Standard Code Library

2019 年 8 月 28 日

目录

1 Data Structure

1.1 Basic ZKW

- 根节点为 1，$[1, n)$ 是内部结点，$[n, 2n)$ 是叶结点，+n 快速找到叶子
- op 需要满足：结合律、交换律

```
template <typename T>
struct ZKW {
    T tree[MAXN * 2];
    int n;
    void build() {
        for(int p = n - 1; p >= 1; p--) tree[p] = op(tree[p << 1], tree[p << 1 |
            ↪1]);
    }
    void update(int p, T val) {//a[p] = val
        tree[p += n] = val;
        for(p >>= 1; p > 0; p >>= 1) tree[p] = op(tree[p << 1], tree[p << 1 | 1]);
    }
    T query(int l, int r) {//[l, r]
        l += n; r += n;
        T res = tree[r];
        for(; l < r; l >>= 1, r >>= 1) {
            if(l & 1) res = op(res, tree[l++]);
            if(r & 1) res = op(res, tree[--r]);
        }
        return res;
    }
};
```

1.2 Segment Tree

线段树设计模式：

- 节点的值：存储什么样的信息才能快速完成各项操作。
- pushdown：把节点的信息等价转化给孩子（将更改下推）。
- merge：把两个节点合并（根据下层的值构造上层的值）。

下面给出一个例子，实现了区间更改/区间查询最小值的功能。

```
class SGT {
    struct Node {
        ll min_, s;
    } tree[maxn * 4];
    void merge(Node &res, Node &l, Node &r) {
        res.min_ = std::min(l.min_ + l.s, r.min_ + r.s);
        res.s = 0;
    }
    void push_down(Node &root, Node &l, Node &r) {
        root.min_ += root.s;
        l.s += root.s;
        r.s += root.s;
        root.s = 0;
    }
public:
    //root节点对应的线段是[left, right]
    void add(int root, int left, int right, int l, int r, ll d) {
        if(right < l || r < left) return;
        if(l <= left && right <= r) tree[root].s += d;
        else {
            int mid = (left + right) / 2;
            push_down(tree[root], tree[root << 1], tree[root << 1 | 1]);
            add(root << 1, left, mid, l, r, d);
            add(root << 1 | 1, mid + 1, right, l, r, d);
            merge(tree[root], tree[root << 1], tree[root << 1 | 1]);
        }
    }
    ll query_min(int root, int left, int right, int l, int r) {
        if(l <= left && right <= r) return tree[root].min_ + tree[root].s;
        else {
            int mid = (left + right) / 2;
            push_down(tree[root], tree[root << 1], tree[root << 1 | 1]);
            ll L = LONG_LONG_MAX, R = LONG_LONG_MAX;
            if(l <= mid) L = query_min(root << 1, left, mid, l, r);
```

```
            if(mid + 1 <= r) R = query_min(root << 1 | 1, mid + 1, right, l, r);
            return std::min(L, R);
        }
    }
} sgt;
```

1.3 Persistent Segment Tree

- 初学 Persistent DS, MIT 的公开课还没看明白。
- pointer machine
- version
- partial / full persistent, thinking / method / analysis

下面是一棵支持单点修改、区间求和的持久化线段树。

```
struct PSGT {
    struct Node {
        int sum;
        int lc, rc;
    } tree[MAXN * 20];
    int cnt_node;
    int copy_of(int u) {
        tree[cnt_node] = tree[u];
        return cnt_node++;
    }
    void init() {
        cnt_node = 1;
    }
    int build(int l, int r) {
        int root = cnt_node++;
        if(l == r) {
            tree[root].lc = tree[root].rc = -1;
            tree[root].sum = 0;
        }
        else {
            int mid = (l + r) / 2;
            int lc = build(l, mid), rc = build(mid + 1, r);
            tree[root].lc = lc;
            tree[root].rc = rc;
            tree[root].sum = tree[lc].sum + tree[rc].sum;
        }
        return root;
    }
    int query(int root, int left, int right, int l, int r) {
        if(l <= left && right <= r) return tree[root].sum;
        if(r < left || right < l) return 0;
        int mid = (left + right) / 2;
        return query(tree[root].lc, left, mid, l, r) + query(tree[root].rc, mid +
            ↪1, right, l, r);
    }
```

```
    int modify(int root, int left, int right, int p, int val) {
        int res = copy_of(root);
        if(left == p && right == p) tree[res].sum += val;
        else {
            int mid = (left + right) / 2;
            if(p <= mid) tree[res].lc = modify(tree[root].lc, left, mid, p, val);
            else tree[res].rc = modify(tree[root].rc, mid + 1, right, p, val);
            tree[res].sum = tree[tree[res].lc].sum + tree[tree[res].rc].sum;
        }
        return res;
    }
} psgt;
```

1.4 Binary Indexed Tree

- 下标范围 $[1, n]$
- 第 i 个位置存放原数组以 a_i 为最后一个元素的 $lowbit(i)$ 个元素的和
- $lowbit$ 可用来快速查找二进制表示中最低位的 1

```cpp
template<typename T>
struct BIT {
    static const int maxn = 1e6;
    T tree[maxn];
    int n;//[1, n]
    std::function<T(T, T)> op;
    inline int lowbit(int i) { return i & (-i); }

    void init(int n, const T &ID, const std::function<T(T, T)> &op) {
        this->n = n, this->op = op;
        fill(tree, tree + n + 1, ID);
    }
    T query_prefix(int r) {//[1, r]
        T res = tree[r];
        for(r -= lowbit(r); r > 0; r -= lowbit(r)) res = op(res, tree[r]);
        return res;
    }
    void update(int p, const T &val) {//a[p] = op(a[p], val)
        while(p <= n) tree[p] = op(tree[p], val), p += lowbit(p);
    }
};
```

1.5 Treap

初始时仅需创建空指针（即一棵空树），大部分接口传入的都是指针的引用。
实现的接口：size, insert, remove, select, lower_bound, upper_bound, contain
***** 瞎搞警告 *****
***** 大常数警告 *****

```
struct treapNode {
    T val;
    int s, p;
    treapNode* c[2];
} buf[MAXN];
int cnt_buf;
void rotate(treapNode* &root, int d) {
    treapNode *y = root->c[d];

    y->s = root->s;
    root->s = size(root->c[d ^ 1]) + size(y->c[d ^ 1]) + 1;

    root->c[d] = y->c[d ^ 1], y->c[d ^ 1] = root, root = y;
}

int size(treapNode* root) { return root == nullptr ? 0 : root->s; }

void insert(treapNode* &root, const T &val) {
    if(root == nullptr) {
        root = &buf[cnt_buf++];
        root->val = val;
        root->s = 1;
        root->p = rand();
        root->c[0] = root->c[1] = nullptr;
    }
    else {
        int d = val < root->val ? 0 : 1;
        insert(root->c[d], val);
        ++root->s;
        if(root->c[d]->p < root->p) rotate(root, d);
    }
}

bool remove(treapNode* &root, const T &val) {
    if(root == nullptr) return false;
    if(root->val == val) {
        for(int i = 0; i < 2; i++) if(root->c[i] == nullptr) {
```

```
            root = root->c[i ^ 1];
            return true;
        }
        int d = root->c[0]->p < root->c[1]->p ? 0 : 1;
        rotate(root, d);
        remove(root->c[d ^ 1], val);
        --root->s;
        return true;
    }
    else if(remove(root->c[val < root->val ? 0 : 1], val)) {
        --root->s;
        return true;
    }
    return false;
}
T& select(treapNode* root, int k) {
    while(root != nullptr) {
        int cur = size(root->c[0]);
        if(cur == k) return root->val;
        root = root->c[k < cur ? 0 : 1];
        if(k > cur) k -= ++cur;
    }
}
int lower_bound(treapNode* root, T val) {
    int ans = 0;
    while(root != nullptr) {
        if(root->val >= val) root = root->c[0];
        else {
            ans += size(root->c[0]) + 1;
            root = root->c[1];
        }
    }
    return ans;
}
int upper_bound(treapNode* root, T val) {
    int ans = 0;
    while(root != nullptr) {
        if(root->val > val) root = root->c[0];
        else {
            ans += size(root->c[0]) + 1;
            root = root->c[1];
        }
    }
```

```
    return ans;
}
bool contain(treapNode* root, const T &val) {
    while(root != nullptr && root->val != val) {
        if(val < root->val) root = root->c[0];
        else root = root->c[1];
    }
    return root != nullptr;
}
```

1.6 Splay Tree

可分割/合并序列可用 std::rope 完成。

```cpp
template<typename T>
struct splayNode {
    T val;
    int s;
    splayNode<T>* c[2];
    splayNode(const T &val) : val(val) {}

    static std::allocator<splayNode<T>> alloc;
    static void rotate(splayNode<T>* &root, int d) {
        auto y = root->c[d];

        y->s = root->s;
        root->s = 1 + size(root->c[d ^ 1]) + size(y->c[d ^ 1]);

        root->c[d] = y->c[d ^ 1], y->c[d ^ 1] = root, root = y;
    }
    static void splay_kth(splayNode<T>* &root, int k) {
        if(root == nullptr) return;
        splayNode<T> *p[2], **p_[2] = { &p[0], &p[1] };
        std::stack<splayNode<T>*> s[2];
        while(true) {
            int cur = size(root->c[0]);
            int d = k < cur ? 0 : 1;
            if(k == cur || root->c[d] == nullptr) break;
            if(d == 0 && k < size(root->c[d]->c[d]) && root->c[d]->c[d] != nullptr)
                rotate(root, d);
            if(d == 1 && k >= size(root) - size(root->c[d]->c[d]) && root->c[d]->c[d] != nullptr)
                rotate(root, d);
            *p_[d ^ 1] = root, p_[d ^ 1] = &root->c[d], s[d ^ 1].push(root);
            if(d == 1) k -= size(root->c[0]) + 1;
            root = root->c[d];
        }
        for(int i = 0; i < 2; i++) {
            *p_[i] = root->c[i], root->c[i] = p[i];
            while(!s[i].empty()) s[i].top()->s = 1 + size(s[i].top()->c[0]) + size(s[i].top()->c[1]), s[i].pop();
        }
        root->s = 1 + size(root->c[0]) + size(root->c[1]);
    }
```

```

    friend int size(splayNode<T>* root) { return root == nullptr ? 0 : root->s; }
    friend void build(splayNode<T>* &root, T* a, int l, int r) {
        if(l > r) { root = nullptr; return; }
        int mid = (l + r) / 2;
        root = alloc.allocate(1), alloc.construct(root, a[mid]);
        build(root->c[0], a, l, mid - 1), build(root->c[1], a, mid + 1, r);
        root->s = 1 + size(root->c[0]) + size(root->c[1]);
    }
    friend void split(splayNode<T>* root, int k, splayNode<T>* &left, splayNode<T>* &right) {
        if(k >= size(root)) left = root, right = nullptr;
        else {
            splay_kth(root, k);
            left = root->c[0], right = root;
            root->s -= size(root->c[0]), root->c[0] = nullptr;
        }
    }
    friend void attach(splayNode<T>* &left, splayNode<T>* right) {
        if(left == nullptr) left = right;
        else splay_kth(left, size(left) - 1), left->c[1] = right, left->s += size(right);
    }
    friend void clear(splayNode<T>* &root) {
        std::stack<splayNode<T>*> s;
        if(root != nullptr) s.push(root);
        while(!s.empty()) {
            auto x = s.top(); s.pop();
            if(x->c[0] != nullptr) s.push(x->c[0]);
            if(x->c[1] != nullptr) s.push(x->c[1]);
            alloc.destroy(x), alloc.deallocate(x, 1);
        }
        root = nullptr;
    }
};
template<typename T>
std::allocator<splayNode<T>> splayNode<T>::alloc;
```

2 Graph

2.1 Maximum Flow

最大流的一些知识点:

- 流网络的切割 (净流量、容量)
- 残量网络的定义、增广路对残量网络的增广
- 最大流最小割定理
- FORD-FULKERSON 方法、EK 算法

最大流的 dinic 算法，每次先在残量网络中 bfs 构建层次图（level），再 dfs 寻找增广路。
需要注意的地方:

- dinic_dfs 里的当前边优化
- 每次 add_edge 的时候都一次加两条，方便找到反向边。(果然不能读书读死了啊。。只要不用邻接矩阵就可以加入反向边的）

```
struct Graph {
    struct { int v, cap, next; } e[MAXM];
    int head[MAXN], cnt_edge;
    void add_edge_(int u, int v, int cap) {
        e[cnt_edge] = { v, cap, head[u] };
        head[u] = cnt_edge++;
    }
    void add_edge(int u, int v, int cap) {
        add_edge_(u, v, cap);
        add_edge_(v, u, 0);
    }
    void init() {
        memset(head, 0xff, sizeof(head));
        cnt_edge = 0;
    }

    int level[MAXN], cur[MAXN];
    bool dinic_bfs(int s, int t) {
        memset(level, 0xff, sizeof(level));
        std::queue<int> q;

        level[s] = 0;
        cur[s] = head[s];
        q.push(s);
        while(!q.empty()) {
            int u = q.front(); q.pop();
```

```
            if(u == t) return true;
            for(int i = head[u]; i != -1; i = e[i].next) if(e[i].cap) {
                if(level[e[i].v] == -1) {
                    level[e[i].v] = level[u] + 1;
                    cur[e[i].v] = head[e[i].v];
                    q.push(e[i].v);
                }
            }
        }
        return false;
    }

    int dinic_dfs(int u, int cur_min, int t) {
        if(u == t) return cur_min;
        if(level[u] >= level[t]) return 0;

        int res = 0;
        for(int& i = cur[u]; cur_min && i != -1; i = e[i].next) if(e[i].cap) {
            if(level[e[i].v] == level[u] + 1) {
                int nxt_min = std::min(cur_min, e[i].cap);
                int x = dinic_dfs(e[i].v, nxt_min, t);
                cur_min -= x;
                e[i].cap -= x;
                e[i ^ 1].cap += x;
                res += x;
            }
        }
        return res;
    }

    int dinic(int s, int t) {
        int res = 0;
        while(dinic_bfs(s, t)) {
            res += dinic_dfs(s, inf, t);
        }
        return res;
    }
} G;
```

2.2 Min Cost Flow

把 EK 算法的 BFS 改成了 SPFA。

```
struct Graph {
    struct { int v, cap, cost, next; } e[MAXN * MAXN];
    int cnt_edge, head[MAXN];
    void init() {
        cnt_edge = 0;
        memset(head, 0xff, sizeof(head));
    }
    void add_edge_(int u, int v, int cap, int cost) {
        e[cnt_edge] = { v, cap, cost, head[u] };
        head[u] = cnt_edge++;
    }
    void add_edge(int u, int v, int cap, int cost) {
        add_edge_(u, v, cap, cost);
        add_edge_(v, u, 0, -cost);
    }
    int dis[MAXN], pre[MAXN];
    bool inque[MAXN];
    bool spfa(int s, int t) {
        memset(dis, 0x3f, sizeof(dis));
        std::queue<int> que;
        bool flag = false;

        dis[s] = 0;
        inque[s] = true;
        que.push(s);
        while(!que.empty()) {
            int u = que.front(); que.pop();
            if(u == t) flag = true;
            inque[u] = false;
            for(int i = head[u]; i != -1; i = e[i].next) if(e[i].cap > 0) {
                if(dis[u] + e[i].cost < dis[e[i].v]) {
                    dis[e[i].v] = dis[u] + e[i].cost;
                    pre[e[i].v] = i;
                    if(!inque[e[i].v]) {
                        inque[e[i].v] = true;
                        que.push(e[i].v);
                    }
                }
            }
        }
```

```
        return flag;
    }
    int min_cost_flow(int s, int t, int &flow) {
        int cost = 0;
        flow = 0;
        while(spfa(s, t)) {
            int min_cap = 1e9;
            for(int u = t; u != s; u = e[pre[u] ^ 1].v) if(min_cap > e[pre[u]].cap)
              ↪ min_cap = e[pre[u]].cap;
            for(int u = t; u != s; u = e[pre[u] ^ 1].v) {
                cost += min_cap * e[pre[u]].cost;
                e[pre[u]].cap -= min_cap;
                e[pre[u] ^ 1].cap += min_cap;
            }
            flow += min_cap;
        }
        return cost;
    }
} G;
```

2.3 Matching and Covers

定义:

- A **vertex cover** of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set.
- An **edge cover** of a graph is a set of edges such that every vertex of the graph is incident to at least one edge of the set.
- A **matching** or **independent edge set** of a graph is a set of edges without common vertices.
- An **independent set** of a graph is a set S of vertices such that for every two vertices in S, there is no edge connecting the two.
  以上都是定义在任何类型的(无向?)图上的?
- A **path cover** of a directed graph is a set of directed paths such that every vertex of the graph belongs to **exactly** one path.

定理:

- 对于全部二分图, |Minimum Vertex Cover| = |Maximum Matching|
- 对于全部(无向?)图 $G = (V, E)$, |Minimum Edge Cover| = |V| - |Maximum Matching|
- 对于全部 DAG，按一般方法把一个点拆成两个，令新二分图的最大匹配为 x，| 原图的最小路径覆盖 $|=|V|-x$
- 对于全部不带有向环的传递闭包，| 最大独立集 | = | 最小路径覆盖 | (为啥啊???)
- HALL's theory

求二**分图**最大匹配的匈牙利算法 **(只适用于二分图)**，单向边和双向边都适用。
每次 dfs 寻找一条未匹配/已匹配边交错出现的路径（与寻找一条网络流模型中的增广路等价）。

```
//链式前向星
bool vis[MAXN];
int link[MAXN];
bool hungarian_dfs(int u) {
    for(int i = head[u]; i != -1; i = e[i].next) if(!vis[e[i].v]) {
        vis[e[i].v] = true;
        if(link[e[i].v] == -1 || hungarian_dfs(link[e[i].v])) {
            link[e[i].v] = u, link[u] = e[i].v;
            return true;
        }
    }
    return false;
}
int hungarian() {
    memset(link, 0xff, sizeof(link));
    int res = 0;
```

```
    for(int i = 0; i < n_vertex; i++) if(link[i] == -1) {
        memset(vis, 0, sizeof(vis));
        if(hungarian_dfs(i)) res++;
    }
    return res;
}
```

2.4 Cut and Bridge and BCC

求**无向图**的割点、割边（桥）、点双连通分量的 tarjan 算法，使用了 dfn 和 low。
bcc 编号从 1 开始。

```
struct Graph {
    struct { int v, next; } e[MAXM];
    int head[MAXN], cnt_edge;
    void add_edge(int u, int v) {
        e[cnt_edge] = { v, head[u] };
        head[u] = cnt_edge++;
    }
    void init() {
        cnt_edge = 0;
        memset(head, 0xff, sizeof(head));
    }
    int dfs_clk, dfn[MAXN];
    bool cut[MAXN], bridge[MAXM];
    int bcc_dfs(int u, int pre) {
        dfn[u] = ++dfs_clk;
        int cnt_child = 0;
        int low_u = dfn[u];
        for(int i = head[u]; i != -1; i = e[i].next) if(e[i].v != pre) {
                int v = e[i].v;
                if(dfn[v] == 0) {
                    cnt_child++;
                    int low_v = bcc_dfs(v, u);
                    if(low_v >= dfn[u]) cut[u] = true;
                    if(low_v >= dfn[v]) bridge[i] = bridge[i ^ 1] = true;
                    low_u = std::min(low_u, low_v);
                }
                else {
                    low_u = std::min(low_u, dfn[v]);
                }
            }
        if(u == pre && cnt_child == 1) cut[u] = false;
        return low_u;
    }
    void calc_bcc() {
        memset(cut, 0, sizeof(cut));
        memset(bridge, 0, sizeof(bridge));
        memset(dfn, 0, sizeof(dfn));
        dfs_clk = 0;
        bcc_dfs(0, 0); //搜索开始的点为0
```

```
        }
} G;
```

- $u\ is\ a\ cut \Leftrightarrow \exists v, < u, v > \in T \wedge Min[v] == d[u]$
- $pre\ is\ a\ bridge \Leftrightarrow Min[u] == d[u] - 1 \wedge \forall < u, v > \in T, Min[v] == d[u]$

2.5 Strongly Connected Components

tarjan 的求强连通分量算法，还是使用 dfn 和 low，只用树边和后向边更新 low，不管 cross-edge。
u 是其所在 scc 的第一个被搜到的点，当且仅当 $low[u] == dfn[u]$
scc 编号从 1 开始。

```cpp
struct Graph {
    struct { int v, next; } e[MAXM];
    int head[MAXN], cnt_edge;
    void add_edge(int u, int v) {
        e[cnt_edge] = {v, head[u]};
        head[u] = cnt_edge++;
    }
    void init() {
        memset(head, 0xff, sizeof(head));
        cnt_edge = 0;
    }

    int dfn[MAXN], low[MAXN], scc_clock;
    int scc_stk[MAXN], cnt_stk;
    int sccno[MAXN], cnt_scc;

    void scc_dfs(int u) {
        dfn[u] = low[u] = ++scc_clock;
        scc_stk[cnt_stk++] = u;

        for(int i = head[u]; i != -1; i = e[i].next) {
            if(!dfn[e[i].v]) {
                scc_dfs(e[i].v);
                low[u] = std::min(low[u], low[e[i].v]);
            }
            else if(!sccno[e[i].v])
                low[u] = std::min(low[u], dfn[e[i].v]);
        }

        if(low[u] == dfn[u]) {
            cnt_scc++;
            int x;
            do {
                x = scc_stk[--cnt_stk];
                sccno[x] = cnt_scc;
            } while(x != u);
        }
    }
```

```
    void find_scc(int n) {
        memset(sccno, 0, sizeof(sccno));
        memset(dfn, 0, sizeof(dfn));
        cnt_stk = cnt_scc = scc_clock = 0;
        for(int i = 0; i < n; i++) if(!dfn[i]) scc_dfs(i);
    }
};
```

3 2D Geometry

3.1 Point

叉积：$(1,0)\times(0,1)=1$

```
const double EPS = 1e-8;
const double PI = acos(-1);
int dcmp(double x) {
    if(fabs(x) <= EPS) return 0;
    return x < 0 ? -1 : 1;
}
struct Point {
    double x, y;
};
Point operator - (Point a, Point b) { return {a.x - b.x, a.y - b.y}; }
double det(Point a, Point b) { return a.x * b.y - a.y * b.x; }
double dot(Point a, Point b) { return a.x * b.x + a.y * b.y; }

Point operator + (Point a, Point b) { return {a.x + b.x, a.y + b.y}; }
Point operator * (double c, Point p) { return {c * p.x, c * p.y}; }
Point operator / (Point p, double c) { return {p.x / c, p.y / c}; }

double sqr(double x) { return x * x; }
double len(Point a) { return sqrt(sqr(a.x) + sqr(a.y)); }
double ang(Point a, Point b) { return acos(dot(a, b) / len(a) / len(b)); }

Point rotate(Point p, double A) {//逆时针旋转
    double tx = p.x, ty = p.y;
    return {tx * cos(A) - ty * sin(A), tx * sin(A) + ty * cos(A)};
}
```

3.2 Segment

```
Point line_intersection(Point p, Point v, Point q, Point w) {//v, w是向量
    Point u = p - q;
    double t = det(w, u) / det(v, w);
    return p + t * v;
}
double dis_to_line(Point p, Point a, Point b) {
    Point v1 = b - a, v2 = p - a;
    return fabs(det(v1, v2)) / len(v1);
}
double dis_to_seg(Point p, Point a, Point b) {
```

```
    if(a == b) return len(p - a);
    Point v1 = b - a, v2 = p - a, v3 = p - b;
    if(dcmp(dot(v1, v2)) < 0) return len(v2);
    else if(dcmp(dot(v1, v3)) > 0) return len(v3);
    else return fabs(det(v1, v2)) / len(v1);
}
Point line_projection(Point p, Point a, Point b) {
    Point v = b - a;
    return a + (dot(v, p - a) / dot(v, v)) * v;
}
bool on_seg(Point p, Point a, Point b) {
    Point v1 = a - p, v2 = b - p;
    return dcmp(det(v1, v2)) == 0 && dcmp(dot(v1, v2)) <= 0;//包含端点
    //return dcmp(det(v1, v2)) == 0 && dcmp(dot(v1, v2)) < 0;//不包含端点
}
bool seg_proper_intersect(Point a1, Point a2, Point b1, Point b2) {
    Point va = a2 - a1, vb = b2 - b1;
    double c1 = det(va, b1 - a1);
    double c2 = det(va, b2 - a1);
    double c3 = det(vb, a1 - b1);
    double c4 = det(vb, a2 - b1);
    return dcmp(c1) * dcmp(c2) < 0 && dcmp(c3) * dcmp(c4) < 0;
}
```

3.3 Circle

```
int circle_cross_seg(Point a, Point b, Point o, double r, Point ret[]) {//返回交点个数
    double x0 = o.x, y0 = o.y;
    double x1 = a.x, y1 = a.y;
    double x2 = b.x, y2 = b.y;
    double dx = x2 - x1, dy = y2 - y1;
    double A = dx * dx + dy * dy;
    double B = 2 * dx * (x1 - x0) + 2 * dy * (y1 - y0);
    double C = sqr(x1 - x0) + sqr(y1 - y0) - sqr(r);
    double delta = B * B - 4 * A * C;
    int num = 0;
    if(dcmp(delta) >= 0) {
        double t1 = (-B - sqrt(delta)) / (2 * A);
        double t2 = (-B + sqrt(delta)) / (2 * A);
        if(dcmp(t1 - 1) <= 0 && dcmp(t1) >= 0) ret[num++] = {x1 + t1 * dx, y1 + t1 * dy};
```

```
        if(dcmp(t2 - 1) <= 0 && dcmp(t2) >= 0) ret[num++] = {x1 + t2 * dx, y1 + t2 * dy};
    }
    return num;
}
struct Circle {
    Point c;
    double r;
    Point point(double rad) {//通过圆心角求坐标
        return (Point){c.x + r * cos(rad), c.y + r * sin(rad)};
    }
};
double angle(Point v) { return atan2(v.y, v.x); }
int CircleIntersection(Circle c1, Circle c2, Point res[]) {
    double d = len(c1.c - c2.c);
    if(dcmp(d) == 0) {
        if(dcmp(c1.r - c2.r) == 0) return -1;//两圆重合
        return 0;
    }
    if(dcmp(c1.r + c2.r - d) < 0) return 0;
    if(dcmp(fabs(c1.r - c2.r) - d) > 0) return 0;

    double a = angle(c2.c - c1.c);
    double da = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2 * c1.r * d));

    Point p1 = c1.point(a - da), p2 = c1.point(a + da);

    res[0] = p1;
    if(p1 == p2) return 1;
    res[1] = p2;
    return 2;
}
```

3.4 Simulate Anneal

```
double rand01() { return (rand() % 10001) / 10000.0; }
Point anneal(Point cur) {
    const double T = 100000, STEP = 0.98;//while大约会执行1500次
    double t = T;
    double cur_val = f(cur);
    while(t > EPS) {
        double tmp = 2 * PI * rand01();
        Point nxt = {cur.x + t * sin(tmp), cur.y + t * cos(tmp)};
```

```

        double nxt_val = f(nxt);
        double dif = cur_val - nxt_val;//最小值
        if(dif >= 0 || exp(dif / t) >= rand01()) cur = nxt, cur_val = nxt_val;
        t *= STEP;
    }
    for(int cnt = 0; cnt < 1000; cnt++) {
        t = 0.01 * rand01();
        double tmp = 2 * PI * rand01();
        Point nxt = {cur.x + t * sin(tmp), cur.y + t * cos(tmp)};

        double nxt_val = f(nxt);
        if(nxt_val < cur_val) cur = nxt, cur_val = nxt_val;//最小值
    }
    return cur;
}
```

4 String algorithm

4.1 AC Automaton

- 状态的 fail 与 KMP 相似，指向与当前状态的后缀相等的最长前缀

```
struct ACA {
    struct Node {
        int next[26], fail;
        int end;
        void init() {
            memset(next, 0, sizeof(next));
            fail = end = 0;
        }
    } t[MAXN];
    int cnt_node;
    int new_node() {
        t[cnt_node].init();
        return cnt_node++;
    }

    int root;
    void init() {
        cnt_node = 1;
        root = new_node();
    }
    void insert(char *str, int len) {
        int cur = root;
        for(int i = 0; i < len; i++) {
            if(t[cur].next[str[i] - 'a'] == 0) {
                t[cur].next[str[i] - 'a'] = new_node();
            }
            cur = t[cur].next[str[i] - 'a'];
        }
        t[cur].end++;
    }
    void build() {
        std::queue<int> que;
        t[root].fail = root;
        for(int i = 0; i < 26; i++) {
            if(t[root].next[i] == 0) t[root].next[i] = root;
            else {
                t[t[root].next[i]].fail = root;
                que.push(t[root].next[i]);
```

```
            }
        }
        while(!que.empty()) {
            int u = que.front(); que.pop();
            for(int i = 0; i < 26; i++) {
                if(t[u].next[i] == 0) t[u].next[i] = t[t[u].fail].next[i];
                else {
                    t[t[u].next[i]].fail = t[t[u].fail].next[i];
                    que.push(t[u].next[i]);
                }
            }
        }
    }
} ac;
```

4.2 Suffix Automaton

需要注意的知识点:

- 每个状态对应一个右端点 right 集合（等价类），串的长度为 [min, max]，短了集合变大，长了集合变小
- parent 树，根为全集，越往下串长越长，max(父亲) + 1 == min(儿子)
- 子串是后缀的前缀，SAM 经常被用来判断子串/统计子串个数

```
struct Node {
    int go[26], par;
    int val;
    void init(int val_) {
        memset(this, 0, sizeof(*this));
        val = val_;
    }
};
struct SAM {
    Node t[MAXN * 2];
    int cnt_node;
    int new_node(int val) {
        t[cnt_node].init(val);
        return cnt_node++;
    }

    int root, last;
    void init() {
        cnt_node = 1;
        root = last = new_node(0);
```

```
    }
    void extend(int w) {
        int p = last;
        int np = new_node(t[last].val + 1);
        //t[np].idx = i;
        while(p != 0 && t[p].go[w] == 0) {
            t[p].go[w] = np;
            p = t[p].par;
        }
        if(p == 0) t[np].par = root;
        else {
            int q = t[p].go[w];
            if(t[q].val == t[p].val + 1) t[np].par = q;
            else {
                int nq = new_node(t[p].val + 1);
                memcpy(t[nq].go, t[q].go, sizeof(t[q].go));
                t[nq].par = t[q].par;
                t[q].par = nq;
                t[np].par = nq;
                while(p != 0 && t[p].go[w] == q) {
                    t[p].go[w] = nq;
                    p = t[p].par;
                }
            }
        }
        last = np;
    }
} sam;
```

4.3 Suffix Array

- 在 da 函数中，rank[i] 的含义是后缀 i 对应的“值”，两个后缀的前 2^k 个字符相同时，对应的 rank 也相同。计算完 height 后，rank[i] 代表后缀 i 在 sa[] 中的位置。
- height[i] 代表 sa[i] 与 sa[i- 1] 的最长公共前缀

```
void count(int *a, int *b, int *val, int n, int m) {//[0, n) [0, m]
    static int cnt[MAXN];
    for(int i = 0; i <= m; i++) cnt[i] = 0;
    for(int i = 0; i < n; i++) cnt[val[a[i]]]++;
    for(int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];
    for(int i = n - 1; i >= 0; i--) b[--cnt[val[a[i]]]] = a[i];
}
void da(int *str, int *sa, int *rank, int n, int m) {
```

```
    static int sa_[MAXN];
    for(int i = 0; i < n; i++) sa_[i] = i;
    for(int i = 0; i < n; i++) rank[i] = str[i];
    count(sa_, sa, rank, n, m);
    for(int k = 0; (1 << k) < n; k++) {
        int l = (1 << k), p = 0;
        for(int i = n - l; i < n; i++) sa_[p++] = i;
        for(int i = 0; i < n; i++) if(sa[i] >= l) sa_[p++] = sa[i] - l;
        count(sa_, sa, rank, n, m);
        int *rank_ = sa_;
        m = 0;
        for(int i = 0; i < n; i++) {
            if(i > 0) {
                if(rank[sa[i]] != rank[sa[i - 1]]) m++;
                else if(sa[i - 1] + l >= n) m++;
                else if(rank[sa[i] + l] != rank[sa[i - 1] + l]) m++;
            }
            rank_[sa[i]] = m;
        }
        for(int i = 0; i < n; i++) rank[i] = rank_[i];
        if(m >= n - 1) break;
    }
}
void calc_height(int *str, int *sa, int *rank, int *height, int n) {
    str[n] = -1; //字符串尾需有结束符
    for(int i = 0; i < n; i++) rank[sa[i]] = i;
    height[0] = 0;
    for(int i = 0, k = 0; i < n; i++) {
        if(k > 0) k--;
        if(rank[i] != 0) {
            while(str[i + k] == str[sa[rank[i] - 1] + k]) k++;
            height[rank[i]] = k;
        }
    }
}
```

4.4 Kmp

```
void getNext(string &P, int next[]) {
    next[0] = 0;
    for(int pre = 0, i = 1; i < P.length(); i++) {
        while(pre > 0 && P[pre] != P[i]) pre = next[pre - 1];
        if(P[pre] == P[i]) pre++;
```

```
        next[i] = pre;
    }
}
void kmpMatch(string &P, string &T, int next[], int kmp[]) {
    for(int pre = 0, i = 0; i < T.length(); i++) {
        while(pre > 0 && P[pre] != T[i]) pre = next[pre - 1];
        if(P[pre] == T[i]) pre++;
        kmp[i] = pre;
        if(pre == P.length()) pre = next[pre - 1];
    }
}
```

4.5 Manacher

```
void manacher(char *str, int len, int p[]) {
    p[0] = 1;
    int key = 0;
    for(int i = 1; i < len; i++) {
        int k = (key + p[key] == i) ? 1 : min(key + p[key] - i, p[2 * key - i]);
        while(i - k >= 0 && i + k < len && str[i + k] == str[i - k]) k++;
        p[i] = k;
        if(i + p[i] > key + p[key]) key = i;
    }
}
```

4.6 Z Algorithm

```
void calc_z(char *str, int len, int *z) {
    int l = 0, r = 0;
    for(int i = 1; i < len; i++) {
        z[i] = 0;
        if(i <= r) z[i] = std::min(z[i - l], r - i + 1);
        while(i + z[i] < len && str[i + z[i]] == str[z[i]]) z[i]++;
        if(i + z[i] - 1 > r) { l = i; r = i + z[i] - 1; }
    }
}
```

4.7 Extend Kmp

```
void getExNext(string &pat, vector<int> &next) {
    int len = pat.length();
    next.resize(len);
```

```
    next[0] = len;
    for(int a = 1, p = 1, i = 1; i < len; i++) {
        if(p <= i) a = p = i;

        int k = min(next[i - a], p - i);
        while(i + k < len && pat[i + k] == pat[k]) k++;
        next[i] = k;

        if(i + k > p) p = i + k, a = i;
    }
}
void exKmpMatch(string &s, string &pat, vector<int> &next, vector<int> &lcp) {
    //getExNext(pat, next);
    int len_s = s.length(), len_p = pat.length();
    lcp.resize(len_s);
    for(int a = 0, p = 0, i = 0; i < len_s; i++) {
        if(p <= i) a = p = i;

        int k = min(next[i - a], p - i);
        while(k < len_p && i + k < len_s && pat[k] == s[i + k]) k++;
        lcp[i] = k;

        if(i + k > p) p = i + k, a = i;
    }
}
```

5 Math

5.1 Prime Number

线性筛，每个合数只被其最小的素因子筛掉。

$n = \prod_{i=1}^{k} p_i^{w_i}$

$\varphi(n) = n \times \prod_{i=1}^{k} (1 - \frac{1}{p_i})$

```
bool vis[MAXN];
int pri[MAXN], cnt_pri;
int phi[MAXN];

void prime() {
    phi[1] = 1;
    for(int i = 2; i < MAXN; i++) {
        if(!vis[i]) {
            pri[cnt_pri++] = i;
            phi[i] = i - 1;
        }
        for(int j = 0; j < cnt_pri; j++) {
            if((long long)i * pri[j] >= MAXN) {
                break;
            }
            vis[i * pri[j]] = true;
            if(i % pri[j] != 0) {
                phi[i * pri[j]] = phi[i] * phi[pri[j]];
            }
            else {
                phi[i * pri[j]] = pri[j] * phi[i];
                break;
            }
        }
    }
}
```

5.2 Matrix Multiplication

```
void mat_mul(int res[][MAXN], int a[][MAXN], int b[][MAXN], int n, int m, int k) {
    static int tmp[MAXN][MAXN];
    for(int i = 0; i < n; i++) for(int j = 0; j < k; j++) {
        long long sum = 0;
        for(int x = 0; x < m; x++) sum += ((long long)a[i][x] * b[x][j]) % MOD;
        tmp[i][j] = sum % MOD;
    }
```

```
    for(int i = 0; i < n; i++) for(int j = 0; j < k; j++) res[i][j] = tmp[i][j];
}
void mat_pow(int res[][MAXN], int a[][MAXN], int x, int n) {
    static int cur[MAXN][MAXN];
    for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) cur[i][j] = a[i][j];
    for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) res[i][j] = (i == j) ? 1 : 0;
    while(x) {
        if(x & 1) mat_mul(res, res, cur, n, n, n);
        mat_mul(cur, cur, cur, n, n, n);
        x >>= 1;
    }
}
```

5.3 Number Theoretic Transform

- 离散傅立叶变换（DFT）：多项式 $(a_0, a_1, a_2, \cdots, a_{n-1})$ 在 $(w^0, w^1, w^2, \cdots, w^{n-1})$ 处的取值，记为 $y = DFT(A)$
- 多项式的系数/点值表示
- 数论变换：$w = g^{\frac{p-1}{n}}$ 的 FFT，p 是质数，$n \mid (p-1)$，g 是 Z_p^* 的生成元，且 $g^n = 1$

```
//NTT函数，要求n为2的幂次，且n <= MAXN
void ntt(int a[], int n, bool reverse) {
    static int buf[MAXN];
    int lg = 0;
    while((1 << lg) < n) lg++;
    for(int i = 0; i < n; i++) {
        int pos = 0;
        for(int j = 0; j < lg; j++) pos |= ((i >> j & 1) << (lg - 1 - j));
        buf[pos] = a[i];
    }
    for(int i = 0; i < n; i++) a[i] = buf[i];

    for(int l = 1; l < n; l <<= 1) {
        int w = powmod(3, (MOD - 1) / (2 * l));
        if(reverse) w = powmod(w, MOD - 2);
        for(int i = 0; i < n; i += 2 * l) {
            int cur = 1;
            for(int j = 0; j < l; j++) {
                int t = mulmod(cur, a[i + j + l]);
                int u = a[i + j];
                a[i + j] = addmod(u, t);
                a[i + j + l] = addmod(u, MOD - t);
```

```
                cur = mulmod(cur, w);
            }
        }
    }
    if(reverse) {
        int inv_n = powmod(n, MOD - 2);
        for(int i = 0; i < n; i++) a[i] = mulmod(a[i], inv_n);
    }
}
void polymul(int res[], int a[], int b[], int n) {
    int m = 1;
    while(m < n) m <<= 1;
    m <<= 1;
    ntt(a, m, false);
    ntt(b, m, false);
    for(int i = 0; i < m; i++) res[i] = mulmod(a[i], b[i]);
    ntt(res, m, true);
}
```

5.4 Extend Euclid

```
int extend_gcd(int a, int b, int &x, int &y) {
    int ans;
    if(b == 0) x = 1, y = 0, ans = a;
    else {
        ans = extend_gcd(b, a % b, y, x);
        y -= (a / b) * x;
    }
    return ans;
}
```

6 Other

6.1 Checker

```
if !(g++ g.cpp -o g && g++ --std=c++11 c1.cpp -o c1 && g++ c2.cpp -o c2)
then
    exit
fi

echo read
read N

i=1
while [ $i -le $N ];
do
    echo $i
    echo $i | ./g | ./c1 > a
    echo $i | ./g | ./c2 > b
    if ! diff a b
    then
        echo $i | ./g > $i.txt
        cat a >> $i.txt
        cat b >> $i.txt
    fi
    i=$((i+1))
done
```

6.2 Fast IO

```
#include <cstdio>
#include <cmath>
template <typename T>
bool nxtInt(T &res) {
    char c;
    bool negative = false;
    while(c=getchar(), c!='-' && !('0'<=c && c<='9'))
        if(c == EOF) return false;
    if(c == '-') negative = true, c = getchar();
    res = 0;
    while('0'<=c && c<='9') res *= 10, res += c - '0', c = getchar();
    if(negative) res *= -1;
    return true;
}
```

```

template <typename T>
void prtInt(T x) {
    if(x < 0) putchar('-'), x *= -1;
    if(x > 9) prtInt(x / 10);
    putchar(x % 10 + '0');
}

template <typename T>
bool nxtNum(T &res) {
    char c;
    bool negative = false;
    while(c=getchar(), c!='.' && c!='-' && !('0'<=c && c<='9'))
        if(c == EOF) return false;
    if(c == '-') negative = true, c = getchar();
    res = 0;
    long long cnt = 0;
    while(c == '.' || ('0'<=c && c<='9')) {
        cnt *= 10;
        if(c == '.') cnt = 1;
        else res *= 10, res += c - '0';
        c = getchar();
    }
    if(negative) res *= -1;
    if(cnt > 0) res *= 1.0/cnt;
    return true;
}

template <typename T>
void prtNum(T x, long precision) {
    if(x < 0) putchar('-'), x *= -1;
    long long flr = floor(x);
    prtInt(flr);
    x -= flr;
    x *= pow(10, precision);
    flr = round(x);
    putchar('.');
    long long cnt = 0, t = 1;
    while(flr >= t) cnt++, t *= 10;
    for(long i=0; i<precision-cnt; i++) putchar('0');
    prtInt(flr);
}
bool nxtChar(char &c) {
```

```
    do {
        c = getchar();
    } while(c == ' ' || c == '\n');
    return c != EOF;
}
```