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 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]);
}
```

1.3 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 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() {}
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
        }
}
```

3.2 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;
};
```

```cpp
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 $O(|E| \cdot |V|^2)$
//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;
}
```

3.3 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;
}
```

3.4 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 $\{\lambda = f(\overrightarrow{x}) = \frac{a(\overrightarrow{x})}{b(\overrightarrow{x})}\}(\overrightarrow{x} \in S, \forall \overrightarrow{x} \in S, b(\overrightarrow{x}) > 0)$
    其中解向量$\overrightarrow{x}$在解空间S内, $a(\overrightarrow{x})$与$b(\overrightarrow{x})$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:$g(\lambda) =$ min $\{a(\overrightarrow{x}) - \lambda \cdot b(\overrightarrow{x})\}(\overrightarrow{x} \in S)$
    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$g(\lambda_1) > g(\lambda_2)$.
    (Dinkelbach定理) 设$\lambda^*$为原规划的最优解, 则$g(\lambda) = 0$当且仅当$\lambda = \lambda^*$.
    设$\lambda^*$为该优化的最优解, 则:
```

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

```
    由该性质, 就可以对最优解$\lambda^*$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$\overrightarrow{x}$满足$\forall x_i \in \{0,1\}$, 即一下形式:
```

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

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

3.5 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$V' \in V$, 满足对于$\forall u \in V'$引出的$\forall < u, v > \in E$, 必有$v \in V'$成立.
     还有一种等价定义为: 满足对于$\forall < u, v > \in E$ , 若有$u \in V'$成立, 必有$v \in V'$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$w_v$(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$\sum_{v \in V'} w_v$.
*/
/*转化为最小割模型$G_N(V_N, E_N)$
```

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.

*/

3.6 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;
}
```

3.7 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).
*/
```

3.8 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];
```

```cpp
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 逆元 Inverse

```
///逆元inverse
//定义：如果$a \cdot b \equiv 1(\%MOD)$，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$c = a \cdot b$，则 $inv[c] = inv[a] \cdot inv[b]\%MOD$
//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:$a^{(p-1)} \equiv 1(\%p)$，其中p是质数，所以a的逆元是$a^{(p-2)}\%p$
//欧拉定理:$x^{\phi(m)} \equiv 1(\%m)$ 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 $a \cdot x + b \cdot y = gcd(a, b)$
//所以$a \cdot x\%MOD = gcd(a, b)\%MOD(b = MOD)$
//$O(\log n)$，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) g = a, x = 1, y = 0;
    else exgcd(b, a % b, g, y, x), y -= (a / b) * x;
}

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)
//$MOD = x \cdot n - y(0 \leq y < n) \Rightarrow x \cdot n = y(\%MOD) \Rightarrow x \cdot n \cdot inv[y] = y \cdot inv[y] = 1(\%MOD)$
//所以$inv[n] = x \cdot inv[y](x = MOD - MOD/n, y = MOD\%n)$
//$O(\log n)$ 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)
```

4.2 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 ≡ b(%p), c ≡ d(%p),则(a + c) ≡ (b + d)(%p), (a - c) ≡ (b - d)(%p), (a * c) ≡ (b * d)(%p), (a/c) ≡ (b/d)(%p)

 3. 费马小定理
    若p是素数， 对任意正整数x,有 x^p ≡ x(%p).
 4. 欧拉定理
    若p与x互素， 则有x^φ(p) ≡ 1(%p).
 5. n! = ap^e, gcd(a, p) == 1, p是素数
    e = (n/p + n/p^2 + n/p^3 + ···)(a不能被p整除)
    威尔逊定理: (p - 1)! ≡ -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) × (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 ≤ 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)! ≡ -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;
}
```

4.3 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);
}
```

4.4 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

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

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

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

推论：令$z=\frac{x}{y}$,则有

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

令$\alpha=-n$(n是正整数),有

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

又令z = −rz，(r为非零常数),有

又令n=1,有

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

令z = −z，有

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

令$\alpha=\frac{1}{2}$，有

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

4. 组合恒等式

1. $\sum_{k=0}^{n}C_n^k=2^n$
2. $\sum_{k=0}^{n}(-1)^k C_n^k=0$
3. 对于正整数 n 和 k,

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

4. 对于正整数 n,

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

5. 对于正整数 n,

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

6. 对于正整数 n,

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

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 $p \leq minm, n$,

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

9. (令 p=m) 对于任何正整数 n,m,

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

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,

$$\sum_{k=0}^{p}C_p^kC_q^kC_{n+k}^{p+q}=C_n^pC_n^q$$

12. 对于非负整数 p,q 和 n,

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

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.

$$\sum_{k=m}^{n}C_k^mC_n^k=C_n^m2^{n-m}$$

18.

$$\sum_{k=0}^{m}(-1)^kC_n^k=(-1)^mC_{n-1}^m$$

*/

4.5 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，$i = 1, 2, \cdots, n$

*/

4.6 母函数 Generating Function

```
/*母函数
 * 普通母函数:
 * 定义: 给定一个无穷序列$(a_0,a_1,a_2,\cdots,a_n,\cdots)$(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$(C_{n-1}^0,-C_n^1,C_{n+1}^2,\cdots,(-1)^kC_{n+k-1}^k,\cdots)$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$(0,1\times2\times3,2\times3\times4,\cdots,n\times(n+1)\times(n+2),\cdots)$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。

```

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

```
    故答案为$a_r=C_{n+r-1}^r$
*/
```

4.7 博弈论和 SG 函数

```
/*博弈论
组合游戏和SG函数
组合游戏定义:两人轮流决策;游戏状态集合有限;参与者操作时可将一状态转移到另一状态,
    对任一状态都有可以到达的状态集合;参与者不能操作时, 游戏结束, 按规则定胜负;
    游戏在有限步内结束(没有平局);参与者有游戏的所有信息.
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
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，$sg(x) = \min\{n|n \in N, n \notin F(x)\}$，
    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$g = sg_1 \wedge sg_2 \wedge \cdots \wedge sg_n$
 */
```

4.8 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。
  .....
 */
```

4.9 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：
*

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

```
 */
```

4.10 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;
}
```

4.11 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
*/
```

4.12 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$0 \le x < \frac{p}{2}$□$0 \le y < \frac{p}{2}$
```

5 字符串

5.1 palindrome 回文串

```
//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;
    }
}
```

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| cosθ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| sinθ
     *  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 - y_y1) = (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;
    stk[top++] = p[0];
    if(n > 1) stk[top++] = p[1];
    if(n > 2)
    {
        for(i = 2; 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);//

    }
}
```