目录

图论

连通性

强连通分量

Kosaraju

```
int n, m;
bool vis[maxn];
int ID[maxn];
vector<int>    G[maxn];//原图
vector<int>    P[maxn];//DAG
vector<int>    rG[maxn];//反图
vector<int>    ps;

void DFS(int u)
{
	vis[u] = true;
	for(int i = 0; i < G[u].size(); i++)
	{
		int v = G[u][i];
		if(!vis[v]) DFS(v);
	}
	ps.push_back(u);
}

void RDFS(int u, int x)
{
	vis[u] = true;
	ID[u] = x;
	for(int i = 0; i < rG[u].size(); i++)
	{
		int v = rG[u][i];
		if(!vis[v]) RDFS(v, x);
	}
}

void init()
{
```

```
	ps.clear();
	for(int i = 1; i <= n; i++)
	{
		G[i].clear();
		P[i].clear();
		rG[i].clear();
	}
}

int Kosaraju()
{
	int res = 0;

	memset(vis, 0, sizeof(vis));
	for(int i =    1; i <= n; i++)
	{
		if(!vis[i]) DFS(i);
	}
	memset(vis, 0, sizeof(vis));
	for(int i = ps.size() - 1; ~i; i--)
	{
		int v = ps[i];
		if(!vis[v]) RDFS(v, ++res);
	}
	return res;
}

void Get_DAG()
{
	for(int u = 1; u <= n; u++)
	{
		for(int i = 0; i < G[u].size(); i++)
		{
			int v = G[u][i];
			if(ID[u] != ID[v])
			{
				P[ID[u]].push_back(ID[v]);
			}
		}
	}
}
```

Tarjan

```
stack<int> S;
vector<int> G[maxn];
vector<int> P[maxn];
bool isinstack[maxn];
int low[maxn], dfn[maxn];
int ID[maxn];
int n, m, scc_num, index;

void Get_DAG()
{
	for(int u = 1; u <= n; u++)
	{
		for(int i = 0; i < G[u].size(); i++)
		{
			int v = G[u][i];
			if(ID[u] != ID[v])
			{
				P[ID[u]].push_back(ID[v]);
			}
		}
	}
}

void init()
{
	memset(dfn, -1, sizeof(dfn));
	memset(isinstack, 0, sizeof(isinstack));
	for(int i = 1; i <= n; i++)
	    P[i].clear(), G[i].clear();
}

void TarDFS(int u)
{
	low[u] = dfn[u] = index++;
	isinstack[u] = 1;
	S.push(u);
	for(int i = 0; i < G[u].size(); i++)
	{
		int v = G[u][i];
		if(dfn[v] == -1)
		{
```

```
            TarDFS(v);
            low[u] = min(low[u], low[v]);
        }
        else if(isinstack[v])
        {
            low[u] = min(low[u], dfn[v]);
        }
    }

    if(dfn[u] ==low[u])
    {
        scc_num++;
        while(S.top() != u)
        {
            isinstack[S.top()] = 0;
            ID[S.top()] = scc_num;
            S.pop();
        }
        isinstack[u] = 0;
        ID[u] = scc_num;
        S.pop();
    }
}

int Tarjan()
{
    scc_num = 0;
    index = 1;
    for(int i = 1; i <= n; i++)
    {
        if(dfn[i] == -1)
        {
            TarDFS(i);
        }
    }
    return scc_num;
}
```

割点/割边

割点

```
int n, m, inde;
vector<int> P[maxn];
int low[maxn], dfn[maxn];
set<int> ans;//割点

void init()
{
	memset(low, -1, sizeof(low));
	memset(dfn, -1, sizeof(dfn));
	ans.clear();
	for(int i = 0; i <= n; i++)
		P[i].clear();
	inde = 0;
}

void tarjan(int u, int fa)
{
	low[u] = dfn[u] = ++inde;
	int flag = 0;
	for(auto v: P[u])
	{
		if(v == fa) continue;
		if(dfn[v] == -1)
		{
			flag++;
			tarjan(v, u);
			low[u] = min(low[u], low[v]);
			if(low[v] >= dfn[u])
				ans.insert(u);
		}
		else
			low[u] = min(low[u], dfn[v]);
	}
	if(fa ==-1 && flag == 1)
		ans.erase(u);
}
```

割边

```
int n, m, inde;
vector<int> P[maxn];
int low[maxn], dfn[maxn];
set<pair<int, int> > ans;//桥

void init()
{
	memset(low, -1, sizeof(low));
	memset(dfn, -1, sizeof(dfn));
	ans.clear();
	for(int i = 0; i <= n; i++)
		P[i].clear();
	inde = 0;
}

void tarjan(int u, int fa)
{
	low[u] = dfn[u] = ++inde;
	for(auto v: P[u])
	{
		if(v == fa) continue;
		if(dfn[v] == -1)
		{
			tarjan(v, u);
			low[u] = min(low[u], low[v]);
			if(low[v] > dfn[u])
			{
				if(u < v)
					ans.insert({u, v});
				else
					ans.insert({v, u});
			}
		}
		else
			low[u] = min(low[u], dfn[v]);
	}
}
```

点/边双联通

点双联通

```
int n, m, inde, dcc_num;
vector<int> P[maxn];
int low[maxn], dfn[maxn], ID[maxn];
set<int> ans;
stack<int> S;

void init()
{
	memset(ID, 0, sizeof(ID));
	memset(low, -1, sizeof(low));
	memset(dfn, -1, sizeof(dfn));
	ans.clear();
	for(int i = 0; i <= 1005; i++)
		P[i].clear();
	dcc_num = inde = 0;
	while(!S.empty()) S.pop();
}

void tarjan(int u, int fa)
{
	low[u] = dfn[u] = ++inde;
	S.push(u);
	int flag = 0;
	for(int i = 0; i < P[u].size(); i++)
	{
		int v = P[u][i];
		if(v == fa) continue;
		if(dfn[v] == -1)
		{
			flag++;
			tarjan(v, u);
			low[u] = min(low[u], low[v]);
			if(low[v] >= dfn[u])
			{
				ans.insert(u);
				dcc_num++;
				do
				{
```

```
                ID[S.top()] = dcc_num;
                S.pop();
            }while(S.top() != u);
        }
    }
    else
        low[u] = min(low[u], dfn[v]);
}
if(fa == -1 && flag == 1)
    ans.erase(u);
}
```

边双联通

```
int n, m, inde, dcc_num;
vector<int> P[maxn];
int low[maxn], dfn[maxn], ID[maxn];
set<pair<int, int>> ans;
stack<int> S;

void init()
{
    memset(ID, 0, sizeof(ID));
    memset(low, -1, sizeof(low));
    memset(dfn, -1, sizeof(dfn));
    ans.clear();
    for(int i = 0; i <= n; i++)
        P[i].clear();
    dcc_num = inde = 0;
    while(!S.empty()) S.pop();
}

void tarjan(int u, int fa)
{
    low[u] = dfn[u] = ++inde;
    S.push(u);
    for(auto v:P[u])
    {
        if(v == fa) continue;
        if(dfn[v] == -1)
        {
            tarjan(v, u);
            low[u] = min(low[u], low[v]);
```

```
                if(low[v] > dfn[u])
                {
                    if(u < v)
                        ans.insert({u, v});
                    else
                        ans.insert({v, u});
                }
            }
            else
                low[u] = min(low[u], dfn[v]);
        }
        if(low[u] == dfn[u])
        {
            dcc_num++;
            while(S.top() != u)
            {
                ID[S.top()] = dcc_num;
                S.pop();
            }
            S.pop();
            ID[u] = dcc_num;
        }
}
```

最短路 && 查分约束

Dijkstra

```
const int inf = 1e9 + 7;
const int maxn = 50005;
int n, m;
ll dis[maxn];
int vis[maxn];

struct Edge
{
    int w;
    int to;
    int next;
} edge[maxn*2];
int edgeNum, head[maxn];

struct Node
```

```
{
    int u;
    int dis;
    bool operator < (const Node &a) const
    {
        return dis > a.dis;
    }
};

void init()
{
    edgeNum=0;
    for(int i = 0; i < maxn; i++)
    {
        vis[i] = 0;
        head[i] = -1;
        dis[i] = inf;
    }
}

void addEdge(int a, int b, int c)
{
    edge[edgeNum].w = c;
    edge[edgeNum].to = b;
    edge[edgeNum].next = head[a];
    head[a] = edgeNum++;
}

void dijkstra(int u = 1)
{
    int i, v;
    Node cur;
    priority_queue<Node> q;
    dis[u] = 0;
    q.push({u,0});
    while(!q.empty())
    {
        cur = q.top();
        q.pop();
        if(vis[cur.u]) continue;
        vis[cur.u] = 1;
        for(i = head[cur.u]; ~i; i = edge[i].next)
        {
            v = edge[i].to;
```

```
                if(dis[v] > dis[cur.u] + edge[i].w)
                {
                    dis[v] = dis[cur.u] + edge[i].w;
                    q.push({v, dis[v]});
                }
            }
        }
        cout << dis[n] << endl;
        return ;
}
```

SPFA

```
int n, m;
int d[maxn];
bool inq[maxn];
int num[maxn];

struct P
{
    int to, next, w;
}edge[maxn];

int head[maxn], cnt;

void add(int u, int v, int w)
{
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    edge[cnt].w = w;
    head[u] = cnt++;
}

void init()
{
    cnt = 0;
    memset(head, -1, sizeof(head));
    memset(num, 0, sizeof(num));
    memset(inq, 0, sizeof(inq));
    for(int i = 1; i <= n; i++)
        d[i] = mod;
}

bool SPFA(int s = 1)
```

```
{
    queue<int> Q;
    Q.push(s);
    d[s] = 0;
    num[s]++;
    inq[s] = 1;
    while(!Q.empty())
    {
        int u = Q.front(); Q.pop();
        for(int i = head[u]; ~i; i= edge[i].next)
        {
            int v = edge[i].to;
            if(d[v] > d[u] + edge[i].w)
            {
                d[v] = d[u] + edge[i].w;
                if(!inq[v])
                {
                    Q.push(v);
                    if(++num[v] > n) return false;
                }
            }
        }
    }
    return true;
}
```

Floyd_Wallshall

```
int n, m;
int d[maxn][maxn];

void pre(int s = 1)
{
    for(int i = 0; i < maxn; i++)
    {
        for(int j = 0; j < maxn; j++)
        {
            d[i][j] = inf;
        }
        d[i][i] = 0;
    }
}
```

```
void Floyd_Wallshall(int s = 1)
{
	for(int k = 1; k <= n; k++)
	{
		for(int i = 1; i <= n; i++)
		{
			for(int j = 1; j <= n; j++)
			{
				d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
			}
		}
	}
}
```

次短路

```
typedef pair<int, int> P;
const int inf = 1000000000;
int n, m;
bool vis[maxn];
int d[maxn][2];
struct edge
{
	int to;
	int cost;
};
vector<edge> M[maxn];

void dijkstra(int s = 1)
{
		for(int i = 1; i <= n; i++)
		{
			d[i][0] = d[i][1] = inf;
			vis[i] = 0;
		}
		d[s][0] = 0;
		priority_queue<P, vector<P>, greater<P>> PQ;
		PQ.push(make_pair(0, s));
		while(!PQ.empty())
		{
			P cur = PQ.top(); PQ.pop();
			int v = cur.second;
			int dis = cur.first;
			if(d[v][1] < dis) continue;
```

```
            for(int i = 0; i < M[v].size(); i++)
            {
                edge e = M[v][i];
                int u = e.to;
                int cost = e.cost + dis;
                if(d[u][0] > cost)
                {
                    swap(d[u][0], cost);
                     PQ.push(make_pair(d[u][0], u));
                }
                if(d[u][1] > cost && d[u][0] < cost)
                {
                     d[u][1] = cost;
                     PQ.push(make_pair(d[u][1], u));
                }
            }
        }
        cout << d[n][1] << endl;
}
```

查分约束

首先根据题目的要求进行不等式组的标准化。

(1)如果要求取最小值，那么求出最长路，那么将不等式全部化成 **xi – xj >= k** 的形式，这样建立 **j->i** 的边，权值为 **k** 的边，如果不等式组中有 **xi – xj > k**，因为一般题目都是对整形变量的约束，化为 **xi – xj >= k+1** 即可，如果 **xi – xj = k** 呢，那么可以变为如下两个：**xi – xj >= k, xi – xj <= k,**进一步变为 **xj – xi >= -k**，建立两条边即可。

(2)如果求取的是最大值，那么求取最短路，将不等式全部化成 **xi – xj <= k** 的形式，这样建立 **j->i** 的边，权值为 **k** 的边，如果像上面的两种情况，那么同样地标准化就行了。

(3)如果要判断差分约束系统是否存在解，一般都是判断环，选择求最短路或者最长路求解都行，只是不等式标准化时候不同，判环地话，用 **spfa** 即可，**n** 个点中如果同一个点入队超过 **n** 次，那么即存在环。

值得注意的一点是：建立的图可能不联通，我们只需要加入一个超级源点，比如说求取最长路时图不联通的话，我们只需要加入一个点 **S**，对其他的每个点建立一条权值为 **0** 的边图就联通了，然后从 **S** 点开始进行 **spfa** 判环。最短路类似。

建好图之后直接 spfa 或 bellman-ford 求解，不能用 dijstra 算法，因为一般存在负边，注意初始化的问题。

2-SAT

模型一：两者（A，B）不能同时取

那么选择了 A 就只能选择 B’，选择了 B 就只能选择 A’

连边 A→B’，B→A’

模型二：两者（A，B）不能同时不取

那么选择了 A’就只能选择 B，选择了 B’就只能选择 A

连边 A’→B，B’→A

模型三：两者（A，B）要么都取，要么都不取

那么选择了 A，就只能选择 B，选择了 B 就只能选择 A，选择了 A’就只能选择 B’，选择了 B’就只能选择 A’

连边 A→B，B→A，A’→B’，B’→A’

模型四：两者（A，A’）必取 A

连边 A’→A

```
int n, top;
bool mark[maxn];
int Sta[maxn];
struct P
{
    int to, next, w;
}edge[maxn];

int head[maxn], cnt;

void add(int u, int v, int w)
{
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    edge[cnt].w = w;
    head[u] = cnt++;
}

bool dfs(int v)
{
    if(mark[v ^ 1]) return false;
    if(mark[v]) return true;
```

```
    Sta[top++] = v;
    mark[v] = true;
    for(int i = head[v]; ~i; i = edge[i].next)
        if(!dfs(edge[i].to)) return false;
    return true;
}

bool solve()
{
    memset(mark, 0, sizeof(mark));
    for(int i = 0; i < 2 * n; i += 2)
    {
        if(!mark[i] && !mark[i + 1])
        {
            top = 0;
            if(!dfs(i)) while(top) mark[Sta[--top]] = false;
            if(!dfs(i + 1)) return false;
        }
    }
    return true;
}
```

生成树

最小生成树

Kruskal

```
struct node
{
    int u, v, w;
    bool operator < (const node &a) const
    {
        return w < a.w;
    }
} edge[maxm];
int fa[maxn];
int Find (int x)
{
    return x == fa[x] ? fa[x] : fa[x] = Find (fa[x]);
}
long long MST ()
```

```
{
    sort (edge, edge+m);
    for (int i = 1; i <= n; i++) fa[i] = i;
    long long ans = 0;
    for (int i = 0; i < m; i++)
    {
        int u = edge[i].u, v = edge[i].v;
        int p1 = Find (u), p2 = Find (v);
        if (p1 != p2)
        {
            fa[p1] = p2;
            ans += edge[i].w;
        }
    }
    return ans;
}
```

Prime

```
int n, m;
int cost[maxn][maxn];
int mincost[maxn];
bool used[maxn];

void pre()
{
    for(int i = 0; i < maxn; i++)
    {
        for(int j = 0; j < maxn; j++)
        {
            cost[i][j] = inf;
        }
        cost[i][i] = 0;
        mincost[i] = inf;
        used[i] = 0;
    }
}

int Prim()
{
        int res = 0;
        mincost[1] = 0;
```

```
        while(true)
        {
            int v = -1;

            for(int u = 1; u <= n; u++)
            {
                if(!used[u] && (v == -1 || mincost[u] < mincost[v]))
                    v = u;
            }
            if(v == - 1)
                break;
            used[v] = 1;
            res += mincost[v];

            for(int i = 1; i <= n; i++)
            {
                mincost[i] = min(mincost[i], cost[v][i]);
            }
        }
        return res;
}
```

最小树形图

朱刘算法

朱-刘算法的大概过程如下：

1、找到除了 root 以为其他点的权值最小的入边。用 In[i]记录

2、如果出现除了 root 以外存在其他孤立的点，则不存在最小树形图。

3、找到图中所有的环，并对环进行缩点，重新编号。

4、更新其他点到环上的点的距离

假设有重新编号之前的节点 u， v；编号之后为 U， V

则 dis[U][V] = dis[u][v] - In[V];

5、重复 3，4 知道没有环为止

```
struct P
{
	int u, v;
	ll c;
}edge[maxn * maxn];

int n, m;
ll in[maxn];
int pre[maxn];
int vis[maxn], id[maxn];

ll MST(int s, int n, int m)
{
	ll res = 0;
	while(true)
	{/*****找所有点最小的入边*****/
		for(int i = 1; i <= n; i++) in[i] = inf;
		for(int i = 0; i < m; i++)
		{
			int u = edge[i].u, v = edge[i].v;
			long long c = edge[i].c;
			if(c < in[v] && u != v)
			{
				pre[v] = u;
				in[v] = c;
			}
		}
		in[s] = 0;
		for(int i = 1; i <= n; i++)
		{
			if(in[i] == inf) return -1; //原图不连通
		}
	/**********判断有没有环******/
		memset(vis, -1, sizeof(vis));
		memset(id, -1, sizeof(id));
		int cnt = 1;
		for(int i = 1; i <= n; i++)
		{
			res += in[i];
			int v = i;
			while(vis[v] != i && v != s && id[v] == -1)
			{
				vis[v] = i;
				v = pre[v];
```

```
                }
                if(v != s && id[v] == -1)
                {
                    for(int u = pre[v]; u != v; u = pre[u]) id[u] = cnt;
                    id[v] = cnt++;
                }
            }
            if(cnt == 1)
            {
                break;
            }

            for(int i = 1; i <= n; i++)
            {
                if(id[i] == -1)
                {
                    id[i] = cnt++;
                }
            }
            /***********建立新图*****************/
            for(int i = 0; i < m; i++)
            {
                int u = edge[i].u;
                int v = edge[i].v;
                edge[i].u = id[u];
                edge[i].v = id[v];
                if(id[u] != id[v]) edge[i].c -= in[v];
            }
            n = cnt - 1;
            s = id[s];
        }
        return res;
}
```

拓扑排序

```
int n, m;
int Que[maxn];
int in[maxn];
vector<vector<int>> Map(maxn);
queue<int> Q;

bool topSort()
{
```

```
    While(!Q.empty()) Q.pop();
    int End = 1;
    for(int i = 1; i <= n; i++)
    {
        if(!in[i])
        {
                Q.push(i);
        }
    }

    while(!Q.empty())
    {
        Que[End] = Q.front();Q.pop();
        int node = Que[End++];
        for(auto x : Map[node])
        {
            in[x]--;
            if(!in[x])
            {
                Q.push(x);
            }
        }
    }
    if(End < n + 1) return 0;
    for(int i = 1; i <= n; i++)
    {
        printf("%d%c", Que[i], i == n ? '\n' : ' ');
    }
    return 1;
}
```

最大团

```
const int maxn = 100;
int n;

namespace MaxClique
{
    bool g[maxn][maxn];
    int ans, adj[maxn][maxn], ma[maxn];
    bool dfs(int now,int dep)
    {
        int i, j, u, v, poi;
        for(int i = 0; i < now; i++)
```

```
            {
                if (dep+now-i <= ans) return false;
                u = adj[dep][i], poi = 0;
                if (dep+ma[u] <= ans) return false;
                for (int j = i+1; j < now; j++) if (v=adj[dep][j], g[u][v])
                    adj[dep+1][poi++] = v;
                if (dfs(poi,dep+1)) return true;
            }
            if(dep > ans)
            {
                ans = dep;
                return true;
            }
            return false;
        }
        int work()
        {
            int i, j, poi;
            ans = 0;
            for (int i = n-1; i >= 0; i--)
            {
                poi = 0;
                for (int j = i+1; j < n; j++)
                    if (g[i][j]) adj[1][poi++] = j;
                dfs(poi,1);
                ma[i] = ans;
            }
            return ans;
        }
}
```

LCA

倍增

先预处理出每个结点向上跳 2^x 层的祖先和每个结点的深度
类似快速幂，拆分 deep[u] - deep[v](假设 deep[u] > deep[v])，每次使 u 向上跳 2^x 步，使 deep[u] = deep[v]
然后再一起往上跳，dis = deep[lca(u,v)] - deep[u]，由于 dis 我们不知道具体多少并且 lca 后面的结点都是 u，v 的公共祖先，所以不能像上面的一样直接拆分，于是换个思路，我们去找的最远的非公共祖先，最后肯定是到 lca 的子节点。

```
const int maxn = 5e5 + 50;
const int maxlog = 20;

int n, q, root;
int par[maxlog + 5][maxn];
int dep[maxn];
int head[maxn], edgecnt;
struct P
{
	int to, next;
}edge[2 * maxn];

void dfs(int u, int fa)
{
	par[0][u] = fa;
	dep[u] = dep[fa] + 1;
	for(int i = head[u]; ~i; i = edge[i].next)
	{
		if(edge[i].to == fa) continue;
		dfs(edge[i].to, u);
	}
}

void init()
{
	dfs(root, -1);
	for(int k = 0; k + 1 < maxlog; k++)
	{
		for(int u = 1; u <= n; u++)
		{
			if(par[k][u] == -1)
				par[k + 1][u] = -1;
			else
				par[k + 1][u] = par[k][par[k][u]];
		}
	}
}

int lca(int u, int v)
{
	if(dep[u] > dep[v])
	{
		swap(u, v);
	}
```

```
    for(int i = 20; ~i; i--)
    {
        if(dep[par[i][v]] >= dep[u]) v = par[i][v];
    }
    if(v == u) return v;

    for(int i = maxlog; ~i; i--)
    {
        if(par[i][u] != par[i][v])
        {
            u = par[i][u];
            v = par[i][v];
        }
    }
    return par[0][u];
}

void add(int u, int v)
{
    edge[edgecnt].to = v;
    edge[edgecnt].next = head[u];
    head[u] = edgecnt++;
}
```

基于 RMQ（ST 表）

vis[i]为第 i 次 dfs 所到达的结点
dep[i]为 vis[i]的深度
id[u]为 u 第一次被 dfs 时的 vis 编号
在 dfs 序中易知 lca(u, v) = vis[{i, id[u] <= i <= id[v]}中 dep 最小的 i]

```
const int maxn = 1e6 + 50;

int n, q, root, cnt;
int head[maxn], edgecnt;

int dp[maxn][maxlog];
int rec[maxn][maxlog];
int dep[maxn];
int vis[maxn];
int id[maxn];
struct P
{
    int to, next;
}edge[maxn];
```

```
void add(int u, int v)
{
	edge[edgecnt].to = v;
	edge[edgecnt].next = head[u];
	head[u] = edgecnt++;
}

void DFS(int u, int fa, int d, int &cnt)
{
	dep[cnt] = d;
	vis[cnt] = u;
	id[u] = cnt++;
	for(int i = head[u]; ~i; i = edge[i].next)
	{
		int v = edge[i].to;
		if(v != fa)
		{
			DFS(v, u, d + 1, cnt);
			dep[cnt] = d;
			vis[cnt++] = u;
		}
	}
}

void ST()
{
	for(int i = 0; i < cnt; i++)
		dp[i][0] = dep[i], rec[i][0] = vis[i];
	for(int j = 1; (1 << j) <= cnt; j++)
	{
		for(int i = 0; i + (1 << j) - 1 < cnt; i++)
		{
			if(dp[i][j - 1] > dp[i + (1 << (j - 1))][j - 1])
				dp[i][j] = dp[i + (1 << (j - 1))][j - 1], rec[i][j] = rec[i + (1 << (j - 1))][j - 1];
			else
				dp[i][j] = dp[i][j - 1], rec[i][j] = rec[i][j - 1];
		}
	}
}

void init()
{
	cnt = 0;
```

```
    DFS(root, -1, 0, cnt);
    ST();
}

inline int RMQ(int l, int r)
{
    int k = 0;
    while(1 << (k + 1) <= r - l + 1) k++;
    if(dp[l][k] > dp[r - (1 << k) + 1][k])
            return rec[r - (1 << k) + 1][k];
    else
            return rec[l][k];
}
```

Tarjan

离线算法
在 dfs 时， 对于查询 u，v 来说，若遍历到 u 时发现 v 已经遍历过了，这时 lca（u，v）就是用并查集维护的集合的 root
注意合并的时候将父节点作为并查集的上级即 u->v 合并时 par[v] = u

```
const int maxn = 1e6 + 50;

int n, q, root;
int head[maxn], edgecnt;
int qhead[maxn], quecnt;
int par[maxn];
int ans[maxn];
int vis[maxn];

struct P
{
    int to, next;
}edge[maxn];
struct P1
{
    int to, num, lca, next;
}que[maxn];

int Find(int u)
{
    if(u == par[u])
        return u;
    return par[u] = Find(par[u]);
```

```
}
void Merge(int x, int y)
{
	x = Find(x);
	y = Find(y);
	if(x != y)
	{
		par[x] = y;
	}
}

void add(int u, int v)
{
	edge[edgecnt].to = v;
	edge[edgecnt].next = head[u];
	head[u] = edgecnt++;
}

void addque(int u, int v, int i)
{
	que[quecnt].to = v;
	que[quecnt].num = i;
	que[quecnt].next = qhead[u];
	qhead[u] = quecnt++;
}

inline int read()
{
	int x=0,flag=0;
	char ch=getchar();
	if(ch=='-') flag=1;
	while(ch<'0'||ch>'9')ch=getchar();
	while(ch>='0'&&ch<='9')x*=10,x+=ch-'0',ch=getchar();
	if(flag) return -x;
	return x;
}

void Tarjan(int u, int fa)
{
	for(int i = head[u]; ~i; i = edge[i].next)
	{
		int v = edge[i].to;
		if(v != fa)
		{
```

```
                Tarjan(v, u);
                Merge(v, u);
                vis[v] = 1;
            }
        }
        vis[u] = 1;
        for(int i = qhead[u]; ~i; i = que[i].next)
        {
            int v = que[i].to;
            if(vis[v])
            {
                que[i].lca = Find(v);
                que[i^1].lca = que[i].lca;
                ans[que[i].num] = que[i].lca;
            }
        }
}

int main()
{
        n=read(),q=read(),root=read();
        memset(head, -1, sizeof(head));
        memset(qhead, -1, sizeof(qhead));
        for(int i = 0; i <= n; i++) par[i] = i;

        int u, v;
        for(int i = 0; i < n - 1; i++)
        {
            u = read(), v = read();
            add(u, v);
            add(v, u);
        }

        for(int i = 0; i < q; i++)
        {
            u = read(), v = read();
            addque(u, v, i);
            addque(v, u, i);
        }
        Tarjan(root, -1);
        for(int i = 0; i < q; i++)
        {
            printf("%d\n", ans[i]);
        }
```

```
    return 0;
}
```

二分图

相关总结

一、二分图最大匹配

定义：匹配是图中一些边的集合，且集合中任意两条边都没有公共点，所有的匹配中，边数最多的就是最大匹配。

算法：用匈牙利算法可以在 O(V*E)的复杂度内求出二分图的最大匹配，不过想真正完全证明这个算法，得去看组合数学。

二、二分图最小点覆盖

定义：点覆盖是图中一些点的集合，且对于图中所有的边，至少有一个端点属于点覆盖，点数最小的覆盖就是最小点覆盖。

定理：最小点覆盖=最大匹配。

简单证明：首先必然有，最小点覆盖>=最大匹配。于是只要证明不等式可以取等号，我们可在最大匹配的基础上构造出一组点覆盖。对右边每一个未匹配的点进行 dfs 找增广路，标记所有 dfs 过程中访问到的点，左边标记的点+右边未标记的点就是这个图的一个点覆盖。因为对于任意一条边，如果他的左边没标记，右边被标记了，那么我们就可找到一条新的增广路，所以每一条边都至少被一个点覆盖。再来证明：最大匹配=左边标记的点+右边未标记的点。对于每条匹配边，只有一个点属于点覆盖。如果这条边在 dfs 过程中被访问了，那么就左端点属于点覆盖，右端点不属于，否则就有左端点不属于点覆盖，右端点属于点覆盖。除此之外，不可能存在其它的点属于最小覆盖了，不然就必然可以找到增广路。所以：左边标记的点+右边未标记的点=最大匹配，对于任意的二分图，我们总能在最大匹配的基础上构造出一组点数等于最大匹配的点覆盖，所以：最小点覆盖=最大匹配。

三、二分图最小边覆盖

定义：边覆盖是图中一些边的集合，且对于图中所有的点，至少有一条集合中的边与其相关联，边数最小的覆盖就是最小边覆盖。

定理：最小边覆盖=图中点的个数-最大匹配。

简单证明：先贪心的选一组最大匹配的边加入集合，对于剩下的每个未匹配的点，随便选一条与之关联的边加入集合，得到的集合就是最小边覆盖，所以有：最小边覆盖=最大匹配+图中点的个数-2*最大匹配=图中点的个数-最大匹配。

四、二分图最大独立集

定义：独立集是图中一些点的集合，且图中任意两点之间都不存在边，点数最大的就是最大独立集。

定理：最大独立集=图中点的个数-最大匹配。

简单证明：可以这样来理解，先把所有的点都加入集合，删除最少的点和与其关联的边使得剩下的点相互之间不存在边，我们就得到了最大独立集。所以有：最大独立集=图中点的个数-最小点覆盖=图中点的个数-最大匹配。

五、有向无环图最小不相交路径覆盖

定义：用最少的不相交路径覆盖所有顶点。

定理：把原图中的每个点 V 拆成 Vx 和 Vy，如果有一条有向边 A->B，那么就加边 Ax-By。这样就得到了一个二分图，最小路径覆盖=原图的节点数-新图最大匹配。

简单证明：一开始每个点都独立的为一条路径，总共有 n 条不相交路径。我们每次在二分图里加一条边就相当于把两条路径合成了一条路径，因为路径之间不能有公共点，所以加的边之间也不能有公共点，这就是匹配的定义。所以有：最小路径覆盖=原图的节点数-新图最大匹配。

六、有向无环图最小可相交路径覆盖

定义：用最小的可相交路径覆盖所有顶点。

算法：先用 floyd 求出原图的传递闭包，即如果 a 到 b 有路，那么就加边 a->b。然后就转化成了最小不相交路径覆盖问题。

七、偏序集的最大反链

定义：偏序集中的最大独立集。

Dilworth 定理：对于任意偏序集都有，最大独立集（最大反链）=最小链的划分（最小不相交路径覆盖）。

通过 Dilworth 定理，我们就可以把偏序集的最大独立集问题转化为最小不相交路径覆盖问题了。

八、二分图带权最大匹配

定义：每个边都有一组权值，边权之和最大的匹配就是带权最大匹配。

算法：KM 算法，复杂度为 O（V^3）。

要注意的是，KM 算法求的是最佳匹配，即在匹配是完备的基础上权值之和最大。这和带权最大匹配是不一样的，不过我们可以加入若干条边权为 0 的边使得 KM 求出来的最佳匹配等于最大权匹配。具体实现的时候最好用矩阵来存图，因为一般点的个数都是 10^2 级别，并且这样默认任意两点之间都存在边权为 0 的边，写起来很方便。如果要求最小权匹配，我们可以用一个很大数减去每条边的边权。

二分图最大匹配

匈牙利算法

复杂度（n*m）

```
int n;
int W[maxn][maxn];
bool vis[maxn];
int par[maxn];

bool check(int u)
{
```

```
    for(int v = 1;    v <= n; v++)
    {
        if(W[u][v])
        if(!vis[v])
        {
            vis[v] = 1;
            if(par[v] == -1 || check(par[v]))
            {
                par[v] = u;
                return true;
            }
        }
    }
    return false;
}

int hungry()
{
    int res = 0;
    memset(par, -1, sizeof(par));
    for(int i = 1; i <= n; i++)
    {
        memset(vis, 0, sizeof(vis));
        if(check(i))
            res++;
    }
    return res;
}
```

Hopcroft-Karp 算法 O(√n*m)

```
struct P
{
    int v, next, c;
}edge[maxn * 2];
int head[maxn], cnt;

void add(int u, int v)
{
    edge[cnt].v = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}
```

```
int n, m;
#define maxL 5005
#define maxR 5005
int Mx[maxL], My[maxR], dx[maxL], dy[maxR];
bool used[maxR];
int dis;
bool searchP ()
{
	queue <int> q;
	dis = 1e9;
	Clear (dx, -1);
	Clear (dy, -1);
	for (int i = 0; i < n; i++)
	{
		if (Mx[i] == -1)
		{
			q.push (i);
			dx[i] = 0;
		}
	}
	while (!q.empty ())
	{
		int u = q.front ();
		q.pop ();
		if (dx[u] > dis) break;
		for (int i = head[u]; i+1; i = edge[i].next)
		{
			int v = edge[i].v;
			if (dy[v] == -1)
			{
				dy[v] = dx[u]+1;
				if (My[v] == -1) dis = dy[v];
				else
				{
					dx[My[v]] = dy[v]+1;
					q.push (My[v]);
				}
			}
		}
	}
	return dis != 1e9;
}
bool dfs (int u)
```

```
{
	for (int i = head[u]; i+1; i = edge[i].next)
	{
		int v = edge[i].v;
		if (!used[v] && dy[v] == dx[u]+1)
		{
			used[v] = 1;
			if (My[v] != -1 && dy[v] == dis) continue;
			if (My[v] == -1 || dfs (My[v]))
			{
				My[v] = u;
				Mx[u] = v;
				return 1;
			}
		}
	}
	return 0;
}
int maxflow ()
{
	int res = 0;
	Clear (Mx, -1);
	Clear (My, -1);
	while (searchP ())
	{
		Clear (used, 0);
		for (int i = 0; i < n; i++) if (Mx[i] == -1 && dfs (i))
				res++;
	}
	return res;
}
```

二分图最大权匹配

复杂度（n^3）

```
const int maxn = 305;
const int inf = 1 << 30;
const int64_t mod = 1e9 + 7;
int w[maxn][maxn];
int lx[maxn], ly[maxn];
int linky[maxn], linkx[maxn];
bool visx[maxn], visy[maxn];
int slack[maxn];
```

```
int n, m;

bool Find(int x)//匈牙利算法求是否有合法匹配点
{
	visx[x] = true;
	for(int y = 0; y < m; y++)
	{
		if(visy[y]) continue;
		int t = lx[x] + ly[y] - w[x][y];

		if(!t)
		{
			visy[y] = true;
			if(linky[y] == -1 || Find(linky[y]))
			{
				linky[y] = x;
				linkx[x] = y;
				return true;
			}
		}
		else
			slack[y] = min(slack[y], t);
	}
	return false;
}
int KM()
{
	memset(linky, -1, sizeof(linky));
	memset(ly, 0, sizeof(ly));//初始时所有都在权值都在 lx
	for(int v = 0; v < n; v++)//逐步增大二分子图
	{
		for(int j = 0; j < m; j++) slack[j] = inf;

		while(true)
		{
			memset(visx, 0, sizeof(visx));
			memset(visy, 0, sizeof(visy));
			if(Find(v))
				break;

			int d = inf;
			for(int i = 0; i < m; i++)
			{
				if(!visy[i])
```

```
                {
                    d = min(d, slack[i]); //算法贪心性质这里体现
                }
            }
            if(d == inf) return -1;
            for(int i = 0; i < n; i++)
            {
                if(visx[i])
                {
                    lx[i] -= d;
                }
            }
            for(int i = 0; i < m; i++)
            {
                if(visy[i])
                    ly[i] += d;
                else
                slack[i] -= d;//在这个 while 循环中是叠加的，所以后面的 slack 要减去当前 d
            }
        }
    }
    int res = 0;
    for(int i = 0; i < m; i++)
    {
        //cout << linkx[i] << " " << linky[i] << endl;
        if(linky[i] != -1) res += w[linky[i]][i];
    }
    return res;
}
int main()
{
    ios_base::sync_with_stdio(0);
    cin.tie(0);

    while(cin >> n)
    {
        m = n;
        for(int i = 0; i < n; i++)
        {
            lx[i] = -inf;
            for(int j = 0; j < m; j++)
            {
                cin >> w[i][j];
                lx[i] = max(lx[i], w[i][j]);
```

```
                }

            }
            cout << KM() << endl;
        }
        return 0;
}
```

网络流

最大流 && 最小割

Ford-Fulkerson

复杂度（F*M）

```
const int maxn = 10005;
bool vis[maxn];
int n, m, s, t, inf = 1e9 + 7;
struct Edge
{
        int to, cap, rev;
};
vector<Edge> G[maxn];

void add_edge(int u, int v, int c)
{
        G[u].push_back({v, c, G[v].size()});
        G[v].push_back({u, 0, G[u].size() - 1});
}

int dfs(int v, int t, int f)
{
        if(v == t) return f;
        vis[v] = true;
        for(int i = 0; i < (int)G[v].size(); i++)
        {
                Edge &e = G[v][i];
                if(!vis[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;
}

int max_flow(int s, int t)
{
        int flow = 0;
        while(true)
        {
            memset(vis, 0, sizeof(vis));
            int f = dfs(s, t, inf);
            if(!f) return flow;
            flow += f;
        }
}
```

Dinic

复杂度（n*n*m）

```
const int maxn = 10005;
int level[maxn];
int iter[maxn];
int n, m, s, t, inf = 1e9 + 7;

struct Edge
{
        int to, cap, rev;
};

vector<Edge> G[maxn];

void add_edge(int u, int v, int c)
{
        G[u].push_back({v, c, G[v].size()});
        G[v].push_back({u, 0, G[u].size() - 1});
}
```

```
void bfs(int s = s)
{
	memset(level, -1, sizeof(level));
	queue<int> Q;
	Q.push(s);
	level[s] = 0;
	while(!Q.empty())
	{
		int cur = Q.front(); Q.pop();
		for(int i = 0; i < (int)G[cur].size(); i++)
		{
			Edge &e = G[cur][i];
			if(e.cap > 0 && level[e.to] < 0)
			{
				level[e.to] = level[cur] + 1;
				Q.push(e.to);
			}
		}
	}
}

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;
```

```
	while(true)
	{
		bfs(s);
		if(level[t] < 0) return flow;
		memset(iter, 0, sizeof(iter));
		int f;
		while((f = dfs(s, t, inf)) > 0)
		{
			flow += f;
		}
	}
}
```

ISAP

```
const int maxn = 10005;
int n, m, s, t, inf = 1e9 + 7;

struct Edge
{
	int to, cap, rev;
};

vector<Edge> G[maxn];

void add_edge(int u, int v, int c)
{
	G[u].push_back({v, c, G[v].size()});
	G[v].push_back({u, 0, G[u].size() - 1});
}

namespace ISAP
{
int gap[maxn];
int iter[maxn];
int level[maxn];

void bfs(int t)
{
	memset(gap,0,sizeof(gap));
	memset(level,-1,sizeof(level));
	queue <int> q;
```

```
    level[t]=1;
    gap[level[t]]=1;
    q.push(t);

    while (!q.empty())
    {
        int x=q.front();
        q.pop();
        for (int i=0; i<(int)G[x].size(); i++)
        {
            Edge &e=G[x][i];
            if (level[e.to]<0)
            {
                level[e.to]=level[x]+1;
                gap[level[e.to]]++;
                q.push(e.to);
            }
        }
    }
}

int dfs(int x, int s, int t, int f)
{
    if (x==t) return f;
    int flow=0;
    for (int &i=iter[x]; i<(int)G[x].size(); i++)
    {
        Edge &e=G[x][i];
        if (e.cap>0&&level[x]==level[e.to]+1)
        {
            int d=dfs(e.to,s,t,min(f-flow,e.cap));
            e.cap-=d;
            G[e.to][e.rev].cap+=d;
            flow+=d;
            if (f==flow) return f;
        }
    }

    gap[level[x]]--;
    if (gap[level[x]]==0)
        level[s]=n+1;
    iter[x]=0;
    gap[++level[x]]++;
    return flow;
```

```
}

int max_flow(int s, int t)
{
	int flow=0;
	bfs(t);
	memset(iter,0,sizeof(iter));
	while (level[s]<=n)
		flow+=dfs(s,s,t,inf);
	return flow;
}
}
```

HLPP

复杂度（n*n*m^1/2）

```
#include<cstdio>
#include<cstring>
#include<algorithm>
#include<queue>
using std::min;
using std::vector;
using std::queue;
using std::priority_queue;
const int N=2e4+5,M=2e5+5,inf=0x3f3f3f3f;
int n,s,t,tot;
int v[M<<1],w[M<<1],first[N],next[M<<1];
int h[N],e[N],gap[N<<1],inq[N];//节点高度是可以到达 2n-1 的
struct cmp
{
	inline bool operator()(int a,int b) const
	{
		return h[a]<h[b];//因为在优先队列中的节点高度不会改变，所以可以直接比较
	}
};
queue<int> Q;
priority_queue<int,vector<int>,cmp> pQ;
inline void add_edge(int from,int to,int flow)
{
	tot+=2;
	v[tot+1]=from;v[tot]=to;w[tot]=flow;w[tot+1]=0;
	next[tot]=first[from];first[from]=tot;
	next[tot+1]=first[to];first[to]=tot+1;
```

```
    return;
}
inline bool bfs()
{
    int now;
    register int go;
    memset(h+1,0x3f,sizeof(int)*n);
    h[t]=0;Q.push(t);
    while(!Q.empty())
    {
        now=Q.front();Q.pop();
        for(go=first[now];go;go=next[go])
            if(w[go^1]&&h[v[go]]>h[now]+1)
                h[v[go]]=h[now]+1,Q.push(v[go]);
    }
    return h[s]!=inf;
}
inline void push(int now)//推送
{
    int d;
    register int go;
    for(go=first[now];go;go=next[go])
        if(w[go]&&h[v[go]]+1==h[now])
        {
            d=min(e[now],w[go]);
            w[go]-=d;w[go^1]+=d;e[now]-=d;e[v[go]]+=d;
            if(v[go]!=s&&v[go]!=t&&!inq[v[go]])
                pQ.push(v[go]),inq[v[go]]=1;
            if(!e[now])//已经推送完毕可以直接退出
                break;
        }
    return;
}
inline void relabel(int now)//重贴标签
{
    register int go;
    h[now]=inf;
    for(go=first[now];go;go=next[go])
        if(w[go]&&h[v[go]]+1<h[now])
            h[now]=h[v[go]]+1;
    return;
}
inline int hlpp()
{
```

```
    int now,d;
    register int i,go;
    if(!bfs())//s 和 t 不连通
        return 0;
    h[s]=n;
    memset(gap,0,sizeof(int)*(n<<1));
    for(i=1;i<=n;i++)
        if(h[i]<inf)
            ++gap[h[i]];
    for(go=first[s];go;go=next[go])
        if(d=w[go])
        {
            w[go]-=d;w[go^1]+=d;e[s]-=d;e[v[go]]+=d;
            if(v[go]!=s&&v[go]!=t&&!inq[v[go]])
                pQ.push(v[go]),inq[v[go]]=1;
        }
    while(!pQ.empty())
    {
        inq[now=pQ.top()]=0;pQ.pop();push(now);
        if(e[now])
        {
            if(!--gap[h[now]])//gap 优化，因为当前节点是最高的所以修改的节点一定不在
优先队列中，不必担心修改对优先队列会造成影响
                for(i=1;i<=n;i++)
                    if(i!=s&&i!=t&&h[i]>h[now]&&h[i]<n+1)
                        h[i]=n+1;
            relabel(now);++gap[h[now]];
            pQ.push(now);inq[now]=1;
        }
    }
    return e[t];
}
int m;
signed main()
{
    int u,v,w;
    scanf("%d%d%d%d",&n,&m,&s,&t);
    while(m--)
    {
        scanf("%d%d%d",&u,&v,&w);
        add_edge(u,v,w);
    }
    printf("%d\n",hlpp());
    return 0;
```

```
}
```

费用流

Spfa

```
const int maxn = 10005;
const int inf = 1e9 + 7;
int n, m;
int dis[maxn];
int prevv[maxn];
int preve[maxn];
bool vis[maxn];
struct Edge
{
	int to, cap, cost, rev;
};
vector<Edge> G[maxn];

void add_edge(int u, int v, int cap, int cost)
{
	G[u].push_back({v, cap, cost, G[v].size()});
	G[v].push_back({u, 0, -cost, G[u].size() - 1});
}

int spfa(int s, int t)
{
	fill(dis, dis + n + 1, inf);
	memset(vis, 0, sizeof(vis));
	queue<int> Q; Q.push(s);
	vis[s] = 1; dis[s] = 0;
	while(!Q.empty())
	{
		int cur = Q.front(); Q.pop();
	   // cout << cur << endl;
		vis[cur] = 0;
		for(int i = 0; i < (int)G[cur].size(); i++)
		{
			Edge &e = G[cur][i];
			//cout << e.to << endl;
			if(e.cap > 0 && dis[e.to] > dis[cur] + e.cost)
			{
				dis[e.to] = dis[cur] + e.cost;
```

```
                        prevv[e.to] = cur;
                        preve[e.to] = i;
                        if(!vis[e.to])
                        {
                            vis[e.to] = 1;
                            Q.push(e.to);
                        }
                    }
                }
            }
            return dis[t];
}

int mcmf(int s, int t)
{
            int flow = 0, res = 0;
            while(spfa(s, t) != inf)
            {
                int Min = inf;
                for(int v = t; v != s; v = prevv[v])
                {
                    Min = min(Min, G[prevv[v]][preve[v]].cap);
                }
                res += Min * dis[t];
                flow += Min;
                for(int v = t; v != s; v = prevv[v])
                {
                    G[prevv[v]][preve[v]].cap -= Min;
                    G[v][G[prevv[v]][preve[v]].rev].cap += Min;
                }
            }
            return res;
}
```

Dijkstra

```
typedef pair<int, int> P; //距离和顶点
const int maxn = 10005;
const int inf = 1e9 + 7;
int n, m;
int dis[maxn];
int h[maxn];
int prevv[maxn], preve[maxn];
```

```
struct Edge
{
	int to, cap, cost, rev;
};
vector<Edge> G[maxn];

void add_edge(int u, int v, int cap, int cost)
{
	G[u].push_back({v, cap, cost, G[v].size()});
	G[v].push_back({u, 0, -cost, G[u].size() - 1});
}

inline int read()
{
	int x=0;
	char c=getchar();
	bool flag=0;
	while(c<'0'||c>'9'){if(c=='-')flag=1;	c=getchar();}
	while(c>='0'&&c<='9'){x=(x<<3)+(x<<1)+c-'0';c=getchar();}
	return flag?-x:x;
}

int mcmf(int s, int t)
{
	int res = 0, flow = 0;
	memset(h, 0, sizeof(h));
	while(true)
	{
		fill(dis, dis + n + 1, inf);
		priority_queue<P, vector<P>, greater<P>> PQ;
		dis[s] = 0;
		PQ.push({0, s});
		while(!PQ.empty())
		{
			P cur = PQ.top(); PQ.pop();
			int v = cur.second;
			if(dis[v] < cur.first) continue;
			for(int i = 0; i < G[v].size(); i++)
			{
				Edge &e = G[v][i];
				if(e.cap > 0 && dis[e.to] > dis[v] + e.cost + h[v] - h[e.to])
				{
					dis[e.to] = dis[v] + e.cost + h[v] - h[e.to];
```

```
                        prevv[e.to] = v;
                        preve[e.to] = i;
                        PQ.push({dis[e.to], e.to});
                    }
                }
            }
            if(dis[t] == inf)
            {
                cout << flow << " " << res << endl;
                return res;
            }
            for(int v = 1; v <= n; v++) h[v] += dis[v];
            int d = inf;
            for(int v = t; v != s; v = prevv[v])
            {
                d = min(d, G[prevv[v]][preve[v]].cap);
            }
            flow += d;
            res += d * h[t];
            for(int v = t; v != s; v = prevv[v])
            {
                G[prevv[v]][preve[v]].cap -= d;
                G[v][G[prevv[v]][preve[v]].rev].cap += d;
            }
        }
}
```

一般数据结构

ST Table

一维

```
//[0,n-1]
void rmq_init ()
{
    for (int i = 0; i < n; i++) dp[i][0] = a[i];
    for (int j = 1; (1<<j) <= n; j++)
    {
        for (int i = 0; i+(1<<j)-1 < n; i++)
        {
            dp[i][j] = __gcd (dp[i][j-1], dp[i+(1<<(j-1))][j-1]);
        }
```

```
    }
}
int rmq (int l, int r)
{
    int k = 0;
    while ((1<<(k+1)) <= r-l+1) k++;
    return __gcd (dp[l][k], dp[r-(1<<k)+1][k]);
}
```

二维

```
//(1,1)~(n,m)
int n, m;
int val[maxn][maxn];
int dp[maxn][maxn][9][9];
void rmq_init ()
{
    for(int row = 1; row <= n; row++)
        for(int col = 1; col <=m; col++)
            dp[row][col][0][0] = val[row][col];
    int mx = log(double(n)) / log(2.0);
    int my = log(double(m)) / log(2.0);
    for(int i=0; i<= mx; i++)
    {
        for(int j = 0; j<=my; j++)
        {
            if(i == 0 && j ==0) continue;
            for(int row = 1; row+(1<<i)-1 <= n; row++)
            {
                for(int col = 1; col+(1<<j)-1 <= m; col++)
                {
                    if(i == 0)//y 轴二分
                        dp[row][col][i][j]=max(dp[row][col][i][j-1],dp[
                                                        row][col+(1<<(j-1))][i][j-1]);
                    else//x 轴二分
                        dp[row][col][i][j]=max(dp[row][col][i-1][j],dp[
                                                        row+(1<<(i-1))][col][i-1][j]);
                }
            }
        }
    }
}
int rmq (int x1,int x2,int y1,int y2)
{
    int kx = log(double(x2-x1+1)) / log(2.0);
    int ky = log(double(y2-y1+1)) / log(2.0);
```

```
    int m1 = dp[x1][y1][kx][ky];
    int m2 = dp[x2-(1<<kx)+1][y1][kx][ky];
    int m3 = dp[x1][y2-(1<<ky)+1][kx][ky];
    int m4 = dp[x2-(1<<kx)+1][y2-(1<<ky)+1][kx][ky];
    return max( max(m1,m2), max(m3,m4));
}
```

树状数组

单点更新

```
void add (int x, int num)
 {
    for (int i = x; i; i -= lowbit (i)) c[i] += num;
}
int sum (int x)
 {
    int ans = 0;
    for (int i = x; i < maxn; i += lowbit (i)) ans += c[i];
    return ans;
}
```

区间更新

```
typedef long long ll;
#define lowbit(i) ((i) & -(i))
const int maxn = 1e5 + 5;

int n;
ll bit0[maxn], bit1[maxn];

ll sum(ll *b, ll i)
{
    ll res = 0;
    while(i > 0)
    {
        res += b[i];
        i -= lowbit(i);
    }
    return res;
}

void add(ll *b, int i, int v)
{
    while(i <= n)
    {
        b[i] += v;
```

```
            i += lowbit(i);
        }
}

int main()
{
    ios_base::sync_with_stdio(0);

    int q;
    while(cin >> n >> q)
    {
        int tmp;
        rep(i, 0, n)
            cin >> tmp, add(bit0, i + 1, tmp);
        string s;
        int l, r, x;
        while(q--)
        {
            cin >> s;
            if(s[0] == 'Q')
            {
                cin >> l >> r;
                ll res = 0;
                res += sum(bit0, r) + 1LL * sum(bit1, r) * r;
                res -= sum(bit0, l - 1) + 1LL * sum(bit1, l - 1) * (l - 1);
                cout << res << endl;
            }
            else
            {
                cin >> l >> r >> x;
                add(bit0, l, -x * 1LL * (l - 1));
                add(bit1, l, x);
                add(bit0, r + 1, 1LL * x * r);
                add(bit1, r + 1, -x);
            }
        }
    }
    return 0;
}
```

树链剖分

```
const int maxn = 1e5 + 50;
struct P
```

```
{
    int to, next;
}edge[maxn * 2];
int head[maxn], cnt;
void add_edge(int u, int v)
{
    edge[cnt].to = v;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}

int n, q, r;
int w[maxn], wt[maxn];
int dep[maxn], fa[maxn], siz[maxn], son[maxn];
int id[maxn], top[maxn];

void dfs1(int f, int u, int d)
{
    fa[u] = f, dep[u] = d, siz[u] = 1;
    int Max = -1;
    for(int i = head[u]; ~i; i = edge[i].next)
    {
        int v = edge[i].to;
        if(v == f) continue;
        dfs1(u, v, d + 1);
        siz[u] += siz[v];
        if(siz[v] > Max)
        {
            Max = siz[v];
            son[u] = v;
        }
    }
}

void dfs2(int u, int topf)
{
    id[u] = cnt; //dfs2 前 cnt 要重新赋为 1
    wt[cnt++] = w[u];
    top[u] = topf;
    if(!son[u]) return;
    dfs2(son[u], topf);
    for(int i = head[u]; ~i; i = edge[i].next)
    {
        if(!id[edge[i].to])
```

```
                dfs2(edge[i].to, edge[i].to);
        }
}
/*********线段树**********/
struct SegTree
{
        int l, r, siz;
        ll w, laz;
}T[maxn * 4];

void push_up(int rt)
{
        T[rt].w = T[rt + rt].w + T[rt + rt + 1].w;
}

void push_dow(int rt)
{
        if(T[rt].laz)
        {
                T[rt + rt].w = T[rt + rt].w + 1LL * T[rt + rt].siz * T[rt].laz;
                T[rt + rt + 1].w = T[rt + rt + 1].w + 1LL * T[rt + rt + 1].siz * T[rt].laz;
                T[rt + rt].laz = T[rt + rt].laz + T[rt].laz;
                T[rt + rt + 1].laz = T[rt + rt + 1].laz + T[rt].laz;
                T[rt].laz = 0;
        }
}

void build(int rt, int l, int r)
{
        T[rt].l = l, T[rt].r = r, T[rt].siz = r - l + 1;
        T[rt].laz = 0;
        if(l == r)
        {
                T[rt].w = wt[l];
        }
        else
        {
                int mid = (l + r) >> 1;
                build(rt + rt, l, mid);
                build(rt + rt + 1, mid + 1, r);
                push_up(rt);
        }
}
```

```
void Add(int rt, int l, int r, int x)
{
	int L = T[rt].l, R = T[rt].r;
	if(l <= L && R <= r)
	{
		T[rt].w += 1LL * T[rt].siz * x;
		T[rt].laz += x;
		return ;
	}
	push_dow(rt);
	int mid = (L + R) >> 1;
	if(l <= mid) Add(rt + rt, l, r, x);
	if(r > mid) Add(rt + rt + 1, l, r, x);
	push_up(rt);
}

ll Sum(int rt, int l, int r)
{
	int L = T[rt].l, R = T[rt].r;
	if(l <= L && R <= r)
		return T[rt].w;
	push_dow(rt);
	int mid = (L + R) >> 1;
	ll res = 0;
	if(l <= mid) res += Sum(rt + rt, l, r);
	if(r > mid) res += Sum(rt + rt + 1, l, r);
	return res;
}
/******************/
void TreeSum(int x, int y)
{
	ll res = 0;
	while(top[x] != top[y])
	{
		if(dep[top[x]] < dep[top[y]]) swap(x, y);
		res += Sum(1, id[top[x]], id[x]);
		x = fa[top[x]];
	}
	if(dep[x] > dep[y]) swap(x, y);
	res += Sum(1, id[x], id[y]);
}

void TreeAdd(int x, int y, int z)
{
```

```
    while(top[x] != top[y])
    {
        if(dep[top[x]] < dep[top[y]]) swap(x, y);
        Add(1, id[top[x]], id[x], z);
        x = fa[top[x]];
    }
    if(dep[x] < dep[y]) swap(x, y);
    Add(1, id[y], id[x], z);
}

void TreeAddSub(int x, int z)
{
    Add(1, id[x], id[x] + siz[x] - 1, z);
}

ll TreeSumSub(int x)
{
    return Sum(1, id[x], id[x] + siz[x] - 1);
}

void init1()
{
    memset(head, -1, sizeof(head)); cnt = 0;
    memset(son, 0, sizeof(son));
    memset(id, 0, sizeof(id));
}

void init2()
{
    dfs1(0, r, 0);
    cnt = 1;
    dfs2(r, r);
    build(1, 1, n);
}
```

平衡二叉树

Splay

```
const int inf = 2e9 + 50;

class node
{
```

```
public:
        node* ch[2], *fa;
        int val;
        int size;
        int recy;
        node(int x)
        {
            ch[0] = ch[1] = fa = NULL;
            val = x;size = 1;recy = 1;
        }
};

inline void push_up(node *v)
{
    v->size = v->recy + (v->ch[0] ? v->ch[0]->size : 0) + (v->ch[1] ? v->ch[1]->size : 0);
}

inline void attach(node *p, node *s, int x)
{
      p->ch[x] = s;
      if(s) s->fa = p;
}

class Splay//存储规则：小左大右，重复节点记录
{
    node* root;
    void rotate(node *v)
    {
        if(v == root) return ;
        node* p = v->fa;
        int flag = p->ch[1] == v;
        if(p->fa)    attach(p->fa, v, p->fa->ch[1] == p);
        else    v->fa = NULL, root = v;
        attach(p, v->ch[flag ^ 1], flag);
        attach(v, p, flag ^ 1);
        push_up(p);
        push_up(v);
    }
    public:
    void init()
    {
        root = NULL;
    }
    node* GetRoot()
```

```
{
	return root;
}
node* splay(node *v)
{
	for(node *p; p = v->fa; rotate(v))
	{
		if(p->fa) rotate((p->fa->ch[0] == p) == (p->ch[0] == v) ? p : v);
	}
	return root = v;
}
node* search(int x)//查找值为 x 的节点  没找到返回值最接近的节点
{
	node* p = root;
	while(p)
	{
		if(p->val == x) break;
		if(p->ch[p->val < x]) p = p->ch[p->val < x];
		else break;
	}
	splay(p);
	return p;
}
node* insert(int x)//插入一个值为 x 的节点
{
	if(!root) return root = new node(x);
	node* p = search(x);
	if(p->val == x)
	{
		p->recy++, p->size++;
		return p;
	}
	node* v = new node(x);
	int flag = p->val > x;
	attach(v, p, flag);
	attach(v, p->ch[flag ^ 1], flag ^ 1);
	p->ch[flag ^ 1] = NULL;
	v->fa = NULL;
	push_up(p);push_up(v);
	return root = v;
}
bool erase(int x) //删除值为 x 的节点
{
	node* p = search(x);
```

```
    if(!p || p->val != x)
        return false;
    if(p->recy > 1)
        p->recy--, p->size--;
    else if(!p->ch[0] && !p->ch[1])
        root = NULL;
    else if(!p->ch[0])
    {
        root = p->ch[1];
        root->fa = NULL;
        delete p;
    }
    else if(!p->ch[1])
    {
        root = p->ch[0];
        root->fa = NULL;
        delete p;
    }
    else
    {
        node* tmp = root->ch[0];
        tmp->fa = NULL;
        root->ch[0] = NULL;
        root = root->ch[1];
        root->fa = NULL;
        search(p->val);
        root->ch[0] = tmp;
        tmp->fa = root;
        delete p;
    }
    if(root) push_up(root);
    return true;
}
int rank(int x)//返回 x 的排名  从 1 开始，重复按第一个算
{
    search(x);
    int res = root->ch[0] ? root->ch[0]->size + 1: 1;
    return res + (root->val < x ? root->recy : 0);
}
int arank(int x)//查询排名 x 的值
{
    if(x <= 0) return -inf;
    node* p = root;
    while(p)
```

```
    {
        if(p->ch[0] && p->ch[0]->size >= x)
            p = p->ch[0];
        else if((p->ch[0] ? p->ch[0]->size : 0) + p->recy >= x)
        {
            splay(p);
            return p->val;
        }
        else
        {
            x -= (p->ch[0] ? p->ch[0]->size : 0) + p->recy;
            p = p->ch[1];
        }
    }
    return inf;
}
 int pre(int x)//求 x 的前驱(前驱定义为小于 x，且最大的数)
{
    search(x);
    node* p = root;
    if(x > p->val) return p->val;
    if(p->ch[0])
    {
        p = p->ch[0];
        int res = p->val;
        while(p->ch[1])
        {
            p = p->ch[1];
            res = p->val;
        }
        return res;
    }
    return -inf;
}
int suc(int x)//求 x 的后继(后继定义为大于 x，且最小的数)
{
    search(x);
    node* p = root;
    if(p->val > x) return p->val;
    if(p->ch[1])
    {
        p = p->ch[1];
        int res = p->val;
        while(p->ch[0])
```

```
                {
                    p = p->ch[0];
                    res = p->val;
                }
                return res;
            }
            return inf;
        }
};
```

区间翻转

```
#include <bits/stdc++.h>

const    int    N = 100000 + 5 ;

int    a [ N ] ;
int    n , m , x , y , ok , inf = 1e9 + 7 ;

struct    Node    {
    Node    * son [ 2 ] ;
    Node    * fa ;
    int    val ;
    int    siz ;
    int    tag ;
    void    update ( ) ;
    void    push_down ( ) ;
}
pool [ N * 32 ] , * tail = pool , * root , * zero = ++ tail ;

void    Node :: update ( ) {
    siz = 1 ;
    if ( son [ 0 ] != zero )    siz += son [ 0 ] -> siz ;
    if ( son [ 1 ] != zero )    siz += son [ 1 ] -> siz ;
}
void    Node :: push_down ( ) {
    if ( tag ) {
        std :: swap ( son [ 0 ] , son [ 1 ] ) ;
        if ( son [ 0 ] != zero )    son [ 0 ] -> tag ^= 1 ;
        if ( son [ 1 ] != zero )    son [ 1 ] -> tag ^= 1 ;
        tag = 0 ;
    }
}

struct    Splay {
```

```
void    init ( ) {
        root = build ( zero , 1 , n + 2 , a ) ;
}
Node    * build ( Node    * p , int    l , int    r , int    * a ) {
        if ( l > r )    return    zero ;
        Node    * nd = ++ tail ;
        int    mid = l + r >> 1 ;
        nd -> fa = p ;
        nd -> val = a [ mid ] ;
        nd -> son [ 0 ] = build ( nd , l , mid - 1 , a ) ;
        nd -> son [ 1 ] = build ( nd , mid + 1 , r , a ) ;
        nd -> update ( ) ;
        return    nd ;
}
void    rotate ( Node    * nd , int    pd ) {
        int    tmp = 1413213 ;
        Node    * p = nd -> fa ; tmp = p -> siz ;
        Node    * s = nd -> son [ ! pd ] ; tmp = s -> siz ;
        Node    * ss = s -> son [ pd ] ; tmp = ss -> siz ;

        if ( p != zero )    p -> son [ nd == p -> son [ 1 ] ] = s ;
        else    root = s ;
        s -> son [ pd ] = nd ;
        nd -> son [ ! pd ] = ss ;

        s -> fa = p ;
        nd -> fa = s ;
        if ( ss != zero )    ss -> fa = nd ;

        nd -> update ( ) ;
        s -> update ( ) ;
}
void    splay ( Node    * nd , Node    * top = zero ) {
        while ( nd -> fa != top ) {
                Node    * p = nd -> fa ;
                Node    * pp = p -> fa ;
                int    nl = nd == p -> son [ 0 ] ;
                int    pl = p == pp -> son [ 0 ] ;
                if ( pp == top )
                        rotate ( p , nl ) ;
                else {
                        if ( pl == nl ) {
                                rotate ( pp , pl ) ;
                                rotate ( p , nl ) ;
```

```
                }
                else {
                    rotate ( p , nl ) ;
                    rotate ( pp , pl ) ;
                }
            }
        }
    }
    Node    * find ( int    pos ) {
        Node    * nd = root ;
        while ( 1 ) {
            nd -> push_down ( ) ;
            if ( pos <= nd -> son [ 0 ] -> siz )
                nd = nd -> son [ 0 ] ;
            else    if ( pos >= nd -> son [ 0 ] -> siz + 2 )
                pos -= nd -> son [ 0 ] -> siz + 1 , nd = nd -> son [ 1 ] ;
            else {
                splay ( nd ) ;
                return    nd ;
            }
        }
    }
    void    erase ( int    pos ) {
        Node    * lnd = find ( pos - 1 ) ;
        Node    * rnd = find ( pos + 1 ) ;
        splay ( lnd ) ;
        splay ( rnd , lnd ) ;
        rnd -> son [ 0 ] -> fa = zero ;
        rnd -> son [ 0 ] = zero ;
        rnd -> update ( ) ;
        lnd -> update ( ) ;
    }
    void    erase ( int    l , int    r ) {
        Node    * lnd = find ( l - 1 ) ;
        Node    * rnd = find ( r + 1 ) ;
        splay ( lnd ) ;
        splay ( rnd , lnd ) ;
        rnd -> son [ 0 ] -> fa = zero ;
        rnd -> son [ 0 ] = zero ;
        rnd -> update ( ) ;
        lnd -> update ( ) ;
    }
    void    reverse ( int    l , int    r ) {
        Node    * lnd = find ( l - 1 ) ;
```

```
            Node    * rnd = find ( r + 1 ) ;
            splay ( lnd ) ;
            splay ( rnd , lnd ) ;
            Node    * ls = rnd -> son [ 0 ] ;
            ls -> tag ^= 1 ;
            ls -> push_down ( ) ;
            splay ( ls ) ;
        }
}
work ;


int    main ( ) {
        scanf ( "%d%d" , & n , & m ) ;
        for ( int    i = 2 ; i <= n + 1 ; i ++ )
            a [ i ] = i - 1 ;
        a [ 1 ] = a [ n + 2 ] = inf ;
        work . init ( ) ;
        for ( int    i = 1 ; i <= m ; i ++ ) {
            scanf ( "%d%d" , & x , & y ) ;
            work . reverse ( x + 1 , y + 1 ) ;
        }
        for ( int    i = 1 ; i <= n ; i ++ )
            printf ( "%d " , work . find ( i + 1 ) -> val ) ;
        return    0 ;
}
```

数学

结论&&推论

一些结论

(a ^ b + a & b) = a + b

B 进制下一个如果一个数各个数位和是 p 的倍数那么这个数被 p 整除的充要条件是 p 是 B-1 的约数

能被 7 整除的数的性质：末三位之前的数和末三位的差能被 7 整除

关于 gcd

gcd(x^a − 1, x^b − 1) = x^gcd(a,b) − 1

gcd(2a − 1, 2b − 1) = 1 当且仅当 gcd(a, b) = 1

gcd(a^n − b^n, a^m − b^m) = a^gcd(n,m) − b^gcd(n,m)

gcd(fib(n), fib(m)) = fib(gcd(n,m))

一些求和公式

平方和：1^2 + 2^2 + ⋯ + n^2 = n(n+1)(n+2)/6

立方和：1^3 + 2^3 + ⋯ + n^3 = (1 + 2 + 3 + ⋯ + n)^2

错位加和：$\sum_{i=1}^{n} i(n-i+1)$ = n(n+1)(n+2)/6

斯特林公式

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} - \frac{571}{2488320n^4} + \cdots\right)$$

对 10 取对数+1 得到数字长度

外观数列

外观数列是这样一个数列，首项为 1，以后的每一项都是对前一项的描述，数列如下

1, 11, 21, 1211, 111221, 312211, 13112221, 1113213211, ……

比如 312211 是对 111221 的描述，即 3 个 1，2 个 2，1 个 1。

1987 年，康威（John Conway）发现，在这个数列中，相邻两数的长度之比越来越接近一个固定的常数，当数列项数趋近无穷大的时候，该比值约为 1.303577，此常数叫做康威常数。外观数列还有一个性质，4 永远不出现。

整数拆分问题

问题：给定一个自然数 n，把它拆分为若干个数的和，记这若干个数的积为 M，求 M 的最大值。

结论：使乘积最大的拆分方案是，先拆分出尽多可能的 3，如果剩余的数为 2 或 0，则拆分结束。如果剩余的数为 1，则将拆分好的 3 拿一个和剩余的 1 组合拆分为两个 2

快速乘法

```
long long mul (long long a, long long b, long long mod) {
    if (b == 0)
        return 0;
    long long ans = mul (a, b>>1, mod);
    ans = ans*2%mod;
    if (b&1) ans += a, ans %= mod;
    return ans;
}
```

逆元

拓展欧几里得

```
long long ex_gcd (long long a, long long b, long long &x, long long &y)
{
	if (a == 0 && b == 0) return -1;
	if (b == 0)
	{
		x=1;
		y=0;
		return a;
	}
	long long d = ex_gcd (b, a%b, y, x);
	y -= a/b*x;
	return d;
}

long long mod_rev (long long a,long long n)
{
	long long x,y;
	long long d = ex_gcd (a,n,x,y);
	if (d == 1) return (x%n+n)%n;
	else return -1;
}
```

[1, n]素数个数

```
//可以求 1e11 以内
#define MAXN 100
#define MAXM 50010
#define MAXP 666666
#define MAX 5000010
#define clr(ar) memset(ar, 0, sizeof(ar))
#define chkbit(ar, i) (((ar[(i) >> 6]) & (1 << (((i) >> 1) & 31))))
#define setbit(ar, i) (((ar[(i) >> 6]) |= (1 << (((i) >> 1) & 31))))
#define isprime(x) (( (x) && ((x)&1) && (!chkbit(ar, (x)))) || ((x) ==2))

namespace pcf
{
long long dp[MAXN][MAXM];
unsigned int ar[(MAX >> 6) + 5] = {0};
```

```
int len = 0, primes[MAXP], counter[MAX];
void Sieve()
{
	setbit(ar, 0), setbit(ar, 1);
	for (int i = 3; (i * i) < MAX; i++, i++)
	{
		if (!chkbit(ar, i))
		{
			int k = i << 1;
			for (int j = (i * i); j < MAX; j += k) setbit(ar, j);
		}
	}
	for (int i = 1; i < MAX; i++)
	{
		counter[i] = counter[i - 1];
		if (isprime(i)) primes[len++] = i, counter[i]++;
	}
}
void init()
{
	Sieve();
	for (int n = 0; n < MAXN; n++)
	{
		for (int m = 0; m < MAXM; m++)
		{
			if (!n) dp[n][m] = m;
			else dp[n][m] = dp[n - 1][m] - dp[n - 1][m / primes[n -
				1]];
		}
	}
}
long long phi(long long m, int n)
{
	if (n == 0) return m;
	if (primes[n - 1] >= m) return 1;
	if (m < MAXM && n < MAXN) return dp[n][m];
	return phi(m, n - 1) - phi(m / primes[n - 1], n - 1);
}
long long Lehmer(long long m)
{
	if (m < MAX) return counter[m];
	long long w, res = 0;
	int i, a, s, c, x, y;
	s = sqrt(0.9 + m), y = c = cbrt(0.9 + m);
```

```
        a = counter[y], res = phi(m, a) + a - 1;
        for (i = a; primes[i] <= s; i++) res = res - Lehmer(m / primes[
                        i]) + Lehmer(primes[i]) - 1;
        return res;
}
}
int main ()
{
        pcf::init ();
        long long n;
        while (cin >> n)
        {
                cout << pcf::Lehmer(n) << endl;
        }
}
```

pell 方程

形如 x^2 - d*y^2 = 1 的不定方程是 pell 方程。如果 d 是完全平方数那么方程无解。
当找到一组最小特解以后其他的解可以利用矩阵迭代算出：

$$\begin{bmatrix} x_k \\ y_k \end{bmatrix} = \begin{bmatrix} x_1 & d \times y_1 \\ y1 & x_1 \end{bmatrix}^{k-1} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

秦九韶算法

$$f(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

可以对这个多项式进行如下改写

$$\begin{aligned} f(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \\ &= a_0 + (a_1 + a_2 x + \ldots + a_n x^{n-1}) x \\ &= \ldots \\ &= a_0 + (a_1 + \ldots + (a_{n-2} + (a_{n-1} + a_n x) x) x \ldots) x \end{aligned}$$

由内到外就是

$$v_0 = a_n$$

$$v_1 = v_0 x + a_{n-1}$$

$$v_2 = v_1 x + a_{n-2}$$

$$v_3 = v_2 x + a_{n-3}$$

$$\cdots\cdots$$

$$v_n = v_{n-1} x + a_0$$

算法复杂度 $O(n)$

求 π

马青公式是英国天文学教授约翰·马青于 1706 年发现的，用它来计算圆周率的值非常高效。公式内容如下

$$\frac{\pi}{4} = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

展开 arctan 为幂级数， 然后用秦九韶算法

```
#include <iostream>
#include <string.h>
#include <stdio.h>

using namespace std;
typedef long long LL;

const int BASE    = 10000;
const int DIGIT = 10000;
const int NUM      = DIGIT >> 2;

int res[NUM], A[NUM], B[NUM], C[NUM];

void set(int res[], int x)
{
      for(int i = 0; i < NUM; i++)
            res[i] = 0;
      res[0] = x;
}
```

```
void copy(int res[], int x[])
{
	for(int i = 0; i < NUM; i++)
		res[i] = x[i];
}

bool zero(int res[])
{
	for(int i = 0; i < NUM; i++)
		if(res[i]) return false;
	return true;
}

void add(int res[], int x[])
{
	for(int i = NUM - 1; i >= 0; i--)
	{
		res[i] += x[i];
		if(res[i] >= BASE && i > 0)
		{
			res[i] -= BASE;
			res[i - 1]++;
		}
	}
}

void sub(int res[], int x[])
{
	for(int i = NUM - 1; i >= 0; i--)
	{
		res[i] -= x[i];
		if(res[i] < 0 && i > 0)
		{
			res[i] += BASE;
			res[i - 1]--;
		}
	}
}

void multi(int res[], int x)
{
	int t = 0;
	for(int i = NUM - 1; i >= 0; i--)
```

```
	{
		res[i] *= x;
		res[i] += t;
		t = res[i] / BASE;
		res[i] %= BASE;
	}
}

void div(int res[], int x)
{
	int t = 0;
	for(int i = 0; i < NUM; i++)
	{
		res[i] += t * BASE;
		t = res[i] % x;
		res[i] /= x;
	}
}

void arctan(int res[], int A[], int B[], int x)
{
	int m = x * x;
	int cnt = 1;
	set(res, 1);
	div(res, x);
	copy(A, res);
	do{
		div(A, m);
		copy(B, A);
		div(B, 2 * cnt + 1);
		if(cnt & 1)
			sub(res, B);
		else
			add(res, B);
		cnt++;
	}while(!zero(B));
}

void print(int res[])
{
	printf("%d.\n", res[0]);
	for(int i = 1; i < NUM; i++)
	{
		printf("%04d", res[i]);
```

```
            if(i % 15 == 0) puts("");
        }
        puts("");
}

void Machin()
{
        arctan(res, A, B, 5);
        multi(res, 4);
        arctan(C, A, B, 239);
        sub(res, C);
        multi(res, 4);
        print(res);
}

int main()
{
        Machin();
        return 0;
}
```

黑科技

求某天是星期几

```
int CaculateWeekDay(int y,int m,int d)
{
        if (m==1||m==2)
        {
                m += 12;
                y--;
        }

        if(y<1752||(y==1752&&m<9)||(y==1752&&m==9&&d<3))
        {
                a = (d+2*m+3*(m+1)/5+y+y/4+5)%7;
        }
        else
        {
                a = (d+2*m+3*(m+1)/5+y+y/4-y/100+y/400)%7;
        }
```

```
    return a + 1;
}
```

扩栈

```
#pragma comment(linker, "/STACK:1024000000,1024000000")
```

JAVA

```
import java.math.BigInteger;
import java.util.Scanner;
public class Main
{
    void solve ()
    {
        Scanner cin = new Scanner(System.in);
        BigInteger f1, f2, f3, f4, ans;
        while (cin.hasNext ())
        {
            int n = cin.nextInt ();
            f1 = BigInteger.valueOf (1);
            f2 = f3 = f4 = ans = f1;
            if (n <= 4)
            {
                System.out.println ("1");
                continue;
            }
            for (int j = 5; j <= n; j++)
            {
                ans = f1.add (f2.add (f3.add (f4)));
                f1 = f2;
                f2 = f3;
                f3 = f4;
                f4 = ans;
            }
            System.out.println (ans);
        }
    }
    public static void main (String[] args)
    {
        Main work = new Main();
        work.solve ();
    }
```

```
}
```

高精度小数，要去掉末尾的后导 0.

```
Bigdecimal a, b;
a = a.divide (b, 100, BigDecimal.ROUND_CEILING); //除数  位数  舍入方式

import java.math.*;
import java.util.*;
public class Main
{
    void solve ()
    {
        //BigInteger a, b, c;
        Scanner cin = new Scanner(System.in);
        BigDecimal a = BigDecimal.valueOf (0);
        BigDecimal b = BigDecimal.valueOf (0);
        while (cin.hasNext ())
        {
            a = cin.nextBigDecimal ();
            b = cin.nextBigDecimal ();
            System.out.println (a.add (b).stripTrailingZeros().toPlainString());
        }
    }
    public static void main (String[] args)
    {
        Main work = new Main();
        work.solve ();
    }
}
```

builtin 函数

GCC 提供了一系列的 builtin 函数，可以实现一些简单快捷的功能来方便程序编写，另外，很多 builtin 函数可用来优化编译结果。这些函数以“__builtin_”作为函数名前缀。
很多 C 标准库函数都有与之对应的 GCC builtin 函数，例如 strcpy()有对应的__builtin_strcpy()内建函数。

下面就介绍一些 builtin 函数及其作用：

__builtin_ffs(x)：返回 x 中最后一个为 1 的位是从后向前的第几位，如__builtin_ffs(0x789)=1, __builtin_ffs(0x78c)=3。于是，__builtin_ffs(x) - 1 就是 x 中最后一个为 1 的位的位置。

__builtin_popcount(x)：x 中 1 的个数。

__builtin_ctz(x)：x 末尾 0 的个数。x=0 时结果未定义。

__builtin_clz(x)：x 前导 0 的个数。x=0 时结果未定义。

上面的宏中 x 都是 unsigned int 型的，如果传入 signed 或者是 char 型，会被强制转换成 unsigned int。

__builtin_parity(x)：x 中 1 的奇偶性。

__builtin_return_address(n)：当前函数的第 n 级调用者的地址，用的最多的就是 __builtin_return_address(0)，即获得当前函数的调用者的地址。注意，该函数实现是体系结构相关的，有些体系结构只实现了 n=0 的返回结果。

uint16_t __builtin_bswap16 (uint16_t x)

uint32_t __builtin_bswap32 (uint32_t x)：按字节翻转 x，返回翻转后的结果。

__builtin_prefetch (const void *addr, ...)：它通过对数据手工预取的方法，在使用地址 addr 的值之前就将其放到 cache 中，减少了读取延迟，从而提高了性能，但该函数也需要 CPU 的支持。该函数可接受三个参数，第一个参数 addr 是要预取的数据的地址，第二个参数可设置为 0 或 1（1 表示我对地址 addr 要进行写操作，0 表示要进行读操作），第三个参数可取 0-3（0 表示不用关心时间局部性，取完 addr 的值之后便不用留在 cache 中，而 1、2、3 表示时间局部性逐渐增强）。

__builtin_constant_p (exp)：判断 exp 是否在编译时就可以确定其为常量，如果 exp 为常量，该函数返回 1，否则返回 0。如果 exp 为常量，可以在代码中做一些优化来减少处理 exp 的复杂度。

__builtin_types_compatible_p(type1, type2)：判断 type1 和 type2 是否是相同的数据类型，相同返回 1，否则返回 0。该函数不区分 const/volatile 这样的修饰符，即 int 和 const int 被认为是相同的类型。

```
#define foo(x)
({
typeof(x) tmp = (x);\
if(__builtin_types_compatible_p(typeof(x), int))\
//do something...\
else \
```

```
//do something...\
tmp;
})
```

__builtin_expect (long exp, long c)：用来引导 gcc 进行条件分支预测。在一条指令执行时，由于流水线的作用，CPU 可以完成下一条指令的取指，这样可以提高 CPU 的利用率。在执行一条条件分支指令时，CPU 也会预取下一条执行，但是如果条件分支跳转到了其他指令，那 CPU 预取的下一条指令就没用了，这样就降低了流水线的效率。内核中的 likely()和 unlikely()就是通过__builtin_expect 来实现的。

__builtin_expect (long exp, long c)函数可以优化程序编译后的指令序列，使指令尽可能的顺序执行，从而提高 CPU 预取指令的正确率。该函数的第二个参数 c 可取 0 和 1，

例如：

```
if (__builtin_expect (x, 0))
        foo ();
```

表示 x 的值大部分情况下可能为 0，因此 foo()函数得到执行的机会比较少。gcc 就不必将 foo()函数的汇编指令紧挨着 if 条件跳转指令。

由于第二个参数只能取整数，所以如果要判断指针或字符串，可以像下面这样写：

```
if (__builtin_expect (ptr != NULL, 1))
        foo (*ptr);
```

表示 ptr 一般不会为 NULL，所以 foo 函数得到执行的概率较大，gcc 会将 foo 函数的汇编指令放在挨着 if 跳转执行的位置。