ACM 模板

dnvtmf

2015

目录

1 数据结构

1.1 RMQ 相关

```
/*区间的rmq问题
 * 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
 * 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，
 *       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
 * 遇到不能改变查询顺序的题，应该用可持久化线段树
 */
/*数组区间颜色数查询
    问题：给定一个数组,要求查询某段区间内有多少种数字
    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，
     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].
*/
```

1.2 Treap

```
//随机化二叉树Treap
struct node
{
    int key;
    int ch[2], fix;
};
struct Treap
{
    node *p;
    int size, root;
    Treap()
    {
        p = new node[NUM];
        srand(time(0));
        size = -1;
        root = -1;
    }
    ~Treap() {delete []p;}
    void rot(int &x, int op)//op = 0 左旋, op = 1右旋
    {
        int y = p[x].ch[!op];
        p[x].ch[!op] = p[y].ch[op];
        p[y].ch[op] = x;
        x = y;
    }
    //如果当前节点的优先级比根大就旋转,
     如果当前节点是根的左儿子就右旋如果当前节点是根的右儿子就左旋.
    void insert(int tkey, int &k = root)
    {
        if(k == -1)
        {
            k = ++size;
            p[k].ch[0] = p[k].ch[1] = -1;
            p[k].key = tkey;
            p[k].fix = rand();
        }
        else if(tkey < p[k].key)
        {
            insert(tkey, p[k].ch[0]);
            if(p[p[k].ch[0]].fix > p[k].fix)
                rot(k, 1);
        }
```

```
        else
        {
            insert(tkey, p[k].ch[1]);
            if(p[p[k].ch[1]].fix > p[k].fix)
                rot(k, 0);
        }
    }
    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
    {
        if(k == -1) return ;
        if(tkey < p[k].key)
            remove(tkey, p[k].ch[0]);
        else if(tkey > p[k].key)
            remove(tkey, p[k].ch[1]);
        else
        {
            if(p[k].ch[0] == -1 && p[k].ch[1] == -1)
                k = -1;
            else if(p[k].ch[0] == -1)
                k = p[k].ch[1];
            else if(p[k].ch[1] == -1)
                k = p[k].ch[0];
            else if(p[p[k].ch[0]].fix < p[p[k].ch[1]].fix)
            {
                rot(k, 0);
                remove(tkey, p[k].ch[0]);
            }
            else
            {
                rot(k, 1);
                remove(tkey, p[k].ch[1]);
            }
        }
    }
};
```

1.3 ST 表

```
///ST表(Sparse Table)
//对静态数组，查询任意区间[l, r]的最大（小）值
// 预处理O(nlog n), 查询O(1)
#define MAX 10000
int st[MAX][32];//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
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))
void pre_Log2()
{
    Log2[1] = 0;
    for(int i = 2; i < NUM; i++)
    {
        Log2[i] = Log2[i - 1];
        if((1 << Log2[i] + 1) == i)
            ++Log2[i];
    }
}
template<class T>
void pre_ST(int n, T ar[])//n 数组长度， ar 数组
{
    int i, j;
    pre_Log2();
    for(i = n - 1; i >= 0; i--)
    {
        st[i][0] = ar[i];
```

```
        for(j = 1; i + (1 << j)   n; j++)
            st[i][j] = max(st[i][j - 1], st[i + (1 << j - 1)][j - 1]);
    }
}
template<class T>
T query(int l, int r)
{
    int k = Log2[r - l + 1];
    return max(st[l][k], st[r - (1 << k) + 1][k]);
}
```

1.4 最长上升子序列 LIS

```
/*最长上升子序列LIS
 * 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）
 * 方法:dp
 * 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$a_i$.
 * 可以用pre数组存储第i个数的最长子序列的前一个数
 */
/*二维偏序的LIS
 * 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
 * 做法：二分+树状数组
 * 将序列[l, r]二分，先处理左边的区间[l, mid]，
 * 再用左边的区间更新右边的区间，即将区间[l,r]按左端点排序，然后依次扫描，
    遇到在左半区间的加入树状数组，
 *      遇到在右半区间的查询比当前y值更小的数对数并更新
 *  然后再递归处理右边的区间[mid+1,r]
 */
```

2 动态规划

3 图论

3.1 无向图的桥，割点，双联通分量

```
/*
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).
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.
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.
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;
割边(桥)：(边连通度为1)删去割边后，连通图变得不连通.
点双连通分量：点连通度大于1的分量.
边双连通分量：边连通度大于1的分量.
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).
*/
///Tarjan算法求割点或桥
/*
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).
割点：该点是根结点且有不止一棵子树，或该结点的任意一个孩子结点，没有到该结点祖先的反向边(存在
    dfn(u) leq low(v))
桥：当且仅当该边(u,v)是树枝边，且dfn[u] < low[v]
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.
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.
点双连通分支：建立一个栈，存储当前双连通分支，在搜索图时，每找到一条树枝边或后向边(非横叉边),
    就把这条边加入栈中. 如果遇到某时满足dfn[u] leq low[v]，说明u是一个割点，
    同时把边从栈顶一个个取出，直到遇到了边(u,v)，取出的这些边与其关联的点，组成一个点双连通分支.
    割点可以属于多个点双连通分支，其余点和每条边只属于且属于一个点双连通分支.
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.
*/

//图
const int MAXV = 100010, MAXE = 100010;
struct edge
{
    int next, to;
} e[MAXE];
int head[MAX_V], htot;
int V, E;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v)
{
    e[htot].to = v;
    e[htot].next = head[u];
    head[u] = htot++;
}

int dfs[MAXV], low[MAXV];
int stk[MAXV], top;//栈 -DCC

//割点 点双连通分量
int cp[MAXV];//记录割点
int id[MAXE], cnt_dcc;//连通分量编号及总数 -DCC
void tarjan(int u, int order, int root, int pree) //重边对应边的id, 否则对应父亲结点fa
{
    dfs[u] = low[u] = order++;
    int num = 0;
```

```
    for(int i = head[u]; i != -1; i = e[i].next)
    {
        int v = e[i].v;
        if(!dfs[v])
        {
            num++;
            stk[top++] = i;//-DCC
            tarjan(v, order, root, i);
            if(low[u] > low[v])
                low[u] = low[v];
            if(u == root && num >= 2)cp[u] = 1;
            else if(u != root && dfs[u] <= low[v])
            {
                cp[u] = 1;//是割点
                //-DCC
                cnt_dcc++;
                do
                {
                    id[stk[--top]] = cnt_dcc;
                }
                while(stk[top] != i);
            }
        }
        else if((i ^ 1) != pree) //-DCC
        {
            if(low[u] > dfs[v])
                low[u] = dfs[v];
            //-DCC
            if(dfs[u] > dfs[v])
                stk[top++] = i;
        }
    }
}
void DCC()
{
    top = cnt_dcc = 0;// -DCC
    memset(dfs, 0, sizeof(dfs));
    memset(cp, 0, sizeof(cp));
    for(int i = 1; i <= V; i++)
        tarjan(1, 1, 1, -1);
}


//桥 边双连通分量
int ce[MAXE], num; //记录桥
int id[MAXV], cnt_dcc;//连通分量编号及总数
void tarjan(int u, int order, int pree)//重边是pree为边的id, 否则为父亲结点fa
{
    dfs[u] = low[u] = order++;
    stk[top++] = u; //-DCC
    for(int i = head[u]; i != -1; i = e[i].next)
    {
        int v = e[i].v;
        if(!dfs[v])
        {
            tarjan(v, order, i);
            if(low[v] < low[u])
                low[u] = low[v];
            if(dfs[u] < low[v])
            {
                ce[num++] = i;
                //-DCC
                cnt_dcc++;
                do
```

```
                {
                    id[stk[--top]] = cnt_dcc;
                }
                while(stk[top] != v);
            }
        }
        else if((i ^ 1) != pree && dfs[v] < low[u])
            low[u] = dfs[v];
    }
}
void DCC()
{
    memset(dfs, 0, sizeof(dfs));
    num = 0;
    top = cnt_dcc = 0;//-DCC
    for(int i = 1; i <= V; i++)
        tarjan(1, 1, -1);
}
```

3.2 最短路 shortest path

```
///最短路 Shortest Path
//Bellman-Ford算法O(|E| * |V|)
//d[v] = min{d[u] + w[e]}(e =< u, v >∈ E)

const int MAXV = 1000, MAXE = 1000, INF = 1000000007;
struct edge {int u, v, cost;} e[MAXE];
int V, E;
//graph G
int d[MAXV];
void Bellman_Ford(int s)
{
    for(int i = 0; i < V; i++)
        d[i] = INF;
    d[s] = 0;
    while(true)
    {
        bool update = false;
        for(int i = 0; i < E; i++)
        {
            if(d[e[i].u] != INF && d[e[i].v] > d[e[i].u] + e[i].cost)
            {
                d[e[i].v] = d[e[i].u] + e[i].cost;
                update = true;
            }
        }
    }
}
//判负圈
bool find_negative_loop()
{
    memset(d, 0, sizeof(d));
    for(int i = 0; i < V; i++)
    {
        for(int j = 0; j < E; j++)
        {
            if(d[e[j].v] > d[e[i].u] + e[j].cost)
            {
                d[e[j].v] = d[e[j].u] + e[j].cost;
                if(i == V - 1)
                    return true;
                //循环了V次后还不能收敛，即存在负圈
```

```
            }
        }
    }
    return false;
}

//spfa算法 O(|E| log |V|)
//适用于负权图和稀疏图, 稳定性不如dijstra
//存在负环返回false
int d[MAXV], outque[MAXV];
bool vis[MAXV];
bool spfa(int s)
{
    for(int i = 0; i < V; i++)
    {
        vis[i] = false;
        d[i] = INF;
        outque[i] = 0;
    }
    d[s] = 0;
    queue<int> que;
    que.push(s);
    vis[s] = true;
    while(!que.empty())
    {
        int u = que.front();
        que.pop();
        vis[u] = false;
        if(++outque[u] > V) return false;;
        for(int i = head[u]; i != -1; i = e[i].next)
        {
            int v = e[i].to;
            if(d[v] > d[u] + e[i].w)
            {
                d[v] = d[u] + e[i].w;
                if(!vis[v])
                {
                    vis[v] = true;
                    que.push(v);
                }
            }
        }
    }
    return true;
}

//dijkstra算法O(|V|^2)
int cost[MAXV][MAXV];
int d[MAXV];
bool vis[MAXV];
void dijkstra(int s)
{
    fill(d, d + V, INF);
    memset(vis, 0, sizeof(vis));
    d[s] = 0;
    while(true)
    {
        int v = -1;
        for(int u = 0; u < V; u++)
            if(!vis[u] && (v == -1 || d[u] < d[v]))
                v = u;
        if(v == -1) break;
        for(int u = 0; u < V; u++)
            d[u] = min(d[u], d[v] + cost[v][u]);
```

```
    }
}

//dijkstra算法 O(|E|log|V|)
struct edge {int v, cost;};
vector<edge> g[MAXV];
int d[MAXV];

void dijkstra(int s)
{
    priority_queue<P, vector<P>, greater<P> > que;
    fill(d, d + V, INF);
    d[s] = 0;
    que.push(P(0, s));
    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int u = p.second;
        if(d[u] < p.first) continue;
        for(int i = 0; i < g[u].size(); i++)
        {
            edge &e = g[u][i];
            if(d[e.v] > d[u] + e.cost)
            {
                d[e.v] = d[u] + e.cost;
                que.push(P(d[e.v], e.v));
            }
        }
    }
}

///任意两点间最短路
//Floyd-Warshall算法 O(|V|^3)
int d[MAX_V][MAX_V];
int V;
void floyd_warshall()
{
    int i, j, k;
    for(k = 0; k < V; k++)
        for(i = 0; i < V; i++)
            for(j = 0; j < V; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

///两点间最短路 —— 一条可行路径还原
/*用prev[u]记录从s到u的最短路上u的前驱结点*/
vector <int> get_path(int t)
{
    vector <int> path;
    for(; t != -1; t = prev[t])
        path.push_back(t);
    reverse(path.begin(), path.end());
    return path;
}

///两点间最短路 —— 所有可行路径还原
/*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|)
  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
*/
//情况1
int vis[MAXV];
vector<edge> G[MAXV];
void add_edge() {}
```

```cpp
void get_pathG(int u)
{
    vis[u] = 1;
    for(int i = 0; i < g[u].size(); i++)
    {
        int v = g[u][i].v, w = g[u][i].w;
        if(d[v] + w == d[u])
        {
            add_edge(u, v);
            add_edge(v, u);
            if(!vis[v])
                get_pathG(v);
        }
    }
}
//情况2
struct edge
{
//...
} e[MAXE];
int head[MAXV], tot;
vector<int> g[MAXV];//所有最短路形成的逆图
int vis[MAXV];
void dijkstra(int s)
{
    //... ... other part see above
    for(int i = head[u]; i != -1; i = e[i].next)
    {
        int v = e[i].to;
        if(d[v] > d[u] + e[i].w)
        {
            g[v].clear();
            g[v].push_back(u);
            d[v] = d[u] + e[i].w;
            que.push(P(d[v], v));
        }
        else if(d[v] == d[u] + e[i].w)
        {
            g[v].push_back(u);
        }
    }
}

void get_all_path(int s, int t)
{
    memset(vis, 0, sizeof(vis));
    queue<int> que;
    que.push(t);
    vis[t] = 1;
    while(!que.empty())
    {
        int u = que.front();
        que.pop();
        for(int i = 0; i < g[u].size(); i++)
            if(!vis[g[u][i]])
            {
                vis[g[u][i]] = 1;
                que.push(g[u][i]);
            }
    }
    for(int i = 1; i <= V; i++)
        if(!vis[i])
        {
            g[i].clear();//清空不是路径上的点
```

```
        }
}
///求最短路网络上的桥
//方法1: 将最短路网络新建一个图, 跑Tarjan算法.
//方法2: 最短路径还原时, 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
//求割点类似
int vis[MAXV];
int ce[MAXE], num;
void get_bridge(int s, int t)//在逆图中运行
{
    priority_queue<P> que;
    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
    memset(vis, 0, sizeof(vis));
    vis[t] = 1;
    num = 0;
    while(!que.empty())
    {
        int u  = que.top().SE;
        que.pop();
        int cnt = 0;
        if(que.empty())
        {
            for(int i = rhead[u]; i != -1; i = re[i].next)
            {
                int v = re[i].to, w = re[i].cost;
                if(dist[v] + w == dist[u])
                {
                    cnt++;
                    if(cnt >= 2) break;
                }
            }
        }
        bool f = que.empty() && cnt == 1;//当且仅当, 队列为空且只有一条路时, 找到桥
        for(int i = rhead[u]; i != -1; i = re[i].next)
        {
            int v = re[i].to, w = re[i].cost;
            if(dist[v] + w == dist[u])
            {
                if(f) ce[num++] = i;
                if(!vis[v])
                {
                    vis[v] = 1;
                    que.push(P(dist[v], v));
                }
            }
        }
    }
}
```

3.3 最大流 maximum flow

```
///最大流 maximum flow
//最大流最小割定理 : 最大流 = 最小割
///FF算法 Ford-Fulkerson算法 O(F|E|) F为最大流量
//1. 初始化: 原边容量不变, 回退边容量为0, max_flow = 0
//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
//3. 增广路中找到瓶颈边, max_flow加上其容量
//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
struct edge
{
    int to, cap, rev;
};
```

```
vector <edge> G[MAXV];
bool used[MAXV];

void add_edge(int from, int to, int cap)
{
    G[from].push_back((edge) {to, cap, G[to].size()});
    G[to].push_back((edge) {from, 0, G[from].size() - 1});
}

//dfs寻找增广路
int dfs(int v, int t, int f)
{
    if(v == t)
        return f;
    used[v] = true;
    for(int i = 0; i < G[v].size(); i++)
    {
        edge &e = G[v][i];
        if(!used[e.to] && e.cap > 0)
        {
            int d = dfs(e.to, t , min(f, e.cap));
            if(d > 0)
            {
                e.cap -= d;
                G[e.to][e.rev].cap += d;
                return d;
            }
        }
    }
    return 0;
}

//求解从s到t的最大流
int max_flow(int s, int t)
{
    int flow = 0;
    for(;;)
    {
        memset(used, 0, sizeof(used));
        int f = dfs(s, t, INF);
        if(f == 0)
            return flow;
        flow += f;
    }
}
///Dinic算法 O(|E|·|V|^2)
//似乎比链式前向星快
struct edge {int to, cap, rev;};
vector <edge> G[MAXV];
int level[MAXV];
int iter[MAXV];
void init()
{
    for(int i = 0; i < MAXV; i++)
        G[i].clear();
}
void add_edge(int from, int to, int cap)
{
    G[from].push_back((edge) {to, cap, G[to].size()});
    G[to].push_back((edge) {from, 0, G[from].size() - 1 });
}
bool bfs(int s, int t)
{
```

```
    memset(level, -1, sizeof(level));
    queue <int> que;
    level[s] = 0;
    que.push(s);
    while(!que.empty())
    {
        int v = que.front();
        que.pop();
        for(int i = 0; i < (int)G[v].size(); i++)
        {
            edge &e = G[v][i];
            if(e.cap > 0 && level[e.to] < 0)
            {
                level[e.to] = level[v] + 1;
                que.push(e.to);
            }
        }
    }
    return level[t] != -1;
}

int dfs(int v, int t, int f)
{
    if(v == t) return f;
    for(int &i = iter[v]; i < (int)G[v].size(); i++)
    {
        edge &e = G[v][i];
        if(e.cap > 0 && level[v] < level[e.to])
        {
            int d = dfs(e.to, t, min(f, e.cap));
            if(d > 0)
            {
                e.cap -= d;
                G[e.to][e.rev].cap += d;
                return d;
            }
        }
    }
    return 0;
}

int max_flow(int s, int t)
{
    int flow = 0, cur_flow;
    while(bfs(s, t))
    {
        memset(iter, 0, sizeof(iter));
        while((cur_flow = dfs(s, t, INF)) > 0) flow += cur_flow;
    }
    return flow;
}
//SAP算法 O(|E|·|V|^2)
#define MAXV 1000
#define MAXE 10000
struct edge
{
    int cap, next, to;
} e[MAXE * 2];
int head[MAXV], tot_edge;
void init()
{
    memset(head, -1, sizeof(head));
    tot_edge = 0;
}
```

```
void add_edge(int u, int v, int cap)
{
    e[tot_edge] = (edge) {cap, head[u], v};
    head[u] = tot_edge++;
}
int V;
int numh[MAXV];//用于GAP优化的统计高度数量数组
int h[MAXV];//距离标号数组
int pree[MAXV], prev[MAXV];//前驱边与结点
int SAP_max_flow(int s, int t)
{
    int i, flow = 0, u, cur_flow, neck = 0, tmp;
    memset(h, 0, sizeof(h));
    memset(numh, 0, sizeof(numh));
    memset(prev, -1, sizeof(prev));
    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
        pree[i] = head[i];
    numh[0] = V;
    u = s;
    while(h[s] < V)
    {
        if(u == t)
        {
            cur_flow = INT_MAX;
            for(i = s; i != t; i = e[pree[i]].to)
            {
                if(cur_flow > e[pree[i]].cap)
                {
                    neck = i;
                    cur_flow = e[pree[i]].cap;
                }
            }//增广成功，寻找"瓶颈"边
            for(i = s; i != t; i = e[pree[i]].to)
            {
                tmp = pree[i];
                e[tmp].cap -= cur_flow;
                e[tmp ^ 1].cap += cur_flow;
            }//修改路径上的边容量
            flow += cur_flow;
            u = neck;//下次增广从瓶颈边开始
        }
        for(i = pree[u]; i != -1; i = e[i].next)
            if(e[i].cap && h[u] == h[e[i].to] + 1)
                break;//寻找可行弧
        if(i != -1)
        {
            pree[u] = i;
            prev[e[i].to] = u;
            u = e[i].to;
        }
        else
        {
            if(0 == --numh[h[u]])break;//GAP优化
            pree[u] = head[u];
            for(tmp = V, i = head[u]; i != -1; i = e[i].next)
                if(e[i].cap)
                    tmp = min(tmp, h[e[i].to]);
            h[u] = tmp + 1;
            ++numh[h[u]];
            if(u != s) u = prev[u];//从标号并且从当前结点的前驱重新增广
        }
    }
    return flow;
}
```

```cpp

///EK算法 O(|V|·|E|^2)
//bfs寻找增广路
const int MAXV = 210;
int g[MAXV][MAXV], pre[MAXV];
int n;
bool vis[MAXV];
bool bfs(int s, int t)
{
    queue <int> que;
    memset(pre, -1, sizeof(pre));
    memset(vis, 0, sizeof(vis));
    que.push(s);
    vis[s] = true;
    while(!que.empty())
    {
        int u = que.front();
        if(u == t) return true;
        que.pop();
        for(int i = 1; i <= n; i++)
            if(g[u][i] && !vis[i])
            {
                vis[i] = true;
                pre[i] = u;
                que.push(i);
            }
    }
    return false;
}
int EK_max_flow(int s, int t)
{
    int u, max_flow = 0, minv;
    while(bfs(s, t))
    {
        minv = INF;
        u = t;
        while(pre[u] != -1)
        {
            minv = min(minv, g[pre[u]][u]);
            u = pre[u];
        }
        ans += minv;
        u = t;
        while(pre[u] != -1)
        {
            g[pre[u]][u] -= minv;
            g[u][pre[u]] += minv;
            u = pre[u];
        }
    }
    return max_flow;
}
```

3.4 最小割 minimum cut

```cpp
///最小割Minimum Cut
/*
定义:
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 ∈ S, 汇t ∈ T. 符号[S,T]代表一个边集合{<u,v> | <u,v>∈E,u∈S,v∈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).
最小割: 该网络中容量最小的割
```

```
(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|.
(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| ≤ c[S,T].
(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:
        (1) f是G的一个最大流
        (2) 残留网络G_f不包含增广路径
        (3) 对G的某个割[S,T], 有|f|=c[S,T]
        即最大流的流值等于最小割的容量
最小割的求法:
    1. 先求的最大流
    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
    注意: 虽然最小割中的边都是满流边, 但满流边不一定都是最小割的边.
*/
int max_flow(int s, int t) {}
int getST(int s, int t, int nd[])
{
    int mincap = max_flow(s, t);
    memset(nd, 0, sizeof(nd));
    queue<int> que;
    que.push(s);
    vis[s] = 1;
    while(!que.empty())
    {
        int u = que.front(); que.pop();
        for(int i = 0; i < (int)g[u].size(); i++)//travel v
            if(g[u][i].cap > 0 && !vis[g[u][i].to])
            {
                vis[g[u][i].to] = 1;
                que.push(g[u][i].to);
            }
    }
}
///无向图全局最小割 Stoer-Wagner算法
/*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)
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
快速求当前图某两点的最小割:
    1. 维护一个集合A, 初始里面只有v_1(可以任意)这个点
    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)
    3. 反复2,直至A集合G集相等
    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
*/
//O(|V|^3)
const int MAXV = 510;
int n;
int g[MAXV][MAXV];//g[u][v]表示u,v两点间的最大流量
int dist[MAXV];//集合A到其他点的距离
int vis[MAXV];
int min_cut_phase(int &s, int &t, int mark) //求某两点间的最小割
{
    vis[t] = mark;
    while(true)
    {
        int u = -1;
        for(int i = 1; i <= n; i++)
            if(vis[i] < mark && (u == -1 || dist[i] > dist[u])) u = i;
        if(u == -1) return dist[t];
        s = t, t = u;
        vis[t] = mark;
        for(int i = 1; i <= n; i++) if(vis[i] < mark) dist[i] += g[t][i];
    }
}

int min_cut()
{
```

```
    int i, j, res = INF, x, y = 1;
    for(i = 1; i <= n; i++)
        dist[i] = g[1][i], vis[i] = 0;
    for(i = 1; i < n; i++)
    {
        res = min(res, min_cut_phase(x, y, i));
        if(res == 0) return res;
        //merge x, y
        for(j = 1; j <= n; j++) if(vis[j] < n) dist[j] = g[j][y] = g[y][j] = g[y][j] + g[x][j];
        vis[x] = n;
    }
    return res;
}
```

3.5 分数规划 Fractional Programming

```
///分数规划 Fractional Programming
//source: <<最小割模型在信息学竞赛中的应用>>
/*
一般形式：min {λ = f(x) = a(x)/b(x)}(x ∈ S, ∀x ∈ S, b(x) > 0)
    其中解向量x在解空间S内, a(x)与b(x)都是连续的实值函数。
    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)，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.
    构造新函数:g(λ) = min {a(x) − λ · b(x)}(x ∈ S)
    函数性质:(单调性) g(λ)是一个严格递减函数，即对于λ1 < λ2，一定有g(λ1) > g(λ2).
    (Dinkelbach定理) 设λ*为原规划的最优解，则g(λ) = 0当且仅当λ = λ*.
    设λ*为该优化的最优解，则:
```

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

```
    由该性质，就可以对最优解λ*进行二分查找。
    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.
*/
///0−1分数规划 0−1 fractional programming
/*是分数规划的解向量x满足∀xi ∈ {0, 1}，即一下形式:
```

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

```
    其中，x表示一个解向量，xe ∈ {0, 1}，即对与每条边都选与不选两种决策，并且选出的边集组成一个s−t边割集. 形式化的，若xe = 1，则e ∈ C，xe = 0，则e ∉ C.
*/
```

3.6 最大闭权图 maximum weight closure of a graph

```
///最大权闭合图 Maximum Weight Closure of a Graph
/*定义:
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.
    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，
    闭合图是这样的一个点集V' ∈ V，满足对于∀u ∈ V'引出的∀ < u, v >∈ E，必有v ∈ V'成立.
    还有一种等价定义为: 满足对于∀ < u, v >∈ E ，若有u ∈ V'成立，必有v ∈ V'成立，
    在布尔代数上这是一个"蕴含(imply)"的运算.
    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wv(任意实数，可正可负).
    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 Σ_{v∈V'} wv.
*/
/*转化为最小割模型GN(VN, EN)
```

在原图点集的基础上增加源s和汇t;
将原图每条有向边 $<u,v>\in E>$ 替换为容量为 $c(u,v)=\infty$ 的有向边 $<u,v>\in E_N$;
增加连接源s到原图每个正权点 $v(w_v>0)$ 的有向边 $<s,v>\in E_N$, 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 $<v,t>\in E_N$, 容量为 $c(v,t)=-w_v$. 其中,
正无限 ∞ 定义为任意一个大于 $\sum_{v\in V}|w_v|$ 的整数。更形式化地, 有:

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

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

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

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.
*/

3.7 最小费用最大流 minimum cost flow

```
///最小费用最大流 miniunm cost flow
//不断寻找最短路增广即可
//复杂度: O(F·MaxFlow(G))
//对于稀疏图的效率较高, 对于稠密图的的效率低
///dijkstra实现 基于0开始的图
const int MAXV = 11000, MAXE = 41000;
struct edge {int next, to, cap, cost;} e[MAXE << 1];
int head[MAXV], htot;
int V;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v, int cap, int cost)
{
    e[htot] = (edge) {head[u], v, cap, cost};
    head[u] = htot++;
    e[htot] = (edge) {head[v], u, 0, -cost};
    head[v] = htot++;
}
int dist[MAXV];
int prev[MAXV], pree[MAXV];
int h[MAXV];
void dijkstra(int s)
{
    priority_queue<P, vector<P>, greater<P> > que;
    fill(dist, dist + V, INF);
    que.push(P(0, s));
    dist[s] = 0;
    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int u = p.SE;
        if(dist[u] < p.FI) continue;
        for(int i = head[u]; ~i; i = e[i].next)
            if(e[i].cap > 0 && dist[e[i].to] > dist[u] + e[i].cost + h[u] - h[e[i].to])
            {
                dist[e[i].to] = dist[u] + e[i].cost + h[u] - h[e[i].to];
                prev[e[i].to] = u;
                pree[e[i].to] = i;
                que.push(P(dist[e[i].to], e[i].to));
            }
    }
}
int min_cost_flow(int s, int t, int flow)
```

```
{
    int min_cost = 0;
    memset(h, 0, sizeof(h));
    while(flow > 0)
    {
        dijkstra(s);
        if(dist[t] == INF) return -1;
        for(int i = 0; i < V; i++) h[i] += dist[t];
        int now_flow = flow;
        for(int u = t; u != s; u = prev[u])//寻找瓶颈边
            now_flow = min(now_flow, e[pree[u]].cap);
        flow -= now_flow;
        min_cost += now_flow * dist[t];
        for(int u = t; u != s; u = prev[u])
        {
            e[pree[u]].cap -= now_flow;
            e[pree[u] ^ 1].cap += now_flow;
        }
    }
    return min_cost;
}
///spfa实现 基于0开始的图
struct edge {int next, to, cap, cost;} e[MAXE << 1];
int head[MAXV], htot;
int V;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v, int cap, int cost)
{
    e[htot] = (edge) {head[u], v, cap, cost};
    head[u] = htot++;
    e[htot] = (edge) {head[v], u, 0, -cost};
    head[v] = htot++;
}
int dist[MAXV];
int prev[MAXV], pree[MAXV];
void spfa(int s)
{
    fill(dist, dist + V, INF);
    dist[s] = 0;
    bool update = true;
    while(update)
    {
        update = false;
        for(int v = 0; v < V; v++)
        {
            if(dist[v] == INF) continue;
            for(int i = head[v]; i != -1; i = e[i].next)
            {
                //edge &e = G[v][i];
                if(e[i].cap > 0 && dist[e[i].to] > dist[v] + e[i].cost)
                {
                    dist[e[i].to] = dist[v] + e[i].cost;
                    prev[e[i].to] = v;
                    pree[e[i].to] = i;
                    update = true;
                }
            }
        }
    }
}
```

```
int h[MAXV];
void dijkstra(int s)
{
    priority_queue<P, vector<P>, greater<P> > que;
    fill(dist, dist + V, INF);
    que.push(P(0, s));
    dist[0] = 0;
    while(!que.empty())
    {
        P p = que.top(); que.pop();
        int u = p.SE;
        if(dist[u] < p.FI) continue;
        for(int i = head[u]; ~i; i = e[i].next)
            if(e[i].cap > 0 && dist[e[i].to] > dist[u] + e[i].cost + h[u] - h[e[i].to])
            {
                dist[e[i].to] = dist[u] + e[i].cost + h[u] - h[e[i].to];
                prev[e[i].to] = u;
                pree[e[i].to] = i;
                que.push(P(dist[e[i].to], e[i].to));
            }
    }
}
int min_cost_flow(int s, int t, int flow)
{
    int min_cost = 0;
    while(flow > 0)
    {
        spfa(s);
        if(dist[t] == INF) return -1;
        int now_flow = flow;
        for(int u = t; u != s; u = prev[u])//寻找瓶颈边
            now_flow = min(now_flow, e[pree[u]].cap);
        flow -= now_flow;
        min_cost += now_flow * dist[t];
        for(int u = t; u != s; u = prev[u])
        {
            e[pree[u]].cap -= now_flow;
            e[pree[u] ^ 1].cap += now_flow;
        }
    }
    return min_cost;
}

///zkw最小费用流，在稠密图上很快
const int MAXV = 11000, MAXE = 41000;
struct edge {int next, to, cap, cost;} e[MAXE << 1];
int head[MAXV], htot;
int V;
void init()
{
    memset(head, -1, sizeof(head));
    htot = 0;
}
void add_edge(int u, int v, int cap, int cost)
{
    e[htot] = (edge) {head[u], v, cap, cost};
    head[u] = htot++;
    e[htot] = (edge) {head[v], u, 0, -cost};
    head[v] = htot++;
}
int dist[MAXV];
int slk[MAXV];
int src, sink;//源和汇
bool vis[MAXV];
```

```
int min_cost;//最小费用
int aug(int u, int f)
{
    int left = f;
    if(u == sink)
    {
        min_cost += f * dist[src];
        return f;
    }
    vis[u] = true;
    for(int i = head[u]; ~i; i = e[i].next)
    {
        int v = e[i].to;
        if(e[i].cap > 0 && !vis[v])
        {
            int t = dist[v] + e[i].cost - dist[u];
            if(t == 0)
            {
                int delta = aug(v, min(e[i].cap, left));
                if(delta > 0) e[i].cap -= delta, e[i ^ 1].cap += delta, left -= delta;
                if(left == 0) return f;
            }
            else
                slk[v] = min(t, slk[v]);
        }
    }
    return f - left;
}
bool modlabel()
{
    int delta = INF;
    for(int i = 0; i < V; i++)
        if(!vis[i]) delta = min(delta, slk[i]), slk[i] = INF;
    if(delta == INF) return false;
    for(int i = 0; i < V; i++)
        if(vis[i]) dist[i] += delta;
    return true;
}
int zkw_min_cost_flow(int s, int t)
{
    src = s, sink = t;
    min_cost = 0;
    int flow = 0;
    memset(dist, 0, sizeof(dist));
    memset(slk, 0x3f, sizeof(slk));
    int tmp = 0;
    do
    {
        do
        {
            memset(vis, false, sizeof(vis));
            flow += tmp;
        }
        while((tmp = aug(src, INF)));
    }
    while(modlabel());
    return min_cost;
}
```

3.8 有上下界的网络流

```
///有上下界的网络流
```

```
//1. 建图——消除上下界
/* 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
    1). <u, T> 容量L  2). <S, v> 容量L 3). <u, v> 容量U - L
    最后添加边<sink, src>, 容量+∞.
    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.
    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.
    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
    实现的时候, 要将从S连向同一结点, 同一结点连向T的多条边合并成一条(容量增加).
*/
```

3.9 最近公共祖先 LCA

```
///最近公共祖先LCA Least Common Ancestors
//Tarjian的离线算法 O(n + q)
struct edge {int next, to, lca;};
//由要查询的<u,v>构成的图
edge qe[MAXE * 2];
int qh[MAXV], qtot;
//原图
edge e[MAXE * 2]
int head[MAXV], tot;
//并查集
int fa[MAXV];
inline int find(int x)
{
    if(fa[x] != x) fa[x] = find(fa[x]);
    return fa[x];
}
bool vis[MAXV];
void LCA(int u)
{
    vis[u] = true;
    fa[u] = u;
    for(int i = head[u]; i != -1; i = e[i].next)
        if(!vis[e[i].to])
        {
            LCA(e[i].to);
            fa[e[i].to] = u;
        }
    for(int i = qh[u]; i != -1; i = qe[i].next)
        if(vis[qe[i].to])
        {
            qe[i].lca = find(eq[i].to);
            eq[i ^ 1].lca = qe[i].lca;//无向图, 入边两次
        }
}

//RMQ的在线算法 O(n log n)
/*算法描述:
    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
    再记录下第一次扫描到结点u时的序号
    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].
*/
struct node
{
    int u;//记录经过的结点
    int depth;//记录当前结点的深度
} vs[2 * MAXV];
```

```
bool operator < (node a, node b) {return a.depth < b.depth;}
int id[MAXV];//记录第一次经过点u时的dfn序号
void dfs(int u, int fa, int dep, int &k)
{
    vs[k] = (node) {u, dep};
    id[u] = k++;
    for(int i = head[u]; i != -1; i = e[i].next)
        if(e[i].to != fa)
        {
            dfs(e[i].to, u, dep + 1, k);
            vs[k++] = (node) {u, dep};
        }
}
//RMQ
//动态查询id[u] 到 id[v] 之间的depth最小的结点
//ST表
int Log2[MAXV * 2];
node st[MAXV * 2][32];
template<class T>
void pre_st(int n, T ar[])
{
    Log2[1] = 0;
    for(int i = 2; i <= n; i++)
    {
        Log2[i] = Log2[i - 1];
        if((1 << Log2[i] + 1) == i) ++Log2[i];
    }
    for(int i = n - 1; i >= 0; i--)
    {
        st[i][0] = ar[i];
        for(int j = 1; i + (1 << j) <= n; j++)
            st[i][j] = min(st[i][j - 1], st[i + (1 << j - 1)][j - 1]);
    }
}
int query(int l, int r)
{
    int k = Log2[r - l + 1];
    return min(st[l][k], st[r - (1 << k) + 1][k]).u;
}

void lca_init()
{
    int k = 0;
    dfs(1, -1, 0, k);
    pre_st(k, vs);
}

int LCA(int u, int v)
{
    u = id[u], v = id[v];
    if(u > v) swap(u, v);
    return query(u, v);
}
```

4 数学专题

4.1 素数 Prime

```
///素数 Prime
///澜素数
int prim[NUM], prim_num;
//O(n log n)
void pre_prime()
{
    prim_num = 0;
    for(int i = 2; i < NUM; i++)
        if(!prim[i])
        {
            prim[prim_num++] = i;
            for(int j = i + i; j < NUM; j += i) prim[j] = 1;
        }
}

//O(n)
void pre_prime()
{
    prim_num = 0;
    for(int i = 2; i < NUM; i++)
    {
        if(!prim[i]) prim[prim_num++] = i;
        for(int j = 0; j < prim_num && i * prim[j] < NUM; j++)
        {
            flag[i * prim[j]] = 1;
            if(i % prim[j] == 0) break;
        }
    }
}

//区间素数
/*要获得区间[L, U]内的素数, L和U很大, 但U - L不大, 那么,
    先线性澜出1到√2147483647 ≤ 46341之间所有的素数, 然后在通过已经澜好的素数澜出给定区间的素数
*/

///素数判定
//试除法: 略过偶数, 试除2到√n间的所有数O(√n)
bool isPrime(int n)
{
    if(n % 2 == 0)return n == 2;
    for(int i = 3; i * i <= n; i += 2)
        if(n % i == 0)
            return false;
    return true;
}
//简单测试: 根据费马小定理p是素数, 则有a^(p−1) ≡ 1(%p), 通过选取[0, p−1]间的任意整数a,
    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
//witness定理:
//Miller_Rabin O(log n)
int qpow(int x, int k, int mod){}
bool witness(int a, int n)
{
    int t = 0, u = n - 1;
    while((u & 1) == 0) t++, u >>= 1;
    int x = qpow(a, u, n), lx;
    for(int i = 1; i <= t; i++)
    {
        lx = x;
        x = x * x % n;
```

```
        if(x == 1 && lx != 1 && lx != n - 1)
            return true;
    }
    return x != 1;
}
bool Miller_Rabin(int n)//出错率为(1/2)^-s
{
    if(n<2) return false;
    if(n == 2 || n == 3 || n == 5 || n == 7) return true;
    if(n % 2 == 0 || n % 3 == 0 || n % 5 == 0 || n % 7 == 0) return false;
    int s = 50;
    while(s--)
        if(witness(rand() % (n - 1) + 1, n))
            return false;
    return true;
}
```

4.2 最大公约数 GCD

```
///最大公约数gcd
/*gcd(a,b)的性质
 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|
 gcd(a,ka) = |a|, (k ∈ Z), gcd(a,b) = n gcd(a,b), gcd(a,gcd(b,c)) = gcd(gcd(a,b),c)
 gcd递归定理: gcd(a,b) = gcd(b,a%b)
 最大公倍数lcm(a,b) = ab/gcd(a,b)
 */
//欧几里得算法O(log n)
//递归
int gcd(int a, int b) {return b ? gcd(b, a % b) : a;}
//循环
int gcd(int a, int b)
{
    int t;
    while(b) t = a % b, a = b, b = t;
    return a;
}
//小数的gcd
//EPS控制精度
double fgcd(double a, double b)
{
    if(-EPS < b && b < EPS) return a;
    return fgcd(b, fmod(a, b));
}
///扩展欧几里得算法
void exgcd(int a, int b, int &g, int &x, int &y)
{
    if(b) exgcd(b, a % b, g, y, x), y -= a / b * x;
    else x = 1, y = 0, g = a;
}
//应用
//1. 求解ax + by = c的x的最小正整数解
int solve(int a, int b, int c)
{
    int g = exgcd(a, b, x, y), t = b / g;
    if(c % g) return -1;//c % gcd(a, b) != 0无解
    int x0 = x * c / g;
    x0 = ((x0 % t) + t) % t;
    int y0 = c - a * x0;
    return x0;
}
//2. 求解a关于p的逆元
//
```

4.3 逆元 Inverse

```
///逆元inverse
//定义：如果a · b ≡ 1(%MOD)，则b 是a的逆元(模逆元， 乘法逆元)
//a的逆元存在条件：gcd(a, MOD) == 1
//性质：逆元是积性函数，如果c = a · b， 则 inv[c] = inv[a] · inv[b]%MOD
//方法一：  循环找解法(暴力)
//O(n)  预处理inv[1-n]：O(n^2)
LL getInv(LL x, LL MOD)
{
    for(LL i = 1; i < MOD; i++)
        if(x * i % MOD == 1)
            return i;
    return -1;
}

//方法二：  费马小定理和欧拉定理
//费马小定理:a^(p-1) ≡ 1(%p)，其中p是质数，所以a的逆元是a^(p-2)%p
//欧拉定理:x^φ(m) ≡ 1(%m) x与m互素，m是任意整数
//O(log n)(配合快速幂)，预处理inv[1-n]：O(n log n)
LL qpow(LL x, LL k, LL MOD) {....}
LL getInv(LL x, LL MOD)
{
    //return qpow(x, euler_phi(MOD) - 1, MOD);
    return qpow(x, MOD - 2, MOD);//MOD是质数
}

//方法三：扩展欧几里得算法
//扩展欧几里得算法可解决  a · x + b · y = gcd(a, b)
//所以a · x%MOD = gcd(a, b)%MOD(b = MOD)
//O(log n)，预处理inv[1-n]：O(n log n)
inline void exgcd(LL a, LL b, LL &g, LL &x, LL &y)
{
    if(b) exgcd(b, a % b, g, y, x), y -= (a / b) * x;
    else g = a, x = 1, y = 0;
}

LL getInv(LL x, LL mod)
{
    LL g, inv, tmp;
    exgcd(x, mod, g, inv, tmp);
    return g != 1 ? -1 : (inv % mod + mod) % mod;
}

//方法四：积性函数
//已处理inv[1] —— inv[n - 1]，求inv[n]， (MOD > n) (MOD为质数,不存在逆元的i干扰结果)
//MOD = x · n - y(0 ≤ y < n) ⇒ x · n = y(%MOD) ⇒ x · n · inv[y] = y · inv[y] = 1(%MOD)
//所以inv[n] = x · inv[y](x = MOD - MOD/n, y = MOD%n)
//O(log n) 预处理inv[1-n]：O(n)
LL inv[NUM];
void inv_pre(LL mod)
{
    inv[0] = inv[1] = 1LL;
    for(int i = 2; i < NUM; i++)
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
}
LL getInv(LL x, LL mod)
{
    LL res = 1LL;
    while(x > 1)
    {
        res = res * (mod - mod / x) % mod;
        x  = mod % x;
    }
```

```
    return res;
}
//方法五：积性函数+因式分解
//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
//采用质因数分解， 可在$O(\log n)$求出任意一个数的逆元
//预处理$O(n\log n)$， 单个$O(\log n)$
```

4.4 模运算 Module

```
/*
 模(Module)
 1. 基本运算
    Add: (a + b) % p = (a % p + b % p) % p
    Subtract:(a - b) % p = ((a%p - b%p)%p + p)%p
    Multiply:(a * b) % p = ((a % p) * (b % p)) % p
    Dvidive: (a / b) % p = (a * $b^{-1}$) % p, $b^{-1}$是b关于p的逆元
    Power: $(a^b)\%p = ((a\%p)^b)\%p$

    对一个数连续取模，有效的取模次数小于$O(\log n)$
 2. 推论
    若$a \equiv b(\%p), c \equiv d(\%p)$,则$(a+c) \equiv (b+d)(\%p), (a-c) \equiv (b-d)(\%p), (a*c) \equiv (b*d)(\%p), (a/c) \equiv (b/d)(\%p)$

 3. 费马小定理
    若p是素数，对任意正整数x,有 $x^p \equiv x(\%p)$.
 4. 欧拉定理
    若p与x互素， 则有$x^{\phi(p)} \equiv 1(\%p)$.
 5. $n! = ap^e, \gcd(a,p) == 1, p$是素数
    $e = (n/p + n/p^2 + n/p^3 + \cdots)$(a不能被p整除)
    威尔逊定理: $(p-1)! \equiv -1(\%p)$(当且仅当p是素数)
    n!中不能被p整除的数的积:$n! = (p-1)!^{(n/p)} \times (n \ mod \ p)!$
    n!中能被p整除的项为:p, 2p, 3p, ... , (n/p)p, 除以p得到1,2,3,...,n/p(问题从缩减到n/p)
    在O(p)时间内预处理除$0 \le n < p$范围内中的n! mod p的表
    可在$O(\log_p n)$时间内算出答案
    若不预处理,复杂度为$O(p\log_p n)$
 */
int fact[MAX_P];//预处理n! mod p的表.$O(p)$
//分解$n! = a \ p^e$.返回a % p. $O(\log_p n)$
int mod_fact(int n, int p, int &e)
{
    e = 0;
    if(n == 0) return 1;
    //计算p的倍数的部分
    int res = mod_fact(n / p, p, e);
    e += n / p;
    //由于$(p-1)! \equiv -1$,因此只需知n/p的奇偶性
    if(n / p % 2) return res * (p - fact[n % p]) % p;
    return res * fact[n % p] % p;
}

/*
 6. $n! = t(p^c)^u, \gcd(t,p^c) == 1, p$是素数
    1 ~ 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
    1 ~ 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
    最后, t还要乘上$p^u$
 */
LL fact[NUM];
LL qpow(LL x, LL k, LL mod);
inline void pre_fact(LL p, LL pc)//预处理$n!\%p^c$, $O(p^c)$
{
    fact[0] = fact[1] = 1;
    for(int i = 2; i < pc; i++)
    {
```

```
        if(i % p) fact[i] = fact[i - 1] * i % pc;
        else fact[i] = fact[i - 1];
    }
}
//分解$n! = t(p^c)^u, n!\%pc = t \cdot p^u\%pc)$
inline void mod_factorial(LL n, LL p, LL pc, LL &t, LL &u)
{
    for(t = 1, u = 0; n; u += (n /= p))
        t = t * fact[n % pc] % pc * qpow(fact[pc - 1], n / pc, pc) % pc;
}
/*
 7. 大组合数求模, mod不是质数
    求$C_n^m\%mod$
    1) 因式分解:$mod = p_1^{c1} p_2^{c2} \cdots p_k^{ck}$
    2) 对每个因子$p^c$, 求$C_n^m\%p^c = \frac{n!\%pc}{m!\%pc(n-m)!\%pc}$
    3) 根据中国剩余定理求解答案(注: 逆元采用扩展欧几里得求法)
 */
LL fact[NUM];
LL prim[NUM], prim_num;
LL pre_prim();
LL pre_fact(LL p, LL pc);
LL mod_factorial(LL n, LL p, LL pc, LL &t, LL &u);
LL qpow(LL x, LL k, LL mod);
LL getInv(LL x, LL mod);

LL C(LL n, LL m, LL mod)
{
    if(n < m) return 0;
    LL p, pc, tmpmod = mod;
    LL Mi, tmpans, t, u, tot;
    LL ans = 0;
    int i, j;
    //将mod因式分解,$mod = p_1^{c1} p_2^{c2} \cdots p_k^{ck}$
    for(i = 0; prim[i] <= tmpmod; i++)
        if(tmpmod % prim[i] == 0)
        {
            for(p = prim[i], pc = 1; tmpmod % p == 0; tmpmod /= p)
                pc *= p;
            //求$C_n^k\%pc$
            pre_fact(p, pc);
            mod_factorial(n, p, pc, t, u);//n!
            tmpans = t;
            tot = u;
            mod_factorial(m, p, pc, t, u);//m!
            tmpans = tmpans * getInv(t, pc) % pc;//求逆元: 采用扩展欧几里得定律
            tot -= u;
            mod_factorial(n - m, p, pc, t, u);//(n - m)!
            tmpans = tmpans * getInv(t, pc) % pc;
            tot -= u;
            tmpans = tmpans * qpow(p, tot, pc) % pc;
            //中国剩余定理
            Mi = mod / pc;
            ans = (ans + tmpans * Mi % mod * getInv(Mi, pc) % mod) % mod;
        }
    return ans;
}

/*
 8. 大组合数求模, mod是素数, Lucas定理
    Lucas定理: $C_n^m\%mod = C_{n/mod}^{m/mod} \cdot C_{n\%mod}^{m\%mod}\%mod$
    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$C_n^k\%mod$
*/
LL fact[NUM], inv[NUM];
void Lucas_init(LL mod);//预处理
```

```
LL Lucas(LL n, LL m, LL mod) //mod是质数
{
    LL a, b, res = 1LL;
    while(n && m)
    {
        a = n % mod, b = m % mod;
        if(a < b) return 0LL;
        res = res * fact[a] % mod * inv[fact[b] * fact[a - b] % mod, mod] % mod;
        n /= mod, m /= mod;
    }
    return res;
}
```

4.5 中国剩余定理和线性同余方程组

```
/*线性同余方程
 a_i × x ≡ b_i(% m_i)  (1 ≤ i ≤ n)
 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.
 对方程逐一求解.令b = 0, m = 1;
 1.x ≡ b(% m)可写为x = b + m·t;
 2.带入第i个式子: a_i(b + m·t) ≡ b_i(% m_i),即a_i·m·t ≡ b_i − a_i·b(% m_i)
 3.当gcd(m_i, a_i·m)无法整除b_i − a_i·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,

中国剩余定理:
    对x ≡ a_i(% m_i)(1 ≤ i ≤ n),其中m_1, m_2, ···, m_n两两互素, a_1, a_2, ···, a_n是任意整数, 则有解:
    M = ∏ m_i, b = ∑_i^n a_i M_i^{-1} M_i (M_i = M/m_i)
*/
int gcd(int a, int b);
int getInv(int x, int mod);
pair<int, int> linear_congruence(const vector<int> &A, const vector<int> &B, const vector<int> &M)
{
    //初始解设为表示所有整数的x ≡ 0(% 1)
    int x = 0, m = 1;
    for(int i = 0; i < A.size(); i++)
    {
        int a = A[i]*m, b = B[i] - A[i] * x, d = gcd(M[i], a);
        if(b % d == 0) return make_pair(0, -1);//无解
        int t = b/d * getInv(a / d, M[i] / d) % (M[i] / d);
        x = x + m * t;
        m *= M[i] / d;
    }
    return make_pair(x % m, m);
}
```

4.6 组合与组合恒等式

```
/*1. 组合: 从n个不同的元素中取r个的方案数C_n^r:
  C_n^r = { n!/(r!(n−r)!),   n ≥ r
          { 1,               n ≥ r = 0
          { 0,               n < r
 推论1: C_n^r = C_n^{n−r}
 推论2(Pascal公式): C_n^r = C_{n−1}^r + C_{n−1}^{r−1}
 推论3: ∑_{k=r−1}^{n−1} C_k^{r−1} = C_{n−1}^{r−1} + C_{n−2}^{r−2} + ... + C_{r−1}^{r−1} = C_n^r
 2. 从重集B = {∞·b_1, ∞·b_2, ..., ∞·b_n}的r−组合数F(n,r)为
    F(n, r) = C_{n+r−1}^r

 3. 二项式定义
    当n是一个正整数时, 对任何x和y有:
```

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

令y=1，有：

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

广义二项式定理：

广义二项式系数：对于任何实数α和整数k，有

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

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

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

推论：令$z=\frac{x}{y}$，则有

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

令$\alpha=-n$(n是正整数)，有

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

又令z = -rz，(r为非零常数)，有

又令n=1，有

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

令z = -z，有

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

令$\alpha=\frac{1}{2}$，有

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

4. 组合恒等式

1. $\sum_{k=0}^{n}C_n^k=2^n$
2. $\sum_{k=0}^{n}(-1)^k C_n^k=0$
3. 对于正整数 n 和 k，
$$C_n^k=\frac{n}{k}C_{n-1}^{k-1}$$
4. 对于正整数 n，
$$\sum_{k=0}^{n}kC_n^k=\sum_{k=1}^{n}kC_n^k=n\cdot 2^{n-1}$$
5. 对于正整数 n，
$$\sum_{k=0}^{n}(-1)^k kC_n^k=0$$
6. 对于正整数 n，
$$\sum_{k=0}^{n}k^2C_n^k=n(n+1)2^{n-2}$$
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 $p \leq minm, n$,

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

9. (令 p=m) 对于任何正整数 n,m,

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

10. (又令 m=n) 对于任何正整数 n,

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

11. 对于非负整数 p,q 和 n,

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

12. 对于非负整数 p,q 和 n,

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

13. 对于非负整数 n,k,

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

14. 对于所有实数 α 和非负整数 k,

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

15.

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

16.

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

17.

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

18.

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

*/

4.7 排列 permutation

/*排列

*排列：从集合A={$a_1, a_2, \cdots, a_n$}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。

* 记其排列数：

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

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

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

*

*圆排列：从集合A={$a_1, a_2, \cdots, a_n$}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。

* 集合A中n个元素的r圆排列的个数为：

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

*

*重排列：从重集B={$k_1 \cdot b_1, k_2 \cdot b_2, \cdots, k_n \cdot b_n$}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。

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

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

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

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

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

* 递归关系式:

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

* 性质:

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

* 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
D_n	265	1845	14833	133496	1334961	14684570
n	12	13	14	15	16	17
D_n	176214841	2290792932	32071101049	481066515734	7697064251745	130850092279664

*相对位置上有限制的排列的问题:
*   求集合$\{1, 2, 3, \cdots, n\}$的不允许出现12,23,34, ... , (n-1)n的全排列数为

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

*   当$n \geq 2$时, 有$Q_n = D_n + D_{n-1}$
*   求集合$\{1, 2, 3, \cdots, n\}$的圆排列中不出现$12, 23, 34, \cdots, (n-1)n, n1$的圆排列个数为:

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

*一般限制的排列:
*   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)
*   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,
            求不同的放法数$r_k(C)$
*   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
*   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
*

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

*   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

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

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

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

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

4.8 母函数 Generating Function

```
/*母函数
 * 普通母函数:
 * 定义: 给定一个无穷序列(a_0, a_1, a_2, ..., a_n, ...)(简记为{a_n}),称函数
```

*

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

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

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

为序列$(a_0, a_1, \cdots, a_n, \cdots)$的指数母函数。
* 常见指数母函数为
* 序列$(1, 1, \cdots, 1, \cdots)$的指数母函数为$f_e(x) = e^x$
* n是整数, 序列$(P_n^0, P_n^1, \cdots, P_n^n)$的指数母函数为$f_e(x) = (1+x)^n$
* 序列$(P_0^0, P_2^1, P_4^2, \cdots, P_{2n}^n, \cdots)$的指数母函数为$f_e(x) = (1-4x)^{-1/2}$
* 序列$(1, \alpha, \alpha^2, \cdots, \alpha^n, \cdots)$的指数母函数为$f_e(x) = e^{\alpha x}$
*
* 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:

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

*
* 母函数的基本运算:
* 设A(x), B(x) , C(x)分别是序列$(a_0, a_1, \cdots, a_r, \cdots), (b_0, b_1, \cdots, b_r, \cdots), (c_0, c_1, \cdots, c_r, \cdots)$的普通 (指数)母函数,则有:
* C(x) = A(x) + B(x) 当且仅当对所有的i, 都有$c_i = a_i + b_i (i = 0, 1, 2, \cdots, r, \cdots)$.
* C(x) = A(x)B(x) 当且仅当对所有的i, 都有$c_i = \sum_{k=0}^{i} a_kb_{i-k} (i = 0, 1, 2, \cdots, r, \cdots)$.
*/

/*母函数在组合排列上的应用

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。

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

故答案为$a_r = C_{n+r-1}^r$
*/

4.9 鸽笼原理与 Ramsey 数

/*鸽笼原理:
* 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.
* 一般形式: 设q_i是正整数$(i = 1, 2, \cdots, n)$, $q \geq q_1 + q_2 + \cdots + q_n - n + 1$, 如果把q个物体放入n个盒子中去, 则存在一个i使得第i个盒子中至少有q_i个物体.
* 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。
* 推论2: 对于正整数$m_i (i = 1, 2, \cdots, n)$, 如果 \frac{\sum_{i=1}^{n}{m_i}}{n} > r − 1,则至少存在一个i, 使得$m_i \geq r$.
* 例: 在给定的n个整数$a_1, a_2, \cdots, a_n$中, 存在k和l$(0 \leq k < l \leq n)$, 使得$a_{k+1} + a_{k+2} + \cdots + a_l$能被n整除
*/

/*Ramsey定理和Ramsey数
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.
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。
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。
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。

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.

```
Ramsey数的性质:
N(a,b) = N(b,a)
N(a,2) = a
当$a,b \geq 2$时, N(a,b)是一个有限数, 并且有$N(a,b) \leq N(a-1,b)+N(a,b-1)$
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		

```
如果把一个完全n角形, 用r中颜色$c_1,c_2,\cdots,c_r$对其边任意着色。
    设$N(a_1,a_2,\cdots,a_r)$是保证下列情况之一出现的最小正整数:
        $c_1$颜色着色的一个完全$a_1$角形
        用$c_2$颜色着色的一个完全$a_2$角形
        .........
        或用颜色$c_r$着色的一个完全$a_r$角形
        则称数$N(a_1,a_2,\cdots,a_r)$为Ramsey数。
    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。
    对于任意正整数m和$a_1,a_2,\cdots,a_m \geq 2$,Ramsey数$N(a_1,a_2,\cdots,a_m)$是存在的。
  .....
 */
```

4.10 容斥原理

```
/*容斥原理
 * 集合S中具有性质$p_i(i=1,2,\cdots,m)$的元素所组成的集合为$A_i$,则S中不具有性质$p_1,p_2,\cdots,p_m$的元素个数为
 * $|\overline{A_1}\cap\overline{A_2}\cap\cdots\overline{A_m}=|S|-\sum_{i=1}^{m}|A_i|+\sum_{i\neq j}A_i\cap A_j-\sum_{i\neq j\neq k}|A_i\cap A_j\cap A_k+\cdots+(-1)^m|A_1\cap A_2\cap\cdots\cap A_m|$
 */
/*重集的r-组合
 * 重集$B=\{k_1\cdot a_1,k_2\cdot a_2,\cdots,k_n\cdot a_n\}$的r-组合数:
 * 利用容斥原理, 求出重集$B'=\{\infty\cdot a_1,\infty\cdot a_2,\cdots,\infty\cdot a_n\}$的r-组合数F(n,r)
 * 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
 * ....
 * 右容斥原理得: 重集B的r-组合数为:
 *
      $$F(n,r)-\sum_{i=1}^{n}F(n,r-k_i-1)+\sum_{i\neq j}F(n,r-k_i-k_j-2)+\cdots+(-1)^nF(n,r-k_1-k_2-\cdots-k_n-n)$$
 */
```

4.11 伪随机数的生成-梅森旋转算法

```
//伪随机数生成——梅森旋转算法(Mersenne twister)
/*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 <= k <= 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*/
//32位算法
int mtrand_init = 0;
int mtrand_index;
int mtrand_MT[624];
void mt_srand(int seed)
{
    mtrand_index = 0;
    mtrand_init = 1;
    mtrand_MT[0] = seed;
    for(int i = 1; i < 624; i++)
    {
        int t = 1812433253 * (mtrand_MT[i - 1] ^ (mtrand_MT[i - 1] >> 30)) + i;//0x6c078965
```

```
        mtrand_MT[i] = t & 0xffffffff;   //取最后的32位赋给MT[i]
    }
}

int mt_rand()
{
    if(!mtrand_init)
        srand((int)time(NULL));
    int y;
    if(mtrand_index == 0)
    {
        for(int i = 0; i < 624; i++)
        {
            //2^31 -1 = 0x7fff ffff  2^31 = 0x8000 0000
            int y = (mtrand_MT[i] & 0x80000000) + (mtrand_MT[(i + 1) % 624] & 0x7fffffff);
            mtrand_MT[i] = mtrand_MT[(i + 397) % 624] ^ (y >> 1);
            if(y & 1) mtrand_MT[i] ^= 2567483615; // 0x9908b0df
        }
    }
    y = mtrand_MT[mtrand_index];
    y = y ^ (y >> 11);
    y = y ^ ((y << 7) & 2636928640);    //0x9d2c5680
    y = y ^ ((y << 15) & 4022730752);   // 0xefc60000
    y = y ^ (y >> 18);
    mtrand_index = (mtrand_index + 1) % 624;
    return y;
}
```

4.12 异或 Xor

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

4.13 博弈论 Game Theory

```
///博弈论 Game Theory
/*
SG-组合游戏
SG-组合游戏定义:    游戏两人参与, 轮流决策, 最优决策;
                当有人无法决策时, 游戏结束, 无法决策的人输;
                游戏可在有限步内结束, 不会多次抵达同一状态, 没有平局;
                游戏某一状态可以到达的状态集合存在且确定.
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.
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.
                必败态(P-position): 对手的必胜态.
                组合游戏中某一状态不是必胜态就是必败态.
                对任意的必胜态, 总存在一种方式转移到必败态.
                对任意的必败态, 只能转移到必胜态.
找出必败态和必胜态: 1. 按照规则, 终止状态设为必败(胜)态.
```

2. 将所有能到达必败态的状态标为必胜态.
3. 将只能到达必胜态的状态标为必败态.
4. 重复2-3, 直到不再产生必败(胜)态.
游戏的和: 考虑任意多个同时进行的SG-组合游戏, 这些SG-组合游戏的和是这样一个SG-组合游戏, 在它进行的过程中, 游戏者可以任意挑选其中的一个单一游戏进行决策, 最终, 没有办法进行决策的人输.
//
SG函数(the Sprague-Grundy function)
定义: 游戏状态为x, sg(x)表示状态x的sg函数值, $sg(x)=\min\{n|n\in N\cap n\notin F(x)\}$, F(x)表示x能够达到的所有状态. 一个状态为必败态则sg(x)=0.
SG定理: 如果游戏G由n个子游戏组成, $G=G_1+G_2+G_3+\cdots+G_n$, 并且第i个游戏sg函数值为sg_i, 则游戏G的sg函数值为$g=sg_1\wedge sg_2\wedge\cdots\wedge sg_n$
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.
//
Nim游戏: 两名游戏者从N堆石子中轮流取石子, 每次任选一堆石子从中取走至少1个石子, 取走最后一个石子的人胜利.
$SG(i)=X(i=1,2,\cdots N)$, 当且仅当$SG(tot)=SG(1)xorSG(2)xor\cdots SG(N)$
定理: 在我们每次只能进行一步操作的情况下, 对于任何的游戏的和, 我们若将其中的任一单一SG-组合游戏换成数目为它的SG 值的一堆石子, 该单一SG-组合游戏的规则变成取石子游戏的规则 (可以任意取, 甚至取完), 则游戏的和的胜负情况不变. 这样, 所有的游戏的和都等价成 nim 游戏.

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.
SJ定理: 对于任意一个 Anti-SG 游戏, 如果我们规定当局面中所有的单一游戏的 SG 值为 0 时, 游戏结束, 则先手必胜当且仅当: (1) 游戏的 SG 函数不为 0 且游戏中某个单一游戏的 SG 函数大于 1; (2) 游戏的 SG 函数为 0 且游戏中没有单一游戏的 SG 函数大于 1.

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.
定义step函数:

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

定理: 对于Every-SG游戏先手必胜当且仅当单一游戏的中最大的step为奇数.

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.
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.

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.
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.

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.
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.
*/

4.14 快速傅里叶变换 FFT

///快速傅里叶变换FFT(Fast Fourier Transformation)
/*原理:
DFT(离散傅里叶变换), 变换公式:

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

W_N被称为旋转因子, 有如下性质:
1. 对称性: $(W_N^{ik})^*=W_N^{-ik}$
2. 周期性: $W_N^{ik}=W_N^{(i+N)k}=W_N^{i(N+k)}$

3. 可约性：$W_N^{ik}=W_{mN}^{mik}, W_N^{ik}=W_{\frac{N}{m}}^{\frac{ik}{m}}$

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

IDFT(DFT逆变换)，变换公式：

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

(基2)FFT推导：

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

所以通过计算

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

可以计算得

$$\begin{cases}X(k) &= X_1(k)+W_N^kX_2(k)\\ X(k+N/2) &= X_1(k)-W_N^kX_2(k)\end{cases}\quad k<N/2$$

DFT变换满足cyclic convolution的性质，即

定义循环卷积c=a(*)b：

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

则有：DFT(a(*)b) = DFT(a) · DFT(b)，所以a(*)b = DFT^{-1}(DFT(a) · DFT(b))

注意：FFT是cyclic的，需要保证高位有足够多的0

FFT算法限制，n必须是2的幂

FFT是定义在复数上的，因此与整数变换可能有精度误差

```
*/
//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
//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
//#incldue <complex>
//typedef std::complex<double> Complex;
struct Complex//复数类，可以直接用STL库中的complex<double>
{
    double r, i;
    Complex(double _r = 0.0, double _i = 0.0) {r = _r, i = _i;}
    Complex operator + (const Complex &b) const {return Complex(r + b.r, i + b.i);}
    Complex operator - (const Complex &b) const {return Complex(r - b.r, i - b.i);}
    Complex operator * (const Complex &b) const
    {
        return Complex(r * b.r - i * b.i, r * b.i + i * b.r);
    }
    double real() {return r;}
    double image() {return i;}
};
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]
{
    int i, j, k;
    for(i = 1, j = N >> 1; i < N - 2; i++)
    {
        if(i < j) swap(p[i], p[j]);
        for(k = N >> 1; j >= k; k >>= 1) j -= k;
        if(j < k) j += k;
    }
}
```

```cpp
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
{
    brc(p, N);
    double p0 = PI * op;
    for(int h = 2; h <= N; h <<= 1, p0 *= 0.5)
    {
        int hf = h >> 1;
        Complex unit(cos(p0), sin(p0));
        for(int i = 0; i < N; i += h)
        {
            Complex w(1.0, 0.0);
            for(int j = i; j < i + hf; j++)
            {
                Complex u = p[j], t = w * p[j + hf];
                p[j] = u + t;
                p[j + hf] = u - t;
                w = w * unit;
            }
        }
    }
}
//Polynomial multiplication多项式相乘X × Y = Z
void polynomial_multi(const vector<int> &a, const vector<int> &b, vector<int> &res, int n)
{
    int N = 1;
    int i = 0;
    while(N < n + n) N <<= 1;//FFT的项数必须是2的幂
    vector<Complex> A(N, Complex(0.0)), B(N, Complex(0.0)), D(N);
    for(i = 0; i < (int)a.size(); i++)A[i] = Complex(a[i], 0.0);
    for(i = 0; i < (int)b.size(); i++) B[i] = Complex(b[i], 0.0);
    FFT(A, N, 1);
    FFT(B, N, 1);
    for(i = 0; i < N; i++) D[i] = A[i] * B[i];
    FFT(D, N, -1);
    for(i = 0, res.clear(); i < N; i++) res.PB(round(D[i].real() / N));
}

/*
应用1：给一个01串S，求有多少对(i, j, k)(i < j < k)使S_i = S_j = S_k = 1，且j - i = k - j
*/


//在整数域上的FFT
/*在整数域上的FFT的推导：
```

0. DFT变换公式：$A(k) = \sum_{i=0}^{N-1} a(i)\varpi^{ik}$

IDFT变换公式：$a(k) = N^{-1}\sum_{i=0}^{N-1} A(i)\varpi^{-ik}$

1. 周期性：由于

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

2. 求和引理：若要实现逆变换，则有：

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

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

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

3. FFT的分治计算：

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

则

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

其中，$0 \leq k < N, 0 \leq k + pN' < N$

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 | N

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

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

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

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$p > \max\{a(i), b(i), c(i)\} (i = 0, 1, 2, \cdots, N-1)$，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)

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

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

适合的p有很多，枚举i，判断(1<<K)*i + 1是否为素数即可

```
*/
//来源：2015多校第三场，1007标程
LL qpow(LL x, LL k, LL mod);
const LL mod = 998244353, wroot = 3;
int wi[NUM << 2];
int fft_init(int n)
{
    int N = 1;
    while(N < n + n) N <<= 1;
    wi[0] = 1, wi[1] = qpow(wroot, (mod - 1) / N, mod);
    for(int i = 2; i < N; i++)
        wi[i] = 1LL * wi[i - 1] * wi[1] % mod;
    return N;
}

```

```cpp
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]
{
    int i, j, k;
    for(i = 1, j = N >> 1; i < N - 2; i++)
    {
        if(i < j) swap(p[i], p[j]);
        for(k = N >> 1; j >= k; k >>= 1) j -= k;
        if(j < k) j += k;
    }
}
void FFT(vector<int> &a, int N, int op)
{
    brc(a, N);
    for(int h = 2; h <= N; h <<= 1)
    {
        int unit = op == -1 ? N - N / h : N / h;
        int hf = h >> 1;
        for(int i = 0; i < N; i += h)
        {
            int w = 0;
            for(int j = i; j < i + hf; j++)
            {
                int u = a[j], t = 1LL * wi[w] * a[j + hf] % mod;
                if((a[j] = u + t) >= mod) a[j] -= mod;
                if((a[j + hf] = u - t) < 0) a[j + hf] += mod;
                if((w += unit) >= N) w -= N;
            }
        }
    }
    if(op == -1)
    {
        int inv = qpow(N, mod - 2, mod);
        for(int i = 0; i < N; i++) a[i] = 1LL * a[i] * inv % mod;
    }
}
```

4.15 莫比乌斯反演 Mobius

```cpp
///莫比乌斯反演 Mobius
//mobius函数
/*
```

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

```cpp
*/
//莫比乌斯反演
//
```

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

```cpp
//
```

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

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

//$\varphi(n) = n \sum_{d|n} \mu(d)/d$

```cpp

//使用1
/*
```

$\sum \mathrm{gcd}(i, j) == D(1 \le i \le a, 1 \le j \le b, a \le b)$，即求gcd(i, j)等于d的对数，$\lfloor x \rfloor$表示下取整

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

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

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

$$\Rightarrow \sum_{d=1}^{\lfloor\frac{a}{D}\rfloor}\mu(d)\lfloor\frac{\lfloor\frac{a}{D}\rfloor}{d}\rfloor\cdot\lfloor\frac{\lfloor\frac{b}{D}\rfloor}{d}\rfloor, d|gcd(i,j) \Leftrightarrow d|i \cup d|j$$

$$D == 1, \sum_{d=1}^{a}\mu(d)\cdot\lfloor\frac{a}{d}\rfloor\cdot\lfloor\frac{b}{d}\rfloor$$

*/

//使用2

/*

$$\sum_{i=1}^{a}\sum_{j=1}^{b}\mathtt{gcd}(i,j), a \le b$$

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

$$\Rightarrow \sum_{d=1}^{a}\sum_{d'=1}^{\lfloor\frac{a}{d}\rfloor} d\cdot\mu(d')\cdot\lfloor\frac{a}{dd'}\rfloor\cdot\lfloor\frac{b}{dd'}\rfloor$$，使用1

$$\Rightarrow \sum_{d=1}^{a}\sum_{d|D} d\cdot\mu(\frac{D}{d})\cdot\lfloor\frac{a}{D}\rfloor\cdot\lfloor\frac{b}{D}\rfloor, D = dd'$$

$$\Rightarrow \sum_{D=1}^{a}\lfloor\frac{a}{D}\rfloor\cdot\lfloor\frac{b}{D}\rfloor\cdot(id\cdot\mu)(D)$$

$$\Rightarrow \sum_{D=1}^{a}\lfloor\frac{a}{D}\rfloor\cdot\lfloor\frac{b}{D}\rfloor\cdot\varphi(D),\quad id\cdot\mu = \varphi$$

*/

///积性函数

//定义在正整数集上的函数$f(n)$(称为算术函数)，若gcd$(a,b) = 1$时有$f(a)\cdot f(b) = f(a\cdot b)$，则称$f(x)$为积性函数。

//一个显然的性质：(非恒等于零的) 积性函数$f(n)$必然满足$f(1) = 1$.

//定义逐点加法$(f+g)(n) = f(x)+g(x), f(x\cdot g) = f(x)\cdot g(x)$.

//一个比较显然的性质：若f, g均为积性函数，则$f\cdot g$也是积性函数。

//积性函数的求值：$n = \prod p_i^{a_i}$ 则$f(n) = \prod f(p_i^{a_i})$，所以只要解决$n = p^a$时$f(n)$的值即可.

//常见积性函数有:

//恒为1的常函数$1(n) = 1$,

//恒等函数$id(n) = n$,

//单位函数$\varepsilon(n) = (n == 1)$，(这三个都是显然为积性)

//欧拉函数$\varphi(n)$(只要证两个集合相等就能证明积性)

//莫比乌斯函数$\mu(n)$ (由定义也是显然的)

//$\mu\cdot id = \varphi$

```
void pre_mobius()
{
    mu[1] = 1;
    for(int i = 2; i < NUM; i++)
        if(!mu[i])
        {
            mu[i] = -1;
            for(int j = i + i; j < NUM; j += i)
                if((j / i) % i == 0)
                    mu[j] = 2;
                else
                {
                    if(mu[j] == 0) mu[j] = -1;
                    else mu[j] = -mu[j];
```

```
                }
        }
        else if(mu[i] == 2 || mu[i] == -2) mu[i] = 0;
}
```

4.16 矩阵的基本运算 Matrix

```
///矩阵 Matrix
const int Matrix_N = 1010, Matrix_M = 1010;
//矩阵类，适用于递推关系式的快速求值
struct Matrix
{
    int n, m;//矩阵的行数和列数
    int a[Matrix_N][Matrix_M];
    void clear()//将矩阵清0
    {
        n = m = 0;
        memset(a, 0, sizeof(a));
    }
    void I()//将矩阵化为单位矩阵，对方阵有效
    {
        memset(a, 0, sizeof(a));
        for(int i = 0; i < n; i++) a[i][i] = 1;
    }
    //实现矩阵加法C = A + B, (A.n = B.n, A.m = B.m)O(n^2)
    Matrix operator + (const Matrix &b) const
    {
        Matrix c;
        c.n = n;
        c.m = m;
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                c.a[i][j] = a[i][j] + b.a[i][j];
        return c;
    }
    //实现矩阵的减法C = A - B(A.n = B.n, A.m = B.m), O(n^2)
    Matrix operator - (const Matrix &b) const
    {
        Matrix c;
        c.n = n;
        c.m = m;
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++)
                c.a[i][j] = a[i][j] - b.a[i][j];
        return c;
    }
    //实现矩阵的乘法C = A × B, (A.m = B.n)O(n^3)
    Matrix operator * (const Matrix &b) const
    {
        Matrix c;
        c.n = n;
        c.m = b.m;
        for(int i = 0; i < n; i++)
            for(int j = 0; j < b.m; j++)
            {
                c.a[i][j] = 0;
                for(int k = 0; k < m; k++)
                    c.a[i][j] += a[i][j] * b.a[i][j];//如果要取模， 要修改这里
            }
        return c;
    }
    //实现矩阵的快速幂O(n^3 log(k)，要求是方阵
```

```
    Matrix operator ^(int k)
    {
        Matrix res, tmp = *this;
        res.n = res.m = n;
        res.I();
        while(k)
        {
            if(k&1) res = res * tmp;
            tmp = tmp * tmp;
            k >>= 1;
        }
        return res;
    }
};
//矩阵类 适用与求矩阵的逆与高斯消元等场合
//行的初等变换
typedef vector<double> VD;
VD operator * (const VD &a, const double b)
{
    int _n = a.size();
    VD c(_n);
    for(int i = 0; i < _n; i++)
        c[i] = a[i] * b;
    return c;
}
VD operator - (const VD &a, const VD &b)
{
    int _n = a.size();
    VD c(_n);
    for(int i = 0; i < _n; i++)
        c[i] = a[i] - b[i];
    return c;
}
VD operator + (const VD &a, const VD &b)
{
    int _n = a.size();
    VD c(_n);
    for(int i = 0; i < _n; i++)
        c[i] = a[i] + b[i];
    return c;
}
struct Matrix
{
    int n, m;
    VD *a;
    void Matrix(int _n = Matrix_N, int _m = Matrix_M)
    {
        n = _n, m = _m;
        a = new VD[n];
        for(int i = 0; i < n; i++)
            a[i].resize(m, 0);
    }
    void ~Matrix()
    {
        delete []a;
    }
    void clear()//0矩阵
    {
        for(int i = 0; i < n; i++)
            a[i].assign(0);
    }
    void I()//单位矩阵
    {
        clear();
```

```
        for(int i = 0; i < n; i++)
            a[i][i] = 1;
    }
    //矩阵加法，同上
    Matrix operator + (const Matrix &b) const
    {
        Matrix c(n, m);
        for(int i = 0; i < n; i++)
            c.a[i] = a[i] + b.a[i];
        return c;
    }
    Matrix operator - (const Matrix &b) const //矩阵减法
    {
        Matrix c(n, m);
        for(int i = 0; i < n; i++)
            c.a[i] = a[i] - b.a[i];
        return c;
    }
    Matrix operator * (const Matrix &b) const //矩阵乘
    {
        Matrix c(n, b.m);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < b.m; j++)
            {
                c[i][j] = 0;
                for(int k = 0; k < m; k++)
                    c.a[i][j] += a[i][j] * b.a[i][j];
            }
        return c;
    }

    //实现求矩阵的逆O(n^3)
    //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，
     将会的到(I,A^-1)的形式
    //注意：
    Matrix inverse()
    {
        Matrix c;
        c.I();
        for(in i = 0; i < n; i++)
        {
            for(int j = i; j < n; j++)
                if(fabs(a[j][i]) > 0)
                {
                    swap(a[i], a[j]);
                    swap(c[i], c[j]);
                    break;
                }
            c[i] = c[i] * (1.0 / a[i][i]);
            a[i] = a[i] * (1.0 / a[i][i]);
            for(int j = 0; j < n; j++)
                if(j != i && fabs(a[j][i]) > 0)
                {
                    c[j] = c[j] - a[i] * a[j][i];
                    a[j] = c[j] - a[i] * a[j][i];
                }
        }
    }
};
//Guass消元
int Guass(double a[][MAXN], bool l[], double ans[], int n)
{//l, ans储存解，l[]表示是否是自由元
    int res = 0, r = 0;
    for(int i = 0; i < n; i++) l[i] = false;
```

```cpp
    for(int i = 0; i < n; i++)
    {
        for(int j = r; j < n; j++)
            if(fabs(a[j][i]) > EPS)
            {
                for(int k = i; k <= n; k++)
                    swap(a[j][k], a[r][k]);
                break;
            }
        if(fabs(a[r][i]) < EPS)
        {
            ++res;
            continue;
        }
        for(int j = 0; j < n; j++)
            if(j != r && fabs(a[j][i]) > EPS)
            {
                double tmp = a[j][i] / a[r][i];
                for(int k = i; k <= n; k++)
                    a[j][k] -= tmp * a[r][k];
            }
        l[i] = true;
        ++r;
    }
    for(int i = 0; i < n; i++)
        if(l[i])
            for(int j = 0; j < n; j++)
                if(fabs(a[j][i]) > 0)
                    ans[i] = a[j][n] / a[j][i];
    return res;//返回解空间的维数
}
//常系数线性齐次递推
```

/*已知$f_x = a_0 f_{x-1} + a_1 f_{x-2} + \cdots a_{n-1} f_{x-n}$和$f_0, f_1, \cdots, f_{n-1}$，给定t，求$f_t$

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β，即

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

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

*/

4.17 一些数学知识

//1. 格雷码：(相邻码之间二进制只有一位不同)，构造方法：$a_i = i^(i >> 1)(a_i$为求第i个格雷码)

/*2. 多边形数：可以排成正多边形的整数

第n个s边形数的公式是：$\frac{n[(s-2)n-(s-4)]}{2}$

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

*/

//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.

//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

```
//manacher算法 O(n)
/*写法一
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；
    在字符串之前再加一个分界符（如'&'），防止比较时越界*/

void manacher(char *s, int len, int p[])
{//s = &s[0]#s[1]#...#s[len]\0
    int i, mx = 0, id;
    for(i = 1; i <= len; i++)
    {
        p[i] = mx > i ? min(p[2*id - i], mx - i) : 1;
        while(s[i + p[i]] == s[i - p[i]]) ++p[i];
        if(p[i] + i > mx) mx = p[i] + (id = i);
        p[i] -= (i & 1) != (p[i] & 1);//去掉分隔符带来的影响
    }
    //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
    //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
}

/*写法二
将位置在[i,j]的回文串的长度信息存储在p[i+j]上
*/
void manacher2(char *s, int len, int p[])
{
    p[0] = 1;
    for(int i = 1, j = 0; i < (len<<1) - 1; ++i)
    {
        int u = i >> 1, v = i - u, r = ((j + 1) >> 1) + p[j] - 1;
        int u = i >> 1, v = i - u, r = ((j + 1) >> 1) + p[j] - 1;
        p[i] = r < v ? 0 : min(r - v + 1, p[(j<<1) - 1]);
        while(u > p[i] - 1 && v + p[i] < len && s[u - p[i]] == s[u + p[i]]) ++p[i];
        if(u + p[i] - 1 > r) j = i;
    }
}
```

5.2 后缀数组 Suffix Array

```
///后缀数组(Suffix Array)
/*
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
*/
//计算后缀数组
//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
///Manber-Myers O(n log^2 n)
int rk;
int sa[NUM], rk[NUM], height[NUM];
int cmp(int i, int j)
{
    if(rk[i] != rk[j])
        return rk[i] < rk[j];
    else
    {
        int ri = i + rk <= n ? rk[i + rk] : -1;
        int rj = i + rk <= n ? rk[j + rk] : -1;
        return ri < rj;
    }
```

```
}
void da(int *a, int n)
{
    int i;
    a[n] = -1;
    for(i = 0; i <= n; i++)
    {
        sa[i] = i;
        rk[i] = a[i];
    }
    for(m = 1; rk[n] < n; m <<= 1)
    {
        sort(sa, sa + n + 1, cmp);
        tmp[sa[0]] = 0;
        for(i = 1; i <= n; i++)
            tmp[sa[i]] = tmp[sa[i - 1]] + (cmp(sa[i], sa[i - 1] || cmp(sa[i - 1], sa[i])));
        for(i = 0; i <= n; i++)
            rk[i] = tmp[i];
    }
}

//应用
//基于后缀数组的字符串匹配
bool contain(string s, int *sa, string t)
{
    int a = 0, b = s.length();
    while(b - a > 1)
    {
        int c = (a + b) / 2;
        if(s.compare(sa[c], t.length(), t) < 0)a = c;
        else
            b = c;
    }
    return s.compare(sa[b], t.length(), t) == 0;
}

///倍增法模板：O(n log n)采用基数排数
//n为字符个数 r[n - 1] 要比所有a[0, n - 2]要小
//r 字符串对应的数组
//m为最大字符值+1
int sa[NUM];
int rk[NUM], height[NUM], sv[NUM], sn[NUM];
void da(char r[], int n, int m)
{
    int i, j, p, *x = rk, *y = height;
    for(i = 0; i < m; i++) sn[i] = 0;
    for(i = 0; i < n; i++) sn[x[i] = r[i]]++;
    for(i = 1; i < m; i++) sn[i] += sn[i - 1];
    for(i = n - 1; i >= 0; i--) sa[--sn[x[i]]] = i;
    for(j = p = 1; p < n; j <<= 1, m = p)
    {
        for(p = 0, i = n - j; i < n; i++) y[p++] = i;
        for(i = 0; i < n; i++) if(sa[i] >= j) y[p++] = sa[i] - j;
        for(i = 0; i < n; i++) sv[i] = x[y[i]];
        for(i = 0; i < m; i++) sn[i] = 0;
        for(i = 0; i < n; i++) sn[sv[i]]++;
        for(i = 1; i < m; i++) sn[i] += sn[i - 1];
        for(i = n - 1; i >= 0; i--) sa[--sn[sv[i]]] = y[i];
        for(swap(x, y), x[sa[0]] = 0, i = 1, p = 1; i < n; i++)
            x[sa[i]] = (y[sa[i]] == y[sa[i - 1]] && y[sa[i] + j] == y[sa[i - 1] + j]) ? p - 1 : p++;
    }
}

///DC3模板：O(3n)
```

```
int sa[NUM * 3], r[NUM * 3]; //sa数组和r数组要开三倍大小的空间
int rk[NUM], height[NUM], sn[NUM], sv[NUM];
#define F(x) ((x) / 3 + ((x) % 3 == 1 ? 0 : tb))
#define G(x) ((x) < tb ? (x) * 3 + 1 : ((x) - tb) * 3 + 2)
int cmp0(int r[], int a, int b)
{return r[a] == r[b] && r[a + 1] == r[b + 1] && r[a + 2] == r[b + 2];}
int cmp12(int r[], int a, int b, int k)
{
    if(k == 2) return r[a] < r[b] || (r[a] == r[b] && cmp12(r, a + 1, b + 1, 1));
    else return r[a] < r[b] || (r[a] == r[b] && sv[a + 1] < sv[b + 1]);
}
void sort(int r[], int a[], int b[], int n, int m)//基数排序
{
    int i;
    for(i = 0; i < m; i++) sn[i] = 0;
    for(i = 0; i < n; i++) sn[sv[i] = r[a[i]]]++;
    for(i = 1; i < m; i++) sn[i] += sn[i - 1];
    for(i = n - 1; i >= 0; i--) b[--sn[sv[i]]] = a[i];
}
void dc3(int r[], int sa[], int n, int m)
{
    int *rn = r + n, *san = sa + n, *wa = height, *wb = rk;
    int i, j, p, ta = 0, tb = (n + 1) / 3, tbc = 0;
    r[n] = r[n + 1] = 0;
    for(i = 0; i < n; i++) if(i % 3 != 0) wa[tbc++] = i;
    sort(r + 2, wa, wb, tbc, m);
    sort(r + 1, wb, wa, tbc, m);
    sort(r, wa, wb, tbc, m);
    for(p = 1, rn[F(wb[0])] = 0, i = 1; i < tbc; i++)
        rn[F(wb[i])] = cmp0(r, wb[i - 1], wb[i]) ? p - 1 : p++;
    if(p < tbc) dc3(rn, san, tbc, p);
    else for(i = 0; i < tbc; i++) san[rn[i]] = i;
    for(i = 0; i < tbc; i++) if(san[i] < tb) wb[ta++] = san[i] * 3;
    if(n % 3 == 1) wb[ta++] = n - 1;
    sort(r, wb, wa, ta, m);
    for(i = 0; i < tbc; i++) sv[wb[i] = G(san[i])] = i;
    for(i = 0, j = 0, p = 0; i < ta && j < tbc; p++)
        sa[p] = cmp12(r, wa[i], wb[j], wb[j] % 3) ? wa[i++] : wb[j++];
    for(; i < ta; p++) sa[p] = wa[i++];
    for(; j < tbc; p++) sa[p] = wb[j++];
}

///高度数组
//height[i] = suffix(sa[i])和suffix(sa[i - 1])的最长公共前缀lcp(sa[i],sa[i-1])
//rk[0..n-1]:rk[i]保存的是原串中suffix[i]的名次
//height数组性质:
//任意两个suffix(j)和suffix(k)(rk[j] < rk[k])的最长公共前缀: min(i=j+1-->k){height[rk[i]]}
//height[i] >= height[i - 1] - 1
int rk[maxn], height[maxn];
void calheight(char *r, int *sa, int n)
{
    int i, j, k = 0;
    for(i = 0; i < n; i++)rk[sa[i]] = i;
    for(i = 0; i < n; height[rk[i++]] = k)
        for(k ? k-- : 0, j = sa[rk[i] - 1]; r[i + k] == r[j + k]; k++);
}
///后缀数组应用
//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]]}
//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
//可重叠最长重复子串: O(n) height数组中的最大值
//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.
```

```
//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.
//不相同的子串个数：等价于所有后缀之间不相同的前缀的个数O(n)：后缀按suffix(sa[1]), suffix(sa[2]), ···, suffix(n)的顺序计算，新进一个后缀suffix(sa[k])，将产生n − sa[k] + 1的新的前缀，其中height[k]的和前面是相同的，所以suffix(sa[k])贡献n − sa[k] + 1 − height[k]个不同的子串. 故答案是∑_{k=1}^{n} n − sa[k] − 1 − height[k].
//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lcp(suffix(k), suffix(2*n + 2 − k)); 偶数lcp(suffix(k), suffix(2*n + 3 − k)) O(n log n) RMQ:O(n)
//连续重复子串：字符串L是有字符串S重复R次得到的.
//给定L，求R的最大值：O(n)，枚举S的长度k，先判断L的长度是否能被k整除，在看lcp(suffix(1), suffix(k+1))是否等于n − k. 求解时只需预处理height数组中的每一个数到height[rk[1]]的最小值即可
//给定字符串，求重复次数最多的连续重复子串O(n log n)：先穷举长度L，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], ···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.
//字符串A和B最长公共前缀O(|A|+|B|)：新串：A+特殊字符+B, height//k : A++B, 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.

//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.
//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).
//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

```
///字典数 Trie
struct Trie
{
    Trie *next[26];//根据字符集的大小变化
    int cnt;//字符串个数，根据应用变化
    Trie()
    {
        memset(next, 0, sizeof(next));
        cnt = 0;
    }
};
Trie *root = NULL;
void clearTrie(Trie *p)//清空整棵字典树
{
    if(p)
    {
        for(int i = 0; i < 26; i++)
            clearTrie(p->next[i]);
        delete p;
    }
}
void initTrie()
{
    clearTrie(root);
    root = new root;
```

```
}
//将字符串str加入字符串中
void createTrie(char str[])
{
    int i = 0, id;
    Trie *p = root;
    while((id = str[i] - 'a') >= 0)//得到该字符的编号
    {
        if(p->next[id] == NULL)
        {
            p->next[id] = new Trie;
        }
        p = p->next[id];
    }
    p->cnt++;
}
//查询某字符串是否在字典树中
bool findTrie(char str[])
{
    int i = 0, id;
    Trie *p = root;
    while((id = str[i++] - 'a') >= 0)
    {
        p = p->next[id];
        if(p == NULL) return false;
    }
    return p->cnt > 0;
}
//从字典树中删除一个字符串
void deleteTrie(char str[])
{
    int i = 0, id;
    Trie *p = root;
    while((id = str[i++] - 'a') >= 0)
    {
        p = p->next[id];
        if(p == NULL) return ;
    }
    if(p->cnt) p->cnt--;
}
```

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

6 计算几何

6.1 计算几何基础

```
//精度设置
const double EPS = 1e-6;
int sgn(double x)
{
    if(x < -EPS)return -1;
    return x > EPS ? 1 : 0;
}
//点（向量)的定义和基本运算
struct Point
{
    double x, y;
    Point(double _x = 0.0, double _y = 0.0):x(_x), y(_y){}
    Point operator + (Point &b)//向量加法
    {
        return Point(x + b.x, y + b.y);
    }
    Point operator - (Ponit &b)//向量减法
    {
        return Point(x - b.x, y - b.y);
    }
    Point operator * (double b)//标量乘法
    {
        return Point(x*b, y*b);
    }
    double operator * (Point &b)//向量点积 a·b = |a||b| cos θ点积为0，表示两向量垂直
    {
        return x*b.x + y*b.y;
    }
    /*向量叉积 a × b = |a||b| sin θ
     *  叉积小于0，表示向量b在当前向量顺时针方向
     *  叉积等于0，表示两向量平行
     *  叉积大于0，表示向量b在当前向量逆时针方向
     */
    double operator ^ (Point b)
    {
        return x * b.y - y * b.x;
    }
    Point rot(double ang)
    {//向量逆时针旋转ang弧度
        return Point(x*cos(ang) - y*sin(ang), x*sin(ang) + y*cos(ang));
    }
};
//直线 线段定义
//直线方程：两点式：(x2 - x1)(y - yy1) = (y2 - y1)(x - x1)
struct Line
{
    Point s, e;
    double k;
    Point(){}
    Point(Point _s, Point _e)
    {
        s = _s, e = _e;
        k = atan2(e.y - s.y, e.x - s.x);
    }
    //求两直线交点
    //返回-1两直线重合，0 相交， 1 平行
    pair<int, Point> operator &(Line &b)
    {
        if(sgn((s - e)^(b.s - b.e)) == 0)
```

```
        {
            if(sgn((s - b.e) ^ (b.s - b.e)) == 0)
                return make_pair(-1, s);//重合
            else
                return make_pari(1, s);//平行
        }
        double t = ((s - b.s)^(b.s - b.e)) / ((s - e)^(b.s - b.e));
        return Point(s.x + (e.x - s.x)*t, s.y + (e.y - s.y)*t);
    }
};

//两点间距离
double dist(Point &a, Point &b)
{
    return sqrt((a - b) * (a - b));
}

/*判断点p在线段l上
 * (p - l.s) ^ (l.s - l.e) = 0; 保证点p在直线L上
 * p在线段l的两个端点l.s,l.e为对角定点的矩形内
 */
bool Point_on_Segment(Point &p, Line &l)
{
    return sgn((p - l.s) ^ (l.s - l.e)) == 0 &&
        sgn((p.x - l.s.x) * (p.x - l.e.x)) <= 0 &&
        sgn((p.y - l.s.y) * (p.y - l.e.y)) <= 0;
}
//判断点p在直线l上
bool Point_on_Line(Point &p, Line &l)
{
    return sgn((p - l.s)^(l.s - l.e)) == 0;
}

/*判断两线段l1, l2相交
 * 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
 * 2. 跨立实验: l2的两个端点是否在线段l1的两端
 */
bool seg_seg_inter(Line seg1, Line seg2)
{
    return
        sgn(max(seg1.s.x, seg1.e.x) - min(seg2.s.x, seg2.e.x)) >= 0 &&
        sgn(max(seg2.s.x, seg2.e.x) - min(seg1.s.x, seg1.e.x)) >= 0 &&
        sgn(max(seg1.s.y, seg1.e.y) - min(seg2.s.y, seg2.e.y)) >= 0 &&
        sgn(max(seg2.s.y, seg2.e.y) - min(seg1.s.y, seg1.e.y)) >= 0 &&
        sgn((seg2.s - seg1.e) ^ (seg1.s - seg1.e)) * sgn((seg2.e - seg1.e) ^ (seg1.s - seg1.e)) <=
    0 &&
        sgn((seg1.s - seg2.e) ^ (seg2.s - seg2.e)) * sgn((seg1.e - seg2.e) ^ (seg2.s - seg2.e)) <=
    0;
}

//判断直线与线段相交
bool seg_line_inter(Line &line, Line &seg)
{
    return sgn((seg.s - line.e) ^ (line.s - line.e)) * sgn((seg.e - line.e) ^ (line.s - line.e)) <=
    0;
}

//点到直线的距离, 返回垂足
Point Point_to_Line(Point p, Point l)
{
    double t = ((p - l.s) * (l.e - l.s)) / ((l.e - l.s) * (l.e - l.s));
    return Point(l.s.x + (l.e.x - l.s.x) * t, l.s.y + (l.e.y - l.s.y) * t);
}
//点到线段的距离
```

```
//返回点到线段最近的点
Point Point_to_Segment(Point p, Line seg)
{
    double t = ((p - l.s) * (l.e - l.s)) / ((l.e - l.s) * (l.e - l.s));
    if(t >= 0 && t <= 1)
        return Point(l.s.x + (l.e.x - l.s.x) * t, l.s.y + (l.e.y - l.s.y) * t);
    else if(sgn(dist(p, l.s) - dist(p, l.e) <= 0))
        return l.s;
    else
        return l.e;
}
```

6.2 多边形

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

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

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

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

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

```
 * 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
```

* 两条垂直平分线: $(x - \frac{x_A+x_B}{2})(x_A - x_B) = -(y_A - y_B)(y - \frac{y_A+y_B}{2})$

* 和 $(x - \frac{x_B+x_C}{2})(x_B - x_C) = -(y_B - y_C)(y - \frac{y_B+y_C}{2})$

```
 *            外心坐标:
```

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

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

```
 * 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);
 * 内切圆: 内心: 角平分线的交点, 半径r:内心到边的距离
 *
 * 三角形的质心: 三条高的交点: Q = (A+B+C)*(1.0/3.0)
 */

 //2. 多边形
 /*判断点在多边形内外
 */
 /*4. 圆
 */
```

6.3 凸包 ConvexHull

```
//凸包Convex Hull
//

//Graham算法O(nlog n)
//写法一: 按直角坐标排序
//直角坐标序比较 (水平序)
bool cmp(Point a, Point b)//先比较x, 后比较x均可
{
    if(sgn(a.x - b.x)) return sgn(a.x - b.x) < 0;
```

```
    return sgn(a.y - b.y) < 0;
}

vector<Point> graham(Point p[], int pnum)
{
    sort(p, p + pnum, cmp);
    vector<Point> res(2 * pnum + 5);
    int i, total = 0, limit = 1;
    for(i = 0; i < pnum; i++)//扫描下凸壳
    {
        while(total > limit && sgn((res[total - 1] - res[total - 2]) ^ (p[i] - res[total - 1])) <= 0) total--;
        res[total++] = p[i];
    }
    limit = total;
    for(i = pnum - 2; i >= 0; i--)//扫描上凸壳
    {
        while(total > limit && sgn((res[total - 1] - res[total - 2]) ^ (p[i] - res[total - 1])) <= 0) total--;
        res[total++] = p[i];
    }
    if(total > 1)total--;//最后一个点和第一个点一样
    res.resize(total);
    return res;
}
//写法二: 按极坐标排序
Point p0;//p0 原点集中最左下方的点
int top;
bool cmp(point p1, point p2) //极角排序函数 , 角度相同则距离小的在前面
{
    int tmp = (p1 - p2) ^ (p0 - p2);
    if(tmp > 0) return true;
    else if(tmp == 0 && (p0 - p1) * (p0 - p1) < (p0 - p2) * (p0 - p2)) return true;
    else return false;
}

vector<Point> graham(Point p[], int pn)
{
    //p0
    for(int i = 1; i < pn; i++)
        if(p[i].x < p[0].x || (p[i].x == p[0].x && p[i].y < p[0].y))
            swap(p[i], p[0]);
    p0 = p[0];
    //sort
    sort(p + 1, p + pn);
    vector<Point> stk(pn * 2 + 5);
    int top = 0;
    for(i = 0; i < n; i++)
    {
        while(top > 1 && ((stk[top - 1] - stk[top - 2]) ^ (p[i] - stk[top - 2])) <= 0) top--;
        stk[top++] = p[i];
    }
    stk.resize(top);
    return stk;
}
```

6.4 立体几何

7 搜索等

```
///二分搜索
//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
//如f(x) = 1 (x <= y) = 0 (x > y), 可以二分搜索出分界值y
//注意: l%2 == 0, r = l + 1时, (l + r)/2 == l 此处易出现死循环
int binary_search(int l, int r)
{
    int mid;
    while(l + 1 < r)
    {
        mid = (l + r) >> 1;
        if(f(mid))
            r = mid;//视情况定
        else
            l = mid;
    }
    for(; l <= r; l++)
        if(f(l))
            return l;
}
///三分搜索
//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
//方便于求(抛物线)的最值
//注意: l % 3 == 0, r = l + 1 | l + 2时, (l + l + r) / 3 == l 容易出现死循环
int three_search(int l, int r)
{
    int ll, rr;
    while(l + 2 < r)
    {
        ll = (l + l + r) / 3;
        rr = (l + r + r) / 3;
        if(f(ll) < f(rr))
            r = rr;
        else
            l = ll;
    }
    return min(f(l), f(r), f(l + 1));
}
```

8 分治

```
///分治
//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
//应用1: 二维偏序求LIS
```

9 Java

```
import java.io.*;
import java.util.*;
import java.math.*;
import java.BigInteger;

public class Main{
    public static void main(String arg[]) throws Exception{
        Scanner cin = new Scnner(System.in);

        BigInteger a, b;
        a = new BigInteger(”123”);
        a = cin.nextBigInteger();
        a.add(b);//a + b
        a.subtract(b);// a – b
        a.multiply(b);// a * b
        a.divide(b);// a / b
        a.negate();// –a
        a.remainder(b);//a%b
        a.abs();//|a|
        a.pow(b);//a^b
        //.... and other math fuction, like log();
        a.toString();
        a.compareTo(b);//

    }
}
```