ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

ACM-ICPC World Finals 2019

Team Reference Document

University of Illinois at Urbana-Champaign:

VIM - Help poor children

Coach

Mattox Beckman

Contestants

Lan Dao, Jiacheng Liu, Zexuan Zhong

Contents

1 Data Structures

1.1 Binary Indexed Tree 2D

```
// Support 2 types of queries:
// - Add v to cell (x, y)
// - Get the sum of rectangle with top-left corner (1, 1)
// and lower-right corner (x, y)

void update(int x, int y, int v) {
    while (x <= n) {
        int z = y;
        while (z <= n) {
            bit[x][z] += v;
            z += (z & (-z));
        }
        x += (x & (-x));
    }
}

int get(int x, int y) {
    if (x == 0 || y == 0) return 0;
    int sum = 0;
    while (x) {
        int z = y;
        while (z) {
            sum += bit[x][z];
            z -= (z & (-z));
        }
        x -= (x & (-x));
    }
    return sum;
}
```

1.2 Segment Tree 2D

```
// Supported:
// - Add a value v to cell (x, y)
// - Get the sum in rectangle with top left corner
// (x1, y1) and bottom right corner (x2, y2)

void build_y(int k_x, int k_y, int l, int r) {
    if (l == r) {
        t[k_x][k_y] = 0;
        return;
    }
    int mid = (l + r) >> 1;
    build_y(k_x, k_y * 2, l, mid);
    build_y(k_x, k_y * 2 + 1, mid + 1, r);
    t[k_x][k_y] = 0;
}

void build_x(int k, int l, int r) {
    build_y(k, 1, 1, n);
    if (l == r) return;
    int mid = (l + r) >> 1;
    build_x(k * 2, l, mid);
    build_x(k * 2 + 1, mid + 1, r);
}

void update_y(int k_x, int l_x, int r_x, int k_y, int l_y, int
     r_y, int y, int v) {
    if (y < l_y || r_y < y) return;
    if (l_y == r_y) {
        if (l_x == r_x)
            t[k_x][k_y] += v;
        else
            t[k_x][k_y] = t[k_x * 2][k_y] + t[k_x * 2 + 1][k_y];
        return;
    }
    int mid = (l_y + r_y) >> 1;
    update_y(k_x, l_x, r_x, k_y * 2, l_y, mid, y, v);
    update_y(k_x, l_x, r_x, k_y * 2 + 1, mid + 1, r_y, y, v);
    t[k_x][k_y] = t[k_x][k_y * 2] + t[k_x][k_y * 2 + 1];
}

void update_x(int k, int l, int r, int x, int y, int v) {
```

```
    if (x < l || r < x) return;
    if (l == r) {
        update_y(k, l, r, 1, 1, n, y, v);
        return;
    }
    int mid = (l + r) >> 1;
    update_x(k * 2, l, mid, x, y, v);
    update_x(k * 2 + 1, mid + 1, r, x, y, v);
    update_y(k, l, r, 1, 1, n, y, v);
}

int get_y(int k_x, int k_y, int l, int r, int y1, int y2) {
    if (y2 < l || r < y1) return 0;
    if (y1 <= l && r <= y2) return t[k_x][k_y];
    int mid = (l + r) >> 1;
    return get_y(k_x, k_y * 2, l, mid, y1, y2) +
           get_y(k_x, k_y * 2 + 1, mid + 1, r, y1, y2);
}

int get_x(int k, int l, int r, int x1, int x2, int y1, int y2) {
    if (r < x1 || x2 < l) return 0;
    if (x1 <= l && r <= x2)
        return get_y(k, 1, 1, n, y1, y2);
    int mid = (l + r) >> 1;
    return get_x(k * 2, l, mid, x1, x2, y1, y2) +
           get_x(k * 2 + 1, mid + 1, r, x1, x2, y1, y2);
}
```

1.3 Persistent Segment Tree

```
struct Node {
    Node() = default;

    Node(int l, int r, int v)
        : left(l), right(r), val(v) {}

    int left, right, val;
};

int build(int k, int l, int r) {
    tree[k].val = 0;
    if (l == r) return k;
    tree[k].left = ++num_node;
    tree[k].right = ++num_node;
    int mid = (l + r) >> 1;
    build(tree[k].left, l, mid);
    build(tree[k].right, mid + 1, r);
    return k;
}

int update(int k, int l, int r, int i, int v) {
    int K = ++num_node;
    if (l == r) {
        tree[K].val = tree[k].val + v;
        return K;
    }
    tree[K].left = tree[k].left;
    tree[K].right = tree[k].right;
    int mid = (l + r) >> 1;
    if (i <= mid)
        tree[K].left = update(tree[K].left, l, mid, i, v);
    else
        tree[K].right = update(tree[K].right, mid + 1, r, i, v);
    tree[K].val = tree[tree[K].left].val +
        tree[tree[K].right].val;
    return K;
}
```

1.4 Splay Tree

```
// Supports reversing a segment.

struct SplayTree {
    struct Node {
        Node *left, *right, *parent;
        int value, size;
```

```
    bool reversed;
};

SplayTree() {
    nilt = new Node();
    nilt->left = nilt->right = nilt->parent = nilt;
    nilt->value = nilt->size = 0;
    nilt->reversed = false;
}

void set_left(Node* x, Node* y) {
    x->left = y;
    y->parent = x;
}

void set_right(Node* x, Node* y) {
    x->right = y;
    y->parent = x;
}

void set_child(Node* x, Node* y, bool is_right) {
    if (is_right) set_right(x, y);
    else set_left(x, y);
}

void build_tree(vector < int >& arr) {
    root = nilt;
    for (int i = 0; i < arr.size(); ++i) {
        Node* x = new Node();
        x->size = arr[i];
        x->value = arr[i];
        x->reversed = false;
        set_left(x, root);
        x->parent = x->right = nilt;
        root = x;
    }
}

void propagate(Node* x) {
    if (x == nilt) return;
    if (x->reversed) {
        swap(x->left, x->right);
        x->left->reversed = !x->left->reversed;
        x->right->reversed = !x->right->reversed;
        x->reversed = false;
    }
}

Node* locate(Node* x, int pos) {
    do {
        propagate(x);
        int num = x->left->size + 1;
        if (num == pos) return x;
        if (num > pos) x = x->left;
        else
            pos -= num, x = x->right;
    } while (true);
    return x;
}

void update(Node* x) {
    x->size = x->left->size + x->right->size + 1;
}

void uptree(Node* x) {
    Node* y = x->parent;
    Node* z = y->parent;
    if (x == y->right) {
        Node* b = x->left;
        set_right(y, b);
        set_left(x, y);
    }
    else {
        Node* b = x->right;
        set_left(y, b);
        set_right(x, y);
    }
    update(y);
    update(x);
    set_child(z, x, z->right == y);
```

```
}

void splay(Node* x) {
    do {
        Node* y = x->parent;
        if (y == nilt) return;
        Node* z = y->parent;
        if (z != nilt) {
            if ((x == y->left) == (y == z->left))
                uptree(y);
            else
                uptree(x);
        }
        uptree(x);
    } while (true);
}

void split(Node* t, int pos, Node*& t1, Node*& t2) {
    if (pos == 0) {
        t1 = nilt;
        t2 = t;
        return;
    }
    if (pos >= t->size) {
        t1 = t;
        t2 = nilt;
        return;
    }
    Node* x = locate(t, pos);
    splay(x);
    t1 = x;
    t2 = x->right;
    t1->right = nilt;
    t2->parent = nilt;
    update(t1);
}

Node* join(Node* t1, Node* t2) {
    if (t1 == nilt) return t2;
    t1 = locate(t1, t1->size);
        splay(t1);
        set_right(t1, t2);
        update(t1);
        return t1;
    }

    Node *root, *nilt;
};
```

1.5 Mo's Algorithm

```
// The array is 1-based

bool cmp_mo(Query i, Query j) {
    int s = (int) sqrt(n);
    return ((i.l - 1) / s < (j.l - 1) / s || ((i.l - 1) / s ==
        (j.l - 1) / s && i.r < j.r));
}
```

2 Graph Theory

2.1 Ford Fulkerson

```
bool find_path() {
    int l = 1, r = 1; ++flag;
    q[1] = source; check[source] = flag;
    while (l <= r) {
        int u = q[l++];
        for (auto v : adj[u])
            if (check[v] != flag && c[u][v] > f[u][v]) {
                pre[v] = u;
                check[v] = flag;
                if (v == target) return true;
                q[++r] = v;
            }
```

```
    }
    return false;
}

void augment() {
    int v = target, delta = oo;
    while (v != source) {
        int u = pre[v];
        delta = min(c[u][v] - f[u][v], delta);
        v = u;
    }
    v = target; flow += delta;
    while (v != source) {
        int u = pre[v];
        f[u][v] += delta;
        f[v][u] -= delta;
        v = u;
    }
}
```

2.2 Tarjan

```
// avail[] initialized to be all 0
void tarjan(int u) {
    num[u] = low[u] = ++num_node;
    st.push(u);
    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i];
        if (!avail[v]) {
            if (num[v] == 0) {
                tarjan(v);
                low[u] = min(low[u], low[v]);
            }
            else low[u] = min(low[u], num[v]);
        }
    }
    if (low[u] == num[u]) {
        int v = -1;
        ++num_comp;
        while (v != u) {
            v = st.top(); st.pop();
            comp[v] = num_comp;
            avail[v] = 1;
        }
    }
}
```

2.3 Topo Sort

```
void topo_sort() {
    for (int i = 1; i <= num_comp; ++i)
        if (deg[i] == 0) q.push(i);
    int num = 0;
    while (!q.empty()) {
        int u = q.front(); q.pop();
        for (int i = 0; i < new_adj[u].size(); ++i) {
            int v = new_adj[u][i];
            --deg[v];
            if (deg[v] == 0) q.push(v);
        }
        position[u] = ++num;
    }
}
```

2.4 2-SAT

```
bool two_sat() {
    for (int i = 0; i < list_node.size(); ++i)
        if (!num[list_node[i]]) tarjan(list_node[i]);
    for (int i = 0; i < list_node.size(); ++i) {
        int u = list_node[i];
        if (comp[u] == comp[neg[u]]) return false;
        for (int j = 0; j < adj[u].size(); ++j) {
            int v = adj[u][j];
```

```
            if (comp[u] == comp[v]) continue;
            new_adj[comp[u]].push_back(comp[v]);
            ++deg[comp[v]];
        }
    }
    topo_sort();
    for (int i = 0; i < list_node.size(); ++i) {
        int u = list_node[i];
        // position[u]: position of u after topo sorted
        if (position[comp[u]] > position[comp[neg[u]]])
            check[u] = 1; // Pick u (otherwise pick !u)
    }
    return true;
}
```

2.5 Lowest Common Ancestor - $O(n \log n)$

```
// Note: Log = ceil(log2(n))
// d[u] = depth of node u + 1 (ie: d[root] = 1)

void buildLCA() {
    for (int i = 1; i <= n; ++i) p[i][0] = par[i];
    for (int j = 1; j <= Log; ++j)
        for (int i = 1; i <= n; ++i)
            p[i][j] = p[p[i][j - 1]][j - 1];
}

int LCA(int u, int v) {
    if (d[u] < d[v]) swap(u, v);
    for (int j = Log; j >= 0; --j)
        if (d[p[u][j]] >= d[v]) u = p[u][j];
    if (u == v) return u;
    for (int j = Log; j >= 0; --j)
        if (p[u][j] != p[v][j]) {
            u = p[u][j];
            v = p[v][j];
        }
    return p[u][0];
}
```

2.6 Centroid Decomposition

```
void build(int u, int p) {
    sze[u] = 1;
    for (int v : adj[u])
        if (!elim[v] && v != p) build(v, u), sze[u] += sze[v];
}

int get_centroid(int u, int p, int num) {
    for (int v : adj[u])
        if (!elim[v] && v != p && sze[v] > num / 2)
            return get_centroid(v, u, num);
    return u;
}

void centroid_decomposition(int u) {
    build(u, -1);
    int root = get_centroid(u, -1, sze[u]);
    // Do stuffs here
    elim[root] = true;
    for (int v : adj[root])
        if (!elim[v]) centroid_decomposition(v, c + 1);
}
```

2.7 Heavy Light Decomposition

```
void build(int u) {
    size_tree[u] = 1;
    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i];
        if (parent[u] == v) continue;
        parent[v] = u;
        build(v);
        size_tree[u] += size_tree[v];
```

```
    }
}

void hld(int u) {
    if (chain_head[num_chain] == 0)
        chain_head[num_chain] = u;
    chain_idx[u] = num_chain;
    arr_idx[u] = ++num_arr;
    node_arr[num_arr] = u;

    int heavy_child = -1;
    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i];
        if (parent[u] == v) continue;
        if (heavy_child == -1 || size_tree[v] >
             size_tree[heavy_child])
            heavy_child = v;
    }

    if (heavy_child != -1)
        hld(heavy_child);

    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i];
        if (v == heavy_child || parent[u] == v) continue;
        ++num_chain;
        hld(v);
    }
}

// u is an ancestor of v
int query_hld(int u, int v) {
    int uchain = chain_idx[u], vchain = chain_idx[v], ans = -1;
    while (true) {
        if (uchain == vchain) {
            get(..., arr_idx[u], arr_idx[v]);
            break;
        }
        get(..., arr_idx[chain_head[vchain]], arr_idx[v]);
        v = parent[chain_head[vchain]];
        vchain = chain_idx[v];
    }
    return ans;
}
```

3 Dynamic Programming

3.1 Convex Hull Trick

```
// Finding max.

typedef long long htype;
typedef pair < htype, htype > line;
vector < line > lst;

bool is_bad(line l1, line l2, line l3) {
    return (1.0 * (l1.second - l2.second)) / (l2.first -
         l1.first) >= (1.0 * (l2.second - l3.second)) /
         (l3.first - l2.first);
}

// Assuming lines' slopes m are strictly increasing.
void add(htype m, htype b) {
    while (lst.size() >= 2 && is_bad(lst[lst.size() - 2],
         lst.back(), {m, b}))
        lst.pop_back();
    lst.push_back({m, b});
}

htype get_value(line d, htype x) {
    return d.first * x + d.second;
}

// Assuming queries' x are strictly increasing.
int pointer = 0;
htype get(htype x) {
    if (pointer > lst.size()) pointer = lst.size() - 1;
```

```
    while (pointer < lst.size() - 1 && get_value(lst[pointer],
         x) < get_value(lst[pointer + 1], x))
        ++pointer;
    return get_value(lst[pointer], x);
}
```

3.2 Dynamic Convex Hull Trick

```
// Slow but correct. Takes O(log n) per add and query.

typedef long long htype;
// Representing a line. To query value x,
// set m = x, is_query = true.
struct Line {
    bool operator < (const Line& rhs) const {
        // Compare lines
        if (!rhs.is_query) return m < rhs.m;

        // Compare queries
        const Line* s = nxt();
        if (s == NULL) return false;
        htype x = rhs.m;
        return s->m * x + s->b > m * x + b;
    }

    htype m, b;
    bool is_query;
    mutable function < const Line*() > nxt;
};

class ConvexHullTrick : public set < Line > {
  public:
    void add(htype m, htype b) {
        auto p = insert({m, b, false});
        if (!p.second) return;
        iterator y = p.first;
        y->nxt = [=] { return (next(y) == end()) ? NULL :
             &(*next(y)); };
        if (is_bad(y)) {
            erase(y);
            return;
        }
        while (next(y) != end() && is_bad(next(y)))
             erase(next(y));
        while (y != begin() && is_bad(prev(y))) erase(prev(y));
    }

    htype get(htype x) {
        iterator y = lower_bound({x, 0, true});
        return y->m * x + y->b;
    }
  private:
    bool is_bad(iterator y) {
        iterator z = next(y);
        if (y == begin())
            return ((z == end()) ? false : y->m == z->m && y->b
                 <= z->b);
        iterator x = prev(y);
        if (z == end())
            return (y->m == x->m && y->b <= x->b);
        return (x->b - y->b) * (z->m - y->m) >= (y->b - z->b) *
             (y->m - x->m);
    }
};
```

4 String

4.1 Suffix Array

```
bool suffix_cmp(int i, int j) {
    if (pos[i] != pos[j]) return pos[i] < pos[j];
    i += gap;
    j += gap;
    return (i < N && j < N) ? pos[i] < pos[j] : i > j;
}
```

```
void build_sa() {
    N = s.size();
    for (int i = 0; i < N; ++i) sa[i] = i, pos[i] = s[i];
    for (gap = 1;; gap *= 2) {
        sort(sa, sa + N, suffix_cmp);
        for (int i = 0; i < N - 1; ++i) tmp[i + 1] = tmp[i] +
            suffix_cmp(sa[i], sa[i + 1]);
        for (int i = 0; i < N; ++i) pos[sa[i]] = tmp[i];
        if (tmp[N - 1] == N - 1) break;
    }
}
```

4.2 Longest Common Prefix

```
// height[i] = length of common prefix of suffix(sa[i]) and
    suffix(sa[i+1])

void build_height () {
    height.assign(n-1, -1);
    for (int i = 0, k = 0; i < n; i++) {
        if (rk[i] == n-1) continue;
        if (k) k--;
        for (int j = sa[rk[i]+1]; i+k<n && j+k<n && s[i+k] ==
            s[j+k]; k++);
        height[rk[i]] = k;
    }
}
```

4.3 Aho-Corasick Automata

```
struct Node {
   Node* next[26];
   Node* fail;
   int cnt;
   Node (Node* root) {
      memset(next, NULL, sizeof(next));
      fail = root;
      cnt = 0;
   }
};
Node* root;

void insert (string s) {
   Node* curr = root;
   for (int i = 0; i < s.length(); i++) {
      int j = s[i] - 'a';
      if (curr->next[j] == NULL) {
         curr->next[j] = new Node(root);
      }
      curr = curr->next[j];
   }
   curr->cnt++;
}

void make_fail () {
   queue<Node*> q;
   for (int i = 0; i < 26; i++) {
      if (root->next[i]) {
         q.push(root->next[i]);
      }
   }
   while (!q.empty()) {
      Node* node = q.front(); q.pop();
      for (int i = 0; i < 26; i++) {
         if (node->next[i]) {
            q.push(node->next[i]);
            Node* f = node->fail;
            while (f != root && !f->next[i]) {
               f = f->fail;
            }
            if (f->next[i]) {
               f = f->next[i];
            }
            node->next[i]->fail = f;
         }
```

```
      }
   }
}

int work (string s) {
   set<Node*> seen;
   int cnt = 0;
   Node* curr = root;
   for (int i = 0; i < s.length(); i++) {
      int j = s[i] - 'a';
      while (curr != root && !curr->next[j]) {
         curr = curr->fail;
      }
      if (curr->next[j]) {
         curr = curr->next[j];
         Node* p = curr;
         while (p != root) {
            if (seen.find(p) != seen.end()) break;
            seen.insert(p);
            cnt += p->cnt;
            p = p->fail;
         }
      }
   }
   return cnt;
}
```

4.4 Palindromic Tree

```
struct Node {
    Node* next[26]; // to palindrome by extending me with a
        letter
    Node* sufflink; // my LSP
    int len; // length of this palindrome substring
    int num; // number of palindrome substrs ending here
};
Node nodes[NMAX];
int n = 0; // number of nodes in tree
vector<int> s;
LL ans = 0;

void build_tree () {
    nodes[0].len = -1; nodes[0].sufflink = &nodes[0]; // root 0
    nodes[1].len = 0; nodes[1].sufflink = &nodes[0]; // root 1
    n = 2;
    Node* suff = &nodes[1]; // node for LSP of processed prefix
    for (int i = 0; i < s.size(); i++) {
        // find LSP xAx
        Node* ptr = suff;
        while (1) {
            int j = i - 1 - ptr->len;
            if (j >= 0 && s[j] == s[i]) break;
            ptr = ptr->sufflink;
        }

        if (ptr->next[s[i]]) { // palindrome substr already
            exists
            suff = ptr->next[s[i]];
        } else { // add a new node
            suff = &nodes[n++];
            suff->len = ptr->len + 2;
            ptr->next[s[i]] = suff;
            if (suff->len == 1) { // current LSP is trivial
                suff->sufflink = &nodes[1];
                suff->num = 1;
            } else {
                // find xAx's LSP xBx
                while (1) {
                    ptr = ptr->sufflink;
                    int j = i - 1 - ptr->len;
                    if (j >= 0 && s[j] == s[i]) break;
                }
                suff->sufflink = ptr->next[s[i]];
                suff->num = suff->sufflink->num + 1;
            }
        }
        ans += suff->num;
    }
```

```
}
```

5 Game Theory

5.1 Nim Product

```
// Note: (i | j) might overflow

int nim_multiply(int x, int y) {
    int p = 0;
    for (int i = 0; i < maxLog + 1; ++i)
        if (x & (1 << i))
            for (int j = 0; j < maxLog + 1; ++j)
                if (y & (1 << j))
                    p ^= mul[i][j];
    return p;
}

void init() {
    for (int i = 0; i < maxLog + 1; ++i)
        for (int j = 0; j <= i; ++j) {
            if ((i & j) == 0) mul[i][j] = 1 << (i | j);
            else {
                mul[i][j] = 1;
                for (int t = 0; t < maxLog + 1; ++t) {
                    int k = (1 << t);
                    if (i & j & k) mul[i][j] =
                        nim_multiply(mul[i][j], ((1 << k) * 3) >>
                        1);
                    else
                        if ((i | j) & k) mul[i][j] =
                            nim_multiply(mul[i][j], (1 << k));
                }
            }
            mul[j][i] = mul[i][j];
        }
}
```

6 Math

6.1 Extended Euclidean Algorithm

```
// Find a * x + b * y = gcd(a, b) = d
//     q = a / b; r = a % b
//     b * x1 + r * y1 = d
// =>  b * x1 + (a - q * b) * y1 = d
// =>  b * x1 + a * y1 - q * b * y1 = d
// =>  a * y1 + (x1 - q * y1) * b = d
// =>  x = y1, y = x1 - q * y1
typedef LL long long;
pair < LL, LL > extended_gcd(LL a, LL b) {
    if (b == 0) return make_pair(1, 0);
    pair < LL, LL > xy = extended_gcd(b, a % b);
    return make_pair(xy.second, xy.first - (a / b) * xy.second);
}

LL inverse_modulo(LL a, LL m) {
    pair < LL, LL > xy = extended_gcd(a, m);
    return (xy.first + modP) % modP;
}
```

6.2 Fast Fourier Transform

```
const double PI = 2 * acos(0);

struct C {
   double a, b;
   C () : a(0), b(0) {}
   C (double a, double b) : a(a), b(b) {}
   C (double theta) : a(cos(theta)), b(sin(theta)) {}
   C bar () const { return C(a, -b); }
   double modsq () const { return a * a + b * b; }
   C operator+ (const C &c) const { return C(a + c.a, b + c.b); }
   C operator* (const C &c) const { return C(a * c.a - b * c.b,
       a * c.b + b * c.a); }
```

```
  C operator/ (const C &c) const {
    C r = (*this) * c.bar();
    return C(r.a / c.modsq(), r.b / c.modsq());
  }
};

// O(nlogn)
// dir is direction of Fourier transform
void fft (C *in, C *out, int step, int size, int dir) {
  if (size < 1) return;
  if (size == 1) { out[0] = in[0]; return; }
  fft(in, out, step*2, size/2, dir);
  fft(in + step, out + size/2, step*2, size/2, dir);
  for (int i = 0; i < size/2; i++) {
    C even = out[i], odd = out[i + size/2];
    out[i] = even + C(dir * 2*PI * i / size) * odd;
    out[i + size/2] = even + C(dir * 2*PI * (i + size/2) /
        size) * odd;
  }
}

// c[i] = sum of a[j] * b[i-j]
// n is power of 2; index is cyclic
void convolve (int n, C *a, C *b, C *c) {
  C *fa = new C[n];
  C *fb = new C[n];
  C *fc = new C[n];
  fft(a, fa, 1, n, 1);
  fft(b, fb, 1, n, 1);
  for (int i = 0; i < n; i++) fc[i] = fa[i] * fb[i];
  fft(fc, c, 1, n, -1);
  for (int i = 0; i < n; i++) c[i] = c[i] / C(n,0);
}
```

6.3 Gaussian Elimination

```
// Note: ax = b
bool gaussian_elimination() {
    vector < int > row;
    for (int i = 0; i < N; ++i) row.push_back(i);
    for (int t = 0; t < N; ++t) {
        int R = -1;
        for (int i = t; i < N; ++i) {
            int r = row[i];
            if (a[r][t] > eps) {
                R = i;
                break;
            }
        }
        if (R == -1) return false;
        swap(row[R], row[t]);
        R = row[t];
        for (int i = t + 1; i < N; ++i) {
            int r = row[i];
            double p = a[r][t] / a[R][t];
            for (int c = 0; c < N; ++c)
                a[r][c] -= p * a[R][c];
            b[r] -= p * b[R];
        }
    }
    for (int i = N - 1; i >= 0; --i) {
        int r = row[i];
        for (int c = N - 1; c > i; --c)
            b[r] -= a[r][c] * res[c];
        res[r] = b[r] / a[r][i];
    }
    return true;
}
```