i]];
41         for (int i = 0; i < n; i++)
42             printf("%d\n", sz[pos[i]]);
43     }
44 }
```

## 4.6 后缀数组

### 定义

- $sa[i]$ : 排名为  $i$  的后缀的起始下标 (排第  $i$  的是哪个后缀)
- $rank[i]$ : 起始下标为  $i$  的后缀  $suffix(i \dots len - 1)$  的排名 (第  $i$  个后缀排第几), 满足  $sa[rank[i]] = i, rank[sa[i]] = i$ .
- $height[i]$ :  $sa[i]$  与  $sa[i - 1]$  (排名相邻的两个后缀) 的最长公共前缀。
- $h[i] = height[rank[i]]$ :  $suffix(i)$  和在他前一名的后缀的最长公共前缀。满足  $h[i] \geq h[i - 1] - 1$ .

```
1  namespace SuffixArray {
2      const int MAXN = 100050, MAXM = 50, MAXLOG = 30;
3
4      char s[MAXN];
5      int n, m;
6      int sa[MAXN], rank[MAXN], height[MAXN], h[MAXN];
7      int t1[MAXN], t2[MAXN], ws[MAXN], wv[MAXN];
8      int log2[MAXN], minh[MAXN][MAXLOG];
9
10     // 计算后缀数组和排名数组
11     void build() {
12         register int i, k, p, *x = t1, *y = t2;
13         n = strlen(s) + 1;           // n 要多出 1
14         for (i = 0; i < n - 1; i++)
15             s[i] -= '0', m = max(m, (int)s[i] + 1);
16         s[n] = 0;                   // important!
17         for (i = 0; i < m; i++)
18             ws[i] = 0;
19         for (i = 0; i < n; i++)
20             ws[x[i]] = s[i]++;
21         for (i = 1; i < m; i++)
22             ws[i] += ws[i-1];
23         for (i = n - 1; i >= 0; i--)
24             sa[--ws[x[i]]] = i;
25         for (k = 1, p = 1; p < n; k <= 1, m = p) {
26             for (p = 0, i = n - k; i < n; i++)
27                 y[p++] = i;
28             for (i = 0; i < n; i++)
29                 if (sa[i] >= k)
30                     y[p++] = sa[i] - k;
31             for (i = 0; i < n; i++)
32                 wv[i] = x[y[i]];
33             for (i = 0; i < m; i++)
34                 ws[i] = 0;
35             for (i = 0; i < n; i++)
```

```

36         ws[wv[i]]++;
37     for (i = 1; i < m; i++)
38         ws[i] += ws[i-1];
39     for (i = n - 1; i >= 0; i--)
40         sa[--ws[wv[i]]] = y[i];
41     swap(x, y);
42     for (p = 1, x[sa[0]] = 0, i = 1; i < n; i++)
43         x[sa[i]] = (y[sa[i-1]] == y[sa[i]] && y[sa[i-1] + k] == y[sa[i] +
44             ↳ k]) ? p-1 : p++;
45     if (p >= n)
46         break;
47 }
48 // 最终结果 sa[i] 和 rank[i] 分别都从 1 开始, 需要将值或下标 -1
49 for (i = n; i > 0; i--)
50     sa[i] = sa[i-1] + 1;
51 for (i = 1; i <= n; i++)
52     rank[sa[i]] = i;
53 }
54 // 计算 height[] 数组
55 void calcHeight() {
56     register int i, j, k = 0;
57     for (i = 1; i <= n; i++) {
58         k = k ? k - 1 : 0;
59         j = sa[rank[i] - 1];
60         while (s[i+k-1] == s[j+k-1])
61             k++;
62         height[rank[i]] = k;
63     }
64 // 预处理 log2 的值
65 void preprocess() {
66     lg2[0] = -1;
67     for (int i = 1; i < MAXN; i++)
68         lg2[i] = (i & (i - 1)) ? lg2[i-1] : lg2[i-1] + 1;
69 }
70 // 预处理 ST 表
71 void initST() {
72     for (int i = 1; i <= n; i++)
73         minh[0][i] = height[i];
74     for (int i = 1; i <= lg2[n]; i++) {
75         int lim = n - (1 << i) + 1;
76         for (int j = 1; j <= lim; j++)
77             minh[i][j] = min(minh[i-1][j], minh[i-1][j + (1 << i >> 1)]);
78     }
79 }
80 // 询问 suffix(a) 和 suffix(b) 的最长公共前缀
81 int lcp(int a, int b) {
82     a = rank[a + 1], b = rank[b + 1];

```

```

83     if (a > b)
84         swap(a, b);
85     a++;
86     int t = lg2[b - a + 1];
87     return min(minh[t][a], minh[t][b - (1 << t) + 1]);
88 }
89 }
```

## 应用

- 求某两个后缀的最长公共前缀: $suffix(j)$  和  $suffix(k)$  的最长公共前缀为:  $\min\{height[rank[j]+1], height[rank[j]+2], \dots, height[rank[k]]\}$ ; 这是一个 RMQ 问题, 可以用 ST 表  $O(nlogn)$  预处理,  $O(1)$  查询。
- 可重叠最长重复子串: 子串一定是某个后缀的前缀; 可重叠的子串则等价于求两个后缀的最长公共前缀, 所以求  $height[]$  数组的最大值即可。
- 不可重叠最长重复子串: 首先在  $[1 \dots max(height)]$  区间二分答案  $k$ , 判断是否存在两个长度为  $k$  的子串是相同且不重叠的。利用  $height$  值对后缀分组, 使得每组的  $height$  值都不小于  $k$  (如果不存在就单独分组)。有希望成为最长公共前缀不小于  $k$  的两个后缀一定在同一组。然后对于每组后缀, 只须判断每个后缀的  $sa$  值的最大值和最小值之差是否不小于  $k$ 。
- 可重叠并出现  $k$  次的最长重复子串: 同样先二分长度答案  $x$  分成若干组, 判断的是有没有一个组的后缀个数不小于  $k$ . 如果存在则满足条件。
- 不相同的子串个数: 因为子串一定是某个后缀的前缀, 问题等价于求所有后缀之间不相同的前缀的个数, 如果所有的后缀按照  $suffix(sa[i])$  (排名) 的顺序计算, 对于新加入的后缀  $suffix(sa[k])$ , 将贡献  $n - sa[k] + 1 - height[k]$  个不同的子串, 累加即可。
- 最长回文子串: 将整个字符串反过来接在原字符的后面, 中间用一个特殊的字符隔开, 求新字符串某两个后缀的最长公共前缀即可。
- 连续重复子串: (定义: 字符串  $L$  由某个字符串  $S$  重复  $r$  次得到, 则  $L$  称为连续重复子串)
- 求  $r$  的最大值: 枚举字符串  $S$  的长度  $k$ , 然后判断是否满足。判断时先看  $L$  的长度能否被  $k$  整除, 再看  $suffix(1)$  和  $suffix(k+1)$  的最长公共前缀是否等于  $n - k$ , 如果是则合法。
- 重复次数最多的连续重复子串: 首先预处理 LCP; 枚举重复部分的长度  $l$ , 然后枚举每一个起始位置  $i$ 。如果重复部分出现大于等于 2 次, 那么一定会有  $s[0], s[l], s[2*l] \dots s[k*l]$  其中两个连续出现在重复组成的串中。所以对于确定的长度,  $O(1)$  查询  $s[i]$  和  $s[i+l]$  的公共前缀, 然后向前向后匹配, 重复串长度即为  $lcp(i, i+l) + (l - k \% l)$ , 然后再检查一下起点  $t = i - l + k \% l$  是否溢出, 以及  $lcp(t, t+l)$  的长度是否大于  $k$  即可更新答案。时间复杂度为  $O(nlogn)$ .
- $A, B$  的最长公共子串: 首先把  $B$  连接到  $A$  的末尾, 两者中间用一个没出现过的字符 (如 \$ 隔开), 然后求新串的后缀数组、height 数组等。然后遍历 height 数组, 当  $suffix(sa[i])$

和  $suffix(sa[i - 1])$  不是同一个字符串中的两个后缀时，最大的  $height[i]$  就满足条件。判断  $suffix(sa[i])$  和  $suffix(sa[i - 1])$  是否为同一个字符串中的两个后缀，只需判断下标的位置即可。

- $A, B$  的长度不小于  $k$  的公共子串个数（可以相同）：思路是计算  $A, B$  的所有后缀之间的 lcp 的长度，统计  $lcp \geq k$  的答案。首先把  $B$  连接到  $A$  的末尾，两者中间用一个没出现过的字符（如 \$ 隔开），计算 height 后按 height 值大于等于  $k$  分组。然后将后缀扫描一遍，遇到一个  $B$  的后缀就统计与前面的  $A$  的后缀能产生多少个长度不小于  $k$  的公共子串。 $A$  的后缀可以用单调栈维护；然后将  $A$  用相同的方法统计一次。
- 给定  $n$  个字符串，求出现在不小于  $k$  个字符串中的最长子串：将  $n$  个字符串连接，中间用没出现过的字符隔开，求后缀数组。然后二分长度  $l$  并利用 height 数组分组，判断每组的后缀是否出现在不小于  $k$  个原串中。
- 给定  $n$  个字符串，求在每个字符串中至少出现两次且不重叠的最长子串：和上面一样，判断的时候，要看是否有一组后缀在每个原来的字符串中至少出现两次，并且在每个原来的字符串中，后缀的起始位置的最大值与最小值之差是否不小于当前答案（判断能否做到不重叠，如果题目中没有不重叠的要求，那么不用做此判断）。
- 给定  $n$  个字符串，求出现或反转后出现在每个字符串中的最长子串：将每个字符串反转后的结果也拼到总串中，求后缀数组。判断的时候，要看是否有一组后缀在每个原来的字符串或反转后的字符串中出现。

## 4.7 后缀自动机

处理与子串相关问题的在线线性算法。

### 名词解释

- $len$  表示从当前  $endpos$  可向前延伸的长度最大值。设当前构造的 SAM 已得到的子串为  $s$ , 从当前字符向前数  $[0, len]$  个字符得到的新子串  $t$ ,  $t$  一定只作为  $s$  的后缀出现。
- 如: 对于母串  $abcdabcdbcd$ , 有三个子串  $d, cd, bcd$  的  $endpos$  集合相同, 而从  $endpos$  向前最多可以延伸 3 个字符满足条件, 所以  $len = 3$ .
- $link$  表示后缀连接。定义一个子串  $v \in endpos(v)$  ( $endpos(v)$  称为  $v$  的  $endpos$  等价类), 该等价类中长度最长的子串为  $w$ , 则  $w$  的“最长的且不在该  $endpos$  等价类中的后缀”记为  $t$ , 令  $link(v) = t$ .
- 如: 对于母串  $abcdabcdcd, abcd, bcd$  在同一个  $endpos$  等价类中。设  $v = bcd$ , 则显然  $w = abcd$ ,  $w$  的最长的且不再该等价类中的后缀显然为  $cd$ , 则令  $link(bcd) = cd$ . Tips: 后缀也有长度为 0 的情况, 即空后缀。
- $nxt[i]$  就是沿着字符 ' $a' + i$  走可以到达的下一个状态。

### 性质

- 字符串  $s$  的一个后缀自动机包含关于字符串  $s$  的所有子串的信息。任意从初始状态  $t_0$  开始的路径, 如果我们将转移路径上的标号写下来, 都会形成  $s$  的一个子串。反之每个  $s$  的子串对应从初始状态  $t_0$  开始的某条路径。
- 每个状态  $s$  代表的子串是区间  $[len_{link(s)} + 1, len_s]$ .
- 一个长度为  $n$  的字符串, 它的 SAM 节点个数最多有  $2n - 1$  个, 连边最多有  $3n - 4$  条。
- 树形结构的性质: 设字符串长度为  $n$ , 考虑 extend 操作中  $cur$  变量的值 (代表当前状态在节点池中的下标), 该节点对应的状态是: 执行 extend 操作时的当前字符串, 得到的  $n$  个节点对应了  $n$  个不同的终点, 第  $i$  个状态对应  $S_{1...i}$ 。如果我们将 SAM 看作一棵树, 树根为 0 号节点 (初始状态), 其余节点  $v$  满足其父亲为该节点的后缀连接  $link(v)$ 。这棵树叫  $parent$  树。
- $parent$  树中的每个节点的终点集合, 等于其子树内所有终点节点对应的终点集合的并集。
- $parent$  树中, 如果节点  $a$  是  $b$  的祖先, 则节点  $a$  对应的字符串是节点  $b$  对应的字符串的后缀。
- 构成的  $parent$  树存在一些与树相关的性质, 如  $S_{1...p}$  和  $S_{1...q}$  的最长公共后缀对应的是  $p, q$  对应节点间的 LCA 的字符串。
- 除了初始状态 (节点 0) 以外, 每个状态  $i$  对应的字串数量是  $len(i) - len(link(i))$ , 因此计算时可以自上而下计算。

```

1  class SuffixAutomaton {
2  private:
3      static const int MAXN = 100050;
4      static const int MAXLOG = 25;
5  public:
6      // 需要维护 right 集合时，加上以下动态开点线段树的代码
7      struct Node {
8          int val, lson, rson;
9      } a[MAXN * 50];
10     int tot;           // 权值线段树节点个数
11
12     void pushUp(int rt) {
13         a[rt].val = a[a[rt].lson].val + a[a[rt].rson].val;
14     }
15     void update(int &rt, int l, int r, int pos) {
16         if (!rt)
17             rt = ++tot, a[rt].lson = a[rt].rson = a[rt].val = 0;
18         if (l == r) {
19             a[rt].val++;
20             return;
21         }
22         int mid = (l + r) >> 1;
23         if (pos <= mid)
24             update(a[rt].lson, l, mid, pos);
25         else
26             update(a[rt].rson, mid + 1, r, pos);
27         pushUp(rt);
28     }
29     int merge(int x, int y) {
30         if (!x || !y)
31             return x + y;
32         int z = ++tot;
33         a[z].lson = merge(a[x].lson, a[y].lson);
34         a[z].rson = merge(a[x].rson, a[y].rson);
35         pushUp(z);
36         return z;
37     }
38     int query(int rt, int l, int r, int k) {
39         if (!rt)
40             return 0;
41         if (l == r)
42             return l;
43         int mid = (l + r) >> 1;
44         if (a[a[rt].lson].val >= k)
45             return query(a[rt].lson, l, mid, k);
46         if (a[a[rt].rson].val >= k - a[a[rt].lson].val)
47             return query(a[rt].rson, mid + 1, r, k - a[a[rt].lson].val);
48         return -1;

```

```

49 }
50
51 /* 后缀自动机 */
52 struct State {
53     int len, link, nxt[26];
54     State(int _len = 0, int _link = 0): len(_len), link(_link) {
55         memset(nxt, 0, sizeof nxt);
56     }
57 } st[MAXN << 1]; // 最多有 2n-1 个节点, 开两倍空间
58
59 // sz: 状态个数, last: 上一次插入的字符对应状态, cnt: 当前产生子串个数, len 字符串
60 // → 长度
61 // ws wv 基数排序数组, endpos[i] 表示 i 状态所代表的 endpos 集合线段树的树根
62 int sz, last, cnt, len;
63 int endpos[MAXN << 1], ws[MAXN << 1], wv[MAXN << 1], pos[MAXN << 1];
64 int f[MAXN << 1][MAXLOG];
65
66 int extend(int c, int idx) {
67     int cur = sz++, p = last;
68     st[cur] = State(st[last].len + 1);
69
70     endpos[cur] = 0;
71     update(endpos[cur], 1, len, idx + 1); // 更新当前点的 endpos, 注意权值线段
72     // → 树值域范围
73
74     while (p != -1 && !st[p].nxt[c])
75         st[p].nxt[c] = cur, p = st[p].link;
76     if (p == -1)
77         st[cur].link = 0;
78     else {
79         int q = st[p].nxt[c];
80         if (st[q].len == st[p].len + 1)
81             st[cur].link = q;
82         else {
83             int clone = sz++;
84             st[clone] = State(st[p].len + 1, st[q].link);
85             memcpy(st[clone].nxt, st[q].nxt, sizeof st[q].nxt);
86
87             endpos[clone] = 0; // 为克隆节点新建 endpos, 但不建树
88
89             while (p != -1 && st[p].nxt[c] == q)
90                 st[p].nxt[c] = clone, p = st[p].link;
91             st[q].link = st[cur].link = clone;
92         }
93     }
94     last = cur;
95     cnt += st[cur].link == -1 ? st[cur].len : st[cur].len -
96     → st[st[cur].link].len; // 字串个数

```

```

94     return cnt;
95 }
96
97 // 基于基数排序的拓扑排序，保证状态间的拓扑关系，即子状态在后，父状态在前
98 // 后缀自动机更新信息时，需要先更新子状态 s，再更新父状态 link[s]
99 void toposort() {
100     memset(ws, 0, sizeof ws);
101     memset(wv, 0, sizeof wv);
102     for (int i = 0; i < sz; i++)
103         wv[st[i].len]++;
104     for (int i = 1; i < sz; i++)
105         wv[i] += wv[i-1];
106     for (int i = 0; i < sz; i++)
107         ws[wv[st[i].len]--] = i;
108 }
109 // 建立后缀自动机
110 void build(char str[]) {
111     len = strlen(str);
112     for (rint i = 0; i < len; i++) {
113         extend(str[i] - 'a', i);
114         pos[i] = last;
115     }
116     // 预处理倍增表
117     for (rint i = 0; i < sz; i++)
118         f[i][0] = st[i].link;
119     for (rint j = 1; j < MAXLOG; j++)
120         for (rint i = 0; i < sz; i++)
121             f[i][j] = f[f[i][j-1]][j-1];
122     toposort();
123     // 如果需要维护 right 集合：从子节点开始，合并 endpos 集合
124     for (rint i = sz; i > 0; i--) {
125         int tmp = ws[i];
126         if (st[tmp].link != -1)
127             endpos[st[tmp].link] = merge(endpos[st[tmp].link], endpos[tmp]);
128     }
129 }
130
131 int solve(int l, int r, int k) {
132     // 首先从 endpos 为 r 的节点开始，倍增找到与目标串一样长的点
133     int p = pos[r], curlen = r - l + 1;
134     for (rint i = MAXLOG - 1; i >= 0; i--)
135         if (st[f[p][i]].len >= curlen)
136             p = f[p][i];
137     int ans = query(endpos[p], 1, len, k);
138
139     if (ans != -1)
140         ans -= (curlen - 1);
141     return ans;

```

```
142     }
143
144     void init() {
145         cnt = 0, tot = 0, sz = 0;
146         st[sz] = State(0, -1), last = sz++;
147         memset(a, 0, sizeof a);      // 清空权值线段树
148     }
149     SuffixAutomaton() { init(); }
150 } SAM;
```

## 4.8 广义后缀自动机

**问题场景** 对多个子串或给定的 Trie 树建立后缀自动机。

```
1  namespace exSAM {
2      #define clr(x) memset(x, 0, sizeof x);
3      const int maxn = 300050,
4                  maxm = maxn << 1,
5                  maxlog= 20,
6                  maxch = 26;
7      int pos[maxn], n;           // pos[i]: 第 i 个字符在 trie 树中的节点编号
8      char s[maxn];
9
10     namespace SAM {
11         struct Node {
12             // len: 状态最大长度, link: 后缀连接, nxt: 节点转移, cnt: 状态代表子串数
13             // → 量
14             int len, cnt, link, nxt[maxch];
15             Node(int _l = 0, int _c = 0, int _lk = 0) {
16                 len = _l, cnt = _c, link = _lk, clr(nxt);
17             }
18         } t[maxm];
19         int tot, c[maxm], rk[maxm], f[maxm][maxlog + 2]; // 拓扑计数和倍增用数组
20         vector<vector<int>> G;
21         void init() {
22             tot = 1;
23         }
24         // 拓展节点, id: 当前字符, last: 从哪一个状态转移来, cnt: Trie 树中的当前状态
25         // → 个数
26         int extend(int id, int last, int cnt) {
27             int cur = ++tot, p;
28             t[cur] = Node(t[last].len + 1, cnt);
29             for (p = last; p && !t[p].nxt[id]; p = t[p].link)
30                 t[p].nxt[id] = cur;
31             if (!p)
32                 t[cur].link = 1;
33             else {
34                 int q = t[p].nxt[id];
35                 if (t[q].len == t[p].len + 1)
36                     t[cur].link = q;
37                 else {
38                     int clone = ++tot;
39                     t[clone] = t[q], t[clone].cnt = 0, t[clone].len = t[p].len + 1;
40                     for (; p && t[p].nxt[id] == q; p = t[p].link)
41                         t[p].nxt[id] = clone;
42                     t[cur].link = t[q].link = clone;
43                 }
44             }
45         }
46     }
```

```

43     return cur;
44 }
45 // 处理每个状态的后缀链接倍增数组 & 递归统计
46 void dfs(int u) {
47     for (int i = 1; i < maxlog; i++)
48         f[u][i] = f[f[u][i-1]][i-1];
49     for (auto &v : G[u])
50         f[v][0] = u, dfs(v), t[u].cnt += t[v].cnt;
51 }
52 // 统计每个节点代表的状态个数
53 void topoSort() {
54     clr(c);
55     for (int i = 1; i <= tot; i++)
56         c[t[i].len]++;
57     for (int i = 1; i <= tot; i++)
58         c[i] += c[i-1];
59     // 如果不需要处理倍增父节点, 那么直接使用拓扑排序计算
60     /*
61     for (int i = 1; i <= tot; i++)
62         rk[c[t[i].len]--] = i;
63     for (int i = tot; i; i--)
64         t[t[rk[i]].link].cnt += t[rk[i]].cnt;
65     */
66
67     // 否则用 DFS 处理
68     G.clear(), G.resize(tot + 1);
69     for (int i = 1; i <= tot; i++)
70         if (t[i].link)
71             G[t[i].link].emplace_back(i);
72     f[1][0] = 0, dfs(1);
73 }
74 int query(int x, int len) {
75     for (int i = maxlog - 1; i >= 0; i--)
76         if (t[f[x][i]].len >= len)
77             x = f[x][i];
78     return t[x].cnt;
79 }
80 } // SAM
81
82 namespace Trie {
83     struct Node {
84         int nxt[maxch], cnt, spos; // sam_pos
85         Node(int _c = 0, int _sp = 0) {
86             cnt = _c, spos = _sp, clr(nxt);
87         }
88     } t[maxm];
89     int rt, tot;
90     vector<vector<int>> G;

```

```

91     void init() {
92         rt = tot = 1;
93     }
94     int add(int p, int ch) {
95         if (!t[p].nxt[ch])
96             t[p].nxt[ch] = ++tot;
97         int now = t[p].nxt[ch];
98         t[now].cnt++;
99         return now;
100    }
101    void dfs(int u) {
102        for (auto &v : G[u])
103            pos[v] = add(pos[u], s[v] - 'A'), dfs(v);
104    }
105    void bfs() {
106        queue<int> q;
107        q.push(1), t[1].spos = 1;
108        while (q.size()) {
109            int u = q.front();
110            q.pop();
111            for (int i = 0, now; i < maxch; i++) {
112                if (!t[u].nxt[i])
113                    continue;
114                now = t[u].nxt[i];
115                t[now].spos = SAM::extend(i, t[u].spos, t[now].cnt);
116                q.push(now);
117            }
118        }
119    }
120    void build() {
121        pos[0] = 1, pos[1] = add(pos[0], s[1] - 'A'); // 首先插入根节点和第一
122        // 个字符, 注意边界
123        // for (int i = 1; i <= n; i++) pos[i] = add(parent, s[i]-'A');
124        dfs(1), bfs(); // 使用 BFS 建立后缀数组
125    }
126}

```

## 4.9 回文树/回文自动机

### 功能

- 求前缀字符串  $s[0] \sim s[i]$  中的本质不同的回文串种类
- 求每个本质不同的回文串出现的次数，求回文串的个数
- 求以下标  $i$  的字符结尾的回文串个数/种类
- 每个本质不同的回文串，包含的本质不同回文串种类

```
1 class PalindromicTree {
2 public:
3     struct Node {
4         // len: 当前节点回文串长度; fail: 失配指针
5         // cnt: 当前节点表示的回文串在 s 中出现的次数
6         // num: 以节点 i 表示的回文串末尾字符结尾, 但不包含本条路径上的回文串的数目
7         // → (fail 指针路径深度)
8         int len, fail, cnt, num, next[26];
9         Node(int l = 0, int c = 0, int f = 0, int n = 0): len(l), fail(f), cnt(c),
10            → num(n) {
11                memset(next, 0, sizeof(next));
12            }
13        } t[MAXN];
14        int p, n, cur, last, s[MAXN];      // p: 节点数, n: 字符数
15
16        int alloc(int len, int fail) {
17            t[p] = Node(len, 0, fail);
18            return p++;
19        }
20
21        void init() {
22            p = n = cur = last = 0;
23            alloc(0, 1), alloc(-1, 0);      // len, fail
24            s[0] = -1;
25        }
26
27        int getFail(int x) {
28            while (s[n - t[x].len - 1] != s[n])
29                x = t[x].fail;
30            return x;
31        }
32
33        void add(int c) {
34            c = c >= 'a' ? c - 'a' : c;
35            s[++n] = c;
```

```

35     int cur = getFail(last);
36     if (!t[cur].next[c])
37         t[cur].next[c] = alloc(t[cur].len + 2,
38             ↳ t[getFail(t[cur].fail)].next[c]);
39     last = t[cur].next[c];
40     t[last].cnt++;
41 }
42 void count() {
43     for (int i = p - 1; i >= 0; i--)
44         t[t[i].fail].cnt += t[i].cnt;
45 }
46 } PT;

```

## 4.10 字符串哈希

```

1 const basePrime = 2333;
2
3 // 利用 unsigned long long 自然溢出, 统计的时候要用 Trie 或者 sort, unique 等
4 unsigned long long overflow_hash(char *s, int len) {
5     unsigned long long hash = 0;
6     for (int i = 0; i < len; i++)
7         hash = hash * basePrime + s[i];
8     return hash;
9 }
10
11 // 模两个质数的双哈希, 建议取一些比较不常见的质数, 比如
12    ↳ 122420729, 163227661, 217636919, 1222827239, 998244353 等
13 pair<int, int> double_hash(char *s, int len, int mod1, int mod2) {
14     int hash1 = 0, hash2 = 0;
15     for (int i = 0; i < len; i++)
16         hash1 = (1LL * hash * basePrime + s[i]) % mod1,
17         hash2 = (1LL * hash * basePrime + s[i]) % mod2;
18     return make_pair(hash1, hash2);
19 }

```

## 4.11 字符串循环同构的最小表示法

```

1 // 返回最小表示下起始位置的下标
2 int MinimumRepresentation(char *s, int l) {
3     int i = 0, j = 1, k = 0, t;

```

```

4     while(i < l && j < l && k < l) {
5         t = s[(i + k) >= l ? i + k - 1 : i + k] - s[(j + k) >= l ? j + k - 1 : j +
6             k];
7         if(!t) k++;
8         else{
9             if(t > 0) i = i + k + 1;
10            else j = j + k + 1;
11            if(i == j) ++ j;
12            k = 0;
13        }
14    }
15    return (i < j ? i : j);
}

```

## 4.12 Lyndon 分解

**Lyndon 串的定义** 对于字符串  $s$ , 如果  $s$  的字典序严格小于  $s$  的所有后缀的字典序, 我们称  $s$  是简单串, 或者 Lyndon 串。

**Lyndon 分解**  $s$  的 Lyndon 分解记为  $s = w_1w_2\dots w_k$ , 其中所有的  $w_i$  为简单串, 且字典序非单调递增, 即  $w_{i-1} \geq w_i$ 。

```

1 namespace Lyndon {
2     vector<string> duval(string s) {
3         int n = s.size(), i = 0;
4         vector<string> res;
5         while (i < n) {
6             int j = i + 1, k = i;
7             while (j < n && s[k] <= s[j]) {
8                 if (s[k] < s[j])
9                     k = i;
10                else
11                    k++;
12                j++;
13            }
14            while (i <= k)
15                res.emplace_back(s.substr(i, j - k)), i += j - k;
16        }
17        return res;
18    }
19 // 使用 Lyndon 分解求最小表示
20 string minimum_representation(string s) {
21     s += s;
22     int n = s.size();

```

```
23     int i = 0, ans = 0;
24     while (i < n / 2) {
25         ans = i;
26         int j = i + 1, k = i;
27         while (j < n && s[k] <= s[j]) {
28             if (s[k] < s[j])
29                 k = i;
30             else
31                 k++;
32             j++;
33         }
34         while (i <= k) i += j - k;
35     }
36     return s.substr(ans, n / 2);
37 }
38 }
```

# Chapter 5

## 数学专题

### 5.1 素数、欧拉函数、莫比乌斯函数

#### 5.1.1 线性筛

```
1  bool notPrime[MAXN];
2  int primes[MAXN], phi[MAXN], mu[MAXN], cnt = 0;           // phi(n) 欧拉函数
3
4  void sieve(int n) {
5      notPrime[0] = notPrime[1] = 1;
6      phi[1] = mu[1] = 1;
7      for (int i = 2; i <= n; i++) {
8          if (!notPrime[i]) {
9              primes[cnt++] = i;
10             phi[i] = i-1, mu[i] = -1;
11         }
12         for (int j = 0; j < cnt && i * primes[j] <= n; j++) {
13             notPrime[i * primes[j]] = 1;
14             if (i % primes[j] == 0) {
15                 phi[i * primes[j]] = phi[i] * primes[j];
16                 mu[i * primes[j]] = 0;
17                 break;
18             } else {
19                 phi[i * primes[j]] = phi[i] * (primes[j] - 1);
20                 mu[i * primes[j]] = -mu[i];
21             }
22         }
23     }
24 }
```

**线性筛计算因数个数** 设  $d(i)$  表示数  $i$  的约数个数,  $num(i)$  表示  $i$  的最小质因数的个数 (e.g.  $i = p_1^{a_1} \cdot p_2^{a_2} \cdot \dots \cdot p_n^{a_n}, p_1 < p_2 < \dots < p_n$ , 则  $num(i) = count(p_1) = a_1$ ).

- 若  $i$  为素数, 则  $\begin{cases} d(i) = 2 \\ num(i) = count(i) = 1 \end{cases}$ .
- 若  $i \% primes[j] \neq 0$ , 则  $d(i \cdot primes[j]) = d(i) \cdot d(primes[j])$ , 因为  $i \cdot primes[j]$  在当前被枚举前, 不包含因数  $primes[j]$ , 所以  $d(i \cdot primes[j]) = d(i) \cdot 2$ ; 又因为  $j$  是从小到大枚举的, 所以  $primes[j]$  从小到大, 故有  $num(i \cdot primes[j]) = 1$ .
- 若  $i \% primes[j] = 0$ , 则说明  $i$  在被枚举前, 已存在质因子  $primes[j]$ , 且  $primes[j]$  为  $i \cdot primes[j]$  的最小质因子, 所以  $d(i \cdot primes[j]) = \frac{d(i)}{num(i)+1} \cdot (num(i)+2)$ ,  $num(i \cdot primes[j]) = num(i) + 1$ .

**线性筛计算因数和** 令  $sumd(n) = (1 + p_1 + p_1^2 + \dots + p_1^{r_1})(1 + p_2 + p_2^2 + \dots + p_2^{r_2}) \dots (1 + p_k + p_k^2 + \dots + p_k^{r_k})$   
其中  $n = p_1^{r_1} p_2^{r_2} \dots p_k^{r_k}$ .

- 若  $i$  为素数,  $sund(i) = 1 + i$ ,  $num(i) = 1 + i$ .
- 若  $i \% primes[j] \neq 0$ ,  $\begin{cases} sund(i \cdot primes[j]) = sund(i) \cdot sumd(primes[j]) \\ num(i \cdot primes[j]) = 1 + primes[j] \end{cases}$
- 若  $i \% primes[j] = 0$ , 则  $\begin{cases} sund(i \cdot primes[j]) = (\frac{sund(i)}{num(i)}) \cdot (num(i) \cdot primes[j]) + 1 \\ num(i \cdot primes[j]) = num(i) \cdot primes[j] + 1 \end{cases}$

### 5.1.2 求单值的欧拉函数

**定义** 对正整数  $n$  的欧拉函数是: 小于等于  $n$  的数中, 与  $n$  互质的数的数目。

```

1 long long euler(long long n) {
2     long long res = n, a = n;
3     for (long long i = 2; i * i <= a; i++) {
4         if (a % i == 0) {
5             res = res / i * (i - 1);
6             while (a % i == 0)
7                 a /= i;
8         }
9     }
10    if (a > 1)
11        res = res / a * (a - 1);
12    return res;
13 }
```

## 5.2 杜教筛和积性函数

### 5.2.1 积性函数

**定义** 对于数论函数  $f(n)$ , 如果对于  $\forall x, y$  满足  $(x, y) = 1$ , 有  $f(xy) = f(x)f(y)$ , 则称  $f(n)$  是一个积性函数。

特别地, 如果不需要满足  $(x, y) = 1$  就有  $f(xy) = f(x)f(y)$ , 则  $f(n)$  是一个完全积性函数。

#### 常见积性函数

- 因数个数  $d(x) = \sum_{i|n} 1$
- 因数和  $\sigma(x) = \sum_{i|n} i$
- 欧拉函数  $\phi(x) = \sum_{i=1}^x [\gcd(x, i) = 1]$
- 莫比乌斯函数  $\mu(x)$
- 单位函数  $e(n) = [n = 1]$
- 常数函数  $I(n(= 1))$
- 恒等函数  $id(n) = n$

**性质** 若  $f(x), g(x)$  为积性函数, 则以下函数均为积性函数:

- $h(x) = f(x^p)$
- $h(x) = f^p(x)$
- $h(x) = f(x)g(x)$
- $h(x) = \sum_{d|x} f(d)g(\frac{x}{d})$

### 5.2.2 狄利克雷卷积

**定义** 两个数论函数  $f, g$  的狄利克雷卷积定义为:

$$(f * g)(n) = \sum_{d|n} f(d)g\left(\frac{n}{d}\right)$$

#### 狄利克雷卷积的性质

- 交换律:  $(f * g) = (g * f)$
- 结合律:  $(f * g) * h = f * (g * h)$
- 分配率:  $f * (g + h) = f * g + f * h$
- $f * \epsilon = f$ ,  $\epsilon$  为 Dirichlet 卷积单位元

## 常见积性函数的狄利克雷卷积

- $\epsilon = \mu \star I \longleftrightarrow \epsilon(n) = \sum_{d|n} \mu(d)$
- $d = I \star I \longleftrightarrow d(n) = \sum_{d|n} 1$
- $\sigma = id \star I \longleftrightarrow \sigma(n) = \sum_{d|n} d$
- $\phi = \mu \star id \longleftrightarrow \phi(n) = \sum_{d|n} d \cdot \mu\left(\frac{n}{d}\right)$

### 5.2.3 杜教筛

**原理** 对于一个积性函数  $f(x)$ , 求  $S(n) = \sum_{i=1}^n f(i)$ , 其中  $n$  范围很大。考虑找到一个合适的函数  $g$  使得  $f, g$  的狄利克雷卷积的前缀和  $f \star g$  容易计算, 可以得到一般使用杜教筛的式子:

$$g(1)S(n) = \sum_{i=1}^n (f \star g)(i) - \sum_{i=2}^n g(i)S(\lfloor \frac{n}{i} \rfloor)$$

如果对于  $(f \star g)$  的前缀和有  $O(1)$  求法, 那么预处理  $f$  函数的前  $n^{\frac{2}{3}}$ , 后面的部分数论分块 + 递归求解。

```
1 // * I = ε
2 // S(n) = 1 - ∑_{d=2}^n S(⌊ n/d ⌋)
3 // * I = id
4 // S(n) = ∑_{i=1}^n i - ∑_{d=2}^n S(⌊ n/d ⌋)
5
6 unordered_map<int, pair<int, ll>> ans;
7 pair<int, ll> solve(ll x) {
8     if (x <= lim - 1)
9         return make_pair(mu[x], phi[x]);
10    if (ans.find(x) != ans.end())
11        return ans[x];
12
13    int mures = 1;
14    ll phires = x * (x + 1) / 2;
15    for (register int l = 2, r; l >= 0 && l <= x; l = r + 1) {
16        r = x / (x / l);
17        pair<int, ll> tmp = solve(x / l);
18        mures -= (r - l + 1) * tmp.first;
19        phires -= (r - l + 1) * tmp.second;
20    }
21    ans[x] = make_pair(mures, phires);
22    return ans[x];
23 }
```

## 5.3 Min25 筛

### 5.3.1 求区间素数之和

```
1  | namespace Min25 {
2  |     int prime[N], id1[N], id2[N], flag[N], ncnt, m;
3  |     LL g[N], sum[N], a[N], T, n;
4  |     inline int ID(LL x) {
5  |         return x <= T ? id1[x] : id2[n / x];
6  |     }
7  |     inline LL calc(LL x) {
8  |         return x * (x + 1) / 2 - 1;
9  |     }
10 |     inline LL f(LL x) {
11 |         return x;
12 |     }
13 |     inline void init() {
14 |         T = sqrt(n + 0.5);
15 |         for (int i = 2; i <= T; i++) {
16 |             if (!flag[i]) prime[++ncnt] = i, sum[ncnt] = sum[ncnt - 1] + i;
17 |             for (int j = 1; j <= ncnt && i * prime[j] <= T; j++) {
18 |                 flag[i * prime[j]] = 1;
19 |                 if (i % prime[j] == 0) break;
20 |             }
21 |         }
22 |         for (LL l = 1; l <= n; l = n / (n / l) + 1) {
23 |             a[++m] = n / l;
24 |             if (a[m] <= T) id1[a[m]] = m; else id2[n / a[m]] = m;
25 |             g[m] = calc(a[m]);
26 |         }
27 |         for (int i = 1; i <= ncnt; i++)
28 |             for (int j = 1; j <= m && (LL)prime[i] * prime[i] <= a[j]; j++)
29 |                 g[j] = g[j] - (LL)prime[i] * (g[ID(a[j] / prime[i])] - sum[i - 1]);
30 |     }
31 |     inline LL solve(LL x) {
32 |         if (x <= 1) return x;
33 |         return n = x, init(), g[ID(n)];
34 |     }
35 | }
```

### 5.3.2 求区间素数个数

```
1  | namespace Min25 {
2  |     int lim, vis[MAXN], prime[MAXN], tot;
3  |     LL g[MAXN], id[MAXN], cnt, pos1[MAXN], pos2[MAXN];
```

```

4   void sieve(int N) {
5       vis[1] = 1;
6       for(int i = 2; i <= N; i++) {
7           if(!vis[i]) prime[++tot] = i;
8           for(int j = 1; j <= tot && i * prime[j] <= N; j++) {
9               vis[i * prime[j]] = 1;
10              if(!(i % prime[j])) break;
11          }
12      }
13  }
14  LL get(LL x) {
15      return x <= lim ? pos1[x] : pos2[N / x];
16  }
17  void solve(ll N) {
18      lim = sqrt(N), sieve(lim);
19      for(LL i = 1, j; i <= N; i = N / j + 1) {
20          j = N / i; id[++cnt] = j; g[cnt] = id[cnt] - 1;
21          j <= lim ? pos1[j] = cnt : pos2[N / j] = cnt;;
22      }
23      for(int j = 1; j <= tot; j++)
24          for(LL i = 1; 1ll * prime[j] * prime[j] <= id[i]; i++)
25              g[i] -= g[get(id[i] / prime[j])] - (j - 1);
26      return g[1];
27  }
28 }
```

## 5.4 GCD 和 LCM

### 5.4.1 欧几里得算法

```

1 // using C++ builtin function: __gcd(a, b)
2 int gcd(int a, int b) {
3     return (b == 0) ? a : gcd(b, a % b);
4 }
5 int lcm(int a, int b) {
6     return a / gcd(a, b) * b;
7 }
```

### 5.4.2 拓展欧几里得算法 (exgcd)

```

1 int x, y;
2 void exgcd(int a, int b, int &x, int &y) {
3     if (b == 0) {
4         x = 1, y = 0;
5         return;
6     }
7     exgcd(b, a % b, x, y);
8     int tmp = x;
9     x = y, y = tmp - (a / b) * y;
10 }

```

## 5.5 快速幂 / 快速乘

### 5.5.1 取模快速幂

```

1 typedef long long ll;
2 ll mod_pow(ll a, ll b, ll mod) {
3     ll ans = 1;
4     while (b) {
5         if (b & 1)
6             ans = (ans * a) % mod;
7         a = a * a % mod;
8         b >>= 1;
9     }
10    return ans;
11 }

```

### 5.5.2 矩阵乘法和快速幂

```

1 struct Matrix {
2     long long a[MAXN][MAXN];
3     int n, m;
4     Matrix operator * (const Matrix b) {
5         Matrix c;
6         memset(c.a, 0, sizeof c.a);
7         c.n = n, c.m = b.m;
8         for (int i = 0; i < n; i++) { // row of a
9             for (int j = 0; j < b.m; j++) { // col of b
10                 for (int k = 0; k < m; k++) { // col of a
11                     c.a[i][j] += (a[i][k] * b.a[k][j]) % MOD;
12                     c.a[i][j] %= MOD;
13                 }
14             }
15         }
16     }
17 }

```

```

14         }
15     }
16     return c;
17 }
18 };
19 Matrix power(Matrix a, int n) {
20     Matrix res;
21     memset(res.a, 0, sizeof res.a);
22     res.n = 3, res.m = 3;
23     res.a[0][0] = res.a[1][1] = res.a[2][2] = 1;
24     while (n) {
25         if (n & 1)
26             res = res * a;
27         a = a * a;
28         n >>= 1;
29     }
30     return res;
31 }
```

## 5.6 快速乘

### 5.6.1 $O(\log n)$ 的快速乘

用于解决  $a \times a \bmod MOD$  爆 long long 的问题，适用于数据较大的情况。

```

1 void add(long long &a, long long &b, long long &c) {
2     a += b;
3     if (a >= c)
4         a -= c;
5 }
6 long long quick_mul(long long a, long long b, long long c) {
7     long long ans = 0;
8     while (b) {
9         if (b & 1)
10            add(ans, a, c);
11         add(a, a, c);
12         b >>= 1;
13     }
14     return ans;
15 }
```

### 5.6.2 $O(1)$ 的快速乘

适用于数据较小 ( $10^{10}$   $10^{18}$ ) 的情况。

```

1 long long quick_mul(long long a, long long b, long long c) {
2     return (a * b - (long long)((long double)a * b / c) * c + c) % c;
3 }
```

## 5.7 乘法逆元

### 5.7.1 exgcd 求逆元

原理:  $ax \equiv 1 \pmod{b}$ , 也即  $ax - by' = 1$ , 令  $y = -y'$  得到  $ax + by = 1$ , 使用 exgcd 求解得到  $x$  后对  $x$  做模  $b$  意义下的处理令  $x \in [0, b - 1]$ ,  $x \in N$ :  $x = ((x \bmod p) + p) \bmod p$  即为所求。

```

1 int x, y;
2 void exgcd(int a, int b, int &x, int &y) {
3     if (b == 0) {
4         x = 1, y = 0;
5         return;
6     }
7     exgcd(b, a % b, x, y);
8     int tmp = x;
9     x = y;
10    y = tmp - (a / b) * y;
11 }
12 int rev(int a, int p) {
13     exgcd(a, p, x, y);
14     return ((x % p) + p) % p;
15 }
```

### 5.7.2 费马小定理求逆元

**费马小定理** 如果  $a$  不是  $p$  的倍数, 则  $a^{p-1} \equiv 1 \pmod{p}$ 。

```

1 long long mod_pow(long long a, long long mod) {
2     long long ans = 1, b = a;
3     while (b) {
4         if (b & 1)
5             ans = ans * a % mod;
6         a = a * a % mod;
7         b >>= 1;
8     }
9     return ans;
10 }
11 long long inv(long long num, long long mod) {
12     return mod_pow(num, mod - 2, mod);
13 }
```

### 5.7.3 线性预处理逆元

```
1 void pretreat_inverse(int n) {
2     inv[1] = 1;
3     for(int i = 2; i < n; i++)
4         inv[i] = (mod - mod / i) * inv[mod % i] % mod;
5 }
```

## 5.8 高斯消元

### 5.8.1 高斯消元求解方程

```
1 double del, cc[MAXN][MAXN]; // coefficient
2 double ans[MAXN];
3 bool gauss(int n) {
4     for (rint i = 1; i <= n; i++) {
5         int r = i;
6         for (rint j = i + 1; j <= n; j++)
7             if (fabs(cc[r][i]) < fabs(cc[j][i]))
8                 r = j;
9         if (fabs(cc[r][i]) < eps)
10            return false;
11         if (i != r)
12             swap(cc[i], cc[r]);
13         double tmp = cc[i][i];
14         for (rint j = i; j <= n + 1; j++)
15             cc[i][j] /= tmp;
16         for (rint j = i + 1; j <= n; j++) {
17             tmp = cc[j][i];
18             for (rint k = i; k <= n + 1; k++)
19                 cc[j][k] -= cc[i][k] * tmp;
20         }
21     }
22     ans[n] = cc[n][n + 1];
23     for (rint i = n - 1; i >= 1; i--) {
24         ans[i] = cc[i][n+1];
25         for (rint j = i + 1; j <= n; j++)
26             ans[i] -= cc[i][j] * ans[j];
27     }
28     return true;
29 }
```

## 5.8.2 高斯消元求矩阵行列式

```
1  ll det(int n) {
2      ll inv, tmp, ans = 1;
3      for (int i = 2; i <= n; i++) {
4          for (int j = i + 1; j <= n; j++)
5              if (!L[i][i] && L[j][i]) {
6                  ans *= -1, swap(L[i], L[j]);
7                  break;
8              }
9          inv = mod_pow(L[i][i], mod - 2);
10         for (int j = i + 1; j <= n; j++) {
11             tmp = L[j][i] * inv % mod;
12             for (int k = i; k <= n; k++)
13                 L[j][k] = (L[j][k] - L[i][k] * tmp % mod + mod) % mod;
14         }
15     }
16     for (int i = 2; i <= n; i++)
17         ans = ans * L[i][i] % mod;
18     return ans;
19 }
```

## 5.9 离散对数 / BSGS 算法

```
1  /**
2  * BSGS 算法: 对于互质的 a, p, 求满足  $a^x \equiv b \pmod{p}$  的 x
3  * (1) 设  $m = \lfloor \sqrt{p} \rfloor$ ,  $x = im - j$ 
4  * (2) 原式转化为  $a^{im-j} \equiv b \pmod{p}$ , 即  $a^{mi} \equiv ba^j \pmod{p}$ 
5  * (3) 从 0~m-1 枚举 j, 将  $ba^j$  存入 hash 表
6  * (4) 从 1~m 枚举 i, 遇到第一个满足  $a^{mi} \equiv ba^j \pmod{p}$  的  $x = im - j$  即为所求
7  * 上述算法有解的充要条件是 p 为质数且  $\gcd(a, p) = 1$ . 如果不满足, 则需要使用 ExBSGS 算
8  * 法:
9  */
10 ll BSGS(ll a, ll b, ll p) {
11     map<ll, ll> mp;
12     a %= p, b %= p;
13     if (a == OLL && b == OLL)
14         return 1LL;
15     if (b == 1LL)
16         return OLL; //  $a^0 \equiv 1 \pmod{p}$ 
17     if (p == OLL) {
18         if (a == OLL)
19             return OLL;
20         else
```

```

20         return 1LL;
21     }
22
23 // ExBGSS
24 long long cnt = 0, t = 1;
25 for (long long d = __gcd(a, p); d != 1LL; d = __gcd(a, p)) {
26     if (b % d)
27         return -1;
28     p /= d, b /= d;
29     t = t * a / d % p;
30     cnt++;
31     if (b == t)
32         return cnt;
33 }
34
35 long long m = (ll)floor(sqrt(1.0 * p - 1)) + 1, ans;
36 for (int i = 0; i < m; i++) {
37     if (i == 0) {
38         ans = b % p;
39         mp[ans] = i;
40         continue;
41     }
42     ans = (ans * a) % p;
43     mp[ans] = i;
44 }
45 long long am = mod_pow(a, m, p);
46 ans = t;           // BSGS: ans=1, ExBSGS: ans=t
47 for (long long i = 1; i <= m; i++) {
48     ans = (ans * am) % p;
49     if (mp[ans]) {
50         long long t = i * m - mp[ans] + cnt;    // ExBSGS 下需要加 cnt
51         return (t % p + p) % p;
52     }
53 }
54 return -1;
55 }

```

## 5.10 欧拉降幂

递归求解  $a^a$  次幂 (见 2019 南京网络赛 B 题)。欧拉降幂公式如下：

$$a^b \equiv \begin{cases} a^{b \mod \phi(p)}, & \gcd(a, p) = 1 \\ a^b, & \gcd(a, b) \neq 1, b < \phi(p) \\ a^{b \mod \phi(p)+\phi(p)}, & \gcd(a, b) \neq 1, b \geq \phi(p) \end{cases} \pmod{p}$$

```

1  ll qm(ll n, ll mod) {
2      return n < mod ? n : (n % mod + mod);
3  }
4  ll mod_pow(ll a, ll b, ll mod) {
5      ll ans = 1;
6      while (b) {
7          if (b & 1)
8              ans = qm(ans * a, mod);
9          a = qm(a * a, mod);
10         b >>= 1;
11     }
12     return ans;
13 }
14 ll solve(ll a, ll b, ll mod) {
15     if (mod == 1)
16         return 0;
17     if (b == 0)
18         return 1;
19     ll tmp = solve(a, b-1, phi[mod]);
20     if (tmp < phi[mod] && tmp)
21         return mod_pow(a, tmp, mod);
22     else
23         return mod_pow(a, tmp + phi[mod], mod);
24 }
```

## 5.11 自适应辛普森积分

```

1  inline double f(double x) { return (c * x + d) / (a * x + b); }
2  inline double simpson(double l, double r) {
3      double mid = (l + r) / 2.0;
4      return (f(l) + 4.0 * f(mid) + f(r)) * (r - l) / 6.0;
5  }
6  inline double solve(double l, double r, double eps) {
7      double mid = (l + r) / 2.0;
8      double s = simpson(l, r), sl = simpson(l, mid), sr = simpson(mid, r);
9      if (fabs(sl + sr - s) <= 15.0 * eps)
10         return (sl + sr + (sl + sr - s) / 15.0);
11     return solve(l, mid, eps / 2.0) + solve(mid, r, eps / 2.0);
12 }
```

## 5.12 二次剩余

**问题场景** 求解  $x^2 = a \pmod{p}$  的  $x$  的两个  $p$  范围内的解中的其中一个解，其中  $p$  是质数（2 被特判掉了），无解返回 -1

```
1 // 写法一
2 ll solve(ll a, ll p) {
3     a %= p;
4     if(a == 0)
5         return 0;
6     if(p == 2)
7         return a;
8     if(mod_pow(a, (p - 1) / 2, p) == p - 1)
9         return -1;
10    ll b, t;
11    while(1) {
12        b = rand() % p;
13        t = b * b - a;
14        w = (t % p + p) % p;
15        if(mod_pow(w, (p - 1) / 2, p) == p - 1)
16            break;
17    }
18    Complex res(b, 1);
19    res = mod_pow(res, (p + 1) / 2);
20    ll x = res.x, y = (p - x) % p;
21    return x <= y ? x : y;
22 }
```

```
1 // 写法二, 已测试
2 namespace quadratic_residue {
3     ll w, p;
4     struct complex { ll x, y; };
5     complex mul(complex a, complex b) {
6         complex ans = {0, 0};
7         ans.x = ((a.x * b.x % p + a.y * b.y % p * w % p) % p + p) % p;
8         ans.y = ((a.x * b.y % p + a.y * b.x % p) % p + p) % p;
9         return ans;
10    }
11
12    ll modpow_real(ll a, ll b) {
13        ll ans = 1;
14        while (b) {
15            if (b & 1)
16                ans = ans * a % p;
17            a = a * a % p, b >= 1;
18        }
19        return ans % p;
20 }
```

```

20 }
21
22 ll modpow_imag(complex a, ll b) {
23     complex ans = {1, 0};
24     while (b) {
25         if (b & 1)
26             ans = mul(ans, a);
27         a = mul(a, a), b >= 1;
28     }
29     return ans.x % p;
30 }
31
32 ll get(ll n, ll prime) {
33     p = prime, n %= p;
34     if (p == 2)
35         return n;
36     if (modpow_real(n, (p-1) / 2) == p-1)
37         return -1;
38     ll a;
39     while (1) {
40         a = rand() % p;
41         w = ((a * a % p - n) % p + p) % p;
42         if (modpow_real(w, (p - 1) / 2) == p - 1)
43             break;
44     }
45     complex x = {a, 1};
46     return modpow_imag(x, (p + 1) / 2);
47 }
48 }
```

## 5.13 中国剩余定理 CRT

**问题场景** 求解如下形式的一元线性方程组：

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \dots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

```

1 int china(int a[], int mods[], int n) {
2     int ans = 0, lcm = 1, x = 0, y = 0;
3     for (int i = 0; i < n; i++)
```

```

4     lcm *= mods[i];
5     for (int i = 0; i < n; i++) {
6         int tp = lcm / mods[i];
7         exgcd(tp, mods[i], x, y);
8         x = (x % mods[i] + mods[i]) % mods[i];
9         ans = (ans + tp * x * a[i]) % lcm;
10    }
11 }
12 }
```

**扩展中国剩余定理 EXCRT** 用于解决模数不互质的情况。

```

1 ll excrt(ll b[], ll a[], ll n) {
2     ll x = 0, y = 0, d = 0;
3     ll M = a[0], R = b[0];
4     for (register int i = 1; i < n; i++) {
5         d = exgcd(M, a[i], x, y);
6         if ((R - b[i]) % d)
7             return -1;
8         x = (R - b[i]) / d * x % a[i];
9         R -= M * x;
10        M = M / d * a[i];
11        R %= M;
12    }
13    return (R % M + M) % M;
14 }
```

## 5.14 素数判定 & 大数质因子分解

```

1 namespace PollardRho {
2     const int test_time = 5;
3
4     default_random_engine e(time(NULL));
5     uniform_int_distribution<ll> u(0, 1ll << 63);
6     map<ll, ll> factors;
7
8     ll quick_mul(ll a, ll b, ll mod) {
9         return (a * b - (ll)((ld)a / mod * b) * mod + mod) % mod;
10    }
11
12    ll mod_pow(ll x, ll p, ll mod) {
13        ll ans = 1;
14        while (p) {
```

```

15     if (p & 1)
16         ans = quick_mul(ans, x, mod);
17         x = quick_mul(x, x, mod), p >= 1;
18     }
19     return ans;
20 }
21
22 bool miller_rabin(ll p) {
23     if (p < 2)
24         return 0;
25     if (p == 2 || p == 3)
26         return 1;
27     ll d = p - 1, r = 0;
28     for (; !(d & 1); r++, d >= 1);
29     for (ll k = 0; k < test_time; k++) {
30         ll a = u(e) % (p - 2) + 2, x = mod_pow(a, d, p);
31         if (x == 1 || x == p-1)
32             continue;
33         for (int i = 0; i < r - 1; i++) {
34             x = quick_mul(x, x, p);
35             if (x == p - 1)
36                 break;
37         }
38         if (x != p - 1)
39             return false;
40     }
41     return true;
42 }
43
44 ll f(ll x, ll c, ll mod) {
45     return (quick_mul(x, x, mod) + c) % mod;
46 }
47
48 ll pollard_rho(ll x) {
49     ll s = 0, t = 0, c = u(e) % (x - 1) + 1, val = 1, d;
50     int step = 0, goal = 1;
51     for (goal = 1;; goal <= 1, s = t, val = 1) {
52         for (step = 1; step <= goal; step++) {
53             t = f(t, c, x), val = quick_mul(val, (ll)abs(t - s), x);
54             if ((step % 127) == 0) {
55                 d = __gcd(val, x);
56                 if (d > 1)
57                     return d;
58             }
59         }
60         d = __gcd(val, x);
61         if (d > 1)
62             return d;

```

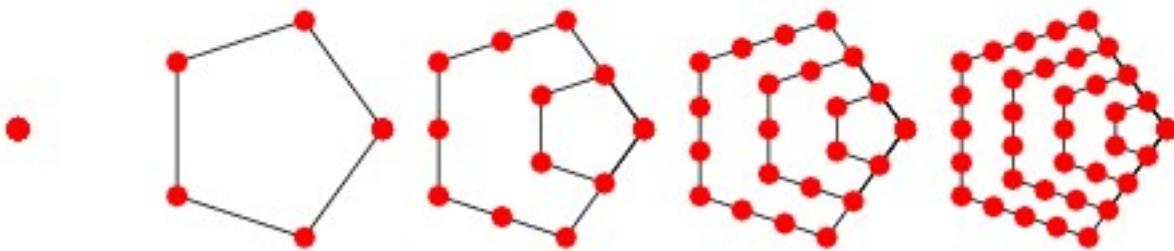
```

63     }
64 }
65
66 void find(ll n) {
67     if (n == 1)
68         return;
69     if (miller_rabin(n)) {
70         factors[n]++;
71         return;
72     }
73     ll p = n;
74     while (p >= n)
75         p = pollard_rho(p);
76     find(p), find(n / p);
77 }
78
79 vector<pair<ll, ll>> solve(ll x) {
80     vector<pair<ll, ll>> ans;
81     factors.clear(), find(x);
82     for (auto fac : factors)
83         ans.emplace_back(fac);
84     return ans;
85 }
86 }

```

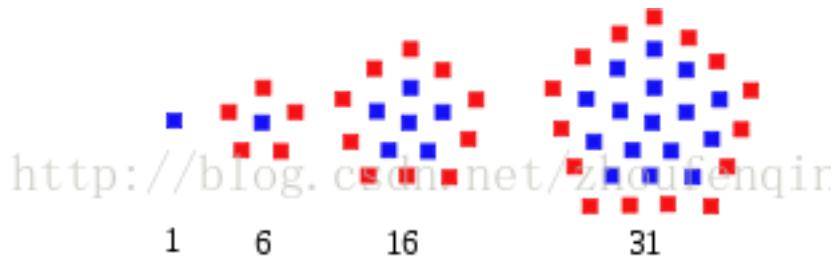
## 5.15 欧拉五边形数 & 整数拆分

**欧拉五边形数** 设第  $n$  个五边形数为  $f(n)$ , 则  $f(n) = \frac{n(3n-1)}{2} = \{1, 5, 12, 22, 35, 51, 70, \dots\}$ 。



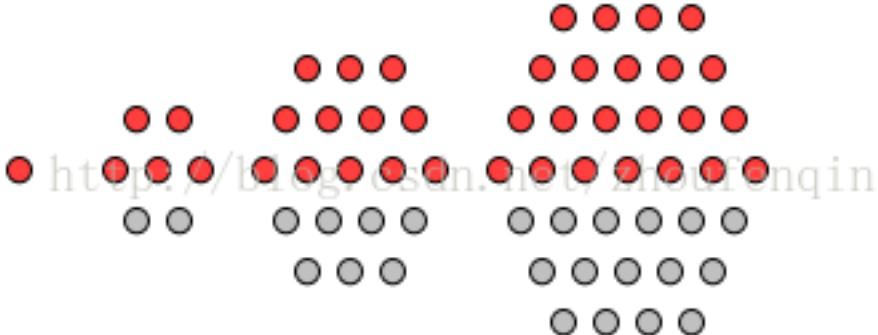
**广义五边形数**  $n$  的取值为  $0, 1, -1, 2, -2, \dots$

$$\text{中心五边形数 } P_n = \frac{5(n-1)^2 + 5(n-1) + 2}{2}$$



$$\text{中心六边形数} \quad P_n = \frac{3n^2-n}{2} + \frac{3(1-n)^2-(1-n)}{2} = 3n(n-1) + 1$$

$$1=1+0 \quad 7=5+2 \quad 19=12+7 \quad 37=22+15$$



$$\text{五边形数测试公式} \quad n = \frac{\sqrt{24x+1}+1}{6}$$

- 若  $n$  是自然数，则  $x$  是五边形数；而且恰为第  $n$  个五边形数。
- 若  $n$  不是自然数，则  $x$  不是五边形数。

$$\text{生成函数} \quad g(x) = \frac{x(2x+1)}{(1-x)^3} = x + 5x^2 + 12x^3 + 22x^4 + \dots$$

```

1 // hdu4658 要求拆分的数中每个数出现的次数不能大于等于 k 次
2 ll dp[maxn];
3 ll five(ll x) {
4     ll ans=3 * x * x - x;
5     return ans / 2;
6 }
7 void init() {
8     dp[0] = 1;
9     for(int i = 1; i < maxn; i++) {
10         for(int j = 1; ; j++) {
11             ll k = five(j); // 获取五边形数
12             if(k <= i) {
13                 if(j % 2 == 0) // 判断奇偶从而判断系数为正还是为负
14                     dp[i] = dp[i] - dp[i-k];
15                 else
16                     dp[i] = dp[i] + dp[i-k];
17                 if(dp[i] >= MOD)

```

```

18     dp[i] %= MOD;
19     if(dp[i] < 0)
20         dp[i] += MOD;
21     k = five(-1 * j);
22     if(k <= i) {
23         if(j % 2 == 0)
24             dp[i] = dp[i] - dp[i-k];
25         else
26             dp[i] = dp[i] + dp[i-k];
27         if(dp[i] >= MOD)
28             dp[i] %= MOD;
29         if(dp[i] < 0)
30             dp[i] += MOD;
31     }
32     else
33         break;
34 }
35 else
36     break;
37 }
38 }
39 }

// hdu4651 将 n 拆成几个小于等于 n 的数相加的形式，问有多少种拆法
int dp[maxn];
void init() {
    memset(dp, 0, sizeof dp), dp[0] = 1;
    for (int i = 1; i < maxn; i++) {
        for (int j = 1, r = 1; i - (3 * j * j - j) / 2 >= 0; j++, r *= -1) {
            dp[i] += dp[i - (3 * j * j - j) / 2] * r;
            dp[i] = (dp[i] % mod + mod) % mod;
            if (i - (3 * j * j + j) / 2 >= 0) {
                dp[i] += dp[i - (3 * j * j + j) / 2] * r;
                dp[i] = (dp[i] % mod + mod) % mod;
            }
        }
    }
}
}

```

## 5.16 线性规划的单纯形算法

**问题场景** 有  $n$  个实数变量和  $m$  个约束，其中第  $i$  个约束形如  $\sum_{j=1}^n a_{ij}x_j \leq b_i(x_j \geq 0)$ ，求目标函数  $F = \sum_{j=1}^n c_jx_j$  的最大值。

转化标准形式

- 如果目标函数为最小化，则可以通过将目标函数取负转化为求最大值： $\max -z = \sum_{j=1}^n c'_j x_j (c'_j = -c_j)$
- 将小于等于形式的不等式约束转化为等式约束：在等式左侧加上松弛变量  $x_{m+i}$
- 将大于等于形式的不等式约束转化为等式约束：在等式左侧减去松弛变量  $x_{m+i}$
- 无约束变量  $\inf < x_j < \inf$  可以转化为两个非负变量的差值： $x_j = x'_j - x''_j$

```

1 const int maxn = 25, maxm = 100050;
2 const double eps = 1e-8, inf = (double)(1ll << 60);
3
4 int n, m, t, id[maxn << 1]; // n: variables, m: constraint
5 double a[maxn][maxn], ans[maxn]; // a: coefficient matrix
6 void pivot(int l, int e) {
7     swap(id[n+1], id[e]);
8     double tmp = a[l][e];
9     a[l][e] = 1;
10    for (int j = 0; j <= n; j++)
11        a[l][j] /= tmp;
12    for (int i = 0; i <= m; i++)
13        if (i != l && abs(a[i][e]) > eps) {
14            tmp = a[i][e], a[i][e] = 0;
15            for (int j = 0; j <= n; j++)
16                a[i][j] -= a[l][j] * tmp;
17        }
18    }
19 bool init() {
20     while (1) {
21         int e = 0, l = 0;
22         for (int i = 1; i <= m; i++)
23             if (a[i][0] < -eps && (!l || (rand() & 1)))
24                 l = i;
25         if (!l)
26             break;
27         for (int j = 1; j <= n; j++)
28             if (a[l][j] < -eps && (!e || (rand() & 1)))
29                 e = j;
30         if (!e) {
31             printf("Infeasible\n"); // 不存在满足约束的解
32             return false;
33         }
34         pivot(l, e);
35     }
36     return true;
37 }
38 bool simplex() {
39     while (1) {

```

```

40     int l = 0, e = 0;
41     double minv = inf;
42     for (int j = 1; j <= n; j++) {
43         if (a[0][j] > eps) {
44             e = j;
45             break;
46         }
47         if (!e)
48             break;
49     for (int i = 1; i <= m; i++)
50         if (a[i][e] > eps && a[i][0] / a[i][e] < minv)
51             minv = a[i][0] / a[i][e], l = i;
52     if (!l) {
53         printf("Unbounded\n");           // 任意解
54         return false;
55     }
56     pivot(l, e);
57 }
58 return true;
59 }
60 int main() {
61     srand(time(NULL));
62     read(n), read(m), read(t);
63     for (int i = 1; i <= n; i++)
64         read(a[0][i]);           // 目标函数  $F(x) = C_i x_i$ 
65     for (int i = 1; i <= m; i++) {
66         for (int j = 1; j <= n; j++)
67             read(a[i][j]);
68         read(a[i][0]);
69     }
70     for (int i = 1; i <= n; i++)
71         id[i] = i;           // 保存变量顺序
72     if (init() && simplex()) {
73         printf("%.8lf\n", -a[0][0]);
74         if (t) {
75             for (int i = 1; i <= m; i++)
76                 ans[id[n + i]] = a[i][0];
77             for (int i = 1; i <= n; i++)
78                 printf("%.8lf ", ans[i]);
79         }
80     }
81     return 0;
82 }
```

# Chapter 6

## 多项式

### 6.1 拉格朗日插值法

```
1 // 给定 n 组 (xi, yi), 求 f(k) mod mod
2 ll Lagrange(ll x[], ll y[], int n, ll k, ll mod) {
3     ll s1 = 0ll, s2 = 0ll, ans = 0ll;
4     for (int i = 0; i < n; i++) {
5         s1 = y[i] % mod, s2 = 1ll;
6         for (int j = 0; j < n; j++)
7             if (i != j)
8                 s1 = s1 * (k - x[j]) % mod,
9                 s2 = s2 * ((x[i] - x[j]) % mod) % mod;
10            ans += s1 * inv(s2) % mod;
11            ans = (ans + mod) % mod;
12        }
13    return ans;
14 }
```

### 6.2 快速傅里叶变换 (FFT)

```
1 namespace FastFourierTransform {
2     using real = double;
3
4     struct C {
5         real x, y;
6         C() : x(0), y(0) {}
7         C(real x, real y) : x(x), y(y) {}
8         inline C operator+(const C &c) const { return C(x + c.x, y + c.y); }
9         inline C operator-(const C &c) const { return C(x - c.x, y - c.y); }
10        inline C operator*(const C &c) const { return C(x * c.x - y * c.y, x * c.y
11            + y * c.x); }
```

```

11     inline C conj() const { return C(x, -y); }
12 }
13
14 const real PI = acosl(-1);
15 int base = 1;
16 vector< C > rts = { {0, 0}, {1, 0} };
17 vector< int > rev = {0, 1};
18
19 void ensure_base(int nbase) {
20     if(nbase <= base) return;
21     rev.resize(1 << nbase), rts.resize(1 << nbase);
22     for(int i = 0; i < (1 << nbase); i++)
23         rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
24     while(base < nbase) {
25         real angle = PI * 2.0 / (1 << (base + 1));
26         for(int i = 1 << (base - 1); i < (1 << base); i++) {
27             rts[i << 1] = rts[i];
28             real angle_i = angle * (2 * i + 1 - (1 << base));
29             rts[(i << 1) + 1] = C(cos(angle_i), sin(angle_i));
30         }
31         ++base;
32     }
33 }
34
35 void fft(vector< C > &a, int n) {
36     assert((n & (n - 1)) == 0);
37     int zeros = __builtin_ctz(n);
38     ensure_base(zeros);
39     int shift = base - zeros;
40     for(int i = 0; i < n; i++)
41         if(i < (rev[i] >> shift))
42             swap(a[i], a[rev[i] >> shift]);
43     for(int k = 1; k < n; k <= 1)
44         for(int i = 0; i < n; i += 2 * k)
45             for(int j = 0; j < k; j++) {
46                 C z = a[i + j + k] * rts[j + k];
47                 a[i + j + k] = a[i + j] - z;
48                 a[i + j] = a[i + j] + z;
49             }
50     }
51
52     vector<ll> multiply(const vector<int> &a, const vector<int> &b) {
53         int need = (int) a.size() + (int) b.size() - 1;
54         int nbase = 1;
55         while((1 << nbase) < need) nbase++;
56         ensure_base(nbase);
57         int sz = 1 << nbase;
58         vector<C> fa(sz);

```

```

59     for(int i = 0; i < sz; i++) {
60         int x = (i < (int) a.size() ? a[i] : 0);
61         int y = (i < (int) b.size() ? b[i] : 0);
62         fa[i] = C(x, y);
63     }
64     fft(fa, sz);
65     C r(0, -0.25 / (sz >> 1)), s(0, 1), t(0.5, 0);
66     for(int i = 0; i <= (sz >> 1); i++) {
67         int j = (sz - i) & (sz - 1);
68         C z = (fa[j] * fa[j] - (fa[i] * fa[i]).conj()) * r;
69         fa[j] = (fa[i] * fa[i] - (fa[j] * fa[j]).conj()) * r;
70         fa[i] = z;
71     }
72     for(int i = 0; i < (sz >> 1); i++) {
73         C A0 = (fa[i] + fa[i + (sz >> 1)]) * t;
74         C A1 = (fa[i] - fa[i + (sz >> 1)]) * t * rts[(sz >> 1) + i];
75         fa[i] = A0 + A1 * s;
76     }
77     fft(fa, sz >> 1);
78     vector< ll > ret(need);
79     for(int i = 0; i < need; i++) {
80         ret[i] = llround(i & 1 ? fa[i >> 1].y : fa[i >> 1].x);
81     }
82     return ret;
83 }
84 };

```

## 6.3 快速数论变换 (NTT)

### 6.3.1 写法一 (int)

```
1  namespace NTT {           // 常数较小，推荐
2      const int mod = 998244353;
3      int base = 1, max_base, root;
4      vector<int> rev = {0, 1}, rts = {0, 1};
5
6      void init() {
7          int tmp = mod - 1;
8          max_base = 0, root = 2;
9          while (tmp % 2 == 0)
10              tmp >>= 1, max_base++;
11          while (mod_pow(root, (mod - 1) >> 1) == 1)
12              ++root;
13          root = mod_pow(root, (mod - 1) >> max_base);
14      }
15      inline unsigned add(unsigned x, unsigned y) {
16          x += y;
17          if(x >= mod) x -= mod;
18          return x;
19      }
20      inline unsigned mul(unsigned a, unsigned b) {
21          return 1ull * a * b % (unsigned long long) mod;
22      }
23      inline int mod_pow(int x, int n) {
24          int ret = 1;
25          while(n > 0) {
26              if(n & 1) ret = mul(ret, x);
27              x = mul(x, x);
28              n >>= 1;
29          }
30          return ret;
31      }
32      inline int inverse(int x) {
33          return mod_pow(x, mod - 2);
34      }
35      void ensure_base(int nbase) {
36          if (nbase <= base)
37              return;
38          rev.resize(1 << nbase), rts.resize(1 << nbase);
39          for(int i = 0; i < (1 << nbase); i++)
40              rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
41          while(base < nbase) {
42              int z = mod_pow(root, 1 << (max_base - 1 - base));
43              for(int i = 1 << (base - 1); i < (1 << base); i++)
```

```

44         rts[i << 1] = rts[i], rts[(i << 1) + 1] = mul(rts[i], z);
45         ++base;
46     }
47 }
48 void ntt(vector<int> &a) {
49     const int n = (int) a.size();
50     int zeros = __builtin_ctz(n);
51     ensure_base(zeros);
52     int shift = base - zeros;
53     for(int i = 0; i < n; i++)
54         if(i < (rev[i] >> shift))
55             swap(a[i], a[rev[i] >> shift]);
56     for(int k = 1; k < n; k <= 1)
57         for(int i = 0; i < n; i += 2 * k)
58             for(int j = 0; j < k; j++) {
59                 int z = mul(a[i + j + k], rts[j + k]);
60                 a[i + j + k] = add(a[i + j], mod - z);
61                 a[i + j] = add(a[i + j], z);
62             }
63     }
64 vector<int> multiply(vector<int> a, vector<int> b) {
65     int need = a.size() + b.size() - 1;
66     int nbase = 1;
67     while((1 << nbase) < need)
68         nbase++;
69     ensure_base(nbase);
70     int sz = 1 << nbase;
71     a.resize(sz, 0), b.resize(sz, 0), ntt(a), ntt(b);
72     int inv_sz = inverse(sz);
73     for(int i = 0; i < sz; i++)
74         a[i] = mul(a[i], mul(b[i], inv_sz));
75     reverse(a.begin() + 1, a.end()), ntt(a), a.resize(need);
76     return a;
77 }
78 };

```

### 6.3.2 写法二 (ll)

```

1 namespace NTT {
2     const int MAXN = 1000;
3     const ll G = 311, Gi = 33274811811;
4     ll A[MAXN * 3], B[MAXN * 3], rev[MAXN * 3], limit, L;
5
6     void process_inverse(ll lim, ll l) { /* 处理逆元 */
7         memset(rev, 0, sizeof rev);
8         limit = lim, L = l;

```

```

9     for (int i = 1; i < lim; i++)
10        rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (L - 1));
11    }
12    void ntt(ll a[], ll r[], int f) {
13      for (int i = 0; i < limit; i++)
14        if (i < r[i])
15          swap(a[i], a[r[i]]);
16      for (int mid = 1; mid < limit; mid <= 1) {
17        ll Wn = mod_pow(f == 1 ? G : Gi, (mod - 1) / (mid << 1), mod);
18        for (int j = 0; j < limit; j += (mid << 1)) {
19          ll w = 1;
20          for (int k = 0; k < mid; k++, w = (w * Wn) % mod) {
21            ll x = a[j + k] % mod, y = w * a[j + k + mid] % mod;
22            a[j + k] = (x + y) % mod,
23            a[j + k + mid] = (x - y + mod) % mod;
24          }
25        }
26      }
27      if (f == -1) {           // 逆变换
28        int in = inv(limit);
29        for (int i = 0; i < limit; i++)
30          a[i] = (1ll * a[i] * in + mod) % mod;
31      }
32    }
33    void multiply(ll a[], int acnt, ll b[], int bcnt, ll ans[], int &cnt) {
34      memset(A, 0, sizeof A), memset(B, 0, sizeof B);
35      for (int i = 0; i < acnt; i++)
36        A[i] = a[i];
37      for (int i = 0; i < bcnt; i++)
38        B[i] = b[i];
39      limit = 1, L = 0;
40      while (limit <= acnt + bcnt)
41        limit <= 1, L++;
42      process_inverse(limit, L);
43      ntt(A, 1), ntt(B, 1);
44      for (int i = 0; i < limit; i++)
45        A[i] = (A[i] * B[i]) % mod;
46      ntt(A, -1);
47      ll ir = inv(limit);
48      for (int i = 0; i <= acnt + bcnt; i++)
49        ans[i] = (A[i] * ir) % mod;
50      cnt = acnt + bcnt;
51    }
52    pair<int, int> ensure_base(int n) {
53      int lim = 1, L = 0;
54      while (lim < (n << 1))
55        lim <= 1, L++;
56      return make_pair(lim, L);

```

```
57     }
58 };
```

### 6.3.3 分治 NTT

```
1 vector<int> solve(int l, int r) {
2     if (l == r) {
3         vector<int> res;
4         res.emplace_back(c[l]), res.emplace_back(b[l]);
5         return res;
6     }
7     int mid = (l + r) >> 1;
8     return ntt.multiply(solve(l, mid), solve(mid + 1, r));
9 }
```

### 6.3.4 任意模数 NTT

```
1 namespace ArbitraryNTT {
2     const int M1 = 998244353, M2 = 1004535809, M3 = 469762049;
3     int n, n1, n2, P;
4     int r[maxn];
5     ll a[3][maxn], b[3][maxn], ans[maxn];
6
7     void process_inverse(int x){
8         int L = 0;
9         for(n = 1; n < x; n <= 1, L++);
10        for(int i = 0; i < n; i++)
11            r[i] = (r[i>>1]>>1) | ((i&1)<<(L-1));
12    }
13
14    ll mul(ll x, int y, ll MOD){
15        ll res = 0;
16        for(; y; x = (x << 1) % MOD, y >= 1)
17            if(y & 1) res = (res + x) % MOD;
18        return res;
19    }
20
21    ll mod_pow(ll x, int y, int MOD){
22        ll res = 1;
23        for(; y; x = x * x % MOD, y >= 1)
24            if(y & 1) res = res * x % MOD;
25        return res;
26    }
```

```

27
28     void ntt(ll *A, int DFT, int MOD){
29         for(int i = 0; i < n; i++)
30             if(i < r[i])
31                 swap(A[i], A[r[i]]);
32
33         for(int s = 1; (1<<s) <= n; s++){
34             int m = 1 << s;
35             ll wn = mod_pow(3, (DFT == 1) ? (MOD-1)/m : (MOD-1)-(MOD-1)/m, MOD);
36             for(int k = 0; k < n; k += m) {
37                 ll w = 1;
38                 for(int j = 0; j < (m >> 1); j++){
39                     ll u = A[k+j], t = w * A[k+j+(m>>1)] % MOD;
40                     A[k+j] = (u+t) % MOD;
41                     A[k+j+(m>>1)] = (u-t+MOD) % MOD;
42                     w = w * wn % MOD;
43                 }
44             }
45         }
46         if(DFT == -1){
47             ll Inv = mod_pow(n, MOD-2, MOD);
48             for(int i = 0; i < n; i++)
49                 A[i] = A[i] * Inv % MOD;
50         }
51     }
52
53     void work(int x, int MOD){
54         ntt(a[x], 1, MOD);
55         ntt(b[x], 1, MOD);
56         for(int i = 0; i < n; i++)
57             a[x][i] = a[x][i] * b[x][i] % MOD;
58         ntt(a[x], -1, MOD);
59     }
60
61     void CRT(){
62         ll M = 111 * M1 * M2;
63         for(int i = 0; i < n; i++){
64             ll temp = 0;
65             temp = (temp + mul(a[0][i] * M2 % M, mod_pow(M2, M1-2, M1), M)) % M;
66             temp = (temp + mul(a[1][i] * M1 % M, mod_pow(M1, M2-2, M2), M)) % M;
67             a[1][i] = temp;
68         }
69         for(int i = 0; i < n; i++){
70             ll temp = (a[2][i] - a[1][i] % M3 + M3) % M3 * mod_pow(M/M3, M3-2, M3)
71             ↳ % M3;
72             ans[i] = (M % P * temp % P + a[1][i] % P) % P;
73         }
74     }

```

```

74
75     void solve() {
76         scanf("%d%d%d", &n1, &n2, &P);
77         n1++; n2++;
78         ll x;
79         for(int i = 0; i < n1; i++) {
80             scanf("%lld", &x);
81             for(int j = 0; j < 3; j++)
82                 a[j][i] = x % P;
83         }
84         for(int i = 0; i < n2; i++) {
85             scanf("%lld", &x);
86             for(int j = 0; j < 3; j++)
87                 b[j][i] = x % P;
88         }
89         process_inverse(n1 + n2);
90         work(0, M1), work(1, M2), work(2, M3);
91         CRT();
92         for(int i = 0; i < n1+n2-1; i++)
93             printf("%lld ", ans[i]);
94     }
95 }
```

## 6.4 多项式求逆

```

1 /**
2 * 多项式求逆，使用 NTT，求 a(x)*b(x) === 1 (mod x^n)
3 * c[] 是临时数组；数组长度开 3 倍；答案在 b[] 中
4 */
5 void polynomial_inverse(ll a[], ll b[], ll c[], int n) {
6     if (n == 1) {
7         b[0] = inv(a[0]);
8         return;
9     }
10    polynomial_inverse(a, b, c, (n + 1) >> 1);
11
12    pair<int, int> tmp = ensure_base(n);
13    int lim = tmp.first, L = tmp.second;
14    process_inverse(lim, L);
15
16    for (int i = 0; i < n; i++)
17        c[i] = a[i];
18    for (int i = n; i < lim; i++)
19        c[i] = 0ll;
```

```

20     ntt(c, 1), ntt(b, 1);
21     for (int i = 0; i < lim; i++)
22         b[i] = ((21l - b[i] * c[i] % mod) + mod) % mod * b[i] % mod;
23     ntt(b, -1);
24     for (int i = n; i < lim; i++)
25         b[i] = 0;
26 }

```

## 6.5 多项式求导 & 指对数运算

```

1  namespace Polynomial {
2      const ll P = 998244353, g = 3, gi = 332748118;
3      static int rev[N];
4      int lim, bit;
5      int add(int a, int b) {
6          return (a += b) >= P ? a - P : a;
7      }
8      int qpow(int a, int b) {
9          int prod = 1;
10         while(b) {
11             if(b & 1) prod = (ll)prod * a % P;
12             a = (ll)a * a % P, b >>= 1;
13         }
14         return (prod + P) % P;
15     }
16     void process_inverse() {
17         for(int i = 1; i < lim; ++i)
18             rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (bit - 1));
19     }
20     void NTT(int *A, int inv) {
21         for(int i = 0; i < lim; ++i)
22             if(i < rev[i]) swap(A[i], A[rev[i]]);
23         for(int mid = 1; mid < lim; mid <= 1) {
24             int tmp = qpow(inv == 1 ? g : gi, (P - 1) / (mid << 1));
25             for(int j = 0; j < lim; j += (mid << 1)) {
26                 int w = 1;
27                 for(int k = 0; k < mid; ++k, w = (ll)w * tmp % P) {
28                     int x = A[j + k], y = (ll)w * A[j + k + mid] % P;
29                     A[j + k] = (x + y) % P;
30                     A[j + k + mid] = (ll)(x - y + P) % P;
31                 }
32             }
33         }
34         if(inv == 1) return;

```

```

35     int invn = qpow(lim, P - 2);
36     for(int i = 0; i < lim; ++i)
37         A[i] = (ll)A[i] * invn % P;
38 }
39 static int x[N], y[N];
40 void mul(int *a, int *b) {
41     memset(x, 0, sizeof x);
42     memset(y, 0, sizeof y);
43     for(int i = 0; i < (lim >> 1); ++i)
44         x[i] = a[i], y[i] = b[i];
45     NTT(x, 1), NTT(y, 1);
46     for(int i = 0; i < lim; ++i)
47         x[i] = (ll)x[i] * y[i] % P;
48     NTT(x, -1);
49     for(int i = 0; i < lim; ++i)
50         a[i] = x[i];
51 }
52 static int c[2][N];
53 void inv(int *a, int n) {
54     int p = 0;
55     memset(c, 0, sizeof c);
56     c[0][0] = qpow(a[0], P - 2);
57     lim = 2, bit = 1;
58     while(lim <= (n << 1))
59     {
60         lim <= 1, bit++;
61         process_inverse();
62         p ^= 1;
63         memset(c[p], 0, sizeof c[p]);
64         for(int i = 0; i <= lim; ++i)
65             c[p][i] = add(c[p^1][i], c[p^1][i]);
66         mul(c[p^1], c[p^1]);
67         mul(c[p^1], a);
68         for(int i = 0; i <= lim; ++i)
69             c[p][i] = add(c[p][i], P - c[p^1][i]);
70     }
71     for(int i = 0; i < lim; ++i)
72         a[i] = c[p][i];
73 }
74 void derivative(int *a, int n) {
75     for(int i = 1; i <= n; ++i)
76         a[i - 1] = (ll)a[i] * i % P;
77     a[n] = 0;
78 }
79 void inter(int *a, int n) {
80     for(int i = n; i >= 1; --i)
81         a[i] = (ll)a[i - 1] * qpow(i, P - 2) % P;
82     a[0] = 0;

```

```

83 }
84 static int b[N], T[N], K[N];
85 void ln(int a[], int n) {
86     memcpy(b, a, sizeof b);
87     inv(b, n), derivative(a, n);
88     while(lim <= (n << 2)) lim <<= 1, bit++;
89     process_inverse();
90     mul(a, b);
91     inter(a, n);
92     for(int i = n + 1; i <= lim; ++i)
93         a[i] = 0;
94 }
95 void exp(int a[], int n) {
96     int z, d;
97     z = lim = 2, d = bit = 1;
98     memset(K, 0, sizeof K), K[0] = 1;
99     while(z <= (n << 1)) {
100         z <<= 1, d++;
101         for(int i = 0; i < (z>>1); ++i)
102             T[i] = K[i];
103         ln(T, (z >> 1) - 1);
104         for(int i = 0; i < (z >> 1); ++i)
105             T[i] = add(a[i] + (i == 0), P - T[i]);
106         lim = z, bit = d, process_inverse(), mul(K, T);
107         for(int i = z; i <= (z << 1); ++i)
108             K[i] = T[i] = 0;
109     }
110     for(int i = 0; i <= n; ++i)
111         a[i] = K[i];
112 }
113 }
```

## 6.6 多项式除法、多点求值、快速插值

来自《多项式相关算法集成》。

```

1 const ll mod_v = 1711 * (1 << 27) + 1;           // 998244353
2 const int MAXL = 18, N = 1 << MAXL, M = 1 << MAXL; // MAXL 最大取 mov_v 的 2 次幂
3 struct polynomial {
4     ll coef[N]; // size 设为多项式系数个数的两倍
5     ll a[N], b[N], d[N], r[N];
6     ll xcoor[N], ycoor[N], v[N], mcoef[N]; // size 设为点个数的两倍
7     vector<vector<ll> > poly_divisor;
8
9     static ll mod_pow(ll x, ll n, ll m) {
```

```

10    ll ans = 1;
11    while(n > 0){
12        if(n & 1)
13            ans = ans * x % m;
14        x = x * x % m, n >= 1;
15    }
16    return ans;
17 }

18 struct FastNumberTheoreticTransform {
19     ll omega[N], omegaInverse[N];
20     int range;
21     void init (const int& n) {
22         range = n;
23         ll base = mod_pow(3, (mod_v - 1) / n, mod_v);
24         ll inv_base = mod_pow(base, mod_v - 2, mod_v);
25         omega[0] = omegaInverse[0] = 1;
26         for (int i = 1; i < n; ++i) {
27             omega[i] = omega[i - 1] * base % mod_v;
28             omegaInverse[i] = omegaInverse[i - 1] * inv_base % mod_v;
29         }
30     }
31     void transform (ll *a, const ll *omega, const int &n) {
32         for (int i = 0, j = 0; i < n; ++i) {
33             if (i > j) std::swap (a[i], a[j]);
34             for(int l = n >> 1; (j ^= l) < l; l >>= 1);
35         }
36         for (int l = 2; l <= n; l <= 1) {
37             int m = l / 2;
38             for (ll *p = a; p != a + n; p += l) {
39                 for (int i = 0; i < m; ++i) {
40                     ll t = omega[range / l * i] * p[m + i] % mod_v;
41                     p[m + i] = (p[i] - t + mod_v) % mod_v;
42                     p[i] = (p[i] + t) % mod_v;
43                 }
44             }
45         }
46     }
47     void dft (ll *a, const int& n) {
48         transform(a, omega, n);
49     }
50     void idft (ll *a, const int& n) {
51         transform(a, omegaInverse, n);
52         for (int i = 0; i < n; ++i) a[i] = a[i] * mod_pow(n, mod_v - 2, mod_v)
53             → % mod_v;
54     }
55 } fnt ;
56

```

```

57 ll mod_trans(ll v) {
58     return abs(v) <= mod_v / 2 ? v : (v < 0 ? v + mod_v : v - mod_v);
59 }
60
61 // 二分求  $\prod x - xi$  的所有二分子多项式的系数
62 void binary_subpoly(int l, int r, int idx) {
63     if (l == r - 1) {
64         poly_divisor[idx].push_back(-xcoor[1]);
65         poly_divisor[idx].push_back(1);
66     } else {
67         int lidx = (idx << 1) + 1, ridx = lidx + 1;
68         binary_subpoly(l, (l + r) / 2, lidx);
69         binary_subpoly((l + r) / 2, r, ridx);
70         int t = poly_divisor[lidx].size() + poly_divisor[ridx].size() - 1;
71         int p = 1;
72         while(p < t) p <= 1;
73         copy(poly_divisor[lidx].begin(), poly_divisor[lidx].end(), a);
74         fill(a + poly_divisor[lidx].size(), a + p, 0);
75         copy(poly_divisor[ridx].begin(), poly_divisor[ridx].end(), b);
76         fill(b + poly_divisor[ridx].size(), b + p, 0);
77         fnt.dft(a, p);
78         fnt.dft(b, p);
79         for (int i = 0; i < p; i++) a[i] *= b[i];
80         fnt.idft(a, p);
81         for (int i = 0; i < t; i++)
82             → poly_divisor[idx].push_back(mod_trans(a[i]));
83     }
84 }
85
86 // 模  $x^{deg}$ , a 为要求逆元的多项式系数, 结果存放在 b[0~deg] 中
87 // T(deg) = T(deg/2) + deg*log(deg), 复杂度 O(deg*log(deg))
88 void polynomial_inverse(int deg, ll* a, ll* b, ll* tmp) {
89     if(deg == 1) {
90         b[0] = mod_pow(a[0], mod_v - 2, mod_v);
91     } else {
92         polynomial_inverse((deg + 1) >> 1, a, b, tmp);
93         int p = 1;
94         while(p < (deg << 1) - 1) p <= 1;
95         copy(a, a + deg, tmp);
96         fill(tmp + deg, tmp + p, 0);
97         fill(b + ((deg + 1) >> 1), b + p, 0);
98         //fnt.init(p);
99         fnt.dft(tmp, p);
100        fnt.dft(b, p);
101        for(int i = 0; i != p; ++i) {
102            b[i] = (2 - tmp[i] * b[i] % mod_v) * b[i] % mod_v;
103            if(b[i] < 0) b[i] += mod_v;
104        }
105    }
106 }

```

```

104         fnt.idft(b, p);
105         fill(b + deg, b + p, 0);
106     }
107 }
108
109 // A = D*B + R,A 为 n 项 n-1 次幂,B 为 m 项 m-1 次幂,D 为 n-m+1 项 n-m 次幂,R 为
110 // → m-1 项 m-2 次幂
111 // 要求 a,b 中系数以低次到多次顺序排列
112 // n >= m, 复杂度 O(n*logn); n < m, 复杂度 O(n)
113 int polynomial_division(int n, int m, ll *A, ll *B, ll *D, ll *R) {
114     if (n < m) {
115         copy(A, A + n, R);
116         return n;
117     } else {
118         static ll A0[N], B0[N], tmp[N]; //数组太大会爆栈, 添加到全局区
119
120         int p = 1, t = n - m + 1;
121         while(p < (t << 1) - 1) p <= 1;
122
123         fill(A0, A0 + p, 0);
124         reverse_copy(B, B + m, A0);
125         polynomial_inverse(t, A0, B0, tmp);
126         fill(B0 + t, B0 + p, 0);
127         fnt.dft(B0, p);
128
129         reverse_copy(A, A + n, A0);
130         fill(A0 + t, A0 + p, 0);
131         fnt.dft(A0, p);
132
133         for(int i = 0; i != p; ++i)
134             A0[i] = A0[i] * B0[i] % mod_v;
135         fnt.idft(A0, p);
136         reverse(A0, A0 + t);
137         copy(A0, A0 + t, D);
138
139         for(p = 1; p < n; p <= 1);
140         fill(A0 + t, A0 + p, 0);
141         fnt.dft(A0, p);
142         copy(B, B + m, B0);
143         fill(B0 + m, B0 + p, 0);
144         fnt.dft(B0, p);
145         for(int i = 0; i != p; ++i)
146             A0[i] = A0[i] * B0[i] % mod_v;
147         fnt.idft(A0, p);
148         for(int i = 0; i != m - 1; ++i)
149             R[i] = ((A[i] - A0[i]) % mod_v + mod_v) % mod_v;
//fill(R + m - 1, R + p, 0);
150     return m - 1;

```

```

151     }
152 }
153
154 // 多项式的点值计算
155 // l 和 r 为存储要求的点数组的左右边界（左闭右开），idx 为除数多项式索引（初始 0）
156 // polycoef 为用于计算点的多项式系数，num 为其系数个数
157 // 设多项式项数 x=num, 点数 y=r-l, n=max(x,y), 复杂度 O(n(logn)^2)
158 void polynomial_calculator(int l, int r, int idx, int num, ll *polycoef) {
159     int mid = (l + r) / 2;
160     int lidx = (idx << 1) + 1, ridx = lidx + 1;
161     int lsize = poly_divisor[lidx].size(), rsize = poly_divisor[ridx].size();
162     ll *lmod_poly = new ll[lsize - 1], *rmod_poly = new ll[rsize - 1];
163     copy(poly_divisor[lidx].begin(), poly_divisor[lidx].end(), a);
164     copy(poly_divisor[ridx].begin(), poly_divisor[ridx].end(), b);
165     int lplen = polynomial_division(num, lsize, polycoef, a, d, lmod_poly);
166     int rplen = polynomial_division(num, rsize, polycoef, b, d, rmod_poly);
167     if (l == mid - 1) {
168         v[l] = mod_trans(lmod_poly[0]);
169     } else {
170         polynomial_calculator(l, (l + r) / 2, lidx, lplen, lmod_poly);
171     }
172     if (r == mid + 1) {
173         v[(l + r) / 2] = mod_trans(rmod_poly[0]);
174     } else {
175         polynomial_calculator((l + r) / 2, r, ridx, rplen, rmod_poly);
176     }
177     delete []lmod_poly;
178     delete []rmod_poly;
179 }
180
181 // 拉格朗日插值：二分+快速数论变换
182 // l 和 r 为存储要求的点数组的左右边界（左闭右开），idx 为由点二分构造出多项式的索
183 // 引（初始 0）
184 // polycoef 为插值得到的多项式结果
185 // 设点个数为 n, 复杂度 O(n*(logn)^2), 结果 polycoef 为 n-1 次多项式
186 void polynomial_interpolate(int l, int r, int idx, ll *polycoef) {
187     if (l == r - 1) {
188         polycoef[0] = ycoor[l] * mod_pow(v[l], mod_v - 2, mod_v) % mod_v;
189     } else {
190         int mid = (l + r) >> 1;
191         int lidx = (idx << 1) + 1, ridx = lidx + 1;
192         int sz = poly_divisor[idx].size() - 1;
193         int lsize = poly_divisor[lidx].size(), rsize =
194             poly_divisor[ridx].size();
195         int p = 1;
196         while (p < sz) p <<= 1;

```

```

197     polynomial_interpolate(mid, r, ridx, rightpoly);
198     copy(poly_divisor[lidx].begin(), poly_divisor[lidx].end(), a);
199     copy(poly_divisor[ridx].begin(), poly_divisor[ridx].end(), b);
200     fill(leftpoly + lsize - 1, leftpoly + p, 0);
201     fill(rightpoly + rsize - 1, rightpoly + p, 0);
202     fill(a + lsize, a + p, 0);
203     fill(b + rsize, b + p, 0);
204     fnt.dft(leftpoly, p);
205     fnt.dft(b, p);
206     for (int i = 0; i < p; i++) leftpoly[i] = leftpoly[i] * b[i] % mod_v;
207     fnt.idft(leftpoly, p);
208     fnt.dft(rightpoly, p);
209     fnt.dft(a, p);
210     for (int i = 0; i < p; i++) rightpoly[i] = rightpoly[i] * a[i] % mod_v;
211     fnt.idft(rightpoly, p);
212     for (int i = 0; i < sz; i++) polycoef[i] = mod_trans((leftpoly[i] +
213         → rightpoly[i]) % mod_v);
214     delete []leftpoly;
215     delete []rightpoly;
216 }
217
218 // 初始化点个数到二分子多项式个数
219 // 调用 polynomial_calculator 和 polynomial_interpolate 前调用
220 void init(int vnum) {
221     int vnum2 = 1;
222     while (vnum2 < vnum) vnum2 <<= 1;
223     for (int i = 0; i < 2 * vnum2 - 1; i++)
224         → poly_divisor.push_back(vector<ll>());
225 } poly;

```

用法：

```

1 // 多项式除法 & 取模
2 poly.fnt.init(1 << MAXL);
3 for(int i = 0; i < n; ++i)
4     cin >> poly.a[i]; // 0 次幂系数开始输入， 缺失的幂系数输入 0
5 for (int i = 0; i < m; i++)
6     cin >> poly.b[i]; // 0 次幂系数开始输入
7 int rlen = poly.polynomial_division(n, m, poly.a, poly.b, poly.d, poly.r);
8 for (int i = 0; i < n - m + 1; i++)
9     cout << poly.d[i] << " ";           // 商
10    for (int i = 0; i < rlen; i++)
11        cout << poly.r[i] << " ";           // 余数
12
13 // 多点求值
14 poly.fnt.init(1 << MAXL);

```

```

15 int n, vnum;
16 cin >> n;
17 for (int i = 0; i < n; i++)
18     cin >> poly.coef[i];           // 多项式系数
19 cin >> vnum;
20 for (int i = 0; i < vnum; i++)
21     cin >> poly.xcoor[i];        // 要求的点  $x_i$ 
22 poly.init(vnum);
23 poly.binary_subpoly(0, vnum / 2, 1), poly.binary_subpoly(vnum / 2, vnum, 2);
24 poly.polynomial_calculator(0, vnum, 0, n, poly.coef);
25 for (int i = 0; i < vnum; i++)
26     cout << poly.v[i] << " ";
27
28 // 快速插值
29 poly.fnt.init(1 << MAXL);
30 int vnum;
31 cin >> vnum;                  // 点的个数
32 for (int i = 0; i < vnum; i++)
33     cin >> poly.xcoor[i] >> poly.ycoor[i];
34 poly.init(vnum);
35 poly.binary_subpoly(0, vnum, 0);
36 for (unsigned int i = 1; i < poly.poly_divisor[0].size(); i++) {
37     poly.mcoef[i - 1] = poly.poly_divisor[0][i] * i % mod_v;
38 }
39 // 遍历 i 计算所有  $\sum_{j \neq i} x_i - x_j$ 
40 poly.polynomial_calculator(0, vnum, 0, poly.poly_divisor[0].size() - 1,
41     → poly.mcoef);
42 // 拉格朗日插值计算多项式
43 poly.polynomial_interpolate(0, vnum, 0, poly.coef);
44 // 输出插值得到的多项式系数
45 for (int i = 0; i < vnum; i++)
46     cout << poly.coef[i] << " ";
47 cout << endl;
48 // 将输入点代入插值得到的多项式中进行验证，输出计算得到的结果
49 poly.polynomial_calculator(0, vnum, 0, vnum, poly.coef);
50 for (int i = 0; i < vnum; i++)
51     cout << poly.v[i] << " ";

```

# Chapter 7

## 组合数学

### 7.1 常见公式和经典问题

- 组合数公式:  $C_n^m = \frac{n!}{m!(n-m)!}$
- 排列数公式  $A_n^m = C_n^m \cdot m! = \frac{n!}{(n-m)!}$
- 二项式定理  $(x + a)^n = \sum_{k=0}^n C_n^k x^k a^{n-k}$
- 从  $n$  个物品中可重复取得  $k$  个的方案数:  $n^k$
- 从  $n$  个物品中不可重复取  $k$  个做排列的方案数:  $C_n^k \cdot k!$
- 从  $n$  个物品中不可重复取  $k$  个做圆排列的方案数:  $\frac{C_n^k \cdot k!}{m}$
- $n$  个物品中, 第  $i$  种物品有  $k_i$  个, 且  $\sum_{i=1}^m k_i = n$ , 它的所有排列种数为  $\frac{n!}{k_1!k_2!\dots k_m!}$
- 从  $n$  个物品中可重复地选  $k$  个做组合的方案数为  $C_{n+k-1}^k$
- 从  $\{1, 2, 3, \dots, n\}$  中选  $k$  个不相邻的数做组合的方案数  $C_{n-k+1}^k$

#### 7.1.1 经典恒等式

- $\sum_{i=0}^n C_n^i = 2^n$
- $\sum_{i=0}^n (-1)^i C_n^i = 0$
- $\sum_{i=0}^n 2^i C_n^i = 3^n$

#### 7.1.2 容斥原理

$$|\cup_{i=1}^n A_i| = \sum_{O \subseteq B} (-1)^{|O|-1} |\cap_{e \in O} e|$$

## 7.2 询问排列数、组合数

```
1  ll fac[maxn], facinv[maxn];
2  void pretreat() {
3      fac[0] = facinv[0] = 1;
4      for (int i = 0; i < maxn; i++)
5          fac[i] = (fac[i] * fac[i - 1]) % mod;
6      facinv[maxn - 1] = inv(fac[maxn - 1]);
7      for (int i = maxn - 2; i > 0; i--)
8          facinv[i] = (facinv[i + 1] * (i + 1)) % mod;
9  }
10 ll C(int n, int m) {
11     if (m > n || m < 0)
12         return 0;
13     return (((fac[n] * facinv[m]) % mod) * facinv[n-m]) % mod;
14 }
15 ll P(int n, int m) {
16     if (m > n || m < 0)
17         return 0;
18     return (fac[n] * facinv[m]) % mod;
19 }
```

## 7.3 卡特兰数

**常见模型** 进出栈序列，括号序列匹配的计数等。

**递推公式**  $f(n) = \sum_{k=0}^{n-1} f(k) \cdot f(n-k-1)$ ,  $f(0) = 1$ .

**结论公式**  $f(n) = C_{2n}^n - C_{2n}^{n+1} = \frac{1}{n+1} C_{2n}^n$ .

```
1  double getCatalan(int x) {
2      double res = 1.0;
3      for (int i = x + 1; i <= 2 * x; i++)
4          res *= ((double)i / (double)(i-x));
5      return (res / (x + 1));
6  }
```

## 7.4 斯特林数

### 7.4.1 第一类斯特林数

**定义** 第一类斯特林数表示将  $n$  个不同元素构成  $m$  个圆排列的数目，可以把“ $m$  个圆”理解为“ $m$  个非空的循环队列”。具有如下性质：

- $s_1(n, 0) = 0$
- $s_1(n, n) = 1$
- $s_1(n, 1) = (n - 1)!$
- $s_1(n, n - 1) = C(n, 2)$
- $s_1(n, 2) = (n - 1)! \cdot \sum_{i=1}^{n-1} \frac{1}{i}$

根据上述性质可以得到第一类斯特林数的递推式： $s_1(n, k) = s_1(n - 1, k - 1) + s_1(n - 1, k) \times (n - 1)$ ,  $n, k \geq 1$ .

```
1 | s[1][1] = 1;
2 | for (int i = 2; i < MAXN; i++) {
3 |     s[i][0] = 0;
4 |     for (int j = 1; j < i; j++) {
5 |         s[i][j] = s[i-1][j-1] + (i-1) * s[i-1][j];
6 |     }
7 |     s[i][i] = 1;
8 | }
```

### 7.4.2 第二类斯特林数

**定义** 第二类斯特林数表示将  $n$  个不同的元素分成  $m$  个非空集合的方案数。具有如下性质：

- $S(n, 0) = 0, n \geq 0$
- $S(n, 1) = 1$
- $S(n, n) = 1$
- $S(n, n - 1) = C(n, 2)$
- $S(n, 2) = S(n - 1, 1) + S(n - 1, 2) \times 2 = 2^{n-1} - 1$
- $S(n, n - 2) = C(n, 3) + 3 \cdot C(n, 4)$
- $S(n, 3) = \frac{1}{2}(3^{n-1} + 1) - 2^{n-1}$
- $S(n, n - 3) = C(n, 4) + 10 \cdot C(n, 5) + 15 \cdot C(n, 6)$

## 常见的问题模型

- 场景 1:  $n$  个不同的球, 放入  $m$  个无区别的盒子, 不允许盒子为空, 方案数  $ans = S(n, m)$ 。
- 场景 2:  $n$  个不同的球, 放入  $m$  个有区别的盒子, 不允许盒子为空, 方案数  $ans = S(n, m) \times A(m, m)$ , 在第二类斯特林数的基础上乘上  $m$  的全排列即可, 即  $ans = m! \cdot S(n, m)$ 。
- 场景 3:  $n$  个不同的球, 放入  $m$  个无区别的盒子, 允许盒子为空, 方案数  $ans = \sum_{k=0}^m S(n, k)$ , 这个也比较容易理解。
- 场景 4:  $n$  个不同的球, 放入  $m$  个有区别的盒子, 允许盒子为空, 方案数  $ans = \sum_{k=0}^m A(m, k) \cdot S(n, k)$ , 因为盒子也是有区别的, 所以这里不是乘上  $k!$ , 而是乘上  $A(m, k)$ .

```
1 s[1][1] = 1;
2 for (int i = 2; i < MAXN; i++) {
3     s[i][0] = 0;
4     for (int j = 1; j < i; j++) {
5         s[i][j] = s[i-1][j-1] + (j) * s[i-1][j];
6     }
7     s[i][i] = 1;
8 }
```

## 7.5 Lucas 定理和扩展 Lucas

**Lucas 定理**  $C_n^m \bmod p = C_{\frac{n}{p}}^{\frac{m}{p}} \cdot C_{n \bmod p}^{m \bmod p} \bmod p$ , 即  $\text{lucas}(n, m) \bmod p = \text{lucas}(\frac{n}{p}, \frac{m}{p}) \cdot C_{n \bmod p}^{m \bmod p} \bmod p$ . 当  $m = 0$  时,  $\text{lucas}(n, m, p)$  返回 1.

**应用条件** Lucas 定理应用前提:  $p$  为质数。

```
1 void init(1l p) {
2     fac[0] = 1ll;
3     for (int i = 1; i <= p; i++)
4         fac[i] = fac[i-1] * (1l)i % p;
5 }
6 1l C(1l n, 1l m, 1l p) {
7     if (m > n)
8         return OLL;
9     return fac[n] * inv(fac[m] * fac[n - m], p) % p;
10    /* n, m 较大的时候不能打表, 直接计算 */
11 }
12 1l lucas(1l n, 1l m, 1l p) {
13     if (m == 0)
14         return 1LL;
15     return C(n % p, m % p, p) * lucas(n / p, m / p, p) % p;
16 }
```

**扩展 Lucas 定理** 当  $n, m$  较大且  $p$  不为质数的时候，令  $p = p_1^{k_1} \cdot \dots \cdot p_n^{k_n}$ ，列出同余方程组：

$$\begin{cases} ans \equiv c_1 \pmod{p_1^{k_1}} \\ ans \equiv c_2 \pmod{p_2^{k_2}} \\ \dots \\ ans \equiv c_n \pmod{p_n^{k_n}} \end{cases}$$

其中  $c_1 \dots c_n$  是对于每一个  $C_n^m \pmod{p_i^{k_i}}$  求出的答案，然后使用中国剩余定理合并即可。

## 7.6 Polya 定理

设  $G$  是  $p$  个对象的一个置换群，用  $m$  种颜色涂染  $p$  个对象，则不同的染色方案为  $L = \frac{1}{G}(m^{c(g_1)} + m^{c(g_2)} + \dots + m^{c(g_n)})$ ，其中  $G = \{g_1, g_2, \dots, g_s\}$ ， $c(g_i)$  为置换  $g_i$  的循环节数目。

例题：给定一个  $n$  个点， $n$  条边的环，有  $n$  种颜色，给每个顶点染色，问有多少种本质不同的染色方案，答案对  $1e9 + 7$  取模。注意本题的本质不同，定义为：只需要不能通过旋转与别的染色方案相同。

```
1 long long n, m, divisor[MAXN];
2 int cnt = 0;
3 void getDivisor(long long x) {
4     cnt = 0;
5     long long i = 1LL;
6     for (; i * i < x; i++)
7         if (x % i == 0LL)
8             divisor[cnt++] = i, divisor[cnt++] = x / i;
9     if (i * i == x)
10        divisor[cnt++] = i;
11 }
12 long long euler(long long n) {
13     long long res = n, a = n;
14     for (long long i = 2LL; i * i <= a; i++) {
15         if (a % i == 0) {
16             res = res / i * (i-1);
17             while (a % i == 0)
18                 a /= i;
19         }
20     }
21     if (a > 1LL)
22         res = res / a * (a-1);
23     return res;
24 }
25 long long cg(long long x) {
26     return __gcd(n, x);
27 }
28 void solve(int n) {
```

```
29     getDivisor(n);
30     long long ans = OLL;
31     for (int i = 0; i < cnt; i++)
32         ans = (ans + euler(divisor[i]) * mod_pow(m, cg(n / divisor[i]))) % MOD;
33     ans = ans * inv(n);
34 }
```

# Chapter 8

## 动态规划

### 8.1 背包问题

#### 8.1.1 0-1 背包（每种物品只有一个）

状态转移方程  $f(i, j)$  表示背包已用容量为  $j$  时考虑第  $i$  件物品装或不装能获得的最大价值。

$$f(i, j) = \max\{f(i - 1, j), f(i - 1, j - v_i) + w_i\}$$

```
1 | for (i = 0; i < m; i++)
2 |   for (j = t; j >= cost[i]; j --) // 这里必须逆序枚举 ~
3 |     dp[j] = max(dp[j], dp[j - cost[i]] + value[i]);
```

#### 8.1.2 完全背包（每种物品无限多个）

在上文 0-1 背包的基础上将  $j$  改为正向枚举即可，这样每种物品就可以被拿多次。

```
1 | for (int i = 1; i <= M; i++)
2 |   for (int j = cost[i]; j <= T; j++)
3 |     dp[j] = max(dp[j], dp[j - cost[i]] + val[i]);
```

#### 8.1.3 多重背包（每种物品有有限多个）

使用二进制思想将物品个数拆分为 2 的幂次之和，然后使用 0-1 背包解决。

```
1 | struct Item {
2 |   int v, w;
3 | } items[MAXN];
4 | int dp[MAXV], cnt = 1;
5 | for (int i = 0; i < m; i++) {
6 |   int c = 1, v, w, n;
```

```

7     scanf("%d%d%d", &v, &w, &n);
8     while (n - c > 0) {
9         n -= c;
10        items[cnt++] = (Item) { c * v, c * w };
11        c *= 2
12    }
13    item[cnt++] = (Item) { k * v, k * w };
14 }
15 memset(dp, 0, sizeof dp);
16 for (int i = 1; i < cnt; i++)
17     for (int j = cap; j >= items[i].w; j--)
18         dp[j] = max(dp[j], dp[j - items[i].w] + items[i].v);

```

### 8.1.4 混合背包 (有的物品有限, 有的物品无限)

分别处理, 根据当前物品的类型, 变更第二维 (容量维) 的枚举顺序即可: 如果是有限物品, 倒序枚举容量; 如果是无限物品, 正序枚举容量。再加上多重背包的情况也是一样的。

### 8.1.5 二维费用背包问题

**定义** 二维背包问题是指: 对于每件物品, 具有两种不同的费用; 两种费用分别对应不同的可付出的最大值 (容量), 求物品的最大价值。设第  $i$  件物品所需的两种费用分别为  $c_i, d_i$ , 价值为  $w_i$ .

**特殊限制** 如果题目限制“最多只能取  $k$  件物品”, 则可以将可取的物品件数也视为费用, 每个物品的费用均为 1.

**转移方程**  $f(i, v, u) = \max\{f(i - 1, v, u), f(i - 1, v - c_i, u - d_i) + w_i\}$

### 8.1.6 分组背包

**定义** 有  $N$  件物品被划分为  $K$  组, 每组的物品互相冲突, 最多可以选一件; 求最大的价值和。

**转移方程**  $f(k, v)$  表示前  $k$  组物品花费  $v$  容量取得的最大权值:  $f(k, v) = \max\{f(k - 1, v), f(k - 1, v - c_i) + w_i | i \in group(k)\}$

```

1 for (int k = 0; k < tot; k++)
2     for (int v = cap; v >= 0; v--)
3         for (int i = 0; i < type[k].size(); i++)
4             if (v >= type[k][i].cost)
5                 f[v] = max(f[v], f[v-type[k][i].cost] + type[k][i].value);

```

## 8.2 最长公共子序列 (LCS)

状态转移方程：

$$f(i, j) = \max \begin{cases} f(i - 1, j) \\ f(i, j - 1) \\ f(i - 1, j - 1) + 1, A[i] = B[j] \end{cases}$$

```
1 | for (int i = 1; i <= n; i++)  
2 |     for (int j = 1; j <= n; j++) {  
3 |         dp[i][j] = max(dp[i-1][j], dp[i][j-1]);  
4 |         if (a[i-1] == b[j-1])  
5 |             dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);  
6 |     }
```

## 8.3 最长上升子序列

**朴素做法**  $O(n^2)$ :  $f(i)$  表示以  $a_i$  结尾的 LIS 长度，则有状态转移方程  $f(i) = \max\{f(j)\} + 1, 1 \leq j < i$ .

**优化做法**  $O(n \log n)$

- 设置一个单调栈（满足栈底到栈顶的元素单调递增） $s$ , 然后将第一个元素加入栈中。
- 接下来开始逐个加入数列中的元素，设当前待入栈的元素为  $a_i$ . 若  $a_i >$  栈顶元素  $s[\text{top}]$ , 则直接让  $a_i$  入栈.
- 若  $a_i \leq$  栈顶元素  $s[\text{top}]$ , 则在栈中二分查找到第一个小于等于  $a_i$  的元素的位置  $\text{pos}$ , 将  $s[\text{pos}]$  替换为  $a_i$ .
- 重复上述步骤，直至所有数都被处理完成.
- 此时栈中的元素个数  $s.size$  即为 LIS 的答案，但注意栈中元素并不是组成 LIS 的元素。

**输出方案** 用  $lower$  数组保存最长上升子序列的长度，设置一个  $pos2$  数组，记录一下数组  $a$  中每个元素在  $lower$  数组中出现的位置；然后从数组  $a$  的最后一个元素开始到第一个元素寻找最长上升子序列。

```
1 // 写法 1, 不需要输出方案  
2 int stk[MAXN], top = 0;  
3 vector<int> a;  
4 stk[top] = a[0];  
5 for (int i = 1; i < a.size(); i++)  
6     if (a[i] > stk[top])  
        stk[++top] = a[i];  
    else {
```

```

9      int pos = lower_bound(stk, stk + top + 1, a[i]) - stk;
10     stk[pos] = a[i];
11 }
12 int ans = top + 1;           // stk.size 即为答案

```

```

1 // 写法 2, 可以输出方案
2 int pos[maxn], answer[maxn];
3 void lis(int a[], int lower[], int n){
4     lower[1] = a[1], pos[1] = 1;
5     int index = 1;
6     for(int i = 2; i <= n; i++) {
7         if(a[i] >= lower[index])
8             lower[++index] = a[i], pos[i] = index;
9         else {
10             int p = upper_bound(lower + 1, lower + index + 1, a[i]) - lower - 1;
11             lower[p] = a[i], pos[i] = p;    // 记录原数组中每个元素在 lower 数组中出
12             ↳ 现的位置
13         }
14     }
15     int maxx = inf;
16     for(int i = n; i >= 1; i--) {
17         if(index == 0)
18             break;
19         if(pos[i] == index && maxx > a[i])
20             answer[index] = i, index--, maxx = a[i];
21     }
}

```

## 8.4 最小编辑距离

**状态表示**  $dp[i][j]$  表示 A 串从第 0 个字符开始到第  $i$  个字符、和 B 串从第 0 个字符开始到第  $j$  个字符，这两个字串的编辑距离（字符串的下标从 1 开始）。

```

1 int dp[maxn][maxn];
2 char a[maxn], b[maxn];
3 int editDistance() {
4     int len1 = strlen(a + 1), len2 = strlen(b + 1);
5     memset(dp, 0x3f, sizeof dp);
6     for(int i = 1; i <= len1; i++)
7         dp[i][0] = i;
8     for(int j = 1; j <= len2; j++)
9         dp[0][j] = j;
10    for(int i = 1; i <= len1; i++) {
11        for(int j = 1; j <= len2; j++) {
12            int flag = a[i] != b[j];

```

```

13         dp[i][j] = min(dp[i-1][j] + 1, min(dp[i][j-1] + 1, dp[i-1][j-1] +
14             → flag));
15     }
16     return dp[len1][len2];
17 }
```

## 8.5 最大连续子段和

状态转移方程:  $f(i) = \max\{f(i-1) + a_i, a_i\}$ .

## 8.6 区间 DP

第一维枚举区间长度 k, 第二维枚举左端点 l。根据区间长度和左端点算出右端点 r=l+k-1, 然后第三维枚举区间 [l,r) 上的分界点 x. 状态转移方程:

$$f(l, r) = \text{better}\{f(l, r), f(l, x) + f(x + 1, r)\}$$

```

1 for (int k = 2; k <= n; k++) {
2     for (int l = 1; l + k - 1 <= n; l++) {
3         int r = l + k - 1;
4         for (int j = l; j < r; j++)
5             if (dp[l][j] == dp[j+1][r]) {
6                 dp[l][r] = std::max(dp[l][r], dp[l][j] + 1);
7                 ans = std::max(ans, dp[l][r]);
8             }
9     }
```

## 8.7 状态压缩 DP

操作	代码
取出整数 n 在二进制表示下的第 k 位	$(n >> k) \& 1$
取出整数 n 在二进制表示下的后 k ( $0 \leq k \leq \log_2 n$ ) 位	$n \& ((1 << k) - 1)$
把整数 n 在二进制表示下的第 k 位取反	$n^{(1 << k)}$
对整数 n 在二进制表示下的第 k 为赋值 1	$n   (1 << k)$
对整数 n 在二进制表示下的第 k 为赋值 1	$n \& ( (1 << k) )$

Table 8.1: 二进制的常用操作

## 8.8 数位 DP

**问题场景** 处理出某一区间  $[l, r]$  范围内，满足条件的数的个数。

### 一般解法

- **数位处理**: 处理不多于  $i$  位的数中，有多少个数满足条件，用  $dp[i]$  表示。
- **状态拓展**: 大部分题目与数字本身有一定的关系（不能出现/必须出现特定数字），则状态数组需要多加一维/多维进行转移： $dp[i][j]$  表示第  $i$  位数字为  $j$  且满足条件的数字个数。
- **处理区间端点**: 用预处理的信息计算出  $0 \sim r$  和  $0 \sim l - 1$  闭区间满足条件的数字个数，然后求解  $[l, r]$  范围的答案，即为  $r$  端的答案减去  $l - 1$  端的答案。
- **具体实现**: 首先在第当前考虑的第  $i$  位上固定一个数字，那么后面就可以随便填。

### DFS 函数的参量

- 基本量：数位数  $pos$ ，最高位限制  $lim$
- 判断前导 0 的标志： $lead$
- 一般需要记录数位中的前一位（或前几位）： $pre$
- 其它用于区分状态的参量

**最高位标记** 当给定的区间  $[0, r]$  不同时，数位 DP 统计时的位数限制也不同，如  $r = 1234$  那么，当第一位为 1 的时候，第二位的取值范围  $[0, 2]$ ；当第一位为 0 的时候第二位则可以取  $[0, 9]$ 。为了分清这样的情况，引入  $lim$  变量：

- 若当前位  $lim = 1$ ，且已经取到了当前位可以取得的最大数字，则下一位  $lim = 1$
- 若当前位  $lim = 1$ ，但取到的数字小于可以取得的最大数字，则下一位  $lim = 0$
- 若当前位  $lim = 0$ ，则下一位  $lim = 0$

**记忆化搜索** DFS 的时候，可以将已经搜索过的状态记录下来，那么下一次在遇到**完全相同**的状态的时候，就可以直接返回；所以 DP 数组的维度需要和 DFS 函数的参量（除了  $lim, lead$ ）相同。

**DFS 参量完全相同**是状态相同的必要非充分条件。需要注意的是  $lim = 1$  和  $lead = 1$  的时候不能直接取记忆值：

```
1  typedef long long ll;
2  ll dp[MAXL][MAXD];
3
4  /**
5   * pos: 当前枚举的位
6   * pre: 之前的状态，例如上一位，视需要的状态不同可能有不同的保存方式
```

```

7  * lead: 是否有前导 0
8  * limit: 当前位是否有限制
9  */
10
11 dfs(int pos, int pre, int lead, int limit) {
12     // 递归边界, 返回 1 表示枚举得当前数合法
13     if (pos == -1)
14         return 1;
15     // 如果 lim 和 lead 都不为 true, 满足记忆化条件则可以返回
16     if (!limit && !lead && dp[pos][pre])
17         return dp[pos][pre];
18     int cur = 0;
19     // 枚举当前位
20     int maxd = limit ? a[pos] : 9;
21     for (int i = 0; i <= maxd; i++) {
22         // 需要针对多种情况具体分析, 例如是否有限制, 是否有前导 0 的影响等等
23         if (lead && !i)
24             cur += dfs(pos - 1, i, true, i == maxd && limit);
25         else if (lead && !i)
26             cur += dfs(pos - 1, i, true, i == maxd && limit);
27         else if (....)
28             cur += ...
29     }
30     // 记忆化存下结果
31     if (!limit && !lead)
32         dp[pos][pre] = cur;
33     return cur;
34 }
35
36 solve(l1 x) {
37     int pos = 0, init_state = ?;           // init_state 表示初始位之前的状态
38     // 拆数
39     while (x) {
40         a[pos++] = x % 10;
41         x /= 10;
42     }
43     return dfs(pos - 1, init_state, true, true);
}

```

## 8.9 斜率优化

- 形如  $f(i) = \min\{f(j) + k(j) \cdot t(i)\}$  这样的最值型决策, 与先前决策相关, 又包含  $i, j$  元素乘积时, 可以使用斜率优化。
- 在决策  $i$  时, 考虑  $i$  先前的两个决策  $x, y (x < y)$ , 且决策  $x$  优于决策  $y$ , 也就是  $f(x) + k(x) \cdot t(i) < f(y) + k(y) \cdot t(i)$ .

- 移项得到  $f(x) - f(y) < t(i) \cdot (k(y) - k(x))$ , 设  $k(i)$  单调 (如单调递减), 则令斜率  $slope(x, y) = \frac{f(x)-f(y)}{k(y)-k(x)}$
- 如果满足  $slope(x, y) < t(i)$ , 则决策时选择  $x$  比选择  $y$  更优。如果  $k$  是有序的, 则维护一个保存下标、关于斜率的单调 (递增) 队列, 满足:
  1. 如果队首的两个元素斜率  $slope(l, l + 1)$  满足  $slope(l, l + 1) \leq t(i)$ , 则队首元素就是最优决策点。
  2. 将当前决策点  $i$  加入队尾时, 如果  $slope(r - 1, r) \geq slope(r - 1, i)$ , 则删除队尾元素后再插入。

示例代码:

```

1  inline double slope(int i, int j) {
2      return 1.0 * (dp[j] - dp[i]) / (land[i + 1].h - land[j + 1].h);
3  }
4  for (int i = 1; i <= cnt; i++) {
5      while (l < r && slope(q[l], q[l + 1]) <= land[i].w)
6          l++;
7      dp[i] = dp[q[l]] + 1LL * land[i].w * land[q[l + 1]].h;
8      while (l < r && slope(q[r - 1], q[r]) >= slope(q[r - 1], i))
9          r--;
10     q[++r] = i;
11 }
```

# Chapter 9

## 计算几何

- 因为计算几何涉及大量浮点计算，为了减少浮点误差引发的悲剧，在比较和计算的时候需要引入  $\epsilon$  精度误差。
- 比较的时候，引入多态函数  $dcmp(x)$ ，某个浮点数  $x$  的绝对值小于  $\epsilon$  返回 0；如果  $x < 0$  返回 -1，否则返回 1。
- 点与向量在 C++ 中的运算和定义是相同的。

### 9.1 二维几何基本定义

```
1 double eps = 1e-10; // epsilon, accuracy tolerance
2
3 /* fix accuracy when comparing */
4 int dcmp (double x) {
5     if (fabs(x) < eps)
6         return 0;
7     else
8         return x < 0 ? -1 : 1;
9 }
10
11 struct Vector {
12     double x, y;
13     /* constructor function */
14     Vector() {}
15     Vector(double x = 0, double y = 0) : x(x), y(y) {}
16     /* reload operator */
17     Vector operator + (Vector B) {
18         return Vector(x + B.x, y + B.y);
19     }
20     Vector operator - (Vector B) {
21         return Vector(x - B.x, y - B.y);
22     }
23     Vector operator * (double p) {
```

```

24     return Vector(x * p, y * p);
25 }
26 Vector operator / (double p) {
27     if (p < eps)
28         p += eps;
29     return Vector(x / p, y / p);
30 }
31 /* comparison */
32 bool operator < (const Vector B) const {
33     return (dcmp(x - B.x) == -1) || (dcmp(x - B.x) == 0 && dcmp(y - B.y) ==
34         -1);
35 }
36 bool operator == (const Vector B) const {
37     return (dcmp(x - B.x) == 0) && (dcmp(y - B.y) == 0);
38 }
39 }
40
41 typedef Vector Point; // the definitions of vector and point are the same.

```

## 9.2 基本运算

### 9.2.1 求向量的极角

向量的极角：从  $x$  轴的正半轴转到该向量方向所需要的角度。比较的时候有精度误差。

```

1 double getVectorPolarAngle(Vector a) {
2     return atan2(a.y, a.x);
3 }

```

### 9.2.2 点积、向量模、两向量夹角

```

1 double dot(Vector a, Vector b) {
2     return a.x * b.x + a.y * b.y;
3 }
4
5 double length(Vector a) {
6     return sqrt(dot(a, a));
7 }
8
9 double cosine(Vector a, Vector b) {
10    return dot(a, b) / length(a) / length(b);
11 }
12

```

```

13  double angle(Vector a, Vector b) {
14      return acos(cosine(a, b));
15  }

```

### 9.2.3 叉积，三角形有向面积

向量叉乘。二维向量  $\vec{a}, \vec{b}$  叉积  $\vec{a} \times \vec{b}$  得到的向量为  $(0, 0, x)$ ,  $x = A_x \cdot B_y - A_y \cdot B_x$ .

```

1  double cross(Vector a, Vector b) {
2      return a.x * b.y - a.y * b.x;
3  }

```

性质：

- 两个向量  $\vec{v}$  和  $\vec{w}$  的叉积等于  $\vec{v}, \vec{w}$  组成的三角形的有向面积的两倍。
- $\vec{v} \times \vec{w} = -\vec{w} \times \vec{v}$ .

**三角形的有向面积** 向量  $\vec{v}, \vec{w}$  构成的三角形，设有向面积为  $A = \frac{1}{2}(\vec{v} \times \vec{w})$ , 如果  $\vec{w}$  在  $\vec{v}$  的左边，则  $A > 0$ ; 如果  $\vec{w}$  在  $\vec{v}$  的右边，则  $A < 0$ .

以  $\vec{AB}, \vec{AC}$  构成的三角形的有向面积：

```

1  double area(Point A, Point B, Point C) {
2      return cross(B - A, C - A) / 2.0;
3  }

```

### 9.2.4 向量位置关系

判断向量  $\vec{a}, \vec{b}$  的位置关系：固定其中一个向量  $\vec{a}$  的方向，令其指向水平向右 ( $x$  轴正方向)；然后计算  $dot(\vec{a}, \vec{b})$  和  $cross(\vec{a}, \vec{b})$ ，将其符号分别对应点  $(x, y)$  参数的符号；按照直角坐标系下点的位置判断两向量的位置关系。

### 9.2.5 向量旋转、求向量的单位法向量

向量绕起点旋转：向量  $\vec{a} = (x, y)$ , 旋转  $\alpha$  角后得到新向量  $\vec{a}' = (x', y')$ , 有：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

```

1  Vector rotateVector(Vector a, double rad) {
2      return Vector(a.x * cos(rad) - a.y * sin(rad), a.x * sin(rad) + a.y *
3          ↳ cos(rad));
}

```

求单位法向量，旋转 90° 后除掉模即可，注意特判一下模长为 0 的情况。

```
1 Vector normalVector(Vector a) {
2     double len = length(a);
3     if (dcmp(len) == 0) {
4         return Vector(0, 0);           // ensure |a| != 0
5     }
6     return Vector(-a.y / len, a.x / len);
7 }
```

## 9.3 点与线

### 9.3.1 直线的表示方法

参数表示：可使用一点  $P_0$  和方向向量  $\vec{v}$  来表示直线；若已知直线两点  $A, B$ ，则  $\vec{v} = B - A$ . 直线上的点为  $P' = P_0 + \vec{v}t$ .

```
1 struct Line {
2     Point p;
3     Vector s;
4     double ang;
5     Line(Point p, Vector s): p(p), s(s) {
6         ang = atan2(s.y, s.x);
7     }
8     bool operator < (const Line L) const {
9         return ang < L.ang;
10    }
11};
```

### 9.3.2 两条直线的交点

将直线化为参数表示，设直线分别为  $P + t\vec{v}, Q + t\vec{w}$ ；向量  $\vec{u} = P - Q$ . 交点在第一条直线的参数为  $t_1$ ，第二条直线上的参数为  $t_2$ . 则交点参数公式为：

$$t_1 = \frac{\text{cross}(w, u)}{\text{cross}(v, w)}, t_2 = \frac{\text{cross}(v, u)}{\text{cross}(v, w)}$$

```
1 Point getLineIntersection(Line a, Line b) {
2     Vector u = a.p - b.p;
3     double t = cross(b.s, u) / cross(a.s, b.s);
4     return a.p + a.s * t;
5 }
```

### 9.3.3 点到直线的距离

在系数表示下，点  $P(x_0, y_0)$  到直线  $Ax + By + C = 0$  的距离  $d = \frac{|Ax_0 + By_0 + C|}{\sqrt{A^2 + B^2}}$ .

在参数表示下，可以在直线上取两点  $A, B$ ，作  $\vec{AB}$  和  $\vec{AP}$ ，两向量的叉积（平行四边形面积）除以向量  $\vec{AB}$  的模长即为点  $P$  到直线  $AB$  的距离。如果结果不取绝对值，则得到的是有向距离。

```
1 double distanceToLine(Point P, Point A, Point B) { // ab is on the line
2     Vector v1 = B - A, v2 = P - A;
3     return fabs(cross(v1, v2) / length(v1));
4 }
```

### 9.3.4 点到线段的距离

如果点  $P$  投影到线段  $AB$  所在的直线上得到的投影点在  $AB$  上，则所求距离即为  $P$  到  $AB$  的距离；如果  $Q$  在射线  $BA$  上，则所求距离就是  $PA$ ；反之所求距离就是  $PB$ 。因此首先要判断  $P$  与点  $A, B$  的位置关系，可以用上文的象限判断法。

```
1 double distanceToSegment(Point P, Point A, Point B) {
2     if (A == B) { // coincide
3         return length(A - P);
4     }
5     Vector v1 = B - A, v2 = P - A, v3 = P - B;
6     if (dcmp(dot(v1, v2)) < 0) // Q is on the left
7         return length(v2);
8     else if (dcmp(dot(v1, v3)) > 0)
9         return length(v3); // Q is on the right
10    else
11        return distanceToLine(P, A, B);
12 }
```

### 9.3.5 点在直线上的投影

设参数表示直线为  $A + t\vec{v}$ ；点  $P$  在直线  $AB$  上的投影点为  $Q$ （对应参数  $t_0$ ），那么  $Q = A + t_0\vec{v}$ . 根据  $PQ \perp AB$ ，有  $\vec{PQ} \cdot \vec{AB} = 0$ ，因此  $\vec{v} \cdot \vec{AP} - (\vec{v} \cdot \vec{v})t_0 = 0$ ，解得  $t_0 = \frac{\vec{v} \cdot \vec{AP}}{\vec{v}^2}$ .

```
1 Point pointProjection(Point P, Point A, Point B) {
2     Vector v = B - A;
3     return A + v * (dot(v, P - A) / dot(v, v));
4 }
```

### 9.3.6 线段相交判定

给定两条线段，判断是否相交。

**规范相交** 两条线段规范相交，当且仅当两条线段恰好有一个公共点，且不在任何一条线段的端点。

它的充要条件是：每条线段的两个端点都在两条线段的两侧。

端点与线段的位置关系可以用上文的象限判断法来判断。

```
1 bool checkIfSegmentProperIntersection(Point A1, Point B1, Point A2, Point B2) {
2     double c1 = cross(B1 - A1, A2 - A1), c2 = cross(B1 - A1, B2 - A1),
3         c3 = cross(B2 - A2, A1 - A2), c4 = cross(B2 - A2, B1 - A2);
4     return dcmp(c1) * dcmp(c2) < 0 && dcmp(c3) * dcmp(c4) < 0;
5 }
```

判断某个点  $P$  是否在线段  $AB$  上：

```
1 bool checkIfPointOnSegment(Point P, Point A, Point B) {
2     return dcmp(cross(A - P, B - P)) == 0 && dcmp(dot(A - P, B - P)) < 0;
3 }
```

两条线段的距离：

```
1 double getDist(Segment l1, Segment l2) {
2     double d1 = distToSegment(l1.p1, l2), d2 = distToSegment(l1.p2, l2),
3         d3 = distToSegment(l2.p1, l1), d4 = distToSegment(l2.p2, l1);
4     return min(min(d1, d2), min(d3, d4));
5 }
```

## 9.4 多边形

多边形的存储记录一般采用的是顺次记录其顶点的方法，如凸包的表示方法。方向一般为逆时针。

### 9.4.1 多边形的面积

任取  $n$  边形的一个点  $P_0$  为顶点，将该点与其它点连接，得到  $n - 2$  个三角形。利用叉积性质计算面积，求和累加后除 2 即可。因为叉积计算的是有向面积，所以两条边叉积得到的三角形，在多边形外面的那部分最后会抵消掉。

```
1 double polygonArea(Point p[], int n) {
2     double area = 0.0;
3     for (int i = 1; i < n-1; i++)
4         area += cross(p[i] - p[0], p[i+1] - p[0]);
5     return area / 2.0;
6 }
```

## 9.4.2 判断点在多边形内外

多边形的顶点必须以逆时针或顺时针顺序给出。

```
1 bool isPointInPolygon(Point p[], int n, Point a) {
2     int wn = 0;
3     for (int i = 0; i < n; i++) {
4         if (checkIfPointOnSegment(a, p[i], p[(i+1) % n]))
5             return true;
6         if (a == p[i])
7             return true;
8         int k = dcmp(cross(p[(i+1)%n] - p[i], a - p[i]));
9         int d1 = dcmp(p[i].y - a.y),
10            d2 = dcmp(p[(i+1)%n].y - a.y);
11         if (k > 0 && d1 <= 0 && d2 > 0)
12             wn++;
13         if (k < 0 && d2 <= 0 && d1 > 0)
14             wn--;
15     }
16     return wn != 0;
17 }
```

## 9.4.3 三角形的重心

三角形的重心是三条中线的交点。设  $AM, BN$  分别是三角形  $BC$  边和  $AC$  边上的中线，则三角形的重心  $G$  即为  $AM$  和  $BN$  的交点，满足公式  $G = \frac{A+B+C}{3}$ ，其中  $A + B + C$  表示坐标分量相加。

性质：重心是中线的一个三等分点。

```
1 Point triangleBaycenter(Point A, Point B, Point C) {
2     return Point((A.x + B.x + C.x) / 3.0, (A.y + B.y + C.y) / 3.0);
3 }
```

## 9.4.4 求质点组的重心

$n$  个质点，位置分别为  $A_1, A_2, \dots, A_n$ ；质量为  $m_1, m_2, \dots, m_n$ ；由重心公式有  $G = \frac{\sum_{k=1}^n m_k A_k}{\sum_{k=1}^n m_k}$ 。

```
1 Point getPollygonBaycenter(Point p[], double mass[], int n) { // not tested
2     Point t = Point(0.0, 0.0);
3     double sumMass = 0.0;
4     for (int i = 0; i < n; i++) {
5         t = t + p[i] * mass[i];
6         sumMass += mass[i];
7     }
8     if (dcmp(sumMass) == 0)      // judge
9         return Point(0, 0);
```

```
10     |     return t / sumMass;
11 }
```

## 9.5 圆

### 9.5.1 定义

```
1 struct Circle {
2     Point c;      // center
3     double r;    // radius
4     Circle(Point c, double r): c(c), r(r) {}
5     // 通过圆心角 rad 求圆上一点的坐标
6     Point pt(double rad) {
7         return Point(c.x + cos(rad) * r, c.y + sin(rad) * r);
8     }
9 };
```

### 9.5.2 直线 AB 与圆 C 的交点

- 方法 1：解方程组：设交点  $P = A + t(B - A)$ ，代入圆方程得到  $(at + b)^2 + (ct + d)^2 = r^2$ ，整理得到二次方程  $et^2 + ft + g = 0$ 。根据判别式区分相离、相切、相交的情况并得到交点。
- 方法 2：先求出圆心  $C$  在  $AB$  上的投影点  $P$ ，再求  $\vec{AB}$  对应的单位向量  $\vec{v}$ ，两个交点分别为  $p - L\vec{v}$  和  $p + L\vec{v}$ ， $L$  为  $P$  到交点的距离，可以由勾股定理算出。

注：第二种方法可以先求圆心到直线距离判断交点个数，再用勾股定理  $L^2 = r^2 - d^2$  算出  $L$ 。

```
1 int getLineCircleIntersection(Line l, Circle C, Point ans[]) {
2     double a = l.s.x, b = l.p.x - C.c.x, c = l.s.y, d = l.p.y - C.c.y;
3     double e = a * a + c * c, f = 2 * (a * b + c * d), g = b * b - d * d - C.r *
4         C.r;
5     double delta = f * f - 4 * e * g;
6     if (dcmp(delta) < 0)           // 0
7         return 0;
8     if (dcmp(delta) == 0) {        // 1
9         double t1 = -f / (2 * e);
10        ans[0] = C.pt(t1);
11        return 1;
12    }
13    double t1 = (-f - sqrt(delta)) / (2 * e),
14        t2 = (-f + sqrt(delta)) / (2 * e);      // 2
15    ans[0] = C.pt(t1), ans[1] = C.pt(t2);
```

```
15     return 2;
16 }
```

```
1 int getLineCircleIntersection2(Line l, Circle C, Point ans[]) {
2     Point proj = l.p + l.s * (dot(l.s, C.c - l.p) / dot(l.s, l.s));
3     Vector normal = normalVector(l.s);
4     double dist = square(proj.x - C.c.x) + square(proj.y - C.c.y);
5     if (dist > C.r)
6         return 0;
7     if (dist == C.r) {
8         ans[0] = proj;
9         return 1;
10    }
11    double len = sqrt(square(C.r) - dist);
12    ans[0] = proj + normal * len, ans[1] = proj - normal * len;
13    return 2;
14 }
```

### 9.5.3 两圆相交

若两圆有交点，则设其中一个交点为  $P_1$ ，圆心距为  $d$ ，根据余弦定理算出  $C_1C_2$  到  $C_1P_1$  的角  $da$  和  $C_1C_2$  的极角  $a$ ， $a \pm da$  角对应的点即为两个交点（或重合为一个交点）。

```
1 int getCirclesIntersection(Circle C1, Circle C2, Point ans[]) {
2     double dist = length(C1.c - C2.c);
3     if (C1.c == C2.c && dcmp(C1.r - C2.r) == 0)
4         return -1; // 两圆重合
5     if (dcmp(dist) == 0 || (dcmp(C1.r + C2.r - dist < 0)) || (dcmp(fabs(C1.r -
6         C2.r) - dist) > 0))
7         return 0; // 内离, 外离
8     Vector d = C2.c - C1.c;
9     double ang = atan2(d.y, d.x),
10    da = acos((C1.r * C1.r + dist * dist - C2.r * C2.r) / (2 * C1.r * dist));
11    ans[0] = C1.pt(ang - da), ans[1] = C1.pt(ang + da);
12    if (ans[0] == ans[1]) // 重合
13        return 1;
14    else
15        return 2;
}
```

### 9.5.4 过定点到圆的切线

首先计算点到圆心的距离判断是否存在切线和条数。如果点在圆上只有一条切线，切线即为点与圆心连线向量旋转 90 度。否则用反正弦函数可以求出切角  $ang$ ，将点与圆心连线分别顺时针和逆时针旋转  $ang$  即可得到两条切线。

```

1 int getTangents(Point p, Circle C, Vector v[]) {
2     Vector u = C.c - p;
3     double dist = length(u);
4     if (dist < C.r)
5         return 0;          // 点在圆内，没有切线
6     else if (dcmp(dist - C.r) == 0) {
7         v[0] = rotateVector(u, acos(-1) / 2.0);
8         return 1;
9     } else {
10        double ang = asin(C.r / dist);
11        v[0] = rotateVector(u, ang), v[1] = rotateVector(u, -ang);
12        return 2;
13    }
14 }
```

### 9.5.5 两圆的公切线

两圆的公切线有六种情况：

- 两圆完全重合，有无数条公切线。
- 两圆内含，没有公切线。
- 两圆内切，有一条公切线。
- 两圆相交，有两条公切线。
- 两圆外切，有三条公切线。
- 两圆相离，有四条公切线。

```

1 int getCirclesTangents(Circle A, Circle B, Point *a, Point *b)  {
2     int cnt = 0;
3     if (A.r < B.r) {
4         std::swap(a, b);
5         std::swap(A, B);
6     }
7     double d2 = lengthSquare(A.c - B.c),
8            rdiff = A.r - B.r, rsum = A.r + B.r;
9     if (d2 < rdiff * rdiff) {
10        return 0;          // 内含
11    }
12    double base = atan2(B.c.y - A.c.y, B.c.x - A.c.x);
13    if (d2 == 0 && dcmp(A.r - B.r) == 0)
14        return -1;         // 重合
15    if (d2 == rdiff * rdiff) {
16        a[cnt] = A.pt(base), b[cnt++] = B.pt(base);
```

```

17     return cnt;
18 }
19 // 相交的基本情况
20 double ang = acos((A.r - B.r) / sqrt(d2));
21 a[cnt] = A.pt(base + ang), b[cnt++] = B.pt(base + ang);
22 a[cnt] = A.pt(base - ang), b[cnt++] = B.pt(base - ang);
23 if (d2 == rsum * rsum) { // 外切
24     a[cnt] = A.pt(base), b[cnt++] = B.pt(base);
25 }
26 else if (d2 > rsum * rsum) { // 外离
27     ang = acos((A.r + B.r) / sqrt(d2));
28     a[cnt] = A.pt(base + ang), b[cnt++] = B.pt(base + ang);
29     a[cnt] = A.pt(base - ang), b[cnt++] = B.pt(base - ang);
30 }
31 return cnt;
32 }

```

### 9.5.6 两点/三点定圆

```

1 Circle getCircle(Point a, Point b) {
2     Point c = Point((a.x + b.x) / 2.0, (a.y + b.y) / 2.0);
3     double r = length(Vector(b - a)) / 2.0;
4     return (Circle) { c, r };
5 }
6
7 Circle threePointCircle(Point a, Point b, Point c) {
8     Point p = (a + b) / 2.0,
9     q = (a + c) / 2.0;
10    Vector v = rotate(b - a, PI / 2.0),
11        w = rotate(c - a, PI / 2.0);
12    Point o;
13    double r;
14    if (dcmp(v ^ w) == 0) { // parallel
15        if (dcmp(length(a - b) + length(b - c) - length(a - c)) == 0)
16            o = (a + c) / 2.0, r = length(c - o);
17        if (dcmp(length(a - b) + length(a - c) - length(b - c)) == 0)
18            o = (b + c) / 2.0, r = length(b - o);
19        if (dcmp(length(a - c) + length(b - c) - length(a - b)) == 0)
20            o = (a + b) / 2.0, r = length(a - o);
21        return (Circle){ o, r };
22    }
23    o = lineIntersection(p, v, q, w);
24    r = length(a - o);
25    return (Circle) { o, r };
26 }

```

### 9.5.7 最小覆盖圆

```
1 Circle MinCoveredCircle(Point p[], int n) {
2     random_shuffle(p, p + n);
3     Circle c = (Circle) {p[0], 0.0};
4
5     for (int i = 1; i < n; i++) {
6         if (checkPointInCircle(p[i], c))
7             continue;
8         c.c = p[i], c.r = 0.0;
9         for (int j = 0; j < i; j++) {
10            if (checkPointInCircle(p[j], c))
11                continue;
12            c = getCircle(p[i], p[j]);
13            for (int k = 0; k < j; k++) {
14                if (checkPointInCircle(p[k], c))
15                    continue;
16                c = threePointCircle(p[i], p[j], p[k]);
17            }
18        }
19    }
20    return c;
21 }
```

### 9.5.8 最小覆盖球的模拟退火算法

```
1 double solve() {
2     const double start = 1000;
3     double step = start, ans = inf, mt;
4     Point z;
5     z.x = z.y = z.z = 0;
6     int s = 0;
7     while (step > eps) {
8         for (int i = 0; i < n; i++)
9             if (dist(z, pts[s]) < dist(z, pts[i]))
10                 s = i;
11         mt = dist(z, pts[s]);
12         ans = min(ans, mt);
13         z.x += (pts[s].x - z.x) / start * step;
14         z.y += (pts[s].y - z.y) / start * step;
15         z.z += (pts[s].z - z.z) / start * step;
16         step *= 0.97;
17     }
18     return ans;
19 }
```

## 9.6 凸包 (Convex Hull)

**凸包的定义** 对于给定的一组点  $p_1, p_2, \dots, p_n$ , 凸包就是把给定点包围在内部的、面积最小的凸多边形。

### 9.6.1 Andrew 算法求凸包

算法思想：先将所有的点按照水平序排序，将第 1, 2 个点加入凸包，然后从第 3 个点开始逐个尝试将点加入栈中；若点不满足在凸包上的条件则将其出栈。点  $P$  在凸包上当且仅当新点在凸包前进方向的左边。

数学体现即为栈顶已有的点  $A_2, A_1$ , 当且仅当  $\vec{A_2 - A_1} \times \vec{P - A_1} \leq 0$  时， $P$  在  $A_2 - A_1$  的左边（可以通过上文的坐标系判断法判断位置）。

算法步骤：

- 先将所给的点去重，然后按照  $x$  的大小排序；如果  $x$  相同则按  $y$  的大小排序
- 先求下凸壳。将排序后的 1,2 个点加入栈中，从第 3 个点开始判断新点是否在已有凸包点的左侧，如果是的话加入栈中；
- 如果不是的话说明找到了更外围的点，将栈中的点删除直到栈中只有一个点或当前点在已有凸壳行进方向的右方。
- 使用同样的方式倒序循环求上凸壳（因为  $x-y$  排序，最右边的点是  $x$  最大的点，此时只求了一半）。注意到最右边的一个点一定在凸壳上，所以要从  $n-2$  开始循环。
- 如果所给的点数多于 1 个，则将凸包中的最后一个点删去。

```
1 int ConvexHull_Andrew(Point p[], int n, Point res[]) {
2     /* step 1: sort points by x,y */
3     std::sort(p, p + n);
4     int m = 0;
5     /* step 2: solve bottom convex shell */
6     for (int i = 0; i < n; i++) {
7         // if current point isn't on the left of convex hell
8         // delete exist points in the stack until empty or point on the left
9         // if P1P2 can't be parallel with P2P3, change >= to >
10        while (m > 1 && cross(res[m-1] - res[m-2], p[i] - res[m-2]) <= 0)
11            m--;
12        res[m++] = p[i];
13    }
14    int k = m;
15    /* step 3: solve top convex shell, point n-1 already in the hell */
```

```

16     for (int i = n - 2; i >= 0; i--) {
17         while (m > k && cross(res[m-1] - res[m-2], p[i] - res[m-2]) <= 0)
18             m--;
19         res[m++] = p[i];
20     }
21     if (n > 1)
22         m--;           // delete point 0
23     return m;
24 }
```

## 9.6.2 Graham 法求凸包

算法思想：Andrew 算法其实是由 Graham 扫描法改进而来。不同之处在于 Graham 扫描法采用极角排序，精度会降低。如非特殊需要应该使用 Andrew 算法。

极角排序规则：从  $x$  轴正向开始，按极角从小到大排序；如果极角相同则按照模长排序。极角排序可以用  $\text{atan2}$  函数求极角，或者用向量叉积。

```

1  bool cmp(Point p1, Point p2) {
2      Vector v = p1 - p[0], w = p2 - p[0];
3      double tmp = cross(v, w);
4      if (dcmp(tmp) == 0)
5          return length(v) < length(w);
6      return dcmp(tmp) > 0;
7  }
8  int k = 0;
9  Point p0 = p[0];
10 for (int i = 0; i < cnt; i++) {
11     if (p0.y > p[i].y || (p0.y == p[i].y && p0.x > p[i].x)) {
12         p0 = p[i];
13         k = i;
14     }
15 }
16 std::swap(p[k], p[0]);
17 std::sort(p, p + cnt, cmp);
18
// the remaining is the same as Andrew, but calculate only once
```

## 9.7 旋转卡壳

**对踵点对** 对于点  $P_i, P_j$ ，如果存在两条分别穿过这两个点的平行直线，把凸包夹在中间（即  $P_i, P_j$  与凸包相切），则这样的点对成为对踵点对。对踵点对最多只能有四对。

用两条平行且与已知凸包相切的直线在凸包上进行扫描，用于解决一些问题，如：凸多边形直径（凸多边形上或平面距离最远的两点）；凸多边形之间的最大、最小距离；最小面积/周长外接矩形。旋转卡壳具体思想如下：

- 使用两条有向直线将凸包夹在中间，上面的水平向左，下面的水平向右；
- 初始的时候找到  $y$  坐标最小和最大的两个点  $P_i$  和  $P_j$ .
- 逆时针旋转两条直线，假设穿过  $P_i, P_j$  的有向直线分别需要逆时针旋转  $\theta_i, \theta_j$  角度才能贴边  $P_iP_{i+1}, P_jP_{j+1}$ .
- 如果  $\theta_i < \theta_j$ ，则当旋转角为  $\theta_i$  时，对踵点对中  $P_i$  变为  $P_{i+1}$ ；同理  $\theta_j > \theta_i$ ，旋转角  $\theta_j$ ， $P_j$  变成  $P_{j+1}$ .
- 如果  $\theta_i = \theta_j$ ，两条直线同时贴住新的边， $P_iP_j, P_{i+1}P_{j+1}$  分别为对踵点对.
- 重复上述过程，直到穿过  $P_i$  的直线倾斜角大于  $\pi$  时终止。

实际上，判断最远点对的时候如何得到距离每条对应边的最远点呢？可以把点到直线的距离化解为三角形的面积，再运用叉积进行比较。同时，凸包上的点依次与对应边产生的距离成单峰函数，面积上升到最高点后，又会下降。利用单峰函数这一性质，我们可以根据凸包上点的顺序，枚举对踵点，直到下一个点的距离小于当前点就可以停止了，而且随着对应边的旋转，最远点也只会顺着这个方向旋转，我们可以从上一次的对踵点开始继续寻找这一次的。

但是下面的模板不具有普遍适用性。

```
1 double rotatingCalipers(Point p[], int n) {
2     double res = 0.0;
3     if (n == 2)
4         return lengthSquare(p[1] - p[0]);
5
6     p[n++] = p[0]; // put first point to the end
7     int m = 1;
8     // foreach every point
9     for (int i = 0; i < n; i++) {
10        Vector v1 = p[i+1] - p[i], v2 = p[m] - p[i], v3 = p[m + 1] - p[i];
11        // compare area
12        while (dcmp(cross(v1, v2) - cross(v1, v3)) < 0) {
13            m = (m + 1) % n;
14            v2 = p[m] - p[i], v3 = p[m + 1] - p[i];
15        }
16        res = std::max(res, std::max(lengthSquare(p[i] - p[m]), lengthSquare(p[i +
17            1] - p[m])));
18    }
19    return res;
}
```

可以改进上面的代码，原理如下：

```

1  for (int i = 0, m = 1; i < n; i++) {
2      // when area(p[i], p[i+1], p[m+1]) <= area(p[i], p[i+1], p[m]), stop rotate
3      // ==> cross(p[i+1]-p[i], p[m+1]-p[i]) - cross(p[i+1]-p[i], p[m]-p[i]) <= 0
4      // because cross(a, b) - cross(a, c) = cross(a, b-c)
5      // thus cross(p[i+1]-p[i], p[m+1]-p[m]) <= 0
6      Vector v1 = p[i+1] - p[i], v2 = p[m+1] - p[m];
7      // compare area
8      while (dcmp(cross(v1, v2)) > 0) {
9          m = (m + 1) % n;
10         v2 = p[m+1] - p[m];
11     }
12     res = std::max(res, std::max(lengthSquare(p[i] - p[m]), lengthSquare(p[i + 1] -
13         ↵ p[m])));
}

```

## 9.8 半平面交

**半平面** 半平面指一条直线的一侧的点构成的点集，一般包含直线上的点。数学表示即  $Ax + By + C \geq (\leq) 0$  或  $y \geq kx + b$ .

### 9.8.1 半平面的表示

使用有向的方向向量和直线上一点的参数表示法表示直线，规定方向向量的左侧即为半平面。

### 9.8.2 半平面交

求形如  $n$  条  $y \geq kx + b$  的半平面的并，类似线性规划。答案应该是形如凸包的下凸壳。如果是  $t \leq kx + b$  的半平面，则求的就是上凸壳。

算法 1：如果半平面的交不封闭，只需要将  $y = kx + b$  对偶成点  $(k, -b)$ ，然后水平排序求这些点的下凸壳。

算法 2：增量法。如果半平面的交封闭，则维护一个双端队列，将边按照极角排序，用求凸包的方法来求。步骤如下：

- 对目标向量进行极角排序（如果排序时遇到共线向量，则取靠近可行域的一个）
- 维护一个凸壳，使用双端单调队列维护。后来加入的向量只会影响最开始加入的或最后加入的边。
- 维护一个交点数组。当单调队列中的元素超过 2 个的时候，就会产生一个交点。对于当前向量，如果上一个交点在这条向量表示的半平面交的异侧，则上一条边就没有意义。
- 后加入的边也可能会影响队首的边。

- 如果半平面交是一个凸  $n$  边形，则最后在交点数组里会得到  $n$  个点。注意判断半平面交不存在或面积为 0 的情况。

```

1  /* 半平面交: 封闭 */
2  bool checkPointOnLeft(Line l, Point P) {
3      return dcmp(cross(l.s, P - l.p)) > 0;
4  }
5  // l[]: 多个半平面的边集, poly[]: 结果数组
6  int halfPlaneIntersection(Line l[], Point poly[], int n) {
7      // step 1: sort lines with angle
8      std::sort(l, l + n);
9      // step 2: create a deque
10     Point *p = new Point[n];    // p[i] is the intersection of q[i] and q[i+1]
11     Line *q = new Line[n];
12     int first, last;
13     // at the beginning there is only a half plane l[0]
14     q[first = last = 0] = l[0];
15     // step 3: for each vector
16     // check the position of intersection of current vector & last/first vector in
17     // the hell
18     for (int i = 1; i < n; i++) {
19         while (first < last && !checkPointOnLeft(l[i], p[last - 1]))
20             last--;
21         while (first < last && !checkPointOnLeft(l[i], p[first]))
22             first++;
23         q[++last] = l[i];        // add current vector
24         // step 4: if vectors are parallel, preserve the inner one
25         if (fabs(cross(q[last].s, q[last - 1].s)) < eps) {
26             last--;
27             if (checkPointOnLeft(q[last], l[i].p))
28                 q[last] = l[i];
29         }
30         // step 5: get the intersection of last 2 vectors(if exists)
31         if (first < last)
32             p[last - 1] = getLineIntersection(q[last - 1], q[last]);
33     }
34     // step 6: delete the useless plane
35     while (first < last && !checkPointOnLeft(q[first], p[last - 1]))
36         last--;
37     // half plane intersection not exists
38     if (last - first <= 1)
39         return 0;
40     // the intersection of first and last intersection
41     p[last] = getLineIntersection(q[last], q[first]);
42     // copy to dist array
43     int m = 0;
44     for (int i = first; i <= last; i++)
45         poly[m++] = p[i];

```

```
45     return m;
46 }
```

## 9.9 扫描线

计算矩形面积并、矩形面积交、矩形周长并等。以下是矩形面积并的模板。

```
1 double x[MAXN];
2 // 扫描线结构体，按照 y 坐标从下到上排序
3 struct ScanLine {
4     double l, r, h;
5     int d;          // 上位边为 1, 下位边为-1
6     ScanLine() {}
7     ScanLine(double l, double r, double h, int d): l(l), r(r), h(h), d(d) {}
8     bool operator < (const ScanLine sl) const {
9         return h < sl.h;
10    }
11 } a[MAXN << 1];
12 // 线段树的每个结点维护的是从当前坐标点到 坐标值 +1 的点的长度
13 struct SegTree {
14     #define lson (rt << 1)
15     #define rson (rt << 1) | 1
16     double sum[MAXN << 2];
17     int cnt[MAXN << 2];
18     void init() {
19         memset(sum, 0, sizeof sum);
20         memset(cnt, 0, sizeof cnt);
21     }
22     void pushUp(int rt, int l, int r) {
23         // 如果覆盖了这段区间，则该区间的长度就是 x[r+1] - x[l]；注意理解坐标的意义
24         if (cnt[rt])
25             sum[rt] = x[r+1] - x[l];
26         else if (l == r)
27             sum[rt] = 0.00;
28         else
29             sum[rt] = sum[lson] + sum[rson];
30     }
31     void update(int rt, int l, int r, int ul, int ur, int d) {
32         if (ul <= l && r <= ur) {
33             cnt[rt] += d;           // 根据上位边或下位边来决定某段区间是否全覆盖
34             pushUp(rt, l, r);
35             return;
36         }
37         int mid = (l + r) >> 1;
38         if (ul <= mid)
```

```

39         update(lson, l, mid, ul, ur, d);
40     if (mid < ur)
41         update(rson, mid + 1, r, ul, ur, d);
42     pushUp(rt, l, r);
43 }
44 } T;
45 int n, lcant = 0, xcnt = 0;
46 // main()
47 // 给 x[] 排序并去重，并对扫描线排序
48 std::sort(a, a + lcant);
49 std::sort(x, x + xcnt);
50 int k = 1;
51 for (int i = 1; i < xcnt; i++) // 尽量不要用 std::unique
52     if (x[i] != x[i-1])
53         x[k++] = x[i];
54 xcnt = k;
55 T.init();
56 for (int i = 0; i < lcant - 1; i++) { // 遍历第 0~lcant-1 条扫描线
57     // 找到扫描线覆盖的区间在离散后的数组 l[] 中的编号
58     int l = std::lower_bound(x, x + xcnt, a[i].l) - x,
59     r = std::lower_bound(x, x + xcnt, a[i].r) - x - 1;
60     // 因为线段树中的每个结点是代表从 [l, r+1] 的长度，所以 r 要-1。
61     // 注意这里的线段树起终点是 0~xcnt。
62     T.update(1, 0, xcnt, l, r, a[i].d);
63     // 答案增量是当前覆盖区间的长度 sum[1] * 当前扫描线与下一条扫描线的高度差
64     ans += T.sum[1] * (a[i+1].h - a[i].h);
65 }

```

求矩形面积并的情况：求面积时，用被覆盖 2 次以上的那一段乘以扫描线的距离即可。具体考虑每个区间的 cnt，如果  $cnt \geq 2$  则可以直接计算；如果  $cnt = 1$ ，若当前区间是叶子结点，则不计算在内；否则区间长度为左右儿子被覆盖过一次的区间长度之和（可以确定，这个区间被完全覆盖了 1 次，而有没有被完全覆盖两次或以上则不知道无法确定，需要从左右区间的信息推）； $cnt = 0$  只能从左右儿子的信息得到。因此只需要修改数组、pushUp 函数和计算和的部分。

```

1 double one[MAXN << 2], two[MAXN << 2];
2 void pushUp(int rt, int l, int r) {
3     if (cnt[rt] >= 2)
4         two[rt] = one[rt] = x[r+1] - x[l];
5     else if (cnt[rt] == 1)
6         one[rt] = x[r+1] - x[l];
7         two[rt] = (l == r) ? 0 : one[lson] + one[rson];
8     else
9         if (l == r)
10             one[rt] = two[rt] = 0;
11         else {
12             one[rt] = one[lson] + one[rson];
13             two[rt] = two[lson] + two[rson];

```

```

14     }
15 }
16 ans += T.two[1] * (a[i+1].h - a[i].h);

```

矩形周长并：用矩形面积并的方法对 x 轴和 y 轴各做一次扫描线。但是在更新答案的时候，需要记录上一次覆盖区间总的长度，并从当前覆盖长度中减去。也不用乘上扫描线间的高。

```

1 int last = 0;
2 // ...
3 ans += abs(T.sum[1] = last), last = sum[1];

```

当然我们也可只对轴做一次扫描线，只要同时维护轴竖线（就是求矩形面积并的时候的高）的个数，记录竖线的个数需要注意的是竖线重合的情况，需要再开变量来判断重合，避免重复计算。

## 9.10 平面最近点对（分治算法）

```

1 double solve(int l, int r) {
2     if (l == r)
3         return INF;
4     if (r - l == 1) {
5         return dist(pts[l], pts[r]);
6     }
7     int mid = (l + r) >> 1;
8     double d = min(solve(l, mid), solve(mid + 1, r));
9     double dx = (pts[mid].x + pts[mid+1].x) / 2.0, dy = (pts[mid].y + pts[mid+1].y)
10    ↵ / 2.0;
11    p1.clear(), p2.clear();
12
13    for (rint i = mid; i >= l; i--) {
14        if (pts[i].x >= dx - 2 * d)
15            p1.push_back(pts[i]);
16        else
17            break;
18    }
19    for (rint i = mid + 1; i <= r; i++) {
20        if (pts[i].x <= dx + 2 * d)
21            p2.push_back(pts[i]);
22        else
23            break;
24    }
25
26    unsigned int sz1 = p1.size(), sz2 = p2.size();
27    for (register unsigned int i = 0; i < sz1; i++)
28        for (register unsigned int j = 0; j < sz2; j++)
d = min(d, dist(p1[i], p2[j]));

```

```
29     |     return d;
30 }
```

平面最近线段对：对每条线段按照左端点排序，然后  $n^2$  枚举并剪枝：如果当前两个线段的左端点  $x$  之差大于答案，继续枚举下一个点即可。

## 9.11 给定半径任意圆能覆盖的最多的点的数量 (Angle-Sweep)

```
1 // LeetCode 1453
2 class Solution {
3 private:
4     struct Angle {
5         bool flag;           // 1: start, 0: end
6         double angle;
7         Angle(double _a = 0, bool _f = 0): angle(_a), flag(_f) {}
8         bool operator < (const Angle &b) const {
9             if (angle != b.angle)
10                return angle < b.angle;
11             return flag;
12         }
13     };
14 public:
15     int dcmp(double x) {
16         if (fabs(x) <= 1e-8)
17             return 0;
18         return x < 0 ? -1 : 1;
19     }
20     double sq(double x) {
21         return x * x;
22     }
23
24     int numPoints(vector<vector<int>>& points, int r) {
25         int n = points.size();
26         int ans = 1;
27         double R = (double)r;
28
29         const double pi = acos(-1);
30
31         for (int i = 0; i < n; i++) {
32             vector<Angle> tmp;
33             for (int j = 0; j < n; j++) {
34                 vector<int>& p0 = points[i], p1 = points[j];
35                 double dist = sqrt(sq(p0[0] - p1[0]) + sq(p0[1] - p1[1]));
36                 if (dcmp(dist) == 0 || dcmp(dist - 2 * R) > 0)
37                     continue;
38                 double base = atan2(p1[1] - p0[1], p1[0] - p0[0]),
```

```

39         range = acos(dist / 2 / R);
40         if (base < 0)
41             base += 2 * pi;
42         double start = base - range, end = base + range;
43         if (start >= 0) {
44             tmp.emplace_back(Angle(start, 1));
45             tmp.emplace_back(Angle(end, 0));
46         }
47         if (end <= 2 * pi) {
48             tmp.emplace_back(Angle(start + 2 * pi, 1));
49             tmp.emplace_back(Angle(end + 2 * pi, 0));
50         }
51     }
52     sort(tmp.begin(), tmp.end());
53     int sz = tmp.size(), cur = 1;
54     for (int j = 0; j < sz; j++) {
55         cur += tmp[j].flag == 1 ? 1 : -1;
56         ans = max(ans, cur);
57     }
58 }
59     return ans;
60 }
61 };

```

## 9.12 三维计算几何

- 基本定义与二维下的点、向量的定义相同。注意三维下点 + 点没有定义，但是点 + 向量 = 点、向量 + 向量 = 向量仍然成立。
- 点积、模长、两向量夹角的计算方法仍然与二维下的相同，只不过需要引入一个新的坐标 z 参与运算。

### 9.12.1 基本定义

```

1 inline int dcmp(double x) {
2     if (fabs(x) < eps)
3         return 0;
4     return x < 0 ? -1 : 1;
5 }
6
7 inline double sq(double x) {
8     return x * x;
9 }

```

```

10
11 struct Point3 {
12     double x, y, z;
13     Point3 (double _x = 0, double _y = 0, double _z = 0): x(_x), y(_y), z(_z) {}
14     bool operator == (const Point3 &b) const {
15         return dcmp(x - b.x) == 0 && dcmp(y - b.y) == 0 && dcmp(z - b.z) == 0;
16     }
17     bool operator < (const Point3 &b) const {
18         return dcmp(x - b.x) == 0 ? dcmp(y - b.y) == 0 ? z < b.z : y < b.y : x <
19             b.x;
20     }
21     double len() {
22         return sqrt(sq(x) + sq(y) + sq(z));
23     }
24     double len2() {
25         return sq(x) + sq(y) + sq(z);
26     }
27     double dist(const Point3 &b) const {
28         return sqrt(sq(x - b.x) + sq(y - b.y) + sq(z - b.z));
29     }
30     Point3 operator + (const Point3 &b) const {
31         return Point3(x + b.x, y + b.y, z + b.z);
32     }
33     Point3 operator - (const Point3 &b) const {
34         return Point3(x - b.x, y - b.y, z - b.z);
35     }
36     Point3 operator * (const double &k) const {
37         return Point3(x * k, y * k, z * k);
38     }
39     Point3 operator / (const double &k) const {
40         return Point3(x / k, y / k, z / k);
41     }
42     double operator * (const Point3 &b) const {
43         return x * b.x + y * b.y + z * b.z;
44     }
45     Point3 operator ^ (const Point3 &b) const {
46         return Point3(y * b.z - b.y * z, z * b.x - x * b.z, x * b.y - y * b.x);
47     }
48     double rad(Point3 a, Point3 b) {
49         Point3 p = (*this);
50         return acos((a - p) * (b - p) / (a.dist(p) * b.dist(p)));
51     }
52     Point3 trunc(double r) {
53         double l = len();
54         if (!dcmp(l))
55             return *this;
56         r /= l;
57         return Point3(x * r, y * r, z * r);

```

```

57     }
58 } a;
59
60 typedef Point3 Vector3;

```

## 9.12.2 求两向量夹角

```

1 double angle(Vector3 a, Vector3 b) {
2     return acos(a * b / a.len() / b.len());
3 }

```

## 9.12.3 点、线、平面

```

1 // 点 p 到平面 p0-n 的距离, p0 为平面内一点, n 为平面的单位法向量
2 double distanceToPlane(Point3 p, Point3 p0, Vector3 n) {
3     return fabs((p - p0) * n);
4 }
5 // 点 p 在平面 p0-n 上的投影, 同上
6 Point3 projection(Point3 p, Point3 p0, Vector3 n) {
7     return p - n * ((p - p0) * n);
8 }
9 // 直线与平面的交点
10 Point3 linePlaneIntersection(Point3 p1, Point3 p2, Point3 p0, Vector3 n) {
11     Vector3 v = p2 - p1;
12     double t = (n * (p0 - p1)) / (n * (p2 - p1));    // 记得判断分母 ≠ 0
13     return p1 + v * t;                                // 如果是线段, 判断 t ∈ [0, 1]
14 }

```

## 9.12.4 叉积应用

```

1 // 过不共线的三点的平面, 法向量为  $(p_2 - p_0) \times (p_1 - p_0)$ .
2
3 // 三角形的有向面积
4 double area2(Point3 a, Point3 b, Point3 c) {
5     return ((b - a) ^ (c - a)).len();
6 }
7
8 // 判断点是否在三角形内
9 bool pointInTriangle(Point3 p, Point3 p0, Point3 p1, Point3 p2) {
10     double s1 = area2(p, p0, p1),
11         s2 = area2(p, p1, p2),
12         s3 = area2(p, p2, p0);

```

```
13     |     return dcmp(s1 + s2 + s3 - area2(p0, p1, p2)) == 0;
14 }
```

## 9.12.5 向量旋转

```
1 // 向量 v 绕向量 u 转角 theta
2 Vector3 doRotate(Vector3 v, Vector3 u, double theta) {
3     Vector3 res = v * cosl(theta) + u * ((u * v) * (1.0 - cosl(theta))) + (u ^ v)
4         * sinl(theta);
5     return res;
}
```

## 9.12.6 点沿直线移动

```
1 // 点 p 沿方向向量 dir 移动距离 d
2 Point3 forward(Point3 p, Vector3 dir, double dist) {
3     double len = dir.len();
4     double x0 = p.x + dir.x / len * dist,
5         y0 = p.y + dir.y / len * dist,
6         z0 = p.z + dir.z / len * dist;
7     return Point3(x0, y0, z0);
8 }
```

## 9.12.7 三维空间点到线段的距离

```
1 // 点到线段的距离
2 double distToSegment(Point3 p, Point3 a, Point3 b) {
3     if (a == b)
4         return (p-a).len();
5     Vector3 v1 = b - a, v2 = p - a, v3 = p - b;
6     if (dcmp(v1 * v2) < 0)
7         return v2.len();
8     else if (dcmp(v1 * v3) > 0)
9         return v3.len();
10    else return ((v1 ^ v2) / v1.len()).len();
11 }
12
13 // 点到直线的距离
14 double distToLine(Point3 p, Point3 a, Point3 b) {
15     Vector3 v1 = b - a, v2 = p - a;
16     return (v1 ^ v2).len() / v1.len();
17 }
```

## 9.12.8 计算四面体的体积

$$V = \frac{1}{3}S \cdot h = \frac{1}{6}((\vec{AB} \times \vec{AC}) \cdot \vec{AD})$$

```
1 | double volume(Point3 a, Point3 b, Point3 c, Point3 d){  
2 |     return 1.0 / 6 * (d - a) * ((b - a) ^ (c - a));  
3 | }
```

## 9.13 常用公式

### 9.13.1 三角形

- 半周长:  $P = \frac{(a+b+c)}{2}$
- 海伦公式:  $S = \sqrt{P(P-a)(P-b)(P-c)}$
- 角  $A$  到  $BC$  边的中线长度公式  $\frac{\sqrt{2(b^2+c^2)-a^2}}{2} = \frac{\sqrt{b^2+c^2-2bccosA}}{2}$
- 角  $A$  的角平分线长度公式:  $\frac{\sqrt{bc((b+c)^2-a^2)}}{(b+c)} = \frac{2bccos(\frac{A}{2})}{(b+c)}$
- $BC$  边的高线长度公式:  $H_a = bsinC = csinB = \sqrt{b^2 - (\frac{a^2+b^2-c^2}{2a})^2}$
- 内切圆半径  $r = asin(\frac{B}{2})sin(\frac{C}{2})sin(\frac{B+C}{2}) = \frac{S}{P}$ , 其它同理。
- 外接圆半径  $R = \frac{abc}{4S} = \frac{a}{2sinA} = \frac{b}{2sinB} = \frac{c}{2sinC}$

### 9.13.2 四边形

记  $D_1, D_2$  为对角线,  $M$  为对角线中点连线,  $A$  为对角线夹角。

- $a^2 + b^2 + c^2 + d^2 = D_1^2 + D_2^2 + 4M^2$
- $S = \frac{D_1 D_2 sin A}{2}$
- 若为圆的内接四边形:  $ac + bd = D_1 D_2$
- 若为圆的内接四边形:  $S = \sqrt{(P-a)(P-b)(P-c)(P-d)}$ , 其中  $P$  为半周长

### 9.13.3 正 n 边形

$R$  为外接圆半径,  $r$  为内切圆半径。

- 中心角  $A = \frac{2\pi}{n}$
- 内角  $C = \frac{(n-2)\pi}{n}$

- 边长  $a = 2\sqrt{R^2 - r^2} = 2R\sin\frac{A}{2} = 2r\tan\frac{A}{2}$
- 面积  $S = \frac{n\dot{a} \cdot \text{dot}r}{2} = \frac{nR^2 \sin A}{2}$ , 其中  $a$  为正  $n$  边形的边长

#### 9.13.4 圆

$A$  为对应的圆心角。

- 弧长  $l = rA$ , 弦长  $a = 2\sqrt{2hr - h^2} = 2r\sin\frac{A}{2}$ , 其中  $h$  是圆心到弦的距离
- 弓形高  $h = r - \sqrt{r^2 - \frac{a^2}{4}} = r(1 - \cos\frac{A}{2})$
- 扇形面积  $S_1 = \frac{rl}{2} = \frac{Ar^2}{2}$
- 弓形面积  $S_2 = \frac{(r_1 - a(r-h))}{2} = r^2 \frac{A - \sin A}{2}$

#### 9.13.5 棱柱

- 体积:  $V = Ah$ ,  $A$  为底面积,  $h$  为高
- 侧面积:  $S = lp$ ,  $l$  为棱长,  $p$  为直截面周长
- 全面积:  $T = S + 2A$ , 侧面积加 2 倍底面积

#### 9.13.6 棱锥

- 体积:  $V = \frac{Ah}{3}$
- 侧面积:  $S = \frac{lp}{2}$ ,  $l$  为斜高 (侧面图形的高),  $p$  为底面周长
- 全面积:  $T = S + A$ , 底面只有一个。

#### 9.13.7 圆锥

- 母线  $l = \sqrt{h^2 + r^2}$
- 侧面积  $S = \pi rl$
- 全面积  $T = \pi r(l + r)$
- 体积  $V = \frac{\pi r^2 h}{3}$

### 9.13.8 圆台

设圆台的下底面半径为  $r_1$ , 上底面半径为  $r_2$ .

- 母线  $l = \sqrt{h^2 + (r_1 - r_2)^2}$
- 侧面积  $S = \pi l(r_1 + r_2)$
- 全面积  $T = \pi r_1(l + r_1) + \pi r_2(l + r_2)$
- 体积  $V = \frac{\pi(r_1^2 + r_2^2 + r_1 r_2)h}{3}$

# Chapter 10

## 博弈论

### 10.1 Nim 游戏

#### 10.1.1 定义

Nim 游戏：有两个玩家，轮流进行操作；是公平游戏，即面对同一局面两个玩家所能进行的操作是相同的；一个玩家是输掉当且仅当他无法进行操作。

一般形式：给定  $n$  堆物品，第  $i$  堆物品有  $A_i$  个；两名玩家分别轮流行动，每次可以任选一堆取走任意多个物品，可把一堆取完但不可以不取，取走最后一件物品者获胜。两人都采取最优策略，问先手能否必胜。

#### 10.1.2 概念和性质

必胜点和必败点的概念：

- P 点：必败点，换而言之，就是谁处于此位置，则在双方操作正确的情况下必败。
- N 点：必胜点，处于此情况下，双方操作均正确的情况下必胜。

必胜点和必败点的性质：

- 所有终结点是必败点 P。（我们以此为基本前提进行推理，换句话说，我们以此为假设）
- 从任何必胜点 N 操作，至少有一种方式可以进入必败点 P。
- 无论如何操作，必败点 P 都只能进入必胜点 N。

#### 10.1.3 结论

一个 Nim 游戏中的状态是必败状态当且仅当每个子游戏的异或和为 0。

Nim 博弈先手必胜，当且仅当  $A_1 \text{ xor } A_2 \text{ xor } \dots \text{ xor } A_n \neq 0$

**SG 定理：**游戏和的 SG 函数等于各个游戏 SG 函数的 Nim 和。

**mex (minimal excludant) 运算：**施加于集合的运算，表示最小的不属于这个集合的非负整数。  
对于任意状态  $x$ ，定义  $SG(x) = \text{mex}(S)$ ，其中  $S$  是  $x$  后继状态的 SG 函数值的集合。

#### 10.1.4 SG 函数打表模板

```
1 // f[N]: 可改变当前状态的方式, N 为方式的种类, f[N] 要在 getSG 之前先预处理
2 // 比如, 如果每次可以取走 1,3,4 个石子, 那么 f[] = {1, 3, 4}
3 // SG[i]: 0~n 的 SG 函数值; S[]: 为 x 后继状态的集合
4 int f[N], SG[MAXN], S[MAXN];
5 void getSG(int n) {
6     int i, j;
7     memset(SG, 0, sizeof SG);
8     // 初始化最终状态的 SG 值, 如 SG[0] = 0;
9     for(i = 1; i <= n; i++) {
10         // 每一次都要将上一状态的后继集合重置
11         memset(S, 0, sizeof S);
12         // 将后继状态的 SG 函数值进行标记
13         for(j = 0; f[j] <= i && j <= N; j++)
14             S[SG[i - f[j]]] = 1;
15         for(j = 0; ; j++)           // mex 运算
16             if (!S[j]) {
17                 SG[i] = j;
18                 break;
19             }
20     }
21 }
```

## 10.2 常见博弈模型

**巴什博弈** 一堆  $n$  个物品，两个人轮流从这堆物品中取物，规定每次至少取一个，最多取  $m$  个。最后取光者得胜。

结论：当  $n \bmod (m + 1) = 0$  时先手必败，否则先手必胜。

**威佐夫博弈** 有两堆各若干个物品，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。

结论：如果  $\text{ratio} \cdot (\max - \min) == \min$ ，则先手必败，否则先手必胜。 $\text{ratio}$  为黄金分割比，计算公式为  $\frac{1+\sqrt{5}}{2}$ .

```
1 bool wythoff(int a, int b) {
2     const double r = (1 + sqrt(5.0)) / 2.0;
3     int mi = min(a, b), mx = max(a, b);
4     if (mi == (int)(r * (mx - mi)))
5         return false;
6     return true;
7 }
```

# Chapter 11

## 杂项

### 11.1 分治

#### 11.1.1 二分

```
1 int x, y, flag, m, ans = -1;
2 int a[MAXN];
3 while (x < y) {
4     m = x + (y - x) / 2;           // 找中点，向下取整
5     if (a[m] == flag) {           // 恰好中点就是结果
6         ans = m;
7         break;
8     } else if (a[m] > flag) {
9         y = m;                     // 往左边找
10    } else {
11        x = m + 1;               // 往右边找，注意起点 +1
12    }
13 }
```

#### 11.1.2 三分法

三分查找是在二分查找分出了两个区间（左区间，右区间）的情况下，再对左区间或右区间进行一次二分，以快速确定最值。三分法要求序列是一个有凹凸性的函数。步骤如下：

- 取得区间的中间值  $mid = \lfloor \frac{(left+right)}{2} \rfloor$
- 取右区间的中间值  $rmid = \lfloor \frac{(mid+right)}{2} \rfloor$
- 判断  $a[rmid]$  和  $a[mid]$  的关系，若  $a[mid]$  比  $a[rmid]$  更接近最值，舍弃右区间搜索左区间；否则搜索右区间。

```

1 // 三分求函数的最小值
2 int l = 0, r = 10000000, ans, ll, rr;
3 while (l + 2 < r) {
4     ll = (l + l + r) / 3;
5     rr = (l + r + r) / 3;
6     if (func(ll) < func(rr))
7         r = rr;
8     else
9         l = ll;
10 }
11 ans = l;
12 for (int i = l + 1; i <= r; i++)
13     if (func(ans) > func(i))
14         ans = i;

```

## 11.2 CDQ 分治求三维偏序

```

1 struct Node {
2     int a, b, c, cnt, id;
3     bool operator < (const Node &t) const {
4         if (a != t.a)
5             return a < t.a;
6         else {
7             if (b != t.b)
8                 return b < t.b;
9             else
10                 return c < t.c;
11         }
12     }
13     bool operator == (const Node &t) const {
14         return a == t.a && b == t.b && c == t.c;
15     }
16 } t[MAXN], tmp[MAXN]; // tmp 是归并用的数组
17 int n, k, cnt = 1;
18 int ans[MAXN], f[MAXN];
19 void cdq(int l, int r) {
20     if (r - l <= 1)
21         return;
22     int m = (l + r) >> 1;
23     cdq(l, m), cdq(m, r);
24     int i = l, j = m, idx = 1;
25     // 统计影响并将左侧可能对右侧造成影响的节点加入树状数组
26     // 如果右侧的节点不受左侧的影响，那么直接更新答案
27     while (i < m && j < r) {
28         if (t[i].b <= t[j].b) {

```

```

29         T.add(t[i].c, t[i].cnt);
30         tmp[idx++] = t[i++];
31     } else {
32         ans[t[j].id] += T.query(t[j].c);
33         tmp[idx++] = t[j++];
34     }
35 }
36 while (j < r) {
37     ans[t[j].id] += T.query(t[j].c);
38     tmp[idx++] = t[j++];
39 }
40 for (int x = l; x < i; x++)
41     T.add(t[x].c, -t[x].cnt);
42 while (i < m)
43     tmp[idx++] = t[i++];
44 for (int i = l; i < r; i++)
45     t[i] = tmp[i];
46 }
```

## 11.3 逆序对

### 11.3.1 归并排序求逆序对

```

1 int n, m, ans = 0;
2 void merge_sort(int x, int y) {
3     if (y - x > 1) {
4         int m = x + (y - x) / 2;
5         int p = x, q = m, i = x;
6         merge_sort(x, m);
7         merge_sort(m, y);
8         while (p < m || q < y) {
9             if (q >= y || (p < m && a[p] <= a[q]))
10                 b[i++] = a[p++];
11             else {
12                 b[i++] = a[q++];
13                 ans += m - p;
14             }
15         }
16         for (i = x; i < y; i++)
17             a[i] = b[i];
18     }
19 }
```

### 11.3.2 树状数组求逆序对

```
1 struct Node {
2     int val, index;
3     bool operator < (const Node &b) const {
4         return val < b.val;
5     }
6 } t[MAXN];
7 BIT T;
8 sort(t, t + n);
9 int ans = 0;
10 for (int i = 0; i < n; i++) {
11     T.add(t[i].index);
12     ans += i - T.query(t[i].index);
13 }
```

### 11.4 莫队算法 (带修改)

```
1 struct Query {
2     int l, r, time, id;
3 } q[MAXM]; // 查询结构体
4 struct Mod {
5     int pos, color;
6 } mods[MAXN]; // 修改内容结构体
7 int n, m, a[MAXN], block[MAXN], // 分块
8     res[MAXN], cnt[MAXL], // res 保存询问的回答
9     qcmt, mcmt, ans, time; // time: 当前时间
10
11 bool cmp(const Query a, const Query b) {
12     return (block[a.l] ^ block[b.l])
13         ? (block[a.l] < block[b.l])
14         : (block[a.r] ^ block[b.r])
15         ? (block[a.r] < block[b.r])
16         : a.time < b.time;
17 // 不带修改的写法
18 /* return (block[a.l] ^ block[a.l])
19     ? block[a.l] < block[b.l]
20     : ((block[a.l] & 1) ? a.r < b.r : a.r > b.r); */
21 }
22 void add(int pos) {
23     if (cnt[a[pos]] == 0)
24         ans++;
25     cnt[a[pos]]++;
26 }
```

```

27 void del(int pos) {
28     cnt[a[pos]]--;
29     if (cnt[a[pos]] == 0)
30         ans--;
31 }
32 void work(int t, int i) {
33     if (mods[t].pos >= q[i].l && mods[t].pos <= q[i].r) {
34         cnt[a[mods[t].pos]]--;
35         if (cnt[a[mods[t].pos]] == 0)
36             ans--;
37         cnt[mods[t].color]++;
38         if (cnt[mods[t].color] == 1)
39             ans++;
40     }
41     swap(mods[t].color, a[mods[t].pos]);
42 }
43
44 // main
45 /* 第一步，计算块大小，读入数据并分块；略去添加查询 */
46 int blockSize = pow(n, 2.0/3.0),
47     blockCnt = ceil((double)n / blockSize);
48 for (int i = 1; i <= n; i++) {
49     read(a[i]);
50     block[i] = (i - 1) / blockSize + 1;
51 }
52 // 如果莫队是带修改的，那么查询的时间轴为当前的 mcnt
53 /* 第二步，对所有的查询询问排序 */
54 sort(q + 1, q + qcmt + 1, cmp);
55 /* 第三步，处理所有询问，对两个指针进行移动直到移到查询区间为止。注意 l,r 的初值 */
56 int l = 1, r = 0;
57 for (int i = 1; i <= qcmt; i++) {
58     while (l < q[i].l)
59         del(l++);
60     while (l > q[i].l)
61         add(--l);
62     while (r < q[i].r)
63         add(++r);
64     while (r > q[i].r)
65         del(r--);
66     // 如果当前时间不在查询时间范围内，那就要处理修改
67     while (time < q[i].time)
68         work(++time, i);
69     while (time > q[i].time)
70         work(time--, i);
71     res[q[i].id] = ans;      // 最后将答案离线到答案数组中
72 }

```

## 11.5 表达式处理

```
1  bool shouldPopFromStack(char current, char fromStack) {
2      int p1 = (current == '*' || current == '/') ? 1 : 0;
3      int p2 = (fromStack == '*' || fromStack == '/') ? 1 : 0;
4      return p1 <= p2;
5  }
6  string convert(string raw) {
7      int length = raw.length();
8      stack<char> s;
9      string res = "";
10     for (int i = 0; i < length; i++) {
11         if (raw[i] >= '0' && raw[i] <= '9') {
12             res += raw[i];
13             continue;
14         }
15         else
16             res += ' ';
17         if (raw[i] == '(')
18             s.push('(');
19         else if (raw[i] == ')') {
20             while (true) {
21                 char now = s.top();
22                 s.pop();
23                 if (now == '(')
24                     break;
25                 else
26                     res += now;
27             }
28         }
29         else {
30             if (s.empty())
31                 s.push(raw[i]);
32             else {
33                 while (!s.empty()) {
34                     char now = s.top();
35                     if (now == '(' || !shouldPopFromStack(raw[i], now))
36                         break;
37                     else {
38                         s.pop();
39                         res += now;
40                     }
41                 }
42                 s.push(raw[i]);
43             }
44         }
45     }
46 }
```

```

46     res += ' ';
47     while (!s.empty()) {
48         char now = s.top();
49         s.pop();
50         res += now;
51     }
52     return res;
53 }
54 int main() {
55     /* ... */
56     stack<long long> s;
57     int length = res.length();
58     long long cur = 0;
59     for (int i = 0; i < length; i++) {
60         if (res[i] >= '0' && res[i] <= '9') {
61             cur = cur * 10 + res[i] - '0';
62         } else if (res[i] == '+') {
63             s.push(cur);
64             cur = 0;
65         }
66     } else {
67         long long s2 = s.top();
68         s.pop();
69         long long s1 = s.top();
70         s.pop();
71         switch (res[i]) {
72             case '+':
73                 s.push(s1 + s2);
74                 break;
75             case '-':
76                 s.push(s1 - s2);
77                 break;
78             case '*':
79                 s.push(s1 * s2);
80                 break;
81             case '/':
82                 s.push(s1 / s2);
83                 break;
84         }
85     }
86 }
87 /* ... */
88 }
```

## 11.6 C++ 大整数模板

```
1 struct BigInt{
2     int d[maxn], len;
3
4     BigInt() { memset(d, 0, sizeof(d)); len = 1; }
5     BigInt(int num) { *this = num; }
6     BigInt(char* num) { *this = num; }
7
8     void clean() { while(len > 1 && !d[len-1]) len--; }
9     BigInt operator = (const char* num){
10         memset(d, 0, sizeof(d)); len = strlen(num);
11         for(int i = 0; i < len; i++) d[i] = num[len-1-i] - '0';
12         clean();
13         return *this;
14     }
15     BigInt operator = (int num){
16         char s[20];
17         sprintf(s, "%d", num);
18         *this = s;
19         return *this;
20     }
21
22     BigInt operator + (const BigInt& b){
23         BigInt c = *this; int i;
24         for (i = 0; i < b.len; i++){
25             c.d[i] += b.d[i];
26             if (c.d[i] > 9) c.d[i] %= 10, c.d[i+1]++;
27         }
28         while (c.d[i] > 9) c.d[i++] %= 10, c.d[i]++;
29         c.len = max(len, b.len);
30         if (c.d[i] && c.len <= i) c.len = i+1;
31         return c;
32     }
33     BigInt operator - (const BigInt& b){
34         BigInt c = *this; int i;
35         for (i = 0; i < b.len; i++){
36             c.d[i] -= b.d[i];
37             if (c.d[i] < 0) c.d[i] += 10, c.d[i+1]--;
38         }
39         while (c.d[i] < 0) c.d[i++] += 10, c.d[i]--;
40         c.clean();
41         return c;
42     }
43     BigInt operator * (const BigInt& b) const{
44         int i, j;
45         BigInt c;
```

```

46     c.len = len + b.len;
47     for(j = 0; j < b.len; j++)
48         for(i = 0; i < len; i++)
49             c.d[i+j] += d[i] * b.d[j];
50     for(i = 0; i < c.len-1; i++)
51         c.d[i+1] += c.d[i] / 10, c.d[i] %= 10;
52     c.clean();
53     return c;
54 }
55 BigInt operator / (const BigInt& b){
56     int i, j;
57     BigInt c = *this, a = 0;
58     for (i = len - 1; i >= 0; i--) {
59         a = a * 10 + d[i];
60         for (j = 0; j < 10; j++)
61             if (a < b*(j+1))
62                 break;
63         c.d[i] = j;
64         a = a - b*j;
65     }
66     c.clean();
67     return c;
68 }
69 BigInt operator % (const BigInt& b){
70     int i, j;
71     BigInt a = 0;
72     for (i = len - 1; i >= 0; i--) {
73         a = a * 10 + d[i];
74         for (j = 0; j < 10; j++) if (a < b*(j+1)) break;
75         a = a - b*j;
76     }
77     return a;
78 }
79 BigInt operator += (const BigInt& b){
80     *this = *this + b;
81     return *this;
82 }
83
84     bool operator < (const BigInt& b) const{
85         if(len != b.len)
86             return len < b.len;
87         for(int i = len-1; i >= 0; i--)
88             if(d[i] != b.d[i])
89                 return d[i] < b.d[i];
90         return false;
91     }
92     bool operator > (const BigInt& b) const{return b < *this;}
93     bool operator <= (const BigInt& b) const{return !(b < *this);}

```

```

94     bool operator >= (const BigInt& b) const{return !(*this < b);}
95     bool operator != (const BigInt& b) const{return b < *this || *this < b;}
96     bool operator == (const BigInt& b) const{return !(b < *this) && !(b > *this);}
97
98     string str() const{
99         char s[maxn]={};
100        for(int i = 0; i < len; i++)
101            s[len-1-i] = d[i]+'0';
102        return s;
103    }
104}
105
106 istream& operator >> (istream& in, BigInt& x) {
107     string s;
108     in >> s;
109     x = s.c_str();
110     return in;
111 }
112
113 ostream& operator << (ostream& out, const BigInt& x) {
114     out << x.str();
115     return out;
116 }
```

## 11.7 快速读入

```

1 // 整数读入 (int, ll)
2 template<class T> void read(T &x) {
3     T a = 0, f = 1;
4     char ch = getchar();
5     while (ch < '0' || ch > '9')
6         f = ch == '-' ? -1 : f, ch = getchar();
7     while (ch >= '0' && ch <= '9')
8         a = a * 10 + ch - '0', ch = getchar();
9     x = a * f;
10 }
11
12 // 浮点数读入
13 inline double read() {
14     double x = 0, y = 1.0;
15     int f = 0;
16     char ch = getchar();
17     while (!isdigit(ch))
18         f |= ch == '-' , ch = getchar();
```

```

19     while (isdigit(ch))
20         x = x * 10 + (ch ^ 48), ch = getchar();
21     ch = getchar();
22     while (isdigit(ch))
23         x += (y /= 10) * (ch ^ 48), ch = getchar();
24     return f ? -x : x;
25 }
```

## 11.8 \_\_int128 输出函数

```

1 void output(__int128 x) {
2     if (!x)
3         return;
4     if (x < 0)
5         putchar('-'), x = -x;
6     output(x / 10);
7     putchar(x % 10 + '0');
8 }
```

## 11.9 开栈

Tips: 并不是在所有的地方都能用。

```

1 //64-bit
2 int size = 1 << 20;      //256M
3 char *p = (char *)malloc(size) + size;
4 __asm__("movq %0, %%rsp\n" :: "r"(p));
5
6 //32-bit
7 int size = 1 << 20;      //256M
8 char *p = (char *)malloc(size) + size;
9 __asm__("movl %0, %%esp\n" :: "r"(p));
```

## 11.10 随机数生成

default\_random\_engine 需要 C++11.

```

1 default_random_engine e;
2 e.seed((int)time(0));
3 }
```

```
4 int randInt(int minInt, int maxInt) {
5     uniform_int_distribution<int> rd(minInt, maxInt);
6     return rd(e);
7 }
8 double randDouble(double minD, double maxD) {
9     uniform_real_distribution<double> rd(minD, maxD);
10    return rd(e);
11 }
```

# Chapter 12

## 语言 & 库参考

### 12.1 ACM Java 速成

```
1 import java.io.*;
2 import java.math.*;
3 import java.util.*;
4 import java.text.*;
5
6 public class Main {
7     public static void main(String[] args) {
8         // 输入
9         Scanner cin = new Scanner(new BuffereredInputStream(System.in));
10        int a = cin.nextInt();
11        double b = cin.nextDouble();
12        BigInteger c = cin.nextBigInteger();
13        String d = cin.nextLine();           // 或者 cin.next(), nextLine 相当于
14        ↪ getline
15        boolean isEOF = !cin.hasNext();      // 判断 EOF
16
17        // 输出
18        System.out.println("No Answer! " + str);
19        System.out.printf("%f%d%c%s", doubleVar, intVar, charVar, stringVar);
20
21        // 字符串
22        char[] stringArray = { 'H', 'e', 'l', 'l', 'o' };
23        String s = new String(stringArray);    // 从 char[] 构造字符串
24        String st = "abcdefg";
25        System.out.println(st.charAt(0));       // st.charAt(i) 就相当于 st[i]
26        char [] ch;
27        ch = st.toCharArray();                // 字符串转换为字符数组
28        if (st.startsWith("a"))              // 判断字符串开头
29            st = st.substring(1);           // 从第 1 位开始截取字符
30
31        // 高精度 (BigInteger 大整数, BigDecimal 高精小数)
```

```

31     BigInteger ba = cin.nextBigInteger(),           // 从输入读取
32         bb = BigInteger.valueOf(233),               // 从 int 转换
33         bc = new BigInteger("114514");             // 从 string 转换
34     int xba = ba.intValue();                      // 转换成 int
35     BigInteger addRes = ba.add(bb),              // +
36         subRes = ba.subtract(bb),                 // -
37         mulRes = ba.multiply(bb),                // ×
38         divRes = ba.divide(bb),                  // ÷
39         modRes = ba.mod(bb);                    // 取模
40     if (addRes.compareTo(bc) == 0) {}            // -1: < 0: = 1: >
41
42     // 进制转换
43     // 把 num 当做 10 进制的数转成 base 进制的 st (base <= 35).
44     String st = Integer.toString(num, base);
45     // 把 st 当做 base 进制, 转成 10 进制的 int
46     // parseInt 有两个参数, 第一个为要转的字符串, 第二个为说明是什么进制
47     int num = Integer.parseInt(st, base);
48     // st 是字符串, base 是 st 的进制.
49     BigInteger m = new BigInteger(st, base),
50         m2 = cin.nextBigInteger(2);           // 以二进制形式读入 BigInteger
51     // 要将一个大数转换成其他进制形式的字符串 使用 m.toString(2);
52
53     // 排序
54     int a[] = new int [n];
55     for (int i = 0; i < n; i++)
56         a[i] = cin.nextInt();
57     Arrays.sort(a);
58     for (int i = 0; i < n; i++)
59         System.out.print(a[i] + " ");
60     }
61 }
```

## 12.2 ACM Python 速成

```

1 # 基本输入输出
2 t = int(input().strip())          # 输入一个 int
3
4 # 输入多组数据直到 EOF, 每组读入 a, b 两个数
5 try:
6     while True:
7         a, b = map(int, input().strip().split())
8         print(a + b)
9 except EOFError:
10    pass
```

```

11 # 数组输入
12 x = input()
13 a = []
14 for i in x.split():
15     a.append(int(i)) # 注意数据类型转换
16
17 # 输出
18 print(a)          # 默认换行
19 print(a, end='') # 输出不换行
20
21 # 数组操作
22 a1 = [] # 空列表
23 a2 = [1] * 10 # 开一个 10 个元素的数组
24 a3 = [1, 1] + [2, 3] # 数组拼接
25 a4 = list(range(8)) # [0, 8), or range(0, 8)
26 a6 = [[1] * 3 for _ in range(3)] # 同样是开一个 3*3 的数组
27
28 # 数组操作的方法
29 len(a1) # 获取数组长度
30 a1.append(8) # 向末尾添加一个数
31 a1[0] = 0 # 访问和赋值
32 a1[-1] = 7 # 从末尾开始访问
33 a1[2:5] # 提取数组的一段 [2, 3, 4]
34 a1[5:2:-1] # 倒序访问 [5, 4, 3]
35 a1.sort() # 数组排序
36 a2[0][0] = 10 # 访问和赋值二维数组
37 for i, a3 in enumerate(a2):
38     for j, v in enumerate(a3):
39         temp = v # 这里的 v 就是 a[i][j]
40

```

## 12.3 字符串处理函数

### 12.3.1 <cstring> 头文件函数

### 12.3.2 STL string

- 迭代器: begin(), end(), rbegin(), rend(), 其中后两者是反着迭代的。
- size(), length(): int 返回字符串的长度, max\_size(): int 返回字符串的最大可存储长度。
- capacity(): int 返回当前已分配空间长度。
- reserve(size) 表示预分配 size 长度的空间; clear() 表示将字符串清空。
- empty(): bool 返回当前字符串是否为空。

函数原型	作用	返回值
memcpy(dest, source, size)	从 source 数组中复制内容到 dest 数组中	*dest
strcpy(dest, source)	复制字符串	*dest
strncpy(dest, source, size)	复制字符串并指定字符数	*dest
strcat(dest, source)	将 source 接到 dest 之后	*dest
strncat(dest, source, size)	将 source 的 size 个字符接到 dest 之后	*dest
memcmp(arr1, arr2, size)	比较 arr1 和 arr2 前 size 个字节的内存	相同 0, 1<2: <0, 1>2: >0
strcmp(str1, str2)	比较字符串 str1 和 str2	同上
strncmp(str1, str2, size)	比较字符串 str1 和 str2 的前 size 个字节	同上
memchr(arr, char, size)	在 arr 的前 size 个字节中寻找字符 char 的位置	指针指向目标, 找不到返回 null
strchr(str, char)	在字符串 str 中寻找字符 char 第一次出现的位置	返回第一个位置的指针, 同上
strrchr(str, char)	在字符串 str 中寻找字符 char 最后出现的位置	同上
strcspn(str1, str2)	字符串 str1 开头连续几个字符都不含 str2 中的字符	返回结果 int
strspn(str1, str2)	字符串 str1 开头连续有几个字符都在 str2 中	返回结果 int
strpbrk(str1, str2)	在 str1 中搜索 str2 中字符第一次出现的位置	同上
strstr(str1, str2)	在 str1 中检索子串 str2 第一次出现的位置	同上
strtok(str1, sep)	将 str1 中第一个 sep 中含的字符设为 0	返回分割后字符串的地址

Table 12.1: cstring 函数合集

- shrink\_to\_fit() 表示减小字符串占用的空间 capacity 来适合它的 size.
- append(string/char[]/...) 和 += 同效, 将字符串插入到已有的 string 后面。erase(iterator) 删除指定迭代器的字符。
- push\_back(char) 把字符插入到以后的 string 后面; pop\_back() 起反作用。
- replace(from, size, str2) 将 str1 从 from 往下数的 size 个字符替换为 str2, str2 长度任意。也可以使用迭代器指定位置。replace(from, size, str2, t1, t2) 则将选定的部分替换为 str2 的 t1 t2 字符。
- str1.swap(str2) 表示将 str1 与 str2 交换。
- c\_str() 函数将 string 转换为 char[] 数组; copy(chr[], len, firstPos) 表示从 chr[] 中拷贝字符到 string 中。
- find(str) 寻找子串在 str 第一次出现的位置, 找不到返回 npos, 否则返回位置的下标 (size\_t)。 rfind(str) 寻找最后一次出现的位置。
- substr(pos, len) 从字符串 str 的第 pos 位开始向下截取 len 位形成新的字符串。
- find\_first\_of, find\_last\_of, find\_first\_not\_of, find\_last\_not\_of, 传入 string, 作用即字面意思, 返回第一个/最后一个传入的 string 中任意字符出现的位置; 后两者的作用则相反, 返回第一个/最后一个不在 string 中的任意字符出现的位置。
- compare(pos, len, str, spos, n) 分别从第一个字符串的 pos 和第二个字符串 str 的 spos 开始向下比较 n 个字符, len 是从 pos 往后数第一个字符的长度。
- getline(cin, str) 读一整行字符串, 类似 gets.

## 12.4 GNU pbds Reference

### 12.4.1 头文件和命名空间定义

```
1 #include <ext/pb_ds/assoc_container.hpp>      // 基本组件
2 #include <ext/pb_ds/tree_policy.hpp>            // set/map/rbt/splay
3 #include <ext/pb_ds/hash_policy.hpp>             // hash table
4 #include <ext/pb_ds/priority_queue.hpp>           // 优先队列
5 using namespace __gnu_pbds;                      // 命名空间
```

### 12.4.2 Hash Table

比 map 更快, 近似认为  $O(n)$ , 建议用 gp\_hash\_table.

```
1 cc_hash_table<int, int> h1;                  // 拉链法哈希表
2 gp_hash_table<int, int> h2;                  // 探测法哈希表
3 h2.insert(std::make_pair(key, value));
4 h2.erase(key);
5 for (gp_hash_table<int, int>::iterator = h2.begin(); itor != h2.end(); itor++) {
6     cout << itor->first << " " << itor->second << endl;
7 }
8 if (h2.find(key) != h2.end())      return;
9 h2.clear();
10 cout << h2.size();
```

### 12.4.3 Priority Queue

```
1 /* definition: priority_queue<value_type, cmp_func, tag> pq; */
2 priority_queue<pair<int, int>, greater<pair<int, int> >, pairing_heap_tag> heap;
3
4 int len = pq.size();
5 bool isEmpty = pq.empty();
6 heap::point_iterator itor = pq.push(el);    // 返回元素的迭代器
7 int cur = pq.top();
8 pq.erase(itor);
9 pq.modify(itor, new_value);                // 修改元素迭代器指向的元素
10 pq.clear();                                // 清空优先队列
11 pq.join(pq2);                            // 合并优先队列
```

各种堆的效率比较:

```

* pairing_heap_tag: push和join为O(1), 其余为均摊 $\Theta(\log n)$ 
* binary_heap_tag: 只支持push和pop, 均为均摊 $\Theta(\log n)$ 
* binomial_heap_tag: push为均摊O(1), 其余为 $\Theta(\log n)$ 
* rc_binomial_heap_tag: push为O(1), 其余为 $\Theta(\log n)$ 
* thin_heap_tag: push为O(1), 不支持join, 其余为 $\Theta(\log n)$ ; 但是如果只有increase_key, 那么modify为均摊O(1)

```

使用 pb\_ds 的优先队列优化 Dijkstra: 如果目标点的 dist 存在优先队列中, 直接修改; 否则 push.

```

1  typedef priority_queue<pair<int, int> > heap;
2  vector<heap::point_iterator> id;
3  id.reserve(MAXM);
4
5  // 松弛时
6  if (id[y] != 0)
7      pq.modify(id[y], make_pair(-dist[y], y));
8  else
9      id[y] = pq.push(make_pair(-dist[y], y));

```

#### 12.4.4 Tree

```

1  /* 各种树的封装 */
2  /* 定义格式为 tree<key_type, value_type, cmp_function, tree_type_tag,
   ↳ [tree_order_statistics_node_update]> T; */
3  /* 最后一个可选的, 加上会额外获得 order_of_key() 和 find_by_order(), 第二个如果没有
   ↳ 映射类型则为 null_type */
4  /* tree_tag 可以是 rb_tree_tag(红黑树), splay_tree_tag (伸展树) */
5  typedef tree<int, int, std::less<int>, rb_tree_tag> pbds_map;
6  pbds_map t; // 大概和 map<int, int> mp
   ↳ 差不多
7  t.insert(make_pair(key, value)); t[key] = value; // 插入元素
8  for (pbds_map::iterator itor = t.begin(); itor != t.end(); itor++) { // 遍历
9      std::cout << itor->first << " " << itor->second << std::endl;
10 }
11 pbds_map::iterator ft = t.find(key); // 根据 key 查找元素
12 t.erase(key); // 根据 key 删除元素
13 t.clear(); // 清空
14 int len = t.size(); bool isEmpty = t.empty();
15 pbds_map t1;
16 t.join(t1); /* 合并 */ t.split(v, t1); // key <= v 的留在 t 中, 其余分到
   ↳ t1 中
17 int lp = t.lower_bound(x) - t, up = t.upper_bound(x) - t;
18 pbds_map::iterator r = t.find_by_order(k); // 找到树中第 k+1 大/小的数
19 int order = t.order_of_key(k); // 返回 key 为 k 的元素是第
   ↳ 几大
20
21 /* 第一个参数必须为字符串类型, tag 也有别的 tag, 但 pat 最快, 与 tree 相同,
   ↳ node_update 支持自定义 */

```

```

22 typedef trie<string, null_type, trie_string_access_traits<>, pat_trie_tag,
23   ~trie_prefix_search_node_update> tr;
24 tr.insert(s);           // 插入 s
25 tr.erase(s);           // 删除 s
26 tr.join(b);            // 将另一棵 Trie 树 b 并入 tr
27 // 遍历 Trie 树
28 // pair 中第一个是起始迭代器, 第二个是终止迭代器, 遍历过去就可以找到所有字符串了。
29 pair range = base.prefix_range(x);
30 for(tr::iterator it = range.first; it != range.second; it++)
    cout << *it << ' ' << endl;

```

## 12.5 重载哈希函数

当将  $\text{pair} < \text{int}, \text{int} >$  作为  $\text{unordered\_map}$  的 key 的时候, 需要重载哈希函数:

```

1 struct hashfunc {
2     template<typename T, typename U>
3     size_t operator()(const pair<T, U> &i) const {
4         return hash<T>()(x.first) ^ hash<U>()(x.second);
5     }
6 };
7 unordered_map<pair<int, int>, int, hashfunc> func_map;

```

或者:

```

1 struct pairhash {
2     template<class T1, class T2>
3     size_t operator() (const pair<T1, T2> &x) const {
4         hash<T1> h1;
5         hash<T2> h2;
6         return h1(x.first) ^ h2(x.second);
7     }
8 };

```

# 附录

## 常用积分表 & 级数表



This page intentionally left blank.

## 附录IV 积 分 表

(一) 含有  $ax+b$  的积分

1.  $\int \frac{dx}{x\sqrt{ax+b}} = \frac{1}{\sqrt{b}} \ln \left| \frac{\sqrt{ax+b} - \sqrt{b}}{\sqrt{ax+b} + \sqrt{b}} \right| + C \quad (b > 0),$
  2.  $\int \frac{dx}{x\sqrt{ax+b}} = \frac{1}{\sqrt{-b}} \arctan \sqrt{\frac{ax+b}{-b}} + C \quad (b < 0).$
  3.  $\int \frac{x}{ax+b} dx = \frac{1}{a} (ax+b - b \ln |ax+b|) + C.$
  4.  $\int \frac{x^2}{ax+b} dx = \frac{1}{a^3} \left[ \frac{1}{2} (ax+b)^2 - 2b(ax+b) + b^2 \ln |ax+b| \right] + C.$
  5.  $\int \frac{dx}{x(ax+b)} = -\frac{1}{b} \ln \left| \frac{ax+b}{x} \right| + C.$
  6.  $\int \frac{dx}{x^2(ax+b)} = -\frac{1}{b} + \frac{a}{b^2} \ln \left| \frac{ax+b}{x} \right| + C.$
  7.  $\int \frac{x}{(ax+b)^2} dx = \frac{1}{a^2} \left( \ln |ax+b| + \frac{b}{ax+b} \right) + C.$
  8.  $\int \frac{x^2}{(ax+b)^2} dx = \frac{1}{a^3} \left( ax+b - 2b \ln |ax+b| - \frac{b^2}{ax+b} \right) + C.$
  9.  $\int \frac{dx}{x(ax+b)^2} = \frac{1}{b(ax+b)} - \frac{1}{b^2} \ln \left| \frac{ax+b}{x} \right| + C.$
- (二) 含有  $\sqrt{ax+b}$  的积分
10.  $\int \sqrt{ax+b} dx = \frac{2}{3a} \sqrt{(ax+b)^3} + C.$
  11.  $\int x\sqrt{ax+b} dx = \frac{2}{15a^2} (3ax-2b) \sqrt{(ax+b)^3} + C.$
  12.  $\int x^2\sqrt{ax+b} dx = \frac{2}{105a^3} (15a^2x^2 - 12abx + 8b^2) \sqrt{(ax+b)^3} + C.$
  13.  $\int \frac{x}{\sqrt{ax+b}} dx = \frac{2}{3a^2} (ax-2b) \sqrt{ax+b} + C.$
  14.  $\int \frac{x^2}{\sqrt{ax+b}} dx = \frac{2}{15a^3} (3a^2x^2 - 4abx + 8b^2) \sqrt{ax+b} + C.$
  15.  $\int \frac{dx}{x\sqrt{ax+b}} = \begin{cases} \frac{1}{\sqrt{b}} \ln \left| \frac{\sqrt{ax+b} - \sqrt{b}}{\sqrt{ax+b} + \sqrt{b}} \right| + C & (b > 0), \\ \frac{2}{\sqrt{-b}} \arctan \sqrt{\frac{ax+b}{-b}} + C & (b < 0). \end{cases}$
  16.  $\int \frac{dx}{x^2\sqrt{ax+b}} = \frac{\sqrt{ax+b}}{bx} - \frac{a}{2b} \int \frac{dx}{x\sqrt{ax+b}}.$
  17.  $\int \frac{\sqrt{ax+b}}{x} dx = 2\sqrt{ax+b} + b \int \frac{dx}{x\sqrt{ax+b}}.$
  18.  $\int \frac{\sqrt{ax+b}}{x^2} dx = -\frac{\sqrt{ax+b}}{x} + \frac{a}{2} \int \frac{dx}{x\sqrt{ax+b}}.$
- (三) 含有  $x^2 \pm a^2$  的积分
19.  $\int \frac{dx}{x^2+a^2} = \frac{1}{a} \arctan \frac{x}{a} + C.$
  20.  $\int \frac{dx}{(x^2+a^2)^n} = \frac{2(n-1)a^2(x^2+a^2)^{n-1} + 2(n-1)a^2}{(x^2+a^2)^{n-1}} \int \frac{dx}{(x^2+a^2)^{n-1}}.$
  21.  $\int \frac{dx}{x^2-a^2} = \frac{1}{2a} \ln \left| \frac{x-a}{x+a} \right| + C.$
- (四) 含有  $ax^2+b$  ( $a>0$ ) 的积分
22.  $\int \frac{dx}{ax^2+b} = \begin{cases} \frac{1}{\sqrt{ab}} \arctan \sqrt{\frac{a}{b}} x + C & (b > 0), \\ \frac{1}{2\sqrt{-ab}} \ln \left| \frac{\sqrt{ax}-\sqrt{-b}}{\sqrt{ax}+\sqrt{-b}} \right| + C & (b < 0). \end{cases}$
  23.  $\int \frac{dx}{ax^2+b} = \frac{1}{2a} \ln |ax^2+b| + C.$
  24.  $\int \frac{x^2}{ax^2+b} dx = \frac{x-b}{a} \int \frac{dx}{ax^2+b}.$
  25.  $\int \frac{dx}{x(ax^2+b)} = \frac{1}{2b} \ln \frac{x^2}{|ax^2+b|} + C.$
  26.  $\int \frac{dx}{x^2(ax^2+b)} = -\frac{1}{b} \frac{a}{x} \int \frac{dx}{ax^2+b}.$
  27.  $\int \frac{dx}{x^3(ax^2+b)} = \frac{a}{2b^2} \ln \frac{|ax^2+b|}{x^2} - \frac{1}{2bx^2} + C.$

$$28. \int \frac{dx}{(ax^2+bx+c)^2} = \frac{x}{2b(ax^2+b)} + \frac{1}{2b} \int \frac{dx}{ax^2+b}.$$

(五) 含有  $ax^2+bx+c$  ( $a>0$ ) 的积分

$$29. \int \frac{dx}{ax^2+bx+c} = \begin{cases} \frac{2}{\sqrt{4ac-b^2}} \arctan \frac{2ax+b}{\sqrt{4ac-b^2}} + C & (b^2 < 4ac), \\ \frac{1}{\sqrt{b^2-4ac}} \ln \left| \frac{2ax+b-\sqrt{b^2-4ac}}{2ax+b+\sqrt{b^2-4ac}} \right| + C & (b^2 > 4ac). \end{cases}$$

$$30. \int \frac{x}{ax^2+bx+c} dx = \frac{1}{2a} \ln |ax^2+bx+c| - \frac{b}{2a} \int \frac{dx}{ax^2+bx+c}.$$

(六) 含有  $\sqrt{x^2+a^2}$  ( $a>0$ ) 的积分

$$31. \int \frac{dx}{\sqrt{x^2+a^2}} = \operatorname{arsh} \frac{x}{a} + C_1 = \ln(x + \sqrt{x^2+a^2}) + C.$$

$$32. \int \frac{dx}{\sqrt{(x^2+a^2)^3}} = \frac{x}{a^2 \sqrt{x^2+a^2}} + C.$$

$$33. \int \frac{x}{\sqrt{x^2+a^2}} dx = \sqrt{x^2+a^2} + C.$$

$$34. \int \frac{x}{\sqrt{(x^2+a^2)^3}} dx = -\frac{1}{\sqrt{x^2+a^2}} + C.$$

$$35. \int \frac{x^2}{\sqrt{x^2+a^2}} dx = \frac{x}{2} \sqrt{x^2+a^2} - \frac{a^2}{2} \ln(x + \sqrt{x^2+a^2}) + C.$$

$$36. \int \frac{x^2}{\sqrt{(x^2+a^2)^3}} dx = -\frac{x}{\sqrt{x^2+a^2}} + \ln(x + \sqrt{x^2+a^2}) + C.$$

$$37. \int \frac{dx}{x \sqrt{x^2+a^2}} = \frac{1}{a} \ln \frac{\sqrt{x^2+a^2}-a}{|x|} + C.$$

$$38. \int \frac{dx}{x^2 \sqrt{x^2+a^2}} = -\frac{\sqrt{x^2+a^2}}{a^2 x} + C.$$

$$39. \int \sqrt{x^2+a^2} dx = \frac{x}{2} \sqrt{x^2+a^2} + \frac{a^2}{2} \ln(x + \sqrt{x^2+a^2}) + C.$$

$$40. \int \sqrt{(x^2+a^2)^3} dx = \frac{x}{8} (2x^2+5a^2) \sqrt{x^2+a^2} + \frac{3}{8} a^4 \ln(x + \sqrt{x^2+a^2}) + C.$$

$$41. \int x \sqrt{x^2+a^2} dx = \frac{1}{3} \sqrt{(x^2+a^2)^3} + C.$$

$$42. \int x^2 \sqrt{x^2+a^2} dx = \frac{x}{8} (2x^2+a^2) \sqrt{x^2+a^2} - \frac{a^4}{8} \ln(x + \sqrt{x^2+a^2}) + C.$$

$$43. \int \frac{\sqrt{x^2+a^2}}{x} dx = \sqrt{x^2+a^2} + a \ln \frac{\sqrt{x^2+a^2}-a}{|x|} + C.$$

$$44. \int \frac{\sqrt{x^2+a^2}}{x^2} dx = -\frac{\sqrt{x^2+a^2}}{x} + \ln(x + \sqrt{x^2+a^2}) + C.$$

(七) 含有  $\sqrt{x^2-a^2}$  ( $a>0$ ) 的积分

$$45. \int \frac{dx}{\sqrt{x^2-a^2}} = \frac{x}{a} \operatorname{arch} \frac{|x|}{a} + C_1 = \ln|x + \sqrt{x^2-a^2}| + C.$$

$$46. \int \frac{dx}{\sqrt{(x^2-a^2)^3}} = -\frac{x}{a^2 \sqrt{x^2-a^2}} + C.$$

$$47. \int \frac{x}{\sqrt{x^2-a^2}} dx = \sqrt{x^2-a^2} + C.$$

$$48. \int \frac{x}{\sqrt{(x^2-a^2)^3}} dx = -\frac{1}{\sqrt{x^2-a^2}} + C.$$

$$49. \int \frac{x^2}{\sqrt{x^2-a^2}} dx = \frac{x}{2} \sqrt{x^2-a^2} + \frac{a^2}{2} \ln|x + \sqrt{x^2-a^2}| + C.$$

$$50. \int \frac{x^2}{\sqrt{(x^2-a^2)^3}} dx = -\frac{x}{\sqrt{x^2-a^2}} + \ln|x + \sqrt{x^2-a^2}| + C.$$

$$51. \int \frac{dx}{x \sqrt{x^2-a^2}} = \frac{1}{a} \arccos \frac{a}{|x|} + C.$$

$$52. \int \frac{dx}{x^2 \sqrt{x^2-a^2}} = \frac{\sqrt{x^2-a^2}}{a^2 x} + C.$$

$$53. \int \sqrt{x^2-a^2} dx = \frac{x}{2} \sqrt{x^2-a^2} - \frac{a^2}{2} \ln|x + \sqrt{x^2-a^2}| + C.$$

$$54. \int \sqrt{(x^2-a^2)^3} dx = \frac{x}{8} (2x^2-5a^2) \sqrt{x^2-a^2} + \frac{3}{8} a^4 \ln|x + \sqrt{x^2-a^2}| + C.$$

$$55. \int x \sqrt{x^2-a^2} dx = \frac{1}{3} \sqrt{(x^2-a^2)^3} + C.$$

$$56. \int x^2 \sqrt{x^2-a^2} dx = \frac{x}{8} (2x^2-a^2) \sqrt{x^2-a^2} - \frac{a^4}{8} \ln|x + \sqrt{x^2-a^2}| + C.$$

$$57. \int \frac{\sqrt{x^2-a^2}}{x} dx = \sqrt{x^2-a^2} - a \arccos \frac{a}{|x|} + C.$$

$$58. \int \frac{\sqrt{x^2-a^2}}{x^2} dx = -\frac{\sqrt{x^2-a^2}}{x} + \ln|x+\sqrt{x^2-a^2}| + C.$$

(九) 含有  $\sqrt{a^2-x^2}$  ( $a>0$ ) 的积分

$$59. \int \frac{dx}{\sqrt{a^2-x^2}} = \arcsin \frac{x}{a} + C.$$

$$60. \int \frac{dx}{\sqrt{(a^2-x^2)^3}} = \frac{x}{a^2\sqrt{a^2-x^2}} + C.$$

$$61. \int \frac{x}{\sqrt{a^2-x^2}} dx = -\sqrt{a^2-x^2} + C.$$

$$62. \int \frac{x}{\sqrt{(a^2-x^2)^3}} dx = \frac{1}{\sqrt{a^2-x^2}} + C.$$

$$63. \int \frac{x^2}{\sqrt{a^2-x^2}} dx = -\frac{x}{2}\sqrt{a^2-x^2} + \frac{a^2}{2}\arcsin \frac{x}{a} + C.$$

$$64. \int \frac{x^2}{\sqrt{(a^2-x^2)^3}} dx = \frac{1}{a} \ln \frac{a-\sqrt{a^2-x^2}}{\sqrt{a^2-x^2}} - \arcsin \frac{x}{a} + C.$$

$$65. \int \frac{dx}{x\sqrt{a^2-x^2}} = \frac{1}{a} \ln \frac{a-\sqrt{a^2-x^2}}{|x|} + C.$$

$$66. \int \frac{dx}{x^2\sqrt{a^2-x^2}} = -\frac{\sqrt{a^2-x^2}}{a^2x} + C.$$

$$67. \int \frac{\sqrt{a^2-x^2}}{dx} dx = \frac{x}{2}\sqrt{a^2-x^2} + \frac{a^2}{2}\arcsin \frac{x}{a} + C.$$

$$68. \int \sqrt{(a^2-x^2)^3} dx = \frac{x}{8}(5a^2-2x^2)\sqrt{a^2-x^2} + \frac{3}{8}a^4\arcsin \frac{x}{a} + C.$$

$$69. \int x\sqrt{a^2-x^2} dx = -\frac{1}{3}\sqrt{(a^2-x^2)^3} + C.$$

$$70. \int x^2\sqrt{a^2-x^2} dx = \frac{x}{8}(2x^2-a^2)\sqrt{a^2-x^2} + \frac{a^4}{8}\arcsin \frac{x}{a} + C.$$

$$71. \int \frac{\sqrt{a^2-x^2}}{x} dx = \sqrt{a^2-x^2} + a \ln \frac{a-\sqrt{a^2-x^2}}{|x|} + C.$$

$$72. \int \frac{\sqrt{a^2-x^2}}{x^2} dx = -\frac{\sqrt{a^2-x^2}}{x} - \arcsin \frac{x}{a} + C.$$

(十) 含有  $\sqrt{\pm ax^2+bx+c}$  ( $a>0$ ) 的积分

$$73. \int \frac{dx}{\sqrt{ax^2+bx+c}} = \frac{1}{\sqrt{a}} \ln|2ax+b+2\sqrt{a}\sqrt{ax^2+bx+c}| + C.$$

$$74. \int \sqrt{ax^2+bx+c} dx = \frac{2ax+b}{4a}\sqrt{ax^2+bx+c} + \frac{4ac-b^2}{8\sqrt{a^3}} \ln|2ax+b+2\sqrt{a}\sqrt{ax^2+bx+c}| + C.$$

$$75. \int \frac{x}{\sqrt{ax^2+bx+c}} dx = \frac{1}{a} \sqrt{ax^2+bx+c} - \frac{b}{2\sqrt{a^3}} \ln|2ax+b+2\sqrt{a}\sqrt{ax^2+bx+c}| + C.$$

$$76. \int \frac{dx}{\sqrt{c+bx-ax^2}} = \frac{1}{\sqrt{a}} \arcsin \frac{2ax-b}{\sqrt{b^2+4ac}} + C.$$

$$77. \int \sqrt{c+bx-ax^2} dx = \frac{2ax-b}{4a}\sqrt{c+bx-ax^2} + \frac{b^2+4ac}{8\sqrt{a^3}} \arcsin \frac{2ax-b}{\sqrt{b^2+4ac}} + C.$$

$$78. \int \frac{x}{\sqrt{c+bx-ax^2}} dx = -\frac{1}{a} \sqrt{c+bx-ax^2} + \frac{b}{2\sqrt{a^3}} \arcsin \frac{2ax-b}{\sqrt{b^2+4ac}} + C.$$

(十一) 含有  $\sqrt{\frac{x-a}{x-b}}$  或  $\sqrt{(x-a)(b-x)}$  的积分

$$79. \int \frac{\sqrt{x-a}}{x-b} dx = (x-b) \sqrt{\frac{x-a}{x-b}} + (b-a) \ln(\sqrt{|x-a|} + \sqrt{|x-b|}) + C.$$

$$80. \int \sqrt{\frac{x-a}{b-x}} dx = (x-b) \sqrt{\frac{x-a}{b-x}} + (b-a) \arcsin \sqrt{\frac{x-a}{b-a}} + C.$$

$$81. \int \frac{dx}{\sqrt{(x-a)(b-x)}} = 2 \arcsin \sqrt{\frac{x-a}{b-a}} + C \quad (a < b).$$

$$82. \int \sqrt{(x-a)(b-x)} dx = \frac{2x-a-b}{4} \sqrt{(x-a)(b-x)} + \frac{(b-a)^2}{4} \arcsin \sqrt{\frac{x-a}{b-a}} + C \quad (a < b).$$

(十二) 含有三角函数的积分

$$83. \int \sin x dx = -\cos x + C.$$

84.  $\int \cos x dx = \sin x + C.$
85.  $\int \tan x dx = -\ln |\cos x| + C.$
86.  $\int \cot x dx = \ln |\sin x| + C.$
87.  $\int \sec x dx = \ln \left| \tan \left( \frac{\pi}{4} + \frac{x}{2} \right) \right| + C = \ln |\sec x + \tan x| + C.$
88.  $\int \csc x dx = \ln \left| \tan \frac{x}{2} \right| + C = \ln |\csc x - \cot x| + C.$
89.  $\int \sec^2 x dx = \tan x + C.$
90.  $\int \csc^2 x dx = -\cot x + C.$
91.  $\int \sec x \tan x dx = \sec x + C.$
92.  $\int \csc x \cot x dx = -\csc x + C.$
93.  $\int \sin^2 x dx = \frac{x}{2} - \frac{1}{4} \sin 2x + C.$
94.  $\int \cos^2 x dx = \frac{x}{2} + \frac{1}{4} \sin 2x + C.$
95.  $\int \sin^n x dx = -\frac{1}{n} \sin^{n-1} x \cos x + \frac{n-1}{n} \int \sin^{n-2} x dx.$
96.  $\int \cos^n x dx = \frac{1}{n} \cos^{n-1} x \sin x + \frac{n-1}{n} \int \cos^{n-2} x dx.$
97.  $\int \frac{dx}{\sin^n x} = -\frac{1}{n-1} \cdot \frac{\cos x}{\sin^{n-1} x} + \frac{n-2}{n-1} \int \frac{dx}{\sin^{n-2} x}.$
98.  $\int \frac{dx}{\cos^n x} = \frac{1}{n-1} \cdot \frac{\sin x}{\cos^{n-1} x} + \frac{n-2}{n-1} \int \frac{dx}{\cos^{n-2} x}.$
99.  $\int \cos^m x \sin^n x dx = \frac{1}{m+n} \cos^{m-1} x \sin^{n+1} x + \frac{m-1}{m+n} \int \cos^{m-2} x \sin^n x dx$   
 $= \frac{1}{m+n} \cos^{m-1} x \sin^{n-1} x + \frac{n-1}{m+n} \int \cos^m x \sin^{n-2} x dx.$
100.  $\int \sin ax \cos bx dx = -\frac{1}{2(a+b)} \cos(ax+b)x - \frac{1}{2(a-b)} \cos(ax-b)x + C.$
101.  $\int \sin ax \sin bx dx = -\frac{1}{2(a+b)} \sin(ax+b)x + \frac{1}{2(a-b)} \sin(ax-b)x + C.$
102.  $\int \cos ax \cos bx dx = \frac{1}{2(a+b)} \sin(ax+b)x + \frac{1}{2(a-b)} \sin(ax-b)x + C.$
103.  $\int \frac{dx}{a+b \sin x} = \frac{1}{\sqrt{a^2-b^2}} \arctan \frac{a \tan \frac{x}{2} + b}{\sqrt{a^2-b^2}} + C \quad (a^2 > b^2).$
104.  $\int \frac{dx}{a+b \sin x} = \frac{1}{\sqrt{b^2-a^2}} \ln \left| \frac{a \tan \frac{x}{2} + b - \sqrt{b^2-a^2}}{a \tan \frac{x}{2} + b + \sqrt{b^2-a^2}} \right| + C \quad (a^2 < b^2).$
105.  $\int \frac{dx}{a+b \cos x} = \frac{2}{a-b} \sqrt{\frac{a+b}{a-b}} \arctan \left( \sqrt{\frac{a-b}{a+b}} \tan \frac{x}{2} \right) + C \quad (a^2 > b^2).$
106.  $\int \frac{dx}{a+b \cos x} = \frac{1}{a+b} \sqrt{\frac{a+b}{b-a}} \ln \left| \frac{\tan \frac{x}{2} + \sqrt{\frac{a+b}{b-a}}}{\tan \frac{x}{2} - \sqrt{\frac{a+b}{b-a}}} \right| + C \quad (a^2 < b^2).$
107.  $\int \frac{dx}{a^2 \cos^2 x + b^2 \sin^2 x} = \frac{1}{ab} \arctan \left( \frac{b}{a} \tan x \right) + C.$
108.  $\int \frac{dx}{a^2 \cos^2 x - b^2 \sin^2 x} = \frac{1}{2ab} \ln \left| \frac{b \tan x + a}{b \tan x - a} \right| + C.$
109.  $\int x \sin ax dx = \frac{1}{a^2} \sin ax - \frac{1}{a} x \cos ax + C.$
110.  $\int x^2 \sin ax dx = -\frac{1}{a} x^2 \cos ax + \frac{2}{a^2} x \sin ax + \frac{2}{a^3} \cos ax + C.$
111.  $\int x \cos ax dx = \frac{1}{a} x^2 \cos ax + \frac{1}{a^2} x \sin ax + C.$
112.  $\int x^2 \cos ax dx = \frac{1}{a} x^2 \sin ax + \frac{2}{a^2} x \cos ax - \frac{2}{a^3} \sin ax + C.$
- (+ -) 含有反三角函数的积分(其中  $a > 0$ )
113.  $\int \arcsin \frac{x}{a} dx = x \arcsin \frac{x}{a} + \sqrt{a^2 - x^2} + C.$
114.  $\int x \arcsin \frac{x}{a} dx = \left( \frac{x^2}{2} - \frac{a^2}{4} \right) \arcsin \frac{x}{a} + \frac{x}{4} \sqrt{a^2 - x^2} + C.$
115.  $\int x^2 \arcsin \frac{x}{a} dx = \frac{x^3}{3} \arcsin \frac{x}{a} + \frac{1}{9} (x^2 + 2a^2) \sqrt{a^2 - x^2} + C.$
116.  $\int \arccos \frac{x}{a} dx = x \arccos \frac{x}{a} - \sqrt{a^2 - x^2} + C.$

## (十四) 含有对数函数的积分

$$117. \int x \arccos \frac{x}{a} dx = \left( \frac{x^2}{2} - \frac{a^2}{4} \right) \arccos \frac{x}{a} - \frac{x}{4} \sqrt{a^2 - x^2} + C.$$

$$118. \int x^2 \arccos \frac{x}{a} dx = \frac{x^3}{3} \arccos \frac{x}{a} - \frac{1}{9} (x^2 + 2a^2) \sqrt{a^2 - x^2} + C.$$

$$119. \int \arctan \frac{x}{a} dx = \arctan \frac{x}{a} - \frac{a}{2} \ln(a^2 + x^2) + C.$$

$$120. \int x \arctan \frac{x}{a} dx = \frac{1}{2} (a^2 + x^2) \arctan \frac{x}{a} - \frac{a}{2} x + C.$$

$$121. \int x^2 \arctan \frac{x}{a} dx = \frac{x^3}{3} \arctan \frac{x}{a} - \frac{a}{6} x^2 + \frac{a^3}{6} \ln(a^2 + x^2) + C.$$

## (十五) 含有指数函数的积分

$$122. \int a^x dx = \frac{1}{\ln a} a^x + C.$$

$$123. \int e^{ax} dx = \frac{1}{a} e^{ax} + C.$$

$$124. \int x e^{ax} dx = \frac{1}{a} x e^{ax} - \frac{n}{a} \int x^{n-1} e^{ax} dx.$$

$$125. \int x^n e^{ax} dx = \frac{1}{a} x^n e^{ax} - \frac{n}{a} \int x^{n-1} e^{ax} dx.$$

$$126. \int x a^x dx = \frac{x}{\ln a} a^x - \frac{1}{(\ln a)^2} a^x + C.$$

$$127. \int x^n a^x dx = \frac{1}{\ln a} x^n a^x - \frac{n}{(\ln a)^2} a^x + C.$$

$$128. \int e^{ax} \sin bx dx = \frac{1}{a^2 + b^2} e^{ax} (a \sin bx - b \cos bx) + C.$$

$$129. \int e^{ax} \cos bx dx = \frac{1}{a^2 + b^2} e^{ax} (b \sin bx + a \cos bx) + C.$$

$$130. \int e^{ax} \sin^n bx dx = \frac{1}{a^2 + b^2} e^{ax} \sin^{n-1} bx (a \sin bx - n b \cos bx) +$$

$$\frac{n(n-1)b^2}{a^2 + b^2} \int e^{ax} \sin^{n-2} bx dx.$$

$$131. \int e^{ax} \cos^n bx dx = \frac{1}{a^2 + b^2} e^{ax} \cos^{n-1} bx (a \cos bx + n b \sin bx) +$$

$$\frac{n(n-1)b^2}{a^2 + b^2} \int e^{ax} \cos^{n-2} bx dx.$$

## (十六) 定积分

$$132. \int \ln x dx = x \ln x - x + C.$$

$$133. \int \frac{dx}{x \ln x} = \ln |\ln x| + C.$$

$$134. \int x^n \ln x dx = \frac{1}{n+1} x^{n+1} \left( \ln x - \frac{1}{n+1} \right) + C.$$

$$135. \int (\ln x)^n dx = x (\ln x)^n - n \int (\ln x)^{n-1} dx.$$

$$136. \int x^m (\ln x)^n dx = \frac{1}{m+1} x^{m+1} (\ln x)^n - \frac{n}{m+1} \int x^m (\ln x)^{n-1} dx.$$

## (十七) 含有双曲函数的积分

$$137. \int \operatorname{sh} x dx = \operatorname{ch} x + C.$$

$$138. \int \operatorname{ch} x dx = \operatorname{sh} x + C.$$

$$139. \int \operatorname{th} x dx = \ln \operatorname{ch} x + C.$$

$$140. \int \operatorname{sh}^2 x dx = -\frac{x}{2} + \frac{1}{4} \operatorname{sh} 2x + C.$$

$$141. \int \operatorname{ch}^2 x dx = \frac{x}{2} + \frac{1}{4} \operatorname{sh} 2x + C.$$

## (十八) 定积分

$$142. \int_{-\pi}^{\pi} \cos nx dx = \int_{-\pi}^{\pi} \sin nx dx = 0.$$

$$143. \int_{-\pi}^{\pi} \cos mx \sin nx dx = 0.$$

$$144. \int_{-\pi}^{\pi} \cos mx \cos nx dx = \begin{cases} 0, & m \neq n, \\ \pi, & m = n. \end{cases}$$

$$145. \int_{-\pi}^{\pi} \sin mx \sin nx dx = \begin{cases} 0, & m \neq n, \\ \pi, & m = n. \end{cases}$$

$$146. \int_0^{\pi} \sin m \sin nx dx = \int_0^{\pi} \cos mx \cos nx dx = \begin{cases} 0, & m \neq n, \\ \frac{\pi}{2}, & m = n. \end{cases}$$

$$147. I_n = \int_0^{\frac{\pi}{2}} \sin^n x dx = \int_0^{\frac{\pi}{2}} \cos^n x dx,$$

$$I_n = \frac{n-1}{n} I_{n-2}$$

$$= \begin{cases} \frac{n-1}{n} \cdot \frac{n-3}{n-2} \cdot \dots \cdot \frac{4}{3} \cdot \frac{2}{3} & (n \text{ 为大于 } 1 \text{ 的正奇数}), I_1 = 1, \\ \frac{n-1}{n} \cdot \frac{n-3}{n-2} \cdot \dots \cdot \frac{3}{2} \cdot \frac{1}{2} \cdot \frac{\pi}{2} & (n \text{ 为正偶数}), I_0 = \frac{\pi}{2}. \end{cases}$$

#### 常用的无穷级数

##### 1. 三角函数

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}$$

$$\begin{aligned}\tan x &= \sum_{n=0}^{\infty} \frac{U_{2n+1} x^{2n+1}}{(2n+1)!} \\ &= \sum_{n=1}^{\infty} \frac{(-1)^{n-1} 2^{2n} (2^{2n} - 1) B_{2n} x^{2n-1}}{(2n)!} \\ &= x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^7}{315} + \dots, \quad \text{for } |x| < \frac{\pi}{2}\end{aligned}$$

$$\begin{aligned}\csc x &= \sum_{n=0}^{\infty} \frac{(-1)^{n+1} 2 (2^{2n-1} - 1) B_{2n} x^{2n-1}}{(2n)!} \\ &= \frac{1}{x} + \frac{x}{6} + \frac{7x^3}{360} + \frac{31x^5}{15120} + \dots, \quad \text{for } 0 < |x| < \pi\end{aligned}$$

$$\begin{aligned}\sec x &= \sum_{n=0}^{\infty} \frac{U_{2n} x^{2n}}{(2n)!} \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n E_n x^{2n}}{(2n)!} \\ &= 1 + \frac{x^2}{2} + \frac{5x^4}{24} + \frac{61x^6}{720} + \dots, \quad \text{for } |x| < \frac{\pi}{2}\end{aligned}$$

$$\begin{aligned}\cot x &= \sum_{n=0}^{\infty} \frac{(-1)^n 2^{2n} B_{2n} x^{2n-1}}{(2n)!} \\ &= \frac{1}{x} - \frac{x}{3} - \frac{x^3}{45} - \frac{2x^5}{945} - \dots, \quad \text{for } 0 < |x| < \pi\end{aligned}$$

$U_n$  是  $n$  次上/下数,

$B_n$  是  $n$  次伯努利数,

$E_n$  (下面的) 是  $n$  次欧拉数

##### 3. 对数函数

$$\ln z = \sum_{n=1}^{\infty} \frac{-(-1)^n}{n} (z-1)^n$$

更有效率的级数是:

$$\ln z = 2 \sum_{n=0}^{\infty} \frac{1}{2n+1} \left( \frac{z-1}{z+1} \right)^{2n+1}$$

对于任何其他底数  $\beta$  , 我们使用:

$$\log_{\beta} x = \frac{\ln x}{\ln \beta}$$

##### 4. 计算特殊常数的

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left( \frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

$$\pi = \frac{1}{2^6} \sum_{n=0}^{\infty} \frac{(-1)^n}{2^{10n}} \left( -\frac{2^5}{4n+1} - \frac{1}{4n+3} + \frac{2^8}{10n+1} - \frac{2^6}{10n+3} - \frac{2^2}{10n+5} - \frac{2^2}{10n+7} + \frac{1}{10n+9} \right)$$

$$\pi = \frac{426880\sqrt{10005}}{\sum_{k=0}^{\infty} \frac{(6n)!}{(n!)^3} \frac{(545140134n+13591409)}{(3n)!} (-640320)^3 n}}$$

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

##### 5. 特殊的函数

$$(x+y)^{\alpha} = \sum_{k=0}^{\infty} \binom{\alpha}{k} x^{\alpha-k} y^k$$

#### 2. 反三角函数

$$\begin{aligned}\arcsin z &= z + \left( \frac{1}{2} \right) \frac{z^3}{3} + \left( \frac{1 \cdot 3}{2 \cdot 4} \right) \frac{z^5}{5} + \left( \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \right) \frac{z^7}{7} + \dots \\ &= \sum_{n=0}^{\infty} \left( \frac{(2n)!}{2^{2n}(n!)^2} \right) \frac{z^{2n+1}}{(2n+1)}; \quad |z| \leq 1\end{aligned}$$

$$\begin{aligned}\arccos z &= \frac{\pi}{2} - \arcsin z \\ &= \frac{\pi}{2} - \left( z + \left( \frac{1}{2} \right) \frac{z^3}{3} + \left( \frac{1 \cdot 3}{2 \cdot 4} \right) \frac{z^5}{5} + \left( \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \right) \frac{z^7}{7} + \dots \right) \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \left( \frac{(2n)!}{2^{2n}(n!)^2} \right) \frac{z^{2n+1}}{(2n+1)}; \quad |z| \leq 1\end{aligned}$$

$$\begin{aligned}\arctan z &= z - \frac{z^3}{3} + \frac{z^5}{5} - \frac{z^7}{7} + \dots \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n z^{2n+1}}{2n+1}; \quad |z| \leq 1 \quad z \neq i, -i\end{aligned}$$

$$\begin{aligned}\operatorname{arccot} z &= \frac{\pi}{2} - \arctan z \\ &= \frac{\pi}{2} - \left( z - \frac{z^3}{3} + \frac{z^5}{5} - \frac{z^7}{7} + \dots \right) \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \frac{(-1)^n z^{2n+1}}{2n+1}; \quad |z| \leq 1 \quad z \neq i, -i\end{aligned}$$

$$\begin{aligned}\operatorname{arcsec} z &= \arccos(z^{-1}) \\ &= \frac{\pi}{2} - \left( z^{-1} + \left( \frac{1}{2} \right) \frac{z^{-3}}{3} + \left( \frac{1 \cdot 3}{2 \cdot 4} \right) \frac{z^{-5}}{5} + \left( \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \right) \frac{z^{-7}}{7} + \dots \right) \\ &= \frac{\pi}{2} - \sum_{n=0}^{\infty} \left( \frac{(2n)!}{2^{2n}(n!)^2} \right) \frac{z^{-(2n+1)}}{(2n+1)}; \quad |z| \geq 1\end{aligned}$$

$$\begin{aligned}\operatorname{arccsc} z &= \arcsin(z^{-1}) \\ &= z^{-1} + \left( \frac{1}{2} \right) \frac{z^{-3}}{3} + \left( \frac{1 \cdot 3}{2 \cdot 4} \right) \frac{z^{-5}}{5} + \left( \frac{1 \cdot 3 \cdot 5}{2 \cdot 4 \cdot 6} \right) \frac{z^{-7}}{7} + \dots \\ &= \sum_{n=0}^{\infty} \left( \frac{(2n)!}{2^{2n}(n!)^2} \right) \frac{z^{-(2n+1)}}{2n+1}; \quad |z| \geq 1\end{aligned}$$

欧拉发现了反正切的更有效的级数:

$$\arctan x = \frac{x}{1+x^2} \sum_{n=0}^{\infty} \prod_{k=1}^n \frac{2kx^2}{(2k+1)(1+x^2)}.$$

注意对  $n=0$  在和中的项是空积 1。

# 现场赛自用 vim 配置

```
1 set nu
2 syntax on
3 set ts=4
4 set shiftwidth=4
5 set cindent
6 set ruler
7 set mouse=a
8 set smartindent
9 set showcmd
10 set expandtab
11 colo evening
12 map <F9> :!g++ -O2 -DLOCAL -std=c++11 % -o %<<cr>
13 map <F10> :!clear && ./%<<cr>
```

## 对拍脚本

### Windows

```
1 :loop
2 datagen.exe
3 problem.exe
4 std.exe
5 fc problem.out std.out
6 if errorlevel 1 pause
7 goto loop
```

### Linux

```
1#!/bin/bash
2while true; do
3    ./data > data.in && ./std < data.in > std.out && ./problem < data.in >
4        ↵ problem.out
5    if diff std.out problem.out; then printf "Accepted\n"
6    else
7        printf "Wrong Answer\n"
8        exit 0
9    fi
done
```