

ACM 模板

dnvtmf

2015

目录

1 数据结构	5
1.1 RMQ 相关	5
1.2 树链剖分	5
1.3 伸展树 Splay	6
1.4 Treap	9
1.5 ST 表	11
1.6 莫队算法分块	11
1.7 并查集 Disjoint Set	12
2 动态规划	13
2.1 最长上升公共子序列 LIS	13
3 图论	14
3.1 拓扑排序	14
3.2 欧拉回路 Euler, 哈密顿回路 Hamilton	14
3.3 无向图的桥, 割点, 双连通分量	17
3.4 最短路 shortest path	19
3.5 最大流 maximum flow	23
3.6 最小割 minimum cut	27
3.7 分数规划 Fractional Programming	29
3.8 最大闭权图 maximum weight closure of a graph	29
3.9 最大密度子图 Maximum Density Subgraph	30
3.10 二分图的最小点权覆盖集与最大点权独立集	30
3.11 最小费用最大流 minimum cost flow	31
3.12 有上下界的网络流	34
3.13 最近公共祖先 LCA	35
4 数学专题	37
4.1 素数 Prime	37
4.2 因式分解 Factorization 和约数	38
4.3 欧拉函数 Euler	41

4.4 快速幂快速乘	41
4.5 最大公约数 GCD	42
4.6 逆元 Inverse	43
4.7 模运算 Module	44
4.8 幂 p 的原根	46
4.9 中国剩余定理和线性同余方程组	47
4.10 组合与组合恒等式	47
4.11 排列 Permutation	49
4.12 全排列 Full Permutation	50
4.13 母函数 Generating Function	52
4.14 整数拆分和 Ferrers 图	52
4.15 鸽笼原理与 Ramsey 数	53
4.16 容斥原理	54
4.17 伪随机数的生成—梅森旋转算法	54
4.18 异或 Xor	55
4.19 博弈论 Game Theory	55
4.20 快速傅里叶变换和数论变换 (FFT 和 NTT)	56
4.21 莫比乌斯反演 Mobius	60
4.22 一些数学知识	61
5 线性代数 Linear Algebra	62
5.1 矩阵 Matrix	62
5.2 矩阵的初等变换和矩阵的逆	64
6 字符串	68
6.1 KMP 以及扩展 KMP	68
6.2 回文串 palindrome	69
6.3 哈希算法 Hash	69
6.4 后缀数组 Suffix Array	70
6.5 字典树 Trie	73
6.6 字符串循环同构的最小表示法	74
6.7 字符串问题汇总	74

7 计算几何	76
7.1 计算几何基础	76
7.2 三角形 Triangle	77
7.3 多边形 Polygon	81
7.4 圆 Circle	82
7.5 凸包 ConvexHull	83
7.6 半平面交 Half-plane Cross	84
7.7 立体几何	86
7.8 格点 Lattice Point	87
8 搜索等	88
9 分治	88
10 Java	89

1 数据结构

1.1 RMQ 相关

```
1 /*区间的rmq问题
2  * 在一维数轴上, 添加或删除若干区间 $[l, r]$ , 询问某区间 $[ql, qr]$ 内覆盖了多少个完整的区间
3  * 做法: 离线, 按照右端点排序, 然后按照左端点建立线段树保存左端点为 $l$ 的区间个数,
4  * 接着按排序结果从小到大依次操作, 遇到询问时, 查询比 $ql$ 大的区间数
5  * 遇到不能改变查询顺序的题, 应该用可持久化线段树
6  */
7 /*数组区间颜色数查询
8  问题: 给定一个数组, 要求查询某段区间内有多少种数字
9  解决: 将查询离线, 按右端点排序; 从左到右依次扫描, 扫描到第 $i$ 个位置时, 将该位置加1,
10  该位置的前驱(上一个出现一样数字的位置)减1, 然后查询所有右端点为 $i$ 的询问的一个区间和 $[l, r]$ .
11 */
```

1.2 树链剖分

```
1 //树链剖分 Heavy-Light Decomposition
2 //将一个树划分为若干个不相交的路径, 使每个结点仅在一条路径上.
3 //满足: 从结点 $u$ 到 $v$ 最多经过 $\log N$ 条路径, 以及 $\log N$ 条不在路径上的边.
4 //采用启发式划分, 即某结点选择与子树中结点数最大的儿子划分为一条路径.
5 //时间复杂度: 用其他数据结构来维护每条链, 复杂度为所选数据结构乘以 $\log N$ .
6 //用split()来进行树链剖分, 其中使用bfs进行划分操作. 对于每一个结点 $v$ , 找到它的size最大的子结点 $u$ .
7 如果 $u$ 不存在, 那么给 $v$ 分配一条新的路径, 否则 $v$ 就延续 $u$ 所属的路径.
8 //查询两个结点 $u, v$ 之间的路径是, 首先判断它们是否属于同一路径. 如果不是,
9 选择所属路径顶端结点 $h$ 的深度较大的结点, 不妨假设是 $v$ , 查询 $v$ 到 $h$ , 并令 $v = \text{father}[h]$ 继续查询, 直至 $u, v$ 属于同一路径. 最后在这条路径上查询并返回.
10 /*
11 sz[u]: 结点 $u$ 的子树的结点数
12 fa[u]: 结点 $u$ 的父结点
13 dep[u]: 结点 $u$ 在树中的深度
14 belong[u]: 结点 $u$ 所在剖分链的编号
15 id[u]: 结点 $u$ 在其路径中的编号, 由深入浅编号
16 start[p]: 链 $p$ 的第一个结点
17 len[p]: 链 $p$ 的长度
18 total: 划分链的数量
19 dfn: 树中结点的遍历顺序
20 */
21 struct edge
22 {
23     int next, to, cost;
24 } e[NUM << 1];
25 int head[NUM], tot;
26 void init()
27 {
28     memset(head, -1, sizeof(head));
29     tot = 0;
30 }
31 void add_edge(int u, int v, int w)
32 {
33     e[tot] = (edge) {head[u], v, w};
34     head[u] = tot++;
35 }
36 int sz[NUM], fa[NUM], dep[NUM], start[NUM], belong[NUM], id[NUM], len[NUM], total;
37 int dfn[NUM];
38 bool vis[NUM];
39 void split()
40 {
41     int tail = 0, top = 0;
```

```

40     dep[dfn[top++] = 1] = 0;
41     fa[1] = 0;
42     ne[1] = 0;
43     val[0] = 0;
44     while(tail < top)
45     {
46         int u = dfn[tail++];
47         for(int i = head[u]; ~i; i = e[i].next)
48             if(e[i].to != fa[u])
49             {
50                 dep[dfn[top++] = e[i].to] = dep[u] + 1;
51                 fa[e[i].to] = u;
52             }
53     }
54     memset(vis, 0, sizeof(vis));
55     total = 0;
56     while(—top >= 0)
57     {
58         int u = dfn[top], v = -1;
59         sz[u] = 1;
60         for(int i = head[u]; ~i; i = e[i].next)
61             if(vis[e[i].to])
62             {
63                 sz[u] += sz[e[i].to];
64                 if(v == -1 || sz[e[i].to] > sz[v])
65                     v = e[i].to;
66             }
67         if(v == -1)
68         {
69             id[u] = len[++total] = 1;
70             belong[start[total] = u] = total;
71         }
72         else
73         {
74             id[u] = ++len[belong[u] = belong[v]];
75             start[belong[u]] = u;
76         }
77         vis[u] = true;
78     }
79 }

```

1.3 伸展树 Splay

```

1  /// 伸展树 Splay
2  // 均摊复杂度  $O(\log n)$  最坏复杂度  $O(n)$ 
3  /* 操作
4  1. 旋转: 左右旋, 将左 (右) 孩子变为根
5  2. splay: 将结点  $x$  旋转至根  $y$  处
6  3. find: 同二叉树查找, 查找成功后 splay
7  4. remove: 查找  $x$ , 如果  $x$  无孩子或一个孩子, 删除  $x$ , splay ( $x$ ) 的根结点;  $x$  有两个孩子, 用  $x$  的后继  $y$  代替  $x$ ,
   splay ( $y$ )
8  5. join, split 合并, 分解数.
9  6. 区间操作:  $[a, b]$ , 将  $a-1$  splay 至根处,  $b+1$  至根的右孩子处, 那根的右孩子的左子树表示区间  $[a, b]$ 
10 */
11 const int NUM = 1000000 + 10;
12 struct Splay_node
13 {
14     int ch[2], fa;
15     int key;
16     int len;
17     bool flag;
18     LL val, sum;

```

```

19 void malloc(int _key)
20 {
21     ch[0] = ch[1] = fa = 0;
22     key = _key;
23     len = 1;
24     flag = false;
25     val = sum = 0;
26 }
27 };
28 struct Splay
29 {
30     Splay_node *e;
31     int tot;
32     int root;
33     Splay()
34     {
35         e = new Splay_node[NUM];
36         tot = 0;
37         root = -1;
38     }
39     ~Splay()
40     {
41         delete []e;
42     }
43     void push_down(int x)//标记下传
44     {
45         if(e[x].flag)
46         {
47             int y = e[x].ch[0];
48             if(y)
49             {
50                 e[y].flag = true;
51                 e[y].val = e[x].val;
52                 e[y].sum = e[y].val * e[y].len;
53             }
54             y = e[x].ch[1];
55             if(y)
56             {
57                 e[y].flag = true;
58                 e[y].val = e[x].val;
59                 e[y].sum = e[y].val * e[y].len;
60             }
61             e[x].flag = false;
62         }
63     }
64     void push_up(int x)//标记上传
65     {
66         e[x].sum = e[x].val;
67         e[x].len = 1;
68         if(e[x].ch[0])
69         {
70             e[x].sum += e[e[x].ch[0]].sum;
71             e[x].len += e[e[x].ch[0]].len;
72         }
73         if(e[x].ch[1])
74         {
75             e[x].sum += e[e[x].ch[1]].sum;
76             e[x].len += e[e[x].ch[1]].len;
77         }
78     }
79     //旋转操作: c = 0: 左旋, 将父节点旋转到结点x的左儿子; c = 1: 右旋, 将父节点旋转到结点x的右儿子
80     void rot(int x, int c)
81     {
82         int y = e[x].fa, z = e[y].fa;

```

```

83     push_down(y), push_down(x);
84     e[y].ch[!c] = e[x].ch[c];
85     if(e[x].ch[c]) e[e[x].ch[c]].fa = y;
86     e[y].fa = x;
87     if(z) e[z].ch[e[z].ch[1] == y] = x;
88     e[x].ch[c] = y;
89     e[x].fa = z;
90     push_up(y);
91     if(y == root) root = x;
92 }
93 //splay操作: 将结点x旋转到结点fa下面
94 void splay(int x, int fa)
95 {
96     int y, z;
97     push_down(x);
98     while((y = e[x].fa) != fa)
99     {
100         z = e[y].fa;
101         if(z == fa)
102             rot(x, e[y].ch[0] == x); // 单旋
103         else
104         {
105             if(e[z].ch[0] == y)
106             {
107                 if(e[y].ch[0] == x)
108                     rot(y, 1), rot(x, 1); // 一字旋
109                 else
110                     rot(x, 0), rot(x, 1); // 之字旋
111             }
112             else
113             {
114                 if(e[y].ch[1] == x)
115                     rot(y, 0), rot(x, 0); // 一字旋
116                 else
117                     rot(x, 1), rot(x, 0); // 之字旋
118             }
119         }
120     }
121     push_up(x);
122 }
123 void insert(int key)
124 {
125     int x = ++tot;
126     e[x].malloc(key);
127     int y = root;
128     if(y == -1)
129     {
130         root = x;
131         e[x].fa = 0;
132         return ;
133     }
134     while(true)
135     {
136         if(e[y].key < key)
137         {
138             if(e[y].ch[1]) y = e[y].ch[1];
139             else
140             {
141                 e[y].ch[1] = x;
142                 e[x].fa = y;
143                 splay(x, 0);
144                 return ;
145             }
146         }

```



```

147         else if(e[y].key > key)
148         {
149             if(e[y].ch[0]) y = e[y].ch[0];
150             else
151             {
152                 e[y].ch[0] = x;
153                 e[x].fa = y;
154                 splay(x, 0);
155                 return ;
156             }
157         }
158         else
159             return ;
160     }
161 }
162 int find(int key)
163 {
164     int x = root;
165     while(e[x].key != key)
166     {
167         x = e[x].ch[e[x].key < key];
168         if(!x)//没有找到
169             return -1;
170     }
171     return x;
172 }
173 void update(int L, int R, LL val)
174 {
175     splay(find(L - 1), 0);
176     splay(find(R + 1), root);
177     int x = e[e[root].ch[1]].ch[0];
178     e[x].flag = true;
179     e[x].val = val;
180     e[x].sum = val * e[x].len;
181 }
182 LL query(int L, int R)
183 {
184     splay(find(L - 1), 0);
185     splay(find(R + 1), root);
186     int x = e[e[root].ch[1]].ch[0];
187     return e[x].sum;
188 }
189 };

```

1.4 Treap

```

1 //随机化二叉树Treap
2 struct node
3 {
4     int key;
5     int ch[2], fix;
6 };
7 struct Treap
8 {
9     node *p;
10    int size, root;
11    Treap()
12    {
13        p = new node[NUM];
14        srand(time(0));
15        size = -1;
16        root = -1;

```

```

17 }
18 ~Treap() {delete []p;}
19 void rot(int &x, int op)//op = 0 左旋, op = 1右旋
20 {
21     int y = p[x].ch[!op];
22     p[x].ch[!op] = p[y].ch[op];
23     p[y].ch[op] = x;
24     x = y;
25 }
26 //如果当前节点的优先级比根大就旋转,
    如果当前节点是根的左儿子就右旋如果当前节点是根的右儿子就左旋.
27 void insert(int tkey, int &k = root)
28 {
29     if(k == -1)
30     {
31         k = ++size;
32         p[k].ch[0] = p[k].ch[1] = -1;
33         p[k].key = tkey;
34         p[k].fix = rand();
35     }
36     else if(tkey < p[k].key)
37     {
38         insert(tkey, p[k].ch[0]);
39         if(p[p[k].ch[0]].fix > p[k].fix)
40             rot(k, 1);
41     }
42     else
43     {
44         insert(tkey, p[k].ch[1]);
45         if(p[p[k].ch[1]].fix > p[k].fix)
46             rot(k, 0);
47     }
48 }
49 void remove(int tkey, int &k)//把要删除的节点旋转到叶节点上, 然后直接删除
50 {
51     if(k == -1) return ;
52     if(tkey < p[k].key)
53         remove(tkey, p[k].ch[0]);
54     else if(tkey > p[k].key)
55         remove(tkey, p[k].ch[1]);
56     else
57     {
58         if(p[k].ch[0] == -1 && p[k].ch[1] == -1)
59             k = -1;
60         else if(p[k].ch[0] == -1)
61             k = p[k].ch[1];
62         else if(p[k].ch[1] == -1)
63             k = p[k].ch[0];
64         else if(p[p[k].ch[0]].fix < p[p[k].ch[1]].fix)
65         {
66             rot(k, 0);
67             remove(tkey, p[k].ch[0]);
68         }
69         else
70         {
71             rot(k, 1);
72             remove(tkey, p[k].ch[1]);
73         }
74     }
75 }
76 };

```

1.5 ST 表

```
1 ///ST表(Sparse Table)
2 //对静态数组, 查询任意区间[l, r]的最大(小)值
3 //预处理O(nlog n), 查询O(1)
4 #define MAX 10000
5 int st[MAX][22]; //st表 — st[i][j]表示从第i个元素起, 连续2^j个元素的最大(小)值
6 int Log2[MAX]; //对应于数x中最大的是2的幂的区间长度, k = floor(log2(R - L + 1))
7 void pre_ST(int n, int ar[]) //n 数组长度, ar 数组
8 {
9     int i, j;
10    Log2[1] = 0;
11    for(i = 2; i <= n; i++) Log2[i] = Log2[i >> 1] + 1;
12    for(i = n - 1; i >= 0; i--)
13    {
14        st[i][0] = ar[i];
15        for(j = 1; i + (1 << j) <= n; j++)
16            st[i][j] = max(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
17    }
18 }
19 int query(int l, int r)
20 {
21     int k = Log2[r - l + 1];
22     return max(st[l][k], st[r - (1 << k) + 1][k]);
23 }
```

1.6 莫队算法分块

```
1 ///莫队算法
2 //复杂度: O(n√n · 转移的复杂度)
3 //n个数, q次区间查询
4 /*使用条件:
5 已知区间[l, r]的答案, 可在O(1) 或O(log n) 内得到[l + 1, r], [l - 1, r], [l, r + 1], [l, r - 1]的答案
6 查询可以离线*/
7 /*做法:
8 将查询以l所在的块为第一关键字, 以r为第二关键字排序, 然后从一个初始状态开始转移
9 注意: 1. 转移的时候, 要先转移使l减小, 和使r增大的部分, 后转移使l增大和使r减小的部分, 避免l > r
10        2. 转移的时候注意先加减l, r, 后加减l, r的问题.
11 */
12 int qst; //每块的大小
13 struct Query
14 {
15     int l, r, id;
16 } qry[NUM];
17 LL ans[NUM];
18 bool operator < (const Query &a, const Query &b)
19 {
20     if((a.l / qst) == (b.l / qst)) return a.r < b.r;
21     return a.l < b.l;
22 }
23 int update(int l, int r); //转移[l, r]到答案的改变
24 void solve(int n)
25 {
26     qst = sqrt(1.0 * n);
27     sort(qry, qry + Q);
28     int l = 0, r = 0;
29     int res = a[0];
30     for(i = 0; i < Q; i++)
31     {
32         while(l > qry[i].l) res += update(--l, r);
33         while(r < qry[i].r) res += update(l, ++r);
34         while(r > qry[i].r) res -= update(l, r--);
35     }
```

```

35     while(l < qry[i].l) res -= update(l++, r);
36     ans[qry[i].id] = res;
37 }
38 }
39 //分块  $O(N\sqrt{N})$ 

```

1.7 并查集 Disjoint Set

```

1 //并查集 Disjoint Set
2 int father[MAX], rk[MAX];
3 void init()
4 {
5     for(int i = 0; i < MAX; i++)
6     {
7         father[i] = i;
8         rk[i] = 0;
9     }
10 }
11 int find(int x)
12 {
13     return father[x] == x ? x : father[x] = find(father[x]);
14 }
15 void gather(int x, int y)
16 {
17     x = find(x);
18     y = find(y);
19     if(x == y) return;
20     if(rk[x] > rk[y]) father[y] = x;
21     else
22     {
23         if(rk[x] == rk[y]) rk[y]++;
24         father[x] = y;
25     }
26 }
27
28 bool same(int x, int y)
29 {
30     return find(x) == find(y);
31 }

```

2 动态规划

2.1 最长上升公共子序列 LIS

```
1 /*最长上升子序列LIS
2     给一个序列，求满足的严格递增的子序列的最大长度(或者子序列)
3     标签：dp
4     做法：dp[i]表示长度为i的子序列在第i位的最小值，每次更新时，找到最大的k使dp[k] ≤ a_i，
        将dp[k+1]的值更新为a_i。
5     可以用pre数组存储第i个数的最长子序列的前一个数。
6 */
7 /*两个互不覆盖的最长上升子序列
8     给一个序列，要求找到两个互不覆盖的最长上升子序列，使其长度之和最大。(n ≤ 1000)
9     标签：二维dp，树状数组优化
10    做法：dp[l][r]表示第一个串以l结尾，第二个串以r结尾的最大长度和，转移方程：
        
$$dp[i][u] = \max \{dp[i][j]\} + 1 (1 \leq j < u)$$

        ,
        
$$dp[u][j] = \max \{dp[i][j]\} + 1 (1 \leq i < u)$$

        .然后用多个树状数组维护区间最大值。
11 */
12 /*二维偏序的LIS
13     给一个二维坐标(x,y)的序列，求满足对任意i < j, 都有x_i < x_j, y_i < y_j的最长子序列
14     做法：二分 + 树状数组 + dp (cdq)
15     将序列[l, r]二分，先处理左边的区间[l, mid],
16     再用左边的区间更新右边的区间，即将区间[l, r]按左端点排序，然后依次扫描，
        遇到在左半区间的加入树状数组，遇到在右半区间的查询比当前y值更小的数对数并更新，
        然后再递归处理右边的区间[mid+1, r]
17 */
```

3 图论

3.1 拓扑排序

```
1 ///拓扑排序 (Topologicla Sorting)
2 //有向无环图  $O(|E| + |V|)$ 
3 /*算法:
4     1. 选择没有前驱(入度为0)的顶点 $v$ , 并输出
5     2. 删除从 $v$ 出发的所有有向边
6     3. 重复前两步, 直至没有入度为0的顶点
7     4. 如果最后还剩下一些顶点, 这该图不是DAG
8 */
9 const int MAXV = 1000;
10 int V;//顶点数
11 int deg[MAXV];//入度
12 int ans[MAXV];//拓扑排序结果
13 bool TopoSort()
14 {
15     int top = 0, tail = 0;
16     for(int i = 0; i <= V; ++i)
17     {
18         if(deg[i] == 0)
19             ans[top++] = i;
20     }
21     while(tail < top)
22     {
23         for(int i = head[tail++]; ~i; i = e[i].next)
24         {
25             if(--deg[e[i].to] == 0)
26                 ans[top++] = e[i].to;
27         }
28     }
29     return top == V;
30 }
```

3.2 欧拉回路 Euler, 哈密顿回路 Hamilton

```
1 ///欧拉回路 Euler
2 /*
3 定义: 寻找一条回路, 经过且仅经过一次所有边, 最后回到出发点
4 存在的充要条件: 1. 该图是连通的 2. 无向图, 度数为奇数的顶点的个数为0; 有向图, 每个顶点入度等于出度
5 构造算法:  $O(|E|)$ 
6     1. 深度搜索, 得出一条回路.
7     2. 如果该回路不是欧拉回路, 则沿该回路回溯, 找到一个没有搜索过的顶点, 重复步骤1,
8     然后将新回路加入答案中.
9 定义欧拉路径: 寻找一条简单路径, 经且仅经过所有边一次
10 存在条件: 连通, (无向图)度数为奇数的顶点个数为0或2, (有向图)只有两个顶点入度不等于出度,
11 且一个入度比出度大1, 另一个小1.
12 */
13 #define MAXE
14 int ans[MAXE], ansi;//ansi等于V表示存在欧拉回路
15 bool vis[2 * MAXE];
16 void dfs(int u)
17 {
18     for(int i = head[u]; ~i; i = e[i].next)//链式前向星存储
19     {
20         if(vis[i]) continue;
21         vis[i ^ 1] = vis[i] = true;//标记当前边及反向边
22         dfs(e[i].to);
23         ans[ansi++] = i;//沿回溯道路将经过的每个点加入答案
24     }
25 }
```

```

22     }
23 }
24 ///有向图欧拉路径
25 int in[MAX_V], out[MAX_V]; //入度和出度
26 int num; //已经有多少边在欧拉路径中
27 int pre_edge[MAX_E]; //存储每一条边的上一条边
28 //通过入度和出度判定是否可能存在欧拉路径, 存在返回起点, 否则返回-1
29 int is_exist_euler()
30 {
31     int sp = -1, tp = -1;
32     bool flag = true;
33     for(int i = 0; flag && i < V; ++i)
34     {
35         if(in[i] == out[i]) continue;
36         if(in[i] - out[i] == 1)
37         {
38             if(tp == -1) tp = i;
39             else flag = false;
40         }
41         else if(out[i] - in[i] == 1)
42         {
43             if(sp == -1) sp = i;
44             else flag = false;
45         }
46         else flag = false;
47     }
48     if(!flag) return -1;
49     if(sp < 0 && tp >= 0) return -1;
50     if(sp >= 0 && tp < 0) return -1;
51     if(sp < 0 && tp < 0)
52     {
53         int i = 0;
54         while(sp == -1)
55         {
56             if(head[i] != -1)
57             {
58                 sp = i;
59             }
60             i++;
61         }
62     }
63     if(!flag)
64         sp = -1;
65     return sp;
66 }
67 int find_path(int u, int pre)
68 {
69     //寻找以u为起点的一条边
70     int i;
71     while(head[u] != -1)
72     {
73         i = head[u];
74         pre_edge[i] = pre; //加入改变进欧拉路径
75         pre = i;
76         head[u] = e[i].next; //将已经访问过的边去掉
77         u = e[i].v; //要访问的下一个结点
78         num++;
79     }
80     return pre; //访问当前路径最后一条边
81 }
82 //存在欧拉路径返回最后一条边的编号, 否则返回-1
83 int oula_path()
84 {
85     int st = is_exist_euler(); //判断是否可能存在欧拉路径

```

```

86     if(st == -1) return -1;
87     num = 0;
88     int ed = find_path(st, -1); // 找到基础路径
89     int ei = ed;
90     while(ei >= 0)
91     {
92         if(head[e[ei].u] != -1)
93         {
94             pre_edge[ei] = find_path(e[ei].u, pre_edge[ei]); // 寻找环
95         }
96         ei = pre_edge[ei];
97     }
98     if(num != E) // 可能存在其他连通块
99         return -1;
100     return ed;
101 }
102 /// 哈密顿回路(Hamilton)
103 /*
104 定义：寻找一条回路，经且仅经过每个顶点一次，最后回到出发点。
105 存在的充分条件：任意两个不同顶点的度数和大于等于顶点数V。
106 构造算法： $O(|V|^2)$ 
107     1. 任找两个相邻顶点S, T
108     2. 分别向两头扩展至无法扩展为止，称头尾结点为S, T
109     3. 若S, T不相邻，在路径 $S \rightarrow T$ 中找到结点 $V_i$ ，其中 $V_i$ 与T相邻， $V_{i+1}$ 与S相邻，
110        令 $S \rightarrow T$ 变为 $S \rightarrow V_i \rightarrow T \rightarrow V_{i+1}$ 。
111     4. ( $S \rightarrow T$ 已为回路)若 $S \rightarrow T$ 中顶点个数不为V，找 $S \rightarrow T$ 中找到顶点 $V_i$ ，其中 $V_i$ 与一个未访问过得顶点相邻，
112        从 $V_i$ 处断开(S为 $V_i$ ，T为 $V_{i+1}$ ，重复2。
113 */
114 #define MAXV
115 int V;
116 int ans[MAXV];
117 bool G[MAXV][MAXV]; // 邻接矩阵存储
118 bool vis[MAXV];
119 void Hamilton()
120 {
121     int s = 1, t;
122     int ansi = 2;
123     int i, j;
124     memset(vis, false, sizeof(vis));
125     for(i = 1; i <= V; i++) if(G[s][i]) break;
126     t = i;
127     ans[0] = s, ans[1] = t;
128     while(1)
129     {
130         while(1)
131         {
132             for(i = 1; i <= V; i++)
133             if(G[t][i] && !vis[i])
134             {
135                 ans[ansi++] = i;
136                 vis[i] = true;
137                 t = i;
138                 break;
139             }
140             if(i > V) break;
141         }
142         reverse(ans, ans + ansi);
143         swap(s, t);
144         while(1)
145         {
146             for(i = 1; i <= V; i++)
147             if(G[t][i] && !vis[i])
148             {

```



```

148         ans[ansi++] = i;
149         vis[i] = true;
150         t = i;
151         break;
152     }
153     if(i > V)break;
154 }
155 if(!G[s][t])
156 {
157     for(i = 1; i < ansi - 2; i++)if(G[s][ans[i + 1]] && G[ans[i]][t]) break;
158     t = ans[++i];
159     reverse(ans + i, ans + ansi);
160 }
161 if(ansi == n)return ;
162 for(j = 1; j <= V; j++)
163     if(!vis[j])
164         for(i = 1; i < ansi - 2; i++)if(G[ans[i]][j])break;
165 s = ans[i - 1];
166 t = j;
167 reverse(ans, ans + i);
168 reverse(ans + i, ans + ansi);
169 ans[ansi++] = j;
170 vis[j] = true;
171 }
172 }

```

3.3 无向图的桥，割点，双连通分量

```

1  /*
2  无向连通图的点连通度：使一个无向连通图变成多个连通块要删去的最少顶点数(及相连的边)。
3  无向连通图的边连通度：使一个无向连通图变成多个连通块要删去的最少边数。
4  当无向图的点连通度或边连通度大于1时，称该图双连通图。
5  割点(关节点,割顶)：(点连通度为1) 删去割点及其相连的边后，该图由连通变为不连通；
6  割边(桥)：(边连通度为1)删去割边后，连通图变得不连通。
7  点双连通分量：点连通度大于1的分量。
8  边双连通分量：边连通度大于1的分量。
9  桥的两个端点是割点，有边相连的两割点之间的边不一定是桥(重边)。
10 */
11 ///Tarjan算法求割点或桥
12 /*
13 定义  $dfn(u)$  为无向图中在深度优先搜索中的遍历次序， $low(u)$ 
    为顶点 $u$ 或 $u$ 的子树中的顶点经过非父子边追溯到的最早的结点 ( $dfn$ 序号最小)。
14
15 割点：该点是根结点且有不只一棵子树，或该结点的任意一个孩子结点，没有到该结点祖先的反向边(存在
     $dfn(u) \leq low(v)$ )
16
17 桥：当且仅当该边 $(u,v)$ 是树枝边，且 $dfn[u] < low[v]$ 
18
19 缩点：缩点后变成一棵 $k$ 个点 $k-1$ 条割边连接成的树，而割点可以存在于多个块中。
20
21 将有桥的连通图加边变为边双连通图(加边数最少)：将边双连通分量缩点，形成一棵树，
    反复将两个最近公共祖先最远的两个叶子节点之间连一条边，(这样形成一个双连通分量，可缩点)，
    刚好 $(leaf+1)/2$ 条边。
22
23 点双连通分支：建立一个栈，存储当前双连通分支，在搜索图时，每找到一条树枝边或后向边(非横叉边)，
    就把这条边加入栈中。如果遇到某时满足 $dfn[u] \leq low[v]$ ，说明 $u$ 是一个割点，
    同时把边从栈顶一个个取出，直到遇到了边 $(u,v)$ ，取出的这些边与其关联的点，组成一个点双连通分支。
    割点可以属于多个点双连通分支，其余点和每条边只属于且属于一个点双连通分支。(树枝边：
    搜索过程中遍历过的边)。
24
25 边双连通分支：求出所有的桥后，把桥边删除，原图变成了多个连通块 则每个连通块就是一个边双连通分支。
    桥不属于任何一个边双连通分支，其余的边和每个顶点都属于且只属于一个边双连通分支。

```

```

26 */
27
28 // 图
29 const int MAXV = 100010, MAXE = 100010;
30 struct edge
31 {
32     int next, to;
33 } e[MAXE];
34 int head[MAXV], htot;
35 int V, E;
36 void init()
37 {
38     memset(head, -1, sizeof(head));
39     htot = 0;
40 }
41 void add_edge(int u, int v)
42 {
43     e[htot].to = v;
44     e[htot].next = head[u];
45     head[u] = htot++;
46 }
47
48 int dfn[MAXV], low[MAXV];
49 int stk[MAXV], top; // 栈 -DCC
50
51 // 割点 点双连通分量
52 int cp[MAXV]; // 记录割点
53 int id[MAXE], cnt_dcc; // 连通分量编号及总数 -DCC
54 int index;
55 void tarjan(int u, int root, int pre) // 重边对应边的id, 否则对应父亲结点fa
56 {
57     dfn[u] = low[u] = index++;
58     int num = 0;
59     for(int i = head[u]; i != -1; i = e[i].next)
60     {
61         int v = e[i].to;
62         if(!dfn[v])
63         {
64             num++;
65             stk[top++] = i; // -DCC
66             tarjan(v, root, i);
67             if(low[u] > low[v])
68                 low[u] = low[v];
69             if((u == root && num >= 2) || (u != root && dfn[u] <= low[v]))
70             {
71                 cp[u] = 1; // 是割点
72                 // -DCC
73                 cnt_dcc++;
74                 do
75                 {
76                     id[stk[--top]] = cnt_dcc;
77                 }
78                 while(stk[top] != i);
79             }
80         }
81         else if((i ^ 1) != pre) // -DCC
82         {
83             if(low[u] > dfn[v])
84                 low[u] = dfn[v];
85             // -DCC
86             if(dfn[u] > dfn[v])
87                 stk[top++] = i;
88         }
89     }

```

```

90 }
91 void DCC(int V)
92 {
93     index = top = cnt_dcc = 0; // -DCC
94     memset(dfn, 0, sizeof(dfn));
95     memset(cp, 0, sizeof(cp));
96     for(int i = 1; i <= V; i++)
97         tarjan(i, i, -1);
98 }
99
100 //桥 边双连通分量
101 //注释部分为求边的双连通分量
102 int ce[MAXE], num; //记录桥
103 //int id[MAXE << 1], cnt_dcc; //连通分量编号及总数
104 int dfn[MAXV], low[MAXV];
105 //int stk[MAXV], top;
106 void tarjan(int u, int order, int pre)
107 {
108     dfn[u] = low[u] = order++;
109     // stk[top++] = u;
110     for(int i = head[u]; ~i; i = e[i].next)
111     {
112         if(!e[i].inpath || (i ^ 1) == pre) continue;
113         int v = e[i].to;
114         if(!dfn[v])
115         {
116             tarjan(v, order, i);
117             if(low[v] < low[u]) low[u] = low[v];
118             if(dfn[u] < low[v])
119             {
120                 ce[num++] = (i >> 1) + 1;
121                 // ++cnt_dcc;
122                 // do
123                 // {
124                 //     id[stk[--top]] = cnt_dcc;
125                 // }
126                 // while(stk[top] != v);
127             }
128         }
129         else if(dfn[v] < low[u])
130             low[u] = dfn[v];
131     }
132 }
133 void DCC()
134 {
135     memset(dfn, 0, sizeof(dfn));
136     num = 0;
137     // top = cnt_dcc = 0;
138     for(int u = 1; u <= V; ++u) if(!dfn[u]) tarjan(u, 1, -1);
139 }

```

3.4 最短路 shortest path

```

1 //最短路 Shortest Path
2 //dijkstra算法  $O(|E| \log |V|)$ 
3 struct edge {int next, to, cost;} e[MAXE << 1];
4 int head[MAXV], tot;
5 int V, E; //结点数和边数(结点编号开始于1)
6 void init();
7 void add_edge(int u, int v, int w);
8 int dist[MAXV];
9 void dijkstra(int s)

```

```

10 {
11     priority_queue<P, vector<P>, greater<P> > que;
12     for(int i = 1; i <= V; ++i) dist[i] = INF;
13     dist[s] = 0;
14     que.push(P(0, s));
15     while(!que.empty())
16     {
17         P p = que.top(); que.pop();
18         int u = p.SE;
19         if(dist[u] < p.FI) continue;
20         for(int i = head[u]; ~i; i = e[i].next)
21         {
22             if(dist[e[i].to] <= dist[u] + e[i].cost) continue;
23             dist[e[i].to] = dist[u] + e[i].cost;
24             que.push(P(dist[e[i].to], e[i].to));
25         }
26     }
27 }
28
29 //dijkstra算法  $O(|V|^2)$ 
30 int cost[MAXV][MAXV];
31 int d[MAXV];
32 bool vis[MAXV];
33 void dijkstra(int s)
34 {
35     fill(d, d + V, INF);
36     memset(vis, 0, sizeof(vis));
37     d[s] = 0;
38     while(true)
39     {
40         int v = -1;
41         for(int u = 0; u < V; u++)
42             if(!vis[u] && (v == -1 || d[u] < d[v]))
43                 v = u;
44         if(v == -1) break;
45         for(int u = 0; u < V; u++)
46             d[u] = min(d[u], d[v] + cost[v][u]);
47     }
48 }
49 //Bellman-Ford算法  $O(|E| \cdot |V|)$ 
50 // $d[v] = \min \{d[u] + w[e]\} (e = \langle u, v \rangle \in E)$ 
51
52 const int MAXV = 1000, MAXE = 1000, INF = 1000000007;
53 struct edge {int u, v, cost;} e[MAXE];
54 int V, E;
55 //graph G
56 int d[MAXV];
57 void Bellman_Ford(int s)
58 {
59     for(int i = 0; i < V; i++)
60         d[i] = INF;
61     d[s] = 0;
62     while(true)
63     {
64         bool update = false;
65         for(int i = 0; i < E; i++)
66         {
67             if(d[e[i].u] != INF && d[e[i].v] > d[e[i].u] + e[i].cost)
68             {
69                 d[e[i].v] = d[e[i].u] + e[i].cost;
70                 update = true;
71             }
72         }
73     }

```

```

74 }
75 //判负圈
76 bool find_negative_loop()
77 {
78     memset(d, 0, sizeof(d));
79     for(int i = 0; i < V; i++)
80     {
81         for(int j = 0; j < E; j++)
82         {
83             if(d[e[j].v] > d[e[i].u] + e[j].cost)
84             {
85                 d[e[j].v] = d[e[j].u] + e[j].cost;
86                 if(i == V - 1)
87                     return true;
88                 //循环了V次后还不能收敛，即存在负圈
89             }
90         }
91     }
92     return false;
93 }
94
95 //spfa算法  $O(|E| \log |V|)$ 
96 //适用于负权图和稀疏图，稳定性不如dijkstra
97 //存在负环返回false
98 int dist[MAXV];
99 int outque[MAXV]; //出队次数，如果大于V，证明有负圈
100 bool vis[MAXV];
101 bool spfa(int s)
102 {
103     for(int i = 0; i < V; i++)
104     {
105         vis[i] = false;
106         dist[i] = INF;
107         outque[i] = 0;
108     }
109     dist[s] = 0;
110     queue<int> que;
111     que.push(s);
112     vis[s] = true;
113     while(!que.empty())
114     {
115         int u = que.front();
116         que.pop();
117         vis[u] = false;
118         if(++outque[u] > V) return false;
119         for(int i = head[u]; i != -1; i = e[i].next)
120         {
121             int v = e[i].to;
122             if(dist[v] <= dist[u] + e[i].cost) continue;
123             dist[v] = dist[u] + e[i].cost;
124             if(vis[v]) continue;
125             vis[v] = true;
126             que.push(v);
127         }
128     }
129     return true;
130 }
131
132 ///任意两点间最短路
133 //Floyd-Warshall算法  $O(|V|^3)$ 
134 int dist[MAXV][MAXV];
135 void floyd_warshall(int V)
136 {
137     int i, j, k;

```

```

138     for(k = 0; k < V; k++)
139         for(i = 0; i < V; i++)
140             for(j = 0; j < V; j++)
141                 dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
142 }
143
144 ///两点间最短路 — 一条可行路径还原
145 /*用prev[u]记录从s到u的最短路上u的前驱结点*/
146 vector<int> get_path(int t)
147 {
148     vector<int> path;
149     for(; t != -1; t = prev[t])
150         path.push_back(t);
151     reverse(path.begin(), path.end());
152     return path;
153 }
154
155 ///两点间最短路 — 所有可行路径还原 O(|E|)
156 /*
157 从终点t反向dfs, 将所有满足dist[u] = e.cost + dist[v]的边e(u,v)加入路径中即可
158 在无向图中直接运行即可, 而在有向图中需要在其逆图中运行.
159 */
160 int InPath[MAXE << 1];
161 void GetPath()
162 {
163     memset(vis, 0, sizeof(vis));
164     //memset(InPath, 0, sizeof(InPath));
165     queue<int> que;
166     que.push(n);
167     vis[n] = true;
168     while(!que.empty())
169     {
170         int u = que.front(); que.pop();
171         for(int i = head[u]; ~i; i = e[i].next)
172         {
173             if(dist[u] != dist[e[i].to] + e[i].cost) continue;
174             InPath[i] = InPath[i ^ 1] = true;
175             if(vis[e[i].to]) continue;
176             vis[e[i].to] = true;
177             que.push(e[i].to);
178         }
179     }
180 }
181
182 ///求最短路网络上的桥
183 //方法1: 将最短路网络新建一个图, 跑Tarjan算法.
184 //方法2: 最短路还原时, 当且仅当队列为空, 且当前结点只有一条边指向s时, 该边为桥
185 //求割点类似
186 int vis[MAXV];
187 int ce[MAXE], num;
188 void get_bridge(int s, int t)//在逆图中运行
189 {
190     priority_queue<P> que;
191     que.push(P(dist[t], t));//按到t的距离远近出队, 保证割点一定后出队
192     memset(vis, 0, sizeof(vis));
193     vis[t] = 1;
194     num = 0;
195     while(!que.empty())
196     {
197         int u = que.top().SE;
198         que.pop();
199         int cnt = 0;
200         if(que.empty())
201         {
202             for(int i = rhead[u]; i != -1; i = re[i].next)

```

```

202     {
203         int v = re[i].to, w = re[i].cost;
204         if(dist[v] + w == dist[u])
205         {
206             cnt++;
207             if(cnt >= 2) break;
208         }
209     }
210 }
211 bool f = que.empty() && cnt == 1; //当且仅当，队列为空且只有一条路时，找到桥
212 for(int i = rhead[u]; i != -1; i = re[i].next)
213 {
214     int v = re[i].to, w = re[i].cost;
215     if(dist[v] + w == dist[u])
216     {
217         if(f) ce[num++] = i;
218         if(!vis[v])
219         {
220             vis[v] = 1;
221             que.push(P(dist[v], v));
222         }
223     }
224 }
225 }
226 }

```

3.5 最大流 maximum flow

```

1  ///最大流 maximum flow
2  //最大流最小割定理：最大流 = 最小割
3  ///FF算法 Ford-Fulkerson算法  $O(F|E|)$   $F$ 为最大流量
4  //1. 初始化：原边容量不变，回退边容量为0，max_flow = 0
5  //2. 在残留网络中找到一条从源S到汇T的增广路，找不到得到最大流max_flow
6  //3. 增广路中找到瓶颈边，max_flow加上其容量
7  //4. 增广路中每条边减去瓶颈边容量，对应回退边加上其容量
8  struct edge
9  {
10     int to, cap, rev;
11 };
12
13 vector <edge> G[MAXV];
14 bool used[MAXV];
15
16 void add_edge(int from, int to, int cap)
17 {
18     G[from].push_back((edge) {to, cap, G[to].size()});
19     G[to].push_back((edge) {from, 0, G[from].size() - 1});
20 }
21
22 //dfs寻找增广路
23 int dfs(int v, int t, int f)
24 {
25     if(v == t)
26         return f;
27     used[v] = true;
28     for(int i = 0; i < G[v].size(); i++)
29     {
30         edge &e = G[v][i];
31         if(!used[e.to] && e.cap > 0)
32         {
33             int d = dfs(e.to, t, min(f, e.cap));
34             if(d > 0)

```

```

35         {
36             e.cap -= d;
37             G[e.to][e.rev].cap += d;
38             return d;
39         }
40     }
41 }
42 return 0;
43 }
44
45 // 求解从s到t的最大流
46 int max_flow(int s, int t)
47 {
48     int flow = 0;
49     for(;;)
50     {
51         memset(used, 0, sizeof(used));
52         int f = dfs(s, t, INF);
53         if(f == 0)
54             return flow;
55         flow += f;
56     }
57 }
58 ///Dinic算法  $O(|E| \cdot |V|^2)$ 
59 // 似乎比链式前向星快
60 struct edge {int to, cap, rev;};
61 vector<edge> G[MAXV];
62 int level[MAXV];
63 int iter[MAXV];
64 void init()
65 {
66     for(int i = 0; i < MAXV; i++)
67         G[i].clear();
68 }
69 void add_edge(int from, int to, int cap)
70 {
71     G[from].push_back((edge) {to, cap, G[to].size()});
72     G[to].push_back((edge) {from, 0, G[from].size() - 1});
73 }
74 bool bfs(int s, int t)
75 {
76     memset(level, -1, sizeof(level));
77     queue<int> que;
78     level[s] = 0;
79     que.push(s);
80     while(!que.empty())
81     {
82         int v = que.front();
83         que.pop();
84         for(int i = 0; i < (int)G[v].size(); i++)
85         {
86             edge &e = G[v][i];
87             if(e.cap > 0 && level[e.to] < 0)
88             {
89                 level[e.to] = level[v] + 1;
90                 que.push(e.to);
91             }
92         }
93     }
94     return level[t] != -1;
95 }
96
97 int dfs(int v, int t, int f)
98 {

```



```

99     if(v == t) return f;
100     for(int &i = iter[v]; i < (int)G[v].size(); i++)
101     {
102         edge &e = G[v][i];
103         if(e.cap > 0 && level[v] < level[e.to])
104         {
105             int d = dfs(e.to, t, min(f, e.cap));
106             if(d > 0)
107             {
108                 e.cap -= d;
109                 G[e.to][e.rev].cap += d;
110                 return d;
111             }
112         }
113     }
114     return 0;
115 }
116
117 int max_flow(int s, int t)
118 {
119     int flow = 0, cur_flow;
120     while(bfs(s, t))
121     {
122         memset(iter, 0, sizeof(iter));
123         while((cur_flow = dfs(s, t, INF)) > 0) flow += cur_flow;
124     }
125     return flow;
126 }
127 ///SAP算法  $O(|E| \cdot |V|^2)$ 
128 #define MAXV 1000
129 #define MAXE 10000
130 struct edge
131 {
132     int cap, next, to;
133 } e[MAXE * 2];
134 int head[MAXV], tot_edge;
135 void init()
136 {
137     memset(head, -1, sizeof(head));
138     tot_edge = 0;
139 }
140 void add_edge(int u, int v, int cap)
141 {
142     e[tot_edge] = (edge) {cap, head[u], v};
143     head[u] = tot_edge++;
144 }
145 int V;
146 int numh[MAXV]; //用于GAP优化的统计高度数量数组
147 int h[MAXV]; //距离标号数组
148 int pree[MAXV], prev[MAXV]; //前驱边与结点
149 int SAP_max_flow(int s, int t)
150 {
151     int i, flow = 0, u, cur_flow, neck = 0, tmp;
152     memset(h, 0, sizeof(h));
153     memset(numh, 0, sizeof(numh));
154     memset(prev, -1, sizeof(prev));
155     for(i = 1; i <= V; i++) //从1开始的图，初识化为当前弧的第一条临接边
156         pree[i] = head[i];
157     numh[0] = V;
158     u = s;
159     while(h[s] < V)
160     {
161         if(u == t)
162         {

```

```

163     cur_flow = INT_MAX;
164     for(i = s; i != t; i = e[pree[i]].to)
165     {
166         if(cur_flow > e[pree[i]].cap)
167         {
168             neck = i;
169             cur_flow = e[pree[i]].cap;
170         }
171     } // 增广成功, 寻找"瓶颈"边
172     for(i = s; i != t; i = e[pree[i]].to)
173     {
174         tmp = pree[i];
175         e[tmp].cap -= cur_flow;
176         e[tmp ^ 1].cap += cur_flow;
177     } // 修改路径上的边容量
178     flow += cur_flow;
179     u = neck; // 下次增广从瓶颈边开始
180 }
181 for(i = pree[u]; i != -1; i = e[i].next)
182     if(e[i].cap && h[u] == h[e[i].to] + 1)
183         break; // 寻找可行弧
184 if(i != -1)
185 {
186     pree[u] = i;
187     prev[e[i].to] = u;
188     u = e[i].to;
189 }
190 else
191 {
192     if(0 == --numh[h[u]]) break; // GAP 优化
193     pree[u] = head[u];
194     for(tmp = V, i = head[u]; i != -1; i = e[i].next)
195         if(e[i].cap)
196             tmp = min(tmp, h[e[i].to]);
197     h[u] = tmp + 1;
198     ++numh[h[u]];
199     if(u != s) u = prev[u]; // 从标号并且从当前结点的前驱重新增广
200 }
201 }
202 return flow;
203 }
204
205 /// EK 算法  $O(|V| \cdot |E|^2)$ 
206 /// bfs 寻找增广路
207 const int MAXV = 210;
208 int g[MAXV][MAXV], pre[MAXV];
209 int n;
210 bool vis[MAXV];
211 bool bfs(int s, int t)
212 {
213     queue<int> que;
214     memset(pre, -1, sizeof(pre));
215     memset(vis, 0, sizeof(vis));
216     que.push(s);
217     vis[s] = true;
218     while(!que.empty())
219     {
220         int u = que.front();
221         if(u == t) return true;
222         que.pop();
223         for(int i = 1; i <= n; i++)
224             if(g[u][i] && !vis[i])
225             {
226                 vis[i] = true;

```

```

227         pre[i] = u;
228         que.push(i);
229     }
230 }
231 return false;
232 }
233 int EK_max_flow(int s, int t)
234 {
235     int u, max_flow = 0, minv;
236     while(bfs(s, t))
237     {
238         minv = INF;
239         u = t;
240         while(pre[u] != -1)
241         {
242             minv = min(minv, g[pre[u]][u]);
243             u = pre[u];
244         }
245         ans += minv;
246         u = t;
247         while(pre[u] != -1)
248         {
249             g[pre[u]][u] -= minv;
250             g[u][pre[u]] += minv;
251             u = pre[u];
252         }
253     }
254     return max_flow;
255 }

```

3.6 最小割 minimum cut

```

1  /// 最小割 Minimum Cut
2  /*
3  定义:
4  割: 网络  $(V, E)$  的割 (cut)  $[S, T]$  将点集  $V$  划分为  $S$  和  $T (T=V-S)$  两个部分, 使得源  $s \in S$ , 汇  $t \in T$ .
      符号  $[S, T]$  代表一个边集合  $\{ \langle u, v \rangle \mid \langle u, v \rangle \in E, u \in S, v \in T \}$ . 穿过割  $[S, T]$  的净流 (net
      flow) 定义为  $f(S, T)$ , 容量 (capacity) 定义为  $c(S, T)$ .
5  最小割: 该网络中容量最小的割
6  (割与流的关系) 在一个流网络  $G(V, E)$  中, 设其任意一个流为  $f$ , 且  $[S, T]$  为  $G$  一个割. 则通过割的净流为  $f(S, T) = |f|$ .
7  (对偶问题的性质) 在一个流网络  $G(V, E)$  中, 设其任意一个流为  $f$ , 任意一个割为  $[S, T]$ , 必有  $|f| \leq c[S, T]$ .
8  (最大流最小割定理) 如果  $f$  是具有源  $s$  和汇  $t$  的流网络  $G(V, E)$  中的一个流, 则下列条件是等价的:
9      (1)  $f$  是  $G$  的一个最大流
10     (2) 残留网络  $G_f$  不包含增广路径
11     (3) 对  $G$  的某个割  $[S, T]$ , 有  $|f| = c[S, T]$ 
12     即最大流的流值等于最小割的容量
13  */
14  /*
15  性质:
16     性质1(不连通): 在给定的流网络中, 去掉割的边集, 则不存在任何一条从源到汇的路径.
17     性质2(两类点): 在给定的流网络中, 任意一个割将点集划分成两部分. 割为两部分点之间的“桥梁”.
18  技巧:
19     技巧1 用正无限容量排除不参与决策的边.
20     技巧2 使用割的定义式来分析最优性.
21     技巧3 利用与源或汇关联的边容量处理点权.
22  */
23  /*
24  最小割的求法:
25     1. 先求的最大流
26     2. 在得到最大流  $f$  后的残留网络  $G_f$  中, 从源  $s$  开始深度优先遍历 (DFS), 所有被遍历的点, 即构成点集  $S$ 
27     注意: 虽然最小割中的边都是满流边, 但满流边不一定是最小割的边.

```

```

28 */
29 int max_flow(int s, int t) {}
30 int getST(int s, int t, int vis[])
31 {
32     int mincap = max_flow(s, t);
33     memset(nd, 0, sizeof(nd));
34     queue<int> que;
35     que.push(s);
36     vis[s] = 1;
37     while(!que.empty())
38     {
39         int u = que.front(); que.pop();
40         for(int i = 0; i < (int)g[u].size(); i++)//travel v
41             if(g[u][i].cap > 0 && !vis[g[u][i].to])
42             {
43                 vis[g[u][i].to] = 1;
44                 que.push(g[u][i].to);
45             }
46     }
47     return mincap;
48 }
49 ///无向图全局最小割 Stoer-Wagner算法
50 /*定理：对于图中任意两点s和t来说，无向图G的最小割要么为s到t的割，要么是生成图G/{s,t}的割(把s和t合并)
51 算法的主要部分就是求出当前图中某两点的最小割，并将这两点合并
52 快速求当前图某两点的最小割：
53     1. 维护一个集合A，初始里面只有v1(可以任意)这个点
54     2. 选一个最大的w(A, y)的点y放入集合A(集合到点的权值为集合内所有点到该点的权值和)
55     3. 反复2，直至A集合G集相等
56     4. 设后两个添加的点为s和t，那么w(G-{t}, t)的值，就是s到t的cut值
57 */
58 //O(|V|^3)
59 const int MAXV = 510;
60 int n;
61 int g[MAXV][MAXV]; //g[u][v]表示u,v两点间的最大流量
62 int dist[MAXV]; //集合A到其他点的距离
63 int vis[MAXV];
64 int min_cut_phase(int &s, int &t, int mark) //求某两点间的最小割
65 {
66     vis[t] = mark;
67     while(true)
68     {
69         int u = -1;
70         for(int i = 1; i <= n; i++)
71             if(vis[i] < mark && (u == -1 || dist[i] > dist[u])) u = i;
72         if(u == -1) return dist[t];
73         s = t, t = u;
74         vis[t] = mark;
75         for(int i = 1; i <= n; i++) if(vis[i] < mark) dist[i] += g[t][i];
76     }
77 }
78
79 int min_cut()
80 {
81     int i, j, res = INF, x, y = 1;
82     for(i = 1; i <= n; i++)
83         dist[i] = g[1][i], vis[i] = 0;
84     for(i = 1; i < n; i++)
85     {
86         res = min(res, min_cut_phase(x, y, i));
87         if(res == 0) return res;
88         //merge x, y
89         for(j = 1; j <= n; j++) if(vis[j] < n) dist[j] = g[j][y] = g[y][j] = g[y][j] + g[x][j];
90         vis[x] = n;
91     }

```

```

92 |     return res;
93 | }

```

3.7 分数规划 Fractional Programming

```

1 | ///分数规划 Fractional Programming
2 | //source: <<最小割模型在信息学竞赛中的应用>>
3 | /*
4 | 一般形式:  $\min \{\lambda = f(\vec{x}) = \frac{a(\vec{x})}{b(\vec{x})}\} (\vec{x} \in S, \forall \vec{x} \in S, b(\vec{x}) > 0)$ 
5 | 其中解向量  $\vec{x}$  在解空间  $S$  内,  $a(\vec{x})$  与  $b(\vec{x})$  都是连续的实值函数.
6 | 解决分数规划问题的一般方法是分析其对偶问题, 还可进行参数搜索(parametric search),
7 | 即对答案进行猜测, 在验证该猜测值的最优性, 将优化问题转化为判定性问题或者其他优化问题.
8 | 构造新函数:  $g(\lambda) = \min \{a(\vec{x}) - \lambda \cdot b(\vec{x})\} (\vec{x} \in S)$ 
9 | 函数性质: (单调性)  $g(\lambda)$  是一个严格递减函数, 即对于  $\lambda_1 < \lambda_2$ , 一定有  $g(\lambda_1) > g(\lambda_2)$ .
10 | (Dinkelbach定理) 设  $\lambda^*$  为原规划的最优解, 则  $g(\lambda) = 0$  当且仅当  $\lambda = \lambda^*$ .
11 | 设  $\lambda^*$  为该优化的最优解, 则:
12 | 
$$\begin{cases} g(\lambda) = 0 \Leftrightarrow \lambda = \lambda^* \\ g(\lambda) < 0 \Leftrightarrow \lambda > \lambda^* \\ g(\lambda) > 0 \Leftrightarrow \lambda < \lambda^* \end{cases}$$

13 | 由该性质, 就可以对最优解  $\lambda^*$  进行二分查找.
14 | 上述是针对最小化目标函数的分数规划, 实际上对于最大化目标函数也一样适用.
15 | */
16 | ///0-1分数规划 0-1 fractional programming
17 | /*是分数规划的解向量  $\vec{x}$  满足  $\forall x_i \in \{0, 1\}$ , 即一下形式:

```

$$\min \{\lambda = f(x) = \frac{\sum_{e \in E} w_e x_e}{\sum_{e \in E} 1 \cdot x_e} = \frac{\vec{w} \cdot \vec{x}}{\vec{e} \cdot \vec{x}}\}$$

其中, \vec{x} 表示一个解向量, $x_e \in \{0, 1\}$, 即对与每条边都选与不选两种决策, 并且选出的边集组成一个 s - t 边割集. 形式化的, 若 $x_e = 1$, 则 $e \in C$, $x_e = 0$, 则 $e \notin C$.

3.8 最大闭权图 maximum weight closure of a graph

```

1 | ///最大权闭合图 Maximum Weight Closure of a Graph
2 | /*定义:
3 | 定义一个有向图  $G(V, E)$  的闭合图(closure)是该有向图的一个点集, 且该点集的所有出边都还指向该点集.
4 | 即闭合图内的任意点的任意后继也一定在闭合图中. 更形式化地说,
5 | 闭合图是这样的一个点集  $V' \subseteq V$ , 满足对于  $\forall u \in V'$  引出的  $\forall \langle u, v \rangle \in E$ , 必有  $v \in V'$  成立.
6 | 还有一种等价定义为: 满足对于  $\forall \langle u, v \rangle \in E$ , 若有  $u \in V'$  成立, 必有  $v \in V'$  成立,
7 | 在布尔代数上这是一个"蕴含(impl)"的运算.
8 | 按照上面的定义, 闭合图是允许超过一个连通块的. 给每个点  $v$  分配一个点权  $w_v$  (任意实数, 可正可负).
9 | 最大权闭合图(maximum weight closure), 是一个点权之和最大的闭合图, 即最大化  $\sum_{v \in V'} w_v$ .
10 | */
11 | /*转化为最小割模型  $G_N(V_N, E_N)$ 
12 | 在原图点集的基础上增加源  $s$  和汇  $t$ ;
13 | 将原图每条有向边  $\langle u, v \rangle \in E$  替换为容量为  $c(u, v) = \infty$  的有向边  $\langle u, v \rangle \in E_N$ ;
14 | 增加连接源  $s$  到原图每个正权点  $v(w_v > 0)$  的有向边  $\langle s, v \rangle \in E_N$ , 容量为  $c(s, v) = w_v$ ,
15 | 增加连接原图每个负权点  $v(w_v < 0)$  到汇  $t$  的有向边  $\langle v, t \rangle \in E_N$ , 容量为  $c(v, t) = -w_v$ . 其中,
16 | 正无限  $\infty$  定义为任意一个大于  $\sum_{v \in V} |w_v|$  的整数. 更形式化地, 有:
17 | 
$$V_N = V \cup \{s, t\}$$

18 | 
$$E_N = E \cup \{\langle s, v \rangle \mid v \in V, w_v > 0\} \cup \{\langle v, t \rangle \mid v \in V, w_v < 0\}$$

19 | 
$$\begin{cases} c(u, v) = \infty & \langle u, v \rangle \in E \\ c(s, v) = w_v & w_v > 0 \\ c(v, t) = -w_v & w_v < 0 \end{cases}$$

20 | 当网络  $N$  的取到最小割时, 其对应的图  $G$  的闭合图将达到最大权.
21 | */

```

3.9 最大密度子图 Maximum Density Subgraph

```
1 //最大密度子图 a Maximum Density Subgraph
2 /*
3 定义：
4     定义无向图 $G=(V,E)$ 的密度(density) $D$  为该图的边数 $|E|$ 与该图的点数 $|V|$ 的比值 $D=\frac{|E|}{|V|}$ .
5     给一个无向图 $G=(V,E)$ ，其中具有最大密度的子图 $G'=(V',E')$ ，称为最大密度子图(maximum density
6     subgraph)，即最大化 $D'=\frac{|E'|}{|V'|}$ .
7 */
8 做法：
9     先转化为分数规划，在转化为最小割即可。
10 */
```

3.10 二分图的最小点权覆盖集与最大点权独立集

```
1 //二分图的最小点权覆盖集与最大点权独立集 Minimum Weight Vertex Covering Set and Maximum Weight
  Vertex Independent Set in a Bipartite Graph
2 /*定义：
3 点覆盖集 (vertex covering set, VCS) 是无向图的 $G$ 的一个点集，
4     使得该图中所有边都至少有一个端点在该集合内。形式化的定义：点覆盖集为  $V' \in V$ ，
5     满足对于 $\forall(u,v) \in E$ ，都有  $u \in V'$  或  $v \in V'$ 。
6 点独立集 (vertex independent set, VIS) 是无向图的 $G$ 的一个点集，
7     使得任意两个在该集合中的点在原图中都不相邻，即导出子图为零图(没有边)的点集。形式化的定义：
8     点独立集是 $V' \in V$ ，满足 $\forall u,v \in V'$ ，都有 $(u,v) \notin E$ 成立。等价的定义：点独立集为 $V' \in V$ ，
9     满足 $\forall(u,v) \in E$ ，都有 $u \in V'$ 与 $v \in V'$ 不同时成立。
10 最小点覆盖集 (minimum vertex covering set, MinVCS) 是在无向图 $G$ 中，点数最少的点覆盖集。
11 最大点独立集 (maximum vertex independent set, MaxVIS) 是在无向图 $G$ 中，点数最多的点独立集。
12 一个带点权无向图 $G=(V,E)$ ，对于 $\forall v \in V$ ，都被分配一个非负点权 $w_v$ 。
13 最小点权覆盖集 (minimum weight vertex covering set, MinWVCS) 是在带点权无向图 $G$ 中，
14     点权之和最小的点覆盖集。
15 最大点权独立集 (maximum weight vertex independent set, MaxWVIS) 是在带点权无向图 $G$ 中，
16     点权之和最大的点独立集。
17 带权二分图 $G=(V,E)$ 中，其中 $V=X \cup Y, X \cap Y = \emptyset$ ，且对于 $\forall v \in V$ ，都被分配了一个非负的权值 $w_v (w_v \geq 0)$ 。
18 */
19 /*二分图的最小点权覆盖集算法 Algorithm for MinWVCS in a Bipartite Graph
20 考虑二分图的网络流解法，它加入了额外的源 $s$ 和汇 $t$ ，将匹配以一条条 $s-u-v-t$ 形式的流路径“串联”起来。
21 同样，如上建图，建立一个源 $s$ ，向 $X$ 部每个点连边；建立一个汇 $t$ ，从 $Y$ 部每个点向汇 $t$ 连边。
22     把二分图看做有向的，则任意一条从 $s$ 到 $t$ 的路径，一定具有 $s-u-v-t$ 的形式。
23     割的性质是不存在一条从 $s$ 到 $t$ 的路径，故路径上的三条边 $(s,u), (u,v), (v,t)$ 中至少有一条在割中。
24     若人为的令边 $(u,v)$ 不在最小割中，则令其容量为正无穷 $c(u,v)=+\infty$ ，则条件简化为  $(s,u), (v,t)$ 
25     至少有一条边在最小割中，正好和点覆盖集的形式相对应( $u \in V'$ 或 $v \in V'$ )。
26 将二分图 $G$ 的最小点权覆盖向网络 $N=(V_N, E_N)$ 的最小割模型的转化：
27     在图 $G$ 的基础上添加源 $s$ 和汇 $t$ ；将每条二分图的边替换为容量为 $c(u,v)=\infty$ 的有向边 $(u,v) \in E_N$ ；
28     增加源 $s$ 到 $X$ 部的点 $u$ 的有向边 $(s,u) \in E_N$ ，容量即为改点的权值 $c(s,u)=w_u$ ；
29     增加 $Y$ 部的点 $v$ 到汇 $t$ 的有向边 $(v,t) \in E_N$ ，同样容量为该点的权 $c(v,t)=w_v$ 。
30 引理：网络 $N$ 的简单割 $[S, T]$ 与图 $G$ 的点覆盖集 $V'=X' \cup Y'$ 存在一一对应关系：
31     点覆盖集中的点在网络 $N$ 中相应的带权边组成一个简单割；反之亦然，即：
32
33     
$$[S, T] = [s, X'] \cup [Y', t]$$

34
35 由于最小点覆盖和最小割的优化方向一致，故带权二分图的最小点覆盖转化为最小割模型。
36 复杂度： $O(\text{MaxFlow}(N))$ 
37 */
38 /*二分图的最大点权独立集算法 Algorithm for MaxWVIS in a Bipartite Graph
39 点独立集定义重写，可以得到其补图的点覆盖集定义。即
```

$$\overline{u \in V' v \in V'} = u \in \overline{V'} \text{ or } v \in \overline{V'}$$

```
23 定理 (覆盖集与独立集互补定理)：若 $\overline{V'}$ 为不含孤立点的任意图的一个点覆盖集当且仅当 $V'$ 是该图的一个点独立集。
24 推论 (最优性)若 $V'$ 为不含孤立点的任意图的一个最小点权覆盖集，则 $\overline{V'}$ 就是该图的一个最大点权独立集。
25 求出最大点权独立点集后还有加上反图中的孤立点才是答案。
```

```
26 | 复杂度:  $O(\text{MaxFlow}(N))$ 
27 | */
```

3.11 最小费用最大流 minimum cost flow

```
1  ///最小费用最大流 miniunm cost flow
2  //不断寻找最短路增广即可
3  //复杂度:  $O(F \cdot \text{MaxFlow}(G))$ 
4  //对于稀疏图的效率较高, 对于稠密图的效率低
5  ///dijkstra实现 基于0开始的图
6  const int MAXV = 11000, MAXE = 41000;
7  struct edge {int next, to, cap, cost;} e[MAXE << 1];
8  int head[MAXV], htot;
9  int V;
10 void init()
11 {
12     memset(head, -1, sizeof(head));
13     htot = 0;
14 }
15 void add_edge(int u, int v, int cap, int cost)
16 {
17     e[htot] = (edge) {head[u], v, cap, cost};
18     head[u] = htot++;
19     e[htot] = (edge) {head[v], u, 0, -cost};
20     head[v] = htot++;
21 }
22 int dist[MAXV];
23 int prev[MAXV], pree[MAXV];
24 int h[MAXV];
25 void dijkstra(int s)
26 {
27     priority_queue<P, vector<P>, greater<P> > que;
28     fill(dist, dist + V, INF);
29     que.push(P(0, s));
30     dist[s] = 0;
31     while(!que.empty())
32     {
33         P p = que.top(); que.pop();
34         int u = p.SE;
35         if(dist[u] < p.FI) continue;
36         for(int i = head[u]; ~i; i = e[i].next)
37             if(e[i].cap > 0 && dist[e[i].to] > dist[u] + e[i].cost + h[u] - h[e[i].to])
38             {
39                 dist[e[i].to] = dist[u] + e[i].cost + h[u] - h[e[i].to];
40                 prev[e[i].to] = u;
41                 pree[e[i].to] = i;
42                 que.push(P(dist[e[i].to], e[i].to));
43             }
44     }
45 }
46 int min_cost_flow(int s, int t, int flow)
47 {
48     int min_cost = 0;
49     memset(h, 0, sizeof(h));
50     while(flow > 0)
51     {
52         dijkstra(s);
53         if(dist[t] == INF) return -1;
54         for(int i = 0; i < V; i++) h[i] += dist[t];
55         int now_flow = flow;
56         for(int u = t; u != s; u = prev[u]) //寻找瓶颈边
57             now_flow = min(now_flow, e[pree[u]].cap);
```

```

58     flow -= now_flow;
59     min_cost += now_flow * dist[t];
60     for(int u = t; u != s; u = prev[u])
61     {
62         e[pree[u]].cap -= now_flow;
63         e[pree[u] ^ 1].cap += now_flow;
64     }
65 }
66 return min_cost;
67 }
68 ///spfa实现 基于0开始的图
69 struct edge {int next, to, cap, cost;} e[MAXE << 1];
70 int head[MAXV], htot;
71 int V;
72 void init()
73 {
74     memset(head, -1, sizeof(head));
75     htot = 0;
76 }
77 void add_edge(int u, int v, int cap, int cost)
78 {
79     e[htot] = (edge) {head[u], v, cap, cost};
80     head[u] = htot++;
81     e[htot] = (edge) {head[v], u, 0, -cost};
82     head[v] = htot++;
83 }
84 int dist[MAXV];
85 int prev[MAXV], pree[MAXV];
86 void spfa(int s)
87 {
88     fill(dist, dist + V, INF);
89     dist[s] = 0;
90     bool update = true;
91     while(update)
92     {
93         update = false;
94         for(int v = 0; v < V; v++)
95         {
96             if(dist[v] == INF) continue;
97             for(int i = head[v]; i != -1; i = e[i].next)
98             {
99                 //edge &e = G[v][i];
100                 if(e[i].cap > 0 && dist[e[i].to] > dist[v] + e[i].cost)
101                 {
102                     dist[e[i].to] = dist[v] + e[i].cost;
103                     prev[e[i].to] = v;
104                     pree[e[i].to] = i;
105                     update = true;
106                 }
107             }
108         }
109     }
110 }
111 int h[MAXV];
112 void dijkstra(int s)
113 {
114     priority_queue<P, vector<P>, greater<P> > que;
115     fill(dist, dist + V, INF);
116     que.push(P(0, s));
117     dist[0] = 0;
118     while(!que.empty())
119     {
120         P p = que.top(); que.pop();
121         int u = p.SE;

```



```

122     if(dist[u] < p.FI) continue;
123     for(int i = head[u]; ~i; i = e[i].next)
124         if(e[i].cap > 0 && dist[e[i].to] > dist[u] + e[i].cost + h[u] - h[e[i].to])
125             {
126                 dist[e[i].to] = dist[u] + e[i].cost + h[u] - h[e[i].to];
127                 prev[e[i].to] = u;
128                 pree[e[i].to] = i;
129                 que.push(P(dist[e[i].to], e[i].to));
130             }
131     }
132 }
133 int min_cost_flow(int s, int t, int flow)
134 {
135     int min_cost = 0;
136     while(flow > 0)
137     {
138         spfa(s);
139         if(dist[t] == INF) return -1;
140         int now_flow = flow;
141         for(int u = t; u != s; u = prev[u])//寻找瓶颈边
142             now_flow = min(now_flow, e[pree[u]].cap);
143         flow -= now_flow;
144         min_cost += now_flow * dist[t];
145         for(int u = t; u != s; u = prev[u])
146             {
147                 e[pree[u]].cap -= now_flow;
148                 e[pree[u] ^ 1].cap += now_flow;
149             }
150     }
151     return min_cost;
152 }
153
154 ///zkw最小费用流, 在稠密图上很快
155 const int MAXV = 11000, MAXE = 41000;
156 struct edge {int next, to, cap, cost;} e[MAXE << 1];
157 int head[MAXV], htot;
158 int V;
159 void init()
160 {
161     memset(head, -1, sizeof(head));
162     htot = 0;
163 }
164 void add_edge(int u, int v, int cap, int cost)
165 {
166     e[htot] = (edge) {head[u], v, cap, cost};
167     head[u] = htot++;
168     e[htot] = (edge) {head[v], u, 0, -cost};
169     head[v] = htot++;
170 }
171 int dist[MAXV];
172 int slk[MAXV];
173 int src, sink;//源和汇
174 bool vis[MAXV];
175 int min_cost;//最小费用
176 int aug(int u, int f)
177 {
178     int left = f;
179     if(u == sink)
180     {
181         min_cost += f * dist[src];
182         return f;
183     }
184     vis[u] = true;
185     for(int i = head[u]; ~i; i = e[i].next)

```

```

186 {
187     int v = e[i].to;
188     if(e[i].cap > 0 && !vis[v])
189     {
190         int t = dist[v] + e[i].cost - dist[u];
191         if(t == 0)
192         {
193             int delta = aug(v, min(e[i].cap, left));
194             if(delta > 0) e[i].cap -= delta, e[i ^ 1].cap += delta, left -= delta;
195             if(left == 0) return f;
196         }
197         else
198             slk[v] = min(t, slk[v]);
199     }
200 }
201 return f - left;
202 }
203 bool modlabel()
204 {
205     int delta = INF;
206     for(int i = 0; i < V; i++)
207         if(!vis[i]) delta = min(delta, slk[i]), slk[i] = INF;
208     if(delta == INF) return false;
209     for(int i = 0; i < V; i++)
210         if(vis[i]) dist[i] += delta;
211     return true;
212 }
213 int zkw_min_cost_flow(int s, int t)
214 {
215     src = s, sink = t;
216     min_cost = 0;
217     int flow = 0;
218     memset(dist, 0, sizeof(dist));
219     memset(slk, 0x3f, sizeof(slk));
220     int tmp = 0;
221     do
222     {
223         do
224         {
225             memset(vis, false, sizeof(vis));
226             flow += tmp;
227         }
228         while((tmp = aug(src, INF)));
229     }
230     while(modlabel());
231     return min_cost;
232 }

```

3.12 有上下界的网络流

```

1  ///有上下界的网络流
2  //1. 建图—消除上下界
3  /* 设原来的源点为src, 汇点为sink. 新建一个超级源S和超级汇T, 对于原网络中的每一条边<u, v>, 上界U,
   下界L, 拆分为三条边
4  1). <u, T> 容量L 2). <S, v> 容量L 3). <u, v> 容量U - L
5  最后添加边<sink, src>, 容量+∞.
6  在新建的网络上, 计算从S到T的最大流, 如果从S出发的每条边都是满流, 说明存在可行流,
   否则不存在可行流.
7  求出可行流后, 要继续求最大流, 将该可行流还原到原网络中, 从src到sink不断增广, 直至找不到增广路.
8  要求最小流: 先不连<sink, src>, 计算S到T的最大流, 然后连<sink, src>容量+∞,
   并不断从S寻找到T的增广路, 这进一步增广的流量就是最小流
9  实现的时候, 要将从S连向同一结点, 同一结点连向T的多条边合并成一条(容量增加).

```

3.13 最近公共祖先 LCA

```

1  ///最近公共祖先LCA Least Common Ancestors
2  //较为暴力的做法:
3  /*
4  预处理: dfs深度搜索, 求出每个结点的深度.
5  单个查询: 查询(u, v)的LCA, 不断寻找深度较大的那个结点的父亲结点, 直至到达同一结点为止.
6  时间复杂度: 预处理:  $O(|V|)$ , 单次查询:  $O(n)$ 
7  */
8  //Tarjan的离线算法  $O(n+q)$ 
9  struct edge {int next, to, lca;};
10 //由要查询的<u,v>构成的图
11 edge qe[MAXE * 2];
12 int qh[MAXV], qtot;
13 //原图
14 edge e[MAXE * 2]
15 int head[MAXV], tot;
16 //并查集
17 int fa[MAXV];
18 inline int find(int x)
19 {
20     if(fa[x] != x) fa[x] = find(fa[x]);
21     return fa[x];
22 }
23 bool vis[MAXV];
24 void LCA(int u)
25 {
26     vis[u] = true;
27     fa[u] = u;
28     for(int i = head[u]; i != -1; i = e[i].next)
29         if(!vis[e[i].to])
30         {
31             LCA(e[i].to);
32             fa[e[i].to] = u;
33         }
34     for(int i = qh[u]; i != -1; i = qe[i].next)
35         if(vis[qe[i].to])
36         {
37             qe[i].lca = find(eq[i].to);
38             eq[i ^ 1].lca = qe[i].lca; //无向图, 入边两次
39         }
40 }
41
42 //RMQ的在线算法  $O(n \log n)$ 
43 /*算法描述:
44     dfs扫描一遍整棵树,
45     记录下经过的每一个结点(每一条边的两个端点)和结点的深度(到根节点的距离), 一共 $2n-1$ 次记录
46     再记录下第一次扫描到结点u时的序号
47     RMQ: 得到dfs中从u到v路径上深度最小的结点, 那就是LCA[u][v].
48 */
49 struct node
50 {
51     int u; //记录经过的结点
52     int depth; //记录当前结点的深度
53 } vs[2 * MAXV];
54 bool operator < (node a, node b) {return a.depth < b.depth;}
55 int id[MAXV]; //记录第一次经过点u时的dfn序号
56 void dfs(int u, int fa, int dep, int &k)
57 {
58     vs[k] = (node) {u, dep};

```

```

58     id[u] = k++;
59     for(int i = head[u]; i != -1; i = e[i].next)
60         if(e[i].to != fa)
61         {
62             dfs(e[i].to, u, dep + 1, k);
63             vs[k++] = (node) {u, dep};
64         }
65     }
66     //RMQ
67     //动态查询id[u] 到 id[v] 之间的depth最小的结点
68     //ST表
69     int Log2[MAXV * 2];
70     node st[MAXV * 2][32];
71     void pre_st(int n, node ar[])
72     {
73         Log2[1] = 0;
74         for(int i = 2; i <= n; i++) Log2[i] = Log2[i >> 1] + 1;
75         for(int i = n - 1; i >= 0; i--)
76         {
77             st[i][0] = ar[i];
78             for(int j = 1; i + (1 << j) <= n; j++)
79                 st[i][j] = min(st[i][j - 1], st[i + (1 << j) - 1][j - 1]);
80         }
81     }
82     int query(int l, int r)
83     {
84         int k = Log2[r - l + 1];
85         return min(st[l][k], st[r - (1 << k) + 1][k]).u;
86     }
87
88     void lca_init()
89     {
90         int k = 0;
91         dfs(1, -1, 0, k);
92         pre_st(k, vs);
93     }
94
95     int LCA(int u, int v)
96     {
97         u = id[u], v = id[v];
98         if(u > v) swap(u, v);
99         return query(u, v);
100 }

```

4 数学专题

4.1 素数 Prime

```
1  ///素数 Prime
2  ///筛素数
3  int prim[NUM], prim_num;
4  //O(n log n)
5  void pre_prime()
6  {
7      prim_num = 0;
8      for(int i = 2; i < NUM; ++i)
9          if(!prim[i])
10             {
11                 prim[prim_num++] = i;
12                 for(int j = i + i; j < NUM; j += i) prim[j] = 1;
13             }
14 }
15
16 //O(n)
17 void pre_prime()
18 {
19     prim_num = 0;
20     for(int i = 2; i < NUM; ++i)
21     {
22         if(!prim[i]) prim[prim_num++] = i;
23         for(int j = 0; j < prim_num && i * prim[j] < NUM; ++j)
24         {
25             prim[i * prim[j]] = 1;
26             if(i % prim[j] == 0) break;
27         }
28     }
29 }
30
31 //区间素数
32 /*要获得区间[L, U]内的素数, L和U很大, 但U - L不大, 那么,
33    先线性筛出1到 $\sqrt{2147483647} \leq 46341$ 之间所有的素数, 然后在通过已经筛好的素数筛出给定区间的素数
34 */
35 ///素数判定
36 //试除法: 略过偶数, 试除2到 $\sqrt{n}$ 间的所有数O( $\sqrt{n}$ )
37 bool isPrime(int n)
38 {
39     if(n % 2 == 0) return n == 2;
40     for(int i = 3; i * i <= n; i += 2)
41         if(n % i == 0)
42             return false;
43     return true;
44 }
45 //简单测试: 根据费马小定理p是素数, 则有 $a^{(p-1)} \equiv 1(\%p)$ , 通过选取[0, p-1]间的任意整数a,
46    如果测试结果不满足上述定理, 则p是合数, 否则, p可能是素数
47 //Miller_Rabin O(log n)
48 int qpow(int x, int k, int mod) {}
49 //以a为基,  $n-1=x*2^t$   $a^{(n-1)}=1(\%n)$  验证n是不是合数
50 //一定是合数返回true, 不一定返回false
51 bool check(LL a, LL n, LL x, LL t)
52 {
53     LL ret = qpow(a, x, n);
54     LL last = ret;
55     for(int i = 1; i <= t; i++)
56     {
57         ret = mult_mod(ret, ret, n);
58         if(ret == 1 && last != 1 && last != n - 1) return true; //合数
59     }
```

```

58     last = ret;
59 }
60 if(ret != 1) return true;
61 return false;
62 }
63
64 // Miller_Rabin()算法素数判定
65 //是素数返回true。(可能是伪素数, 但概率极小)
66 //合数返回false;
67
68 bool Miller_Rabin(LL n)
69 {
70     if(n < 2) return false;
71     if(n == 2) return true;
72     if((n & 1) == 0) return false; //偶数
73     LL x = n - 1;
74     LL t = 0;
75     while((x & 1) == 0) {x >>= 1; t++;}
76     int check_time = 50;
77     for(int i = 0; i < check_time; i++)
78     {
79         LL a = rand() % (n - 1) + 1;
80         if(check(a, n, x, t))
81             return false; //合数
82     }
83     return true;
84 }

```

4.2 因式分解 Factorization 和约数

```

1  ///因式分解Factorization
2  ///唯一分解理论
3  /*
4  所有正整数 $N$ 皆可表示为素数之积, 即 $N = \prod_{i=1}^m p_i^{k_i}$ ,  $p_i$ 是素数.
5  因子的性质:
6
7      1. 因子个数函数 $\tau$ 定义为正整数 $n$ 的所有正因子个数, 记为 $\tau(n)$ , 则 $\tau(n) = \prod_{i=1}^m (k_i + 1)$ 
8
9      2. 因子和函数 $\sigma$ 定义为整数 $n$ 的所有正因子之和, 记为 $\sigma(n)$ , 则 $\sigma(n) = \prod_{i=1}^m \frac{p_i^{k_i+1} - 1}{p_i - 1}$ 
10
11     3. 因子和函数 $\sigma$ 和因子个数函数 $\tau$ 是乘性函数.
12
13     4. 设 $a = \prod_{i=1}^m p_i^{x_i}$ ,  $b = \prod_{i=1}^m p_i^{y_i}$ , 则
14
15         
$$\gcd(a, b) = \prod_{i=1}^m \min(x_i, y_i)$$

16
17         ,
18
19         
$$\text{lcm}(a, b) = \prod_{i=1}^m \max(x_i, y_i)$$

20
21     5. 质因数个数:  $10^5$ 以内的数质因数至多7个,  $10^6$ 以内的数的质因数至多8个,
22      $10^9$ 以内的数的质因数至多10个
23 */
24 ///分解质因数
25 //暴力试除法,  $O(\sqrt{N})$ 
26 int prim[NUM], tot; //素数表
27 vector<P> FAC(int n)
28 {
29     vector<P> fac;
30     fac.clear();
31     for(int i = 0; i < tot && prim[i] * prim[i] <= n; ++i)
32     {
33         if(n % prim[i]) continue;
34         int cnt = 0;

```

```

23     while(n % prim[i] == 0)
24     {
25         ++cnt;
26         n /= prim[i];
27     }
28     fac.push_back(P(prim[i], cnt));
29 }
30 if(n > 1) fac.push_back(P(n, 1));
31 return fac;
32 }
33
34 int facs[NUM];
35 int find_fac(int n)
36 {
37     int cnt = 0;
38     for(int i = 2; i * i <= n; i += 2)
39     {
40         while(!(n % i))
41         {
42             n /= i;
43             facs[cnt++] = i;
44         }
45         if(i == 2) --i;
46     }
47     if(n > 1) facs[cnt++] = n;
48     return cnt;
49 }
50 /// 预处理1~n间所有数的约数  $O(n \log n)$ 
51 vector<int> facs[NUM];
52 int prim[NUM], tot;
53 void pre_fac()
54 {
55     for(int i = 2; i < NUM; ++i)
56     {
57         if(prim[i]) continue;
58         prim[tot++] = i;
59         facs[i].push_back(i);
60         for(int j = i + i; j < NUM; j += i)
61         {
62             prim[j] = 1;
63             facs[j].push_back(i);
64         }
65     }
66 }
67 ///pollard_rho启发式方法  $O(\sqrt[3]{n})$ 
68 //对较大的数整数进行分解
69 /*
70 选取  $[2, n - 1]$  间的随机数, 通过随机函数  $x = (x^2 + c) \% n$ , 如果序列出现循环, 则退出;
71 否则计算  $p = \gcd(x_{i-1}, x_i)$ , 如果  $p = 1$ , 则继续直到  $p > 1$  为止, 若  $p = n$ , 则  $n$  是素数,
72 否则  $p$  是  $n$  的一个约数, 并递归分解  $p$  和  $n/p$ .
73 当  $p$  为素数用 pollard_rho 较为耗时, 所以可以先用 milller_rabin 判断  $p$  是否为素数.
74 */
75 LL factor[100]; //质因数分解结果(刚返回时是无序的)
76 int tol; //质因数的个数. 数组小标从0开始
77 LL mult_mod(LL a, LL b, LL c)
78 {
79     a %= c;
80     b %= c;
81     LL ret = 0;
82     while(b)
83     {
84         if(b & 1) {ret += a; ret %= c;}
85         a <<= 1;

```

```

85     if(a >= c) a %= c;
86     b >>= 1;
87 }
88 return ret;
89 }
90 LL gcd(LL a, LL b)
91 {
92     if(a == 0) return 1;
93     if(a < 0) return gcd(-a, b);
94     while(b)
95     {
96         LL t = a % b;
97         a = b;
98         b = t;
99     }
100    return a;
101 }
102
103 LL Pollard_rho(LL x, LL c)
104 {
105     LL i = 1, k = 2;
106     LL x0 = rand() % x;
107     LL y = x0;
108     while(1)
109     {
110         i++;
111         x0 = (mult_mod(x0, x0, x) + c) % x;
112         LL d = gcd(y - x0, x);
113         if(d != 1 && d != x) return d;
114         if(y == x0) return x;
115         if(i == k) {y = x0; k += k;}
116     }
117 }
118 //对n进行素因子分解
119 void findfac(LL n)
120 {
121     if(Miller_Rabin(n))//素数
122     {
123         factor[tol++] = n;
124         return;
125     }
126     LL p = n;
127     while(p >= n) p = Pollard_rho(p, rand() % (n - 1) + 1);
128     findfac(p);
129     findfac(n / p);
130 }
131
132
133 //枚举约数  $O(\sqrt{n})$ 
134 void DIV(int n)
135 {
136     vector<int> d;
137     int i;
138     for(i = 1; i * i < n; ++i)
139         if(n % i == 0)
140         {
141             d.push_back(i);
142             d.push_back(n / i);
143         }
144     if(i * i == n) d.push_back(i);
145 }

```


4.3 欧拉函数 Euler

```
1  ///欧拉函数 (Euler's totient function)
2  /*
3  对正整数 $n$ , 欧拉函数是小于或等于 $n$ 的数中与 $n$ 互质的数的数目
4  通式:  $\phi(n) = n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_k})$ , 其中 $p_1, p_2, \cdots, p_k$  为 $n$ 的所有质因数,  $n$ 是不为 $0$ 的整数.
5   $\phi(1) = 1$  (唯一和 $1$ 互质的数(小于等于 $1$ )就是 $1$ 本身);
6   $\phi(p) = p - 1$ ,  $p$ 为素数
7  若 $n$ 是质数 $p$ 的 $k$ 次幂,  $\phi(n) = p^k - p^{k-1} = (p - 1)p^{k-1}$ , 因为除了 $p$ 的倍数外, 其他数都跟 $n$ 互质。
8  欧拉函数是积性函数——若 $m, n$ 互质,  $\phi(mn) = \phi(m)\phi(n)$ .
9  当 $n$ 为奇数时,  $\phi(2n) = \phi(n)$ .
10 */
11 ///欧拉定理: 对任何两个互质的正整数 $a, m, m \geq 2$ 有  $a^\phi(m) \equiv 1(\%m)$ 
12 ///费马小定理: 当 $p$ 是质数时, 为:  $a^{p-1} \equiv 1(\%p)$ 
13 ///降幂公式:  $A^x = A^{x \% \phi(C) + \phi(C)} \% C, (x > \phi(C))$ 
14
15 //求欧拉函数
16 int Euler(int n)
17 {
18     int euler = n;
19     for(int i = 1; i < primen; i++)
20         if(n % prime[i])
21         {
22             euler = euler / prime[i] * (prime[i] - 1);
23         }
24     return euler;
25 }
26 //预处理
27 int euler[NUM];
28 bool notPrim[NUM];
29 int Euler()
30 {
31     int i;
32     for(i = 0; i < NUM; i++)
33         euler[i] = i;
34     for(i = 2; i < NUM; i++)
35         if(!notPrim[i])
36         {
37             euler[i] = i - 1;
38             for(int j = i + i; j < NUM; j += i)
39             {
40                 notPrim[j] = true;
41                 euler[j] = euler[j] / i * (i - 1);
42             }
43         }
44 }
```

4.4 快速幂快速乘

```
1  //快速幂, 快速模幂
2  LL qpow(LL x, int k, LL mod)
3  {
4      LL ans = 1;
5      while(k)
6      {
7          if(k & 1) ans = ans * x % mod;
8          x = x * x % mod;
9          k >>= 1;
10     }
11     return ans;
12 }
```

```

14 //快速模乘  $a \times b \% mod$ 
15 //用于  $a \times b$  可能爆LL
16 LL mod_mult(LL a, LL b, LL mod)
17 {
18     LL res = 0;
19     if(a >= mod) a %= mod;
20     while(b)
21     {
22         if(b & 1)
23         {
24             res += a;
25             if(res >= mod) res -= mod;
26         }
27         a <<= 1;
28         if(a >= mod) a -= mod;
29         b >>= 1;
30     }
31     return res;
32 }

```

4.5 最大公约数 GCD

```

1 ///最大公约数gcd
2 /*gcd(a,b)的性质
3 gcd(0,0)=0, gcd(a,b)=gcd(b,a), gcd(a,b)=gcd(-a,b), gcd(a,b)=gcd(|a|,|b|), gcd(a,0)=|a|
4 gcd(a,ka)=|a|, ( $k \in Z$ ), gcd(a,b)=n gcd(a,b), gcd(a,gcd(b,c))=gcd(gcd(a,b),c)
5 gcd递归定理: gcd(a,b)=gcd(b,a%b)
6 最大公倍数 lcm(a,b)= $\frac{ab}{gcd(a,b)}$ 
7 n个数的gcd和lcm, 记第i个数

```

$$a_i = \prod_{k=1}^l p_k^{g_{ik}}$$

, 则

$$\gcd(a_1, a_2, \dots, a_n) = \prod_{k=1}^l p_k^{\min\{g_{1k}, g_{2k}, \dots, g_{nk}\}}$$

,

$$\text{lcm}(a_1, a_2, \dots, a_n) = \prod_{k=1}^l p_k^{\max\{g_{1k}, g_{2k}, \dots, g_{nk}\}}$$

.

```

8 一段区间  $[l, r]$  ( $r = l \rightarrow n$ ) 的gcd最多变化  $\log$  次
9 1, 2, ..., n 的lcm为, 如果n是某质数p的幂, 则  $\text{lcm}(n) = \text{lcm}(n-1) \times p$ , 否则  $\text{lcm}(n) = \text{lcm}(n-1)$ .

```

```

10 */
11 //欧几里得算法  $O(\log n)$ 
12 //递归
13 int gcd(int a, int b) {return b ? gcd(b, a % b) : a;}
14 //循环
15 int gcd(int a, int b)
16 {
17     int t;
18     while(b) t = a % b, a = b, b = t;
19     return a;
20 }
21 //小数的gcd
22 //EPS控制精度
23 double fgcd(double a, double b)
24 {
25     if(-EPS < b && b < EPS) return a;
26     return fgcd(b, fmod(a, b));
27 }
28 ///扩展欧几里得算法
29 void exgcd(int a, int b, int &g, int &x, int &y)
30 {
31     if(b) exgcd(b, a % b, g, y, x), y -= a / b * x;
32     else x = 1, y = 0, g = a;

```

```

33 }
34 //应用
35 //1. 求解 $ax + by = c$ 的 $x$ 的最小正整数解
36 // $x$ 的通解为 $x_0 + b/\gcd(a, b) * k$ 
37 int solve(int a, int b, int c)
38 {
39     int g = exgcd(a, b, x, y), t = b / g;
40     if(c % g) return -1;// $c \% \gcd(a, b) \neq 0$ 无解
41     int x0 = x * c / g;
42     x0 = ((x0 % t) + t) % t;
43     int y0 = (c - a * x0) / b;
44     return x0;
45 }
46 //2. 求解 $a$ 关于 $p$ 的逆元
47 //
48
49 /*题目:
50 1. 给 $n$ 个数,  $q$ 个询问, 每个询问查询区间 $[l, r]$ 中每个子区间的区间 $\gcd$ 之和,
    即查询 $\sum_{i=l}^r \sum_{j=i}^r \gcd(a_i, a_{i+1}, \dots, a_j)$ . ( $1 \leq n, q \leq 10^4$ )
51 来源: 2015年多校第八场1002, hdu5381 The sum of gcd
52 标签: gcd, 莫队, ST, 二分
53 做法: 由于以 $l$ 为左端点的所有区间中, 它们的 $\gcd$ 最多变化 $\log a_l$ 次, 并且 $\gcd$ 是递减的. 因此用ST表预处理,
    可以在 $O(1)$ 时间内求出任意区间的 $\gcd$ , 然后用二分求出对于每个左端点 $l$ 和每个右端点 $r$ ,
    找出每种 $\gcd$ 的范围, 这样就可以在 $\log N$ 时间内求出任意以 $l$ 为左端点, 或以 $r$ 为右端点,
    到某位置的所有 $\gcd$ 之和, 即 $\sum_{i=l}^r \gcd(a_l, a_{l+1}, \dots, a_i)$ , 最后离线询问, 用莫队求解. 时间复杂度:
     $O(N \log^2 N)$ .
54 */

```

4.6 逆元 Inverse

```

1  ///逆元inverse
2  //定义: 如果 $a \cdot b \equiv 1(\%MOD)$ , 则 $b$ 是 $a$ 的逆元(模逆元, 乘法逆元)
3  //a的逆元存在条件:  $\gcd(a, MOD) == 1$ 
4  //性质: 逆元是积性函数, 如果 $c = a \cdot b$ , 则  $inv[c] = inv[a] \cdot inv[b] \% MOD$ 
5  //方法一: 循环找解法(暴力)
6  // $O(n)$  预处理 $inv[1-n]$ :  $O(n^2)$ 
7  LL getInv(LL x, LL MOD)
8  {
9      for(LL i = 1; i < MOD; i++)
10         if(x * i % MOD == 1)
11             return i;
12     return -1;
13 }
14
15 //方法二: 费马小定理和欧拉定理
16 //费马小定理: $a^{(p-1)} \equiv 1(\%p)$ , 其中 $p$ 是质数, 所以 $a$ 的逆元是 $a^{(p-2)} \% p$ 
17 //欧拉定理: $x^{\phi(m)} \equiv 1(\%m)$   $x$ 与 $m$ 互素,  $m$ 是任意整数
18 // $O(\log n)$ (配合快速幂), 预处理 $inv[1-n]$ :  $O(n \log n)$ 
19 LL qpow(LL x, LL k, LL MOD) {...}
20 LL getInv(LL x, LL MOD)
21 {
22     //return qpow(x, euler_phi(MOD) - 1, MOD);
23     return qpow(x, MOD - 2, MOD);//MOD是质数
24 }
25
26 //方法三: 扩展欧几里得算法
27 //扩展欧几里得算法可解决  $a \cdot x + b \cdot y = \gcd(a, b)$ 
28 //所以 $a \cdot x \% MOD = \gcd(a, b) \% MOD (b = MOD)$ 
29 // $O(\log n)$ , 预处理 $inv[1-n]$ :  $O(n \log n)$ 
30 inline void exgcd(LL a, LL b, LL &g, LL &x, LL &y)
31 {

```

```

32     if(b) exgcd(b, a % b, g, y, x), y -= (a / b) * x;
33     else g = a, x = 1, y = 0;
34 }
35
36 LL getInv(LL x, LL mod)
37 {
38     LL g, inv, tmp;
39     exgcd(x, mod, g, inv, tmp);
40     return g != 1 ? -1 : (inv % mod + mod) % mod;
41 }
42
43 //方法四：积性函数
44 //已处理inv[1] — inv[n-1], 求inv[n], (MOD > n) (MOD为质数,不存在逆元的i干扰结果)
45 //MOD = x · n - y (0 ≤ y < n) ⇒ x · n = y(%MOD) ⇒ x · n · inv[y] = y · inv[y] = 1(%MOD)
46 //所以inv[n] = x · inv[y] (x = MOD - MOD/n, y = MOD%n)
47 //O(log n) 预处理inv[1~n]: O(n)
48 LL inv[NUM];
49 void inv_pre(LL mod)
50 {
51     inv[0] = inv[1] = 1LL;
52     for(int i = 2; i < NUM; i++)
53         inv[i] = (mod - mod / i) * inv[mod % i] % mod;
54 }
55 LL getInv(LL x, LL mod)
56 {
57     LL res = 1LL;
58     while(x > 1)
59     {
60         res = res * (mod - mod / x) % mod;
61         x = mod % x;
62     }
63     return res;
64 }
65 //方法五：积性函数+因式分解
66 //预处理出所有质数的逆元, 采用exgcd来实现素数O(log n)求逆
67 //采用质因数分解, 可在O(log n)求出任意一个数的逆元
68 //预处理O(n log n), 单个O(log n)

```

4.7 模运算 Module

```

1  /*
2   模 (Module)
3   1. 基本运算
4       Add: (a + b) % p = (a % p + b % p) % p
5       Subtract: (a - b) % p = ((a % p - b % p) % p + p) % p
6       Multiply: (a * b) % p = ((a % p) * (b % p)) % p
7       Divide: (a / b) % p = (a * b-1) % p, b-1是b关于p的逆元
8       Power: (ab) % p = ((a % p)b) % p
9
10      对一个数连续取模, 有效的取模次数小于O(log n)
11  2. 推论
12      若a ≡ b(%p), c ≡ d(%p), 则(a + c) ≡ (b + d)(%p), (a - c) ≡ (b - d)(%p), (a * c) ≡ (b * d)(%p), (a/c) ≡ (b/d)(%p)
13
14  3. 费马小定理
15      若p是素数, 对任意正整数x, 有 xp ≡ x(%p).
16  4. 欧拉定理
17      若p与x互素, 则有 xφ(p) ≡ 1(%p).
18  5. n! = ape, gcd(a, p) = 1, p是素数
19      e = (n/p + n/p2 + n/p3 + ...) (a不能被p整除)
20      威尔逊定理: (p-1)! ≡ -1(%p) (当且仅当p是素数)
21      n!中不能被p整除的数的积: n! = (p-1)!(n/p) × (n mod p)!
22      n!中能被p整除的项为: p, 2p, 3p, ..., (n/p)p, 除以p得到1, 2, 3, ..., n/p (问题从缩减到n/p)

```

```

23 在 $O(p)$ 时间内预处理除 $0 \leq n < p$ 范围内的 $n! \bmod p$ 的表
24 可在 $O(\log_p n)$ 时间内算出答案
25 若不预处理,复杂度为 $O(p \log_p n)$ 
26 */
27 int fact[MAX_P]; // 预处理 $n! \bmod p$ 的表. $O(p)$ 
28 // 分解 $n! = a \cdot p^e$ . 返回 $a \% p$ .  $O(\log_p n)$ 
29 int mod_fact(int n, int p, int &e)
30 {
31     e = 0;
32     if(n == 0) return 1;
33     // 计算 $p$ 的倍数的部分
34     int res = mod_fact(n / p, p, e);
35     e += n / p;
36     // 由于 $(p-1)! \equiv -1$ , 因此只需知 $n/p$ 的奇偶性
37     if(n / p % 2) return res * (p - fact[n % p]) % p;
38     return res * fact[n % p] % p;
39 }
40
41 /*
42 6.  $n! = t(p^c)^u$ ,  $\gcd(t, p^c) = 1$ ,  $p$ 是素数
43  $1 \sim n$ 中不能被 $p$ 整除的项模 $p^c$ , 以 $p^c$ 为循环节, 预处理出 $n! \% p^c$ 的表
44  $1 \sim n$ 中能被 $p$ 整除的项, 提取  $n/p$  个 $p$ 出来, 剩下阶乘 $(n/p)!$ , 递归处理
45 最后,  $t$ 还要乘上 $p^u$ 
46 */
47 LL fact[NUM];
48 LL qpow(LL x, LL k, LL mod);
49 inline void pre_fact(LL p, LL pc) // 预处理 $n! \% p^c$ ,  $O(p^c)$ 
50 {
51     fact[0] = fact[1] = 1;
52     for(int i = 2; i < pc; i++)
53     {
54         if(i % p) fact[i] = fact[i - 1] * i % pc;
55         else fact[i] = fact[i - 1];
56     }
57 }
58 // 分解 $n! = t(p^c)^u$ ,  $n! \% pc = t \cdot p^u \% pc$ 
59 inline void mod_factorial(LL n, LL p, LL pc, LL &t, LL &u)
60 {
61     for(t = 1, u = 0; n; u += (n /= p))
62         t = t * fact[n % pc] % pc * qpow(fact[pc - 1], n / pc, pc) % pc;
63 }
64 /*
65 7. 大组合数求模,  $mod$ 不是质数
66 求 $C_n^m \% mod$ 
67 1) 因式分解:  $mod = p_1^{c_1} p_2^{c_2} \cdots p_k^{c_k}$ 
68 2) 对每个因子 $p^c$ , 求 $C_n^m \% p^c = \frac{n! \% p^c}{m! \% p^c (n-m)! \% p^c} \% p^c$ 
69 3) 根据中国剩余定理求答案(注: 逆元采用扩展欧几里得求法)
70 */
71 LL fact[NUM];
72 LL prim[NUM], prim_num;
73 LL pre_prim();
74 LL pre_fact(LL p, LL pc);
75 LL mod_factorial(LL n, LL p, LL pc, LL &t, LL &u);
76 LL qpow(LL x, LL k, LL mod);
77 LL getInv(LL x, LL mod);
78
79 LL C(LL n, LL m, LL mod)
80 {
81     if(n < m) return 0;
82     LL p, pc, tmpmod = mod;
83     LL Mi, tmpans, t, u, tot;
84     LL ans = 0;
85     int i, j;
86     // 将 $mod$ 因式分解,  $mod = p_1^{c_1} p_2^{c_2} \cdots p_k^{c_k}$ 

```

```

87     for(i = 0; prim[i] <= tmpmod; i++)
88         if(tmpmod % prim[i] == 0)
89             {
90                 for(p = prim[i], pc = 1; tmpmod % p == 0; tmpmod /= p)
91                     pc *= p;
92                 // 求  $C_n^{k \% p^c}$ 
93                 pre_fact(p, pc);
94                 mod_factorial(n, p, pc, t, u); //  $n!$ 
95                 tmpans = t;
96                 tot = u;
97                 mod_factorial(m, p, pc, t, u); //  $m!$ 
98                 tmpans = tmpans * getInv(t, pc) % pc; // 求逆元: 采用扩展欧几里得定律
99                 tot -= u;
100                mod_factorial(n - m, p, pc, t, u); //  $(n - m)!$ 
101                tmpans = tmpans * getInv(t, pc) % pc;
102                tot -= u;
103                tmpans = tmpans * qpow(p, tot, pc) % pc;
104                // 中国剩余定理
105                Mi = mod / pc;
106                ans = (ans + tmpans * Mi % mod * getInv(Mi, pc) % mod) % mod;
107            }
108     return ans;
109 }
110
111 /*
112 8. 大组合数求模, mod是素数, Lucas定理
113 Lucas定理:  $C_n^m \% mod = C_{n/mod}^{m/mod} \cdot C_{n \% mod}^{m \% mod} \% mod$ 
114 采用  $O(n)$  方法预处理  $0 \sim n-1$  的  $n! \% mod$  和每个数的逆元, 则可在  $O(\log n)$  时间求出  $C_n^k \% mod$ 
115 */
116 LL fact[NUM], inv[NUM];
117 void Lucas_init(LL mod); // 预处理
118 LL Lucas(LL n, LL m, LL mod) // mod是质数
119 {
120     LL a, b, res = 1LL;
121     while(n && m)
122     {
123         a = n % mod, b = m % mod;
124         if(a < b) return 0LL;
125         res = res * fact[a] % mod * inv[fact[b] * fact[a - b] % mod, mod] % mod;
126         n /= mod, m /= mod;
127     }
128     return res;
129 }

```

4.8 幂 p 的原根

```

1  /// 幂模 p 与原根 primitive root
2  // a 模 p 的次数 (或阶) 指  $e_p(a) = (\text{使得 } a^e \equiv 1(\%p) \text{ 的最小指数 } e \geq 1)$ 
3  // (次数整除性质), 设 a 是不被素数 p 整除的整数, 假设  $a^n \equiv 1(\%p)$ , 则次数  $e_p(a)$  整除 n, 特别地,
   次数  $e_p(a)$  总整除  $p-1$ .
4  // 原根: 具有最高次数  $e_p(g) = p-1$  的数 g 被称为模 p 的原根
5  // 原根定理: 每个素数 p 都有恰好  $\phi(p-1)$  个原根
6  // 求最小原根: 原根分布较广, 而且最小的原根通常也比较小,
   可以从小到大通过枚举每个正整数来快速的寻找一个原根. 对于一个待检查的 p, 对于  $p-1$  的每一个素因子 a,
   检查  $g^{(p-1)/a} \equiv 1(\%p)$  是否成立, 如果成立则不是.
7  //  $1e9 + 7$  的最小原根为 5,  $1e9 + 7$  的最小原根为 13
8  LL qpow(LL x, LL k, LL mod);
9  bool g_test(LL g, LL p, vector<LL> &factor)
10 {
11     for(int i = 0; i < (int)factor.size(); i++)
12         if(qpow(g, (p - 1) / factor[i], p) == 1)
13             return false;

```

```

14     return true;
15 }
16 LL primitive_root(LL p)
17 {
18     vector<LL> factor;
19     //求p-1的素因子
20     LL tmp = p - 1;
21     for(LL i = 2; i * i <= tmp; i++)
22         if(tmp % i == 0)
23         {
24             factor.push_back(i);
25             while(tmp % i == 0) tmp /= i;
26         }
27     if(tmp != 1) factor.push_back(tmp);
28     LL g = 1;
29     while(true)
30     {
31         if(g_test(g, p, factor))
32             return g;
33         g++;
34     }
35 }

```

4.9 中国剩余定理和线性同余方程组

```

1  /*线性同余方程
2   $a_i \times x \equiv b_i (\% m_i) \quad (1 \leq i \leq n)$ 
3  如果方程组有解，那么一定有无穷有无穷多解，解的全集可写为  $x \equiv b (\% m)$  的形式。
4  对方程逐一求解。令  $b = 0, m = 1$ ;
5  1.  $x \equiv b (\% m)$  可写为  $x = b + m \cdot t$ ;
6  2. 带入第  $i$  个式子:  $a_i(b + m \cdot t) \equiv b_i (\% m_i)$ , 即  $a_i \cdot m \cdot t \equiv b_i - a_i \cdot b (\% m_i)$ 
7  3. 当  $\gcd(m_i, a_i \cdot m)$  无法整除  $b_i - a_i \cdot b$  时原方程组无解, 否则用  $\text{exgcd}$ , 求出满组条件的最小非负整数  $t$ ,
8
9  中国剩余定理:
10     对  $x \equiv a_i (\% m_i) (1 \leq i \leq n)$ , 其中  $m_1, m_2, \dots, m_n$  两两互素,  $a_1, a_2, \dots, a_n$  是任意整数, 则有解:
11      $M = \prod m_i, b = \sum_i a_i M_i^{-1} M_i (M_i = M / m_i)$ 
12 */
13 int gcd(int a, int b);
14 int getInv(int x, int mod);
15 pair<int, int> linear_congruence(const vector<int> &A, const vector<int> &B, const vector<int> &M)
16 {
17     //初始解设为表示所有整数的  $x \equiv 0 (\% 1)$ 
18     int x = 0, m = 1;
19     for(int i = 0; i < A.size(); i++)
20     {
21         int a = A[i]*m, b = B[i] - A[i] * x, d = gcd(M[i], a);
22         if(b % d == 0) return make_pair(0, -1); // 无解
23         int t = b/d * getInv(a / d, M[i] / d) % (M[i] / d);
24         x = x + m * t;
25         m *= M[i] / d;
26     }
27     return make_pair(x % m, m);
28 }

```

4.10 组合与组合恒等式

```

1  /*1. 组合: 从  $n$  个不同的元素中取  $r$  个的方案数  $C_n^r$ :

```

$$C_n^r = \begin{cases} \frac{n!}{r!(n-r)!}, & n \geq r \\ 1, & n \geq r = 0 \\ 0, & n < r \end{cases}$$

推论1: $C_n^r = C_n^{n-r}$

推论2(Pascal公式): $C_n^r = C_{n-1}^r + C_{n-1}^{r-1}$

推论3: $\sum_{k=r-1}^{n-1} C_k^{r-1} = C_{n-1}^{r-1} + C_{n-2}^{r-2} + \dots + C_{r-1}^{r-1} = C_n^r$

2. 从重集 $B = \{\infty \cdot b_1, \infty \cdot b_2, \dots, \infty \cdot b_n\}$ 的 r -组合数 $F(n, r)$ 为 $F(n, r) = C_{n+r-1}^r$

3. 二项式定义

当 n 是一个正整数时, 对任何 x 和 y 有:

$$(x+y)^n = \sum_{k=0}^n C_n^k x^k y^{n-k}$$

令 $y=1$, 有:

$$(1+x)^n = \sum_{k=0}^n C_n^k x^k = \sum_{k=0}^n C_n^{n-k} x^k$$

广义二项式定理:

广义二项式系数: 对于任何实数 α 和整数 k , 有

$$C_\alpha^k = \begin{cases} \frac{\alpha(\alpha-1)\dots(\alpha-k+1)}{k!} & k > 0 \\ 1 & k = 0 \\ 0 & k < 0 \end{cases}$$

设 α 是一个任意实数, 则对满足 $|\frac{x}{y}| < 1$ 的所有 x 和 y , 有

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

推论: 令 $z = \frac{x}{y}$, 则有

$$(1+z)^\alpha = \sum_{k=0}^{\infty} C_\alpha^k z^k, |z| < 1$$

令 $\alpha = -n$ (n 是正整数), 有

$$(1+z)^{-n} = \frac{1}{(1+z)^n} = \sum_{k=0}^{\infty} (-1)^k C_{n+k-1}^k z^k$$

又令 $z = -rz$, (r 为非零常数), 有

又令 $n=1$, 有

$$\frac{1}{1+z} = \sum_{k=0}^{\infty} (-1)^k z^k$$

令 $z = -z$, 有

$$\frac{1}{1-z} = \sum_{k=0}^{\infty} z^k$$

令 $\alpha = \frac{1}{2}$, 有

$$\sqrt{1+z} = 1 + \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k \cdot 2^{2k-1}} C_{2k-2}^{k-1} z^k$$

4. 组合恒等式

$$1. \sum_{k=0}^n C_n^k = 2^n$$

$$2. \sum_{k=0}^n (-1)^k C_n^k = 0$$

3. 对于正整数 n 和 k ,

$$C_n^k = \frac{n}{k} C_{n-1}^{k-1}$$

4. 对于正整数 n ,

$$\sum_{k=0}^n k C_n^k = \sum_{k=1}^n k C_n^k = n \cdot 2^{n-1}$$

5. 对于正整数 n ,

$$\sum_{k=0}^n (-1)^k k C_n^k = 0$$

6. 对于正整数 n ,

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

7. 对于正整数 n ,

$$\sum_{k=0}^n \frac{1}{k+1} C_n^k = \frac{2^{n+1}-1}{n+1}$$

8. (Vandermonde 恒等式) 对于正整数 n, m 和 p , 有 $p \leq \min m, n$,

$$\sum_{k=0}^p C_n^k C_m^{p-k} = C_{m+n}^p$$

9. (令 $p=m$) 对于任何正整数 n, m ,

$$\sum_{k=0}^m C_m^k C_n^k = C_{m+n}^m$$

10. (又令 $m=n$) 对于任何正整数 n ,

$$\sum_{k=0}^n (C_n^k)^2 = C_{2n}^n$$

11. 对于非负整数 p, q 和 n ,

$$\sum_{k=0}^p C_p^k C_q^k C_{n+k}^{p+q} = C_n^p C_n^q$$

12. 对于非负整数 p, q 和 n ,

$$\sum_{k=0}^p C_p^k C_q^k C_{n+p+q-k}^{p+q} = C_{n+p}^p C_{n+q}^q$$

13. 对于非负整数 n, k ,

$$\sum_{i=0}^n C_i^k = C_{n+1}^{k+1}$$

14. 对于所有实数 α 和非负整数 k ,

$$\sum_{j=0}^k C_{\alpha+j}^j = C_{\alpha+k+1}^k$$

15.

$$\sum_{k=0}^n \frac{2^{k+1}}{k+1} C_n^k = \frac{3^{n+1}-1}{n+1}$$

16.

$$\sum_{k=0}^m C_{n-k}^{m-k} = C_{n+1}^m$$

17.

$$\sum_{k=m}^n C_k^m C_n^k = C_n^m 2^{n-m}$$

18.

$$\sum_{k=0}^m (-1)^k C_n^k = (-1)^m C_{n-1}^m$$

31 | */

4.11 排列 Permutation

1 | /*排列

2 | *排列: 从集合 $A=\{a_1, a_2, \dots, a_n\}$ 的 n 个元素中取 r 个按照一定的次序排列起来, 称为集合 A 的 r -排列。

3 | * 记其排列数:

$$P_n^r = \begin{cases} 0, & n < r \\ 1, & n \geq r = 0 \\ n(n-1) \cdots (n-r+1) = \frac{n!}{(n-r)!}, & r \leq n \end{cases}$$

4 | * 推论: 当 $n \geq r \geq 2$ 时, 有 $P_n^r = n P_{n-1}^{r-1}$

5 | * 当 $n \geq r \geq 2$ 是, 有 $P_n^r = r P_{n-1}^{r-1} + P_{n-1}^r$

6 | *

7 | *圆排列: 从集合 $A=\{a_1, a_2, \dots, a_n\}$ 的 n 个元素中取出 r 个元素按照某种顺序排成一个圆圈, 称这样的排列为圆排列。

8 | * 集合 A 中 n 个元素的 r 圆排列的个数为:

$$\frac{P_n^r}{r} = \frac{n!}{r(n-r)!}$$

9 | *

10 | *重排列: 从重集 $B=\{k_1 \cdot b_1, k_2 \cdot b_2, \dots, k_n \cdot b_n\}$ 中选取 r 个元素按照一定的顺序排列起来, 称这种 r -排列为重排列。

11 | * 重集 $B=\{\infty \cdot b_1, \infty \cdot b_2, \dots, \infty \cdot b_n\}$ 的 r -排列的个数为 n^r 。

12 | * 重集 $B = \{n_1 \cdot b_1, n_2 \cdot b_2, \dots, n_k \cdot b_k\}$ 的全排列的个数为

$$\frac{n!}{n_1! \cdot n_2! \cdot \dots \cdot n_k!}, n = \sum_{i=1}^k n_i$$

13 | *

14 | * 错排: $\{1, 2, \dots, n\}$ 的全排列, 使得所有的 i 都有 $a_i \neq i$, $a_1 a_2 \dots a_n$ 是其的一个排列

15 | * 错排数

$$D_n = n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \dots + (-1)^n \frac{1}{n!}\right)$$

16 | * 递归关系式:

$$\begin{cases} D_n = (n-1)(D_{n-1} + D_{n-2}), & n > 2 \\ D_0 = 1, D_1 = 0 \end{cases}$$

17 | * 性质:

$$\lim_{n \rightarrow \infty} \frac{D_n}{n!} = e^{-1}$$

18 | * 前 17 个错排值

	n	0	1	2	3	4	5
	D_n	1	0	1	2	9	44
	n	6	7	8	9	10	11
19	D_n	265	1845	14833	133496	1334961	14684570
	n	12	13	14	15	16	17
	D_n	176214841	2290792932	32071101049	481066515734	7697064251745	130850092279664

20 |

21 | * 相对位置上有限制的排列的问题:

22 | * 求集合 $\{1, 2, 3, \dots, n\}$ 的不允许出现 $12, 23, 34, \dots, (n-1)n$ 的全排列数为

23 |

$$Q_n = n! - C_{n-1}^1(n-1)! + C_{n-1}^2(n-2)! - \dots + (-1)^{n-1} C_{n-1}^{n-1} \cdot 1!$$

24 | * 当 $n \geq 2$ 时, 有 $Q_n = D_n + D_{n-1}$

25 | * 求集合 $\{1, 2, 3, \dots, n\}$ 的圆排列中不出现 $12, 23, 34, \dots, (n-1)n, n1$ 的圆排列个数为:

26 |

$$(n-1)! - C_n^1(n-2)! + \dots + (-1)^{n-1} C_n^{n-1} 0! + (-1)^n C_n^n \cdot 1$$

27 |

28 | * 一般限制的排列:

29 | * 棋盘: 设 n 是一个正整数, $n \times n$ 的格子去掉某些格后剩下的部分称为棋盘 (可能不去掉)

30 | * 棋子问题: 在给定棋盘 C 中放入 k 个无区别的棋子, 要求每个棋子只能放一格, 且各子不同行不同列,

31 | 求不同的放法数 $r_k(C)$

32 | * 棋子多项式: 给定棋盘 C , 令 $r_0(C) = 1$, n 为 C 的格子数, 则称

$$R(C) = \sum_{k=0}^n r_k(C) x^k$$

为棋盘 C 的棋子多项式

33 | * 定理 1: 给定棋盘 C , 指定 C 中某格 A , 令 C_i 为 C 中删去 A 所在列与行所剩的棋盘, C_e 为 C 中删去格 A 所剩的棋盘, 则

34 | *

$$R(C) = xR(C_i) + R(C_e)$$

35 | * 设 C_1 和 C_2 是两个棋盘, 若 C_1 的所有格都不与 C_2 的所有格同行同列, 则称两个棋盘是独立的.

36 | * 定理 2: 若棋盘 C 可分解为两个独立的棋盘 C_1 和 C_2 , 则

$$R(C) = R(C_1)R(C_2)$$

37 | * n 元有禁位的排列问题: 求集合 $\{1, 2, \dots, n\}$ 的所有满足 $i(i = 1, 2, \dots, n)$ 不排在某些已知位的全排列数.

38 | * n 元有禁位的排列数为

$$n! - r_1(n-1)! + r_2(n-2)! - \dots + (-1)^n r_n$$

其中 r_i 为将 i 个棋子放入禁区棋盘的方式数, $i = 1, 2, \dots, n$

39 | */

4.12 全排列 Full Permutation

1 // 全排列 Full permutation

2 // 排列数为 $n!$

3 // 求全排列

4 // next_permutation 求下一个排列, 如果是最后一个排列则返回 false

```

5 void FullPermutation()
6 {
7     //求所有全排列需排好序
8     do
9     {
10         //do someting
11     }
12     while(next_permutation(perm2, perm2 + n));
13 }
14
15 ///康托展开
16 //把一个整数X展开成如下形式:  $X = a_n(n-1)! + a_{n-1}(n-2)! + \dots + a_i(i-1)! + \dots + a_2 \cdot 1! + a_1 \cdot 0!$ 
17 //其中, a为整数, 并且  $0 \leq a_i < i (1 \leq i \leq n)$ . 这就是康托展开.
18 //应用
19 //1~n的全排列中某一排列在所有排列中的位置
20 //字典序比排列P小的排列个数为:  $X_P = \sum_{i=1}^n count_{j=i+1}^n a[j] < a[i] \times (n-i-1)!$ 
21 //例: {1, 3, 2, 4}在1~4的所有全排列的位置为 ( $X_P = 0 \times 3! + 1 \times 2! + 0 \times 1! + 0 \times 0!$ ) + 1 = 3
22 const int PermSize = 12;
23 long long fac[PermSize] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800}; //n!
24 long long Cantor(int s[], int n)
25 {
26     int i, j, temp;
27     long long num = 0;
28     for(i = 0; i < n; i++)
29     {
30         temp = 0;
31         for(int j = i + 1; j < n; j++)
32         {
33             if(s[j] < s[i])
34                 temp++; //判断几个数小于它
35         }
36         num += fac[n - i - 1] * temp; //(或num=num+fac[n-i-1]*temp;)
37     }
38     return num + 1;
39 }
40
41
42 //逆运算 逆康托展开
43 //已知1~n的全排列中的某一排列permutation在所以全排列中的位置, 求该排列
44 //第pos个排列, a[i]为排除a[1...i-1]后第((pos-1)/(n-i-1)! + 1)小的数字
45 //例: 1~4中第3大的数为: 2/3!=0...2, p[0]=1(1, 2, 3, 4第1小的数); 2/2!=1...0, p[1]=3(2, 3, 4第2小的数; 0/1!=0...0, p[2]=2(2, 4第1小的数); 0/0!=0...0, p[3]=4(4第1小的数); 故第3个的排列为{1,3,2,4}
46 int fac[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880};
47 int perm[9];
48 void unCantor(int pos, int n)
49 {
50     int i, j;
51     bool vis[12] = {0};
52     pos--;
53     for(i = 0; i < n; i++)
54     {
55         int t = pos / fac[n - i - 1];
56         for(j = 1; t && j <= n; j++)
57             if(!vis[j]) t--;
58         vis[perm[i] = j] = 1;
59         pos %= fac[n - i - 1];
60     }
61 }

```

4.13 母函数 Generating Function

```

1 /*母函数
2  * 普通母函数:
3  * 定义: 给定一个无穷序列  $(a_0, a_1, a_2, \dots, a_n, \dots)$  (简记为  $\{a_n\}$ ), 称函数
4  *

$$f(x) = a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n + \dots = \sum_{i=1}^{\infty} a_i x^i$$

5  * 为序列  $\{a_n\}$  的普通母函数
6  * 常见普通母函数:
7  * 序列  $(C_n^0, C_n^1, C_n^2, \dots, C_n^n)$  的普通母函数为  $f(x) = (1+x)^n$ 
8  * 序列  $(1, 1, \dots, 1, \dots)$  的普通母函数为  $f(x) = \frac{1}{1-x}$ 
9  * 序列  $(C_{n-1}^0, -C_n^1, C_{n+1}^2, \dots, (-1)^k C_{n+k-1}^k, \dots)$  的普通母函数为  $f(x) = (1+x)^{-n}$ 
10 * 序列  $(C_0^0, C_2^1, C_4^2, \dots, C_{2n}^n, \dots)$  的普通母函数为  $f(x) = (1-4x)^{-1/2}$ 
11 * 序列  $(0, 1 \times 2 \times 3, 2 \times 3 \times 4, \dots, n \times (n+1) \times (n+2), \dots)$  的普通母函数为  $\frac{6}{(1-x)^4}$ 
12 *
13 * 指数母函数
14 * 定义: 称函数

$$f_e(x) = a_0 + a_1 \frac{x^1}{1!} + a_2 \frac{x^2}{2!} + \dots + a_n \frac{x^n}{n!} + \dots = \sum_{n=0}^{\infty} a_n \frac{x^n}{n!}$$

15 为序列  $(a_0, a_1, \dots, a_n, \dots)$  的指数母函数。
16 * 常见指数母函数为
17 * 序列  $(1, 1, \dots, 1, \dots)$  的指数母函数为  $f_e(x) = e^x$ 
18 *  $n$  是整数, 序列  $(P_n^0, P_n^1, \dots, P_n^n)$  的指数母函数为  $f_e(x) = (1+x)^n$ 
19 * 序列  $(P_0^0, P_2^1, P_4^2, \dots, P_{2n}^n, \dots)$  的指数母函数为  $f_e(x) = (1-4x)^{-1/2}$ 
20 * 序列  $(1, \alpha, \alpha^2, \dots, \alpha^n, \dots)$  的指数母函数为  $f_e(x) = e^{\alpha x}$ 
21 *
22 * 指数母函数和普通母函数的关系: 对同一序列的  $\{a_n\}$  的普通母函数  $f(x)$  和指数母函数  $f_e(x)$  有:

$$f(x) = \int_0^\infty e^{-s} f_e(sx) ds$$

23 *
24 * 母函数的基本运算:
25 * 设  $A(x), B(x)$ ,
 $C(x)$  分别是序列  $(a_0, a_1, \dots, a_r, \dots), (b_0, b_1, \dots, b_r, \dots), (c_0, c_1, \dots, c_r, \dots)$  的普通 (指数) 母函数, 则有:
26 *  $C(x) = A(x) + B(x)$  当且仅当对所有的  $i$ , 都有  $c_i = a_i + b_i (i = 0, 1, 2, \dots, r, \dots)$ .
27 *  $C(x) = A(x)B(x)$  当且仅当对所有的  $i$ , 都有  $c_i = \sum_{k=0}^i a_k b_{i-k} (i = 0, 1, 2, \dots, r, \dots)$ .
28 */
29 /*母函数在组合排列上的应用
30 从  $n$  个不同的物体中允许重复地选取  $r$  个物体, 但是每个物体出现偶数次的方式数。
31

$$f(x) = (1 + x^2 + x^4 + \dots)^n = \left(\frac{1}{1-x^2}\right)^n = \sum_{r=0}^{\infty} C_{n+r-1}^r x^{2r}$$

32 故答案为  $a_r = C_{n+r-1}^r$ 
33 */

```

4.14 整数拆分和 Ferrers 图

```

1 /*整数拆分
2 问题: 将正整数  $n$  拆分为若干个正整数部分, 或将  $n$  个无区别的球放入一些无区别的盒子中 (非空).
3 性质: 母函数应用
4 定义:
5 1. 用  $P(n)$  表示  $n$  拆分类数。
6 2. 用  $P_k(n)$  表示  $n$  拆分成  $1, 2, \dots, k$  的允许重复的方法数。
7 3. 用  $P_o(n)$  表示  $n$  拆分成奇整数的方法数。
8 4. 用  $P_d(n)$  表示  $n$  拆分成不同的整数的方法数。
9 5. 用  $P_t(n)$  表示  $n$  拆分成  $2$  的不同幂的方法数。
10 定理 1: 设  $a, b, c, \dots$  是大于  $\theta$  的正整数, 则

$$\frac{1}{(1-x^a)(1-x^b)(1-x^c)\dots}$$

11 的级数展开式中的  $x^n$  的系数等于把正整数  $n$  拆分成  $a, b, c, \dots$  的方法数  $P(n)$ .
12 推论 1:  $\{P_k(n)\}$  的普通母函数为

```

12

$$\frac{1}{(1-x)(1-x^2)\cdots(1-x^k)}$$

13 推论2: $\{P(n)\}$ 的普通母函数为

14

$$\frac{1}{(1-x)(1-x^2)(1-x^3)\cdots}$$

15 推论3: $\{P_o(n)\}$ 的普通母函数为

16

$$\frac{1}{(1-x)(1-x^3)(1-x^5)(1-x^7)\cdots}$$

17 定理2: 设 a, b, c, \dots 都是大于0的正整数, 则

18

$$(1+x^a)(1+x^b)(1+x^c)\cdots$$

19 的级数展开式中 x^n 项的系数就是把 n 拆分成 a, b, c, \dots 的和, 且 a, b, c, \dots 最多只出现一次的方法数。20 推论1: $\{P_a(n)\}$ 的普通母函数是

21

$$(1+x)(1+x^2)(1+x^3)(1+x^4)\cdots$$

22 推论2: $\{P_t(n)\}$ 的普通母函数是

23

$$(1+x)(1+x^2)(1+x^4)(1+x^8)\cdots$$

24 定理3: (Euler) 对于正整数 n 都有

25

$$P_o(n) = P_a(n)$$

26 定理4: (Sylvester) 对于正整数 n , 有

27

$$P_t(n) = 1$$

28 定理5: 对于正整数 n , 有

29

$$P(n) < e^{3\sqrt{n}}$$

30 6. 将 $p(n)$ 生成函数配合五边形数定理, 可以得到以下的递归关系式

$$p(n) = \sum_i (-1)^{i-1} p(n - q_i)$$

32 其中 q_i 是第 i 个广义五边形数。33 Ferrers 图: 设 n 的一个拆分为

$$n = a_1 + a_2 + \cdots + a_k$$

34 并假设 $a_1 \geq a_2 \geq a_3 \geq \cdots \geq a_k \geq 1$

35

画一个由一行行的点所组成的图, 第一行有 a_1 的点, 第二行有 a_2 个点, \dots , 第 k 行有 a_k 个点, 则称此图
为 Ferrers 图。共轭图: 将一个 Ferrers 图的行列互换后仍是 Ferrers 图, 称互换后的 Ferrers 图为原图的共轭图。定
理 6: 正整数 n 拆分成 m 项的和的方式数等于 n 拆分成最大数为 m 的方式数 定理 7: 正整数 n 拆分成最多不超过 m 个项的
方式数等于 n 拆分成最大的数不超过 m 的方式数。*//应用: 将一个大小为 n 的集合划分为 m 个子集 (可以为空), 对任
意的 k 可以知道该划分中至少有 k 个元素的子集的数目 d , 问最坏情况下可以确定该集合至少有多少元素?
($1 \leq n \leq 10^8, 1 \leq m \leq 20000$) (即将整数 n 拆分为不超过 m 个整数后, 最坏情况下某一 Ferrers 图中最大矩形的点
数) source: Gym 100490B(ASC38) solve: 二分枚举划分的最大元素数 p , 使得任意的 $kd \leq p$, 那么第 2 个子集的元素
数为 $p/2, \dots$, 第 m 个子集的元素数为 p/m , 如果方案可行, 则 $p/1 + p/2 + \cdots + p/m \geq n$ */

4.15 鸽笼原理与 Ramsey 数

1 /*鸽笼原理:

2 * 简单形式: 如果把 $n+1$ 个物体放到 n 个盒子中去, 则至少有一个盒子中放有两个或更多的物体。3 * 一般形式: 设 q_i 是正整数 ($i = 1, 2, \dots, n$), $q \geq q_1 + q_2 + \cdots + q_n - n + 1$,如果把 q 个物体放入 n 个盒子中去, 则存在一个 i 使得第 i 个盒子中至少有 q_i 个物体。4 * 推论1: 如果把 $n(r-1)+1$ 个物体放入 n 个盒子中, 则至少存在一个盒子放有不少于 r 个物体。5 * 推论2: 对于正整数 $m_i (i = 1, 2, \dots, n)$, 如果 $\frac{\sum_{i=1}^n m_i}{n} > r -$ 1, 则至少存在一个 i , 使得 $m_i \geq r$ 。6 * 例: 在给定的 n 个整数 a_1, a_2, \dots, a_n 中, 存在 k 和 $l (0 \leq k < l \leq n)$, 使得 $a_{k+1} + a_{k+2} + \cdots + a_l$ 能被 n 整除

7 */

8 /*Ramsey 定理和 Ramsey 数

9 在人数为 6 的一群人中, 一定有三个人彼此相识, 或者彼此不相识。

10 在人数为 10 的一群人中, 一定有 3 个人彼此不相识或者 4 个人彼此相识。

11 在人数为 10 的一群人中, 一定有 3 个人彼此相识或者 4 个人彼此不相识。

12 在人数为 20 的一群人中, 一定有 4 个人彼此相识或者 4 个人彼此不相识。

```

13 |
14 | 设  $a, b$  为正整数, 令  $N(a, b)$  是保证有  $a$  个人彼此相识或者有  $b$  个人彼此不相识所需的最少人数, 则称  $N(a, b)$  为 Ramsey 数.
15 | Ramsey 数的性质:
16 |  $N(a, b) = N(b, a)$ 
17 |  $N(a, 2) = a$ 
18 | 当  $a, b \geq 2$  时,  $N(a, b)$  是一个有限数, 并且有  $N(a, b) \leq N(a-1, b) + N(a, b-1)$ 
19 | 当  $N(a-1, b)$  和  $N(a, b-1)$  都是偶数时, 则有  $N(a, b) \leq N(a-1, b) + N(a, b-1) - 1$ 
20 |
21 |
22 |
23 |
24 |
25 |
26 |
27 |
28 |
29 |
30 |
31 |

```

$N(a, b)$	2	3	4	5	6	7	8	9
2	2	3	4	5	6	7	8	9
3		6	9	14	18	23	28	36
4			18	24	44	66		
5				55	94	156		
6					178	322		
7						626		

```

21 | 如果把一个完全  $n$  角形, 用  $r$  中颜色  $c_1, c_2, \dots, c_r$  对其边任意着色。
22 | 设  $N(a_1, a_2, \dots, a_r)$  是保证下列情况之一出现的最小正整数:
23 |  $c_1$  颜色着色的一个完全  $a_1$  角形
24 | 用  $c_2$  颜色着色的一个完全  $a_2$  角形
25 | .....
26 | 或用颜色  $c_r$  着色的一个完全  $a_r$  角形
27 | 则称数  $N(a_1, a_2, \dots, a_r)$  为 Ramsey 数。
28 | 对与所有大于 1 的整数  $a_1, a_2, a_3$ , 数  $N(a_1, a_2, a_3)$  是存在的。
29 | 对于任意正整数  $m$  和  $a_1, a_2, \dots, a_m \geq 2$ , Ramsey 数  $N(a_1, a_2, \dots, a_m)$  是存在的。
30 | .....
31 | */

```

4.16 容斥原理

```

1 | /*容斥原理
2 | * 集合  $S$  中具有性质  $p_i (i = 1, 2, \dots, m)$  的元素所组成的集合为  $A_i$ , 则  $S$  中不具有性质  $p_1, p_2, \dots, p_m$  的元素个数为
3 | *  $|A_1 \cap A_2 \cap \dots \cap A_m| = |S| - \sum_{i=1}^m |A_i| + \sum_{i \neq j} |A_i \cap A_j| - \sum_{i \neq j \neq k} |A_i \cap A_j \cap A_k| + \dots + (-1)^m |A_1 \cap A_2 \cap \dots \cap A_m|$ 
4 | */
5 | /*重集的  $r$ -组合
6 | * 重集  $B = \{k_1 \cdot a_1, k_2 \cdot a_2, \dots, k_n \cdot a_n\}$  的  $r$ -组合数:
7 | * 利用容斥原理, 求出重集  $B' = \{\infty \cdot a_1, \infty \cdot a_2, \dots, \infty \cdot a_n\}$  的  $r$ -组合数  $F(n, r)$ 
8 | * 在求出满足至少含  $k_i + 1$  个  $a_i (1 \leq i \leq n)$  的  $r$ -组合数, 等同于重集  $B'$  的  $r - k_i - 1$ -组合数
9 | * .....
10 | * 右容斥原理得: 重集  $B$  的  $r$ -组合数为:
11 | *

```

$$F(n, r) - \sum_{i=1}^n F(n, r - k_i - 1) + \sum_{i \neq j} F(n, r - k_i - k_j - 2) + \dots + (-1)^n F(n, r - k_1 - k_2 - \dots - k_n - n)$$

```

12 | */

```

4.17 伪随机数的生成—梅森旋转算法

```

1 | //伪随机数生成—梅森旋转算法 (Mersenne twister)
2 | /*是一个伪随机数发生算法. 对于一个  $k$  位的长度, Mersenne Twister 会在  $[0, 2^k - 1] (1 \leq k \leq 623)$ 
   | 的区间之间生成离散型均匀分布的随机数. 梅森旋转算法的周期为梅森素数  $2^{19937} - 1$  */
3 | //32位算法
4 | int mtrand_init = 0;
5 | int mtrand_index;
6 | int mtrand_MT[624];
7 | void mt_srand(int seed)
8 | {
9 |     mtrand_index = 0;
10 |    mtrand_init = 1;
11 |    mtrand_MT[0] = seed;
12 |    for(int i = 1; i < 624; i++)
13 |    {

```

```

14     int t = 1812433253 * (mtrand_MT[i - 1] ^ (mtrand_MT[i - 1] >> 30)) + i; //0x6c078965
15     mtrand_MT[i] = t & 0xffffffff; //取最后的32位赋给MT[i]
16 }
17 }
18
19 int mt_rand()
20 {
21     if(!mtrand_init)
22         srand((int)time(NULL));
23     int y;
24     if(mtrand_index == 0)
25     {
26         for(int i = 0; i < 624; i++)
27         {
28             //2^31 - 1 = 0x7fff ffff 2^31 = 0x8000 0000
29             int y = (mtrand_MT[i] & 0x80000000) + (mtrand_MT[(i + 1) % 624] & 0x7fffffff);
30             mtrand_MT[i] = mtrand_MT[(i + 397) % 624] ^ (y >> 1);
31             if(y & 1) mtrand_MT[i] ^= 2567483615; // 0x9908b0df
32         }
33     }
34     y = mtrand_MT[mtrand_index];
35     y = y ^ (y >> 11);
36     y = y ^ ((y << 7) & 2636928640); //0x9d2c5680
37     y = y ^ ((y << 15) & 4022730752); // 0xefc60000
38     y = y ^ (y >> 18);
39     mtrand_index = (mtrand_index + 1) % 624;
40     return y;
41 }

```

4.18 异或 Xor

```

1  /// 异或Xor
2  /*
3  性质:  1.  $0 \oplus 1 = 0, 1 \oplus 0 = 1, 0 \oplus 0 = 0, 1 \oplus 1 = 1$ 
4          2. (交换律)  $a \oplus b = b \oplus a$ 
5          3. (结合律)  $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ 
6          4.  $a \oplus a = 0$ 
7          5.  $0 \oplus a = a$ 
8          6. (二进制分解)  $a \oplus b = \sum_{i=0}^{\infty} a_i \oplus b_i$ , 其中  $a_i, b_i$  是数  $a, b$  的二进制表达的第  $i$  位
9          不同位置上运算互不影响
10         7. 若  $a$  为偶数, 则  $a \oplus (a + 1) = 1, a \oplus 1 = (a + 1), (a + 1) \oplus 1 = a$ 
11 */

```

4.19 博弈论 Game Theory

```

1  /// 博弈论 Game Theory
2  /*
3  SG-组合游戏
4  SG-组合游戏定义:  游戏两人参与, 轮流决策, 最优决策;
5                      当有人无法决策时, 游戏结束, 无法决策的人输;
6                      游戏可在有限步内结束, 不会多次抵达同一状态, 没有平局;
7                      游戏某一状态可以到达的状态集合存在且确定.
8  将游戏的每一个状态看做结点, 状态间的转移看做是有向边, 则SG-组合游戏是一个无环图.
9  必胜态和必败态:  必胜态(N-position): 当前玩家有策略使得对手无论做什么操作, 都能保证自己胜利.
10                  必败态(P-position): 对手的必胜态.
11                  组合游戏中某一状态不是必胜态就是必败态.
12                  对任意的必胜态, 总存在一种方式转移到必败态.
13                  对任意的必败态, 只能转移到必胜态.

```

```

14 找出必败态和必胜态：1. 按照规则，终止状态设为必败(胜)态。
15      2. 将所有能到达必败态的状态标为必胜态。
16      3. 将只能到达必胜态的状态标为必败态。
17      4. 重复2-3，直到不再产生必败(胜)态。
18 游戏的和：考虑任意多个同时进行的SG-组合游戏，这些SG-组合游戏的和是这样一个SG-组合游戏，
    在它进行的过程中，游戏者可以任意挑选其中的一个单一游戏进行决策，最终，没有办法进行决策的人输。
19 *//*
20 SG函数(the Sprague-Grundy function)
21 定义：游戏状态为 $x$ ， $sg(x)$ 表示状态 $x$ 的sg函数值， $sg(x) = \min\{n | n \in N \cap n \notin F(x)\}$ ，
     $F(x)$ 表示 $x$ 能够达到的所有状态。一个状态为必败态则 $sg(x)=0$ 。
22 SG定理：如果游戏 $G$ 由 $n$ 个子游戏组成， $G = G_1 + G_2 + G_3 + \dots + G_n$ ，并且第 $i$ 个子游戏sg函数值为 $sg_i$ ，
    则游戏 $G$ 的sg函数值为 $g = sg_1 \oplus sg_2 \oplus \dots \oplus sg_n$ 
23 SG函数与组合游戏： $sg(x)$ 为 $0$ ，则状态 $x$ 为必败态；否则 $x$ 为必胜态。
24 *//*
25 Nim游戏：两名游戏者从 $N$ 堆石子中轮流取石子，每次任选一堆石子从中取走至少1个石子，
    取走最后一个石子的人胜利。
26  $SG(i) = X(i = 1, 2, \dots, N)$ ，当且仅当 $SG(tot) = SG(1) \oplus SG(2) \oplus \dots \oplus SG(N)$ 
27 定理：在我们每次只能进行一步操作的情况下，对于任何的游戏的和，
    我们若将其中的任一单一SG-组合游戏换成数目为它的SG值的一堆石子，该单一
    SG-组合游戏的规则变成取石子游戏的规则（可以任意取，甚至取完），则游戏的和的胜负情况不变。这样，
    所有的游戏的和都等价成nim游戏。
28
29 anti-nim游戏：游戏规则和nim游戏一样，除了最后一个取走石子的人输。
30 SJ定理：对于任意一个Anti-SG游戏，如果我们规定当局面中所有的单一游戏的SG值为0时，游戏结束，
    则先手必胜当且仅当：(1) 游戏的SG函数不为0且游戏中某个单一游戏的SG函数大于1；(2) 游戏的
    SG函数为0且游戏中没有单一游戏的SG函数大于1。
31
32 Every-SG游戏：对任何没有结束的单一游戏，决策者都必须对该游戏进行一步决策，其他规则与普通SG游戏一样。
33 定义step函数：

$$step(v) = \begin{cases} 0 & v \text{ 为终止状态} \\ \max((step(u)) + 1) & SG(v) > 0 \cap u \text{ 为 } v \text{ 的后继状态} \cap SG(u) = 0 \\ \min((step(u)) + 1) & SG(v) = 0 \cap u \text{ 为 } v \text{ 的后继状态} \end{cases}$$

34 定理：对于Every-SG游戏先手必胜当且仅当单一游戏中最大的step为奇数。
35
36 翻硬币游戏： $N$ 枚硬币排成一排，有的正面朝上，有的反面朝上。我们从左开始对硬币按1到 $N$ 编号。
    游戏者根据某些约束翻硬币(如：每次只能翻一或两枚，或者每次只能翻连续的几枚)，但他所翻动的硬币中，
    最右边的必须是从正面翻到反面。谁不能翻谁输。
37 结论：局面的SG值为局面中每个正面朝上的棋子单一存在时的SG值的异或和。
38
39 树的删边游戏 给出一个有 $N$ 个点的树，有一个点作为树的根节点。游戏者轮流从树中删去边，删去一条边后，
    不与根节点相连的部分将被移走。谁无路可走谁输。
40 定理：叶子节点的SG值为0；中间节点的SG值为它的所有子节点的SG值加1后的异或和。
41
42 无向图的删边游戏：一个无相联通图，有一个点作为图的根。游戏者轮流从图中删去边，删去一条边后，
    不与根节点相连的部分将被移走。谁无路可走谁输。
43 著名的定理——Fusion Principle：我们可以对无向图做如下改动：将图中的任意一个偶环缩成一个新点，
    任意一个奇环缩成一个新点加一个新边；所有连到原先环上的边全部改为与新点相连。
    这样的改动不会影响图的SG值。
44
45 Crazy Nim: (Gym 100338D)
46 有三堆石子，分别有 $a, b, c$ 个石子( $a \neq b, a \neq c, b \neq c, 0 < a, b, c \leq 10^9$ )，
    两人轮流从任意一堆石子中拿任意数量的石子，要求如果拿完后还有石子，
    则石子数不能与另外两堆的数量相同。谁不能拿谁输。
47 后手必赢当且仅当 $(a+1) \oplus (b+1) \oplus (c+1)$ 
48 */

```

4.20 快速傅里叶变换和数论变换 (FFT 和 NTT)

```

1 ///快速傅里叶变换FFT(Fast Fourier Transformation)
2 /*原理：

```


3 | **DFT**(离散傅里叶变换), 变换公式:

$$\begin{cases} X(k) = \sum_{i=0}^{N-1} x(i)W_N^{ik} & k = 0, 1, \dots, N-1 \\ W_N = e^{-j\frac{2\pi}{N}} \end{cases}$$

4 | W_N 被称为旋转因子, 有如下性质:

- 5 | 1. 对称性: $(W_N^{ik})^* = W_N^{-ik}$
- 6 | 2. 周期性: $W_N^{ik} = W_N^{(i+N)k} = W_N^{i(N+k)}$
- 7 | 3. 可约性: $W_N^{ik} = W_{mN}^{mik}, W_N^{ik} = W_{\frac{N}{m}}^{\frac{ik}{m}}$

8 | 所以: $W_N^{i(N-k)} = W_N^{(N-i)k} = W_N^{-ik}, W_N^{N/2} = -1, W_N^{k+N/2} = -W_N^k$

9 | **IDFT**(DFT逆变换), 变换公式:

$$x(k) = \frac{1}{N} \sum_{i=0}^{N-1} X(i)W_N^{-ik}$$

10 | (基2)FFT推导:

$$\begin{aligned} X(k) &= \sum_{i=0}^{N-1} x(i)W_N^{ik} \\ &= \sum_{r=0}^{N/2-1} x(2r)W_N^{2rk} + \sum_{r=0}^{N/2-1} x(2r+1)W_N^{(2r+1)k} \\ &= \sum_{r=0}^{N/2-1} x(2r)W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1)W_{N/2}^{rk} \\ &= \begin{cases} \sum_{r=0}^{N/2-1} x(2r)W_{N/2}^{rk} + W_N^k \sum_{r=0}^{N/2-1} x(2r+1)W_{N/2}^{rk} & k < N/2 \\ \sum_{r=0}^{N/2-1} x(2r)W_{N/2}^{rk'} - W_N^k \sum_{r=0}^{N/2-1} x(2r+1)W_{N/2}^{rk'} & k \geq N/2, k' = k - N/2 \end{cases} \end{aligned}$$

11 | 所以通过计算

$$X_1(k) = \sum_{r=0}^{N/2-1} x(2r)W_{N/2}^{rk}, X_2(k) = \sum_{r=0}^{N/2-1} x(2r+1)W_{N/2}^{rk}, k < N/2$$

12 | 可以计算得

$$\begin{cases} X(k) &= X_1(k) + W_N^k X_2(k) \\ X(k + N/2) &= X_1(k) - W_N^k X_2(k) \end{cases} \quad k < N/2$$

13 | **DFT**变换满足cyclic convolution的性质, 即

14 | 定义循环卷积 $c=a(*)b$:

$$c_r = \sum_{x+y=r(\%N)} a_x b_y$$

15 | 则有: $DFT(a(*)b) = DFT(a) \cdot DFT(b)$, 所以 $a(*)b = DFT^{-1}(DFT(a) \cdot DFT(b))$

16 | 注意: **FFT**是cyclic的, 需要保证高位有足够多的0

17 | **FFT**算法限制, n 必须是2的幂

18 | **FFT**是定义在复数上的, 因此与整数变换可能有精度误差

19 | */

20 | //FFT常被用来为多项式乘法加速, 即在 $O(n \log n)$ 复杂度内完成多项式乘法

21 | //也需要用FFT算法来解决需要构造多项式相乘来进行计数的问题

22 | `//#include <complex>`

23 | `//typedef std::complex<double> Complex;`

24 | `struct Complex//复数类, 可以直接用STL库中的complex<double>`

25 | {

26 | `double r, i;`

27 | `Complex(double _r = 0.0, double _i = 0.0) {r = _r, i = _i;}`

28 | `Complex operator + (const Complex &b) const {return Complex(r + b.r, i + b.i);}`

29 | `Complex operator - (const Complex &b) const {return Complex(r - b.r, i - b.i);}`

30 | `Complex operator * (const Complex &b) const`

31 | {

32 | `return Complex(r * b.r - i * b.i, r * b.i + i * b.r);`

33 | }

34 | `double real() {return r;}`

35 | `double image() {return i;}`

36 | };

37 | `void brc(vector<Complex> &p, int N)//蝶形变换, 交换位置i与逆序i, 如N=2^3, 交换p[011=3]与p[110=6]`

38 | {

39 | `int i, j, k;`

40 | `for(i = 1, j = N >> 1; i < N - 2; i++)`

41 | {

```

42     if(i < j) swap(p[i], p[j]);
43     for(k = N >> 1; j >= k; k >>= 1) j -= k;
44     if(j < k) j += k;
45 }
46 }
47 void FFT(vector<Complex> &p, int N, int op)//op = 1表示DFT傅里叶变换, op=-1表示傅里叶逆变换
48 {
49     brc(p, N);
50     double p0 = PI * op;
51     for(int h = 2; h <= N; h <<= 1, p0 *= 0.5)
52     {
53         int hf = h >> 1;
54         Complex unit(cos(p0), sin(p0));
55         for(int i = 0; i < N; i += h)
56         {
57             Complex w(1.0, 0.0);
58             for(int j = i; j < i + hf; j++)
59             {
60                 Complex u = p[j], t = w * p[j + hf];
61                 p[j] = u + t;
62                 p[j + hf] = u - t;
63                 w = w * unit;
64             }
65         }
66     }
67 }
68 //Polynomial multiplication多项式相乘  $\vec{X} \times \vec{Y} = \vec{Z}$ 
69 void polynomial_multi(const vector<int> &a, const vector<int> &b, vector<int> &res, int n)
70 {
71     int N = 1;
72     int i = 0;
73     while(N < n + n) N <<= 1;//FFT的项数必须是2的幂
74     vector<Complex> A(N, Complex(0.0)), B(N, Complex(0.0)), D(N);
75     for(i = 0; i < (int)a.size(); i++) A[i] = Complex(a[i], 0.0);
76     for(i = 0; i < (int)b.size(); i++) B[i] = Complex(b[i], 0.0);
77     FFT(A, N, 1);
78     FFT(B, N, 1);
79     for(i = 0; i < N; i++) D[i] = A[i] * B[i];
80     FFT(D, N, -1);
81     for(i = 0, res.clear(); i < N; i++) res.pb(round(D[i].real() / N));
82 }
83
84 /*
85 应用1: 给一个01串S, 求有多少对(i, j, k)(i < j < k)使Si = Sj = Sk = 1, 且j - i = k - j
86 */
87
88
89 //数论变换(Number Theory Transformation, NTT)
90 /*NTT的推导:
91 0. DFT变换公式:  $A(k) = \sum_{i=0}^{N-1} a(i)\varpi^{ik}$ 
92 IDFT变换公式:  $a(k) = N^{-1} \sum_{i=0}^{N-1} A(i)\varpi^{-ik}$ 
93 1. 周期性: 由于

```

$$\begin{aligned}
 & \Rightarrow \begin{aligned} & A(k) \cdot B(k) = C(k) \\ & \left[\sum_{i=0}^{N-1} a(i)\varpi^{ik} \right] \cdot \left[\sum_{j=0}^{N-1} b(j)\varpi^{jk} \right] = \sum_{i=0}^{N-1} c(i)\varpi^{ik} \end{aligned} \\
 & \Rightarrow \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a(i)b(j)\varpi^{(i+j)k} = \sum_{i=0}^{N-1} \left(\sum_{x+y=i(\%N)} a(x)b(y) \right) \varpi^{ik} \\
 & \Rightarrow \sum_{j=0}^i a(i)b(i-j)\varpi^{ik} + \sum_{j=i+1}^{N-1} a(i)b(N+i-j)\varpi^{(i+N)k} = \sum_{x+y=i(\%N)} a(x)b(y)\varpi^{ik}, \quad (\forall i \in [0, N-1]) \\
 & \Rightarrow \forall i \in [0, N-1], k \in [0, N-1], \varpi^{(i+N)k} = \varpi^{ik} \\
 & \Rightarrow \varpi \text{ 具有周期为 } N \text{ 的周期性, 即 } \varpi^N = 1
 \end{aligned}$$

94 | 2. 求和引理：若要实现逆变换，则有：

$$\begin{aligned} A(k) &= \sum_{i=0}^{N-1} a(i) \varpi^{ik} \\ a(k) &= N^{-1} \sum_{i=0}^{N-1} \left[\sum_{j=0}^{N-1} a(j) \varpi^{ij} \right] \varpi^{-ik} \\ &= N^{-1} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} a(j) \varpi^{i(j-k)} \\ &= N^{-1} \sum_{i=0}^{N-1} a(k) + N^{-1} \sum_{j \neq k} a(j) \left[\sum_{i=0}^{N-1} (\varpi^{j-k})^i \right] \end{aligned}$$

95 | 所以，有 $\forall i \neq 0, \sum_{j=0}^{N-1} (\varpi^i)^j = \frac{(\varpi^i)^N - 1}{(\varpi^i) - 1} = \frac{(\varpi^N)^i - 1}{(\varpi^i) - 1} = 0$

96 | 即 $\varpi^N = 1$ ，且 $\varpi^i \neq 1 (i \neq 0)$

97 | 3. FFT的分治计算：

98 | 由 $N = \prod_{i=1}^m p_i^{k_i}$ ，令 $n = p_i (i = 1, 2, \dots, m)$ ， $N' = \frac{N}{n}$

99 | 则

$$\begin{aligned} &A(k + pN') \\ &= \sum_{i=0}^{N-1} a(i) \varpi^{i(k+pN')} \\ &= \sum_{i=0}^{n-1} \left[\sum_{j=0}^{N'-1} a(i + jn) \varpi^{(i+jn)(k+pN')} \right] \\ &= \sum_{i=0}^{n-1} \left[\sum_{j=0}^{N'-1} a(i + jn) \varpi^{ik+jnk+ipN'+jpN} \right] \\ &= \sum_{i=0}^{n-1} [\varpi^{i(k+pN')} \sum_{j=0}^{N'-1} a(i + jn) \varpi^{jn k}] \end{aligned}$$

100 | 其中， $0 \leq k < N, 0 \leq k + pN' < N$

101 | 现将规模为 N 的问题分解为 n 个规模为 N' 的子问题，如此分治，有： $T(N) = nT(\frac{N}{n}) + O(Nn)$ ，其中 $n \mid N$

102 | 于是，总的时间复杂度为： $T(N) = O(N \cdot \sum_{i=1}^m (p_i k_i))$ ，若 $N = 2^m, T(N) = O(N \log N)$

103 | 总结一下，现在对某一整数 N ，如果要进行再整数域上的FFT，必须满足存在旋转因子 ϖ ，使

$$\varpi^i \begin{cases} \neq 1, & i = 1, 2, \dots, N-1 \\ = 1, & i = N \end{cases}$$

104 | 要在整数域了满足上述条件的，可以是关于素数 p 的取模运算，若 a 是在模 p 意义下的原根，旋转因子为 $a^{\frac{p-1}{N}}$ ，
要求 N 整除 $p-1$ ，最后的结果为对 p 取模后的答案（如果要求准确答案，
需要满足 $p > \max\{a(i), b(i), c(i)\} (i = 0, 1, 2, \dots, N-1)$ ，或者对不同素数 p 进行多次计算，
然后用中国剩余定理求解）

105 | 适合 $p = 998244353 = 119 \times 2^{23} + 1 (2^{23} > 8.3e6)$ ，3是 p 的原根

106 | $p = 985661441$ ，3是 p 的原根， $(p-1) = 2^{20} * i + 1$

107 | 适合的 p 有很多，枚举 i ，判断 $(1 < i) * i + 1$ 是否为素数即可

108 | */

109 | //来源：2015多校第三场，1007标程

110 | LL qpow(LL x, LL k, LL mod);

111 | const LL mod = 998244353, wroot = 3;

112 | int wi[NUM << 2];

113 | int ntt_init(int n)

114 | {

115 | int N = 1;

116 | while(N < n + n) N <<= 1;

117 | wi[0] = 1, wi[1] = qpow(wroot, (mod - 1) / N, mod);

118 | for(int i = 2; i < N; i++)

119 | wi[i] = 1LL * wi[i - 1] * wi[1] % mod;

120 | return N;

121 | }

122 |

123 | void brc(vector<int> &p, int N) //蝶形变换，交换位置 i 与逆序 i ，如 $N=2^3$ ，交换 $p[011=3]$ 与 $p[110=6]$

124 | {

125 | int i, j, k;

126 | for(i = 1, j = N >> 1; i < N - 2; i++)

127 | {

128 | if(i < j) swap(p[i], p[j]);

```

129     for(k = N >> 1; j >= k; k >>= 1) j -= k;
130     if(j < k) j += k;
131 }
132 }
133 void NTT(vector<int> &a, int N, int op)
134 {
135     brc(a, N);
136     for(int h = 2; h <= N; h <= 1)
137     {
138         int unit = op == -1 ? N - N / h : N / h;
139         int hf = h >> 1;
140         for(int i = 0; i < N; i += h)
141         {
142             int w = 0;
143             for(int j = i; j < i + hf; j++)
144             {
145                 int u = a[j], t = 1LL * wi[w] * a[j + hf] % mod;
146                 if((a[j] = u + t) >= mod) a[j] -= mod;
147                 if((a[j + hf] = u - t) < 0) a[j + hf] += mod;
148                 if((w += unit) >= N) w -= N;
149             }
150         }
151     }
152     if(op == -1)
153     {
154         int inv = qpow(N, mod - 2, mod);
155         for(int i = 0; i < N; i++) a[i] = 1LL * a[i] * inv % mod;
156     }
157 }

```

4.21 莫比乌斯反演 Mobius

```

1 // 莫比乌斯反演 Mobius
2 // mobius 函数
3 /*

```

$$\mu(x) = \begin{cases} 1 & x = 1 \\ (-1)^r & x = p_1 \cdot p_2 \cdots p_r, \text{ 其中 } p_i (i = 1, 2, \cdots, r) \text{ 是素数} \\ 0 & \text{其他} \end{cases}$$

```

4 */
5 // 莫比乌斯反演
6 //

```

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

```

7 //

```

$$\sum_{d|n} \mu(d) = \begin{cases} 1 & n = 1 \\ 0 & n \neq 1 \end{cases}$$

```

8 // \sum_{d|n} \phi(d) = n, \phi(d) 为欧拉函数

```

```

9 // \phi(n) = n \sum_{d|n} \mu(d)/d

```

```

10

```

```

11 // 使用 1

```

```

12 /*

```

```

13     \sum_{i=1}^a \sum_{j=1}^b gcd(i, j) == D (1 \le i \le a, 1 \le j \le b, a \le b), 即求 gcd(i, j) 等于 d 的对数, [x] 表示下取整

```

```

14     \sum_{i=1}^a \sum_{j=1}^b gcd(i, j) == D

```

```

15     \Rightarrow \sum_{i=1}^{\lfloor \frac{a}{D} \rfloor} \sum_{j=1}^{\lfloor \frac{b}{D} \rfloor} gcd(i, j) == 1

```

```

16     \Rightarrow \sum_{i=1}^{\lfloor \frac{a}{D} \rfloor} \sum_{d|gcd(i, j)} \mu(d), 使用 mobius 函数和的性质替换 gcd(i, j) == 1

```

```

17 |      $\Rightarrow \sum_{d=1}^{\lfloor \frac{a}{D} \rfloor} \mu(d) \lfloor \frac{\lfloor \frac{a}{D} \rfloor}{d} \rfloor \cdot \lfloor \frac{\lfloor \frac{b}{D} \rfloor}{d} \rfloor, d | \gcd(i, j) \Leftrightarrow d | i \cup d | j$ 
18 |      $D == 1, \sum_{d=1}^a \mu(d) \cdot \lfloor \frac{a}{d} \rfloor \cdot \lfloor \frac{b}{d} \rfloor$ 
19 | */
20 |
21 | // 使用2
22 | /*
23 |      $\sum_{i=1}^a \sum_{j=1}^b \gcd(i, j), a \leq b$ 
24 |      $\Rightarrow \sum_{d=1}^a \sum_{i=1}^{\lfloor \frac{a}{d} \rfloor} \sum_{j=1}^{\lfloor \frac{b}{d} \rfloor} d \cdot (\gcd(i, j) == 1)$ 
25 |      $\Rightarrow \sum_{d=1}^a \sum_{d'=1}^{\lfloor \frac{a}{d} \rfloor} d \cdot \mu(d') \cdot \lfloor \frac{a}{dd'} \rfloor \cdot \lfloor \frac{b}{dd'} \rfloor$ , 使用1
26 |      $\Rightarrow \sum_{d=1}^a \sum_{d|D} d \cdot \mu(\frac{D}{d}) \cdot \lfloor \frac{a}{D} \rfloor \cdot \lfloor \frac{b}{D} \rfloor, D = dd'$ 
27 |      $\Rightarrow \sum_{D=1}^a \lfloor \frac{a}{D} \rfloor \cdot \lfloor \frac{b}{D} \rfloor \cdot (id \cdot \mu)(D)$ 
28 |      $\Rightarrow \sum_{D=1}^a \lfloor \frac{a}{D} \rfloor \cdot \lfloor \frac{b}{D} \rfloor \cdot \phi(D), id \cdot \mu = \phi$ 
29 | */
30 |
31 | /// 积性函数
32 | // 定义在正整数集上的函数  $f(n)$  (称为算术函数), 若  $\gcd(a, b) = 1$  时有  $f(a) \cdot f(b) = f(a \cdot b)$ , 则称  $f(x)$  为积性函数。
33 | // 一个显然的性质: (非恒等于零的) 积性函数  $f(n)$  必然满足  $f(1) = 1$ 。
34 | // 定义逐点加法  $(f + g)(n) = f(n) + g(n), f(x \cdot g) = f(x) \cdot g(x)$ 。
35 | // 一个比较显然的性质: 若  $f, g$  均为积性函数, 则  $f \cdot g$  也是积性函数。
36 | // 积性函数的求值:  $n = \prod p_i^{a_i}$  则  $f(n) = \prod f(p_i^{a_i})$ , 所以只要解决  $n = p^a$  时  $f(n)$  的值即可。
37 |
38 | // 常见积性函数有:
39 | // 恒为1的常函数  $1(n) = 1$ ,
40 | // 恒等函数  $id(n) = n$ ,
41 | // 单位函数  $\varepsilon(n) = (n == 1)$ , (这三个都是显然为积性)
42 | // 欧拉函数  $\phi(n)$  (只要证两个集合相等就能证明积性)
43 | // 莫比乌斯函数  $\mu(n)$  (由定义也是显然的)
44 | // 因子个数函数  $\tau$ 
45 | // 因子和函数  $\sigma$ 
46 | //  $\mu \cdot id = \phi$ 
47 | void pre_mobius()
48 | {
49 |     mu[1] = 1;
50 |     for(int i = 2; i < NUM; i++)
51 |         if(!mu[i])
52 |         {
53 |             mu[i] = -1;
54 |             for(int j = i + i; j < NUM; j += i)
55 |                 if((j / i) % i == 0)
56 |                     mu[j] = 2;
57 |             else
58 |             {
59 |                 if(mu[j] == 0) mu[j] = -1;
60 |                 else mu[j] = -mu[j];
61 |             }
62 |         }
63 |     else if(mu[i] == 2 || mu[i] == -2) mu[i] = 0;
64 | }

```

4.22 一些数学知识

```

1 | //1. 格雷码: (相邻码之间二进制只有一位不同), 构造方法:  $a_i = i \text{ xor } (i \gg 1)$  ( $a_i$  为求第  $i$  个格雷码)
2 | /*2. 多边形数: 可以排成正多边形的整数

```

```

3      第  $n$  个  $S$  边形数的公式是:  $\frac{n[(s-2)n-(s-4)]}{2}$ 
4      费马多边形定理: 每一个正整数最多可以表示成  $n$  个  $n$ -边形数之和
5  */
6  //3. 四平方和定理: 每个正整数均可表示为4个整数的平方和。它是费马多边形数定理和华林问题的特例.
7  //4. 即对任意奇素数  $p$ , 同余方程  $x^2 + y^2 + 1 \equiv 0 \pmod{p}$  必有一组整数解  $x, y$  满足  $0 \leq x < \frac{p}{2}, 0 \leq y < \frac{p}{2}$ 
8  /// 勾股数
9  /*
10  勾股数: 对正整数  $a, b, c$ , 如果有  $a^2 + b^2 = c^2$ , 称  $(a, b, c)$  为勾股数
11  性质:  $a, b$  中一个为奇数, 一个为偶数,  $c$  一定为奇数.
12  本原勾股数: 满足  $\gcd(a, b, c) = 1$  的勾股数
13  本原勾股数定理: 如果对奇数  $s > t \geq 1$ , 且  $\gcd(s, t) = 1$ , 则有:  $a = s \times t, b = \frac{s^2 - t^2}{2}, c = \frac{s^2 + t^2}{2}$ ,  $(a, b, c)$  是本原勾股数
14  */
15  /* 数组映射循环同构
16      对于一个数组, 我们假设不关心它的每个数值得大小, 只关心它们是否相同, 而且它是循环的,
      即从任意位置都可以看做是数组的起点. You are given a set of  $N$  vectors, each vector consists of  $K$ 
      integers. Vector  $X$  is equivalent to  $Y$  (denoted  $X \equiv Y$ ). if there exist a bijection  $f: Z \rightarrow Z$ 
      and an integer  $r$ , such that  $X[i] = f(Y[(i + r) \bmod K])$  for each  $i$  in the range  $[0..K - 1]$ . For
      example,  $(1, 2, 2, 3) \equiv (22, 3, 4, 22)$ , with  $r = 2$  and  $f(22) = 2, f(3) = 3$  and  $f(4) = 1$ . But  $(22, 3,$ 
       $22, 4)$  is not equivalent to  $(1, 2, 2, 3)$ .
17      我们将对于每个位置, 我们找出下一个相同数字的位置, 用它们的位置差表示该数组, 然后用最小表示法,
      即可唯一的表示出该等价关系.
18      例: 对  $(22, 3, 4, 22)$ , 用位置差代替该数字, 得:  $(3, 4, 4, 1)$ , 最小表示法得:  $(1, 3, 4,$ 
       $4)$  这样所有等价的数组有且仅有这么一种表示方法.
19  */

```

5 线性代数 Linear Algebra

5.1 矩阵 Matrix

```

1  /// 矩阵 Matrix
2  /* 定义:
3      矩阵:  $m$  行  $n$  列.
4       $n$  阶方阵:  $n$  行  $n$  列的矩阵.
5      行矩阵:  $1 \times n$  的矩阵.
6      列矩阵:  $n \times 1$  的矩阵.
7      零矩阵  $O_{m \times n}$ : 全为  $0$  的矩阵.
8      单位矩阵  $I_n$ : 对角线全为  $1$  的  $n$  阶方阵.
9      同型矩阵  $A, B$ :  $A, B$  的行数和列数都相同.
10  */
11  /* 操作和性质:
12  线性运算:
13      两矩阵  $A, B$  相等:  $A, B$  是同型矩阵, 且对应元相等.
14      矩阵的加法和减法: 当  $A, B$  是同型矩阵时可加减, 结果是对应元相加减的同型矩阵.
15       $A$  的负矩阵  $-A$ :  $A$  的每个元取反.
16      矩阵的数乘  $kA$ :  $A$  的每个元乘以  $k$ .
17  线性运算的性质:
18      1. (交换律)  $A + B = B + A$ 
19      2. (结合律)  $(A + B) + C = A + (B + C)$ 
20      3.  $A + O = A, A + (-A) = O$ 
21      4.  $1A = A$ 
22      5.  $k(lA) = (kl)A$ 
23      6.  $k(A + B) = kA + kB$ 
24      7.  $(k + l)A = kA + lA$ 
25  矩阵的乘法:
26       $m \times p$  矩阵  $A = (a_{ij})_{m \times p}$ ,  $p \times n$  矩阵  $B = (b_{ij})_{p \times n}$  的乘积为  $m \times n$  矩阵  $C = (c_{ij})_{m \times n}$ , 其中

```

$$c_{ij} = \sum_{k=1}^p a_{ik} b_{kj}$$

```

27      .
      要求  $A$  的列数等于  $B$  的行数时才能相乘

```

28 矩阵的乘法的性质：

29 1. (结合律) $(AB)C = A(BC)$

30 2. (数乘结合律) $k(AB) = (kA)B = A(kB)$

31 3. (分配率) $A(B + C) = AB + AC$, $(B + C)A = BA + CA$

32 4. 矩阵的乘法一般不满足交换律，当 $AB = BA$ 时，称 A 与 B 可交换。

33 5. 不满足消去率，即由 $AB - AC = A(B - C) = 0$ 不能推出 $B - C = 0$

34 6. $I_m A_{m \times n} = A_{m \times n} I_m = A_{m \times n}$

35 7. n 阶单位矩阵与任意 n 阶矩阵 A 是可交换的，即 $IA = AI = A$ 。

36 方阵的幂：

37 设 A 是 n 阶方阵， k 是正整数，定义：

$$\begin{cases} A^0 = I \\ A^1 = A \\ A^{k+1} = A^k A, \quad k = 1, 2, \dots \end{cases}$$

38 方阵的幂的性质：

39 1. $A^m A^k = A^{m+k}$

40 2. $(A^m)^k = A^{mk}$

41 3. 一般 $(AB)^k \neq A^k B^k$ ，当 $AB = BA$ 时， $(AB)^k = A^k B^k = B^k A^k$ ，其逆不真。

42 矩阵的转置：

43 将矩阵 A 的行列互换，所得的矩阵称为 A 的转置，即为 A^T 。

44 $m \times n$ 矩阵的转置是 $n \times m$ 矩阵

45 对称矩阵 A : $A^T = A$

46 反对称矩阵 A : $A^T = -A$

47 矩阵的转置的性质：

48 1. $(A^T)^T = A$

49 2. $(A + B)^T = A^T + B^T$

50 3. $(kA)^T = kA^T$, k 为数

51 4. $(AB)^T = B^T A^T$

52 分块矩阵：

53 将 $m \times n$ 矩阵分为 $r \times c$ 个子矩阵，分块矩阵的乘法相当于将把各个子矩阵当作数是的乘法。

```
54 */
55 const int Matrix_N = 55, Matrix_M = 55;
56 struct Matrix
57 {
58     int a[Matrix_N][Matrix_M];
59     int n, int m;
60     Matrix(int _n = 0, int _m = 0, bool I = false) {init(_n, _m, I);}
61     void init(int _n = 0, int _m = 0, bool I = false)
62     {
63         memset(a, 0, sizeof(a));
64         n = _n;
65         m = _m;
66         if(I) for(int i = 1; i <= n; ++i) a[i][i] = 1;
67     }
68     //C = A + B
69     Matrix operator + (const Matrix &B) const
70     {
71         Matrix res(n, m);
72         for(int i = 1; i <= n; ++i)
73             for(int j = 1; j < m; ++j)
74                 res.a[i][j] = a[i][j] + B.a[i][j];
75         return Matrix;
76     }
77     //C = A - B
78     Matrix operator - (const Matrix &B) const
79     {
80         Matrix res(n, m);
81         for(int i = 1; i <= n; ++i)
82             for(int j = 1; j < m; ++j)
83                 res.a[i][j] = a[i][j] - B.a[i][j];
84         return Matrix;
85     }
86     //A += B
87     Matrix operator += (const Matrix &B) const
88     {
```

```

89     for(int i = 1; i <= n; ++i)
90         for(int j = 1; j < m; ++j)
91             a[i][j] += B.a[i][j];
92     return *this;
93 }
94 //C = AB
95 Matrix operator * (const Matrix & B) const
96 {
97     Matrix res(n, B.m);
98     int i, j, k;
99     for(i = 1; i <= n; ++i)
100         for(j = 1, res.a[i][j] = 0; j <= B.m; ++j)
101             for(k = 1; k <= m; ++k)
102                 res.a[i][j] = a[i][k] * B.a[k][j];
103 //             if((res.a[i][j] += a[i][k] * B.a[k][j] % mod) >= mod)
104 //                 res.a[i][j] -= mod;
105     return res;
106 }
107 //方阵的k次幂
108 Matrix operator ^(int k)
109 {
110     Matrix x = *this;
111     Matrix res(x.n, x.n, true);
112     while(k)
113     {
114         if(k & 1) res = res * x;
115         x = x * x;
116         k >>= 1;
117     }
118     return res;
119 }
120 void out()
121 {
122     printf("N = %d, M = %d\n", n, m);
123     for(int i = 1; i <= n; ++i)
124         for(int j = 1; j <= m; ++j)
125             printf("%d%c", a[i][j], j == n ? '\n' : ' ');
126 }
127 };
128
129 /*应用:
130 1. A是1×n的行矩阵, B是n阶方阵, 则

```

$$C_{1 \times n} = ABB \cdots B = A(B^m)$$

```

131 2. 设A是n×1的列矩阵, B是n阶方阵, 求  $\sum_{k=0}^{m-1} AB^k$ 

```

```

132 做法: 令C为分块矩阵

```

$$C^m = \begin{bmatrix} B & A \\ O & I \end{bmatrix}^m = \begin{bmatrix} B^m & \sum_{k=1}^{m-1} AB^k \\ O & I \end{bmatrix}$$

```

133 */

```

5.2 矩阵的初等变换和矩阵的逆

```

1 ///矩阵的初等变换和矩阵的逆
2 const int Matrix_N = 1010, Matrix_M = 1010;
3 //矩阵类 适用与求矩阵的逆与高斯消元等场合
4 //行的初等变换
5 typedef vector<double> VD;
6 VD operator * (const VD &a, const double b)

```



```

7 {
8     int _n = a.size();
9     VD c(_n);
10    for(int i = 0; i < _n; i++)
11        c[i] = a[i] * b;
12    return c;
13 }
14 VD operator - (const VD &a, const VD &b)
15 {
16     int _n = a.size();
17     VD c(_n);
18     for(int i = 0; i < _n; i++)
19         c[i] = a[i] - b[i];
20     return c;
21 }
22 VD operator + (const VD &a, const VD &b)
23 {
24     int _n = a.size();
25     VD c(_n);
26     for(int i = 0; i < _n; i++)
27         c[i] = a[i] + b[i];
28     return c;
29 }
30 struct Matrix
31 {
32     int n, m;
33     VD *a;
34     void Matrix(int _n = Matrix_N, int _m = Matrix_M)
35     {
36         n = _n, m = _m;
37         a = new VD[n];
38         for(int i = 0; i < n; i++)
39             a[i].resize(m, 0);
40     }
41     void ~Matrix()
42     {
43         delete []a;
44     }
45     void clear()//0矩阵
46     {
47         for(int i = 0; i < n; i++)
48             a[i].assign(0);
49     }
50     void I()//单位矩阵
51     {
52         clear();
53         for(int i = 0; i < n; i++)
54             a[i][i] = 1;
55     }
56     //矩阵加法, 同上
57     Matrix operator + (const Matrix &b) const
58     {
59         Matrix c(n, m);
60         for(int i = 0; i < n; i++)
61             c.a[i] = a[i] + b.a[i];
62         return c;
63     }
64     Matrix operator - (const Matrix &b) const //矩阵减法
65     {
66         Matrix c(n, m);
67         for(int i = 0; i < n; i++)
68             c.a[i] = a[i] - b.a[i];
69         return c;
70     }

```

```

71 Matrix operator * (const Matrix &b) const // 矩阵乘
72 {
73     Matrix c(n, b.m);
74     for(int i = 0; i < n; i++)
75         for(int j = 0; j < b.m; j++)
76             {
77                 c[i][j] = 0;
78                 for(int k = 0; k < m; k++)
79                     c[i][j] += a[i][k] * b.a[k][j];
80             }
81     return c;
82 }
83
84 // 实现求矩阵的逆  $O(n^3)$ 
85 // 将原矩阵  $A$  和一个单位矩阵  $I$  做一个大矩阵  $(A, I)$ , 用行的初等变换将大矩阵中的  $A$  变为  $I$ ,
86 // 将会得到  $(I, A^{-1})$  的形式
87 // 注意:
88 Matrix inverse()
89 {
90     Matrix c;
91     c.I();
92     for(int i = 0; i < n; i++)
93     {
94         for(int j = i; j < n; j++)
95             if(fabs(a[j][i]) > 0)
96             {
97                 swap(a[i], a[j]);
98                 swap(c[i], c[j]);
99                 break;
100             }
101         c[i] = c[i] * (1.0 / a[i][i]);
102         a[i] = a[i] * (1.0 / a[i][i]);
103         for(int j = 0; j < n; j++)
104             if(j != i && fabs(a[j][i]) > 0)
105             {
106                 c[j] = c[j] - a[i] * a[j][i];
107                 a[j] = c[j] - a[i] * a[j][i];
108             }
109     }
110 };
111 // Gauss 消元
112 int Gauss(double a[][MAXN], bool l[], double ans[], int n)
113 { // l, ans 储存解, l[] 表示是否是自由元
114     int res = 0, r = 0;
115     for(int i = 0; i < n; i++) l[i] = false;
116     for(int i = 0; i < n; i++)
117     {
118         for(int j = r; j < n; j++)
119             if(fabs(a[j][i]) > EPS)
120             {
121                 for(int k = i; k <= n; k++)
122                     swap(a[j][k], a[r][k]);
123                 break;
124             }
125         if(fabs(a[r][i]) < EPS)
126         {
127             ++res;
128             continue;
129         }
130         for(int j = 0; j < n; j++)
131             if(j != r && fabs(a[j][i]) > EPS)
132             {
133                 double tmp = a[j][i] / a[r][i];

```

```

134         for(int k = i; k <= n; k++)
135             a[j][k] -= tmp * a[r][k];
136     }
137     l[i] = true;
138     ++r;
139 }
140 for(int i = 0; i < n; i++)
141     if(l[i])
142         for(int j = 0; j < n; j++)
143             if(fabs(a[j][i]) > 0)
144                 ans[i] = a[j][n] / a[j][i];
145 return res;//返回解空间的维数
146 }
147 //常数线性齐次递推
148 /*已知  $f_x = a_0 f_{x-1} + a_1 f_{x-2} + \cdots + a_{n-1} f_{x-n}$  和  $f_0, f_1, \cdots, f_{n-1}$ , 给定  $t$ , 求  $f_t$ 
149  $f$  的递推可以看做是一个  $n \times n$  的矩阵  $A$  乘以一个  $n$  维列向量  $\beta$ , 即
150
```

$$A = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{n-1} & a_{n-2} & a_{n-3} & \cdots & a_0 \end{bmatrix}, \beta_n = \begin{bmatrix} f_{x-n} \\ f_{x-n+1} \\ \vdots \\ f_{x-2} \\ f_{x-1} \end{bmatrix}$$

```

151 则  $\beta_t = A^{t-n+1} \beta_0 (t \geq n)$ 
152 */

```

6 字符串

6.1 KMP 以及扩展 KMP

```
1  ///KMP
2  //O(n + m)
3  /*
4  next[] 的含义:  $x[i - next[i] \dots i] = x[0 \dots next[i] - 1]$ 
5  next[i] 为满足  $x[i - z \dots i - 1] = x[0 \dots z - 1]$  的最大 z 值 (就是 x 的自身匹配)
6  */
7  void pre_kmp(char x[], int m, int kmpNext[])
8  {
9      int i, j;
10     j = kmpNext[i = 0] = -1;
11     while(i < m)
12     {
13         while(j != -1 && x[i] != x[j]) j = kmpNext[j];
14         if(x[++i] == x[++j]) kmpNext[i] = kmpNext[j];
15         else kmpNext[i] = j;
16     }
17 }
18
19 //返回x在y中出现的次数, 可以重叠
20 //x是模式串, y是文本串
21 int KMP_Count(char x[], int m, char y[], int n, int next[])
22 {
23     int i = 0, j = 0, ans = 0;
24     pre_kmp(x, m, next);
25     while(i < n)
26     {
27         while(j != -1 && y[i] != x[j]) j = next[j];
28         ++i, ++j;
29         if(j >= m)
30         {
31             ans++;
32             j = next[j];
33         }
34     }
35     return ans;
36 }
37
38 ///扩展KMP
39 /*
40 复杂度:  $O(n + m)$ 
41 next[i]:  $x[i \dots m - 1]$  与  $x[0 \dots m - 1]$  的最长公共前缀
42 extend[i]:  $y[i \dots n - 1]$  与  $x[0 \dots m - 1]$  的最长公共前缀
43 */
44 void pre_exkmp(const char x[], int m, int next[])
45 {
46     for(int i = 0, j = -1, k, p; i < m; i++, j--)
47         if(j < 0 || i + next[i - k] >= p)
48         {
49             if(j < 0) j = 0, p = i;
50             while(p < m && x[p] == x[j]) j++, p++;
51             next[k = i] = j;
52         }
53     else
54         next[i] = next[i - k];
55 }
56
57 //x是模式串, y是文本串
58 void exkmp(char x[], int m, char y[], int n, int next[], int extend[])
59 {
60     pre_exkmp(x, m, next);
```

```

60     for(int i = 0, j = -1, k, p; i < n; i++, j++)
61         if(j < 0 || i + next[i - k] >= p)
62         {
63             if(j < 0) j = 0, p = i;
64             while(p < n && j < m && x[p] == y[j]) j++, p++;
65             extend[k = i] = j;
66         }
67     else
68         extend[i] = next[i - k];
69 }

```

6.2 回文串 palindrome

```

1 //manacher算法 O(n)
2 /*写法一
3 预处理：在字符串中加入一个分隔符（不在字符串中的符号），将奇数长度的回文串和偶数长度的回文串统一；
4 在字符串之前再加一个分界符（如'&'），防止比较时越界*/
5
6 void manacher(char *s, int len, int p[])
7 { //s = &s[0]#s[1]#...#s[len]\0
8     int i, mx = 0, id;
9     for(i = 1; i <= len; i++)
10     {
11         p[i] = mx > i ? min(p[2*id - i], mx - i) : 1;
12         while(s[i + p[i]] == s[i - p[i]]) ++p[i];
13         if(p[i] + i > mx) mx = p[i] + (id = i);
14         p[i] -= (i & 1) != (p[i] & 1); //去掉分隔符带来的影响
15     }
16     //此时，p[(2<<i) + 1]为以s[i]为中心的奇数长度的回文串的长度
17     //p[(2<<i)]为以s[i]和s[i+1]为中心的偶数长度的回文串的长度
18 }
19
20 /*写法二
21 将位置在[i, j]的回文串的长度信息存储在p[i+j]上
22 */
23 void manacher2(char *s, int len, int p[])
24 {
25     p[0] = 1;
26     for(int i = 1, j = 0; i < (len<<1) - 1; ++i)
27     {
28         int u = i >> 1, v = i - u, r = ((j + 1) >> 1) + p[j] - 1;
29         int u = i >> 1, v = i - u, r = ((j + 1) >> 1) + p[j] - 1;
30         p[i] = r < v ? 0 : min(r - v + 1, p[(j<<1) - 1]);
31         while(u > p[i] - 1 && v + p[i] < len && s[u - p[i]] == s[u + p[i]]) ++p[i];
32         if(u + p[i] - 1 > r) j = i;
33     }
34 }

```

6.3 哈希算法 Hash

```

1 ///滚动哈希算法 O(n + m)
2 //使用于求字符串s在字符串t中出现的位置或次数 可以简单的推到二维的情况
3 //哈希函数 $H(S) = (s_1B^{m-1} + s_2B^{m-2} + \dots + s_mB^0)\%h$ , 其中字符串 $S = s_1s_2 \dots c_m, m = |S|$ , B为基数,  $1 < B < h$ 
   且h与b互素
4 // $H(s_{k-1}s_k \dots s_{k+m-1}) = (H(s_k s_{k+1} \dots s_{k+m}) - s_k B^m + s_{k+m})\%h$ 
5 //单hash一般足够, 也可以使用双hash
6 //常用h:  $10^9 + 7, 10^9 + 9$ , B取比字符集大的一个素数.
7 //注意: 将要hash的每一个字符应该至少从1开始编号, 即不能为0.
8

```

```

9 //应用1: 在许多字符串中寻找与目标串相同的字符串的个数
10 LL B = 71, mod = 1000000007;
11 int cal(string s, string t)//查询s的子串是t的个数
12 {
13     int lens = s.length(), lent = t.length();
14     if(lent > lens) return 0;
15     LL BN = 1, hasht = 0, hashes = 0;
16     for(int i = 0; i < lent; i++) hasht = (hasht * B + (t[i] - 'a' + 1)) % mod, BN = BN * B % mod;
17     for(int i = 0; i < lens; i++) hashes = (hashes * B + (s[i] - 'a' + 1)) % mod;
18     int cnt = (hasht == hashes);
19     for(int i = lent; i < lens; i++)
20     {
21         hashes = (hashes * B + (s[i] - 'a' + 1)) % mod - BN * (s[i - lent] - 'a' + 1) % mod;
22         hashes = (hashes % mod + mod) % mod;
23         if(hashes == hasht) cnt++;
24     }
25     return hashes;
26 }

```

6.4 后缀数组 Suffix Array

```

1 ///后缀数组(Suffix Array)
2 /*
3 后缀数组是指将某个字符串的所有后缀按字典序排序后得到的数组
4 */
5 //计算后缀数组
6 //朴素做法 将所有后缀进行排序 $O(n^2 \log n)$ 采用快排 适用于 $m$ 比较大的时候
7 //Manber-Myers  $O(n \log^2 n)$ 
8 int rk;
9 int sa[NUM], rk[NUM], height[NUM];
10 int cmp(int i, int j)
11 {
12     if(rk[i] != rk[j])
13         return rk[i] < rk[j];
14     else
15     {
16         int ri = i + rk <= n ? rk[i + rk] : -1;
17         int rj = j + rk <= n ? rk[j + rk] : -1;
18         return ri < rj;
19     }
20 }
21 void da(int *a, int n)
22 {
23     int i;
24     a[n] = -1;
25     for(i = 0; i <= n; i++)
26     {
27         sa[i] = i;
28         rk[i] = a[i];
29     }
30     for(m = 1; rk[n] < n; m <= 1)
31     {
32         sort(sa, sa + n + 1, cmp);
33         tmp[sa[0]] = 0;
34         for(i = 1; i <= n; i++)
35             tmp[sa[i]] = tmp[sa[i - 1]] + (cmp(sa[i], sa[i - 1]) || cmp(sa[i - 1], sa[i]));
36         for(i = 0; i <= n; i++)
37             rk[i] = tmp[i];
38     }
39 }
40
41 //应用

```

```

42 // 基于后缀数组的字符串匹配
43 bool contain(string s, int *sa, string t)
44 {
45     int a = 0, b = s.length();
46     while(b - a > 1)
47     {
48         int c = (a + b) / 2;
49         if(s.compare(sa[c], t.length(), t) < 0) a = c;
50         else
51             b = c;
52     }
53     return s.compare(sa[b], t.length(), t) == 0;
54 }
55
56 /// 倍增法模板:  $O(n \log n)$  采用基数排序
57 ///  $n$  为字符个数  $r[n-1]$  要比所有  $a[0, n-2]$  要小
58 ///  $r$  字符串对应的数组
59 ///  $m$  为最大字符值+1
60 int sa[NUM];
61 int rk[NUM], height[NUM], sv[NUM], sn[NUM];
62 void da(char r[], int n, int m)
63 {
64     int i, j, p, *x = rk, *y = height;
65     for(i = 0; i < m; i++) sn[i] = 0;
66     for(i = 0; i < n; i++) sn[x[i]]++;
67     for(i = 1; i < m; i++) sn[i] += sn[i-1];
68     for(i = n-1; i >= 0; i--) sa[sn[x[i]]] = i;
69     for(j = p = 1; p < n; j <= 1, m = p)
70     {
71         for(p = 0, i = n-j; i < n; i++) y[p++] = i;
72         for(i = 0; i < n; i++) if(sa[i] >= j) y[p++] = sa[i] - j;
73         for(i = 0; i < n; i++) sv[i] = x[y[i]];
74         for(i = 0; i < m; i++) sn[i] = 0;
75         for(i = 0; i < n; i++) sn[sv[i]]++;
76         for(i = 1; i < m; i++) sn[i] += sn[i-1];
77         for(i = n-1; i >= 0; i--) sa[sn[sv[i]]] = y[i];
78         for(swap(x, y), x[sa[0]] = 0, i = 1, p = 1; i < n; i++)
79             x[sa[i]] = (y[sa[i]] == y[sa[i-1]] && y[sa[i] + j] == y[sa[i-1] + j]) ? p-1 : p++;
80     }
81 }
82
83 /// DC3 模板:  $O(3n)$ 
84 int sa[NUM * 3], r[NUM * 3]; // sa 数组和 r 数组要开三倍大小的空间
85 int rk[NUM], height[NUM], sn[NUM], sv[NUM];
86 #define F(x) ((x) / 3 + ((x) % 3 == 1 ? 0 : tb))
87 #define G(x) ((x) < tb ? (x) * 3 + 1 : ((x) - tb) * 3 + 2)
88 int cmp0(int r[], int a, int b)
89 {return r[a] == r[b] && r[a+1] == r[b+1] && r[a+2] == r[b+2];}
90 int cmp12(int r[], int a, int b, int k)
91 {
92     if(k == 2) return r[a] < r[b] || (r[a] == r[b] && cmp12(r, a+1, b+1, 1));
93     else return r[a] < r[b] || (r[a] == r[b] && sv[a+1] < sv[b+1]);
94 }
95 void sort(int r[], int a[], int b[], int n, int m) // 基数排序
96 {
97     int i;
98     for(i = 0; i < m; i++) sn[i] = 0;
99     for(i = 0; i < n; i++) sn[sv[i] = r[a[i]]]++;
100     for(i = 1; i < m; i++) sn[i] += sn[i-1];
101     for(i = n-1; i >= 0; i--) b[sn[sv[i]]] = a[i];
102 }
103 void dc3(int r[], int sa[], int n, int m)
104 {
105     int *rn = r + n, *san = sa + n, *wa = height, *wb = rk;

```

```

106     int i, j, p, ta = 0, tb = (n + 1) / 3, tbc = 0;
107     r[n] = r[n + 1] = 0;
108     for(i = 0; i < n; i++) if(i % 3 != 0) wa[tbc++] = i;
109     sort(r + 2, wa, wb, tbc, m);
110     sort(r + 1, wb, wa, tbc, m);
111     sort(r, wa, wb, tbc, m);
112     for(p = 1, rn[F(wb[0])] = 0, i = 1; i < tbc; i++)
113         rn[F(wb[i])] = cmp0(r, wb[i - 1], wb[i]) ? p - 1 : p++;
114     if(p < tbc) dc3(rn, san, tbc, p);
115     else for(i = 0; i < tbc; i++) san[rn[i]] = i;
116     for(i = 0; i < tbc; i++) if(san[i] < tb) wb[ta++] = san[i] * 3;
117     if(n % 3 == 1) wb[ta++] = n - 1;
118     sort(r, wb, wa, ta, m);
119     for(i = 0; i < tbc; i++) sv[wb[i] = G(san[i])] = i;
120     for(i = 0, j = 0, p = 0; i < ta && j < tbc; p++)
121         sa[p] = cmp12(r, wa[i], wb[j], wb[j] % 3) ? wa[i++] : wb[j++];
122     for(; i < ta; p++) sa[p] = wa[i++];
123     for(; j < tbc; p++) sa[p] = wb[j++];
124 }
125
126 ///高度数组longest common prefix
127 ///height[i] = suffix(sa[i])和suffix(sa[i - 1])的最长公共前缀lcp(sa[i], sa[i - 1])
128 ///rk[0..n-1]:rk[i]保存的是原串中suffix[i]的名次
129 ///height数组性质:
130 ///任意两个suffix(j)和suffix(k)(rank[j] < rank[k])的最长公共前缀:  $\min_{i=j+1 \rightarrow k} \{height[rk[i]]\}$ 
131 ///height[rk[i]] ≥ height[rk[i - 1]] - 1
132 int rk[maxn], height[maxn];
133 void cal_height(char *r, int *sa, int n)
134 {
135     int i, j, k = 0;
136     for(i = 0; i < n; i++) rk[sa[i]] = i;
137     for(i = 0; i < n; height[rk[i++]] = k)
138         for(k ? k-- : 0, j = sa[rk[i] - 1]; r[i + k] == r[j + k]; k++);
139 }
140 ///后缀数组应用
141 ///询问任意两个后缀的最长公共前缀: RMQ问题,  $\min(i=j+1 \rightarrow k) \{height[rk[i]]\}$ 
142 ///重复子串: 字符串R在字符串L中至少出现2次, 称R是L的重复子串
143 ///可重叠最长重复子串:  $O(n)$  height数组中的最大值
144 ///不可重叠最长重复子串:  $O(n \log n)$ 变为二分答案, 判断是否存在两个长度为k的子串是相同且不重叠的.
145     将排序后后缀分为若干组, 其中每组的后缀的height值都不小于k,
146     然后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,
147     判断sa的最大值和最小值之差不小于k, 如果一组满足, 则存在, 否则不存在.
148 ///可重叠的k次最长重复子串:  $O(n \log n)$  二分答案, 将后缀分为若干组, 判断有没有一个组的后缀个数不小于k.
149 ///不相同的子串个数: 等价于所有后缀之间不相同的前缀的个数 $O(n)$ : 后缀按suffix(sa[1]), suffix(sa[2]), ...,
150     suffix(n)的顺序计算, 新进一个后缀suffix(sa[k]), 将产生 $n - sa[k] + 1$ 的新的前缀,
151     其中height[k]的和前面是相同的, 所以suffix(sa[k])贡献 $n - sa[k] + 1 - height[k]$ 个不同的子串.
152     故答案是 $\sum_{k=1}^n n - sa[k] - 1 - height[k]$ .
153 ///最长回文子串: 字符串S(长度n)变为字符串+特殊字符+反写的字符串,
154     求以某字符(位置k)为中心的最长回文子串(长度为奇数或偶数), 长度为: 奇数lcp(suffix(k), suffix(2*n
155     + 2 - k)); 偶数lcp(suffix(k), suffix(2*n + 3 - k))  $O(n \log n)$  RMQ: $O(n)$ 
156 ///连续重复子串: 字符串L是有字符串S重复R次得到的.
157 ///给定L, 求R的最大值:  $O(n)$ , 枚举S的长度k, 先判断L的长度是否能被k整除, 在看lcp(suffix(1),
158     suffix(k+1))是否等于 $n - k$ . 求解时只需预处理height数组中的每一个数到height[rk[1]]的最小值即可
159 ///给定字符串, 求重复次数最多的连续重复子串 $O(n \log n)$ : 先穷举长度L,
160     然后求长度为L的子串最多能连续出现几次. 首先连续出现1次是肯定可以的,
161     所以这里只考虑至少2次的情况. 假设在原字符串中连续出现2次, 记这个子字符串为S,
162     那么S肯定包括了字符 $r[0]$ ,  $r[L]$ ,  $r[L*2]$ ,  $r[L*3]$ , ...中的某相邻的两个.
163     所以只须看字符 $r[L*i]$ 和 $r[L*(i+1)]$ 往前和往后各能匹配到多远, 记这个总长度为K,
164     那么这里连续出现了 $K/L+1$ 次. 最后看最大值是多少.
165 ///字符串A和B最长公共前缀 $O(|A|+|B|)$ : 新串: A+特殊字符+B,height//k : A++B,
166     对后缀数组分组(每组height值都不小于k), 每组中扫描到B时,
167     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, 统计得结果.

```



```

153 // 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  $O(n \log n)$ ：连接所有字符串，二分答案，然后分组，
    判断每组后缀是否出现在至少  $k$  个不同的原串中。
154 // 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  $O(n \log n)$ ：做法同上，
    也是先将  $n$  个字符串连起来，中间用不相同的且没有出现在字符串中的字符隔开，求后缀数组。
    然后二分答案，再将后缀分组。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）。
155 // 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：只需要先将每个字符串都反过来写一遍，
    中间用一个互不相同的且没有出现在字符串中的字符隔开，再将  $n$  个字符串全部连起来，
    中间也是用一个互不相同的且没有出现在字符串中的字符隔开，求后缀数组。然后二分答案，再将后缀分组。
    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。
    这个做法的时间复杂度为  $O(n \log n)$ 。

```

6.5 字典树 Trie

```

1  ///字典数 Trie
2  struct Trie
3  {
4      Trie *next[26]; // 根据字符集的大小变化
5      int cnt; // 字符串个数，根据应用变化
6      Trie()
7      {
8          memset(next, 0, sizeof(next));
9          cnt = 0;
10     }
11 };
12 Trie *root = NULL;
13 void clearTrie(Trie *p) // 清空整棵字典树
14 {
15     if(p)
16     {
17         for(int i = 0; i < 26; i++)
18             clearTrie(p->next[i]);
19         delete p;
20     }
21 }
22 void initTrie()
23 {
24     clearTrie(root);
25     root = new Trie;
26 }
27 // 将字符串 str 加入字符串中
28 void createTrie(char str[])
29 {
30     int i = 0, id;
31     Trie *p = root;
32     while((id = str[i] - 'a') >= 0) // 得到该字符的编号
33     {
34         if(p->next[id] == NULL)
35         {
36             p->next[id] = new Trie;
37         }
38         p = p->next[id];
39     }
40     p->cnt++;
41 }
42 // 查询某字符串是否在字典树中
43 bool findTrie(char str[])
44 {
45     int i = 0, id;
46     Trie *p = root;

```

```

47 while((id = str[i++] - 'a') >= 0)
48 {
49     p = p->next[id];
50     if(p == NULL) return false;
51 }
52 return p->cnt > 0;
53 }
54 //从字典树中删除一个字符串
55 void deleteTrie(char str[])
56 {
57     int i = 0, id;
58     Trie *p = root;
59     while((id = str[i++] - 'a') >= 0)
60     {
61         p = p->next[id];
62         if(p == NULL) return ;
63     }
64     if(p->cnt) p->cnt--;
65 }

```

6.6 字符串循环同构的最小表示法

```

1  ///最小表示法 Minimum Representation
2  //对于一个字符串S, 求S的循环的同构字符串S' 中字典序最小的一个
3  //O(|S|)
4  int MinimumRepresentation(char *s, int len)
5  {
6      int i = 0, j = 1, k = 0, t;
7      while(i < len && j < len && k < len)
8      {
9          t = s[(i + k) >= len ? i + k - len : i + k] - s[(j + k) >= len ? j + k - len : j + k];
10         if(!t) k++;
11         else
12         {
13             if(t > 0) i = i + k + 1;
14             else j = j + k + 1;
15             if(i == j) ++j;
16             k = 0;
17         }
18     }
19     return (i < j ? i : j);
20 }

```

6.7 字符串问题汇总

1. 字符串A的任意子串x和字符串B的任意子串y(x, y可以为空)形成一个新的字符串xy, 能构成多少种字符串?
来源: hdu5344, 2015多校第5场1001
做法: 问题关键在于去重. 答案为: A中不同子串的个数 \times B中不同子串的个数 $- \sum_{c='a'}^{z'} z'_c$
(A中以字符c结尾的不同子串数 \times B中以字符c开头的不同子串数). 将A反转, 然后用后缀数组维护A,
B中不同子串的个数, 和以某字符开头的不同子串数即可. 答案会爆long long, 要用unsigned long long
2. 求后缀数组中, 某一子串S[l, r]出现的首次出现的位置, 或者最后出现的位置,
或者求某一公共子串出现的次数
标签: ST表, 二分
做法: 用ST表预处理高度数组, 那么可以在O(1)时间内求出任意两个后缀之间的最长公共前缀;
在某后缀后面(或前面)的后缀与该后缀的lcp是非增(非减)的,
因此可以用二分求出距该后缀最远的一个与该后缀有最长公共前缀长度大于等于len的后缀.
3. 询问一个字符串S中, 比某子串S[L, R]字典序小的子串数.

10 来源: Gym 100418C
 11 标签: 后缀数组, 二分, 区间更新查询
 12 做法: 将查询离线处理, 用二分得到每个被询问的子串在后缀数组中第一次出现的位置,
 然后按照后缀的字典序遍历, (即遍历后缀数组) 得到比每个后缀小的子串数, 在此过程中,
 处理询问的答案。比某个后缀小的子串数, 为不是该后缀的前缀, 但比该后缀小的子串数 +
 是该后缀的前缀的子串数。处理时, 随高度数组处理, 有前缀长度为 i ($i = 0, 1, 2, \dots, \text{height}[i + 1]$),
 比该后缀小的子串数。直接处理会超时, 要用线段树更新和查询区间。

13
 14 4. 求有多少个 z , 使得由 $z = (z \cdot a + c) / k (\% m)$ 产生的一个长度为 n 的数字序列, 将小于 $m/2$ 的数字标记为 0 ,
 其他的标记为 1 后与目标串 b 相同。

15 来源: Gym 100523G
 16 标签: hash, 倍增
 17 做法: 分析生成函数, 可以发现这是一个有 m 个状态的有限状态自动机,
 然后可以在 $O(m)$ 时间内求出每个状态将会转移到的下一个状态。类似于 ST 表的思想,
 我们预处理出从每个状态出发, 长度为 2 的幂的生成字符串的 hash 值, 并保存下一个会转移到的位置。
 就有转移方程:

$$\text{hash}[i][j] = \text{hash}[i][j - 1] \times B[1 \ll (j - 1)] + \text{hash}[\text{pos}[i][j - 1]][j - 1], \text{pos}[i][j] = \text{pos}[\text{pos}[i][j - 1]][j - 1],$$

 然后我们就可以在 $O(m \log n)$ 时间内求出每个 z 出发, 长度为 n 的生成字符串的 hash 值,
 然后与目标串 b 的 hash 值比较, 然后统计即可。然而, 由于内存限制, 我们要用滚动数组预处理,
 并在此过程中求出长度为 n 的生成字符串的 hash 值。

18 时间复杂度: $O(m \log n)$

7 计算几何

7.1 计算几何基础

```
1 //精度设置
2 const double EPS = 1e-6;
3 //点(向量)的定义和基本运算
4 /*
5 向量点积  $a \cdot b = |a||b| \cos \theta$ 
6 点积  $> 0$ , 表示两向量夹角为锐角
7 点积  $= 0$ , 表示两向量垂直
8 点积  $< 0$ , 表示两向量夹角为钝角
9 */
10 /*
11 向量叉积  $a \times b = |a||b| \sin \theta$ 
12 叉积  $> 0$ , 表示向量  $b$  在当前向量逆时针方向
13 叉积  $= 0$ , 表示两向量平行
14 叉积  $< 0$ , 表示向量  $b$  在当前向量顺时针方向
15 */
16 inline int sgn(double x) {if(x < -EPS) return -1; return x > EPS ? 1 : 0;}
17 struct Point
18 {
19     double x, y;
20     Point(double _x = 0.0, double _y = 0.0): x(_x), y(_y) {}
21     Point operator + (const Point &b) const {return Point(x + b.x, y + b.y);} // 向量加法
22     Point operator - (const Point &b) const {return Point(x - b.x, y - b.y);} // 向量减法
23     double operator * (const Point &b) const {return x * b.x + y * b.y;} // 向量点积
24     double operator ^ (const Point &b) const {return x * b.y - y * b.x;} // 向量叉积
25     Point operator * (double b) {return Point(x * b, y * b);} // 标量乘法
26     Point rot(double ang) {return Point(x * cos(ang) - y * sin(ang), x * sin(ang) + y *
        cos(ang));} // 旋转
27     double norm() {return sqrt(x * x + y * y);} // 向量的模
28 };
29 // 直线 线段定义
30 // 直线方程: 两点式:  $(x_2 - x_1)(y - y_1) = (y_2 - y_1)(x - x_1)$ 
31 struct Line
32 {
33     Point s, e;
34     // double k;
35     Line() {}
36     Line(Point _s, Point _e)
37     {
38         s = _s, e = _e;
39         // k = atan2(e.y - s.y, e.x - s.x);
40     }
41     // 求两直线交点
42     // 返回 -1 两直线重合, 0 相交, 1 平行
43     pair<int, Point> operator &(Line b)
44     {
45         if(sgn((s - e) ^ (b.s - b.e)) == 0)
46         {
47             if(sgn((s - b.e) ^ (b.s - b.e)) == 0) return make_pair(-1, s); // 重合
48             else return make_pair(1, s); // 平行
49         }
50         double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));
51         return make_pair(0, Point(s.x + (e.x - s.x) * t, s.y + (e.y - s.y) * t));
52     }
53 };
54
55 // 两点间距离
56 double dist(Point &a, Point &b) {return (a - b).norm();}
57
58 /* 判断点 p 在线段 l 上
```

```

59 *  $(p - l.s) \wedge (l.s - l.e) = 0$ ; 保证点 $p$ 在直线 $L$ 上
60 *  $p$ 在线段 $l$ 的两个端点 $l.s, l.e$ 为对角定点的矩形内
61 */
62 bool Point_on_Segment(Point &p, Line &l)
63 {
64     return sgn((p - l.s) ^ (l.s - l.e)) == 0 &&
65         sgn((p.x - l.s.x) * (p.x - l.e.x)) <= 0 &&
66         sgn((p.y - l.s.y) * (p.y - l.e.y)) <= 0;
67 }
68 //判断点 $p$ 在直线 $l$ 上
69 bool Point_on_Line(Point &p, Line &l)
70 {
71     return sgn((p - l.s) ^ (l.s - l.e)) == 0;
72 }
73
74 /*判断两线段 $l1, l2$ 相交
75 * 1. 快速排斥实验: 判断以 $l1$ 为对角线的矩形是否与以 $l2$ 为对角线的矩形是否相交
76 * 2. 跨立实验:  $l2$ 的两个端点是否在线段 $l1$ 的两端
77 */
78 bool seg_seg_inter(Line seg1, Line seg2)
79 {
80     return
81         sgn(max(seg1.s.x, seg1.e.x) - min(seg2.s.x, seg2.e.x)) >= 0 &&
82         sgn(max(seg2.s.x, seg2.e.x) - min(seg1.s.x, seg1.e.x)) >= 0 &&
83         sgn(max(seg1.s.y, seg1.e.y) - min(seg2.s.y, seg2.e.y)) >= 0 &&
84         sgn(max(seg2.s.y, seg2.e.y) - min(seg1.s.y, seg1.e.y)) >= 0 &&
85         sgn((seg2.s - seg1.e) ^ (seg1.s - seg1.e)) * sgn((seg2.e - seg1.e) ^ (seg1.s - seg1.e)) <=
86         0 &&
87         sgn((seg1.s - seg2.e) ^ (seg2.s - seg2.e)) * sgn((seg1.e - seg2.e) ^ (seg2.s - seg2.e)) <=
88         0;
89 }
90 //判断直线与线段相交
91 bool seg_line_inter(Line &line, Line &seg)
92 {
93     return sgn((seg.s - line.e) ^ (line.s - line.e)) * sgn((seg.e - line.e) ^ (line.s - line.e)) <=
94     0;
95 }
96 //点到直线的距离, 返回垂足
97 Point Point_to_Line(Point p, Point l)
98 {
99     double t = ((p - l.s) * (l.e - l.s)) / ((l.e - l.s) * (l.e - l.s));
100     return Point(l.s.x + (l.e.x - l.s.x) * t, l.s.y + (l.e.y - l.s.y) * t);
101 }
102 //点到线段的距离
103 //返回点到线段最近的点
104 Point Point_to_Segment(Point p, Line seg)
105 {
106     double t = ((p - seg.s) * (seg.e - seg.s)) / ((seg.e - seg.s) * (seg.e - seg.s));
107     if(t >= 0 && t <= 1)
108         return Point(seg.s.x + (seg.e.x - seg.s.x) * t, seg.s.y + (seg.e.y - seg.s.y) * t);
109     else if(sgn(dist(p, seg.s) - dist(p, seg.e)) <= 0)
110         return seg.s;
111     else
112         return seg.e;
113 }

```

7.2 三角形 Triangle

```

1 /// 三角形
2 /*

```

3 设三角形 $\triangle ABC$ 的三个顶点 A, B, C (对应内角角度为 A, B, C)；对应的三条边为 a, b, c (对应边长为 a, b, c)。
4 三点坐标对应为： $A(x_1, y_1), B(x_2, y_2), C(x_3, y_3)$
5 性质：
6 三角形不等式：三角形两边之和大于第三边，两边之差的绝对值小于第三边。
7 三角形任意一个外角大于不相邻的一个内角。
8 勾股定理：三角形是直角三角形($C = 90^\circ$)，则 $a^2 + b^2 = c^2$ 。反之亦然。
9 正弦定理： $(R$ 为三角形外接圆半径)

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} = 2R$$

10 余弦定理：

$$11 \quad a^2 = b^2 + c^2 - 2bc \cdot \cos A$$

$$12 \quad b^2 = a^2 + c^2 - 2ac \cdot \cos B$$

$$13 \quad c^2 = a^2 + b^2 - 2ab \cdot \cos C$$

14 角度：

15 三角形两内角之和，等于第三角的外角。

16 三角形的内角和为 180° 。

17 三角形分类：钝角三角形(其中一角是钝角($> 90^\circ$)的三角形)，

直角三角形(其中一角是直角($= 90^\circ$)的三角形)，和锐角三角形(三个角都是锐角($< 90^\circ$)的三角形)。

18 等边三角形的内角均为 60° 。

19 等腰三角形两底角相等。

20 面积：

21 设 a, b 为已知的两边， C 为其夹角，三角形面积 $\Delta = \frac{1}{2}ab \sin C$

22 已知三角形的三边，有：

23 海伦公式：令 $p = \frac{a+b+c}{2}$ ，则 $\Delta = \sqrt{p(p-a)(p-b)(p-c)}$

24 秦九韶的三斜求积法： $\Delta = \sqrt{\frac{1}{4} \left[c^2 a^2 - \left(\frac{c^2 + a^2 - b^2}{2} \right)^2 \right]}$

25 幂和： $\Delta = \frac{1}{4} \sqrt{(a^2 + b^2 + c^2)^2 - 2(a^4 + b^4 + c^4)}$

26 已知一边 a ，和该边上的高 h_a ，则 $\Delta = \frac{ah_a}{2}$

27 已知三点坐标，面积为下列行列式的绝对值：

$$\frac{1}{2} \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

28 三角的五心：

29 重心(形心 *Centroid*)

30 定义：三条中线的交点。

31 坐标：三点坐标的算术平均，即 $\left(\frac{x_1 + x_2 + x_3}{2}, \frac{y_1 + y_2 + y_3}{2} \right)$ 。

32 三角形的重心与三顶点连线，所形成的六个三角形面积相等。

33 顶点到重心的距离是中线的 $\frac{2}{3}$ 。

34 重心到三角形3个顶点距离的平方和最小。

35 以重心为起点，以三角形三定点为终点的三条向量之和等于零向量。

36 */

37 **Point MassCenter(Point A, Point B, Point C)**

38 {

39 **return** (A + B + C) * (1.0 / 3.0);

40 }

41 /*

42 内心*I*(*Inner Center*)

43 定义：三个内角的角平分线的交点。

44 坐标：三点坐标的面积加权平均，即 $\left(\frac{ax_1 + bx_2 + cx_3}{a + b + c}, \frac{ay_1 + by_2 + cy_3}{a + b + c} \right)$ 。

45 内心到三角形三边的距离相等，等于内切圆半径 r 。

46 内心是三角形内切圆的圆心，内切圆半径 r ，有 $\Delta = \frac{1}{2}(a + b + c)r$ 。

47 直角三角形两股和等于斜边长加上该三角形内切圆直径，即 $a + b = c + 2r$ ，

48 由此性质再加上勾股定理 $a^2 + b^2 = c^2$ ，可推得： $\Delta = r(r + c)$

49 */

50 **Point InnerCenter(Point A, Point B, Point C)**

51 {

52 **double** a = dist(B, C), b = dist(A, C), c = dist(A, B);

53 **return** (A * a + B * b + C * c) * (1.0 / (a + b + c));

54 }

55 /*

```

56 | 外心(Circum Center)
57 | 定义：三条边垂直平分线的交点。
58 | 坐标：

```

$$\left(\begin{array}{c|c|c} \begin{vmatrix} x_1^2 + y_1^2 & 1 & y_1 \\ x_2^2 + y_2^2 & 1 & y_2 \\ x_3^2 + y_3^2 & 1 & y_3 \end{vmatrix} & \begin{vmatrix} x_1^2 + y_1^2 & x_1 & 1 \\ x_2^2 + y_2^2 & x_2 & 1 \\ x_3^2 + y_3^2 & x_3 & 1 \end{vmatrix} \\ \hline \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} & \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \end{array} \right)$$

```

59 | 外心到三个顶点的距离都相等，等于外接圆的半径R。
60 | 直角三角形的外心是斜边的中点，外接圆半径R为斜边的一半；钝角三角形的外心在三角形外，
    | 靠近最长边；锐角三角形的外心在三角形内。
61 | 外心是三角形外接圆的圆心，外接圆半径R，有  $\Delta = \frac{abc}{4R}$ 
62 | */
63 | //采用求垂直平分线相交的方法
64 | Point CircumCenter(Point A, Point B, Point C)
65 | {
66 |     return (Line((A + B) * 0.5, (A - B).rot(PI * 0.5) + ((A + B) * 0.5)) &
67 |             Line((B + C) * 0.5, (B - C).rot(PI * 0.5) + ((B + C) * 0.5))).second;
68 | }
69 | //公式法
70 | Pointy CircumCenter(Point A, Point B, Point C)
71 | {
72 |     Point t1 = B - A, t2 = C - A, t3((t1 * t1) * 0.5, (t2 * t2) * 0.5);
73 |     swap(t1.y, t2.x);
74 |     return A + Point(t3 ^ t2, t1 ^ t3) * (1.0 / (t1 ^ t2));
75 | }
76 | /*
77 | 垂心H(Ortho Center)
78 | 定义：三条高的交点。
79 | 坐标：

```

$$\left(\begin{array}{c|c|c} \begin{vmatrix} x_2x_3 + y_2y_3 & 1 & y_1 \\ x_3x_1 + y_3y_1 & 1 & y_2 \\ x_1x_2 + y_1y_2 & 1 & y_3 \end{vmatrix} & \begin{vmatrix} x_2x_3 + y_2y_3 & x_1 & 1 \\ x_3x_1 + y_3y_1 & x_2 & 1 \\ x_1x_2 + y_1y_2 & x_3 & 1 \end{vmatrix} \\ \hline \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} & \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \end{array} \right)$$

```

80 | 垂心分每条高线的两部分乘积相等，即  $AH \times HH_A = BH \times HH_B = CH \times HH_C$ 。
81 | 直角三角形垂心为C( $\angle C = 90^\circ$ )，锐角三角形的垂心在三角形内部，钝角三角形的垂心在三角形外部。
82 | 垂心到三角形一顶点距离为此三角形外心到此顶点对边距离的2倍。
83 | 一个三角形ABC的三个顶点A, B, C和它的垂心H构成一个垂心组：A, B, C, H。也就是说，
    | 这四点中任意的三点的垂心都是第四点。
84 | 由海伦公式和三角形面积公式，可以推出各边上的高的长度。
85 | 反海伦公式
86 | 如果设  $h_s = \frac{h_a^{-1} + h_b^{-1} + h_c^{-1}}{2}$ ，那么有以下类似于海伦公式的三角形面积公式

```

$$S^{-1} = 4\sqrt{h_s(h_s - h_a^{-1})(h_s - h_b^{-1})(h_s - h_c^{-1})}$$

```

87 | */
88 | Point OrthoCenter(Point A, Point B, Point C)
89 | {
90 |     return MassCenter(A, B, C) * 3.0 - CircumCenter(A, B, C) * 2.0;
91 | }
92 | /*
93 | 旁心J
94 | 定义：三角形一内角平分线和另外两顶点处的外角平分线的点。
95 | 旁心是三角形旁切圆(与三角形的一边和其他两边的延长线相切的圆)的圆心，每个三角形有三个旁心，
    | 而且一定在三角形外。
96 | 旁心到三边的距离相等。
97 | 旁切圆与三角形的边(或其延长线)相切的点称为旁切点。
98 | 某顶点和其对面的旁切点将三角形的圆周等分为两半。

```

三个旁心与内心组成一个垂心组，也就是说内心是三个旁心所组成的三角形的垂心，而相应的三个垂足则是旁心所对的顶点。

旁心坐标和旁切圆半径：

$$J_A = \left(\frac{bx_2 + cx_3 - ax_1}{b + c - a}, \frac{by_2 + cy_3 - ay_1}{b + c - a} \right), r_A = \frac{2\Delta}{b + c - a}$$

$$J_B = \left(\frac{ax_1 + cx_3 - bx_2}{a + c - b}, \frac{ay_1 + cy_3 - by_2}{a + c - b} \right), r_B = \frac{2\Delta}{a + c - b}$$

$$J_C = \left(\frac{bx_2 + ax_1 - cx_3}{a + b - c}, \frac{by_2 + ay_1 - cy_3}{a + b - c} \right), r_C = \frac{2\Delta}{a + b - c}$$

关系：

等边三角形四心(除旁心)重合。

等腰三角形重心，中心和垂心都位于顶点向底边的垂线。

欧拉线：三角形的垂心，外心，重心和九点圆圆心的一条直线。

欧拉线上的四点中九点圆圆心到垂心和外心的距离相等，而且重心到外心的距离是重心到垂心距离的一半。注意内心一般不在欧拉线上，除了等腰三角形外。

重心，内心，奈格尔点，类似重心四点共线。

三角形的外接圆半径 R ，内切圆半径 r 以及内外心间距 OI 之间有如下关系： $R^2 - OI^2 = 2Rr$ 。

内切圆在一边上的切点与旁切圆在该边的切点之间的距离恰好是另外两边的差(绝对值)。

对于一个顶点(比如 A)所对的旁切圆，三角形 ABC 的外接圆半径 R ，

A 所对旁切圆半径 r_A 以及内外心间距 OJ_A 之间关系： $OJ_A^2 - R^2 = 2Rr_A$

内心 I ， B ， C ， J_A 四点共圆，其中 IJ_A 是这个圆的直径，而圆心 P_A 在三角形 ABC 的外接圆上，并且过 BC 的中垂线，即等分劣弧 BC 。对其它两边也有同样的结果。

三角形内切圆的半径 r 与三个顶点上的高 h_a, h_b, h_c 有如下的关系：

$$\frac{1}{r} = \frac{1}{h_a} + \frac{1}{h_b} + \frac{1}{h_c}$$

三个旁切圆的半径也和高等有类关系：

$$\frac{1}{r_A} = -\frac{1}{h_a} + \frac{1}{h_b} + \frac{1}{h_c}$$

$$\frac{1}{r_B} = \frac{1}{h_a} - \frac{1}{h_b} + \frac{1}{h_c}$$

$$\frac{1}{r_C} = \frac{1}{h_a} + \frac{1}{h_b} - \frac{1}{h_c}$$

费马点：

三角形内到三角形的三个顶点 A, B, C 的距离之和 $PA+PB+PC$ 最小的点 P 。

每个三角形只有一个费马点。

费马点求法：

当有一个内角不小于 120° 时，费马点为此角对应顶点。

当三角形的内角都小于 120° 时，以三角形的每一边为底边，向外做三个正三角形 $\triangle ABC'$ ， $\triangle BCA'$ ， $\triangle CAB'$ ，连接 CC' ， BB' ， AA' ，则三条线段的交点就费马点。(此时 $\angle APB = \angle APC = \angle BPC = 120^\circ$)

九点圆(又称欧拉圆，费尔巴哈圆)，在平面几何中，对任何三角形，九点圆通过三角形三边的中点，三高的垂足，和顶点到垂心的三条线段的中点。

九点圆定理指出对任何三角形，这九点必定共圆；

九点圆的半径是外接圆的一半，且九点圆平分垂心与外接圆上的任一点的连线；

圆心在欧拉线上，且在垂心到外心的线段的中点；

九点圆和三角形的内切圆和旁切圆相切(费尔巴哈定理)，切点称为费尔巴哈点；

圆周上四点任取三点做三角形，四个三角形的九点圆圆心共圆(库利奇一大上定理)。

奈格尔点：

旁切点 J_A, J_B, J_C 分别是三旁切圆和三条边的切点，直线 AJ_A, BJ_B, CJ_C 共点交于三角形 ABC 的奈格尔点 N 。

另一种方法构造 J_A ，从点 A 出发沿着三角形 ABC 的边走到半周长位置，类似的得到 J_B 和 J_C 。

因为这个构造，奈格尔点有时也被称为平分周长点(或译界心)。

半角定理：三角形的三个角的半角的的正切和三边有如下关系：

$$\tan \frac{A}{2} = \frac{p}{b+c-a}, \tan \frac{B}{2} = \frac{p}{a+c-b}, \tan \frac{C}{2} = \frac{p}{a+b-c}$$

$$\text{其中, } p = \sqrt{\frac{(b+c-a)(a+c-b)(a+b-c)}{a+b+c}}$$

角平分线长度

设在三角形 ABC 中，已知三边 a, b, c ，若三个角 A, B, C 的角平分线分别为 t_a, t_b, t_c

则用三边表示三条内角平分线长度公式为

$$t_a = \frac{1}{b+c} \sqrt{(b+c+a)(b+c-a)bc}$$

$$t_b = \frac{1}{a+c} \sqrt{(a+c+b)(a+c-b)ac}$$

$$t_c = \frac{1}{a+b} \sqrt{(a+b+c)(a+b-c)ab}$$

等角共轭


```

138 | 几何学中，设点  $P$  是三角形  $ABC$  平面上一点，作直线  $PA$ ,  $PB$  和  $PC$  分别关于角  $A$ ,  $B$  和  $C$ 
    | 的平分线的反射，这三条反射线必然交于一点，称此点为  $P$  关于三角形  $ABC$  的等角共轭。
    | (这个定义只对点，不是对三角形  $ABC$  的边。)
139 | 内心  $I$  的等角共轭点是自身。垂心  $H$  的等角共轭点是外心  $O$ 。重心的等角共轭点是类似重心  $K$ 。
140 | */

```

7.3 多边形 Polygon

```

1  /// 多边形
2  /**判断点在凸多边形内
3  //点形成一个凸包，而且按逆时针排序（如果是顺时针把里面的 < 0 改为 > 0）
4  //点的编号：0~n-1
5  //返回值：-1: 点在凸多边形外；0: 点在凸多边形边界上；1: 点在凸多边形内
6  int inConvexPoly(Point a, Point p[], int n)
7  {
8      p[n] = p[0];
9      for(int i = 0; i < n; i++)
10     {
11         if(sgn((p[i] - a) ^ (p[i + 1] - a)) < 0) return -1;
12         else if(OnSeg(a, Line(p[i], p[i + 1]))) return 0;
13     }
14     return 1;
15 }
16 /**判断点在任意多边形内
17 //射线法，poly[]的顶点数要大于等于3,点的编号0~n-1
18 //返回值
19 //-1:点在凸多边形外
20 //0:点在凸多边形边界上
21 //1:点在凸多边形内
22 int inPoly(Point p, Point poly[], int n)
23 {
24     int cnt;
25     Line ray, side;
26     cnt = 0;
27     ray.s = p;
28     ray.e.y = p.y;
29     ray.e.x = -1000000000000.0; // -INF, 注意取值防止越界
30     for(int i = 0; i < n; i++)
31     {
32         side.s = poly[i];
33         side.e = poly[(i + 1) % n]; /**判断点在任意多边形内
34         if(OnSeg(p, side)) return 0;
35         //如果平行轴则不考虑
36         if(sgn(side.s.y - side.e.y) == 0)
37             continue;
38         if(OnSeg(side.s, ray))
39         {
40             if(sgn(side.s.y - side.e.y) > 0) cnt++;
41         }
42         else if(OnSeg(side.e, ray))
43         {
44             if(sgn(side.e.y - side.s.y) > 0) cnt++;
45         }
46         else if(inter(ray, side))
47             cnt++;
48     }
49     if(cnt % 2 == 1) return 1;
50     else return -1;
51 }
52
53 //判断凸多边形
54 //允许共线边

```

```

55 //点可以是顺时针给出也可以是逆时针给出
56 //点的编号1~n-1
57 bool isconvex(Point poly[], int n)
58 {
59     bool s[3];
60     memset(s, false, sizeof(s));
61     for(int i = 0; i < n; i++)
62     {
63         s[sgn((poly[(i + 1) % n] - poly[i]) ^ (poly[(i + 2) % n] - poly[i])) + 1] = true;
64         if(s[0] && s[2]) return false;
65     }
66     return true;
67 }
68 //多边形的面积：分解为多个三角形的面积和
69 double CalArea(vector<Point> p)
70 {
71     double res = 0;
72     p.push_back(p[0]);
73     for(int i = 1; i < (int)p.size(); i++)
74         res += (p[i] ^ p[i - 1]); //计算由原点, p[i], p[i-1]构成的三角形的有向面积
75     return fabs(res * 0.5); //三角面积需乘以0.5, 以及取正
76 }
77 //多边形的质心：三角质心坐标的面积加权和
78 Point Centroid(vector<Point> p) //按顺时针方向或逆时针方向排列
79 {
80     Point c = Point(0.0, 0.0);
81     double S = 0.0, s;
82     for(int i = 2; i < (int)p.size(); i++)
83     {
84         s = ((p[i] - p[0]) ^ (p[i] - p[i - 1])); //求三角形面积(*0.5)
85         S += s;
86         c = c + (p[0] + p[i - 1] + p[i]) * s; //求三角形质心(/3.0)
87     }
88     c = c * (1.0 / (3.0 * S));
89     return c;
90 }

```

7.4 圆 Circle

```

1 /// 圆
2 /*
3 圆心(x, y), 半径r
4 面积:  $\pi r^2$ , 周长:  $2\pi r$ 
5 弧长:  $\theta r$ 
6 三点圆方程:

```

$$\begin{vmatrix} x^2 + y^2 & x & y & 1 \\ x_1^2 + y_1^2 & x_1 & y_1 & 1 \\ x_2^2 + y_2^2 & x_2 & y_2 & 1 \\ x_3^2 + y_3^2 & x_3 & y_3 & 1 \end{vmatrix} = 0$$

```

7 四边形的内切圆和外切圆:
8     不是所有的四边形都有内切圆, 拥有内切圆的四边形称为圆外切四边形.
9     凸四边形ABCD有内切圆当且仅当两对对边之和相等:  $AB + CD = AD + BC$ .
10    圆外切四边形的面积和内切圆半径的关系为:  $S_{ABCD} = rs$ , 其中s为半周长.
11    同时拥有内切圆和外接圆的四边形称为双心四边形. 这样的四边形有无限多个. 若一个四边形为双心四边形,
12    那么其内切圆在两对对边的切点的连线相互垂直. 而只要在一个圆上选取两条相互垂直的弦,
13    并过相应的顶点做切线, 就能得到一个双心四边形.
14    正多边形必然有内切圆, 而且其内切圆的圆心和外接圆的圆心重合, 都在正多边形的中心. 边长为a
15    的正多边形的内切圆半径为:
16     $r_n = \frac{a}{2} \cot\left(\frac{\pi}{n}\right)$ 
17    其内切圆的面积为:
18     $s_n = \pi r_n^2 = \frac{\pi a^2}{4} \cot^2\left(\frac{\pi}{n}\right)$ 

```

14 | 内切圆面积 s_n 与正多边形的面积 S_n 之比为：

15 |
$$\varphi_n = \frac{s_n}{S_n} = \frac{\frac{\pi a^2}{4} \cot^2(\frac{\pi}{n})}{\frac{\pi a^2}{2} [\frac{1}{2} \cot(\frac{\pi}{n})]} = \frac{\pi}{n} \cot(\frac{\pi}{n})$$

16 | 故此，当正多边形的边数 n 趋向无穷时，

$$\lim_{n \rightarrow \infty} \varphi_n = \lim_{n \rightarrow \infty} \frac{\pi}{n} \cot\left(\frac{\pi}{n}\right) = \lim_{n \rightarrow \infty} \cos^2\left(\frac{\pi}{n}\right) = 1$$

17 | (婆罗摩笈多公式) 若圆内接四边形的四边边长分别是 a, b, c, d ，则其面积为 $S = \sqrt{(p-a)(p-b)(p-c)(p-d)}$ ，

其中 p 为半周长： $p = \frac{a+b+c+d}{2}$ 。在所有周长为定值 $2p$ 的圆内接四边形中，面积最大的是正方形。

18 | 四边形外接圆的半径为， $R = \frac{\sqrt{(ac+bd)(ad+bc)(ab+cd)}}{4S}$

19 | (泰勒斯定理) 若 A, B, C 是圆周上的三点，且 AC 是该圆的直径，那么 $\angle ABC$ 必然为直角。或者说，直径所对的圆周角是直角。

20 | 泰勒斯定理的逆定理同样成立，即：直角三角形中，直角的顶点在以斜边为直径的圆上。

21 | 所有的正多边形都有外接圆，外接圆的圆心和正多边形的中心重合。边长为 a 的 n 边正多边形外接圆的半径为：

$$R_n = \frac{a}{2 \sin(\frac{\pi}{n})} = \frac{a}{2} \csc\left(\frac{\pi}{n}\right)$$

23 | 面积为：

$$A_n = \pi R_n^2 = \frac{\pi a^2}{4 \sin^2(\frac{\pi}{n})} = \frac{\pi a^2}{4} \csc^2\left(\frac{\pi}{n}\right)$$

24 | 正 n 边形的面积 s_n 与其外接圆的面积 A_n 之比为

$$\rho_n = \frac{S_n}{A_n} = \frac{\frac{\pi a^2}{4} \cot^2(\frac{\pi}{n})}{\frac{\pi a^2}{4} \csc^2(\frac{\pi}{n})} = \frac{n}{\pi} \cos\left(\frac{\pi}{n}\right) \sin\left(\frac{\pi}{n}\right) = \frac{n}{2\pi} \sin\left(\frac{2\pi}{n}\right)$$

25 | 故此，当 n 趋向无穷时，

$$\lim_{n \rightarrow \infty} \rho_n = \lim_{n \rightarrow \infty} \frac{n}{2\pi} \sin\left(\frac{2\pi}{n}\right) = 1$$

26 | 另外，其内切圆的面积 S_n 与其外接圆的面积 A_n 之比为：

$$\tau_n = \frac{S_n}{A_n} = \frac{s_n}{S_n} \cdot \frac{S_n}{A_n} = \varphi_n \rho_n = \left[\frac{\pi}{n} \cot\left(\frac{\pi}{n}\right) \right] \left[\frac{n}{\pi} \cos\left(\frac{\pi}{n}\right) \sin\left(\frac{\pi}{n}\right) \right] = \cos^2\left(\frac{\pi}{n}\right)$$

27 | */

28 | // 圆与(直线)线段的相交

29 | // num 表示圆 $O(o, r)$ 与线段 (s, e) 的交点数，res 里存储的是交点

30 | void Circle_cross_Segment(Point s, Point e, Point o, double r, Point res[], int &num)

31 | {

32 | double dx = e.x - s.x, dy = e.y - s.y;

33 | double A = dx * dx + dy * dy;

34 | double B = 2.0 * dx * (s.x - o.x) + 2.0 * dy * (s.y - o.y);

35 | double C = sqr(s.x - o.x) + sqr(s.y - o.y) - r * r;

36 | double delta = B * B - 4.0 * A * C;

37 | num = 0;

38 | if(sgn(delta) < 0) return ;

39 | delta = sqrt(max(0.0, delta));

40 | double k1 = (-B - delta) / (2.0 * A);

41 | double k2 = (-B + delta) / (2.0 * A);

42 | //if(sgn(k1 - 1.0) <= 0 && sgn(k1) >= 0) // 圆与线段相交条件判断

43 | res[num++] = Point(s.x + k1 * dx, s.y + k1 * dy);

44 | //if(sgn(k2 - 1.0) <= 0 && sgn(k2) >= 0)

45 | res[num++] = Point(s.x + k2 * dx, s.y + k2 * dy);

46 | }

7.5 凸包 ConvexHull

1 | // 凸包 Convex Hull

2 | //

3 |

4 | //Graham 算法 $O(n \log n)$

5 | // 写法一：按直角坐标排序

6 | // 直角坐标序比较 (水平序)

7 | bool cmp(const Point &a, const Point &b) // 先比较 x ，后比较 y 均可

```

8 {
9     return a.x < b.x || (a.x == b.x && a.y < b.y);
10 }
11 vector<Point> Graham(vector<Point> p)
12 {
13     int n = p.size();
14     sort(p.begin(), p.end(), cmp);
15     vector<Point> res(n + n + 5);
16     int top = 0;
17     for(int i = 0; i < n; i++)//扫描下凸壳
18     {
19         while(top > 1 && sgn((res[top - 1] - res[top - 2]) ^ (p[i] - res[top - 2])) <= 0) top--;
20         res[top++] = p[i];
21     }
22     int k = top;
23     for(int i = n - 2; i >= 0; i--)//扫描上凸壳
24     {
25         while(top > k && sgn((res[top - 1] - res[top - 2]) ^ (p[i] - res[top - 2])) <= 0) top--;
26         res[top++] = p[i];
27     }
28     if(top > 1) top--;//最后一个点和第一个点一样，可以不去掉，某些计算时方便一些
29     res.resize(top);
30     return res;
31 }
32
33 //写法二：按极坐标排序
34 //可能由于精度问题出现RE
35 Point p0;//p0 原点集中最左下方的点
36 bool cmp(const Point &p1, const Point &p2) //极角排序函数，角度相同则距离小的在前面
37 {
38     double tmp = (p0 - p2) ^ (p1 - p2);
39     if(sgn(tmp) > 0) return true;
40     else if(sgn(tmp) == 0 && sgn(((p0 - p1) * (p0 - p1)) - ((p0 - p2) * (p0 - p2))) < 0) return true;
41     else return false;
42 }
43
44 vector<Point> Graham(vector<Point> p)
45 {
46     //p0
47     int pn = p.size();
48     for(int i = 1; i < pn; i++)
49         if(p[i].x < p[0].x || (p[i].x == p[0].x && p[i].y < p[0].y))
50             swap(p[i], p[0]);
51     p0 = p[0];
52     //sort
53     sort(p.begin() + 1, p.end(), cmp);
54     vector<Point> stk(pn * 2 + 5);
55     int top = 0;
56     for(int i = 0; i < pn; i++)
57     {
58         while(top > 1 && sgn((stk[top - 1] - stk[top - 2]) ^ (p[i] - stk[top - 2])) <= 0) top--;
59         stk[top++] = p[i];
60     }
61     stk.resize(top);
62     return stk;
63 }

```

7.6 半平面交 Half-plane Cross

```

1 //半平面交
2 inline int sgn(double x)

```

```

3 {
4     if(x < -EPS) return -1;
5     return x > EPS ? 1 : 0;
6 }
7 struct Point
8 {
9     double x, y;
10    Point(double xx = 0.0, double yy = 0.0): x(xx), y(yy) {}
11    void in()
12    {
13        scanf("%lf%lf", &x, &y);
14    }
15    Point operator + (Point b)
16    {
17        return Point(x + b.x, y + b.y);
18    }
19    Point operator - (Point b)
20    {
21        return Point(x - b.x, y - b.y);
22    }
23    Point operator *(double b)
24    {
25        return Point(x * b, y * b);
26    }
27    double operator *(Point b)
28    {
29        return x * b.x + y * b.y;
30    }
31    double operator ^(Point b)
32    {
33        return x * b.y - y * b.x;
34    }
35 } p[NUM], ans[NUM];
36 struct Line
37 {
38     Point s, e;
39     double k;
40     Line() {}
41     Line(Point _s, Point _e): s(_s), e(_e) {k = atan2(e.y - s.y, e.x - s.x);}
42     void K()
43     {
44         k = atan2(e.y - s.y, e.x - s.x);
45     }
46     Point operator &(Line &b)
47     {
48         return s + (e - s) * (((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e)));
49     }
50 } L[NUM], que[NUM];
51 int ln;
52
53 inline bool HPIcmp(Line a, Line b)
54 {
55     if(sgn(a.k - b.k) != 0) return sgn(a.k - b.k) < 0;
56     return ((a.s - b.s) ^ (b.e - b.s)) < 0;
57 }
58 void HPI(Line line[], int n, Point res[], int &resn)
59 {
60     int tot = n;
61     sort(line, line + n, HPIcmp);
62     tot = 1;
63     for(int i = 1; i < n; i++)
64     {
65         if(sgn(abs(line[i].k - line[i - 1].k)) != 0)
66             line[tot++] = line[i];

```

```

67     }
68     int head = 0, tail = 1;
69     que[0] = line[0];
70     que[1] = line[1];
71     resn = 0;
72     for(int i = 2; i < tot; i++)
73     {
74         if(sgn((que[tail].e - que[tail].s) ^ (que[tail - 1].e - que[tail - 1].s)) == 0 ||
75            sgn((que[head].e - que[head].s) ^ (que[head + 1].e - que[head + 1].s)) == 0)
76             return ;
77         while(head < tail && sgn(((que[tail]&que[tail - 1]) - line[i].s) ^ (line[i].e - line[i].s))
78            > 0)
79             tail--;
80         while(head < tail && sgn(((que[head]&que[head + 1]) - line[i].s) ^ (line[i].e - line[i].s))
81            > 0)
82             head++;
83         que[++tail] = line[i];
84     }
85     while(head < tail && sgn(((que[tail]&que[tail - 1]) - que[head].s) ^ (que[head].e -
86        que[head].s)) > 0)
87         tail--;
88     while(head < tail && sgn(((que[head]&que[head + 1]) - que[tail].s) ^ (que[tail].e -
89        que[tail].s)) > 0)
90         head++;
91     if(tail <= head + 1) return ;
92     for(int i = head; i < tail; i++)
93         res[resn++] = que[i] & que[i + 1];
94     if(head < tail - 1)
95         res[resn++] = que[head] & que[tail];
96 }

```

7.7 立体几何

```

1  /// 三维几何
2  /// 点
3  inline int sgn(double x) {if(x < -EPS) return -1; return x > EPS ? 1 : 0;}
4  struct Point
5  {
6      double x, y, z;
7      Point(double _x = 0.0, double _y = 0.0, double _z = 0.0) {x = _x, y = _y, z = _z;}
8      void in() {scanf("%lf%lf%lf", &x, &y, &z);}
9      double operator *(const Point b) const {return x * b.x + y * b.y + z * b.z;} // 点积
10     Point operator ^(const Point b) const {return Point(y * b.z - z * b.y, z * b.x - x * b.z, x *
11        b.y - y * b.x);} // 叉积
12     Point operator *(const double b) const {return Point(x * b, y * b, z * b);} // 标量乘
13     Point operator + (const Point b) const {return Point(x + b.x, y + b.y, z + b.z);} // 向量加
14     Point operator - (const Point b) const {return Point(x - b.x, y - b.y, z - b.z);} // 向量减
15     Point rot(double ang){}
16 };
17 /// 判断四点共面 (coplanar)
18 bool is_coplanar(Point p[4])
19 {
20     Point normal = (p[0] - p[1]) ^ (p[1] - p[2]); // 求前三点的法向量
21     return sgn(normal * (p[3] - p[0])) == 0; // 判断与第四点的连线是否与法向量垂直
22 }
23 /// 线
24 /// 面 Plane
25 /// 三点确定一个平面，其法向量 (normal vector) 为：
26 Point getPlaneNormal(Point a, Point b, Point c)
27 {
28     return (a - b) ^ (a - c);

```

```

29 }
30 ///体
31 //四面体体积
32 double V(Point a, Point b, Point c, Point d)
33 {
34     return ((a - b) ^ (a - c)) * (a - d) * (1.0 / 6.0);
35 }
36 //四面体重心
37 Point zhongxin(Point a, Point b, Point c, Point d)
38 {
39     //
40     // Point res;
41     // res.x = (a.x + b.x + c.x + d.x)/4.0;
42     // res.y = (a.y + b.y + c.y + d.y)/4.0;
43     // res.z = (a.z + b.z + c.z + d.z)/4.0;
44     // return res;
45     return (a + b + c + d) * 0.25;
46 }
47 //多面体的重心
48 //将多面体分解为若干个四面体，重心为体积的加权和 zhongxin = (v1*zx1 + v2*zx2 + ... + vn*zxn)/(v1 +
    v2 + ... + v3)
49
50 //空间三角形的面积
51 double Area(Point a, Point b, Point c)
52 {
53     return (a - b) * (a - c);
54 }
55 //点到平面的距离  $h = 3.0 * V / S$ 
56 Point dis(Point p, Point a, Point b, Point c)
57 {
58     return fabs(3.0 * V(p, a, b, c) / S(a, b, c));
59 }

```

7.8 格点 Lattice Point

```

1 //格点(Lattice Point)上的几何
2 //定义：直角坐标系中横纵坐标均为整数的点称为格点(或整点)。
3 /*性质：
4 1. 格点多边形的面积必为整数或半整数(奇数的一半)
5 2. 格点关于格点的对称点为格点
6 3. 格点多边形面积公式(pick公式):
7     格点多边形的面积 $S$  = 多边形内部格点数 $a$  + 边上的格点数 $b/2 + 1$ 
8 4. 格点正多边形只能是正方形
9 5. 格点三角形边界上无其他格点，内部有一个格点，则该点为此三角形的重心。
10 */

```

8 搜索等

```
1 ///二分搜索
2 //对于某些满足单调性质的数列,或函数,可以二分搜索答案,在 $O(\log n)$ 时间内求解
3 //如 $f(x) = 1 (x \leq y) = 0 (x > y)$ , 可以二分搜索出分界值 $y$ 
4 //注意:  $l \% 2 == 0, r = l + 1$ 时,  $(l + r) / 2 == l$  此处易出现死循环
5 int binary_search(int l, int r)
6 {
7     int mid;
8     int ans = l;
9     while(l <= r)
10    {
11        mid = (l + r) >> 1;
12        if(f(mid))
13        {
14            r = mid - 1; //视情况定
15            ans = mid;
16        }
17        else
18            l = mid + 1;
19    }
20    return ans;
21 }
22 ///三分搜索
23 //对于满足抛物线性质的数列或函数,可以三分答案,在 $O(\log n)$ 时间内求解
24 //方便于求(抛物线)的最值
25 //注意:  $l \% 3 == 0, r = l + 1 \mid l + 2$ 时,  $(l + l + r) / 3 == l$  容易出现死循环
26 int three_search(int l, int r)
27 {
28     int ll, rr;
29     while(l + 2 < r)
30    {
31        ll = (l + l + r) / 3;
32        rr = (l + r + r) / 3;
33        if(f(ll) < f(rr))
34            r = rr;
35        else
36            l = ll;
37    }
38    return min(f(l), f(r), f(l + 1));
39 }
```

9 分治

```
1 ///分治
2 //对于某些统计类问题, 可以将问题分为两半, 然后统计跨过两区间的符合条件的数目即可
3 //应用1: 二维偏序求LIS
```


10 Java

```
1 import java.io.*;
2 import java.util.*;
3 import java.math.*;
4 import java.BigInteger;
5
6 public class Main{
7     public static void main(String arg[]) throws Exception{
8         Scanner cin = new Scanner(System.in);
9
10        BigInteger a, b;
11        a = new BigInteger("123");
12        a = cin.nextBigInteger();
13        a.add(b); // a + b
14        a.subtract(b); // a - b
15        a.multiply(b); // a * b
16        a.divide(b); // a / b
17        a.negate(); // -a
18        a.remainder(b); // a % b
19        a.abs(); // |a|
20        a.pow(b); // a^b
21        //.... and other math fuction, like log();
22        a.toString();
23        a.compareTo(b); //
24    }
25 }
26 }
```