



## ACM-ICPC World Finals 2017

### *Team Reference Document*

University of Illinois at Urbana-Champaign:  
Time Limit Exceeded

---

#### ***Coach***

Uttam Thakore

#### ***Contestants***

---

Tong Li, Yuting Zhang, Yewen Fan

## 1 Data Structures

1.1	Bitmasks	2
1.2	Union-Find Disjoint Sets	2
1.3	Segment Tree	2
1.4	Fenwick Tree	3
1.5	Treap	3
1.6	Splay	4
1.7	Trie	6
<b>2</b>	<b>Graph Theory</b>	<b>6</b>
2.1	Articulation Points and Bridges	6
2.2	Tarjan's Algorithm	7
2.3	Bipartite Graph Check	8
2.4	Kruskal's Algorithm	8
2.5	Prim's Algorithm	8
2.6	Dijkstra's Algorithm	9
2.7	Bellman Ford's Algorithm	9
2.8	Floyd Warshall's Algorithm	9
2.9	Shortest Path Faster Algorithm	9
2.10	Network Flow	10
2.11	Euler Tour	10
2.12	Max Cardinality Bipartite Matching	11
2.13	Min-Cost Flow	12
<b>3</b>	<b>Math</b>	<b>13</b>
3.1	Sieve of Eratosthenes	13
3.2	Euler Phi function	13
3.3	GCD mod related (CRT)	13
3.4	Enumerate Combination	14
3.5	Gauss Elimination	14
3.6	FFT	14
3.7	Simplex	15
3.8	Pell Function	16
3.9	二次剩余	16
3.10	Schröder-Hipparchus Number	17

3.11 Catalan Numbers . . . . .	17
<b>Computational Geometry</b>	<b>17</b>
<b>String Processing</b>	<b>21</b>
5.1 KMP . . . . .	21
5.2 Suffix Array . . . . .	22

# Data Structures

## 1.1 Bitmasks

---

```
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1 << n) - 1)
#define modulo(S, N) ((S) & (N - 1)) // returns S % N, where N
    is a power of 2
#define isPowerOfTwo(S) (!(S & (S - 1)))
#define nearestPowerOfTwo(S) ((int)pow(2.0,
    (int)((log((double)S) / log(2.0)) + 0.5)))
#define turnOffLastBit(S) ((S) & (S - 1))
#define turnOnLastZero(S) ((S) | (S + 1))
#define turnOffLastConsecutiveBits(S) ((S) & (S + 1))
#define turnOnLastConsecutiveZeroes(S) ((S) | (S - 1))
```

---

## 1.2 Union-Find Disjoint Sets

---

```
struct DisjointSets{
    void addelements(int num){
        while (num-->0)
            s.push_back(-1);
    }
    int find(int elem) {
        return s[elem] < 0 ? elem : s[elem] = find(s[elem]);
    }
    void setunion(int a, int b) {
        int root1 = find(a), root2 = find(b);
        int newSize = s[root1] + s[root2];
        if (s[root1] <= s[root2]){
            s[root2] = root1;
            s[root1] = newSize;
        }
        else{
            s[root1] = root2;
            s[root2] = newSize;
        }
    }
}
```

---

```
vector<int> s;
};
```

---

## 1.3 Segment Tree

---

```
// Segment tree for range sum queries.
struct segment_tree {
    vector<long long> st, lazy;
    const vector<long long> &A;
    size_t n;
    inline int left(int p) { return p << 1; }
    inline int right(int p) { return (p << 1) + 1; }
    void propagate(int p, int L, int R) {
        if (lazy[p] != 0) {
            if (L != R) {
                lazy[left(p)] += lazy[p];
                lazy[right(p)] += lazy[p];
            }
            st[p] += (R - L + 1) * lazy[p];
            lazy[p] = 0;
        }
    }
    void build(int p, int L, int R) {
        if (L == R)
            st[p] = A[L];
        else {
            build(left(p), L, (L + R) / 2);
            build(right(p), (L + R) / 2 + 1, R);
            st[p] = st[left(p)] + st[right(p)];
        }
    }
    long long update(int p, int L, int R, int i, int j, long
        long val) {
        propagate(p, L, R);
        if (L > j || R < i)
            return st[p];
        if (L >= i && R <= j) {
            lazy[p] = val;
            propagate(p, L, R);
            return st[p];
        }
    }
}
```

---

```

    }
    return st[p] = update(left(p), L, (L + R) / 2, i, j,
        val) +
        update(right(p), (L + R) / 2 + 1, R, i, j,
            val);
}
long long query(int p, int L, int R, int i, int j) {
    if (L > j || R < i)
        return 0;
    propagate(p, L, R);
    if (L >= i && R <= j)
        return st[p];
    return query(left(p), L, (L + R) / 2, i, j) +
        query(right(p), (L + R) / 2 + 1, R, i, j);
}
segment_tree(const vector<long long> &_A): A(_A) {
    n = A.size();
    st.assign(n * 4, 0);
    lazy.assign(n * 4, 0);
    build(1, 0, n - 1);
}
void update(int i, int j, long long val) {
    update(1, 0, n - 1, i, j, val);
}
long long query(int i, int j) {
    return query(1, 0, n - 1, i, j);
}
};

```

## 1.4 Fenwick Tree

```

#define LSOne(S) (S & (-S))
class FenwickTree {
private:
    vi ft;
public:
    FenwickTree() {}
    // initialization: n + 1 zeroes, ignore index 0
    FenwickTree(int n) { ft.assign(n + 1, 0); }

```

```

int rsq(int b) { // returns
    RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }
int rsq(int a, int b) { // returns
    RSQ(a, b)
    return rsq(b) - (a == 1 ? 0 : rsq(a - 1)); }
// adjusts value of the k-th element by v (v can be +ve/inc
// or -ve/dec)
void adjust(int k, int v) { // note: n =
    ft.size() - 1
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};

```

## 1.5 Treap

```

template<typename T>
struct treap{
    treap(){
        srand(time(0));
        root = nullptr;
    }
    void insert(const T& elem){
        insert(root, elem);
    }
    void remove(const T& elem){
        remove(root, elem);
    }
    struct node_t{
        T elem;
        shared_ptr<node_t> left, right;
        int priority;
    };

    shared_ptr<node_t> root;
    shared_ptr<node_t> rotateLeft(shared_ptr<node_t> node){
        shared_ptr<node_t> right = node->right, rightLeft =
            right->left;

```

```

    right->left = node;
    node->right = rightLeft;
    return right;
}

shared_ptr<node_t> rotateRight(shared_ptr<node_t> node){
    shared_ptr<node_t> left = node->left, leftRight =
        left->right;
    left->right = node;
    node->left = leftRight;
    return left;
}

void insert(shared_ptr<node_t>& node, const T& elem){
    if (node == nullptr){
        node = make_shared<node_t>();
        node->elem = elem;
        node->left = node->right = nullptr;
        node->priority = rand();
        return;
    }
    // We do not allow multiple keys with the same value
    if (node->elem == elem)
        return;

    if (node->elem > elem){
        insert(node->left, elem);
        if (node->priority < node->left->priority)
            node = rotateRight(node);
    }else{
        insert(node->right, elem);
        if (node->priority < node->right->priority)
            node = rotateLeft(node);
    }
}

void remove(shared_ptr<node_t>& node, const T& elem){
    if (node == nullptr)
        return;
    if (node->elem == elem){
        if (!node->left && !node->right)
            node = nullptr;
        // Keep rotating until the node to be deleted becomes

```

```

        a leaf node.
    else if (!node->left || (node->left && node->right &&
        node->left->priority < node->right->priority)){
        node = rotateLeft(node);
        remove(node->left, elem);
    }
    else{
        node = rotateRight(node);
        remove(node->right, elem);
    }
}
else if (node->elem > elem)
    remove(node->right, elem);
else
    remove(node->left, elem);
}};

```

## 1.6 Splay

```

const int maxNodeCnt = 111111;
int nodeCnt, root, type[maxNodeCnt], parent[maxNodeCnt],
    childs[maxNodeCnt][2],
    size[maxNodeCnt], stack[maxNodeCnt], reversed[maxNodeCnt];
// ...
void clear() {
    root = 0;
    size[0] = 0;
    nodeCnt = 1;
}

int malloc() {
    type[nodeCnt] = 2;
    childs[nodeCnt][0] = childs[nodeCnt][1] = 0;
    size[nodeCnt] = 1;
    reversed[nodeCnt] = 0;
    return nodeCnt++;
}

void update(int x) {
    size[x] = size[childs[x][0]] + 1 + size[childs[x][1]];
}

```

```

    // ...
}
void pass(int x) {
    // NOTICE: childs[x][i] == 0
    if (reversed[x]) {
        swap(childs[x][0], childs[x][1]);
        type[childs[x][0]] = 0;
        reversed[childs[x][0]] ^= 1;
        type[childs[x][1]] = 1;
        107reversed[childs[x][1]] ^= 1;
        reversed[x] = 0;
    }
    // ...
}
void rotate(int x) {
    int t = type[x],
    y = parent[x],
    z = childs[x][1 - t];
    type[x] = type[y];
    parent[x] = parent[y];
    if (type[x] != 2) {
        childs[parent[x]][type[x]] = x;
    }
    type[y] = 1 - t;
    parent[y] = x;
    childs[x][1 - t] = y;
    if (z) {
        type[z] = t;
        parent[z] = y;
    }
    childs[y][t] = z;
    update(y);
}
void splay(int x) {
    int stackCnt = 0;
    stack[stackCnt++] = x;
    for (int i = x; type[i] != 2; i = parent[i]) {
        stack[stackCnt++] = parent[i];
    }
    for (int i = stackCnt - 1; i > -1; -- i) {

```

```

        pass(stack[i]);
    }
    while (type[x] != 2) {
        int y = parent[x];
        if (type[x] == type[y]) {
            rotate(y);
        } else {
            rotate(x);
        }
        if (type[x] == 2) {
            break;
            108}
        rotate(x);
    }
    update(x);
}
int find(int x, int rank) {
    while (true) {
        pass(x);
        if (size[childs[x][0]] + 1 == rank) {
            break;
        }
        if (rank <= size[childs[x][0]]) {
            x = childs[x][0];
        } else {
            rank -= size[childs[x][0]] + 1;
            x = childs[x][1];
        }
    }
    return x;
}
void split(int &x, int &y, int a) {
    // NOTICE: x, y != 0
    y = find(x, a + 1);
    splay(y);
    x = childs[y][0];
    type[x] = 2;
    childs[y][0] = 0;
    update(y);
}

```

```

void split3(int &x, int &y, int &z, int a, int b) {
    split(x, z, b);
    split(x, y, a - 1);
}
void join(int &x, int y) {
    // NOTICE x, y != 0
    x = find(x, size[x]);
    splay(x);
    childs[x][1] = y;
    type[y] = 1;
    parent[y] = x;
    109update(x);
}
void join3(int &x, int y, int z) {
    join(y, z);
    join(x, y);
}
int getRank(int x) {
    splay(x);
    root = x;
    return size[childs[x][0]];
}
void reverse(int a, int b) {
    int x, y;
    split3(root, x, y, a + 1, b + 1);
    reversed[x] ^= 1;
    join3(root, x, y);
}

```

## 1.7 Trie

```

const int maxnode = 4000 * 100 + 10;
const int sigma_size = 26;

// This template use unnecessary large memory.
// should replace ch[maxnode][sigma_size] by vector<node>.
struct Trie {
    int ch[maxnode][sigma_size];

```

```

    int val[maxnode];
    int sz; // the number of node
    void clear() { sz = 1; memset(ch[0], 0, sizeof(ch[0])); }
    int idx(char c) { return c - 'a'; }

    // insert string s, with additional information v
    // v has to be non-zero, zero means "this node is not word
    node"
    void insert(const char *s, int v) {
        int u = 0, n = strlen(s);
        for(int i = 0; i < n; i++) {
            int c = idx(s[i]);
            if(!ch[u][c]) { // the node not exist
                memset(ch[sz], 0, sizeof(ch[sz]));
                val[sz] = 0;
                ch[u][c] = sz++;
            }
            u = ch[u][c]; // going down
        }
        val[u] = v;
    }
};

```

## 2 Graph Theory

### 2.1 Articulation Points and Bridges

```

vi dfs_low; // additional information for articulation
           points/bridges/SCCs
vi articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;
int DFS_WHITE = -1; // unvisited

void articulationPointAndBridge(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u]
    <= dfs_num[u]
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];

```

```

if (dfs_num[v.first] == DFS_WHITE) {                // a
    tree edge
    dfs_parent[v.first] = u;
    if (u == dfsRoot) rootChildren++; // special case, count
    children of root

    articulationPointAndBridge(v.first);

    if (dfs_low[v.first] >= dfs_num[u])              // for
        articulation point
        articulation_vertex[u] = true;              // store this
        information first
    if (dfs_low[v.first] > dfs_num[u])                // for
        bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
        dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); // update
        dfs_low[u]
}
else if (v.first != dfs_parent[u]) // a back edge and not
    direct cycle
    dfs_low[u] = min(dfs_low[u], dfs_num[v.first]); // update
    dfs_low[u]
} }

//inside int main()
printThis("Articulation Points & Bridges (the input graph
    must be UNDIRECTED)");
dfsNumberCounter = 0; dfs_num.assign(V, DFS_WHITE);
dfs_low.assign(V, 0);
dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
printf("Bridges:\n");
for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE) {
        dfsRoot = i; rootChildren = 0;
        articulationPointAndBridge(i);
        articulation_vertex[dfsRoot] = (rootChildren > 1); } //
        special case
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
    if (articulation_vertex[i])

```

```
printf(" Vertex %d\n", i);
```

## 2.2 Tarjan's Algorithm

```

vi S, visited;                                     // additional global
variables
int numSCC;

void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u]
    <= dfs_num[u]
    S.push_back(u);                                // stores u in a vector based on order
    of visitation
    visited[u] = 1;
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];
        if (dfs_num[v.first] == DFS_WHITE)
            tarjanSCC(v.first);
        if (visited[v.first])                      // condition
            for update
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
    }

    if (dfs_low[u] == dfs_num[u]) {                // if this is a root
        (start) of an SCC
        printf("SCC %d:", ++numSCC);               // this part is done
        after recursion
        while (1) {
            int v = S.back(); S.pop_back(); visited[v] = 0;
            printf(" %d", v);
            if (u == v) break;
        }
        printf("\n");
    } }

//inside int main()
printThis("Strongly Connected Components (the input graph
    must be DIRECTED)");

```



---

```

dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0);
    visited.assign(V, 0);
dfsNumberCounter = numSCC = 0;
for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
        tarjanSCC(i);

```

---

## 2.3 Bipartite Graph Check

---

```

queue<int> q; q.push(s);
vi color(V, INF); color[s] = 0;
bool isBipartite = true;
while (!q.empty() & isBipartite){
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++){
        ii v = AdjList[u][j];
        if (color[v.first] == INF){
            color[v.first] = 1 - color[u];
            q.push(v.first);
        }
        else if (color[v.first] == color[u]){
            isBipartite = false; break;
        }
    }
}

```

---

## 2.4 Kruskal's Algorithm

---

```

vector< pair<int, ii> > EdgeList; // (weight, two vertices)
                                // of the edge
for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w); // read the triple: (u,
    // v, w)
    EdgeList.push_back(make_pair(w, ii(u, v))); // (w, u,
    // v)
    AdjList[u].push_back(ii(v, w));
    AdjList[v].push_back(ii(u, w));
}

```

---

```

sort(EdgeList.begin(), EdgeList.end()); // sort by edge
                                // weight O(E log E)
                                // note: pair object has built-in comparison
                                // function
int mst_cost = 0;
UnionFind UF(V); // all V are disjoint sets
initially
for (int i = 0; i < E; i++) { // for each edge,
    // O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second))
        { // check
            mst_cost += front.first; // add the weight of e
            // to MST
            UF.unionSet(front.second.first, front.second.second); //
            // link them
        }
    // note: the runtime cost of UFDS is
    // very light

    // note: the number of disjoint sets must eventually be 1 for
    // a valid MST
printf("MST cost = %d (Kruskal's)\n", mst_cost);

```

---

## 2.5 Prim's Algorithm

---

```

vi taken; // global boolean flag to
// avoid cycle
priority_queue<ii> pq; // priority queue to help choose
// shorter edges

void process(int vtx) { // so, we use -ve sign to reverse the
    // sort order
    taken[vtx] = 1;
    for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
        ii v = AdjList[vtx][j];
        if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
    }
    // sort by (inc) weight then by
    // (inc) id
}

```

---

---

```
// inside int main() --- assume the graph is stored in AdjList,
    pq is empty
    taken.assign(V, 0);           // no vertex is taken at the
        beginning
    process(0); // take vertex 0 and process all edges incident
        to vertex 0
    mst_cost = 0;
    while (!pq.empty()) { // repeat until V vertices (E=V-1
        edges) are taken
        ii front = pq.top(); pq.pop();
        u = -front.second, w = -front.first; // negate the id and
            weight again
        if (!taken[u])           // we have not connected this
            vertex yet
            mst_cost += w, process(u); // take u, process all edges
                incident to u
    }                             // each edge is in pq only
        once!
    printf("MST cost = %d (Prim's)\n", mst_cost);
```

---

## 2.6 Dijkstra's Algorithm

---

```
// Dijkstra routine
vi dist(V, INF); dist[s] = 0;           // INF = 1B to
    avoid overflow
priority_queue< ii, vector<ii>, greater<ii> > pq;
    pq.push(ii(0, s));
        // ^to sort the pairs by increasing
            distance from s
while (!pq.empty()) {
    main loop
    ii front = pq.top(); pq.pop(); // greedy: pick shortest
        unvisited vertex
    int d = front.first, u = front.second;
    if (d > dist[u]) continue; // this check is important, see
        the explanation
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];           // all outgoing
```

```
        edges from u
        if (dist[u] + v.second < dist[v.first]) {
            dist[v.first] = dist[u] + v.second;           // relax
                operation
            pq.push(ii(dist[v.first], v.first));
        } } // note: this variant can cause duplicate items in the
            priority queue
```

---

## 2.7 Bellman Ford's Algorithm

---

```
// Bellman Ford routine
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++) // relax all E edges V-1
    times, overall O(VE)
for (int u = 0; u < V; u++)           // these two
    loops = O(E)
for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j]; // we can record SP spanning
        here if needed
    dist[v.first] = min(dist[v.first], dist[u] + v.second);
        // relax
}
```

---

## 2.8 Floyd Warshall's Algorithm

---

```
for (int k = 0; k < V; k++) // common error: remember that
    loop order is k->i->j
for (int i = 0; i < V; i++)
for (int j = 0; j < V; j++)
    AdjMatrix[i][j] = min(AdjMatrix[i][j], AdjMatrix[i][k] +
        AdjMatrix[k][j]);
```

---

## 2.9 Shortest Path Faster Algorithm

---

```

// SPFA from source S
// initially, only S has dist = 0 and in the queue
vi dist(n, INF); dist[S] = 0;
queue<int> q; q.push(S);
vi in_queue(n, 0); in_queue[S] = 1;

while (!q.empty()) {
    int u = q.front(); q.pop(); in_queue[u] = 0;
    for (j = 0; j < (int)AdjList[u].size(); j++) { // all
        outgoing edges from u
        int v = AdjList[u][j].first, weight_u_v =
            AdjList[u][j].second;
        if (dist[u] + weight_u_v < dist[v]) { // if can relax
            dist[v] = dist[u] + weight_u_v; // relax
            if (!in_queue[v]) { // add to the queue only if it's
                not in the queue
                q.push(v);
                in_queue[v] = 1;
            }}
    }
}

```

---

## 2.10 Network Flow

---

```

void augment(int v, int min_edge){
    if (v == s){
        flow = min_edge;
        return;
    }
    else if (parent[v] != -1){
        int u = parent[v];
        augment(u, min(min_edge, residue[u][v]));
        residue[u][v] -= flow;
        residue[v][u] += flow;
    }
}

void Dinic(){
    max_flow = 0;
    while (true){
        parent.assign(V, -1);

```

```

vector<bool> visited(V, false);
queue<int> q;
q.push(s);
visited[s] = true;
while (!q.empty()){
    int u = q.front();
    q.pop();
    if (u == t)
        break;
    for (int v : adjList[u])
        if (!visited[v] && residue[u][v] > 0){
            parent[v] = u;
            visited[v] = true;
            q.push(v);
        }
    int new_flow = 0;
    for (int u : adjList[t]){
        if (residue[u][t] <= 0)
            continue;
        flow = 0;
        augment(u, residue[u][t]);
        residue[u][t] -= flow;
        residue[t][u] += flow;
        new_flow += flow;
    }
    if (new_flow == 0)
        break;
    max_flow += new_flow;
}
}

```

---

## 2.11 Euler Tour

---

```

void Euler_tour(int u, list<int> &tour, list<int>::iterator it,
    vector<vector<pair<int, bool>>> &adj_list) {
    for (auto &edge : adj_list[u]) {
        if (edge.second) {
            int v = edge.first;
            edge.second = false;

```

```

    for (auto &bi_edge : adj_list[v])
        if (bi_edge.first == u && bi_edge.second) {
            bi_edge.second = false;
            break;
        }
    Euler_tour(v, tour, tour.insert(it, u), adj_list);
}
}

```

## 2.12 Max Cardinality Bipartite Matching

```

int N, M, P, limit;
#define MAXN 50500
#define MAXE 150500
int pair_left[MAXN], pair_right[MAXN], dist_left[MAXN],
    dist_right[MAXN];
bool visited[MAXN];
int adjlist[MAXN];
int node[MAXE];
int link[MAXE];
bool BFS() {
    queue<int> q;
    memset(dist_right, -1, sizeof dist_right);
    memset(dist_left, -1, sizeof dist_left);
    for (int i = 0; i < N; i++) {
        if (pair_left[i] == -1) {
            dist_left[i] = 0;
            q.push(i);
        }
    }
    limit = INT_MAX;
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        if (dist_left[u] > limit)
            break;
        for (int i = adjlist[u]; i != -1; i = link[i]) {
            int v = node[i];
            if (dist_right[v] == -1) {
                dist_right[v] = dist_left[u] + 1;

```

```

                if (pair_right[v] == -1)
                    limit = dist_right[v];
                else {
                    dist_left[pair_right[v]] = dist_right[v] + 1;
                    q.push(pair_right[v]);
                }
            }
        }
        return limit != INT_MAX;
    }
}

```

```

bool DFS(int u) {
    for (int i = adjlist[u]; i != -1; i = link[i]) {
        int v = node[i];
        if (!visited[v] && dist_right[v] == dist_left[u] + 1) {
            visited[v] = true;
            if (pair_right[v] != -1 && dist_right[v] == limit)
                continue;
            if (pair_right[v] == -1 || DFS(pair_right[v])) {
                pair_right[v] = u;
                pair_left[u] = v;
                return true;
            }
        }
    }
    return false;
}

```

```

int main() {
    scanf("%d %d %d", &N, &M, &P);

    memset(pair_left, -1, sizeof pair_left);
    memset(pair_right, -1, sizeof pair_right);
    memset(link, -1, sizeof link);
    memset(adjlist, -1, sizeof adjlist);

    for (int i = 0; i < P; i++) {
        int u, v;
        scanf("%d %d", &u, &v);
        node[i] = v - 1;
        link[i] = adjlist[u - 1];
        adjlist[u - 1] = i;
    }
    int matching = 0;

```

```

while (BFS()) {
    memset(visited, 0, sizeof visited);
    for (int i = 0; i < N; i++)
        if (pair_left[i] == -1)
            if (DFS(i))
                matching++;
}
printf("%d\n", matching);
return 0;
}

```

## 2.13 Min-Cost Flow

```

const int inf = 1000000000;
int s, t, node, totalCost;
vector<int> head, dist, vtx, next, c, cost;
vector<bool> vis;
void resize(vector<T> &a, int size, T init) //设大小、初始值
void init(int source, int target, int nodeCount)
    //初始化, 记得清空
void add(int a, int b, int cc, int cst) //双向加边
void spfa() {
    resize(vis, node, false); resize(dist, node, -inf);
    queue<int> q; q.push(t); vis[t]=true; dist[t]=0;
    while (q.size()) {
        int u = q.front(); q.pop();
        vis[u] = false;
        for (int p = head[u]; p != -1; p = next[p]) {
            if (c[p ^ 1] && dist[u] + cost[p ^ 1] > dist[vtx[p]]) {
                dist[vtx[p]] = dist[u] + cost[p ^ 1];
                if (!vis[vtx[p]]) {
                    vis[vtx[p]] = true; q.push(vtx[p]);
                    if (dist[q.back()] < dist[q.front()]) {
                        swap(q.front(), q.back());
                    }
                }
            }
        }
    }
}
int dfs(int u, int limit) {
    if (u == t) {
        totalCost += limit * dist[s];
    }
}

```

```

        return limit;
    }
    int current = 0;
    vis[u] = true;
    for (int p = head[u]; p != -1; p = next[p]) {
        if (c[p] && !vis[vtx[p]] && dist[vtx[p]] + cost[p] ==
            dist[u]) {
            int delta = dfs(vtx[p], min(limit - current, c[p]));
            c[p] -= delta; c[p ^ 1] += delta;
            current += delta;
            if (current == limit) {
                break;
            }
        }
    }
}
return current;
}
inline bool adjust() {
    int maxi = -inf;
    for (int i = 0; i < node; ++i) {
        if (vis[i]) {
            for (int p = head[i]; p != -1; p = next[p]) {
                if (c[p] && !vis[vtx[p]]) {
                    assert(dist[vtx[p]] + cost[p] != dist[i]);
                    maxi = max(maxi, dist[vtx[p]] + cost[p] - dist[i]);
                }
            }
        }
    }
    if (maxi == -inf) {
        return false;
    }
    for (int i = 0; i < node; ++i) {
        if (vis[i]) {
            dist[i] += maxi;
        }
    }
    return true;
}
int maxCostFlow() {

```

```

spfa();
totalCost = 0;
do{
    do{
        resize(vis, node, false);
    }while (dfs(s, inf));
}while (adjust());
return totalCost;
}

```

---

## 3 Math

### 3.1 Sieve of Eratosthenes

```

#define BOUND 1000000
bitset<BOUND> bs;
vector<long long> primes;
void sieve() {
    bs.set();
    bs[0] = bs[1] = 0;
    for (long long i = 2; i <= BOUND; i++) {
        if (bs[i]) {
            for (long long j=i*i; j<=BOUND; j+=i) bs[j] = 0;
            primes.push_back(i);
        }
    }
}

```

---

### 3.2 Euler Phi function

```

int euler_phi(int n){
    int m = (int)sqrt(n+0.5);
    int ans = n;
    for(int i=2;i<=m;i++){
        if(n%i==0){
            ans = ans/i*(i-1);
            while(n%i==0) n /= i;
        }
    }
    if(n>1) ans = ans/n*(n-1);
    return ans;
}

```

---

```

}
if(n>1) ans = ans/n*(n-1);
return ans;}
void euler_phi_table(int n, int *phi){
    for(int i=2;i<=n;i++) phi[i] = 0;
    phi[1] = 1;
    for(int i=2;i<=n;i++){
        if(!phi[i])
            for(int j=i;j<=n;j+=i){
                if(!phi[j]) phi[j] = j;
                phi[j] = phi[j]/i*(i-1);
            }
    }
}

```

---

### 3.3 GCD mod related (CRT)

```

//ax+by=gcd(a,b), min abs(x)+abs(y) x,y may be negative
void gcd(LL a, LL b, LL & d, LL & x, LL & y) {
    if(!b) { d = a; x = 1; y = 0; }
    else { gcd(b, a%b, d, y, x); y-=x*(a/b); }
}
// calculate inv(a) mod n. If not exist, return -1
LL inv(LL a, LL n) {
    LL d, x, y; gcd(a, n, d, x, y);
    return d == 1 ? (x+n)%n : -1; }
// n functions: x=a[i] (mod m[i]) m[i] co-prime
LL CRT(int n, int * a, int * m) {
    LL M = 1, d, y, x = 0;
    for(int i=0;i<n;i++) M *= m[i];
    for(int i=0;i<n;i++) {
        LL w = M / m[i];
        gcd(m[i], w, d, d, y);
        x = (x + y*w*a[i]) % M;
    }
    return (x+M)%M; }
// solve a^x=b mod n. n prime. If no solution, return -1
int log_mod(int a, int b, int n) {
    int m, v, e = 1;
    m = (int)sqrt(n+0.5);
    v = inv(pow_mod(a, m, n), n);
}

```

```

map<int, int> x; x[1] = 0;
for(int i=1;i<m;i++) {
    e = mul_mod(e, a, n);
    if(!x.count(e)) x[e] = i;
}
for(int i=0;i<m;i++) {
    if(x.count(b)) return i*m + x[b];
    b = mul_mod(b, v, n);
}
return -1;}

```

---

### 3.4 Enumerate Combination

```

const int maxn = 1000;
int com[maxn];
bool next_Com(int num, int k){ //0,1...num-1 choose k
    if(k == 0) return false;
    if(com[k-1]!=num-1){ com[k-1]++; return true;}
    int i;
    for(i=k-1;i>=0;i--)
        if(com[i]!=num-k+i) break;
    if(i===-1) return false;
    com[i]++;
    for(int j=i+1;j<k;j++)
        com[j] = com[i]+(j-i);
    return true; }
void makeFirstCom(int k){
    for(int i=0;i<k;i++) com[i] = i;
}

```

---

### 3.5 Gauss Elimination

```

const int maxn = 110;
typedef double Matrix[maxn][maxn];
// require matrix A invertible
// A is augmented matrix, A[i][n] = bi

```

---

// After execution, A[i][n] is the value of i-th variable

```

void gauss_elimination(Matrix A, int n) {
    int i, j, k, r;
    for (i=0; i<n; i++) {
        r = i;
        for (j=i+1; j<n; j++) {
            if (fabs(A[j][i]) > fabs(A[r][i])) r = j;
        }
        if (r != i)
            for (j=0; j<=n; j++)
                swap(A[r][j], A[i][j]);
        for (j=n; j>=i; j--)
            for (k=i+1; k<n; ++k)
                A[k][j] -= A[k][i] / A[i][i] * A[i][j];
    }
    for (i=n-1; i>=0; i--) {
        for (j=i+1; j<n; j++)
            A[i][n] -= A[j][n] * A[i][j];
        A[i][n] /= A[i][i];
    }
}

```

---

### 3.6 FFT

```

const long double PI = acos(0.0) * 2.0;
typedef complex<double> CD;
inline void FFT(vector<CD> &a, bool inverse) {
    int n = a.size();
    for(int i = 0, j = 0; i < n; i++) {
        if(j > i) swap(a[i], a[j]);
        int k = n;
        while(j & (k >>= 1)) j &= ~k;
        j ^= k;
    }
    double pi = inverse ? -PI : PI;
    for(int step = 1; step < n; step <= 1) {
        double alpha = pi / step;
        for(int k = 0; k < step; k++) {
            CD omegak = exp(CD(0, alpha*k));

```

```

    for(int Ek = k; Ek < n; Ek += step << 1) {
        int Ok = Ek + step;
        CD t = omegak * a[Ok];
        a[Ok] = a[Ek] - t;
        a[Ek] += t;
    }
}
if(inverse)
    for(int i = 0; i < n; i++) a[i] /= n;
}
inline vector<double> operator * (const vector<double>& v1,
    const vector<double>& v2) {
    int s1 = v1.size(), s2 = v2.size(), S = 2;
    while(S < s1 + s2) S <<= 1;
    vector<CD> a(S,0), b(S,0);
    for(int i = 0; i < s1; i++) a[i] = v1[i];
    FFT(a, false);
    for(int i = 0; i < s2; i++) b[i] = v2[i];
    FFT(b, false);
    for(int i = 0; i < S; i++) a[i] *= b[i];
    FFT(a, true);
    vector<double> res(s1 + s2 - 1);
    for(int i = 0; i < s1 + s2 - 1; i++) res[i] = a[i].real();
    return res;
} // 用FFT实现的快速多项式乘法

```

### 3.7 Simplex

//输入矩阵a描述线性规划的标准形式。a为m+1行n+1列，其中行0~m-1为不等式  
 //行m为目标函数（最大化），列0~n-1为变量0~n-1的系数，列n为常数项  
 //第i个约束为 $a[i][0]*x[0] + a[i][1]*x[1] + \dots \leq a[i][n]$   
 //目标为 $\max(a[m][0]*x[0] + a[m][1]*x[1] + \dots + a[m][n-1]*x[n-1] - a[m][n])$   
 //注意：变量均有非负约束 $x[i] \geq 0$   
 const int maxm = 500; // 约束数目上限  
 const int maxn = 500; // 变量数目上限  
 const double INF = 1e100;

```

const double eps = 1e-10;
struct Simplex {
    int n; // 变量个数
    int m; // 约束个数
    double a[maxm][maxn]; // 输入矩阵
    int B[maxm], N[maxn]; // 算法辅助变量
    void pivot(int r, int c) {
        swap(N[c], B[r]);
        a[r][c] = 1 / a[r][c];
        for(int j = 0; j <= n; j++) if(j != c) a[r][j] *= a[r][c];
        for(int i = 0; i <= m; i++) if(i != r) {
            for(int j = 0; j <= n; j++) if(j != c) a[i][j] -= a[i][c]
                * a[r][j];
            a[i][c] = -a[i][c] * a[r][c];
        }
    }
    bool feasible() {
        for(;;) {
            int r, c;
            double p = INF;
            for(int i = 0; i < m; i++) if(a[i][n] < p) p = a[r = i][n];
            if(p > -eps) return true;
            p = 0;
            for(int i = 0; i < n; i++) if(a[r][i] < p) p = a[r][c = i];
            if(p > -eps) return false;
            p = a[r][n] / a[r][c];
            for(int i = r+1; i < m; i++) if(a[i][c] > eps) {
                double v = a[i][n] / a[i][c];
                if(v < p) { r = i; p = v; }
            }
            pivot(r, c);
        }
    }
}
//解有界返回1，无解返回0，无界返回-1。b[i]为x[i]的值，ret为目标函数的值
int simplex(int n, int m, double x[maxn], double& ret) {
    this->n = n;
    this->m = m;
    for(int i = 0; i < n; i++) N[i] = i;
    for(int i = 0; i < m; i++) B[i] = n+i;
    if(!feasible()) return 0;

```



```

for(;;) {
    int r, c;
    double p = 0;
    for(int i = 0; i < n; i++) if(a[m][i] > p) p = a[m][c = i];
    if(p < eps) {
        for(int i = 0; i < n; i++) if(N[i] < n) x[N[i]] = 0;
        for(int i = 0; i < m; i++) if(B[i] < n) x[B[i]] =
            a[i][n];
        ret = -a[m][n];
        return 1;
    }
    p = INF;
    for(int i = 0; i < m; i++) if(a[i][c] > eps) {
        double v = a[i][n] / a[i][c];
        if(v < p) { r = i; p = v; }
    }
    if(p == INF) return -1;
    pivot(r, c);
}
};

```

### 3.8 Pell Function

```

//求x^2-ny^2=1的最小正整数根,n不是完全平方数
p[1]=1;p[0]=0; q[1]=0;q[0]=1; a[2]=(int)(floor(sqrt(n)+1e-7));
g[1]=0;h[1]=1;
for (int i=2;i;++i) {
    g[i]=-g[i-1]+a[i]*h[i-1]; h[i]=(n-sqr(g[i]))/h[i-1];
    a[i+1]=(g[i]+a[2])/h[i]; p[i]=a[i]*p[i-1]+p[i-2];
    q[i]=a[i]*q[i-1]+q[i-2]; 检查p[i],q[i]是否为解, 如果是, 则退出
}

```

### 3.9 二次剩余

```

/*a*x^2+b*x+c==0 (mod P) 求0..P-1的根 */
int pDiv2,P,a,b,c,Pb,d;

```

```

inline int calc(int x,int Time){
    if (!Time) return 1; int tmp=calc(x,Time/2);
    tmp=(long long)tmp*tmp%P;
    if (Time&1) tmp=(long long)tmp*x%P; return tmp;
}
inline int rev(int x){ if (!x) return 0; return calc(x,P-2);}
inline void Compute(){
    while (1) { b=rand()%(P-2)+2; if (calc(b,pDiv2)+1==P)
        return; }
}
int main(){
    srand(time(0)^312314); int T;
    for (scanf("%d",&T);T;--T) {
        scanf("%d%d%d%d",&a,&b,&c,&P);
        if (P==2) {
            int cnt=0; for (int i=0;i<2;++i) if
                ((a*i+i+b*i+c)%P==0) ++cnt;
            printf("%d",cnt);
            for (int i=0;i<2;++i) if ((a*i+i+b*i+c)%P==0)
                printf(" %d",i);
            puts("");
        }else {
            int delta=(long long)b*rev(a)*rev(2)%P;
            a=(long long)c*rev(a)%P-sqr( (long long)delta )%P;
            a%=P;a+=P;a%=P; a=P-a;a%=P; pDiv2=P/2;
            if (calc(a,pDiv2)+1==P) puts("0");
            else {
                int t=0,h=pDiv2; while (!(h%2)) ++t,h/=2;
                int root=calc(a,h/2);
                if (t>0) { Compute(); Pb=calc(b,h); }
                for (int i=1;i<=t;++i) {
                    d=(long long)root*root*a%P;
                    for (int j=1;j<=t-i;++j) d=(long long)d*d%P;
                    if (d+1==P) root=(long long)root*Pb%P;
                    Pb=(long long)Pb*Pb%P;
                }
                root=(long long)a*root%P;
                int root1=P-root; root-=delta;
                root%=P; if (root<0) root+=P;
                root1-=delta; root1%=P; if (root1<0) root1+=P;
            }
        }
    }
}

```

```

        if (root>root1) { t=root;root=root1;root1=t; }
        if (root==root1) printf("1 %d\n",root);
        else printf("2 %d %d\n",root,root1);
    } } return 0; }

```

### 3.10 Schröder-Hipparchus Number

$$S(n) = \frac{1}{n}((6n-9)S(n-1) - (n-3)S(n-2))$$

### 3.11 Catalan Numbers

$$Cat(n) = \frac{2n!}{n! \times n! \times (n+1)}$$

$$Cat(n+1) = \frac{(2n+2) \times (2n+1)}{(n+2) \times (n+1)} \times Cat(n)$$

## 4 Computational Geometry

```

struct Point{
    double x, y;
    Point(double x=0, double y=0):x(x), y(y){}
};
typedef Point Vector;
// Vector + Vector = Vector / Point + Vector = Point
Vector operator + (Vector A, Vector B){
    return Vector(A.x + B.x, A.y + B.y);}
// Point - Point = Vector
Vector operator - (Point A, Point B){
    return Vector(A.x - B.x, A.y - B.y);}
Vector operator * (Vector A, double p){
    return Vector(A.x * p, A.y * p);}
Vector operator / (Vector A, double p){
    return Vector(A.x / p, A.y / p);}
const double eps = 1e-10;
int dcmp(double x){
    if(fabs(x) < eps) return 0;
    return x < 0 ? -1 : 1; }
bool operator < (const Point& a, const Point& b){

```

```

    return dcmp(a.x - b.x) < 0 || (dcmp(a.x-b.x)==0 && dcmp(a.y -
        b.y) < 0); }
bool operator == (const Point& a, const Point& b){
    return dcmp(a.x-b.x) == 0 && dcmp(a.y-b.y) == 0; }
double Dot(Vector A, Vector B){
    return A.x*B.x + A.y*B.y; }
double Length(Vector A){
    return sqrt(Dot(A,A)); }
// polar angle theta is the counterclockwise angle from the
// x-axis at which a point in the xy-plane lies
// (-pi, pi]
double angle(Vector v) {
    return atan2(v.y, v.x); }
// counterclockwise angle from A to B [0, pi]
double Angle(Vector A, Vector B){
    return acos(Dot(A,B)/Length(A)/Length(B)); }
double Cross(Vector A, Vector B){
    return A.x*B.y - A.y*B.x; }
// counterclockwisely rotate A for rad
Vector Rotate(Vector A, double rad){
    return Vector(A.x*cos(rad)-A.y*sin(rad),
        A.x*sin(rad)+A.y*cos(rad)); }
// unit normal vector for A (left rotate pi/2) A != 0
Vector Normal(Vector A){
    double L = Length(A);
    return Vector(-A.y/L, A.x/L);}
// P+tv,Q+tw should have only one intersection,iff Cross(v,w)!=0
Point GetLineIntersection(Point P, Vector v, Point Q, Vector w){
    Vector u = P-Q;
    double t = Cross(w,u)/Cross(v,w);
    return P+v*t;}
// distance from P to line AB
double DistanceToLine(Point P, Point A, Point B){
    Vector v1 = B-A, v2 = P-A;
    return fabs(Cross(v1,v2))/Length(v1);}
// distance from P to segment AB
double DistanceToSegment(Point P, Point A, Point B){
    if(A == B) return Length(P-A);
    Vector v1 = B-A, v2 = P-A, v3 = P-B;
    if(dcmp(Dot(v1,v2))<0) return Length(v2);}

```

```

    if(dcmp(Dot(v1,v3))>0) return Length(v3);
    return fabs(Cross(v1,v2))/Length(v1);}
Point GetLineProjection(Point P, Point A, Point B){
    Vector v = B-A;
    return A+v*(Dot(v,P-A) / Dot(v,v)); }
// determine segment a1a2 and b1b2 normal intersection (only
// one intersection, not endpoint)
// if allowing intersecting on endpoints:
// 1) c1 = c2 = 0: on the same line, probably intersecting
// 2) otherwise, one endpoint on the other segment (Use
//    OnSegment() method)
bool segmentProperIntersection(Point a1, Point a2, Point b1,
    Point b2){
    double c1 = Cross(a2-a1,b1-a1);
    double c2 = Cross(a2-a1,b2-a1);
    double c3 = Cross(b2-b1,a1-b1);
    double c4 = Cross(b2-b1,a2-b1);
    return dcmp(c1)*dcmp(c2)<0 && dcmp(c3)*dcmp(c4)<0;}
// determine P on segment a1a2 (endpoint excluded)
bool OnSegment(Point p, Point a1, Point a2) {
    return dcmp(Cross(a1-p,a2-p))==0 && dcmp(Dot(a1-p,a2-p))<0;}
// calculate the direct area for polygon (not necessarily
// convex)
double PolygonArea(Point* p, int n) {
    double area = 0;
    for(int i=1;i<n-1;i++)
        area += Cross(p[i]-p[0],p[i+1]-p[0]);
    return area/2;}
// convex hull: n points in array p, ch array for output,
// return the number of points on hull
// no duplicate points in input; the order of input points is
// not preserved
// if want input points on edges of hull, change two <= to <
int ConvexHull(Point* p, int n, Point* ch) {
    sort(p,p+n); int m = 0;
    for(int i=0;i<n;i++){
        while(m>1 && dcmp(Cross(ch[m-1]-ch[m-2], p[i]-ch[m-2])) <= 0)
            m--;
        ch[m++] = p[i];}
    int k = m;

```

```

    for(int i=n-2;i>=0;i--){
        while(m>k && dcmp(Cross(ch[m-1]-ch[m-2], p[i]-ch[m-2])) <= 0)
            m--;
        ch[m++] = p[i];}
    if(n>1) m--;
    return m;}
// return the diameter of set of points (Rotating Calipers
// Algorithm)
// ch: already convex hull (no three points in a line) n: the
// number of points
double diameter(Point* ch, int n) {
    if(n == 1) return 0;
    if(n == 2) return Length(ch[0] - ch[1]);
    ch[n] = ch[0];
    double ans = 0;
    for(int u = 0, v = 1; u < n; u++) {
        for(;;) {
            double diff = Cross(ch[u+1]-ch[u], ch[v+1]-ch[v]);
            if(dcmp(diff) <= 0) {
                ans = max(ans, Length(ch[u]-ch[v]));
                if(dcmp(diff) == 0)
                    ans = max(ans, Length(ch[u]-ch[v+1]));
                break;
            } v = (v + 1) % n;
        }
    }
    return ans;}
// poly: polygon n: the number of points
// return value: (-2, vertex) (-1, edges) (0, outside) (1,
// inside)
// determine if point on the left side of all edges (vertex
// already counterclock ordered)
int isPointInPolygon(Point p, Point* poly, int n){
    int wn = 0;
    for(int i=0;i<n;i++){
        if(p == poly[i]) return -2;
        if(OnSegment(p, poly[i], poly[(i+1)%n])) return -1;
        int k = dcmp(Cross(poly[(i+1)%n]-poly[i], p-poly[i]));
        int d1 = dcmp(poly[i].y - p.y);
        int d2 = dcmp(poly[(i+1)%n].y - p.y);
        if(k>0 && d1<=0 && d2>0) wn++;
    }

```

```

    if(k<0 && d2<=0 && d1>0) wn--;
}
if(wn != 0) return 1;
return 0;
}
struct Line{
    Point p; Vector v;
    Line(Point p, Vector v):p(p),v(v){}
    Point point(double t) {return p + v*t;}
    Line move(double d) {return Line(p + Normal(v)*d, v);}
};
struct Circle{
    Point c;
    double r;
    Circle(Point c, double r):c(c),r(r){}
    Point point(double a){return Point(c.x + cos(a)*r, c.y +
        sin(a)*r);}
};
// return number of intersection, sol has all intersection
// intersection  $P = A + t(B-A)$ , simplify to  $et^2+ft+g = 0$ 
int getLineCircleIntersection(Line L, Circle C, double& t1,
    double& t2, vector<Point>& sol){
    double a = L.v.x, b = L.p.x - C.c.x, c = L.v.y, d = L.p.y -
        C.c.y;
    double e = a*a + c*c, f = 2*(a*b+c*d), g = b*b + d*d -
        C.r*C.r;
    double delta = f*f - 4*e*g;
    if(dcmp(delta) < 0) return 0;
    if(dcmp(delta) == 0){
        t1 = t2 = -f / (2*e);
        sol.push_back(L.point(t1));
        return 1; }
    t1 = (-f - sqrt(delta)) / (2*e);
    sol.push_back(L.point(t1));
    t2 = (-f + sqrt(delta)) / (2*e);
    sol.push_back(L.point(t2));
    return 2;}
// return the number of intersection
// if two circle identical, then return -1
int getCircleCircleIntersection(Circle C1, Circle C2,

```

```

    vector<Point>& sol){
    double d = Length(C1.c-C2.c);
    if(dcmp(d) == 0){
        if(dcmp(C1.r-C2.r) == 0) return -1;
        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));
    // angle from C1C2 to C1P1
    Point p1 = C1.point(a-da), p2 = C1.point(a+da);
    sol.push_back(p1);
    if(p1 == p2) return 1;
    sol.push_back(p2);
    return 2;}
// tangent lines from P to C
// v[i]: i-th tangent lines, return the number of tangent lines
int getTangents(Point p, Circle C, Vector* v){
    Vector u = C.c - p;
    double dist = Length(u);
    if(dist < C.r) return 0;
    else if(dcmp(dist-C.r)==0){
        v[0] = Rotate(u,PI/2);
        return 1;
    } else {
        double ang = asin(C.r / dist);
        v[0] = Rotate(u, -ang); v[1] = Rotate(u, +ang);
        return 2;
    }
}
// return the number of tangents, -1 means inf
// a[i], b[i]: point of tangency with i-th tangent on A, B;
// same when internally or externally tangent
int getTangents(Circle A, Circle B, Point* a, Point* b) {
    int cnt = 0;
    if(A.r < B.r){ swap(A, B); swap(a, b); }
    double d2 = (A.c.x-B.c.x)*(A.c.x-B.c.x) +
        (A.c.y-B.c.y)*(A.c.y-B.c.y);
    double rdif = A.r - B.r;
    double rsum = A.r + B.r;

```

```

if(dcmp(d2 - rdiffrdiff) < 0) // containing
    return 0;
double base = atan2(B.c.y-A.c.y, B.c.x-A.c.x);
if(dcmp(d2)==0 && dcmp(A.r-B.r)==0) // infinite tangents
    return -1;
if(dcmp(d2-rdiffrdiff) == 0){ // inscribe, one tangent
    a[cnt] = A.point(base); b[cnt] = B.point(base);
    cnt++; return 1;
}
double ang = acos((A.r-B.r)/sqrt(d2)); // two external common
    tangents
a[cnt] = A.point(base + ang);
b[cnt] = B.point(base + ang); cnt++;
a[cnt] = A.point(base - ang);
b[cnt] = B.point(base - ang); cnt++;
if(dcmp(d2-rsum*rsum) == 0){
    a[cnt] = A.point(base);
    b[cnt] = B.point(PI + base); cnt++;
}
else if(dcmp(d2 - rsum*rsum) > 0){ // two internal common
    tangents
    double ang = acos((A.r+B.r) / sqrt(d2));
    a[cnt] = A.point(base+ang);
    b[cnt] = B.point(PI+base+ang); cnt++;
    a[cnt] = A.point(base-ang);
    b[cnt] = B.point(PI+base-ang); cnt++;
}
return cnt;}

void CircleCenter(point p0 , point p1 , point p2 , point &cp ){
    double a1=p1.x-p0.x , b1=p1.y-p0.y , c1=(sqr(a1)+sqr(b1)) /
        2 ;
    double a2=p2.x-p0.x , b2=p2.y-p0.y , c2=(sqr(a2)+sqr(b2)) /
        2 ;
    double d = a1*b2 - a2*b1 ;
    cp.x = p0.x + ( c1*b2 - c2*b1 ) / d ;
    cp.y = p0.y + ( a1*c2 - a2*c1 ) / d ;}

double Incenter(point A, point B, point C, point &cp ){
    double s , p , r , a , b , c ;
    a = dis(B, C) , b = dis(C, A) , c = dis(A, B) ; p = (a +b +c)

```

```

    / 2 ;
    s = sqrt ( p * ( p-a ) * ( p-b ) * ( p-c ) ) ; r = s / p ;
    cp.x = ( a*A.x + b*B. x + c*C.x ) / ( a + b + c ) ;
    cp.y = ( a*A.y + b*B. y + c*C.y ) / ( a + b + c ) ;
    return r ;}

void Orthocenter(point A, point B, point C, point &cp ){
    CircleCenter(A, B, C, cp );
    cp.x = A.x + B.x + C.x - 2 * cp.x ;cp.y = A.y + B.y + C.y - 2
        * cp.y ;}

double twoCircleAreaUnion(point a, point b , double r1, double
    r2){
    if (r1+r2<=(a-b).dist()) return 0;
    if (r1+(a-b).dist()<=r2) return pi*r1*r1;
    if (r2+(a-b).dist()<=r1) return pi*r2*r2;
    double c1, c2, ans=0;
    c1=(r1*r1-r2*r2+(a-b).dis()/(a-b).dist()/r1/2.0;
    c2=(r2*r2-r1*r1+(a-b).dis()/(a-b).dist()/r2/2.0;
    double s1,s2; s1=acos(c1); s2=acos(c2);
    ans+=s1*r1*r1-r1*r1*sin(s1)*cos(s1);
    ans+=s2*r2*r2-r2*r2*sin(s2)*cos(s2);
    return ans;
}////==两圆面积交 dist=是距离, dis是平方

double area2(point pa, point pb) {
    if (pa.len() < pb.len()) swap(pa, pb); if (pb.len() < eps)
        return 0;
    double a, b, c, B, C, sinB, cosB, sinC, cosC, S, h, theta;
    a = pb.len(); b = pa.len(); c = (pb-pa).len();
    cosB=dot(pb,pb-pa)/a/c; sinB=fabs(det(pb,pb-pa)/a/c);
    cosC=dot(pa, pb) / a / b; sinC=fabs(det(pa,pb)/a/b);
    B=atan2(sinB , cosB); C=atan2(sinC, cosC);
    if (a > r) { S = C/2*r*r; h = a*b*sinC/c;
        if (h < r && B < PI/2) S -= (acos(h/r)*r*r -
            h*sqrt(r*r-h*h));
    }
    else if (b > r) { theta = PI - B - asin(sinB/r*a);
        S = .5*a*r*sin(theta) + (C-theta)/2*r*r; }
    else S = .5*sinC*a*b; return S; }// a, b, c, r fixed

double area(const point &o) {

```

```

double S = 0; point oa = a-o, ob = b-o, oc = c-o;
S += area2(oa, ob) * sign(det(oa, ob));
S += area2(ob, oc) * sign(det(ob, oc));
S += area2(oc, oa) * sign(det(oc, oa)); return abs(S);
}

```

//=====多边形和圆相交的面积用有向面积,划分成一个三角形和圆的面积之交}

随机增量最小覆盖圆

```

const double eps=1e-7;
const int maxn=100000;
class circle{
    point o;
    double r;
}
point a[maxn];
int n;
circle ans;
double area(point a, point b, point c){
    return ((b.x-a.x)*(c.y-a.y)-(b.y-a.y)*(c.x-a.x));
}
double dis(point a, point b){
    return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y);
}
void init(){
    int i,j,k;
    scanf("%d",&n);
    rep(i,n) scanf("%lf%lf",&a[i].x,&a[i].y);
}
bool check(const point &a){
    return sqr(a.x-ans.o.x) + sqr(a.y-ans.o.y) <= ans.r + zero;
}
void Mincircle(){
    int i,j,k;
    ans.r=0; ans.x=0; ans.y=0;
    rep(i,n) if (!check(a[i])) {
        ans.o=a[i]; ans.r=0;
        rep(j,i) if (!check(a[j])) {
            CircleCenter(a[i],a[j],ans.o);
            28ans.r=dis(ans.o,a[i]);

```

```

        rep(k,j) if (!check(a[k])) {
            CircleCenter(a[i],a[j],a[k],ans.o);
            ans.r=dis(ans.o,a[i]);
        }
    }
    printf("%.4lf\n",sqrt(ans.r));
}

```

## 5 String Processing

### 5.1 KMP

```
#define MAX_N 100010
```

```

char T[MAX_N], P[MAX_N]; // T = text, P = pattern
int b[MAX_N], n, m; // b = back table, n = length of T, m =
length of P

```

```

void kmpPreprocess() { // call this before calling kmpSearch()
    int i = 0, j = -1; b[0] = -1; // starting values
    while (i < m) { // pre-process the pattern string P
        while (j >= 0 && P[i] != P[j]) j = b[j]; // if different,
            reset j using b
        i++; j++; // if same, advance both pointers
        b[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2,
            3, 4
    } } // in the example of P = "SEVENTY SEVEN" above

```

```

void kmpSearch() { // this is similar as kmpPreprocess(), but
    on string T
    int i = 0, j = 0; // starting values
    while (i < n) { // search through string T
        while (j >= 0 && T[i] != P[j]) j = b[j]; // if different,
            reset j using b
        i++; j++; // if same, advance both pointers
        if (j == m) { // a match found when j == m

```

```

    printf("P is found at index %d in T\n", i - j);
    j = b[j]; // prepare j for the next possible match
} } }

```

## 5.2 Suffix Array

```

#define MAX_N 100010 // second approach: O(n log n)
char T[MAX_N]; // the input string, up to 100K characters
int n; // the length of input string
int RA[MAX_N], tempRA[MAX_N]; // rank array and temporary rank
    array
int SA[MAX_N], tempSA[MAX_N]; // suffix array and temporary
    suffix array
int c[MAX_N]; // for counting/radix sort

char P[MAX_N]; // the pattern string (for string matching)
int m; // the length of pattern string

int Phi[MAX_N]; // for computing longest common prefix
int PLCP[MAX_N];
int LCP[MAX_N]; // LCP[i] stores the LCP between previous
    suffix T+SA[i-1]
    // and current suffix
    T+SA[i]

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; } //
    compare

void constructSA_slow() { // cannot go beyond 1000
    characters
    for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1,
        2, ..., n-1}
    sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) =
        O(n^2 log n)
}

void countingSort(int k) { //
    O(n)

```

```

int i, sum, maxi = max(300, n); // up to 255 ASCII chars or
    length of n
memset(c, 0, sizeof c); // clear frequency
    table
for (i = 0; i < n; i++) // count the frequency of each
    integer rank
    c[i + k < n ? RA[i + k] : 0]++;
for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t;
}
for (i = 0; i < n; i++) // shuffle the suffix array if
    necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
for (i = 0; i < n; i++) // update the suffix
    array SA
    SA[i] = tempSA[i];
}

void constructSA() { // this version can go up to 100000
    characters
int i, k, r;
for (i = 0; i < n; i++) RA[i] = T[i]; // initial
    rankings
for (i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2,
    ..., n-1}
for (k = 1; k < n; k <= 1) { // repeat sorting process log n
    times
    countingSort(k); // actually radix sort: sort based on the
        second item
    countingSort(0); // then (stable) sort based on the
        first item
    tempRA[SA[0]] = r = 0; // re-ranking; start from rank
        r = 0
    for (i = 1; i < n; i++) // compare adjacent
        suffixes
        tempRA[SA[i]] = // if same pair => same rank r; otherwise,
            increase r
            (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k])
                ? r : ++r;
    for (i = 0; i < n; i++) // update the rank

```

```

    array RA
    RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;          // nice optimization
    trick
} }

void computeLCP_slow() {
    LCP[0] = 0;                             // default value
    for (int i = 1; i < n; i++) {           // compute LCP by definition
        int L = 0;                          // always reset L to 0
        while (T[SA[i] + L] == T[SA[i-1] + L]) L++; // same L-th
        char, L++
        LCP[i] = L;
    } }

void computeLCP() {
    int i, L;
    Phi[SA[0]] = -1; // default value
    for (i = 1; i < n; i++) // compute Phi in O(n)
        Phi[SA[i]] = SA[i-1]; // remember which suffix is behind
        this suffix
    for (i = L = 0; i < n; i++) { // compute Permuted LCP in O(n)
        if (Phi[i] == -1) { PLCP[i] = 0; continue; } // special case
        while (T[i + L] == T[Phi[i] + L]) L++; // L increased max n
        times
        PLCP[i] = L;
        L = max(L-1, 0); // L decreased max n times
    }
    for (i = 0; i < n; i++) // compute LCP in O(n)
        LCP[i] = PLCP[SA[i]]; // put the permuted LCP to the correct
        position
    }

ii stringMatching() { // string matching in O(m log n)
    int lo = 0, hi = n-1, mid = lo; // valid matching = [0..n-1]
    while (lo < hi) { // find lower bound
        mid = (lo + hi) / 2; // this is round down
        int res = strcmp(T + SA[mid], P, m); // try to find P in
        suffix 'mid'
        if (res >= 0) hi = mid; // prune upper half (notice the
        >= sign)
    }
    else lo = mid + 1; // prune lower half including mid
    } // observe '=' in "res >= 0" above
    if (strcmp(T + SA[lo], P, m) != 0) return ii(-1, -1); // if
    not found
    ii ans; ans.first = lo;
    lo = 0; hi = n - 1; mid = lo;
    while (lo < hi) { // if lower bound is found, find upper
        bound
        mid = (lo + hi) / 2;
        int res = strcmp(T + SA[mid], P, m);
        if (res > 0) hi = mid; // prune upper half
        else lo = mid + 1; // prune lower half including
        mid
    } // (notice the selected branch when res == 0)
    if (strcmp(T + SA[hi], P, m) != 0) hi--; // special case
    ans.second = hi;
    return ans;
} // return lower/upperbound as first/second item of the pair,
    respectively

ii LRS() { // returns a pair (the LRS length and its index)
    int i, idx = 0, maxLCP = -1;
    for (i = 1; i < n; i++) // O(n), start from i = 1
        if (LCP[i] > maxLCP)
            maxLCP = LCP[i], idx = i;
    return ii(maxLCP, idx);
}

int owner(int idx) { return (idx < n-m-1) ? 1 : 2; }

ii LCS() { // returns a pair (the LCS length and its index)
    int i, idx = 0, maxLCP = -1;
    for (i = 1; i < n; i++) // O(n), start from i = 1
        if (owner(SA[i]) != owner(SA[i-1]) && LCP[i] > maxLCP)
            maxLCP = LCP[i], idx = i;
    return ii(maxLCP, idx);
}

```