Contents

2-SAT 问题

- 题目：poj3678

- 依赖：

模板描述

给定 n 个布尔变量 $a_1, a_2, \ldots, a_n$，m 个限定条件 $limit(f, i, j, v)$ 限制 $f(a_i, a_j) = v$。求该组限制条件是否可以达成，并给出一组解。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$

算法简述

将每个布尔变量拆成两个点 a_i^p, a_i^n，分别代表点 i 的正值和负值。每一个布尔操作的断言 $f(a_i, a_j) = v$ 都可以转化成某两个拆开的点后的连接。在程序内表现为 `constrain(p, 0, q, 0)`，代表若变量 $a_p = 0$ 则 $a_q = 1$。跑一遍强连通分量，若一个点拆开的两个点在一个强连通分量中则出现矛盾，否则 dfs 所有点即可获得一组解。所有可能解的总数为 2^{bcnt}，其中 $bcnt$ 为强连通分量的个数。

调用方法

- `int maxn`：点的数量上限。
- `void set_true(int p)`：$a_p = 1$
- `void set_false(int p)`：$a_p = 0$
- `void require_and(int p, int q)`：$a_p \wedge a_q = 1$
- `void require_or(int p, int q)`：$a_p \vee a_q = 1$
- `void require_nand(int p, int q)`：$a_p \wedge a_q = 0$
- `void require_nor(int p, int q)`：$a_p \vee a_q = 0$
- `void require_xor(int p, int q)`：$a_p \oplus a_q = 1$
- `void require_xnor(int p, int q)`：$a_p \oplus a_q = 0$
- `bool eval(int[] res)`：求解限制组，结果存入 res，若无解返回 `false`。
- `void init(int n)`：初始化点数为 n 的限制组。

```
class TwoSAT
{
public:
    int n;
    #define constrain(_p, _vp, _q, _vq)  \
            graph.add_edge(2 * (_p) - (_vp), 2 * (_q) - (_vq))
    void set_true(int p) {
```

```
    constrain(p, 0, p, 1); }
void set_false(int p) {
    constrain(p, 1, p, 0); }
void require_and(int p, int q) {
    constrain(p, 0, p, 1);
    constrain(q, 0, q, 1); }
void require_or(int p, int q) {
    constrain(p, 0, q, 1);
    constrain(q, 0, p, 1); }
void require_nand(int p, int q) {
    constrain(p, 1, q, 0);
    constrain(q, 1, p, 0); }
void require_nor(int p, int q) {
    constrain(p, 1, p, 0);
    constrain(q, 1, q, 0); }
void require_xor(int p, int q) {
    constrain(p, 1, q, 0);
    constrain(q, 1, p, 0);
    constrain(p, 0, q, 1);
    constrain(q, 0, p, 1); }
void require_xnor(int p, int q) {
    constrain(p, 1, q, 1);
    constrain(q, 1, p, 1);
    constrain(p, 0, q, 0);
    constrain(q, 0, p, 0); }
void require_eq(int p, int q) {
    require_xnor(p, q); }
void require_neq(int p, int q) {
    require_xor(p, q); }
#undef constrain
void dfs(int p, int res[])
{
    int id = (p + 1) / 2;
    if (res[id] != -1)
        return ;
    res[id] = 2 * id - p;
    for (auto* ep = graph.edges[p]; ep; ep = ep->next)
        dfs(ep->v, res);
    return ;
}
bool eval(int res[])
{
    graph.eval();
    rep(i, 1, n)
        if (graph.belong[2 * i] == graph.belong[2 * i - 1])
            return false;
    rep(i, 1, n)
        res[i] = -1;
    rep(i, 1, n)
        if (res[i] == -1)
            dfs(2 * i, res);
    return true;
}
void init(int n)
```

```
    {
        this->n = n;
        graph.init(2 * n);
        return ;
    }
} tsat;
```

Bron-Kerbosch 最大团算法

- 题目：poj1419
- 依赖：

模板描述

给定 n 个点 m 条边的无向图 $G = (V, E)$，求无向图的最大全连通分量。

最大独立集：取 $V' \subseteq V$ 且 $|V'| = k$，同时 V' 内点两两无连接。

最大团：取 $V' \subseteq V$ 且 $|V'| = k$，同时 V' 内点两两相连，也作全连通分量。

复杂度

- Space:
  - Worst Case: $O(n^2)$
  - Amortized: $O(n^2)$
  - Best Case: $O(n^2)$
- Time:
  - Worst Case: $O(3^{\frac{n}{3}})$
  - Amortized: $O(3^{\frac{n}{3}})$
  - Best Case: $O(n^2)$

算法简述

一个无向图的最大独立集为其补图的最大团。

补图：$G' = (V, U \backslash E)$ 是 G 的补图，即边联通状态完全相反。

暴力搜索所有团，适当剪枝。

一个图的最大团数量不会超过 $3^{\frac{n}{3}}$ 个。

调用方法

- `int maxn`：点的数量上限。
- `void add_edge(int u, int v)`：添加一条 $u \leftrightarrow v$ 的有向边。
- `void remove_edge(int u, int v)`：删除原来在图中的一条 $u \leftrightarrow v$ 的无向边。
- `int[] eval()`：计算该无向图的最大团大小，及其点构成。

- `void init(int n, bool state)`：对任意的边 (u,v)，初始其状态为 $state$。

```
class BronKerbosch
{
public:
    int n;
    bool edge[maxn][maxn];
    bool vis[maxn];
    void add_edge(int u, int v)
    {
        edge[u][v] = edge[v][u] = true;
        return ;
    }
    void remove_edge(int u, int v)
    {
        edge[u][v] = edge[v][u] = false;
        return ;
    }
    void dfs(int p, int& curn, int& bestn, vector<int>& res)
    {
        if (p > n) {
            res.clear();
            rep(i, 1, n)
                if (vis[i])
                    res.push_back(i);
            bestn = curn;
            return ;
        }
        bool flag = true;
        rep(i, 1, p - 1)
            if (vis[i] && !edge[i][p]) {
                flag = false;
                break;
            }
        if (flag) {
            curn += 1;
            vis[p] = true;
            dfs(p + 1, curn, bestn, res);
            curn -= 1;
            vis[p] = false;
        }
        if (curn + n - p > bestn)
            dfs(p + 1, curn, bestn, res);
        return ;
    }
    vector<int> eval(void)
    {
        int curn = 0, bestn = 0;
        vector<int> res;
        dfs(1, curn, bestn, res);
        return res;
    }
```

```
    void init(int n, bool state = false)
    {
        this->n = n;
        rep(i, 1, n)
            rep(j, 1, n)
                edge[i][j] = state;
        memclr(vis, n);
        return ;
    }
} graph;
```

中国剩余定理

- 题目：
- 依赖：euclid_gcd

模板描述

给定以下方程组，求 x 的最小非负整数解：

$$\begin{cases} x \equiv a_1 \bmod m_1 \\ x \equiv a_2 \bmod m_2 \\ x \equiv a_3 \bmod m_3 \\ \dots \\ x \equiv a_n \bmod m_n \end{cases}$$

其中 $m_1, m_2, \dots, m_n$ 两两互质。

进一步地，当 $m_1, m_2, \dots, m_n$ 两两不一定互质时，求解对应的 x。

复杂度

- Space:
  - Worst Case: $O(n)$
  - Amortized: $O(n)$
  - Best Case: $O(n)$
- Time:
  - Worst Case: $O(n \log m_i)$
  - Amortized: $O(n \log m_i)$
  - Best Case: $O(n)$

算法简述

记 M 为所有 m_i 的最小公倍数，则应有：

$$M = \prod_{i=1}^{n} m_i$$

又记 t_i 为同余方程 $\frac{M}{m_i}t_i \equiv 1 \bmod m_i$ 的最小非负整数解，则存在所求的解 x 满足：

$$x = \sum_{i=1}^{n} a_i \frac{M}{m_i} t_i$$

通解则为 $x^* = x + kM$。

进一步地，推广到当 $m_1, m_2, \ldots, m_n$ 不一定两两互质的条件下，我们假定前 $n-1$ 个同余方程已经被求解，且此时已有 $M = \prod_{i=1}^{n-1} m_i$，$n$ 个同余方程构成的方程组的解应为：

$$t_n M \equiv a_n - x \bmod m_n$$

我们可以调用扩展欧几里得算法求得此同余方程。本质上，扩展中国剩余定理就是求解 n 次扩展欧几里得。

调用方法

- `lli crt_solve(lli[] a, lli[] m, int n)`：求解由 n 个方程构成的同余方程组，其中保证 m 内整数两两互质，返回最小的非负整数解 x。
- `lli excrt_solve(lli[] a, lli[] m, int n)`：求解由 n 个方程构成的同余方程组，其中 m 内整数不一定两两互质，返回最小的非负整数解 x。若该方程组无解，返回 -1。

```
lli crt_solve(lli a[], lli m[], int n)
{
    lli res = 0, lcm = 1, t, tg, x, y;
    rep(i, 1, n)
        lcm *= m[i];
    rep(i, 1, n) {
        t = lcm / m[i];
        exgcd(t, m[i], x, y);
        x = ((x % m[i]) + m[i]) % m[i];
        res = (res + t * x * a[i]) % lcm;
    }
    return (res + lcm) % lcm;
}

lli excrt_solve(lli a[], lli m[], int n)
{
    lli cm = m[1], res = a[1], x, y;
    rep(i, 2, n) {
        lli A = cm, B = m[i], C = (a[i] - res % B + B) % B,
            gcd = exgcd(A, B, x, y),
            Bg = B / gcd;
        if (C % gcd != 0)
            return -1;
        x = (x * (C / gcd)) % Bg;
        res += x * cm;
        cm *= Bg;
```

```
        res = (res % cm + cm) % cm;
    }
    return (res % cm + cm) % cm;
}
```

Tarjan 边双连通分量

- 题目：poj3352
- 依赖：

模板描述

给定 n 个点 m 条边的无向图 $G = (V, E)$，求 G 的边双连通分量。

割边：删掉该边以后无向连通图被分割成两个连通子图，也称作桥。

边双连通图：不存在割边的无向连通图，保证任意两个点间至少存在两条不含共同边的路径。

边双连通分量：一个无向图的极大边双联通子图。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$

算法简述

记数组 dfn_i 为点 i 被 dfs 到的次序编号（时间戳），low_i 为 i 和 i 的子树能够追溯到的最早的堆栈中的节点的时间戳。维护一个堆栈用于储存要处理的连通分量。两遍 dfs，第一次标记所有桥，第二次通过桥的标记求出分量。具体做法见代码。

边双连通图有以下性质：

- 在分量内的任意两个点总可以找到两条边不相同的路径互相到达。
- 若一个边被删除后形成两个边连通图，则该边为原图的桥。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：无向边的数量上限。
- `int[] belong`：点 i 属于第 $belong_i$ 个边双连通分量。

- `int[] bsize`：第 i 个边双连通分量的大小为 $bsize_i$。
- `edge[] bridges`：所有割边的点对，保证 $u < v$。
- `bool edge.is_bridge`：标记该边是否为桥。
- `void add_edge(int u, int v)`：添加一条 $u \leftrightarrow v$ 的无向边。
- `int eval()`：求图 G 的边双连通分量，并返回边双连通分量的个数。
- `void init(int n)`：初始化点数为 n 的图。

```
class Tarjan
{
public:
    struct edge
    {
        int u, v;
        bool is_bridge;
        edge *next, *rev;
    } epool[maxm], *edges[maxn];
    int n, ecnt, dcnt, bcnt;
    int dfn[maxn], low[maxn];
    int belong[maxn], bsize[maxn];
    vector<pair<int, int>> bridges;
    void add_edge(int u, int v)
    {
        edge *p = &epool[++ecnt],
             *q = &epool[++ecnt];
        p->u = u; p->v = v; p->is_bridge = false;
        p->next = edges[u]; edges[u] = p;
        q->u = v; q->v = u; q->is_bridge = false;
        q->next = edges[v]; edges[v] = q;
        p->rev = q; q->rev = p;
        return ;
    }
    void dfs1(int p, int par)
    {
        dfn[p] = low[p] = ++dcnt;
        for (edge *ep = edges[p]; ep; ep = ep->next) {
            int q = ep->v;
            if (!dfn[q]) {
                dfs1(q, p);
                minimize(low[p], low[q]);
                if (low[q] > dfn[p])
                    ep->is_bridge = ep->rev->is_bridge = true;
            } else if (dfn[q] < dfn[p] && q != par) {
                minimize(low[p], dfn[q]);
            }
        }
        return ;
    }
    void dfs2(int p)
    {
        dfn[p] = true;
        belong[p] = bcnt;
        bsize[bcnt] += 1;
```

```
        for (edge *ep = edges[p]; ep; ep = ep->next) {
            if (ep->is_bridge) {
                if (ep->u < ep->v)
                    bridges.push_back(make_pair(ep->u, ep->v));
                continue;
            }
            if (!dfn[ep->v])
                dfs2(ep->v);
        }
        return ;
    }
    int eval(void)
    {
        dcnt = bcnt = 0;
        rep(i, 1, n) {
            dfn[i] = low[i] = 0;
            belong[i] = 0;
        }
        rep(i, 1, n)
            if (!dfn[i])
                dfs1(i, 0);
        rep(i, 1, n)
            dfn[i] = false;  // use dfs[] as vis[]
        bridges.clear();
        rep(i, 1, n)
            if (!dfn[i]) {
                bsize[++bcnt] = 0;
                dfs2(i);
            }
        return bcnt;
    }
    void init(int n)
    {
        this->n = n;
        ecnt = 0;
        rep(i, 1, n)
            edges[i] = nullptr;
        return ;
    }
} graph;
```

Tarjan 点双连通分量

- 题目：poj1144
- 依赖：

模板描述

给定 n 个点 m 条边的无向图 $G = (V, E)$，求 G 的点双连通分量。

割点：删掉该点以后无向连通图被分割成两个连通子图。

点双连通图：不存在割点的无向连通图。

点双连通分量：一个无向图的极大点双联通子图。

复杂度

- Space:
  - Worst Case: $O(n+m)$
  - Amortized: $O(n+m)$
  - Best Case: $O(n+m)$
- Time:
  - Worst Case: $O(n+m)$
  - Amortized: $O(n+m)$
  - Best Case: $O(n+m)$

算法简述

记数组 dfn_i 为点 i 被 dfs 到的次序编号（时间戳），low_i 为 i 和 i 的子树能够追溯到的最早的堆栈中的节点的时间戳。维护一个堆栈用于储存要处理的连通分量。若回溯时目标节点 low_i 不小于当前点 dfn 值，则不断出栈直到目标节点。具体做法见代码。

点双连通图有以下性质：

- 任意两点间至少存在两条点不重复的路径。
- 图中删去任意一个点都不会改变图的连通性。
- 若点双连通分量之间存在公共点，则这个点为原图的割点。
- 无向连通图中割点必定属于至少两个点双连通分量，非割点属于且恰属于一个。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：无向边的数量上限。
- `int[] belong`：点 i 属于第 $belong_i$ 个点双连通分量。
- `int[] bsize`：第 i 个点双连通分量的大小为 $bsize_i$。
- `bool[] is_cut`：标记点 i 是否为割点。
- `void add_edge(int u, int v)`：添加一条 $u \leftrightarrow v$ 的无向边。
- `int eval()`：求图 G 的点双连通分量，并返回点双连通分量的个数。
- `void init(int n)`：初始化点数为 n 的图。

```
class Tarjan
{
public:
    struct edge
    {
        int u, v;
        edge *next;
```

```
} epool[maxm], *edges[maxn];
int n, ecnt, dcnt, bcnt;
stack<edge*> stk;
int instk[maxn], dfn[maxn], low[maxn];
int belong[maxn], bsize[maxn];
bool is_cut[maxn];
void add_edge(int u, int v)
{
    edge *p = &epool[++ecnt],
         *q = &epool[++ecnt];
    p->u = u; p->v = v;
    p->next = edges[u]; edges[u] = p;
    q->u = v; q->v = u;
    q->next = edges[v]; edges[v] = q;
    return ;
}
void dfs(int p, int par)
{
    int child = 0;
    dfn[p] = low[p] = ++dcnt;
    for (edge *ep = edges[p]; ep; ep = ep->next) {
        int q = ep->v;
        if (!dfn[q]) {
            stk.push(ep);
            child += 1;
            dfs(ep->v, p);
            minimize(low[p], low[q]);
            if (dfn[p] <= low[q]) {
                is_cut[p] = true;
                bcnt += 1;
                edge *eq = nullptr;
                do {
                    eq = stk.top();
                    stk.pop();
                    if (belong[eq->u] != bcnt) {
                        belong[eq->u] = bcnt;
                        bsize[bcnt] += 1;
                    }
                    if (belong[eq->v] != bcnt) {
                        belong[eq->v] = bcnt;
                        bsize[bcnt] += 1;
                    }
                } while (eq->u != p || eq->v != q);
            }
        } else if (dfn[q] < dfn[p] && q != par) {
            stk.push(ep);
            minimize(low[p], dfn[q]);
        }
    }
    if (par == 0 && child == 1)
        is_cut[p] = false;
    return ;
}
int eval(void)
```

```
    {
        while (!stk.empty())
            stk.pop();
        dcnt = bcnt = 0;
        rep(i, 1, n) {
            dfn[i] = low[i] = 0;
            instk[i] = iscut[i] = false;
            belong[i] = 0;
        }
        rep(i, 1, n)
            if (!dfn[i])
                dfs(i, 0);
        return bcnt;
    }
    void init(int n)
    {
        this->n = n;
        ecnt = 0;
        rep(i, 1, n)
            edges[i] = nullptr;
        return ;
    }
} graph;
```

Dijkstra 最短路

- 题目：hdu2544
- 依赖：

模板描述

给定 n 个点 m 条边的有向图 $G = (V, E)$，$s \in V$ 以及每条边的长度（无负权回路），求点 s 到 V 中任意一点的最短距离及满足距离最短的任意一条最短路径。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:
  - Worst Case: $O((n + m)\log n)$
  - Amortized: $O((n + m)\log n)$
  - Best Case: $O((n + m)\log n)$

算法简述

松弛操作：对于边 $e(i, j) \in E$，$dist_j := min(dist_j, dist_i + len_e)$

朴素 Dijkstra 算法：从起点开始，松弛每一个可达的点，并依次递归处理这些被松弛的点。

堆优化，将距离源点近的点优先弹出队列，保证最大程度减小无用松弛的次数。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：有向边的数量上限。
- `lli infinit`：规定的无限远距离。
- `lli[] dist`：对于每个点 $i \in [1,n]$，点 i 到源点 s 的距离。
- `edge[] from`：从源点到点 i 的最短路径上指向 i 的边的地址。
- `void add_edge(int u, int v, lli len)`：添加一条 $u \to v$，长度为 len 的有向边。
- `void eval(int s)`：以 s 为源点，计算到 $[1,n]$ 所有点的最短路。
- `int[] get_path(int p)`：获取一个节点编号构成的列表，构成一条 s 到 p 的最短路径。
- `void init(int n)`：初始化点数为 n 的图。

```
class Dijkstra
{
public:
    struct edge
    {
        int u, v;
        lli len;
        edge *next;
    } epool[maxm], *edges[maxn], *from[maxn];
    int n, ecnt;
    lli dist[maxn];
    bool vis[maxn];
    typedef pair<lli, int> pli;
    void add_edge(int u, int v, lli len)
    {
        edge *p = &epool[++ecnt];
        p->u = u; p->v = v; p->len = len;
        p->next = edges[u]; edges[u] = p;
        return ;
    }
    void eval(int s)
    {
        priority_queue<pli, vector<pli>, greater<pli>> pq;
        rep(i, 0, n) {
            vis[i] = false;
            dist[i] = infinit;
            from[i] = nullptr;
        }
        dist[s] = 0;
        pq.push(make_pair(dist[s], s));
```

```
        while (!pq.empty()) {
            pli pr = pq.top();
            pq.pop();
            int p = pr.second;
            if (vis[p])
                continue;
            vis[p] = true;
            for (edge *ep = edges[p]; ep; ep = ep->next)
                if (!vis[ep->v] && dist[p] + ep->len < dist[ep->v])
{
                    dist[ep->v] = dist[p] + ep->len;
                    from[ep->v] = ep;
                    pq.push(make_pair(dist[ep->v], ep->v));
                }
        }
        return ;
    }
    vector<int> get_path(int p)
    {
        stack<int> stk;
        stk.push(p);
        edge *ep = from[p];
        while (ep) {
            stk.push(ep->u);
            ep = from[ep->u];
        }
        vector<int> res;
        while (!stk.empty()) {
            res.push_back(stk.top());
            stk.pop();
        }
        return res;
    }
    void init(int n)
    {
        this->n = n;
        ecnt = 0;
        rep(i, 1, n)
            edges[i] = nullptr;
        return ;
    }
} graph;
```

Dinic 最大流

- 题目：hdu3549
- 依赖：

模板描述

给定 n 个点 m 条边的有向图 $G = (V, E)$，每条边的权值（即流量），给定源点 s 和汇点 t，求从点 s 到点 t 的最大流量。

复杂度

- Space:
  - Worst Case: $O(n+m)$
  - Amortized: $O(n+m)$
  - Best Case: $O(n+m)$
- Time:
  - Worst Case: $O(n^2m)$
  - Amortized: $O(n^2m)$
  - Best Case: $O(n+m)$

算法简述

在最大流的题目中，图被称为网络，而每条边的边权被称作流量。我们可以把最大流问题想象成这样：源点有一个水库有无限吨水，汇点也有一个水库，希望得到最多的水。有一些水管，单位时间内可以输水 $e(i,j)$ 吨。

正式地讲，最大流是指网络中满足弧流量限制条件和平衡条件且具有最大流量的可行流。

定义前向弧为和有向边方向相同的弧，反之称之为后向弧。

定义增广路为一条链，其中链上的前向弧都不饱和，后向弧都非零，从源点开始汇点结束。

增广过程为寻找一条增广路的过程。Dinic 算法的本质就是不断寻找增广路直到找不到为止。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：有向边的数量上限。
- `lli infinit`：规定的无限大流量。
- `void add_edge(int u, int v, lli flow)`：添加一条 $u \to v$，流量为 $flow$ 的有向边。
- `lli eval()`：计算该有向图的最大流。
- `void init(int n, int s, int t)`：初始化点数为 n，源点为 s，汇点为 t 的图。

```
class Dinic
{
public:
    struct edge
    {
        int u, v;
        lli flow;
        edge *next, *rev;
    } epool[maxm], *edges[maxn];
    int n, s, t, ecnt, level[maxn];
    void add_edge(int u, int v, lli flow, lli rflow = 0)
```

```
{
    edge *p = &epool[++ecnt],
         *q = &epool[++ecnt];
    p->u = u; p->v = v; p->flow = flow;
    p->next = edges[u]; edges[u] = p;
    q->u = v; q->v = u; q->flow = rflow;
    q->next = edges[v]; edges[v] = q;
    p->rev = q; q->rev = p;
    return ;
}
bool make_level(void)
{
    rep(i, 1, n)
        level[i] = 0;
    queue<int> que;
    level[s] = 1;
    que.push(s);
    while (!que.empty()) {
        int p = que.front();
        que.pop();
        for (edge *ep = edges[p]; ep; ep = ep->next)
            if (ep->flow > 0 && !level[ep->v]) {
                level[ep->v] = level[p] + 1;
                que.push(ep->v);
            }
        if (level[t] > 0)
            return true;
    }
    return false;
}
lli find(int p, lli mn)
{
    if (p == t)
        return mn;
    lli tmp = 0, sum = 0;
    for (edge *ep = edges[p]; ep && sum < mn; ep = ep->next)
        if (ep->flow && level[ep->v] == level[p] + 1) {
            tmp = find(ep->v, min(mn, ep->flow));
            if (tmp > 0) {
                sum += tmp;
                ep->flow -= tmp;
                ep->rev->flow += tmp;
                return tmp;
            }
        }
    if (sum == 0)
        level[p] = 0;
    return 0;
}
lli eval(void)
{
    lli tmp, sum = 0;
    while (make_level()) {
        bool found = false;
```

```
            while (tmp = find(s, infinit)) {
                sum += tmp;
                found = true;
            }
            if (!found)
                break;
        }
        return sum;
    }
    void init(int n, int s, int t)
    {
        this->n = n;
        this->s = s;
        this->t = t;
        ecnt = 0;
        rep(i, 1, n)
            edges[i] = nullptr;
        return ;
    }
} graph;
```

并查集

- 题目：
- 依赖：

模板描述

给定 n 个集合 $\{1\},\{2\},\ldots,\{n\}$，在线完成以下操作：

- $Query(i,j)$：查询两个数 i,j 是否在一个集合内
- $Merge(i,j)$：合并两个数 i,j 所在集合

复杂度

- Space:
  - Worst Case: $O(n)$
  - Amortized: $O(n)$
  - Best Case: $O(n)$
- $Query$ Time:
  - Worst Case: $O(\log(n))$
  - Amortized: $O(\alpha(n))$
  - Best Case: $O(1)$
- $Merge$ Time:
  - Worst Case: $O(\log(n))$
  - Amortized: $O(\alpha(n))$
  - Best Case: $O(1)$

算法简述

复杂度分析中的 $\alpha(n)$ 函数为反 Ackermann 函数，在一般数据范围内视作常数复杂度。

每个数记一个 $id[i]$ 标记，为该数所在的集合的标志。合并两个集合 p,q 时，仅需设 $id[p]:=q$ 即可。这样形成了一个树结构，但为了避免路径长度变成 $O(n)$ 复杂度，我们需要执行路径压缩。具体地就是在查询某个点 p 的祖先 q 时，将该条路径上所有点的标记 $id[i]$ 修改成 q。这样我们可以保证每个非祖先点被访问的次数与查询的次数之比不超过某个常数。进一步地，为防止不完整的路径压缩，我们将大小更小的集合合并到更大的集合上，即若 $size[p]<size[q]$，则有 $id[p]:=q$，反之 $id[q]:=p$。

在并查集上可维护多种值，例如集合的和、异或和等等。

TODO：各种花式并查集技巧待填坑

调用方法

- `int find(int p)`：查询某点 p 的祖先。
- `bool joined(int p, int q)`：询问两个点 p,q 是否处在一个集合内，即 $Query$ 操作。
- `void join(int p, int q)`：将两个点 p,q 所在集合合并，即 $Merge$ 操作。

```
class DisjointSet
{
public:
    int n, par[maxn], size[maxn];
    lli vsm[maxn];
    int find(int p)
    {
        if (par[p] != p)
            par[p] = find(par[p]);
        return par[p];
    }
    bool joined(int p, int q)
    {
        return find(p) == find(q);
    }
    void join(int p, int q)
    {
        int gp = find(p),
            gq = find(q);
        if (gp == gq)
            return ;
        if (size[gq] < size[gp])
            swap(gq, gp);
        par[gp] = gq;
        size[gq] += size[gp];
        vsm[gq] += vsm[gp];
        return ;
```

```
    }
    void init(int n, lli w[] = nullptr)
    {
        rep(i, 1, n) {
            par[i] = i;
            size[i] = 1;
            vsm[i] = w ? w[i] : 0;
        }
        return ;
    }
} dsu;
```

Euclid 算法

- 题目：
- 依赖：

模板描述

给定整数 a, b，计算 $\gcd(a, b)$。

给定整数 a, b，计算 $\DeclareMathOperator{lcm}{lcm}\lcm(a, b)$。

求解同余方程 $ax + by = \gcd(a, b)$。

复杂度

- Space:
  - Worst Case: $O(1)$
  - Amortized: $O(1)$
  - Best Case: $O(1)$
- Time:
  - Worst Case: $O(\log(a + b))$
  - Amortized: $O(\log(a + b))$
  - Best Case: $O(1)$

算法简述

若有 $r_i = r_{i+1}q_{i+1} + r_{i+2}$，且 $r_n = 0$，则有 $r_{i+2} = r_i \mathrm{mod} r_{i+1}$。由共 $n - 2$ 个式子逆推可知任意的 r_i 可整除 r_n，正确性易得证。

调用方法

- `lli gcd_iterative(lli a, lli b)`：递推式写法的最大公因数，计算 $\gcd(a, b)$。
- `lli gcd(lli a, lli b)`：递归式写法的最大公因数，计算 $\gcd(a, b)$。
- `lli lcm(lli a, lli b)`：计算 a, b 的最小公倍数。
- `lli exgcd(lli a, lli b, lli& x, lli &y)`：求解同余方程 $ax + by = gcd(a, b)$，将方程结果存在 `x` 和 `y` 内，调用时传入引用。

```
lli gcd_iterative(lli a, lli b)
{
    if (a < b)
        swap(a, b);
    while (b != 0) {
        lli c = a % b;
        a = b;
        b = c;
    }
    return a;
}

lli gcd(lli a, lli b)
{
    return b == 0 ? a : gcd(b, a % b);
}

lli lcm(lli a, lli b)
{
    return a / gcd(a, b) * b;
}

lli exgcd(lli a, lli b, lli& x, lli& y)
{
    if (b == 0) {
        x = 1, y = 0;
        return a;
    }
    int q = exgcd(b, a % b, y, x);
    y -= lli(a / b) * x;
    return q;
}
```

快速幂

- 题目：bzoj1008, *bzoj3142*
- 依赖：

模板描述

给定整数 a, b，计算 a^b。

复杂度

- Space:
  - Worst Case: $O(1)$
  - Amortized: $O(1)$
  - Best Case: $O(1)$
- Time:
  - Worst Case: $O(\log b)$
  - Amortized: $O(\log b)$

- Best Case: $O(\log b)$

算法简述

$$a^b = \begin{cases} a \cdot a^{b-1} & b \equiv 1 \bmod 2 \\ (a^{\lfloor \frac{b}{2} \rfloor})^2 & b \equiv 0 \bmod 2 \end{cases}$$

用递推方式书写保证运算次数恰为 $\log b$ 轮。

快速幂函数 $fastpow(a,b)$ 可以被拓展成以下形式：$f(a,b,\circ_1,\circ_2)$，其中 $a \circ_1 b \equiv a+b$，$a \circ_2 b = a^b$。一般地，运算符 $\circ_1,\circ_2$ 满足 $\langle \mathbb{Z},\circ_1 \rangle$ 和 $\langle \mathbb{Z},\circ_2 \rangle$ 均为半群，且 $\underbrace{a \circ_1 a \circ_1 \ldots \circ_1 a}_{b\text{个}a} = a \circ_2 b$。为防止溢出，我们有时使用快速幂的一个变种“快速乘”。注意使用快速乘保证不溢出的快速幂时间复杂度增长到了 $O((\log b)^2)$。

另一种快速乘使用了 `long double` 保证精度，在确保其精度正确性的前提下可代替非模意义下的快速乘。经过粗略检验该方法的出错率应当小于 10^{-8}。

调用方法

- `lli fastmul_old(lli a, lli b, lli m)`：用快速幂写法的 $O(\log b)$ 快速乘，计算 $a \cdot b \bmod m$。
- `lli fastmul(lli a, lli b, lli m)`：利用 `long double` 作为中介运算器的快速乘，计算 $a \cdot b \bmod m$。
- `lli fastpow(lli a, lli b, lli m)`：快速幂，计算 $a^b \bmod m$。
- `lli fastpow_safe(lli a, lli b, lli m)`：适用于模数 $m \geq 2^{31}-1$ 时，为防止溢出，调用了安全的快速乘的快速幂，同样计算 $a^b \bmod m$。

```
lli fastmul_old(lli a, lli b, lli m)
{
    lli res = 0, tmp = a;
    while (b > 0) {
        if (b & 1)
            res = (res + tmp) % m;
        tmp = (tmp + tmp) % m;
        b >>= 1;
    }
    return res;
}

lli fastmul(lli a, lli b, lli m)
{
    a %= m, b %= m;
    return ((a * b - (lli)((long double)a / m * b + 1.0e-8) * m) + m
) % m;
}

lli fastpow(lli a, lli b, lli m)
{
    lli res = 1, tmp = a;
    while (b > 0) {
```

```
        if (b & 1)
            res = res * tmp % m;
        tmp = tmp * tmp % m;
        b >>= 1;
    }
    return res;
}

lli fastpow_safe(lli a, lli b, lli m)
{
    lli res = 1, tmp = a;
    while (b > 0) {
        if (b & 1)
            res = fastmul(res, tmp, m);
        tmp = fastmul(tmp, tmp, m);
        b >>= 1;
    }
    return res;
}
```

Floyd-Warshall 最短路

- 题目：hdu1690
- 依赖：

模板描述

给定 n 个点的有向图 $G = (V, E)$ 以及每条边的长度，求 V 中任意两点之间的最短距离。

复杂度

- Space:
  - Worst Case: $O(n^3)$
  - Amortized: $O(n^3)$
  - Best Case: $O(n^3)$
- Time:
  - Worst Case: $O(n^2)$
  - Amortized: $O(n^2)$
  - Best Case: $O(n^2)$

算法简述

松弛操作：对于边 $e(i,j) \in E$，$dist_{s,j} := min(dist_{s,j}, dist_{s,i} + len_e)$

用边 $(i,k),(k,j)$ 松弛边 (i,j)，循环方式从外到内分别为 k,i,j。

事实上无论何种循环方式，只要执行三次 Floyd 即可保证结果正确。

调用方法

- `int maxn`：点的数量上限。
- `lli infinit`：规定的无限远距离。
- `lli[][] dist`：点 i 到点 j 的最短路径长度。
- `void add_edge(int u, int v, lli len)`：添加一条 $u \to v$，长度为 len 的有向边。
- `void eval()`：计算 n 个点中任意两个点间的最短路径长度。
- `void init(int n)`：初始化点数为 n 的图。

```
class Floyd
{
public:
    lli dist[maxn][maxn];
    int n;
    void add_edge(int u, int v, lli len)
    {
        dist[u][v] = len;
        return ;
    }
    void eval(void)
    {
        rep(k, 1, n)
            rep(i, 1, n)
                rep(j, 1, n)
                    if (dist[i][k] + dist[k][j] < dist[i][j])
                        dist[i][j] = dist[i][k] + dist[k][j];
        return ;
    }
    void init(int n)
    {
        this->n = n;
        rep(i, 1, n)
            rep(j, 1, n)
                dist[i][j] = i == j ? 0 : infinit;
        return ;
    }
} graph;
```

Floyd-Warshall 传递闭包

- 题目：poj3660
- 依赖：

模板描述

给定 n 个点的有向图 $G = (V, E)$，求 V 上任意两点之间的连通性。

复杂度

- Space:

  - Worst Case: $O(n^3)$
  - Amortized: $O(n^3)$
  - Best Case: $O(n^3)$
- Time:
  - Worst Case: $O(n^2)$
  - Amortized: $O(n^2)$
  - Best Case: $O(n^2)$

算法简述

松弛操作：对于边 $e(i,j) \in E$，$conn_{s,j} := conn_{s,j} \vee (conn_{s,i} \wedge conn_{i,j})$

用边 $(i,k),(k,j)$ 松弛边 (i,j)，循环方式从外到内分别为 k,i,j。

调用方法

- `int maxn`：点的数量上限。
- `bool[][] conn`：点 i 到点 j 是否存在路径。
- `void add_edge(int u, int v)`：添加一条 $u \to v$ 的有向边。
- `void eval()`：计算 n 个点中任意两个点间的连通性。
- `void init(int n)`：初始化点数为 n 的图。

```
class FloydClosure
{
public:
    bool conn[maxn][maxn];
    int n;
    void add_edge(int u, int v)
    {
        conn[u][v] = true;
        return ;
    }
    void eval(void)
    {
        rep(k, 1, n)
            rep(i, 1, n)
                rep(j, 1, n)
                    conn[i][j] |= conn[i][k] && conn[k][j];
        return ;
    }
    void init(int n)
    {
        this->n = n;
        rep(i, 1, n)
            rep(j, 1, n)
                conn[i][j] = false;
        return ;
    }
} graph;
```

HLPP 最大流

- 题目：luogu4722
- 依赖：

模板描述

给定 n 个点 m 条边的有向图 $G = (V, E)$，每条边的权值（即流量），给定源点 s 和汇点 t，求从点 s 到点 t 的最大流量。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:
  - Worst Case: $O(n^2\sqrt{m})$
  - Amortized: $O(n^2\sqrt{m})$
  - Best Case: $O(n + m)$

算法简述

HLPP，Highest Label Preflow Push，是预流推进算法的改进版。所谓预流推进就是不停从有剩余流量的点往外推出流量，直到没有任何点可以继续往外推出流量为止。HLPP 是 Tarjan 得到的一个结论，每次推进标号 $level_i$ 最高的点可以保证时间复杂度下降到 $O(n^2\sqrt{m})$。

在极端数据上，HLPP 的速度比普通 ISAP 要快 3 倍以上。用 `vector<edge>` 的写法在事实上比 `edge*` 的链式前向星写法还要快 5 倍。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：有向边的数量上限，使用 `vector` 写法的不需要此常量。
- `lli infinit`：规定的无限大流量。
- `void add_edge(int u, int v, lli flow)`：添加一条 $u \to v$，流量为 $flow$ 的有向边。
- `lli eval()`：计算该有向图的最大流。
- `void init(int n, int s, int t)`：初始化点数为 n，源点为 s，汇点为 t 的图。

```
class HLPPPointer
{
public:
    struct edge
    {
        int u, v;
        lli flow;
```

```
    edge *next, *rev;
} epool[maxm], *edges[maxn];
int n, s, t, ecnt;
int hlevel, level[maxn], cntl[maxn], wcounter;
deque<int> lst[maxn];
vector<int> gap[maxn];
lli dist[maxn];
void add_edge(int u, int v, lli flow, lli rflow = 0)
{
    edge *p = &epool[++ecnt],
         *q = &epool[++ecnt];
    p->u = u; p->v = v; p->flow = flow;
    p->next = edges[u]; edges[u] = p;
    q->u = v; q->v = u; q->flow = rflow;
    q->next = edges[v]; edges[v] = q;
    p->rev = q; q->rev = p;
    return ;
}
void update_height(int p, int nh)
{
    wcounter += 1;
    if (level[p] != n + 2)
        cntl[level[p]] -= 1;
    level[p] = nh;
    if (nh == n + 2)
        return ;
    cntl[nh] += 1;
    hlevel = nh;
    gap[nh].push_back(p);
    if (dist[p] > 0)
        lst[nh].push_back(p);
    return ;
}
void relabel(void)
{
    memclr(cntl, n);
    rep(i, 0, n)
        level[i] = n + 2;
    rep(i, 0, hlevel) {
        lst[i].clear();
        gap[i].clear();
    }
    wcounter = 0;
    queue<int> que;
    level[t] = 0;
    que.push(t);
    while (!que.empty()) {
        int p = que.front();
        que.pop();
        for (edge *ep = edges[p]; ep; ep = ep->next)
            if (level[ep->v] == n + 2 && ep->rev->flow > 0) {
                que.push(ep->v);
                update_height(ep->v, level[p] + 1);
            }
```

```
            hlevel = level[p];
        }
        return ;
    }
    void push(int p, edge *ep)
    {
        if (dist[ep->v] == 0)
            lst[level[ep->v]].push_back(ep->v);
        lli flow = min(dist[p], ep->flow);
        ep->flow -= flow;
        ep->rev->flow += flow;
        dist[p] -= flow;
        dist[ep->v] += flow;
        return ;
    }
    void discharge(int p)
    {
        int nh = n + 2;
        for (edge *ep = edges[p]; ep; ep = ep->next)
            if (ep->flow > 0) {
                if (level[p] == level[ep->v] + 1) {
                    push(p, ep);
                    if (dist[p] <= 0)
                        return ;
                } else {
                    nh = min(nh, level[ep->v] + 1);
                }
            }
        if (cntl[level[p]] > 1) {
            update_height(p, nh);
        } else {
            rep(i, level[p], n + 2 - 1) {
                for (auto j : gap[i])
                    update_height(j, n + 2);
                gap[i].clear();
            }
        }
        return ;
    }
    lli eval(void)
    {
        memclr(dist, n);
        dist[s] = infinit;
        dist[t] = - infinit;
        hlevel = 0;
        relabel();
        for (edge *ep = edges[s]; ep; ep = ep->next)
            push(s, ep);
        for (; hlevel >= 0; hlevel--)
            while (!lst[hlevel].empty()) {
                int p = lst[hlevel].back();
                lst[hlevel].pop_back();
                discharge(p);
                if (wcounter > 4 * n)
```

```
                    relabel();
                }
            return dist[t] + infinit;
        }
        void init(int n, int s, int t)
        {
            this->n = n;
            this->s = s;
            this->t = t;
            ecnt = 0;
            memclr(edges, n);
            return ;
        }
    } graph;

    class HLPPVeryFast
    {
    public:
        struct edge
        {
            int v, rev;
            lli flow;
            edge(int _1, int _2, lli _3) {
                v = _1, rev = _2, flow = _3; }
        };
        vector<edge> edges[maxn];
        int n, s, t, ecnt;
        int hlevel, level[maxn], cntl[maxn], wcounter;
        deque<int> lst[maxn];
        vector<int> gap[maxn];
        lli dist[maxn];
        void add_edge(int u, int v, lli flow, lli rflow = 0)
        {
            edges[u].push_back(edge(v, edges[v].size(), flow));
            edges[v].push_back(edge(u, edges[u].size() - 1, rflow));
            return ;
        }
        void update_height(int p, int nh)
        {
            wcounter += 1;
            if (level[p] != n + 2)
                cntl[level[p]] -= 1;
            level[p] = nh;
            if (nh == n + 2)
                return ;
            cntl[nh] += 1;
            hlevel = nh;
            gap[nh].push_back(p);
            if (dist[p] > 0)
                lst[nh].push_back(p);
            return ;
        }
        void relabel(void)
        {
```

```
        memclr(cntl, n);
        rep(i, 0, n)
            level[i] = n + 2;
        rep(i, 0, hlevel) {
            lst[i].clear();
            gap[i].clear();
        }
        wcounter = 0;
        queue<int> que;
        level[t] = 0;
        que.push(t);
        while (!que.empty()) {
            int p = que.front();
            que.pop();
            for (edge &ep : edges[p])
                if (level[ep.v] == n + 2 && edges[ep.v][ep.rev].flow
> 0) {
                    que.push(ep.v);
                    update_height(ep.v, level[p] + 1);
                }
            hlevel = level[p];
        }
        return ;
    }
    void push(int p, edge& ep)
    {
        if (dist[ep.v] == 0)
            lst[level[ep.v]].push_back(ep.v);
        lli flow = min(dist[p], ep.flow);
        ep.flow -= flow;
        edges[ep.v][ep.rev].flow += flow;
        dist[p] -= flow;
        dist[ep.v] += flow;
        return ;
    }
    void discharge(int p)
    {
        int nh = n + 2;
        for (edge& ep : edges[p])
            if (ep.flow > 0) {
                if (level[p] == level[ep.v] + 1) {
                    push(p, ep);
                    if (dist[p] <= 0)
                        return ;
                } else {
                    nh = min(nh, level[ep.v] + 1);
                }
            }
        if (cntl[level[p]] > 1) {
            update_height(p, nh);
        } else {
            rep(i, level[p], n + 2 - 1) {
                for (auto j : gap[i])
                    update_height(j, n + 2);
```

```
                gap[i].clear();
            }
        }
        return ;
    }
    lli eval(void)
    {
        memclr(dist, n);
        dist[s] = infinit;
        dist[t] = - infinit;
        hlevel = 0;
        relabel();
        for (edge& ep : edges[s])
            push(s, ep);
        for (; hlevel >= 0; hlevel--)
            while (!lst[hlevel].empty()) {
                int p = lst[hlevel].back();
                lst[hlevel].pop_back();
                discharge(p);
                if (wcounter > 4 * n)
                    relabel();
            }
        return dist[t] + infinit;
    }
    void init(int n, int s, int t)
    {
        this->n = n;
        this->s = s;
        this->t = t;
        ecnt = 0;
        rep(i, 1, n)
            edges[i].clear();
        return ;
    }
} graph;
```

匈牙利算法

- 题目：poj3041
- 依赖：

模板描述

给定 n 个点 m 条边的二分图 $G = (V, E)$，求二分图的最大匹配数。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:

- Worst Case: $O(nm)$
- Amortized: $O(nm)$
- Best Case: $O(n+m)$

算法简述

二分图 $G=(V,E)$ 的点集 V 可以分割成互补的两个点集 V_1,V_2，且两个点集内分别不存在点集内连边，即 $\forall(u,v)\in E,u\in V_1,u\in V_2$。匈牙利算法试图去匹配任意一个 V_2 内的点，如果该点没有被另一个 V_1 内的点匹配，直接创建点对；反之，记原来已经匹配好的点对 $(u',v')s.t.u'\in V_1,v'\in V_2$，此时可以试图给 u' 再分配一个 V_2 中点 v''。

换言之，匈牙利算法本质上就是询问一个点对是否能找到其他可行匹配，可行则贪心匹配一对即可。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：有向边的数量上限。
- `void add_edge(int u, int v)`：添加一条 $u\to v$ 的有向边。
- `lli eval()`：计算该二分图的匹配数。
- `void init(int n)`：初始化点数为 n 的图。

```
class HungaryMatch
{
public:
    struct edge
    {
        int u, v;
        edge *next;
    };
    edge epool[maxm], *edges[maxn];
    int n, ecnt, from[maxn], vis[maxn];
    void add_edge(int u, int v)
    {
        edge *p = &epool[++ecnt];
        p->u = u; p->v = v;
        p->next = edges[u]; edges[u] = p;
        return ;
    }
    bool find(int p)
    {
        for (edge *ep = edges[p]; ep; ep = ep->next)
            if (!vis[ep->v]) {
                vis[ep->v] = true;
                if (!from[ep->v] || find(from[ep->v])) {
                    from[ep->v] = p;
                    return true;
                }
            }
        return false;
```

```
    }
    int eval(void)
    {
        int res = 0;
        rep(i, 1, n) {
            memclr(vis, n);
            if (find(i))
                res += 1;
        }
        return res;
    }
    void init(int n)
    {
        this->n = n;
        ecnt = 0;
        memclr(edges, n);
        return ;
    }
} graph;
```

ISAP 最大流

- 题目：hdu3549
- 依赖：

模板描述

给定 n 个点 m 条边的有向图 $G = (V, E)$，每条边的权值（即流量），给定源点 s 和汇点 t，求从点 s 到点 t 的最大流量。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:
  - Worst Case: $O(n^2 m)$
  - Amortized: $O(n^2 m)$
  - Best Case: $O(n + m)$

算法简述

ISAP，Improved Shortest Augmenting Path，从汇点开始反向 bfs 建立层次图，但是每次 dfs 的时候每个点的层次随着算法进行不断提高，这样就不用多次 bfs 重建层次图了。当 s 的深度超过 n，则增广完毕，结束算法。

一般地，优化 ISAP 速度能比朴素 Dinic 快 1 倍左右。此外，虽然 HLPP 拥有较优的渐进时间复杂度 $O(n^2\sqrt{m})$，但是常数较大，在普通随机数据上表现也不如 ISAP。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：有向边的数量上限。
- `lli infinit`：规定的无限大流量。
- `void add_edge(int u, int v, lli flow)`：添加一条 $u \to v$，流量为 $flow$ 的有向边。
- `lli eval()`：计算该有向图的最大流。
- `void init(int n, int s, int t)`：初始化点数为 n，源点为 s，汇点为 t 的图。

```
class ISAP
{
public:
    struct edge
    {
        int u, v;
        lli flow;
        edge *next, *rev;
    } epool[maxm], *edges[maxn], *cedge[maxn];
    int n, s, t, ecnt, level[maxn], gap[maxn];
    lli maxflow;
    void add_edge(int u, int v, lli flow, lli rflow = 0)
    {
        edge *p = &epool[++ecnt],
             *q = &epool[++ecnt];
        p->u = u; p->v = v; p->flow = flow;
        p->next = edges[u]; edges[u] = p;
        q->u = v; q->v = u; q->flow = rflow;
        q->next = edges[v]; edges[v] = q;
        p->rev = q; q->rev = p;
        return ;
    }
    void make_level(void)
    {
        memclr(level, n);
        memclr(gap, n);
        queue<int> que;
        level[t] = 1;
        gap[level[t]] += 1;
        que.push(t);
        while (!que.empty()) {
            int p = que.front();
            que.pop();
            for (edge *ep = edges[p]; ep; ep = ep->next)
                if (!level[ep->v]) {
                    level[ep->v] = level[p] + 1;
                    gap[level[ep->v]] += 1;
                    que.push(ep->v);
                }
        }
        return ;
```

```
    }
    lli find(int p, lli flow)
    {
        if (p == t)
            return flow;
        lli used = 0;
        for (edge *ep = cedge[p]; ep; ep = ep->next)
            if (ep->flow > 0 && level[ep->v] + 1 == level[p]) {
                lli tmp = find(ep->v, min(ep->flow, flow - used));
                if (tmp > 0) {
                    ep->flow -= tmp;
                    ep->rev->flow += tmp;
                    used += tmp;
                    cedge[p] = ep;
                }
                if (used == flow)
                    return used;
            }
        gap[level[p]] -= 1;
        if (gap[level[p]] == 0)
            level[s] = n + 1;
        level[p] += 1;
        gap[level[p]] += 1;
        cedge[p] = edges[p];
        return used;
    }
    lli eval(void)
    {
        lli res = 0;
        make_level();
        rep(i, 1, n)
            cedge[i] = edges[i];
        while (level[s] <= n)
            res += find(s, infinit);
        return res;
    }
    void init(int n, int s, int t)
    {
        this->n = n;
        this->s = s;
        this->t = t;
        ecnt = 0;
        memclr(edges, n);
        return ;
    }
} graph;
```

Knuth-Morris-Pratt 字符串匹配

- 题目：hdu1686, hdu2087
- 依赖：

模板描述

给定模板串 str 和模式串 $patt$，求 str 有多少个（相交或互不相交）子串与 $patt$ 相等。

记 $|str| = n, |patt| = m$。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$

算法简述

记 $next_i$ 为前 i 个字符中最长的长度 k，使得 $patt[1, k] = patt[i - k + 1, n]$。由于 $next$ 数组可以在 $O(m)$ 时间内用递推法求出，且可以辅助跳过已经匹配过的后缀（前缀符合，所以下一个符合匹配条件的就直接跳到 $next_j$ 即可。

本模板内数组下标从 0 开始。

调用方法

- `int maxn`：字符串长度上限。
- `int[] match(bool overlap)`：用模式串 $patt$ 匹配模板串 str，参数 $overlap$ 决定匹配时是否允许匹配结果相互交叉。
- `void init_patt(int m, char[] patt)`：加载模式串。
- `void init_str(int n, char[] str)`：加载模板串。

```
class KMP
{
public:
    int m, patt[maxn], nxt[maxn];
    int n, str[maxn];
    void get_next(void)
    {
        int i = 0, j = -1;
        nxt[0] = -1;
        for (int i = 0, j = -1; i < m; ) {
            if (j == -1 || patt[i] == patt[j])
                i += 1, j += 1, nxt[i] = j;
            else
                j = nxt[j];
        }
        return ;
    }
```

```
    vector<int> match(bool overlap = true)
    {
        vector<int> res;
        for (int i = 0, j = 0; i < n; ) {
            if (j == -1 || str[i] == patt[j])
                i += 1, j += 1;
            else
                j = nxt[j];
            if (j == m)
                res.push_back(i), j = overlap ? nxt[j] : 0;
        }
        return res;
    }
    void init_patt(int m, char patt[])
    {
        this->m = m;
        rep(i, 0, m - 1)
            this->patt[i] = patt[i];
        memclr(nxt, m);
        get_next();
        return ;
    }
    void init_str(int n, char str[])
    {
        this->n = n;
        rep(i, 0, n - 1)
            this->str[i] = str[i];
        return ;
    }
} kmp;
```

Kruskal 最小生成树

- 题目：hdu3371
- 依赖：disjoint_set

模板描述

给定 n 个点 m 条边的无向图 $G = (V, E)$，求图 G 的最小生成树 $G' = (V, E' \subseteq E)$，且 $\sum_{e \in E'} |e|$ 最小。

复杂度

- Space:
  - Worst Case: $O(m)$
  - Amortized: $O(m)$
  - Best Case: $O(m)$
- Time:
  - Worst Case: $O(m\log m)$
  - Amortized: $O(m\log m)$
  - Best Case: $O(n\log m)$

算法简述

我们可以贪心地选择最小的边，将它们连起来，如果它们本来不在一个连通块上的话。理由如下：倘若存在边 $e_1 = (u, v), e_2 = (v, w), e_3 = (u, w)$，且 $|e_1| < |e_2| < |e_3|$，则同样可以连起来的方法中总是以 e_1 和 e_2 优先。若我们舍弃了一个较短的边，则在达成条件的前提下总是比选择短边的方案要差。

如此我们用一个 $O(m\log m)$ 的排序来贪心，用并查集维护点联通，若存在事先已连接好的点则将其在并查集上预处理，即可。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：无向边的数量上限。
- `void add_edge(int u, int v, lli len)`：添加一条 $u \leftrightarrow v$，长度为 len 的无向边。
- `void join(int u, int v)`：预先连接点 u 和点 v。
- `lli eval()`：计算图 G 的最小生成树的边权之和，在代码中对应位置修改可求得该生成树的构造。
- `void init(int n)`：初始化点数为 n 的图。

```
class Kruskal
{
public:
    struct edge
    {
        int u, v;
        lli len;
        bool operator < (const edge& b) const
        {
            return this->len < b.len;
        }
    } edges[maxm];
    int n, m, mst_cnt;
    void add_edge(int u, int v, lli len)
    {
        edge *ep = &edges[++m];
        ep->u = u; ep->v = v; ep->len = len;
        return ;
    }
    void join(int u, int v)
    {
        if (!dsu.joined(u, v)) {
            dsu.join(u, v);
            mst_cnt += 1;
        }
        return ;
    }
    lli eval(void)
    {
        lli min_span = 0;
```

```
        sort(edges + 1, edges + m + 1);
        rep(i, 1, m) {
            if (mst_cnt >= n - 1)
                break;
            int u = edges[i].u, v = edges[i].v, len = edges[i].len;
            if (dsu.joined(u, v))
                continue;
            dsu.join(u, v);
            // printf("add_edge %d -> %d : %lld\n", u, v, len);
            min_span += len;
            mst_cnt += 1;
        }
        if (mst_cnt < n - 1)
            min_span = -1;
        return min_span;
    }
    void init(int n)
    {
        this->n = n;
        m = 0;
        mst_cnt = 0;
        dsu.init(n);
        return ;
    }
} graph;
```

线性筛

- 题目：
- 依赖：

模板描述

计算 $[1,n]$ 内的所有正整数的 $\varphi(i)$，以及筛出该区间内的质数。

复杂度

- Space:
  - Worst Case: $O(n)$
  - Amortized: $O(n)$
  - Best Case: $O(n)$
- Time:
  - Worst Case: $O(n)$
  - Amortized: $O(n)$
  - Best Case: $O(n)$

算法简述

在朴素的 $O(n^2)$ 筛法上优化成为了 $O(n)$ 的欧拉筛，使每个数不会被筛两次，且保证所有合数被消除。

利用 $\varphi(x)$ 的积性函数性质可以在筛质数的过程中计算 φ 函数。

调用方法

- `int maxn`：筛选的质数的数量上限。
- `int maxp`：$[1,n]$ 内质数的数量上限。
- `void filter(int n)`：筛选出 $[1,n]$ 范围内的质数及其 $\varphi(i)$ 值。
- `bool[] isprime`：长度为 $maxn$ 的数组，代表该位置是否为质数。
- `int[] primes`：长度为 $maxp$ 的数组，$primes[0]$ 代表一共有多少质数，$primes[i]$ 为该范围内第 i 个质数。$[1,n]$ 内质数的个数满足：

$$primes[0]=\begin{cases}1229 & n=10^4\\ 9592 & n=10^5\\ 78498 & n=10^6\\ 664579 & n=10^7\\ 5761455 & n=10^8\end{cases}$$

- `int[] phi`：长度为 $maxn$ 的数组，满足 $phi[i]=\varphi(i)$。

```
bool isprime[maxn];
int primes[maxp];
int phi[maxn];

void filter(int n)
{
    rep(i, 2, n)
        isprime[i] = true;
    isprime[1] = false;
    primes[0] = 0;
    phi[1] = 1;
    for (int i = 2; i <= n; i++) {
        if (isprime[i]) {
            primes[++primes[0]] = i;
            phi[i] = i - 1;
        }
        for (int j = 1; j <= primes[0] && i * primes[j] <= n; j++) {
            isprime[i * primes[j]] = false;
            if (i % primes[j] == 0) {
                phi[i * primes[j]] = phi[i] * primes[j];
                break;
            }
            phi[i * primes[j]] = phi[i] * (primes[j] - 1);
        }
    }
    return ;
}
```

矩阵

- 题目：
- 依赖：

模板描述

给定矩阵 A, B，对矩阵进行运算。

复杂度

- Space:
  - Worst Case: $O(n^2)$
  - Amortized: $O(n^2)$
  - Best Case: $O(n^2)$
- Time:
  - Worst Case: $O(n^2)$
  - Amortized: $O(n^2)$
  - Best Case: $O(n^2)$
- Multiplication Time:
  - Worst Case: $O(n^3)$
  - Amortized: $O(n^3)$
  - Best Case: $O(n^3)$
- Determinant Time:
  - Worst Case: $O(n^3)$
  - Amortized: $O(n^3)$
  - Best Case: $O(n^3)$
- Inversion Time:
  - Worst Case: $O(n^5)$
  - Amortized: $O(n^5)$
  - Best Case: $O(n^5)$

算法简述

矩阵求行列式值方法：逐行消掉元素，仅保留上三角矩阵，记录扩倍的值，最后将主对角线上元素相乘除掉消掉元素过程中乘上的值。

矩阵求行列式的定义法：$|A| = \sum_{p_1,p_2,\dots,p_n} (-1)^{a(p_1,p_2,\dots,p_n)} \cdot a_{1,p_1} \cdot a_{2,p_2} \cdots a_{n,p_n}$，递归求或递推求均可。

矩阵求逆的方法：$A^{-1} = \frac{A^*}{|A|}$，其中 A^* 为 A 的增广矩阵，$A^*_{j,i}$ 为 A 去除第 i 行第 j 列对应的代数余子式。

调用方法

- `typedef typ`：矩阵内元素的类型。
- `int maxn`：矩阵的长宽上限。

- `typ operator () (int i, int j)`：寻址第 i 行第 j 列。
- `matrix eye(int n)`：生成 $n \times n$ 的单位矩阵。
- `matrix zeros(int n, int m)`：生成 $n \times m$ 的零矩阵。
- `matrix ones(int n, int m)`：生成 $n \times m$ 的各位均为 1 的矩阵。
- `matrix A + matrix B = matrix`：求 $A + B$。
- `matrix A - matrix B = matrix`：求 $A - B$。
- `matrix A += matrix B`：求值 $A := A + B$。
- `matrix A -= matrix B`：求值 $A := A - B$。
- `matrix prod(matrix A, matrix B)`：若矩阵 B 维度与 A 相同，则求两个矩阵对应位置相乘得到的新矩阵 $A .^* B$；若 B 为列向量，则 A 的每一列与 B 对应元素两两相乘；若 B 为行向量，则 A 的每一行与 B 对应元素两两相乘；若 B 为 1×1 矩阵，则 A 每个元素均与 $B_{1,1}$ 相乘。
- `matrix prod(matrix A, typ b)`：求矩阵 A 和常数 b 的积 $b \cdot A$。
- `matrix A * matrix B = matrix`：求两个矩阵相乘 $A \times B$。
- `matrix A * typ b = matrix`：求矩阵 A 和常数 b 的积 $b \cdot A$。
- `typ a * matrix B = matrix`：求常数 a 和矩阵 $$B 的积 $a \cdot B$。
- `matrix A *= matrix B`：求值 $A := A \times B$。
- `matrix A *= typ b`：求值 $A := b \cdot A$。
- `matrix pow(matrix A, lli b)`：求 A^b。
- `matrix transpose(matrix A)`：求矩阵 A 的转置 A^T。
- `matrix det_def(matrix A)`：用定义法（$O(n^{n+1})$）求矩阵 A 的行列式 $|A|$。
- `matrix det(matrix A)`：用高斯消元法求矩阵 A 的行列式 $|A|$。
- `matrix inv(matrix A)`：求矩阵 A 的逆。

```
typedef llf typ;

struct matrix
{
protected:
    int n, m;
    typ arr[maxn][maxn];
public:
    matrix(void)
    {
        n = m = 1;
        arr[0][0] = 0;
    }
    matrix(int n, int m)
    {
        this->n = n, this->m = m;
        rep(i, 0, n - 1)
            rep(j, 0, m - 1)
                arr[i][j] = 0;
    }
    typ& operator () (int i, int j)
    {
```

```
        if (i < 1 || i > n || j < 1 || j > m)
            throw std::out_of_range("invalid indices");
        return arr[i - 1][j - 1];
    }
    matrix operator () (int top, int bottom, int left, int right);
    friend matrix eye(int);
    friend matrix zeros(int, int);
    friend matrix ones(int, int);
    friend matrix operator + (const matrix&, const matrix&);
    friend matrix operator - (const matrix&, const matrix&);
    matrix& operator += (const matrix&);
    matrix& operator -= (const matrix&);
    friend matrix prod(const matrix&, const matrix&);
    friend matrix prod(const matrix&, const typ);
    friend matrix operator * (const matrix&, const matrix&);
    friend matrix operator * (const matrix&, const typ);
    friend matrix operator * (const typ, const matrix&);
    matrix& operator *= (const matrix&);
    matrix& operator *= (const typ);
    friend matrix pow(const matrix& a, lli b);
    friend matrix transpose(const matrix&);
    friend typ det_def(const matrix&);
    friend typ det(const matrix&);
    friend matrix inv(const matrix&);
    friend ostream& operator << (ostream&, const matrix&);
};

matrix matrix::operator () (int top, int bottom, int left, int right
)
{
    if (top < 1 || top > n || bottom < 1 || bottom > n || top > bott
om ||
        left < 1 || left > m || right < 1 || right > m || left > rig
ht)
        throw std::out_of_range("invalid indices");
    int h = bottom - top + 1, w = right - left + 1;
    matrix mat(h, w);
    rep(i, 0, h - 1)
        rep(j, 0, w - 1)
            mat.arr[i][j] = arr[top - 1 + i][left - 1 + j];
    return std::move(mat);
}

matrix eye(int n)
{
    matrix mat(n, n);
    rep(i, 0, n - 1)
        mat.arr[i][i] = 1;
    return std::move(mat);
}

matrix zeros(int n, int m = -1)
{
```

```
    if (m == -1)
        m = n;
    matrix mat(n, m);
    return std::move(mat);
}

matrix ones(int n, int m = -1)
{
    if (m == -1)
        m = n;
    matrix mat(n, m);
    rep(i, 0, n - 1)
        rep(j, 0, m - 1)
            mat.arr[i][j] = 1;
    return std::move(mat);
}

matrix operator + (const matrix& a, const matrix& b)
{
    matrix c(a.n, a.m);
    if (b.n != c.n || b.m != c.m)
        throw std::domain_error("nonconformant arguments");
    rep(i, 0, c.n - 1)
        rep(j, 0, c.m - 1)
            c.arr[i][j] = a.arr[i][j] + b.arr[i][j];
    return std::move(c);
}

matrix operator - (const matrix& a, const matrix& b)
{
    matrix c(a.n, a.m);
    if (b.n != c.n || b.m != c.m)
        throw std::domain_error("nonconformant arguments");
    rep(i, 0, c.n - 1)
        rep(j, 0, c.m - 1)
            c.arr[i][j] = a.arr[i][j] - b.arr[i][j];
    return std::move(c);
}

matrix& matrix::operator += (const matrix& b)
{
    if (b.n != n || b.m != m)
        throw std::domain_error("nonconformant arguments");
    rep(i, 0, n - 1)
        rep(j, 0, m - 1)
            arr[i][j] = arr[i][j] + b.arr[i][j];
    return *this;
}

matrix& matrix::operator -= (const matrix& b)
{
    if (b.n != n || b.m != m)
        throw std::domain_error("nonconformant arguments");
```

```
    rep(i, 0, n - 1)
        rep(j, 0, m - 1)
            arr[i][j] = arr[i][j] - b.arr[i][j];
    return *this;
}

matrix prod(const matrix& a, const matrix& b)
{
    matrix c(a.n, a.m);
    if (a.n == b.n && a.m == b.m) {
        rep(i, 0, a.n - 1)
            rep(j, 0, a.m - 1)
                c.arr[i][j] = a.arr[i][j] * b.arr[i][j];
    } else if (a.n == b.n && b.m == 1) {
        rep(i, 0, a.n - 1)
            rep(j, 0, a.m - 1)
                c.arr[i][j] = a.arr[i][j] * b.arr[i][0];
    } else if (a.m == b.m && b.n == 1) {
        rep(i, 0, a.n - 1)
            rep(j, 0, a.m - 1)
                c.arr[i][j] = a.arr[i][j] * b.arr[0][j];
    } else if (b.n == 1 && b.m == 1) {
        rep(i, 0, a.n - 1)
            rep(j, 0, a.m - 1)
                c.arr[i][j] = a.arr[i][j] * b.arr[0][0];
    } else {
        throw std::domain_error("nonconformant arguments");
    }
    return std::move(c);
}

matrix prod(const matrix& a, typ b)
{
    matrix c(a.n, a.m);
    rep(i, 0, a.n - 1)
        rep(j, 0, a.m - 1)
            c.arr[i][j] = a.arr[i][j] * b;
    return std::move(c);
}

matrix operator * (const matrix& a, const matrix& b)
{
    if (a.m != b.n)
        throw std::domain_error("nonconformant arguments");
    matrix c(a.n, b.m);
    rep(i, 0, c.n - 1)
        rep(j, 0, c.m - 1) {
            c.arr[i][j] = 0;
            rep(k, 0, a.m - 1)
                c.arr[i][j] += a.arr[i][k] * b.arr[k][j];
        }
    return std::move(c);
}
```

```
matrix operator * (const matrix& a, const typ b)
{
    return std::move(prod(a, b));
}

matrix operator * (const typ a, const matrix& b)
{
    return std::move(prod(b, a));
}

matrix& matrix::operator *= (const matrix& b)
{
    if (m != b.n)
        throw std::domain_error("nonconformant arguments");
    *this = *this * b;
    return *this;
}

matrix& matrix::operator *= (const typ b)
{
    rep(i, 0, n - 1)
        rep(j, 0, m - 1)
            arr[i][j] = arr[i][j] * b;
    return *this;
}

matrix pow(const matrix& a, lli b)
{
    if (a.n != a.m)
        throw std::domain_error("nonconformant arguments");
    matrix res = eye(a.n), tmp = a;
    while (b > 0) {
        if (b & 1)
            res *= tmp;
        tmp *= tmp;
        b >>= 1;
    }
    return std::move(res);
}

matrix transpose(const matrix& a)
{
    matrix mat(a.m, a.n);
    rep(i, 0, a.n - 1)
        rep(j, 0, a.m - 1)
            mat.arr[j][i] = a.arr[i][j];
    return std::move(mat);
}

typ det_def(const matrix& a)
{
    if (a.n != a.m)
```

```
        throw std::domain_error("nonconformant arguments");
    int n = a.n;
    typ res = 0;
    static int vec[maxn];
    static bool vis[maxn];
    rep(i, 0, n - 1)
        vec[i] = vis[i] = 0;
    stack<pair<int, int>> dfs_stk;  // dfs stack
    rep_(i, n - 1, 0)
        dfs_stk.push(make_pair(1, i + 1));
    while (!dfs_stk.empty()) {
        pair<int, int> pr = dfs_stk.top();
        dfs_stk.pop();
        int depth = pr.first,
            p = abs(pr.second) - 1,
            rm = pr.second < 0;
        if (rm) {  // restore previous state
            vis[p] = false;
            vec[depth - 1] = 0;
            continue;
        } else if (depth == n) {  // final state
            vis[p] = true;
            vec[depth - 1] = p + 1;
            // calculate count inverse
            lli cnt_inv = 0;
            rep(i, 0, n - 1)
                rep(j, i + 1, n - 1)
                    if (vec[i] > vec[j])
                        cnt_inv += 1;
            // (-1) ^ cnt_inv(vec) * sum(arr[i][vec[i]])
            typ tmp = 0;
            tmp = cnt_inv % 2 == 1 ? -1 : 1;
            rep(i, 0, n - 1)
                tmp *= a.arr[i][vec[i] - 1];
            // sum(tmp)
            res += tmp;
            // restore previous state
            vis[p] = false;
            vec[depth - 1] = 0;
            continue;
        }
        // mark restoration state
        dfs_stk.push(make_pair(depth, - (p + 1)));
        vis[p] = true;
        vec[depth - 1] = p + 1;
        // children states
        rep_(i, n - 1, 0)
            if (!vis[i])
                dfs_stk.push(make_pair(depth + 1, i + 1));
    }
    return res;
}

typ det(const matrix& a)
```

```
{
    if (a.n != a.m)
        throw std::domain_error("nonconformant arguments");
    int n = a.n;
    matrix m(n, n);
    rep(i, 0, n - 1)
        rep(j, 0, n - 1)
            m.arr[i][j] = a.arr[i][j];
    // arrange rows
    typ comp = 1;  // compensate
    rep_(i, n - 1, 0) {
        // find non-zero row to dissipate other values
        int flag = -1;
        rep_(j, i, 0)
            if (m.arr[i][j] != 0) {
                flag = j;
                break;
            }
        if (flag == -1)
            return 0;
        // swap rows
        if (flag != i)
            rep(j, 0, n - 1)
                swap(m.arr[j][flag], m.arr[j][i]);
        // eliminate other rows
        rep(j, 0, i - 1) {
            typ pa = m.arr[i][j],
                pb = m.arr[i][i];
            comp *= pb;
            rep(k, 0, n - 1)
                m.arr[k][j] = m.arr[k][j] * pb - m.arr[k][i] * pa;
        }
    }
    // multiply diagonal elements
    typ res = 1;
    rep(i, 0, n - 1)
        res *= m.arr[i][i];
    // divide by compensate
    res /= comp;
    return res;
}

matrix inv(const matrix& a)
{
    if (a.n != a.m)
        throw std::domain_error("nonconformant arguments");
    int n = a.n;
    matrix b(n, n), tmp(n - 1, n - 1);
    typ det_a = det(a);
    if (det_a == 0)
        throw std::invalid_argument("not inversible");
    rep(i, 0, n - 1)
        rep(j, 0, n - 1) {
            rep(_i, 0, n - 2)
```

```
                rep(_j, 0, n - 2)
                    tmp.arr[_i][_j] = a.arr[_i < i ? _i : (_i + 1)]
                                           [_j < j ? _j : (_j + 1)];
            b.arr[j][i] = ((i + j) % 2 == 1 ? -1 : 1) * det(tmp) / d
et_a;
        }
    return std::move(b);
}

ostream& operator << (ostream& out, const matrix& mat)
{
    rep(i, 0, mat.n - 1) {
        out << "|";
        rep(j, 0, mat.m - 1)
            out << " " << mat.arr[i][j];
        out << " |";
        if (i < mat.n - 1)
            out << "\n";
    }
    return out;
}
```

Miller-Rabin 质数判别法

- 题目：
- 依赖：fast_exponentiation

模板描述

给定质数 n，判定 n 是否为质数。

复杂度

- Space:
  - Worst Case: $O(k)$
  - Amortized: $O(k)$
  - Best Case: $O(k)$
- Time:
  - Worst Case: $O(k\log^2 n)$
  - Amortized: $O(k\log^2 n)$
  - Best Case: $O(k\log^2 n)$

算法简述

Fermat 小定理：若 p 为质数，则必有 $a^{p-1} \equiv 1 \bmod p$。

反之，若有 $a^{p-1} \equiv 1 \bmod p$，则 p 大概率是质数，若 p 为合数定义为 Carmichael 数。

二次探测：如果 p 是一个质数，且 $0 < x < p$，则方程 $x^2 \equiv 1 \bmod p$ 的解为 $x = 1$ 或 $x = p - 1$。

采用多个质数多次对某数 n 进行上述检测，若次数足够多可以确定 n 是否为质数。经过古今中外无数勇士的贡献与检验，得到一组最少的质数表如下：

$$p_i = \begin{cases} 2 & n \le 2.04 \cdot 10^3 \\ 31,73 & n \le 9.08 \cdot 10^6 \\ 2,7,61 & n \le 4.75 \cdot 10^9 \\ 2,13,23,1662803 & n \le 1.12 \cdot 10^{12} \\ 2,3,5,7,11,13,17,19,23,29,31,37 & n \le 3.18 \cdot 10^{23} \end{cases}$$

在 `is_prime.samples` 可修改该质数表，其中 `samples[0]` 代表质数表的大小（上述复杂度分析中的常数 k 也指代质数表的大小。需要注意的是，由于复杂度太高，Miller-Rabin 算法不应用于筛选质数。

调用方法

- `bool miller_rabin_test(lli n, lli p)`：利用质数 p 以一定概率检验 n 是否是质数。
- `bool is_prime(lli n)`：利用事先确定的质数表确定地检验 n 是否为质数。

```
bool miller_rabin_test(lli n, lli k)
{
    if (fastpow(k, n - 1, n) != 1)
        return false;
    lli t = n - 1, tmp;
    while (t % 2 == 0) {
        t >>= 1;
        tmp = fastpow(k, t, n);
        if (tmp != 1 && tmp != n - 1)
            return false;
        if (tmp == n - 1)
            return true;
    }
    return true;
}

bool is_prime(lli n)
{
    if (n == 1 || (n > 2 && n % 2 == 0))
        return false;
    // lli samples[13] = { 12, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 3
1, 37 };
    lli samples[4] = { 4, 2, 7, 61 };
    rep(i, 1, samples[0]) {
        if (n == samples[i])
            return true;
        if (n > samples[i] && !miller_rabin_test(n, samples[i]))
            return false;
    }
```

```
    return true;
}
```

朴素质因数分解

- 题目：
- 依赖：linear_sieve

模板描述

给定整数 n，求 n 的所有质因子。

复杂度

- Space:
  - Worst Case: $O(1)$
  - Amortized: $O(1)$
  - Best Case: $O(1)$
- Time:
  - Worst Case: $O(\frac{\sqrt{n}}{\log\sqrt{n}})$
  - Amortized: $O(\frac{\sqrt{n}}{\log\sqrt{n}})$
  - Best Case: $O(\frac{\sqrt{n}}{\log\sqrt{n}})$

算法简述

整数 n 最多含有一个大于 $\sqrt{n}$ 的质因子，所以仅需暴力判别小于等于 $\sqrt{n}$ 的所有质数即可。

调用该方法前需先用线性筛求出一定范围内的所有质数。

调用方法

- `lli[] factorize(lli n)`：质因数分解 n，将结果（可能重复地）按顺序放入结果中。例如若 $n = 2^3 \cdot 3$，则 $factors = [2,2,2,3]$。

```
vector<lli> factorize(lli n)
{
    lli tmp = n;
    vector<lli> factors;
    rep(i, 1, primes[0]) {
        lli p = primes[i];
        if (p * p > n || tmp <= 1)
            break;
        while (tmp % p == 0) {
            factors.push_back(p);
            tmp /= p;
        }
    }
    if (tmp > 1)
```

```
            factors.push_back(tmp);
        return factors;
}
```

Pollard's Rho 质因数分解

- 题目：
- 依赖：euclid_gcd, fast_exponentiation

模板描述

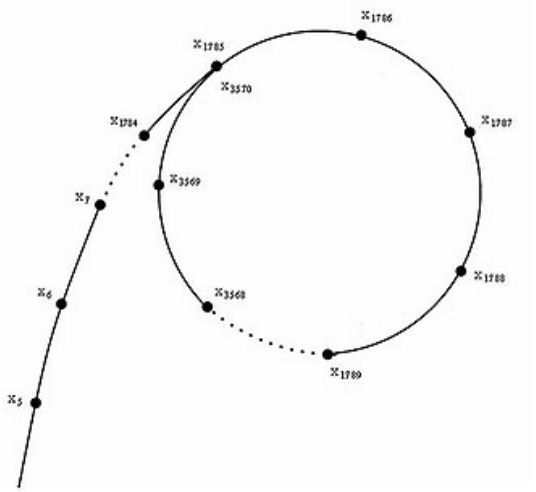
给定整数 n，求 n 的所有质因子。

复杂度

- Space:
  - Worst Case: $O(1)$
  - Amortized: $O(1)$
  - Best Case: $O(1)$
- Time:
  - Worst Case: $O(n)$
  - Amortized: $O(n^{\frac{1}{4}})$
  - Best Case: $O(\sqrt{p})$

算法简述

构造伪随机生成器 $g(x)$，则生成的序列为 x_0，$x_1 = g(x_0)$，$x_2 = g(g(x_0))$，…，$x_n = g(x_{n-1})$。这时候我们发现当 $g(x)$ 的值域有限时它一定会重复，并且最终会成环（这也是为什么算法被称为 ρ 的原因），如下图：



Rho

根据生日悖论，如果共存在 m 种可能的值，则在成环之前出现的值的数量期望为 $\sqrt{m}$ 个。随后我们只需用 Floyd 判环算法找到这个环即可。因此算法的期望

复杂度为 $O(\sqrt{m}) = O(n^{\frac{1}{4}})$，或者若最小因子为 p，其期望复杂度也可记为 $O(\sqrt{p})$。

事实上这个算法在分解大量数字的情况下性能不如借用线性筛的朴素算法好。

调用方法

- `void pollard_rho(lli n, lli[] factors)`：随机化质因数分解 n，将结果无序地放入 $factors$ 中。
- `lli[] pollard_rho(lli n)`：随机化质因数分解 n，有序地返回所有质因子。例如若 $n = 2^3 \cdot 3$，则 $factors = [2,2,2,3]$。

```
void pollard_rho(lli n, vector<lli>& factors)
{
    if (is_prime(n)) {
        factors.push_back(n);
        return ;
    }
    lli a, b, c, d;
    while (true) {
        c = rand() % n;
        a = b = rand() % n;
        b = (fastmul(b, b, n) + c) % n;
        while (a != b) {
            d = a - b;
            d = gcd(abs(d), n);
            if (d > 1 && d < n) {
                pollard_rho(d, factors);
                pollard_rho(n / d, factors);
                return ;
            }
            a = (fastmul(a, a, n) + c) % n;
            b = (fastmul(b, b, n) + c) % n;
            b = (fastmul(b, b, n) + c) % n;
        }
    }
    return ;
}

vector<lli> pollard_rho(lli n)
{
    lli tmp = n;
    vector<lli> factors;
    pollard_rho(n, factors);
    sort(factors.begin(), factors.end());
    return factors;
}
```

Prim 最小生成树

- 题目：hdu1102

- 依赖：

模板描述

给定 n 个点 m 条边的无向图 $G=(V,E)$，求图 G 的最小生成树 $G'=(V,E'\subseteq E)$，且 $\sum_{e\in E'}|e|$ 最小。

复杂度

- Space:
  - Worst Case: $O(n^2)$
  - Amortized: $O(n^2)$
  - Best Case: $O(n^2)$
- Time:
  - Worst Case: $O(n^2)$
  - Amortized: $O(n^2)$
  - Best Case: $O(n^2)$

算法简述

以任意点为起点，维护一个点集 S，初始为 $S=\{1\}$。选择一个点 $p\notin S$ 使得在所有的满足 $a\in S, b\notin S$ 的点对中 p 对应到某一个 $dist_{a,b}$ 最小的点对。随后将 p 加入 S。显然这样的做法是对的，证法类同 Kruskal 算法。

注意 Fibonacci 堆的常数较大，所以邻接表写法的 Fibonacci 堆优化 Prim 的复杂度虽然是 $O(m+n\log n)$ 的，其运行速度将大大不如 $O((n+m)\log n)$ 的二叉堆优化 Prim。进一步地，堆优化 Prim 在稠密图上表现并不明显好于朴素的 Prim，且常数较大。故若非稠密图，尽量应采用 Kruskal 算法。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：无向边的数量上限。
- `lli infinit`：规定的无限远距离。
- `void add_edge(int u, int v, lli len)`：添加一条 $u\leftrightarrow v$，长度为 len 的无向边。
- `void join(int u, int v)`：预先连接点 u 和点 v。
- `lli eval()`：计算图 G 的最小生成树的边权之和，在代码中对应位置修改可求得该生成树的构造。
- `void init(int n)`：初始化点数为 n 的图。

```
class Prim
{
public:
    lli dist[maxn][maxn];
    int n, vis[maxn], min_cost[maxn];
    void add_edge(int u, int v, lli len)
    {
```

```
        dist[u][v] = dist[v][u] = len;
        return ;
    }
    void join(int u, int v)
    {
        add_edge(u, v, 0);
        return ;
    }
    lli eval(void)
    {
        lli min_span = 0;
        rep(i, 1, n) {
            vis[i] = false;
            min_cost[i] = 1;
        }
        vis[1] = true;
        rep(i, 1, n) {
            int p = 0;
            rep(j, 1, n)
                if (!vis[j] && dist[min_cost[j]][j] < dist[min_cost[
p]][p])
                    p = j;
            if (p == 0)
                break;
            min_span += dist[min_cost[p]][p];
            // printf("add_edge %d -> %d : %lld\n", p, min_cost[p],
            //        dist[p][min_cost[p]]);
            vis[p] = true;
            rep(j, 1, n)
                if (dist[p][j] < dist[min_cost[j]][j])
                    min_cost[j] = p;
        }
        return min_span;
    }
    void init(int n)
    {
        this->n = n;
        rep(i, 0, n)
            rep(j, 0, n)
                dist[i][j] = infinit;
        return ;
    }
} graph;
```

Tarjan 强连通分量

- 题目：poj1236, poj2186
- 依赖：

模板描述

给定 n 个点 m 条边的有向图 $G = (V, E)$，求 G 的强连通分量。

强连通分量：该子图内任意两点间总存在一条仅由子图内边构成的路径。

复杂度

- Space:
  - Worst Case: $O(n+m)$
  - Amortized: $O(n+m)$
  - Best Case: $O(n+m)$
- Time:
  - Worst Case: $O(n+m)$
  - Amortized: $O(n+m)$
  - Best Case: $O(n+m)$

算法简述

记数组 dfn_i 为点 i 被 dfs 到的次序编号（时间戳），low_i 为 i 和 i 的子树能够追溯到的最早的堆栈中的节点的时间戳。维护一个堆栈用于储存要处理的强连通分量。具体做法见代码。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：无向边的数量上限。
- `int[] belong`：点 i 属于第 $belong_i$ 个强连通分量。
- `int[] bsize`：第 i 个强连通分量的大小为 $bsize_i$。
- `void add_edge(int u, int v)`：添加一条 $u \to v$ 的有向边。
- `int eval()`：求图 G 的强连通分量，并返回强连通分量的个数。
- `void init(int n)`：初始化点数为 n 的图。

```
class Tarjan
{
public:
    struct edge
    {
        int u, v;
        edge *next;
    } epool[maxm], *edges[maxn];
    int n, ecnt, dcnt, bcnt;
    stack<int> stk;
    int instk[maxn], dfn[maxn], low[maxn];
    int belong[maxn], bsize[maxn];
    void add_edge(int u, int v)
    {
        edge *p = &epool[++ecnt];
        p->u = u; p->v = v;
        p->next = edges[u]; edges[u] = p;
        return ;
    }
    void dfs(int p)
    {
```

```
        low[p] = dfn[p] = ++dcnt;
        stk.push(p);
        instk[p] = true;
        for (edge *ep = edges[p]; ep; ep = ep->next) {
            int q = ep->v;
            if (!dfn[q]) {
                dfs(q);
                if (low[q] < low[p])
                    low[p] = low[q];
            } else if (instk[q] && dfn[q] < low[p]) {
                low[p] = dfn[q];
            }
        }
        if (dfn[p] == low[p]) {
            bsize[++bcnt] = 0;
            int q = 0;
            do {
                q = stk.top();
                stk.pop();
                instk[q] = false;
                belong[q] = bcnt;
                bsize[bcnt] += 1;
            } while (q != p);
        }
        return ;
    }
    int eval(void)
    {
        while (!stk.empty())
            stk.pop();
        dcnt = bcnt = 0;  // dfs counter, component counter
        rep(i, 1, n) {
            dfn[i] = low[i] = 0;
            instk[i] = false;
            belong[i] = bsize[i] = 0;
        }
        rep(i, 1, n)
            if (!dfn[i])
                dfs(i);
        return bcnt;
    }
    void init(int n)
    {
        this->n = n;
        ecnt = 0;
        rep(i, 1, n)
            edges[i] = nullptr;
        return ;
    }
} graph;
```

SPFA 最短路

- 题目：poj3259
- 依赖：

模板描述

给定 n 个点 m 条边的有向图 $G=(V,E)$，$s \in V$ 以及每条边的长度，判定图是否存在负权回路，若无则求点 s 到 V 中任意一点的最短距离及满足距离最短的任意一条最短路径。

复杂度

- Space:
  - Worst Case: $O(n+m)$
  - Amortized: $O(n+m)$
  - Best Case: $O(n+m)$
- Time:
  - Worst Case: $O(nm)$
  - Amortized: $O(km)$
  - Best Case: $O(km)$

算法简述

松弛操作：对于边 $e(i,j) \in E$，$dist_j := min(dist_j, dist_i + len_e)$

写法类似 Dijkstra 算法，不使用堆优化，并保存一个 $qcnt$ 数组代表每个点进入队列的次数。另记一个数组 $inque$ 代表一个数是否在队列里。如果一个点重复入队超过 $n-1$ 次，则原图必存在负权回路。

通过比较当前松弛点的距离和队首的距离选择是推入队首还是队末的方法被称为前向星优化。注意在某些情况下该优化方法会被出题人针对并被卡掉，这时需去除前向星优化使用朴素方法。

获得最短路径的方法已在 Dijkstra 模板中记录，此处不予冗述。

一般地，前向星优化能够加速 1 到 10 倍不等，取决于数据强度。在无负权回路的图下，考虑到玄学常数 k，强烈建议使用堆优化 Dijkstra 而不是 SPFA。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：有向边的数量上限。
- `lli infinit`：规定的无限远距离。
- `lli[] dist`：对于每个点 $i \in [1,n]$，点 i 到源点 s 的距离。
- `edge[] from`：从源点到点 i 的最短路径上指向 i 的边的地址。
- `void add_edge(int u, int v, lli len)`：添加一条 $u \to v$，长度为 len 的有向边。

- `bool eval(int s)`：以 s 为源点，计算到 $[1,n]$ 所有点的最短路。同时若函数返回 $false$ 则该图包含负权回路，反之则不存在负权回路。
- `void init(int n)`：初始化点数为 n 的图。

```
class SPFA
{
public:
    struct edge
    {
        int u, v;
        lli len;
        edge *next;
    } epool[maxm], *edges[maxn], *from[maxn];
    int n, ecnt;
    lli dist[maxn];
    int qcnt[maxn], inque[maxn];
    void add_edge(int u, int v, lli len)
    {
        edge *p = &epool[++ecnt];
        p->u = u; p->v = v; p->len = len;
        p->next = edges[u]; edges[u] = p;
        return ;
    }
    bool eval(int s)
    {
        #define USE_SLF
        #ifdef USE_SLF
        deque<int> que;
        #else
        queue<int> que;
        #endif
        rep(i, 0, n) {
            qcnt[i] = 0;
            inque[i] = false;
            dist[i] = infinit;
            from[i] = nullptr;
        }
        dist[s] = 0;
        qcnt[s] += 1;
        inque[s] = true;
        #ifdef USE_SLF
        que.push_back(s);
        #else
        que.push(s);
        #endif
        while (!que.empty()) {
            int p = que.front();
            #ifdef USE_SLF
            que.pop_front();
            #else
            que.pop();
            #endif
            inque[p] = false;
            for (edge *ep = edges[p]; ep; ep = ep->next)
```

```
                if (dist[p] + ep->len < dist[ep->v]) {
                    dist[ep->v] = dist[p] + ep->len;
                    from[ep->v] = ep;
                    if (!inque[ep->v]) {
                        inque[ep->v] = true;
                        qcnt[ep->v] += 1;
                        if (qcnt[ep->v] >= n)
                            return false;
                        #ifdef USE_SLF
                        if (que.empty() || dist[ep->v] > dist[que.fr
ont()])
                            que.push_back(ep->v);
                        else
                            que.push_front(ep->v);
                        #else
                        que.push(ep->v);
                        #endif
                    }
                }
            }
            return true;
        }
        void init(int n)
        {
            this->n = n;
            ecnt = 0;
            rep(i, 1, n)
                edges[i] = nullptr;
            return ;
        }
} graph;
```

SPFA 费用流

- 题目：luogu3381
- 依赖：

模板描述

给定 n 个点 m 条边的有向图 $G = (V, E)$，每条边的流量和代价，给定源点 s 和汇点 t，求从点 s 到点 t 的最大流量，以及达成最大流量前提下的最小总花费。

复杂度

- Space:
  - Worst Case: $O(n + m)$
  - Amortized: $O(n + m)$
  - Best Case: $O(n + m)$
- Time:
  - Worst Case: $O(n^2 m)$
  - Amortized: $O(n^2 m)$

- Best Case: $O(n+m)$

算法简述

SPFA 费用流每次求一条代价最小的增广路，然后将这条增广路上的最大流量求出，并去掉这条流量。进一步地，可以使用多路增广，用类似 Dinic 的写法加速每次 BFS 增广的流量。我们甚至可以借用 Dijkstra 写法中的优先队列来优化 BFS 速度。注意这里 `set` 去重的效率并不如 `priority_queue` 直接推入。

调用方法

- `int maxn`：点的数量上限。
- `int maxm`：有向边的数量上限。
- `lli infinit`：规定的无限大流量。
- `void add_edge(int u, int v, lli flow, lli cost)`：添加一条 $u \to v$，流量为 $flow$，代价为 $cost$ 的有向边。
- `<lli, lli> eval()`：计算该有向图的最大流和对应的最小费用。
- `void init(int n, int s, int t)`：初始化点数为 n，源点为 s，汇点为 t 的图。

```
class SPFACostFlow
{
public:
    typedef pair<lli, int> pli;
    struct edge
    {
        int u, v;
        lli flow, cost;
        edge *next, *rev;
    } epool[maxm], *edges[maxn], *from[maxn];
    int n, s, t, ecnt;
    int vis[maxn];
    lli dist[maxn], height[maxn];
    void add_edge(int u, int v, lli flow, lli cost)
    {
        edge *p = &epool[++ecnt],
             *q = &epool[++ecnt];
        p->u = u; p->v = v; p->flow = flow; p->cost = cost;
        p->next = edges[u]; edges[u] = p;
        q->u = v; q->v = u; q->flow = 0; q->cost = - cost;
        q->next = edges[v]; edges[v] = q;
        p->rev = q; q->rev = p;
        return ;
    }
    bool spfa(void)
    {
        rep(i, 1, n)
            dist[i] = infinit;
        priority_queue<pli, vector<pli>, greater<pli>> pq;
        dist[s] = 0;
        pq.push(make_pair(dist[s], s));
```

```
        while (!pq.empty()) {
            pli pr = pq.top();
            int p = pr.second;
            pq.pop();
            if (dist[p] < pr.first)
                continue;
            for (edge *ep = edges[p]; ep; ep = ep->next)
                if (ep->flow > 0 && dist[p] + ep->cost +
                        height[p] - height[ep->v] < dist[ep->v]) {
                    dist[ep->v] = dist[p] + ep->cost + height[p] -
                                  height[ep->v];
                    from[ep->v] = ep;
                    pq.push(make_pair(dist[ep->v], ep->v));
                }
        }
        return dist[t] < infinit;
    }
    lli dfs(int p, lli flow, lli& rcost)
    {
        if (p == t || flow == 0)
            return flow;
        vis[p] = true;
        lli used = 0;
        for (edge *ep = edges[p]; ep; ep = ep->next)
            if (!vis[ep->v] && ep->flow > 0 && height[ep->v] ==
                    height[p] + ep->cost) {
                lli tmp = dfs(ep->v, min(flow - used, ep->flow), rco
st);
                used += tmp;
                ep->flow -= tmp;
                ep->rev->flow += tmp;
                rcost += tmp * ep->cost;
                if (used == flow)
                    break;
            }
        vis[p] = false;
        return used;
    }
    pair<lli, lli> eval(void)
    {
        memclr(height, n);
        lli rflow = 0, rcost = 0;
        while (spfa()) {
            rep(i, 1, n)
                height[i] = min(infinit, height[i] + dist[i]);
            lli tmp = dfs(s, infinit, rcost);
            if (tmp == 0)
                break;
            rflow += tmp;
        }
        return make_pair(rflow, rcost);
    }
    void init(int n, int s, int t)
    {
```

```
        this->n = n;
        this->s = s;
        this->t = t;
        ecnt = 0;
        memclr(edges, n);
        return ;
    }
} graph;
```

Splay 树

- 题目：
- 依赖：

模板描述

Splay

复杂度

- Space:
  - Worst Case: $O(n)$
  - Amortized: $O(n)$
  - Best Case: $O(n)$
- Time:
  - Worst Case: $O(n)$
  - Amortized: $O(\log n)$
  - Best Case: $O(\log n)$

算法简述

Splay（待补

调用方法

- `int maxn`：点的数量上限。
- `void init(int n)`：初始化点数为 n 的限制组。

```
class SplayTree
{
public:
    int n, root, ncnt;
    int ch[maxn][2], parent[maxn], size[maxn];
    lli val[maxn], vsm[maxn], vmn[maxn], vmx[maxn];
    lli lazyadd[maxn];
    bool lazyswp[maxn];
    #define lc(x) ch[x][0]
    #define rc(x) ch[x][1]
    #define par(x) parent[x]
    void push_down(int p)
    {
        rep(_, 0, 1)
```

```
        if (ch[p][_]) {
            lazyadd[ch[p][_]] += lazyadd[p];
            val[ch[p][_]] += lazyadd[p];
            vsm[ch[p][_]] += size[ch[p][_]] * lazyadd[p];
            vmn[ch[p][_]] += lazyadd[p];
            vmx[ch[p][_]] += lazyadd[p];
            lazyswp[ch[p][_]] ^= lazyswp[p];
        }
    if (lazyswp[p])
        swap(lc(p), rc(p));
    lazyadd[p] = 0;
    lazyswp[p] = false;
    return ;
}
void pull_up(int p)
{
    size[p] = size[lc(p)] + 1 + size[rc(p)];
    vsm[p] = vmn[p] = vmx[p] = val[p];
    rep(_, 0, 1)
        if (ch[p][_]) {
            vsm[p] += vsm[ch[p][_]];
            vmn[p] = min(vmn[p], vmn[ch[p][_]]);
            vmx[p] = max(vmx[p], vmx[ch[p][_]]);
        }
    return ;
}
int makenode(int q, lli v)
{
    int p = ++ncnt; n += 1;
    lc(p) = rc(p) = 0;
    par(p) = q;
    size[p] = 1;
    val[p] = vsm[p] = vmn[p] = vmx[p] = v;
    lazyadd[p] = 0;
    lazyswp[p] = false;
    return p;
}
void rotate(int p)
{
    int q = par(p), g = par(q);
    push_down(q);
    push_down(p);
    int x = p == rc(q);
    // relink connections
    ch[q][x] = ch[p][!x];
    if (ch[q][x]) par(ch[q][x]) = q;
    ch[p][!x] = q; par(q) = p;
    par(p) = g;
    if (g) ch[g][q == rc(g)] = p;
    pull_up(q);
    pull_up(p);
    return ;
}
int pre(int p)
```

```
{
    if (!lc(p)) {
        while (p == lc(par(p)))
            p = par(p);
        p = par(p);
    } else {
        p = lc(p);
        while (rc(p))
            p = rc(p);
    }
    return p;
}
int suc(int p)
{
    if (!rc(p)) {
        while (p == rc(par(p)))
            p = par(p);
        p = par(p);
    } else {
        p = rc(p);
        while (lc(p))
            p = lc(p);
    }
    return p;
}
void splay(int p, int t)
{
    for (int q = 0; (q = par(p)) && q != t; rotate(p))
        if (par(q) && par(q) != t)
            rotate((p == lc(q)) == (q == lc(par(q))) ? q : p);
    if (t == 0) root = p;
    return ;
}
int find(int x)
{
    int p = root;
    while (x > 0 && p) {
        push_down(p);
        if (x <= size[lc(p)]) {
            p = lc(p);
            continue;
        } x -= size[lc(p)];
        if (x <= 1) {
            return p;
        } x -= 1;
        p = rc(p);
    }
    return 0;
}
int find_bin_geq(lli v)
{
    // first p s.t. val[p] >= v
    int p = root;
    int q = 2;
```

```
    while (p) {
        push_down(p);
        if (val[p] == v) {
            return p;
        } else if (val[p] < v) {
            p = rc(p);
        } else if (val[p] > v) {
            q = val[p] < val[q] ? p : q;
            p = lc(p);
        }
    }
    return q;
}
int find_bin_leq(lli v)
{
    // last p. s.t. val[p] <= v
    int p = root;
    int q = 1;
    while (p) {
        push_down(p);
        if (val[p] == v) {
            return p;
        } else if (val[p] > v) {
            p = lc(p);
        } else if (val[p] < v) {
            q = val[p] > val[q] ? p : q;
            p = rc(p);
        }
    }
    return q;
}
void insert(int x, lli v)
{
    int lp = find(x + 1), rp = suc(lp);
    splay(rp, 0);
    splay(lp, root);
    int c = makenode(lp, v);
    rc(lp) = c;
    pull_up(lp);
    pull_up(rp);
    return ;
}
void insert_bin(lli v)
{
    int p = root;
    int q = 0;
    while (p) {
        push_down(p);
        if (v <= val[p]) {
            if (!lc(p)) {
                lc(p) = makenode(p, v);
                q = lc(p);
                break;
            }
```

```
            p = lc(p);
        } else {
            if (!rc(p)) {
                rc(p) = makenode(p, v);
                q = rc(p);
                break;
            }
            p = rc(p);
        }
    }
    splay(q, 0);
    return ;
}
void remove(int l, int r)
{
    int lp = find(l - 1 + 1), rp = find(r + 1 + 1);
    splay(rp, 0);
    splay(lp, root);
    int c = rc(lp);
    par(c) = 0;
    rc(lp) = 0;
    pull_up(lp);
    pull_up(rp);
    n -= r - l + 1;
    return ;
}
lli query(int l, int r, int mode)
{
    // mode = 1: count, 2: sum, 3: min, 4: max
    if (l > r) {
        if (mode == 1)
            return 0;
        if (mode == 2)
            return 0;
        if (mode == 3)
            return infinit;
        if (mode == 4)
            return - infinit;
    }
    int lp = find(l - 1 + 1), rp = find(r + 1 + 1);
    splay(rp, 0);
    splay(lp, root);
    int p = rc(lp);
    if (mode == 1)
        return size[p];
    else if (mode == 2)
        return vsm[p];
    else if (mode == 3)
        return vmn[p];
    else if (mode == 4)
        return vmx[p];
    return 0;
}
lli query_bin(lli lb, lli rb, int mode)
```

```
{
    // mode = 1: count, 2: sum, 3: min, 4: max
    if (lb > rb) {
        if (mode == 1)
            return 0;
        if (mode == 2)
            return 0;
        if (mode == 3)
            return infinit;
        if (mode == 4)
            return - infinit;
    }
    int lp = find_bin_leq(lb - 1), rp = find_bin_geq(rb + 1);
    printf("found %d %d\n", lp, rp);
    splay(rp, 0);
    splay(lp, root);
    int p = rc(lp);
    if (mode == 1)
        return size[p];
    else if (mode == 2)
        return vsm[p];
    else if (mode == 3)
        return vmn[p];
    else if (mode == 4)
        return vmx[p];
    return 0;
}
void modify_add(int l, int r, lli v)
{
    int lp = find(l - 1 + 1), rp = find(r + 1 + 1);
    splay(rp, 0);
    splay(lp, root);
    int p = rc(lp);
    lazyadd[p] += v;
    val[p] += v;
    vsm[p] += size[p] * v;
    vmn[p] += v;
    vmx[p] += v;
    pull_up(lp);
    pull_up(rp);
    return ;
}
void modify_swp(int l, int r)
{
    int lp = find(l - 1 + 1), rp = find(r + 1 + 1);
    splay(rp, 0);
    splay(lp, root);
    int p = rc(lp);
    lazyswp[p] ^= 1;
    return ;
}
void init(void)
{
    n = ncnt = 0;
```

```
        root = makenode(0, - infinit);
        rc(root) = makenode(root, infinit);
        pull_up(rc(root));
        pull_up(root);
        return ;
    }
} splay;
```

Trie 字典树

- 题目：hdu1251
- 依赖：

模板描述

Trie

复杂度

- Space:
  - Worst Case: $O(n)$
  - Amortized: $O(n)$
  - Best Case: $O(n)$
- Time:
  - Worst Case: $O(n)$
  - Amortized: $O(\log n)$
  - Best Case: $O(\log n)$ 待补

算法简述

Trie 待补

调用方法

- `int maxn`：点的数量上限。
- `void init(int n)`：初始化点数为 n 的限制组。

```
class Trie
{
public:
    int ncnt, root;
    int ch[maxn][26];
    bool flag[maxn];
    int size[maxn], data[maxn];
    int make_node(void)
    {
        int p = ++ncnt;
        memclr(ch[p], 25);
        flag[p] = false;
        size[p] = data[p] = 0;
        return p;
```

```
    }
    void insert(char str[], int vdata = 0)
    {
        int p = root;
        for (int i = 0; str[i] != '\0'; i++) {
            int v = str[i] - 'a';
            size[p] += 1;
            if (!ch[p][v])
                ch[p][v] = make_node();
            p = ch[p][v];
        }
        size[p] += 1;
        flag[p] = true;
        data[p] = vdata;
        return ;
    }
    void remove(char str[])
    {
        int p = root;
        for (int i = 0; str[i] != '\0'; i++) {
            int v = str[i] - 'a';
            size[p] -= 1;
            p = ch[p][v];
        }
        size[p] -= 1;
        flag[p] = false;
        return ;
    }
    bool find(char str[])
    {
        int p = root;
        for (int i = 0; str[i] != '\0'; i++) {
            int v = str[i] - 'a';
            if (!ch[p][v])
                return false;
            p = ch[p][v];
        }
        // return anything you want here
        return false;
    }
    void init(void)
    {
        ncnt = 0;
        root = make_node();
        return ;
    }
} trie;
```