

# Standard Code Library

2019 年 1 月 2 日



# 目录

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Data Structure</b>                   | <b>1</b>  |
| 1.1      | Basic ZKW . . . . .                     | 1         |
| 1.2      | Segment Tree . . . . .                  | 2         |
| 1.3      | Persistent Segment Tree . . . . .       | 4         |
| 1.4      | Binary Indexed Tree . . . . .           | 6         |
| 1.5      | Treap . . . . .                         | 7         |
| 1.6      | Splay Tree . . . . .                    | 10        |
| <b>2</b> | <b>Graph</b>                            | <b>12</b> |
| 2.1      | Maximum Flow . . . . .                  | 12        |
| 2.2      | Min Cost Flow . . . . .                 | 14        |
| 2.3      | Matching and Covers . . . . .           | 16        |
| 2.4      | Cut and Bridge and BCC . . . . .        | 18        |
| 2.5      | Strongly Connected Components . . . . . | 20        |
| <b>3</b> | <b>2D Geometry</b>                      | <b>22</b> |
| 3.1      | Point . . . . .                         | 22        |
| 3.2      | Segment . . . . .                       | 22        |
| 3.3      | Circle . . . . .                        | 23        |
| 3.4      | Simulate Anneal . . . . .               | 24        |
| <b>4</b> | <b>String algorithm</b>                 | <b>26</b> |
| 4.1      | AC Automaton . . . . .                  | 26        |
| 4.2      | Suffix Automaton . . . . .              | 27        |
| 4.3      | Suffix Array . . . . .                  | 28        |
| 4.4      | Kmp . . . . .                           | 29        |
| 4.5      | Manacher . . . . .                      | 30        |
| 4.6      | Z Algorithm . . . . .                   | 30        |
| 4.7      | Extend Kmp . . . . .                    | 30        |
| <b>5</b> | <b>Math</b>                             | <b>32</b> |
| 5.1      | Prime Number . . . . .                  | 32        |
| 5.2      | Matrix Multiplication . . . . .         | 32        |
| 5.3      | Number Theoretic Transform . . . . .    | 32        |
| 5.4      | Extend Euclid . . . . .                 | 33        |
| <b>6</b> | <b>Other</b>                            | <b>35</b> |
| 6.1      | Checker . . . . .                       | 35        |
| 6.2      | Fast IO . . . . .                       | 35        |

# 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

---

```

1  template <typename T>
2  struct ZKW {
3      T tree[MAXN * 2];
4      int n;
5      void build() {
6          for(int p = n - 1; p >= 1; p--) tree[p] = op(tree[p << 1], tree[p << 1 |
              ↪ 1]);
7      }
8      void update(int p, T val) { // a[p] = val
9          tree[p += n] = val;
10         for(p >>= 1; p > 0; p >>= 1) tree[p] = op(tree[p << 1], tree[p << 1 | 1]);
11     }
12     T query(int l, int r) { // [l, r]
13         l += n; r += n;
14         T res = tree[r];
15         for(; l < r; l >>= 1, r >>= 1) {
16             if(l & 1) res = op(res, tree[l++]);
17             if(r & 1) res = op(res, tree[--r]);
18         }
19         return res;
20     }
21 };

```

---

## 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。

---

```

1  class SGT {
2      struct Node {
3          ll min_, s;
4      } tree[maxn * 4];
5      void merge(Node &res, Node &l, Node &r) {
6          res.min_ = std::min(l.min_ + l.s, r.min_ + r.s);
7          res.s = 0;
8      }
9      void push_down(Node &root, Node &l, Node &r) {
10         root.min_ += root.s;
11         l.s += root.s;
12         r.s += root.s;
13         root.s = 0;
14     }
15 public:
16     //root节点对应的线段是[left, right]
17     void add(int root, int left, int right, int l, int r, ll d) {
18         if(right < l || r < left) return;
19         if(l <= left && right <= r) tree[root].s += d;
20         else {
21             int mid = (left + right) / 2;
22             push_down(tree[root], tree[root << 1], tree[root << 1 | 1]);
23             add(root << 1, left, mid, l, r, d);
24             add(root << 1 | 1, mid + 1, right, l, r, d);
25             merge(tree[root], tree[root << 1], tree[root << 1 | 1]);
26         }
27     }
28     ll query_min(int root, int left, int right, int l, int r) {
29         if(l <= left && right <= r) return tree[root].min_ + tree[root].s;
30         else {
31             int mid = (left + right) / 2;
32             push_down(tree[root], tree[root << 1], tree[root << 1 | 1]);
33             ll L = LONG_LONG_MAX, R = LONG_LONG_MAX;
34             if(l <= mid) L = query_min(root << 1, left, mid, l, r);

```

```
35         if(mid + 1 <= r) R = query_min(root << 1 | 1, mid + 1, right, l, r);
36         return std::min(L, R);
37     }
38 }
39 } 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。

---

```

1 struct PSGT {
2     struct Node {
3         int sum;
4         int lc, rc;
5     } tree[MAXN * 20];
6     int cnt_node;
7     int copy_of(int u) {
8         tree[cnt_node] = tree[u];
9         return cnt_node++;
10    }
11    void init() {
12        cnt_node = 1;
13    }
14    int build(int l, int r) {
15        int root = cnt_node++;
16        if(l == r) {
17            tree[root].lc = tree[root].rc = -1;
18            tree[root].sum = 0;
19        }
20        else {
21            int mid = (l + r) / 2;
22            int lc = build(l, mid), rc = build(mid + 1, r);
23            tree[root].lc = lc;
24            tree[root].rc = rc;
25            tree[root].sum = tree[lc].sum + tree[rc].sum;
26        }
27        return root;
28    }
29    int query(int root, int left, int right, int l, int r) {
30        if(l <= left && right <= r) return tree[root].sum;
31        if(r < left || right < l) return 0;
32        int mid = (left + right) / 2;
33        return query(tree[root].lc, left, mid, l, r) + query(tree[root].rc, mid +
34            ↪1, right, l, r);
35    }

```

```
35     int modify(int root, int left, int right, int p, int val) {
36         int res = copy_of(root);
37         if(left == p && right == p) tree[res].sum += val;
38         else {
39             int mid = (left + right) / 2;
40             if(p <= mid) tree[res].lc = modify(tree[root].lc, left, mid, p, val);
41             else tree[res].rc = modify(tree[root].rc, mid + 1, right, p, val);
42             tree[res].sum = tree[tree[res].lc].sum + tree[tree[res].rc].sum;
43         }
44         return res;
45     }
46 } 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

---

```
1  template<typename T>
2  struct BIT {
3      static const int maxn = 1e6;
4      T tree[maxn];
5      int n; //[1, n]
6      std::function<T(T, T)> op;
7      inline int lowbit(int i) { return i & (-i); }
8
9      void init(int n, const T &ID, const std::function<T(T, T)> &op) {
10         this->n = n, this->op = op;
11         fill(tree, tree + n + 1, ID);
12     }
13     T query_prefix(int r) { //[1, r]
14         T res = tree[r];
15         for(r -= lowbit(r); r > 0; r -= lowbit(r)) res = op(res, tree[r]);
16         return res;
17     }
18     void update(int p, const T &val) { //a[p] = op(a[p], val)
19         while(p <= n) tree[p] = op(tree[p], val), p += lowbit(p);
20     }
21 };
```

---

## 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 \*\*\*

---

```

1  struct treapNode {
2      T val;
3      int s, p;
4      treapNode* c[2];
5  } buf[MAXN];
6  int cnt_buf;
7  void rotate(treapNode* &root, int d) {
8      treapNode *y = root->c[d];
9
10     y->s = root->s;
11     root->s = size(root->c[d ^ 1]) + size(y->c[d ^ 1]) + 1;
12
13     root->c[d] = y->c[d ^ 1], y->c[d ^ 1] = root, root = y;
14 }
15
16 int size(treapNode* root) { return root == nullptr ? 0 : root->s; }
17
18 void insert(treapNode* &root, const T &val) {
19     if(root == nullptr) {
20         root = &buf[cnt_buf++];
21         root->val = val;
22         root->s = 1;
23         root->p = rand();
24         root->c[0] = root->c[1] = nullptr;
25     }
26     else {
27         int d = val < root->val ? 0 : 1;
28         insert(root->c[d], val);
29         ++root->s;
30         if(root->c[d]->p < root->p) rotate(root, d);
31     }
32 }
33
34 bool remove(treapNode* &root, const T &val) {
35     if(root == nullptr) return false;
36     if(root->val == val) {
37         for(int i = 0; i < 2; i++) if(root->c[i] == nullptr) {

```

```

38         root = root->c[i ^ 1];
39         return true;
40     }
41     int d = root->c[0]->p < root->c[1]->p ? 0 : 1;
42     rotate(root, d);
43     remove(root->c[d ^ 1], val);
44     --root->s;
45     return true;
46 }
47 else if(remove(root->c[val < root->val ? 0 : 1], val)) {
48     --root->s;
49     return true;
50 }
51 return false;
52 }
53 T& select(treapNode* root, int k) {
54     while(root != nullptr) {
55         int cur = size(root->c[0]);
56         if(cur == k) return root->val;
57         root = root->c[k < cur ? 0 : 1];
58         if(k > cur) k -= ++cur;
59     }
60 }
61 int lower_bound(treapNode* root, T val) {
62     int ans = 0;
63     while(root != nullptr) {
64         if(root->val >= val) root = root->c[0];
65         else {
66             ans += size(root->c[0]) + 1;
67             root = root->c[1];
68         }
69     }
70     return ans;
71 }
72 int upper_bound(treapNode* root, T val) {
73     int ans = 0;
74     while(root != nullptr) {
75         if(root->val > val) root = root->c[0];
76         else {
77             ans += size(root->c[0]) + 1;
78             root = root->c[1];
79         }
80     }

```

```
81     return ans;
82 }
83 bool contain(treapNode* root, const T &val) {
84     while(root != nullptr && root->val != val) {
85         if(val < root->val) root = root->c[0];
86         else root = root->c[1];
87     }
88     return root != nullptr;
89 }
```

---

## 1.6 Splay Tree

可分割/合并序列可用 `std::rope` 完成。

---

```

1  template<typename T>
2  struct splayNode {
3      T val;
4      int s;
5      splayNode<T>* c[2];
6      splayNode(const T &val) : val(val) {}
7
8      static std::allocator<splayNode<T>> alloc;
9      static void rotate(splayNode<T>* &root, int d) {
10         auto y = root->c[d];
11
12         y->s = root->s;
13         root->s = 1 + size(root->c[d ^ 1]) + size(y->c[d ^ 1]);
14
15         root->c[d] = y->c[d ^ 1], y->c[d ^ 1] = root, root = y;
16     }
17     static void splay_kth(splayNode<T>* &root, int k) {
18         if(root == nullptr) return;
19         splayNode<T> *p[2], **p_[2] = { &p[0], &p[1] };
20         std::stack<splayNode<T>*> s[2];
21         while(true) {
22             int cur = size(root->c[0]);
23             int d = k < cur ? 0 : 1;
24             if(k == cur || root->c[d] == nullptr) break;
25             if(d == 0 && k < size(root->c[d]->c[d]) && root->c[d]->c[d] != nullptr)
26                 rotate(root, d);
27             if(d == 1 && k >= size(root) - size(root->c[d]->c[d]) && root->c[d]->c[
28                 ↪d] != nullptr)
29                 rotate(root, d);
30             *p_[d ^ 1] = root, p_[d ^ 1] = &root->c[d], s[d ^ 1].push(root);
31             if(d == 1) k -= size(root->c[0]) + 1;
32             root = root->c[d];
33         }
34         for(int i = 0; i < 2; i++) {
35             *p_[i] = root->c[i], root->c[i] = p[i];
36             while(!s[i].empty()) s[i].top()->s = 1 + size(s[i].top()->c[0]) + size(
37                 ↪s[i].top()->c[1]), s[i].pop();
38         }
39         root->s = 1 + size(root->c[0]) + size(root->c[1]);
40     }

```

```

39
40     friend int size(splayNode<T>* root) { return root == nullptr ? 0 : root->s; }
41     friend void build(splayNode<T>* &root, T* a, int l, int r) {
42         if(l > r) { root = nullptr; return; }
43         int mid = (l + r) / 2;
44         root = alloc.allocate(1), alloc.construct(root, a[mid]);
45         build(root->c[0], a, l, mid - 1), build(root->c[1], a, mid + 1, r);
46         root->s = 1 + size(root->c[0]) + size(root->c[1]);
47     }
48     friend void split(splayNode<T>* root, int k, splayNode<T>* &left, splayNode<T>*
49         ↪ &right) {
50         if(k >= size(root)) left = root, right = nullptr;
51         else {
52             splay_kth(root, k);
53             left = root->c[0], right = root;
54             root->s -= size(root->c[0]), root->c[0] = nullptr;
55         }
56     }
57     friend void attach(splayNode<T>* &left, splayNode<T>* right) {
58         if(left == nullptr) left = right;
59         else splay_kth(left, size(left) - 1), left->c[1] = right, left->s += size(
60             ↪ right);
61     }
62     friend void clear(splayNode<T>* &root) {
63         std::stack<splayNode<T>*> s;
64         if(root != nullptr) s.push(root);
65         while(!s.empty()) {
66             auto x = s.top(); s.pop();
67             if(x->c[0] != nullptr) s.push(x->c[0]);
68             if(x->c[1] != nullptr) s.push(x->c[1]);
69             alloc.destroy(x), alloc.deallocate(x, 1);
70         }
71         root = nullptr;
72     }
73 };
74
75 template<typename T>
76 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)

---

```

1 struct Graph {
2     struct { int v, cap, next; } e[MAXM];
3     int head[MAXN], cnt_edge;
4     void add_edge_(int u, int v, int cap) {
5         e[cnt_edge] = { v, cap, head[u] };
6         head[u] = cnt_edge++;
7     }
8     void add_edge(int u, int v, int cap) {
9         add_edge_(u, v, cap);
10        add_edge_(v, u, 0);
11    }
12    void init() {
13        memset(head, 0xff, sizeof(head));
14        cnt_edge = 0;
15    }
16
17    int level[MAXN], cur[MAXN];
18    bool dinic_bfs(int s, int t) {
19        memset(level, 0xff, sizeof(level));
20        std::queue<int> q;
21
22        level[s] = 0;
23        cur[s] = head[s];
24        q.push(s);
25        while(!q.empty()) {
26            int u = q.front(); q.pop();

```

```

27         if(u == t) return true;
28         for(int i = head[u]; i != -1; i = e[i].next) if(e[i].cap) {
29             if(level[e[i].v] == -1) {
30                 level[e[i].v] = level[u] + 1;
31                 cur[e[i].v] = head[e[i].v];
32                 q.push(e[i].v);
33             }
34         }
35     }
36     return false;
37 }
38
39 int dinic_dfs(int u, int cur_min, int t) {
40     if(u == t) return cur_min;
41     if(level[u] >= level[t]) return 0;
42
43     int res = 0;
44     for(int& i = cur[u]; cur_min && i != -1; i = e[i].next) if(e[i].cap) {
45         if(level[e[i].v] == level[u] + 1) {
46             int nxt_min = std::min(cur_min, e[i].cap);
47             int x = dinic_dfs(e[i].v, nxt_min, t);
48             cur_min -= x;
49             e[i].cap -= x;
50             e[i ^ 1].cap += x;
51             res += x;
52         }
53     }
54     return res;
55 }
56
57 int dinic(int s, int t) {
58     int res = 0;
59     while(dinic_bfs(s, t)) {
60         res += dinic_dfs(s, inf, t);
61     }
62     return res;
63 }
64 } G;

```

---



## 2.2 Min Cost Flow

把 EK 算法的 BFS 改成了 SPFA。

---

```

1  struct Graph {
2      struct { int v, cap, cost, next; } e[MAXN * MAXN];
3      int cnt_edge, head[MAXN];
4      void init() {
5          cnt_edge = 0;
6          memset(head, 0xff, sizeof(head));
7      }
8      void add_edge_(int u, int v, int cap, int cost) {
9          e[cnt_edge] = { v, cap, cost, head[u] };
10         head[u] = cnt_edge++;
11     }
12     void add_edge(int u, int v, int cap, int cost) {
13         add_edge_(u, v, cap, cost);
14         add_edge_(v, u, 0, -cost);
15     }
16     int dis[MAXN], pre[MAXN];
17     bool inque[MAXN];
18     bool spfa(int s, int t) {
19         memset(dis, 0x3f, sizeof(dis));
20         std::queue<int> que;
21         bool flag = false;
22
23         dis[s] = 0;
24         inque[s] = true;
25         que.push(s);
26         while(!que.empty()) {
27             int u = que.front(); que.pop();
28             if(u == t) flag = true;
29             inque[u] = false;
30             for(int i = head[u]; i != -1; i = e[i].next) if(e[i].cap > 0) {
31                 if(dis[u] + e[i].cost < dis[e[i].v]) {
32                     dis[e[i].v] = dis[u] + e[i].cost;
33                     pre[e[i].v] = i;
34                     if(!inque[e[i].v]) {
35                         inque[e[i].v] = true;
36                         que.push(e[i].v);
37                     }
38                 }
39             }
40         }
41     }

```

```
41     return flag;
42 }
43 int min_cost_flow(int s, int t, int &flow) {
44     int cost = 0;
45     flow = 0;
46     while(spfa(s, t)) {
47         int min_cap = 1e9;
48         for(int u = t; u != s; u = e[pre[u] ^ 1].v) if(min_cap > e[pre[u]].cap)
49             ↪ min_cap = e[pre[u]].cap;
50         for(int u = t; u != s; u = e[pre[u] ^ 1].v) {
51             cost += min_cap * e[pre[u]].cost;
52             e[pre[u]].cap -= min_cap;
53             e[pre[u] ^ 1].cap += min_cap;
54         }
55         flow += min_cap;
56     }
57     return cost;
58 } 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,  $|\text{Minimum Vertex Cover}| = |\text{Maximum Matching}|$
- 对于全部 (无向?) 图  $G = (V, E)$ ,  $|\text{Minimum Edge Cover}| = |V| - |\text{Maximum Matching}|$
- 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)。

---

```

1 //链式前向星
2 bool vis[MAXN];
3 int link[MAXN];
4 bool hungarian_dfs(int u) {
5     for(int i = head[u]; i != -1; i = e[i].next) if(!vis[e[i].v]) {
6         vis[e[i].v] = true;
7         if(link[e[i].v] == -1 || hungarian_dfs(link[e[i].v])) {
8             link[e[i].v] = u, link[u] = e[i].v;
9             return true;
10        }
11    }
12    return false;
13 }
14 int hungarian() {
15     memset(link, 0xff, sizeof(link));
16     int res = 0;

```

```
17     for(int i = 0; i < n_vertex; i++) if(link[i] == -1) {
18         memset(vis, 0, sizeof(vis));
19         if(hungarian_dfs(i)) res++;
20     }
21     return res;
22 }
```

---

## 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。

---

```

1  struct Graph {
2      struct { int v, next; } e[MAXM];
3      int head[MAXN], cnt_edge;
4      void add_edge(int u, int v) {
5          e[cnt_edge] = { v, head[u] };
6          head[u] = cnt_edge++;
7      }
8      void init() {
9          cnt_edge = 0;
10         memset(head, 0xff, sizeof(head));
11     }
12     int dfs_clk, dfn[MAXN];
13     bool cut[MAXN], bridge[MAXM];
14     int bcc_dfs(int u, int pre) {
15         dfn[u] = ++dfs_clk;
16         int cnt_child = 0;
17         int low_u = dfn[u];
18         for(int i = head[u]; i != -1; i = e[i].next) if(e[i].v != pre) {
19             int v = e[i].v;
20             if(dfn[v] == 0) {
21                 cnt_child++;
22                 int low_v = bcc_dfs(v, u);
23                 if(low_v >= dfn[u]) cut[u] = true;
24                 if(low_v >= dfn[v]) bridge[i] = bridge[i ^ 1] = true;
25                 low_u = std::min(low_u, low_v);
26             }
27             else {
28                 low_u = std::min(low_u, dfn[v]);
29             }
30         }
31         if(u == pre && cnt_child == 1) cut[u] = false;
32         return low_u;
33     }
34     void calc_bcc() {
35         memset(cut, 0, sizeof(cut));
36         memset(bridge, 0, sizeof(bridge));
37         memset(dfn, 0, sizeof(dfn));
38         dfs_clk = 0;
39         bcc_dfs(0, 0); //搜索开始的点为0

```

40        }

41   } **G**;

---

- *u is a cut*  $\Leftrightarrow \exists v, \langle u, v \rangle \in T \wedge \text{Min}[v] == d[u]$
- *pre is a bridge*  $\Leftrightarrow \text{Min}[u] == d[u] - 1 \wedge \forall \langle u, v \rangle \in T, \text{Min}[v] == d[u]$

## 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。

---

```

1  struct Graph {
2      struct { int v, next; } e[MAXM];
3      int head[MAXN], cnt_edge;
4      void add_edge(int u, int v) {
5          e[cnt_edge] = {v, head[u]};
6          head[u] = cnt_edge++;
7      }
8      void init() {
9          memset(head, 0xff, sizeof(head));
10         cnt_edge = 0;
11     }
12
13     int dfn[MAXN], low[MAXN], scc_clock;
14     int scc_stk[MAXN], cnt_stk;
15     int sccno[MAXN], cnt_scc;
16
17     void scc_dfs(int u) {
18         dfn[u] = low[u] = ++scc_clock;
19         scc_stk[cnt_stk++] = u;
20
21         for(int i = head[u]; i != -1; i = e[i].next) {
22             if(!dfn[e[i].v]) {
23                 scc_dfs(e[i].v);
24                 low[u] = std::min(low[u], low[e[i].v]);
25             }
26             else if(!sccno[e[i].v])
27                 low[u] = std::min(low[u], dfn[e[i].v]);
28         }
29
30         if(low[u] == dfn[u]) {
31             cnt_scc++;
32             int x;
33             do {
34                 x = scc_stk[--cnt_stk];
35                 sccno[x] = cnt_scc;
36             } while(x != u);
37         }
38     }

```

```
39     void find_scc(int n) {  
40         memset(sccno, 0, sizeof(sccno));  
41         memset(dfn, 0, sizeof(dfn));  
42         cnt_stk = cnt_scc = scc_clock = 0;  
43         for(int i = 0; i < n; i++) if(!dfn[i]) scc_dfs(i);  
44     }  
45 };
```

---



## 3 2D Geometry

### 3.1 Point

叉积:  $(1, 0) \times (0, 1) = 1$

---

```

1  const double EPS = 1e-8;
2  const double PI = acos(-1);
3  int dcmp(double x) {
4      if(fabs(x) <= EPS) return 0;
5      return x < 0 ? -1 : 1;
6  }
7  struct Point {
8      double x, y;
9  };
10 Point operator - (Point a, Point b) { return {a.x - b.x, a.y - b.y}; }
11 double det(Point a, Point b) { return a.x * b.y - a.y * b.x; }
12 double dot(Point a, Point b) { return a.x * b.x + a.y * b.y; }
13
14 Point operator + (Point a, Point b) { return {a.x + b.x, a.y + b.y}; }
15 Point operator * (double c, Point p) { return {c * p.x, c * p.y}; }
16 Point operator / (Point p, double c) { return {p.x / c, p.y / c}; }
17
18 double sqr(double x) { return x * x; }
19 double len(Point a) { return sqrt(sqr(a.x) + sqr(a.y)); }
20 double ang(Point a, Point b) { return acos(dot(a, b) / len(a) / len(b)); }
21
22 Point rotate(Point p, double A) { // 逆时针旋转
23     double tx = p.x, ty = p.y;
24     return {tx * cos(A) - ty * sin(A), tx * sin(A) + ty * cos(A)};
25 }

```

---

### 3.2 Segment

---

```

1  Point line_intersection(Point p, Point v, Point q, Point w) { // v, w 是向量
2      Point u = p - q;
3      double t = det(w, u) / det(v, w);
4      return p + t * v;
5  }
6  double dis_to_line(Point p, Point a, Point b) {
7      Point v1 = b - a, v2 = p - a;
8      return fabs(det(v1, v2)) / len(v1);
9  }
10 double dis_to_seg(Point p, Point a, Point b) {

```

```

11     if(a == b) return len(p - a);
12     Point v1 = b - a, v2 = p - a, v3 = p - b;
13     if(dcmp(dot(v1, v2)) < 0) return len(v2);
14     else if(dcmp(dot(v1, v3)) > 0) return len(v3);
15     else return fabs(det(v1, v2)) / len(v1);
16 }
17 Point line_projection(Point p, Point a, Point b) {
18     Point v = b - a;
19     return a + (dot(v, p - a) / dot(v, v)) * v;
20 }
21 bool on_seg(Point p, Point a, Point b) {
22     Point v1 = a - p, v2 = b - p;
23     return dcmp(det(v1, v2)) == 0 && dcmp(dot(v1, v2)) <= 0; //包含端点
24     //return dcmp(det(v1, v2)) == 0 && dcmp(dot(v1, v2)) < 0; //不包含端点
25 }
26 bool seg_proper_intersect(Point a1, Point a2, Point b1, Point b2) {
27     Point va = a2 - a1, vb = b2 - b1;
28     double c1 = det(va, b1 - a1);
29     double c2 = det(va, b2 - a1);
30     double c3 = det(vb, a1 - b1);
31     double c4 = det(vb, a2 - b1);
32     return dcmp(c1) * dcmp(c2) < 0 && dcmp(c3) * dcmp(c4) < 0;
33 }

```

---

### 3.3 Circle

```

1 int circle_cross_seg(Point a, Point b, Point o, double r, Point ret[]) { //返回交点
    ↪ 个数
2     double x0 = o.x, y0 = o.y;
3     double x1 = a.x, y1 = a.y;
4     double x2 = b.x, y2 = b.y;
5     double dx = x2 - x1, dy = y2 - y1;
6     double A = dx * dx + dy * dy;
7     double B = 2 * dx * (x1 - x0) + 2 * dy * (y1 - y0);
8     double C = sqr(x1 - x0) + sqr(y1 - y0) - sqr(r);
9     double delta = B * B - 4 * A * C;
10    int num = 0;
11    if(dcmp(delta) >= 0) {
12        double t1 = (-B - sqrt(delta)) / (2 * A);
13        double t2 = (-B + sqrt(delta)) / (2 * A);
14        if(dcmp(t1 - 1) <= 0 && dcmp(t1) >= 0) ret[num++] = {x1 + t1 * dx, y1 + t1
            ↪ * dy};

```

```

15         if(dcmp(t2 - 1) <= 0 && dcmp(t2) >= 0) ret[num++] = {x1 + t2 * dx, y1 + t2
           ↪ * dy};
16     }
17     return num;
18 }
19 struct Circle {
20     Point c;
21     double r;
22     Point point(double rad) { //通过圆心角求坐标
23         return (Point){c.x + r * cos(rad), c.y + r * sin(rad)};
24     }
25 };
26 double angle(Point v) { return atan2(v.y, v.x); }
27 int CircleIntersection(Circle c1, Circle c2, Point res[]) {
28     double d = len(c1.c - c2.c);
29     if(dcmp(d) == 0) {
30         if(dcmp(c1.r - c2.r) == 0) return -1; //两圆重合
31         return 0;
32     }
33     if(dcmp(c1.r + c2.r - d) < 0) return 0;
34     if(dcmp(fabs(c1.r - c2.r) - d) > 0) return 0;
35
36     double a = angle(c2.c - c1.c);
37     double da = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2 * c1.r * d));
38
39     Point p1 = c1.point(a - da), p2 = c1.point(a + da);
40
41     res[0] = p1;
42     if(p1 == p2) return 1;
43     res[1] = p2;
44     return 2;
45 }

```

---

### 3.4 Simulate Anneal

```

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

```

```
9
10     double nxt_val = f(nxt);
11     double dif = cur_val - nxt_val; //最小值
12     if(dif >= 0 || exp(dif / t) >= rand01()) cur = nxt, cur_val = nxt_val;
13     t *= STEP;
14 }
15 for(int cnt = 0; cnt < 1000; cnt++) {
16     t = 0.01 * rand01();
17     double tmp = 2 * PI * rand01();
18     Point nxt = {cur.x + t * sin(tmp), cur.y + t * cos(tmp)};
19
20     double nxt_val = f(nxt);
21     if(nxt_val < cur_val) cur = nxt, cur_val = nxt_val; //最小值
22 }
23 return cur;
24 }
```

---

## 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

---

```

1 struct ACA {
2     struct Node {
3         int next[26], fail;
4         int end;
5         void init() {
6             memset(next, 0, sizeof(next));
7             fail = end = 0;
8         }
9     } t[MAXN];
10    int cnt_node;
11    int new_node() {
12        t[cnt_node].init();
13        return cnt_node++;
14    }
15
16    int root;
17    void init() {
18        cnt_node = 1;
19        root = new_node();
20    }
21    void insert(char *str, int len) {
22        int cur = root;
23        for(int i = 0; i < len; i++) {
24            if(t[cur].next[str[i] - 'a'] == 0) {
25                t[cur].next[str[i] - 'a'] = new_node();
26            }
27            cur = t[cur].next[str[i] - 'a'];
28        }
29        t[cur].end++;
30    }
31    void build() {
32        std::queue<int> que;
33        t[root].fail = root;
34        for(int i = 0; i < 26; i++) {
35            if(t[root].next[i] == 0) t[root].next[i] = root;
36            else {
37                t[t[root].next[i]].fail = root;
38                que.push(t[root].next[i]);

```

```

39         }
40     }
41     while(!que.empty()) {
42         int u = que.front(); que.pop();
43         for(int i = 0; i < 26; i++) {
44             if(t[u].next[i] == 0) t[u].next[i] = t[t[u].fail].next[i];
45             else {
46                 t[t[u].next[i]].fail = t[t[u].fail].next[i];
47                 que.push(t[u].next[i]);
48             }
49         }
50     }
51 }
52 } 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， $\max(\text{父亲}) + 1 == \min(\text{儿子})$
- 子串是后缀的前缀，SAM 经常被用来判断子串/统计子串个数

---

```

1 struct Node {
2     int go[26], par;
3     int val;
4     void init(int val_) {
5         memset(this, 0, sizeof(*this));
6         val = val_;
7     }
8 };
9 struct SAM {
10     Node t[MAXN * 2];
11     int cnt_node;
12     int new_node(int val) {
13         t[cnt_node].init(val);
14         return cnt_node++;
15     }
16
17     int root, last;
18     void init() {
19         cnt_node = 1;
20         root = last = new_node(0);

```

```

21     }
22     void extend(int w) {
23         int p = last;
24         int np = new_node(t[last].val + 1);
25         //t[np].idx = i;
26         while(p != 0 && t[p].go[w] == 0) {
27             t[p].go[w] = np;
28             p = t[p].par;
29         }
30         if(p == 0) t[np].par = root;
31         else {
32             int q = t[p].go[w];
33             if(t[q].val == t[p].val + 1) t[np].par = q;
34             else {
35                 int nq = new_node(t[p].val + 1);
36                 memcpy(t[nq].go, t[q].go, sizeof(t[q].go));
37                 t[nq].par = t[q].par;
38                 t[q].par = nq;
39                 t[np].par = nq;
40                 while(p != 0 && t[p].go[w] == q) {
41                     t[p].go[w] = nq;
42                     p = t[p].par;
43                 }
44             }
45         }
46         last = np;
47     }
48 } 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

```

1 void count(int *a, int *b, int *val, int n, int m) { //[0, n) [0, m]
2     static int cnt[MAXN];
3     for(int i = 0; i <= m; i++) cnt[i] = 0;
4     for(int i = 0; i < n; i++) cnt[val[a[i]]]++;
5     for(int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];
6     for(int i = n - 1; i >= 0; i--) b[--cnt[val[a[i]]]] = a[i];
7 }
8 void da(int *str, int *sa, int *rank, int n, int m) {

```

```

9     static int sa_[MAXN];
10    for(int i = 0; i < n; i++) sa_[i] = i;
11    for(int i = 0; i < n; i++) rank[i] = str[i];
12    count(sa_, sa, rank, n, m);
13    for(int k = 0; (1 << k) < n; k++) {
14        int l = (1 << k), p = 0;
15        for(int i = n - l; i < n; i++) sa_[p++] = i;
16        for(int i = 0; i < n; i++) if(sa[i] >= l) sa_[p++] = sa[i] - l;
17        count(sa_, sa, rank, n, m);
18        int *rank_ = sa_;
19        m = 0;
20        for(int i = 0; i < n; i++) {
21            if(i > 0) {
22                if(rank[sa[i]] != rank[sa[i - 1]]) m++;
23                else if(sa[i - 1] + l >= n) m++;
24                else if(rank[sa[i] + l] != rank[sa[i - 1] + l]) m++;
25            }
26            rank_[sa[i]] = m;
27        }
28        for(int i = 0; i < n; i++) rank[i] = rank_[i];
29        if(m >= n - 1) break;
30    }
31 }
32 void calc_height(int *str, int *sa, int *rank, int *height, int n) {
33     str[n] = -1; //字符串尾需有结束符
34     for(int i = 0; i < n; i++) rank[sa[i]] = i;
35     height[0] = 0;
36     for(int i = 0, k = 0; i < n; i++) {
37         if(k > 0) k--;
38         if(rank[i] != 0) {
39             while(str[i + k] == str[sa[rank[i] - 1] + k]) k++;
40             height[rank[i]] = k;
41         }
42     }
43 }

```

---

#### 4.4 Kmp

```

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

```



---

```

6         next[i] = pre;
7     }
8 }
9 void kmpMatch(string &P, string &T, int next[], int kmp[]) {
10     for(int pre = 0, i = 0; i < T.length(); i++) {
11         while(pre > 0 && P[pre] != T[i]) pre = next[pre - 1];
12         if(P[pre] == T[i]) pre++;
13         kmp[i] = pre;
14         if(pre == P.length()) pre = next[pre - 1];
15     }
16 }

```

---

## 4.5 Manacher

---

```

1 void manacher(char *str, int len, int p[]) {
2     p[0] = 1;
3     int key = 0;
4     for(int i = 1; i < len; i++) {
5         int k = (key + p[key] == i) ? 1 : min(key + p[key] - i, p[2 * key - i]);
6         while(i - k >= 0 && i + k < len && str[i + k] == str[i - k]) k++;
7         p[i] = k;
8         if(i + p[i] > key + p[key]) key = i;
9     }
10 }

```

---

## 4.6 Z Algorithm

---

```

1 void calc_z(char *str, int len, int *z) {
2     int l = 0, r = 0;
3     for(int i = 1; i < len; i++) {
4         z[i] = 0;
5         if(i <= r) z[i] = std::min(z[i - l], r - i + 1);
6         while(i + z[i] < len && str[i + z[i]] == str[z[i]]) z[i]++;
7         if(i + z[i] - 1 > r) { l = i; r = i + z[i] - 1; }
8     }
9 }

```

---

## 4.7 Extend Kmp

---

```

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

```

```
4
5     next[0] = len;
6     for(int a = 1, p = 1, i = 1; i < len; i++) {
7         if(p <= i) a = p = i;
8
9         int k = min(next[i - a], p - i);
10        while(i + k < len && pat[i + k] == pat[k]) k++;
11        next[i] = k;
12
13        if(i + k > p) p = i + k, a = i;
14    }
15 }
16 void exKmpMatch(string &s, string &pat, vector<int> &next, vector<int> &lcp) {
17     //getExNext(pat, next);
18     int len_s = s.length(), len_p = pat.length();
19     lcp.resize(len_s);
20     for(int a = 0, p = 0, i = 0; i < len_s; i++) {
21         if(p <= i) a = p = i;
22
23         int k = min(next[i - a], p - i);
24         while(k < len_p && i + k < len_s && pat[k] == s[i + k]) k++;
25         lcp[i] = k;
26
27         if(i + k > p) p = i + k, a = i;
28     }
29 }
```

---

## 5 Math

### 5.1 Prime Number

素数筛,  $a[i]$  代表  $i$  的素因子分解式中最小的素数。

---

```

1 void prime(int a[], int n) {
2     for(int i = 1; i <= n; i++) a[i] = i;
3     for(int i = 2; i * i <= n; i++) if(a[i] == i) {
4         for(int j = i * i; j <= n; j += i) a[j] = min(a[j], i);
5     }
6 }
```

---

### 5.2 Matrix Multiplication

---

```

1 void mat_mul(int res[][MAXN], int a[][MAXN], int b[][MAXN], int n, int m, int k) {
2     static int tmp[MAXN][MAXN];
3     for(int i = 0; i < n; i++) for(int j = 0; j < k; j++) {
4         long long sum = 0;
5         for(int x = 0; x < m; x++) sum += ((long long)a[i][x] * b[x][j]) % MOD;
6         tmp[i][j] = sum % MOD;
7     }
8     for(int i = 0; i < n; i++) for(int j = 0; j < k; j++) res[i][j] = tmp[i][j];
9 }
10 void mat_pow(int res[][MAXN], int a[][MAXN], int x, int n) {
11     static int cur[MAXN][MAXN];
12     for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) cur[i][j] = a[i][j];
13     for(int i = 0; i < n; i++) for(int j = 0; j < n; j++) res[i][j] = (i == j) ? 1
14         ↪: 0;
15     while(x) {
16         if(x & 1) mat_mul(res, res, cur, n, n, n);
17         mat_mul(cur, cur, cur, n, n, n);
18         x >>= 1;
19     }
20 }
```

---

### 5.3 Number Theoretic Transform

- 离散傅立叶变换 (DFT): 多项式  $(a_0, a_1, a_2, \dots, a_{n-1})$  在  $(w^0, w^1, w^2, \dots, w^{n-1})$  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  $g^n = 1$

---

```

1 //NTT函数, 要求n为2的幂次, 且n <= MAXN
2 void ntt(int a[], int n, bool reverse) {
3     static int buf[MAXN];
4     int lg = 0;
5     while((1 << lg) < n) lg++;
6     for(int i = 0; i < n; i++) {
7         int pos = 0;
8         for(int j = 0; j < lg; j++) pos |= ((i >> j & 1) << (lg - 1 - j));
9         buf[pos] = a[i];
10    }
11    for(int i = 0; i < n; i++) a[i] = buf[i];
12
13    for(int l = 1; l < n; l <= 1) {
14        int w = powmod(3, (MOD - 1) / (2 * l));
15        if(reverse) w = powmod(w, MOD - 2);
16        for(int i = 0; i < n; i += 2 * l) {
17            int cur = 1;
18            for(int j = 0; j < l; j++) {
19                int t = mulmod(cur, a[i + j + l]);
20                int u = a[i + j];
21                a[i + j] = addmod(u, t);
22                a[i + j + l] = addmod(u, MOD - t);
23                cur = mulmod(cur, w);
24            }
25        }
26    }
27    if(reverse) {
28        int inv_n = powmod(n, MOD - 2);
29        for(int i = 0; i < n; i++) a[i] = mulmod(a[i], inv_n);
30    }
31 }
32 void polymul(int res[], int a[], int b[], int n) {
33     int m = 1;
34     while(m < n) m <= 1;
35     m <= 1;
36     ntt(a, m, false);
37     ntt(b, m, false);
38     for(int i = 0; i < m; i++) res[i] = mulmod(a[i], b[i]);
39     ntt(res, m, true);
40 }

```

---

---

## 5.4 Extend Euclid

---

```
1 int extend_gcd(int a, int b, int &x, int &y) {  
2     int ans;  
3     if(b == 0) x = 1, y = 0, ans = a;  
4     else {  
5         ans = extend_gcd(b, a % b, y, x);  
6         y -= (a / b) * x;  
7     }  
8     return ans;  
9 }
```

---

## 6 Other

### 6.1 Checker

---

```
1  if !(g++ g.cpp -o g && g++ --std=c++11 c1.cpp -o c1 && g++ c2.cpp -o c2)
2  then
3      exit
4  fi
5
6  echo read
7  read N
8
9  i=1
10 while [ $i -le $N ];
11 do
12     echo $i
13     echo $i | ./g | ./c1 > c1.txt
14     echo $i | ./g | ./c2 > c2.txt
15     if ! diff c1.txt c2.txt
16     then
17         echo $i | ./g > ${i}.txt
18     fi
19     i=$((i+1))
20 done
```

---

### 6.2 Fast IO

---

```
1  #include <cstdio>
2  #include <cmath>
3  template <typename T>
4  bool nxtInt(T &res) {
5      char c;
6      bool negative = false;
7      while(c=getchar(), c!='-' && !('0'<=c && c<='9'))
8          if(c == EOF) return false;
9      if(c == '-') negative = true, c = getchar();
10     res = 0;
11     while('0'<=c && c<='9') res *= 10, res += c - '0', c = getchar();
12     if(negative) res *= -1;
13     return true;
14 }
15
16 template <typename T>
```

```
17 void prtInt(T x) {
18     if(x < 0) putchar('-'), x *= -1;
19     if(x > 9) prtInt(x / 10);
20     putchar(x % 10 + '0');
21 }
22
23 template <typename T>
24 bool nxtNum(T &res) {
25     char c;
26     bool negative = false;
27     while(c=getchar(), c!='.' && c!='-' && !('0'<=c && c<='9'))
28         if(c == EOF) return false;
29     if(c == '-') negative = true, c = getchar();
30     res = 0;
31     long long cnt = 0;
32     while(c == '.' || ('0'<=c && c<='9')) {
33         cnt *= 10;
34         if(c == '.') cnt = 1;
35         else res *= 10, res += c - '0';
36         c = getchar();
37     }
38     if(negative) res *= -1;
39     if(cnt > 0) res *= 1.0/cnt;
40     return true;
41 }
42
43 template <typename T>
44 void prtNum(T x, long precision) {
45     if(x < 0) putchar('-'), x *= -1;
46     long long flr = floor(x);
47     prtInt(flr);
48     x -= flr;
49     x *= pow(10, precision);
50     flr = round(x);
51     putchar('.');
52     long long cnt = 0, t = 1;
53     while(flr >= t) cnt++, t *= 10;
54     for(long i=0; i<precision-cnt; i++) putchar('0');
55     prtInt(flr);
56 }
57 bool nxtChar(char &c) {
58     do {
59         c = getchar();
```

```
60     } while(c == ' ' || c == '\n');  
61     return c != EOF;  
62 }
```

---