

# ACM 模板

dnvtmf

2015

# 目录

1 数据结构	4
1.1 RMQ 相关	4
1.2 Treap	4
1.3 ST 表	5
1.4 最长上升子序列 LIS	6
1.5 并查集 Disjoint Set	6
2 动态规划	7
3 图论	8
3.1 无向图的桥, 割点, 双连通分量	8
3.2 最短路 shortest path	10
3.3 最大流 maximum flow	14
3.4 最小割 minimum cut	18
3.5 分数规划 Fractional Programming	20
3.6 最大闭权图 maximum weight closure of a graph	20
3.7 最小费用最大流 minimum cost flow	21
3.8 有上下界的网络流	25
3.9 最近公共祖先 LCA	25
4 数学专题	27
4.1 素数 Prime	27
4.2 快速幂快速乘	28
4.3 最大公约数 GCD	28
4.4 逆元 Inverse	29
4.5 模运算 Module	30
4.6 中国剩余定理和线性同余方程组	32
4.7 组合与组合恒等式	33
4.8 排列 Permutation	35
4.9 全排列 Full Permutation	36
4.10 母函数 Generating Function	37

4.11 鸽笼原理与 Ramsey 数 . . . . .	38
4.12 容斥原理 . . . . .	39
4.13 伪随机数的生成-梅森旋转算法 . . . . .	39
4.14 异或 Xor . . . . .	40
4.15 博弈论 Game Theory . . . . .	40
4.16 快速傅里叶变换 FFT . . . . .	41
4.17 莫比乌斯反演 Mobius . . . . .	45
4.18 矩阵的基本运算 Matrix . . . . .	47
4.19 一些数学知识 . . . . .	50
5 字符串 . . . . .	51
5.1 回文串 palindrome . . . . .	51
5.2 后缀数组 Suffix Array . . . . .	51
5.3 字典树 Trie . . . . .	54
5.4 字符串问题汇总 . . . . .	55
6 计算几何 . . . . .	57
6.1 计算几何基础 . . . . .	57
6.2 多边形 Polygon . . . . .	59
6.3 凸包 ConvexHull . . . . .	61
6.4 半平面交 Half-plane Cross . . . . .	62
6.5 立体几何 . . . . .	63
6.6 格点 Lattice Point . . . . .	64
7 搜索等 . . . . .	65
8 分治 . . . . .	65
9 Java . . . . .	66

# 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 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,
27    //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.
28    void insert(int tkey, int &k = root)
29    {
30        if(k == -1)
31        {
32            k = ++size;
33            p[k].ch[0] = p[k].ch[1] = -1;
34            p[k].key = tkey;
35            p[k].fix = rand();
36        }
37        else if(tkey < p[k].key)
38        {
39            insert(tkey, p[k].ch[0]);
40            if(p[p[k].ch[0]].fix > p[k].fix)
41                rot(k, 1);
42        }
43    }
```

```

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 };

```

### 1.3 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 }

```

## 1.4 最长上升子序列 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  * 方法:dp
4  * 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 $dp[k] \leq a_i$ ，
   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 /*二维偏序的LIS
8  * 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
9  * 做法：二分+树状数组
10 * 将序列[l, r]二分，先处理左边的区间[l, mid]，
11 * 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，
12 * 遇到在右半区间的查询比当前y值更小的数对数并更新
13 * 然后再递归处理右边的区间[mid+1,r]
14 */
```

## 1.5 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

## 3 图论

### 3.1 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) ≤ 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] ≤ 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
```



```

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 int ce[MAXE], num; //记录桥
102 int id[MAXV], cnt_dcc; //连通分量编号及总数
103 void tarjan(int u, int order, int pre) //重边是pre为边的id, 否则为父亲结点fa
104 {
105     dfs[u] = low[u] = order++;
106     stk[top++] = u; //DCC
107     for(int i = head[u]; i != -1; i = e[i].next)
108     {
109         int v = e[i].v;
110         if(!dfs[v])
111         {
112             tarjan(v, order, i);
113             if(low[v] < low[u])
114                 low[u] = low[v];

```

```

115         if(dfs[u] < low[v])
116         {
117             ce[num++] = i;
118             //DCC
119             cnt_dcc++;
120             do
121             {
122                 id[stk[—top]] = cnt_dcc;
123             }
124             while(stk[top] != v);
125         }
126     }
127     else if((i ^ 1) != pre && dfs[v] < low[u])
128         low[u] = dfs[v];
129 }
130 }
131 void DCC()
132 {
133     memset(dfs, 0, sizeof(dfs));
134     num = 0;
135     top = cnt_dcc = 0; //DCC
136     for(int i = 1; i <= V; i++)
137         tarjan(1, 1, -1);
138 }

```

### 3.2 最短路 shortest path

```

1  ///最短路 Shortest Path
2  //Bellman-Ford算法  $O(|E| \cdot |V|)$ 
3  // $d[v] = \min \{d[u] + w[e]\} (e = \langle u, v \rangle \in E)$ 
4
5  const int MAXV = 1000, MAXE = 1000, INF = 1000000007;
6  struct edge {int u, v, cost;} e[MAXE];
7  int V, E;
8  //graph G
9  int d[MAXV];
10 void Bellman_Ford(int s)
11 {
12     for(int i = 0; i < V; i++)
13         d[i] = INF;
14     d[s] = 0;
15     while(true)
16     {
17         bool update = false;
18         for(int i = 0; i < E; i++)
19         {
20             if(d[e[i].u] != INF && d[e[i].v] > d[e[i].u] + e[i].cost)
21             {
22                 d[e[i].v] = d[e[i].u] + e[i].cost;
23                 update = true;
24             }
25         }
26     }
27 }
28 //判负圈
29 bool find_negative_loop()
30 {
31     memset(d, 0, sizeof(d));
32     for(int i = 0; i < V; i++)
33     {
34         for(int j = 0; j < E; j++)
35         {

```

```

36         if(d[e[j].v] > d[e[i].u] + e[j].cost)
37         {
38             d[e[j].v] = d[e[j].u] + e[j].cost;
39             if(i == V - 1)
40                 return true;
41             //循环了V次后还不能收敛，即存在负圈
42         }
43     }
44 }
45 return false;
46 }
47
48 //spfa算法  $O(|E| \log |V|)$ 
49 //适用于负权图和稀疏图，稳定性不如dijkstra
50 //存在负环返回false
51 int d[MAXV], outque[MAXV];
52 bool vis[MAXV];
53 bool spfa(int s)
54 {
55     for(int i = 0; i < V; i++)
56     {
57         vis[i] = false;
58         d[i] = INF;
59         outque[i] = 0;
60     }
61     d[s] = 0;
62     queue<int> que;
63     que.push(s);
64     vis[s] = true;
65     while(!que.empty())
66     {
67         int u = que.front();
68         que.pop();
69         vis[u] = false;
70         if(++outque[u] > V) return false;;
71         for(int i = head[u]; i != -1; i = e[i].next)
72         {
73             int v = e[i].to;
74             if(d[v] > d[u] + e[i].w)
75             {
76                 d[v] = d[u] + e[i].w;
77                 if(!vis[v])
78                 {
79                     vis[v] = true;
80                     que.push(v);
81                 }
82             }
83         }
84     }
85     return true;
86 }
87
88 //dijkstra算法 $O(|V|^2)$ 
89 int cost[MAXV][MAXV];
90 int d[MAXV];
91 bool vis[MAXV];
92 void dijkstra(int s)
93 {
94     fill(d, d + V, INF);
95     memset(vis, 0, sizeof(vis));
96     d[s] = 0;
97     while(true)
98     {
99         int v = -1;

```

```

100     for(int u = 0; u < V; u++)
101         if(!vis[u] && (v == -1 || d[u] < d[v]))
102             v = u;
103     if(v == -1) break;
104     for(int u = 0; u < V; u++)
105         d[u] = min(d[u], d[v] + cost[v][u]);
106 }
107 }
108
109 //dijkstra算法  $O(|E| \log |V|)$ 
110 struct edge {int v, cost;};
111 vector<edge> g[MAXV];
112 int d[MAXV];
113
114 void dijkstra(int s)
115 {
116     priority_queue<P, vector<P>, greater<P> > que;
117     fill(d, d + V, INF);
118     d[s] = 0;
119     que.push(P(0, s));
120     while(!que.empty())
121     {
122         P p = que.top(); que.pop();
123         int u = p.second;
124         if(d[u] < p.first) continue;
125         for(int i = 0; i < g[u].size(); i++)
126         {
127             edge &e = g[u][i];
128             if(d[e.v] > d[u] + e.cost)
129             {
130                 d[e.v] = d[u] + e.cost;
131                 que.push(P(d[e.v], e.v));
132             }
133         }
134     }
135 }
136
137 ///任意两点间最短路
138 //Floyd-Warshall算法  $O(|V|^3)$ 
139 int d[MAXV][MAXV];
140 void floyd_warshall(int V)
141 {
142     int i, j, k;
143     for(k = 0; k < V; k++)
144         for(i = 0; i < V; i++)
145             for(j = 0; j < V; j++)
146                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
147 }
148
149 ///两点间最短路 — 一条可行路径还原
150 /*用prev[u]记录从s到u的最短路上u的前驱结点*/
151 vector<int> get_path(int t)
152 {
153     vector<int> path;
154     for(; t != -1; t = prev[t])
155         path.push_back(t);
156     reverse(path.begin(), path.end());
157     return path;
158 }
159
160 ///两点间最短路 — 所有可行路径还原
161 /*如果无重边，从终点t反向dfs，将所有满足d[u] + e.w = e[v]的边e(u,v)加入路径中即可  $O(|E|)$ 
162 其他情况，在计算最短路时，将源点s到其他所有点的最短路加入最短路逆图中，然后从终点t反向bfs，
    标记所有经过的点，最后将所有连接到非标记点的边去掉即可

```

```

163 */
164 //情况1
165 int vis[MAXV];
166 vector<edge> G[MAXV];
167 void add_edge() {}
168 void get_pathG(int u)
169 {
170     vis[u] = 1;
171     for(int i = 0; i < g[u].size(); i++)
172     {
173         int v = g[u][i].v, w = g[u][i].w;
174         if(d[v] + w == d[u])
175         {
176             add_edge(u, v);
177             add_edge(v, u);
178             if(!vis[v])
179                 get_pathG(v);
180         }
181     }
182 }
183 //情况2
184 struct edge
185 {
186     //...
187 } e[MAXE];
188 int head[MAXV], tot;
189 vector<int> g[MAXV]; //所有最短路形成的逆图
190 int vis[MAXV];
191 void dijkstra(int s)
192 {
193     //... ... other part see above
194     for(int i = head[u]; i != -1; i = e[i].next)
195     {
196         int v = e[i].to;
197         if(d[v] > d[u] + e[i].w)
198         {
199             g[v].clear();
200             g[v].push_back(u);
201             d[v] = d[u] + e[i].w;
202             que.push(P(d[v], v));
203         }
204         else if(d[v] == d[u] + e[i].w)
205         {
206             g[v].push_back(u);
207         }
208     }
209 }
210
211 void get_all_path(int s, int t)
212 {
213     memset(vis, 0, sizeof(vis));
214     queue<int> que;
215     que.push(t);
216     vis[t] = 1;
217     while(!que.empty())
218     {
219         int u = que.front();
220         que.pop();
221         for(int i = 0; i < g[u].size(); i++)
222             if(!vis[g[u][i]])
223             {
224                 vis[g[u][i]] = 1;
225                 que.push(g[u][i]);
226             }

```

```

227     }
228     for(int i = 1; i <= V; i++)
229         if(!vis[i])
230         {
231             g[i].clear();//清空不是路径上的点
232         }
233 }
234 ///求最短路网络上的桥
235 //方法1: 将最短路网络新建一个图, 跑Tarjan算法.
236 //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
237 //求割点类似
238 int vis[MAXV];
239 int ce[MAXE], num;
240 void get_bridge(int s, int t)//在逆图中运行
241 {
242     priority_queue<P> que;
243     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
244     memset(vis, 0, sizeof(vis));
245     vis[t] = 1;
246     num = 0;
247     while(!que.empty())
248     {
249         int u = que.top().SE;
250         que.pop();
251         int cnt = 0;
252         if(que.empty())
253         {
254             for(int i = rhead[u]; i != -1; i = re[i].next)
255             {
256                 int v = re[i].to, w = re[i].cost;
257                 if(dist[v] + w == dist[u])
258                 {
259                     cnt++;
260                     if(cnt >= 2) break;
261                 }
262             }
263         }
264         bool f = que.empty() && cnt == 1;//当且仅当, 队列为空且只有一条路时, 找到桥
265         for(int i = rhead[u]; i != -1; i = re[i].next)
266         {
267             int v = re[i].to, w = re[i].cost;
268             if(dist[v] + w == dist[u])
269             {
270                 if(f) ce[num++] = i;
271                 if(!vis[v])
272                 {
273                     vis[v] = 1;
274                     que.push(P(dist[v], v));
275                 }
276             }
277         }
278     }
279 }

```

### 3.3 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 }

```

### 3.4 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 最小割的求法：
14     1. 先求的最大流
15     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 $S$ 
16     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。
17 */
18 int max_flow(int s, int t) {}
19 int getST(int s, int t, int nd[])
20 {
21     int mincap = max_flow(s, t);
22     memset(nd, 0, sizeof(nd));
23     queue<int> que;
24     que.push(s);
25     vis[s] = 1;
26     while(!que.empty())
27     {
28         int u = que.front(); que.pop();
29         for(int i = 0; i < (int)g[u].size(); i++)//travel v
30             if(g[u][i].cap > 0 && !vis[g[u][i].to])
31             {
32                 vis[g[u][i].to] = 1;
33                 que.push(g[u][i].to);
34             }
35     }
36 }
37 ///无向图全局最小割 Stoer-Wagner算法
38 /*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)
39 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
40 快速求当前图某两点的最小割：
41     1. 维护一个集合 $A$ ，初始里面只有 $v_1$ (可以任意)这个点
42     2. 区一个最大的 $w(A, y)$ 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)
43     3. 反复2，直至 $A$ 集合 $G$ 集相等
44     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
45 */
46 //O( $|V|^3$ )
47 const int MAXV = 510;
48 int n;
49 int g[MAXV][MAXV]; //g[u][v]表示u,v两点间的最大流量
50 int dist[MAXV]; //集合A到其他点的距离
51 int vis[MAXV];
52 int min_cut_phase(int &s, int &t, int mark) //求某两点间的最小割
53 {
54     vis[t] = mark;
55     while(true)
56     {
57         int u = -1;
58         for(int i = 1; i <= n; i++)
59             if(vis[i] < mark && (u == -1 || dist[i] > dist[u])) u = i;
60         if(u == -1) return dist[t];
61         s = t, t = u;
62         vis[t] = mark;
63         for(int i = 1; i <= n; i++) if(vis[i] < mark) dist[i] += g[t][i];

```

```

64     }
65 }
66
67 int min_cut()
68 {
69     int i, j, res = INF, x, y = 1;
70     for(i = 1; i <= n; i++)
71         dist[i] = g[1][i], vis[i] = 0;
72     for(i = 1; i < n; i++)
73     {
74         res = min(res, min_cut_phase(x, y, i));
75         if(res == 0) return res;
76         //merge x, y
77         for(j = 1; j <= n; j++) if(vis[j] < n) dist[j] = g[j][y] = g[y][j] = g[x][j] + g[x][j];
78         vis[x] = n;
79     }
80     return res;
81 }

```

### 3.5 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:

```

$$\begin{cases} g(\lambda) = 0 \Leftrightarrow \lambda = \lambda^* \\ g(\lambda) < 0 \Leftrightarrow \lambda > \lambda^* \\ g(\lambda) > 0 \Leftrightarrow \lambda < \lambda^* \end{cases}$$

```

11     由该性质, 就可以对最优解  $\lambda^*$  进行二分查找.
12     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.
13 */
14 ///0-1分数规划 0-1 fractional programming
15 /*是分数规划的解向量  $\vec{x}$  满足  $\forall x_i \in \{0, 1\}$ , 即一下形式:

```

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

```

16     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,
17     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  $e \notin C$ .
18 */

```

### 3.6 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  $\sum_{v \in V'} w_v$ .

```

```

4 */
5 /*转化为最小割模型 $G_N(V_N, E_N)$ 
6 在原图点集的基础上增加源 $s$ 和汇 $t$ ;
    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 $\langle u, v \rangle \in E_N$ ;
    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 $c(s, v) = w_v$ ,
    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,
    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:

```

$$V_N = V \cup \{s, t\}$$

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

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

```

7 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.
8 */

```

### 3.7 最小费用最大流 minimum cost flow

```

1 ///最小费用最大流 miniumm 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 }

```

```

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 }

```

```

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 }

```



### 3.8 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).
10 */
```

### 3.9 最近公共祖先 LCA

```
1 ///最近公共祖先LCA Least Common Ancestors
2 //Tarjian的离线算法  $O(n+q)$ 
3 struct edge {int next, to, lca;};
4 //由要查询的<u,v>构成的图
5 edge qe[MAXE * 2];
6 int qh[MAXV], qtot;
7 //原图
8 edge e[MAXE * 2];
9 int head[MAXV], tot;
10 //并查集
11 int fa[MAXV];
12 inline int find(int x)
13 {
14     if(fa[x] != x) fa[x] = find(fa[x]);
15     return fa[x];
16 }
17 bool vis[MAXV];
18 void LCA(int u)
19 {
20     vis[u] = true;
21     fa[u] = u;
22     for(int i = head[u]; i != -1; i = e[i].next)
23         if(!vis[e[i].to])
24         {
25             LCA(e[i].to);
26             fa[e[i].to] = u;
27         }
28     for(int i = qh[u]; i != -1; i = qe[i].next)
29         if(vis[qe[i].to])
30         {
31             qe[i].lca = find(eq[i].to);
32             eq[i ^ 1].lca = qe[i].lca; //无向图, 入边两次
33         }
34 }
35
36 //RMQ的在线算法  $O(n \log n)$ 
37 /*算法描述:
38 dfs扫描一遍整棵树,
   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
39 再记录下第一次扫描到结点u时的序号
40 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].
41 */
42 struct node
43 {
```

```

44     int u; //记录经过的结点
45     int depth; //记录当前结点的深度
46 } vs[2 * MAXV];
47 bool operator < (node a, node b) {return a.depth < b.depth;}
48 int id[MAXV]; //记录第一次经过点u时的dfn序号
49 void dfs(int u, int fa, int dep, int &k)
50 {
51     vs[k] = (node) {u, dep};
52     id[u] = k++;
53     for(int i = head[u]; i != -1; i = e[i].next)
54         if(e[i].to != fa)
55         {
56             dfs(e[i].to, u, dep + 1, k);
57             vs[k++] = (node) {u, dep};
58         }
59 }
60 //RMQ
61 //动态查询id[u] 到 id[v] 之间的depth最小的结点
62 //ST表
63 int Log2[MAXV * 2];
64 node st[MAXV * 2][32];
65 void pre_st(int n, node ar[])
66 {
67     Log2[1] = 0;
68     for(int i = 2; i <= n; i++) Log2[i] = Log2[i >> 1] + 1;
69     for(int i = n - 1; i >= 0; i--)
70     {
71         st[i][0] = ar[i];
72         for(int j = 1; i + (1 << j) <= n; j++)
73             st[i][j] = min(st[i][j - 1], st[i + (1 << j) - 1][j - 1]);
74     }
75 }
76 int query(int l, int r)
77 {
78     int k = Log2[r - l + 1];
79     return min(st[l][k], st[r - (1 << k) + 1][k]).u;
80 }
81
82 void lca_init()
83 {
84     int k = 0;
85     dfs(1, -1, 0, k);
86     pre_st(k, vs);
87 }
88
89 int LCA(int u, int v)
90 {
91     u = id[u], v = id[v];
92     if(u > v) swap(u, v);
93     return query(u, v);
94 }

```

## 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             flag[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 //witness定理:
48 //Miller_Rabin O(log n)
49 int qpow(int x, int k, int mod){}
50 bool witness(int a, int n)
51 {
52     int t = 0, u = n - 1;
53     while((u & 1) == 0) t++, u >>= 1;
54     int x = qpow(a, u, n), lx;
55     for(int i = 1; i <= t; i++)
56     {
57         lx = x;
58         x = x * x % n;
```

```

58     if(x == 1 && lx != 1 && lx != n - 1)
59         return true;
60 }
61 return x != 1;
62 }
63 bool Miller_Rabin(int n)//出错率为 $(\frac{1}{2})^{-s}$ 
64 {
65     if(n<2) return false;
66     if(n == 2 || n == 3 || n == 5 || n == 7) return true;
67     if(n % 2 == 0 || n % 3 == 0 || n % 5 == 0 || n % 7 == 0) return false;
68     int s = 50;
69     while(s—)
70         if(witness(rand() % (n - 1) + 1, n))
71             return false;
72     return true;
73 }

```

## 4.2 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 }
13
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 }

```

## 4.3 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 ∈ 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  $lcm(a, b) = \frac{a \cdot b}{gcd(a, b)}$ 
7 | */
8 | //欧几里得算法  $O(\log n)$ 
9 | //递归
10 | int gcd(int a, int b) {return b ? gcd(b, a % b) : a;}
11 | //循环
12 | int gcd(int a, int b)
13 | {
14 |     int t;
15 |     while(b) t = a % b, a = b, b = t;
16 |     return a;
17 | }
18 | //小数的gcd
19 | //EPS控制精度
20 | double fgcd(double a, double b)
21 | {
22 |     if(-EPS < b && b < EPS) return a;
23 |     return fgcd(b, fmod(a, b));
24 | }
25 | ///扩展欧几里得算法
26 | void exgcd(int a, int b, int &g, int &x, int &y)
27 | {
28 |     if(b) exgcd(b, a % b, g, y, x), y -= a / b * x;
29 |     else x = 1, y = 0, g = a;
30 | }
31 | //应用
32 | //1. 求解  $ax + by = c$  的  $x$  的最小正整数解
33 | int solve(int a, int b, int c)
34 | {
35 |     int g = exgcd(a, b, x, y), t = b / g;
36 |     if(c % g) return -1; //  $c \% gcd(a, b) \neq 0$  无解
37 |     int x0 = x * c / g;
38 |     x0 = ((x0 % t) + t) % t;
39 |     int y0 = c - a * x0;
40 |     return x0;
41 | }
42 | //2. 求解  $a$  关于  $p$  的逆元
43 | //

```

## 4.4 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  $inv[1] \sim inv[n-1]$ , 求  $inv[n]$ , ( $MOD > n$ ) ( $MOD$ 为质数, 不存在逆元的  $i$  干扰结果)
45 //  $MOD = x \cdot n - y (0 \leq y < n) \Rightarrow x \cdot n = y \% MOD \Rightarrow x \cdot n \cdot inv[y] = y \cdot inv[y] = 1 \% MOD$ 
46 //所以  $inv[n] = x \cdot inv[y] (x = MOD - MOD/n, y = MOD \% n)$ 
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  $O(\log n)$ 

```

## 4.5 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 Ddivide:  $(a / b) \% p = (a * b^{-1}) \% p$ ,  $b^{-1}$ 是 $b$ 关于 $p$ 的逆元
8 Power:  $(a^b) \% p = ((a \% p)^b) \% p$ 
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  $a \equiv b \% p, c \equiv d \% p$ , 则  $(a + c) \equiv (b + d) \% p, (a - c) \equiv (b - d) \% p, (a * c) \equiv (b * d) \% p, (a / c) \equiv (b / d) \% p$ 

```

```

13
14 3. 费马小定理
15 若p是素数, 对任意正整数x, 有  $x^p \equiv x(\%p)$ .
16 4. 欧拉定理
17 若p与x互素, 则有  $x^{\phi(p)} \equiv 1(\%p)$ .
18 5.  $n! = ap^e, \gcd(a, p) = 1, p$ 是素数
19  $e = (n/p + n/p^2 + n/p^3 + \dots)$  (a不能被p整除)
20 威尔逊定理:  $(p-1)! \equiv -1(\%p)$  (当且仅当p是素数)
21  $n!$ 中不能被p整除的数的积:  $n! = (p-1)!^{(n/p)} \times (n \bmod p)!$ 
22  $n!$ 中能被p整除的项:  $p, 2p, 3p, \dots, (n/p)p$ , 除以p得到  $1, 2, 3, \dots, n/p$  (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:  $mod = p_1^{c_1} p_2^{c_2} \dots p_k^{c_k}$ 
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 = p1c1 p2c2 ... pkck
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 Cnk % pc
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: Cnm % mod = Cn/modm/mod · Cn%modm%mod % mod
114 采用O(n)方法预处理0~n-1的n!%mod和每个数的逆元, 则可在O(log n)时间求出Cnk % 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 }

```

## 4.6 中国剩余定理和线性同余方程组

```

1 /*线性同余方程
2 ai × x ≡ bi (% mi) (1 ≤ i ≤ 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 ≡ b (% m) 的形式.
4 对方程逐一求解. 令 b = 0, m = 1;
5 1. x ≡ b (% m) 可写为 x = b + m · t;
6 2. 带入第i个式子: ai(b + m · t) ≡ bi (% mi), 即 ai · m · t ≡ bi - ai · b (% mi)

```



```

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 | }

```

## 4.7 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 $C_n^r$ :
2 | 
$$C_n^r = \begin{cases} \frac{n!}{r!(n-r)!}, & n \geq r \\ 1, & n \geq r = 0 \\ 0, & n < r \end{cases}$$

3 | 推论1:  $C_n^r = C_n^{n-r}$ 
4 | 推论2(Pascal公式):  $C_n^r = C_{n-1}^r + C_{n-1}^{r-1}$ 
5 | 推论3:  $\sum_{k=r-1}^{n-1} C_k^{r-1} = C_{n-1}^{r-1} + C_{n-2}^{r-2} + \dots + C_{r-1}^{r-1} = C_n^r$ 
6 | 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 $F(n, r) = C_{n+r-1}^r$ 
7 |
8 | 3. 二项式定义
9 | 当n是一个正整数时, 对任何x和y有:
10 |

```

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

```

11 | 令y=1, 有:
12 | 
$$(1+x)^n = \sum_{k=0}^n C_n^k x^k = \sum_{k=0}^n C_n^{n-k} x^k$$

13 | 广义二项式定理:
14 | 广义二项式系数: 对于任何实数 $\alpha$ 和整数k, 有
15 |

```

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

```

16 | 设 $\alpha$ 是一个任意实数, 则对满足 $|\frac{x}{y}| < 1$ 的所有x和y, 有
17 |

```

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

```

18 | 推论: 令 $z = \frac{x}{y}$ , 则有
19 |

```

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

20 | 令  $\alpha = -n$  ( $n$  是正整数), 有  
21 |

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

22 | 又令  $z = -rz$ , ( $r$  为非零常数), 有  
23 | 又令  $n=1$ , 有  
24 |

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

25 | 令  $z = -z$ , 有  
26 |

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

27 | 令  $\alpha = \frac{1}{2}$ , 有  
28 |

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

29 | 4. 组合恒等式  
30 |

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  $\alpha$  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 | \*/

## 4.8 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

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

13 | \*

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

15 | \* 错排数

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

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 | \*

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

26 | \* 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:

27 |

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

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 | \*

36 | \* 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.

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 | \*

38 | \*  $n$ 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。

\*  $n$ 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 | \*/

## 4.9 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 something
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)!)/(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,
46     4第2小的数; 0/1!=0...0, p[2]=2{2,4第1小的数}; 0/0!=0...0,
47     p[3]=4{4第1小的数}; 故第3个的排列为{1,3,2,4}
48 int fac[] = {1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880};
49 int perm[9];
50 void unContor(int pos, int n)
51 {
52     int i, j;
53     bool vis[12] = {0};
54     pos--;
55     for(i = 0; i < n; i++)
56     {
57         int t = sum / fac[n - i - 1];
58         for(j = 1; t && j <= n; j++)
59             if(!vis[j]) t--;
60         vis[perm[i] = j] = 1;
61         sum %= fac[n - i - 1];
62     }
63 }

```

## 4.10 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(a0, a1, a2, ..., an, ...) (简记为{an}), 称函数
4  *

```

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

```

5  * 为序列{an}的普通母函数
6  * 常见普通母函数:
7  * 序列(Cn0, Cn1, Cn2, ..., Cnn)的普通母函数为f(x) = (1+x)n
8  * 序列(1, 1, ..., 1, ...)的普通母函数为f(x) =  $\frac{1}{1-x}$ 
9  * 序列(Cn-10, -Cn1, Cn+12, ..., (-1)kCn+k-1k, ...)的普通母函数为f(x) = (1+x)-n
10 * 序列(C00, C21, C42, ..., C2nn, ...)的普通母函数为f(x) = (1-4x)-1/2
11 * 序列(0, 1×2×3, 2×3×4, ..., n×(n+1)×(n+2), ...)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

```

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

```

15 为序列(a0, a1, ..., an, ...)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 $(P_0^0, P_2^1, P_4^2, \dots, P_{2n}^n)$ 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 \*/

## 4.11 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 $(i = 1, 2, \dots, n)$ ,  $q \geq q_1 + q_2 + \dots + q_n - n + 1$ ,  
 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  $\frac{\sum_{i=1}^n m_i}{n} > r - 1$ ,  
 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 $a_{k+1} + a_{k+2} + \dots + a_l$ 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 $N(a, b) \leq N(a-1, b) + N(a, b-1) - 1$

$N(a, b)$	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 |     */

```

## 4.12 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 | *  $|\overline{A_1} \cap \overline{A_2} \cap \dots \cap \overline{A_m}| = |S| - \sum_{i=1}^m |A_i| + \sum_{i \neq j} |A_i \cap A_j| - \sum_{i \neq j \neq k} |A_i \cap A_j \cap A_k| + \dots + (-1)^m |A_1 \cap A_2 \cap \dots \cap A_m|$ 
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 | * 在求出满足自少含  $k_i + 1$  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 | */

```

## 4.13 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 }

```

## 4.14 异或 Xor

```

1  ///异或Xor
2  /*
3  性质:   1. 0 xor 1 = 0, 1 xor 0 = 1, 0 xor 0 = 0, 1 xor 1 = 1
4          2. (交换律) a xor b = b xor a
5          3. (结合律) (a xor b) xor c = a xor (b xor c)
6          4. a xor a = 0
7          5. 0 xor a = a
8
9          6. (二进制分解)  $a \text{ xor } b = \sum_{i=0}^{\infty} a_i \text{ xor } b_i$ , 其中  $a_i, b_i$  是数  $a, b$  的二进制表达的第  $i$  位
10
11         不同位置上运算互不影响
12         7. 若  $a$  为偶数, 则  $a \text{ xor } (a + 1) = 1$ ,  $a \text{ xor } 1 = (a + 1)$ ,  $(a + 1) \text{ xor } 1 = a$ 
13 */

```

## 4.15 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,
19             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.
20 */
21 SG函数(the Sprague-Grundy function)
22 定义: 游戏状态为  $x$ ,  $sg(x)$  表示状态  $x$  的  $sg$  函数值,  $sg(x) = \min\{n | n \in N \cap n \notin F(x)\}$ ,
23        $F(x)$  表示  $x$  能够达到的所有状态. 一个状态为必败态则  $sg(x)=0$ .
24  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  $sg_i$ ,
25       则游戏  $G$  的  $sg$  函数值为  $g = sg_1 \wedge sg_2 \wedge \dots \wedge sg_n$ 
26  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.
27 */
28 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,
29           取走最后一个石子的人胜利.
30  $SG(i) = X(i = 1, 2, \dots, N)$ , 当且仅当  $SG(tot) = SG(1) \text{ xor } SG(2) \text{ xor } \dots \text{ xor } SG(N)$ 
31 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,
32       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 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。

29 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，

30 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 \*/

## 4.16 快速傅里叶变换 FFT

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) · DFT(b), 所以a(\*)b = DFT<sup>-1</sup>(DFT(a) · 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 | //在整数域上的FFT
90 | /*在整数域上的FFT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

```

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

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

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

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 $\varpi$ ，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  $(1 < K) * i + 1$  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 fft\_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 FFT(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 }

```

## 4.17 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 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) · f(b) = f(a · 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(x) + g(x), f(x · g) = f(x) · g(x)。
35 | // 一个比较显然的性质: 若f, g均为积性函数, 则f · g也是积性函数。
36 | // 积性函数的求值: n = ∏ p_i^{a_i} 则f(n) = ∏ 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ε(n) = (n == 1), (这三个都是显然为积性)
42 | // 欧拉函数φ(n) (只要证两个集合相等就能证明积性)
43 | // 莫比乌斯函数μ(n) (由定义也是显然的)
44 | // μ · id = φ
45 | void pre_mobius()
46 | {
47 |     mu[1] = 1;
48 |     for(int i = 2; i < NUM; i++)
49 |         if(!mu[i])
50 |         {
51 |             mu[i] = -1;
52 |             for(int j = i + i; j < NUM; j += i)
53 |                 if((j / i) % i == 0)
54 |                     mu[j] = 2;
55 |             else
56 |             {
57 |                 if(mu[j] == 0) mu[j] = -1;
58 |                 else mu[j] = -mu[j];

```

```

59     }
60 }
61 else if(mu[i] == 2 || mu[i] == -2) mu[i] = 0;
62 }

```

## 4.18 矩阵的基本运算 Matrix

```

1  ///矩阵 Matrix
2  const int Matrix_N = 1010, Matrix_M = 1010;
3  //矩阵类, 适用于递推关系式的快速求值
4  struct Matrix
5  {
6      int n, m; //矩阵的行数和列数
7      int a[Matrix_N][Matrix_M];
8      void clear() //将矩阵清0
9      {
10         n = m = 0;
11         memset(a, 0, sizeof(a));
12     }
13     void I() //将矩阵化为单位矩阵, 对方阵有效
14     {
15         memset(a, 0, sizeof(a));
16         for(int i = 0; i < n; i++) a[i][i] = 1;
17     }
18     //实现矩阵加法  $C = A + B, (A.n = B.n, A.m = B.m) O(n^2)$ 
19     Matrix operator + (const Matrix &b) const
20     {
21         Matrix c;
22         c.n = n;
23         c.m = m;
24         for(int i = 0; i < n; i++)
25             for(int j = 0; j < m; j++)
26                 c.a[i][j] = a[i][j] + b.a[i][j];
27         return c;
28     }
29     //实现矩阵的减法  $C = A - B (A.n = B.n, A.m = B.m), O(n^2)$ 
30     Matrix operator - (const Matrix &b) const
31     {
32         Matrix c;
33         c.n = n;
34         c.m = m;
35         for(int i = 0; i < n; i++)
36             for(int j = 0; j < m; j++)
37                 c.a[i][j] = a[i][j] - b.a[i][j];
38         return c;
39     }
40     //实现矩阵的乘法  $C = A \times B, (A.m = B.n) O(n^3)$ 
41     Matrix operator * (const Matrix &b) const
42     {
43         Matrix c;
44         c.n = n;
45         c.m = b.m;
46         for(int i = 0; i < n; i++)
47             for(int j = 0; j < b.m; j++)
48             {
49                 c.a[i][j] = 0;
50                 for(int k = 0; k < m; k++)
51                     c.a[i][j] += a[i][k] * b.a[k][j]; //如果要取模, 要修改这里
52             }
53         return c;
54     }
55     //实现矩阵的快速幂  $O(n^3 \log(k))$ , 要求是方阵

```

```

56     Matrix operator ^(int k)
57     {
58         Matrix res, tmp = *this;
59         res.n = res.m = n;
60         res.I();
61         while(k)
62         {
63             if(k&1) res = res * tmp;
64             tmp = tmp * tmp;
65             k >>= 1;
66         }
67         return res;
68     }
69 };
70 //矩阵类 适用与求矩阵的逆与高斯消元等场合
71 //行的初等变换
72 typedef vector<double> VD;
73 VD operator * (const VD &a, const double b)
74 {
75     int _n = a.size();
76     VD c(_n);
77     for(int i = 0; i < _n; i++)
78         c[i] = a[i] * b;
79     return c;
80 }
81 VD operator - (const VD &a, const VD &b)
82 {
83     int _n = a.size();
84     VD c(_n);
85     for(int i = 0; i < _n; i++)
86         c[i] = a[i] - b[i];
87     return c;
88 }
89 VD operator + (const VD &a, const VD &b)
90 {
91     int _n = a.size();
92     VD c(_n);
93     for(int i = 0; i < _n; i++)
94         c[i] = a[i] + b[i];
95     return c;
96 }
97 struct Matrix
98 {
99     int n, m;
100     VD *a;
101     void Matrix(int _n = Matrix_N, int _m = Matrix_M)
102     {
103         n = _n, m = _m;
104         a = new VD[n];
105         for(int i = 0; i < n; i++)
106             a[i].resize(m, 0);
107     }
108     void ~Matrix()
109     {
110         delete []a;
111     }
112     void clear()//0矩阵
113     {
114         for(int i = 0; i < n; i++)
115             a[i].assign(0);
116     }
117     void I()//单位矩阵
118     {
119         clear();

```



```

120     for(int i = 0; i < n; i++)
121         a[i][i] = 1;
122 }
123 //矩阵加法, 同上
124 Matrix operator + (const Matrix &b) const
125 {
126     Matrix c(n, m);
127     for(int i = 0; i < n; i++)
128         c.a[i] = a[i] + b.a[i];
129     return c;
130 }
131 Matrix operator - (const Matrix &b) const //矩阵减法
132 {
133     Matrix c(n, m);
134     for(int i = 0; i < n; i++)
135         c.a[i] = a[i] - b.a[i];
136     return c;
137 }
138 Matrix operator * (const Matrix &b) const //矩阵乘
139 {
140     Matrix c(n, b.m);
141     for(int i = 0; i < n; i++)
142         for(int j = 0; j < b.m; j++)
143         {
144             c[i][j] = 0;
145             for(int k = 0; k < m; k++)
146                 c.a[i][j] += a[i][k] * b.a[k][j];
147         }
148     return c;
149 }
150
151 //实现求矩阵的逆 $O(n^3)$ 
152 //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,
153 //将会得到(I, A-1)的形式
154 //注意:
155 Matrix inverse()
156 {
157     Matrix c;
158     c.I();
159     for(int i = 0; i < n; i++)
160     {
161         for(int j = i; j < n; j++)
162             if(fabs(a[j][i]) > 0)
163             {
164                 swap(a[i], a[j]);
165                 swap(c[i], c[j]);
166                 break;
167             }
168         c[i] = c[i] * (1.0 / a[i][i]);
169         a[i] = a[i] * (1.0 / a[i][i]);
170         for(int j = 0; j < n; j++)
171             if(j != i && fabs(a[j][i]) > 0)
172             {
173                 c[j] = c[j] - a[i] * a[j][i];
174                 a[j] = a[j] - a[i] * a[j][i];
175             }
176     }
177 };
178 //Guass消元
179 int Guass(double a[][MAXN], bool l[], double ans[], int n)
180 { //l, ans储存解, l[]表示是否是自由元
181     int res = 0, r = 0;
182     for(int i = 0; i < n; i++) l[i] = false;

```

```

183     for(int i = 0; i < n; i++)
184     {
185         for(int j = r; j < n; j++)
186             if(fabs(a[j][i]) > EPS)
187             {
188                 for(int k = i; k <= n; k++)
189                     swap(a[j][k], a[r][k]);
190                 break;
191             }
192         if(fabs(a[r][i]) < EPS)
193         {
194             ++res;
195             continue;
196         }
197         for(int j = 0; j < n; j++)
198             if(j != r && fabs(a[j][i]) > EPS)
199             {
200                 double tmp = a[j][i] / a[r][i];
201                 for(int k = i; k <= n; k++)
202                     a[j][k] -= tmp * a[r][k];
203             }
204         l[i] = true;
205         ++r;
206     }
207     for(int i = 0; i < n; i++)
208         if(l[i])
209             for(int j = 0; j < n; j++)
210                 if(fabs(a[j][i]) > 0)
211                     ans[i] = a[j][n] / a[j][i];
212     return res; //返回解空间的维数
213 }
214 //常系数线性齐次递推
215 /*已知 $f_x = a_0 f_{x-1} + a_1 f_{x-2} + \dots + a_{n-1} f_{x-n}$ 和 $f_0, f_1, \dots, f_{n-1}$ , 给定 $t$ , 求 $f_t$ 
216  $f$ 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
217

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

218 则 $\beta_t = A^{t-n+1} \beta_0 (t \geq n)$ 
219 */

```

## 4.19 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： $a_i = i \text{ xor } (i >> 1)$ ( $a_i$ 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 $0 \leq x < \frac{p}{2}, 0 \leq y < \frac{p}{2}$

## 5 字符串

### 5.1 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 }
```

### 5.2 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 }
```

```

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:  $O(n \log n)$  采用基数排数
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]] = r[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]);
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 commonest 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→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.
    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,
    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,
    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.

```

```

145 //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.
146 //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 $\text{suffix}(sa[1]), \text{suffix}(sa[2]), \dots$ 
    ,  $\text{suffix}(n)$ 的顺序计算, 新进一个后缀 $\text{suffix}(sa[k])$ , 将产生 $n - sa[k] + 1$ 的新的前缀,
    其中 $\text{height}[k]$ 的和前面是相同的, 所以 $\text{suffix}(sa[k])$ 贡献 $n - sa[k] + 1 - \text{height}[k]$ 个不同的子串.
    故答案是 $\sum_{k=1}^n n - sa[k] - 1 - \text{height}[k]$ .
147 //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,
    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 $\text{lcp}(\text{suffix}(k), \text{suffix}(2*n$ 
    +  $2 - k))$ ; 偶数 $\text{lcp}(\text{suffix}(k), \text{suffix}(2*n + 3 - k))$   $O(n \log n)$  RMQ: $O(n)$ 
148 //连续重复子串: 字符串S是有字符串S重复R次得到的.
149 //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 $\text{lcp}(\text{suffix}(1),$ 
     $\text{suffix}(k+1))$ 是否等于 $n - k$ . 求解时只需预处理height数组中的每一个数到 $\text{height}[\text{rk}[1]]$ 的最小值即可
150 //给定字符串, 求重复次数最多的连续重复子串 $O(n \log n)$ : 先枚举长度L,
    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,
    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,
    那么S肯定包括了字符 $r[0], r[L], r[L*2], r[L*3], \dots$ 中的某相邻的两个.
    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,
    那么这里连续出现了 $K/L+1$ 次. 最后看最大值是多少.
151 //字符串A和B最长公共前缀 $O(|A|+|B|)$ : 新串: A+特殊字符+B,  $\text{height}/k : A+B$ ,
    对后缀数组分组(每组height值都不小于k), 每组中扫描到B时,
    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.
152
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 $O(n \log n)$ .

```

## 5.3 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 }

```

## 5.4 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 \* B中不同子串的个数 -  $\sum_{c='a'..'z'} (A中以字符c结尾的不同子串数 * B中以字符c开头的不同子串数)$ . 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;  
 在某后缀前面(或前面)的后缀与该后缀的lcp是非增的,  
 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.  
 来源: Gym 100418C  
 标签: 后缀数组, 二分, 区间更新查询  
 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。



## 6 计算几何

### 6.1 计算几何基础

```
1 //精度设置
2 const double EPS = 1e-6;
3 int sgn(double x)
4 {
5     if(x < -EPS)return -1;
6     return x > EPS ? 1 : 0;
7 }
8 //点 (向量)的定义和基本运算
9 struct Point
10 {
11     double x, y;
12     Point(double _x = 0.0, double _y = 0.0): x(_x), y(_y) {}
13     Point operator + (Point &b)//向量加法
14     {
15         return Point(x + b.x, y + b.y);
16     }
17     Point operator - (Point &b)//向量减法
18     {
19         return Point(x - b.x, y - b.y);
20     }
21     Point operator * (double b)//标量乘法
22     {
23         return Point(x * b, y * b);
24     }
25     /*向量点积  $a \cdot b = |a||b| \cos \theta$ 
26     点积<0, 表示两向量夹角为锐角
27     点积=0, 表示两向量垂直
28     点积>0, 表示两向量夹角为钝角
29     */
30     double operator * (Point &b)
31     {
32         return x * b.x + y * b.y;
33     }
34     /*向量叉积  $a \times b = |a||b| \sin \theta$ 
35     * 叉积小于0, 表示向量b在当前向量顺时针方向
36     * 叉积等于0, 表示两向量平行
37     * 叉积大于0, 表示向量b在当前向量逆时针方向
38     */
39     double operator ^ (Point b)
40     {
41         return x * b.y - y * b.x;
42     }
43     Point rot(double ang)
44     {
45         //向量逆时针旋转ang弧度
46         return Point(x * cos(ang) - y * sin(ang), x * sin(ang) + y * cos(ang));
47     }
48 };
49 //直线 线段定义
50 //直线方程: 两点式:  $(x_2 - x_1)(y - y_1) = (y_2 - y_1)(x - x_1)$ 
51 struct Line
52 {
53     Point s, e;
54     double k;
55     Point() {}
56     Point(Point _s, Point _e)
57     {
58         s = _s, e = _e;
59         k = atan2(e.y - s.y, e.x - s.x);
60     }
61 }
```

```

60     }
61     //求两直线交点
62     //返回-1两直线重合, 0 相交, 1 平行
63     pair<int, Point> operator &(Line &b)
64     {
65         if(sgn((s - e) ^ (b.s - b.e)) == 0)
66         {
67             if(sgn((s - b.e) ^ (b.s - b.e)) == 0)
68                 return make_pair(-1, s); //重合
69             else
70                 return make_pair(1, s); //平行
71         }
72         double t = ((s - b.s) ^ (b.s - b.e)) / ((s - e) ^ (b.s - b.e));
73         return make_pair(0, Point(s.x + (e.x - s.x) * t, s.y + (e.y - s.y) * t));
74     }
75 };
76
77 //两点间距离
78 double dist(Point &a, Point &b)
79 {
80     return sqrt((a - b) * (a - b));
81 }
82
83 /*判断点p在线段l上
84 * (p - l.s) ^ (l.s - l.e) = 0; 保证点p在直线L上
85 * p在线段l的两个端点l.s, l.e为对角定点的矩形内
86 */
87 bool Point_on_Segment(Point &p, Line &l)
88 {
89     return sgn((p - l.s) ^ (l.s - l.e)) == 0 &&
90           sgn((p.x - l.s.x) * (p.x - l.e.x)) <= 0 &&
91           sgn((p.y - l.s.y) * (p.y - l.e.y)) <= 0;
92 }
93 //判断点p在直线l上
94 bool Point_on_Line(Point &p, Line &l)
95 {
96     return sgn((p - l.s) ^ (l.s - l.e)) == 0;
97 }
98
99 /*判断两线段l1, l2相交
100 * 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
101 * 2. 跨立实验: l2的两个端点是否在线段l1的两端
102 */
103 bool seg_seg_inter(Line seg1, Line seg2)
104 {
105     return
106         sgn(max(seg1.s.x, seg1.e.x) - min(seg2.s.x, seg2.e.x)) >= 0 &&
107         sgn(max(seg2.s.x, seg2.e.x) - min(seg1.s.x, seg1.e.x)) >= 0 &&
108         sgn(max(seg1.s.y, seg1.e.y) - min(seg2.s.y, seg2.e.y)) >= 0 &&
109         sgn(max(seg2.s.y, seg2.e.y) - min(seg1.s.y, seg1.e.y)) >= 0 &&
110         sgn((seg2.s - seg1.e) ^ (seg1.s - seg1.e)) * sgn((seg2.e - seg1.e) ^ (seg1.s - seg1.e)) <=
111         0 &&
112         sgn((seg1.s - seg2.e) ^ (seg2.s - seg2.e)) * sgn((seg1.e - seg2.e) ^ (seg2.s - seg2.e)) <=
113         0;
114 }
115 //判断直线与线段相交
116 bool seg_line_inter(Line &line, Line &seg)
117 {
118     return sgn((seg.s - line.e) ^ (line.s - line.e)) * sgn((seg.e - line.e) ^ (line.s - line.e)) <=
119     0;
120 }
121 //点到直线的距离, 返回垂足

```

```

121 Point Point_to_Line(Point p, Point l)
122 {
123     double t = ((p - l.s) * (l.e - l.s)) / ((l.e - l.s) * (l.e - l.s));
124     return Point(l.s.x + (l.e.x - l.s.x) * t, l.s.y + (l.e.y - l.s.y) * t);
125 }
126 //点到线段的距离
127 //返回点到线段最近的点
128 Point Point_to_Segment(Point p, Line seg)
129 {
130     double t = ((p - l.s) * (l.e - l.s)) / ((l.e - l.s) * (l.e - l.s));
131     if(t >= 0 && t <= 1)
132         return Point(l.s.x + (l.e.x - l.s.x) * t, l.s.y + (l.e.y - l.s.y) * t);
133     else if(sgn(dist(p, l.s) - dist(p, l.e)) <= 0)
134         return l.s;
135     else
136         return l.e;
137 }

```

## 6.2 多边形 Polygon

```

1 /*1. 三角形
2  * 顶点A,B,C,边a, b, c
3  * 内接圆半径r, 外接圆半径R
4  * 三角形面积:

```

$$S_{\triangle ABC} = \frac{1}{2}ab \sin \alpha = \frac{1}{2} \times |\vec{AB} \times \vec{AC}|$$

$$S_{\triangle ABC} = \frac{1}{2}hc$$

$$S_{\triangle ABC} = \frac{abc}{4R} = \frac{(a+b+c)r}{2}$$

$$S_{\triangle ABC} = \sqrt{p(p-a)(p-b)(p-c)} \quad (p = \frac{1}{2}(a+b+c))$$

```

5  * 外接圆: 圆心(外心): 三条边上垂直平分线的交点, 半径R: 外心到顶点距离
6  * 两条垂直平分线:  $(x - \frac{x_A+x_B}{2})(x_A - x_B) = -(y_A - y_B)(y - \frac{y_A+y_B}{2})$ 
7  * 和  $(x - \frac{x_B+x_C}{2})(x_B - x_C) = -(y_B - y_C)(y - \frac{y_B+y_C}{2})$ 
8  * 外心坐标:

```

$$x = \frac{\frac{(x_A - x_B)(x_A + x_B)}{2y_A - 2y_B} - \frac{(x_B - x_C)(x_B + x_C)}{2y_B - 2y_C} + \frac{y_A + y_B}{2} - (y_B + y_C)}{\frac{x_A - x_B}{y_A - y_B} - \frac{x_B - x_C}{y_B - y_C}}$$

$$y = \frac{\frac{(y_A - y_B)(y_A + y_B)}{2x_A - 2x_B} - \frac{(y_B - y_C)(y_B + y_C)}{2x_B - 2x_C} + \frac{x_A + x_B}{2} - (x_B + x_C)}{\frac{y_A - y_B}{x_A - x_B} - \frac{y_B - y_C}{x_B - x_C}}$$

```

9  * 外心: Line((A+B)*0.5, (A-B).rot(PI*0.5)+(A+B)*0.5)&Line((B+C)*0.5, (B-C).rot(PI*0.5)+(B+C)*0.5);
10 * 内切圆: 内心: 角平分线的交点, 半径r: 内心到边的距离
11 */
12 //三角形的质心Centroid: 三条高的交点: Q = (A+B+C)*(1.0/3.0)
13 Point Centroid(Point A, Point B, Point C)
14 {
15     return (A + B + C) * (1.0 / 3.0);
16 }
17 //2. 多边形
18 /*判断点在凸多边形内
19 //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)
20 //点的编号: 0~n-1
21 //返回值:
22 //-1: 点在凸多边形外
23 //0: 点在凸多边形边界上
24 //1: 点在凸多边形内
25 int inConvexPoly(Point a, Point p[], int n)
26 {
27     p[n] = p[0];

```

```

28     for(int i = 0; i < n; i++)
29     {
30         if(sgn((p[i] - a) ^ (p[i + 1] - a)) < 0)return -1;
31         else if(OnSeg(a, Line(p[i], p[i + 1])))return 0;
32     }
33     return 1;
34 }
35 /**判断点在任意多边形内
36 //射线法, poly[]的顶点数要大于等于3,点的编号0~n-1
37 //返回值
38 //-1:点在凸多边形外
39 //0:点在凸多边形边界上
40 //1:点在凸多边形内
41 int inPoly(Point p, Point poly[], int n)
42 {
43     int cnt;
44     Line ray, side;
45     cnt = 0;
46     ray.s = p;
47     ray.e.y = p.y;
48     ray.e.x = -1000000000000.0; //-INF, 注意取值防止越界
49     for(int i = 0; i < n; i++)
50     {
51         side.s = poly[i];
52         side.e = poly[(i + 1) % n]; /**判断点在任意多边形内
53         if(OnSeg(p, side))return 0;
54         //如果平行轴则不考虑
55         if(sgn(side.s.y - side.e.y) == 0)
56             continue;
57         if(OnSeg(side.s, ray))
58         {
59             if(sgn(side.s.y - side.e.y) > 0)cnt++;
60         }
61         else if(OnSeg(side.e, ray))
62         {
63             if(sgn(side.e.y - side.s.y) > 0)cnt++;
64         }
65         else if(inter(ray, side))
66             cnt++;
67     }
68     if(cnt % 2 == 1)return 1;
69     else return -1;
70 }
71
72 //判断凸多边形
73 //允许共线边
74 //点可以是顺时针给出也可以是逆时针给出
75 //点的编号1~n-1
76 bool isconvex(Point poly[], int n)
77 {
78     bool s[3];
79     memset(s, false, sizeof(s));
80     for(int i = 0; i < n; i++)
81     {
82         s[sgn((poly[(i + 1) % n] - poly[i]) ^ (poly[(i + 2) % n] - poly[i])) + 1] = true;
83         if(s[0] && s[2])return false;
84     }
85     return true;
86 }
87 //多边形的面积: 分解为多个三角形的面积和
88 double CalArea(vector<Point> p)
89 {
90     double res = 0;
91     p.push_back(p[0]);

```

```

92     for(int i = 1; i < (int)p.size(); i++)
93         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
94     return fabs(res * 0.5); //三角面积需乘以0.5, 以及取正
95 }
96 //多边形的质心: 三角质心坐标的面积加权和
97 Point Centroid(vector<Point> p) //按顺时针方向或逆时针方向排列
98 {
99     Point c = Point(0.0, 0.0);
100     double S = 0.0, s;
101     for(int i = 2; i < (int)p.size(); i++)
102     {
103         s = ((p[i] - p[0]) ^ (p[i] - p[i - 1])); //求三角形面积(*0.5)
104         S += s;
105         c = c + (p[0] + p[i - 1] + p[i]) * s; //求三角形质心(/3.0)
106     }
107     c = c * (1.0 / (3.0 * S));
108     return c;
109 }
110 /*4. 圆
111 圆心(x, y), 半径r
112 面积:  $\pi r^2$ , 周长:  $2\pi r$ 
113 弧长:  $\theta r$ 
114 */

```

## 6.3 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x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

```

## 6.4 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 }

```

## 6.5 立体几何

```

1  ///三维几何
2  struct Point
3  {
4      double x, y, z;
5      Point() {x = y = z = 0;}
6      Point(double _x, double _y, double _z) {x = _x, y = _y, z = _z;}
7      double operator *(const Point a)const//点积
8      {
9          return x * a.x + y * a.y + z * a.z;
10     }
11     Point operator ^(const Point a)const //叉积
12     {
13         return Point(y * a.z - z * a.y, z * a.x - x * a.z, x * a.y - y * a.z);
14     }
15     Point operator *(const double a)const
16     {
17         x *= a, y *= a, z *= a;
18         return *this;
19     }
20     Point operator /(const double a)const
21     {
22         x /= a, y /= a, z /= a;
23         return *this;
24     }
25     Point operator + (const Point a)const {return Point(x + a.x, y + a.y, z + a.z);}

```

```

26     Point operator - (const Point a) const {return Point(x - a.x, y - a.y, z - a.z);}
27     double length()
28     {
29         return sqrt(x * x + y * y + z * z);
30     }
31 };
32 //四面体体积
33 double V(Point a, Point b, Point c, Point d)
34 {
35     return ((a - b) ^ (a - c)) * (a - d) / 6.0;
36 }
37 //四面体重心
38 Point zhongxin(Point a, Point b, Point c, Point d)
39 {
40     //
41     // Point res;
42     // res.x = (a.x + b.x + c.x + d.x)/4.0;
43     // res.y = (a.y + b.y + c.y + d.y)/4.0;
44     // res.z = (a.z + b.z + c.z + d.z)/4.0;
45     // return res;
46     return (a + b + c + d) / 4.0;
47 }
48 //多面体的重心
49 //将多面体分解为若干个四面体，重心为体积的加权 and zhongxin = (v1*zx1 + v2*zx2 + ... + vn*zxn)/(v1 +
    v2 + ... + v3)
50
51 //空间三角形的面积
52 double Area(Point a, Point b, Point c)
53 {
54     return (a - b) * (a - c);
55 }
56 //点到平面的距离 h = 3.0*V/S
57 Point dis(Point p, Point a, Point b, Point c)
58 {
59     return fabs(3.0 * V(p, a, b, c) / S(a, b, c));
60 }

```

## 6.6 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 */

```



## 7 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      while(l + 1 < r)
9      {
10         mid = (l + r) >> 1;
11         if(f(mid))
12             r = mid; //视情况定
13         else
14             l = mid;
15     }
16     for(; l <= r; l++)
17         if(f(l))
18             return l;
19 }
20 ///三分搜索
21 //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
22 //便于求(抛物线)的最值
23 //注意:  $l \% 3 == 0$ ,  $r = l + 1 \mid l + 2$ 时,  $(l + l + r) / 3 == l$  容易出现死循环
24 int three_search(int l, int r)
25 {
26     int ll, rr;
27     while(l + 2 < r)
28     {
29         ll = (l + l + r) / 3;
30         rr = (l + r + r) / 3;
31         if(f(ll) < f(rr))
32             r = rr;
33         else
34             l = ll;
35     }
36     return min(f(l), f(r), f(l + 1));
37 }
```

## 8 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
```

## 9 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 }
```