

algorithm template

zinan.xu.dev@gmail.com

1st Jan., 2026

Contents

数论	3
素性检测	3
大合数分解	4
二次剩余-膜意义下开方	5
Nim积	7
树	9
树的直径	9
最近公共祖先	10
树同构	12
图论	15
单源最短路	15
有向图-强连通分量	16
计算几何	19
vector重排	19
平面最近点对	19
点集排序-逆时针排序	21
凸包	22

数论

素性检测

```
1  #include <vector>
2  namespace PrimeTest {
3      long long mul(long long a, long long b, long long mod){
4          return (__int128) a * b % mod;
5      }
6
7      long long Pow(long long a, long long b, long long mod){
8          //mod <= 10^18.
9          long long res = 1;
10         while(b){
11             if (b&1) res = mul(res, a, mod);
12             b >>= 1;
13             a = mul(a, a, mod);
14         }
15         return res;
16     }
17
18     std::vector<long long> pr = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,
19     ↪ 37};
20
21     bool rabin_test(long long a, long long n, long long s, long long d){
22         long long u = Pow(a, d, n);
23         if (u == 1 or u == n - 1) return false;
24
25         for(long long i = 1; i < s; i++){
26             u = mul(u, u, n);
27             if (u == n - 1) return false;
28         }
29         return true;
30     }
31
32     bool rabin_miller(long long n){
33         if (n < 2) return false;
34         if (n % 2 == 0) return n==2;
35         long long res = 1;
36         long s = 0, d = n-1;
37         while(d%2==0) {
38             s++;
39             d>>=1;
40         }
```

```

41         for(long long i = 0; i < pr.size(); i++){
42             if (n%pr[i] == 0) {
43                 return n == pr[i];
44             }
45             if (rabin_test(pr[i], n, s, d)){
46                 return false;
47             }
48         }
49         return true;
50     }
51 }

```

大合数分解

该板子依赖了【数论】-【素性检测】的板子

```

1  #include <vector>
2  #include <algorithm>
3  namespace Factorize {
4      /*
5       * 可以分解不超过 LONG_LONG_MAX 的合数的所有因子，期望复杂度为
6       *  $\hookrightarrow O(n^{1/4})$ 
7       */
8      using namespace PrimeTest;
9
10     /*
11     * 牛逼plus的gcd
12     * 由于不用做除法和取模，只用了二进制命令和减法，所以速度非常快
13     * 这里不是朴素的更相减损术，而是用了一些优雅的性质
14     */
15     long long gcd(long long _a, long long _b) {
16         unsigned long long a = abs(_a), b = abs(_b);
17         if (a == 0) return b;
18         if (b == 0) return a;
19         int shift = __builtin_ctzll(a|b); // 拿到了最多有多少个2
20         a >>= __builtin_ctzll(a); // 这里保证了a是奇数
21         do {
22             b >>= __builtin_ctzll(b); // 这里保证了b是奇数
23             if (a > b) std::swap(a, b);
24             b -= a; // 这里两个奇数相减，一定是偶数，在下一个循环中至少二进
25             // 制会少一位
26         } while (b);
27         return (a << shift);
28     }
29 }

```

```

28  /*
29  * 返回n的任意一个素因子
30  * 用一个优雅的近似随机数来判断是否是素数
31  * 如果随机过程均匀，期望复杂度是  $O(n^{\{1/4\}} * \log(n))$ 
32  */
33  long long pollard_single(long long n) {
34      if (rabin_miller(n)) return n;
35      if (n%2==0) return 2;
36      long long st = 0;
37      auto f = [&](long long x) { return (__int128(x) * x + st) % n; };
38      while (true) {
39          st++;
40          long long x = st, y = f(x);
41          while (true) {
42              long long p = gcd((y - x + n), n);
43              if (p == 0 or p == n) break;
44              if (p != 1) return p;
45              x = f(x);
46              y = f(f(y));
47          }
48      }
49  }
50
51  /*
52  * 返回n的所有素因子
53  * n的因子越多，该算法越快
54  * 近似复杂度可以用  $O(n^{\{1/4\}})$  估计
55  */
56  std::vector<long long> pollard(long long n) {
57      if(n==1) return {};
58      auto x = pollard_single(n);
59      if (x==n) return {x};
60      auto l = pollard(x);
61      auto r = pollard(n/x);
62      l.insert(l.end(), r.begin(), r.end());
63      return l;
64  }
65  }

```

二次剩余-膜意义下开方

```

1  #include <vector>
2  namespace SqrtMod {
3      long long mod, i2;
4      struct Complex {

```

```

5     long long a, b;
6     Complex(long long _a, long long _b) : a(_a), b(_b) {}
7     Complex operator*(const Complex& other) {
8         return Complex((a*other.a + i2*b%mod*other.b%mod)%mod,
9             ↪ (a*other.b + b*other.a)%mod);
10    }
11    Complex operator%(long long _mod) {
12        return *this;
13    }
14};
15
16// long long的快速幂和Complex的放一起了
17template<typename T>
18void qpow(T a, long long b, long long mod, T& ans) {
19    while (b) {
20        if (b & 1) ans = ans * a % mod;
21        a = a * a % mod;
22        b >>= 1;
23    }
24}
25
26/*
27 * 膜意义下的开方（二次剩余）：找到  $x$  使得  $x^2 \equiv a \pmod{p}$ 
28 * 复杂度:  $O(\log p)$ 
29 * 限制:
30 *   -  $a < 2^{31}$ 
31 *   -  $p < 2^{31}$ 
32 *   -  $p$  是素数
33 * 返回:  $p$ 范围内的所有满足条件的 $x$ 
34 */
35std::vector<int> sqrt_mod(int a, int p) {
36    if(a<=1 || p == 2) return {a};
37    // 判断a是否是二次剩余
38    long long remind = 1;
39    if(qpow((long long)a, (p-1)/2, p, remind); remind != 1) {
40        return {-1};
41    }
42
43    long long b, d; //  $d = b*b - a$  是二次非剩余
44    while(true) {
45        b = rand() % p;
46        d = (b*b - a) % p;
47        if(d < 0) d += p;
48        remind = 1;
49        qpow(d, (p-1)/2, p, remind);
50        if(remind != 1) break;

```

```

50     }
51     mod = p;
52     i2 = d;
53
54     // 计算  $(a+i)^{(p+1)/2}$ 
55     Complex data_(b, 1);
56     Complex power(1, 0);
57     qpow(data_, (p+1)/2, p, power);
58
59     int ans = power.a % p;
60     return (ans > p-ans ? std::vector<int>{p-ans, ans} :
        ↪ std::vector<int>{ans, p-ans});
61 }
62 }
63

```

Nim积

```

1  #include <cstring>
2  namespace NimProd {
3      using ull = unsigned long long;
4      ull cache[257][257];
5      /*
6       * 计算nim积
7       * 限制:
8       *     -  $x, y$  在 unsigned long long 范围内
9       * 复杂度:  $O(\log^2(\max(x, y)))$ 
10     */
11     ull solve(ull x, ull y, int p = 32) {
12         if (x <= 1 || y <= 1) return x * y;
13         if (p < 8 && cache[x][y]) return cache[x][y]; // 小缓存优化
14
15         ull a = x >> p;
16         ull b = ((1ull << p) - 1) & x;
17         ull c = y >> p;
18         ull d = ((1ull << p) - 1) & y;
19
20         ull bd = solve(b, d, p >> 1);
21         ull ac = solve(solve(a, c, p >> 1), 1ull << p >> 1, p >> 1);
22
23         ull ans = ((solve(a ^ b, c ^ d, p >> 1) ^ bd) << p) ^ ac ^ bd;
24         if (p < 8) cache[x][y] = cache[y][x] = ans;
25         return ans;
26     }
27     void init() {

```

```
28         memset(cache, 0, sizeof(cache));
29         for(int i = 0; i < 256; i++) {
30             for(int j = 0; j < 256; j++) {
31                 solve(i, j);
32             }
33         }
34     }
35 }
36
```


树

树的直径

```
1  #include <vector>
2  #include <tuple>
3  namespace TreeDiameter {
4      /*
5       * 无向正权树的最大直径，限制：
6       * 1. 直径需要<= LONG_LONG_MAX
7       * 2. 单颗树，而非森林
8       */
9      using namespace std;
10     using Graph = vector<vector<pair<int, long long>>>; // 起点对应的
        ↳ 边(终点 & 权值)
11
12     /*
13     * Input: 树，起始点
14     * Output: 离起始点最大的距离，对应的点
15     * 复杂度: O(边数)
16     */
17     pair<long long, int> dfs(const vector<vector<pair<int, long long>>>
        ↳ &g, int cur, int par = -1) {
18         pair<long long, int> ret(0, cur);
19         for (auto e : g[cur]) {
20             if (e.first == par) continue;
21             auto cost = dfs(g, e.first, cur);
22             cost.first += e.second;
23             ret = max(ret, cost);
24         }
25         return ret;
26     }
27     /*
28     * Input: 树
29     * Output: 直径起点，直径终点，直径长度
30     */
31     tuple<int, int, long long> tree_diameter(const
        ↳ vector<vector<pair<int, long long>>> &g) {
32         auto u = dfs(g, 0, -1).second;
33         long long dist;
34         int v;
35         tie(dist, v) = dfs(g, u, -1);
36         return make_tuple(u, v, dist);
37     }
38 }
```

```

39  /*
40  * 会搜索出一条从 $cur$ 到 $goal$ 的路径，结果会放在 $path$ 里面
41  * Input: 树
42  * Output: 路径
43  * 复杂度:  $O(\text{边数})$ 
44  */
45  void path_restoration(const vector<vector<pair<int, long long>>> &g,
46  ↪ vector<int> &path, int cur, int par, int &goal) {
47      path.push_back(cur);
48      if (cur == goal) {
49          goal = -1;
50          return;
51      }
52      for (auto e : g[cur]) {
53          int nxt = e.first;
54          if (nxt == par) continue;
55          path_restoration(g, path, nxt, cur, goal);
56          if (goal == -1) return;
57      }
58
59      if (goal == -1) {
60          return;
61      }
62      path.pop_back();
63  }
64  }
65

```

最近公共祖先

```

1  #include <vector>
2
3  namespace LCA {
4  #define V vector
5      /*
6      * 最近公共祖先，限制：
7      * 1.  $root$ 必须为0 / 1
8      * 2. 总复杂度  $O(q \log(n) + n \log(n))$  //  $q$ 次查询，一共 $n$ 个节点，建树过
9      ↪ 程  $n \log(n)$ ，每次查询  $\log(n)$ 
10     */
11     using namespace std;
12     class Tree {
13     private:
14         int n_;

```

```

14     int root_;
15     int lg;
16     V<int> depth;
17     V<V<int>> father;
18     V<V<int>> son;
19     /*
20      * 从跟节点dfs, 来构建depth数组
21      * 复杂度: O(n)
22      */
23     void dfs(int now, int pre = -1, int dep = 1) {
24         depth[now] = dep;
25         father[0][now] = pre;
26         for(auto s : son[now]) {
27             if(s==pre) continue;
28             dfs(s, now, dep+1);
29         }
30     }
31     /*
32      * 构建祖先关系, 倍增构建
33      * 复杂度: O(n*log(n))
34      */
35     void build_father() {
36         for(int i = 1; i < lg; i++) {
37             for(int j = root_; j < n_ + (root_==1); j++) {
38                 father[i][j] = (father[i-1][j] == -1) ? -1 :
39                     ↪ father[i-1][father[i-1][j]];
40             }
41         }
42     public:
43         Tree(int root, int n) {
44             root_ = root;
45             n_ = n;
46             lg = 1;
47             while((1<<lg) < n) lg++;
48             depth.resize(n+(root==1));
49             father = V<V<int>>(lg, V<int>(n+(root==1), -1));
50             son = V<V<int>>(n+(root==1), V<int>(0));
51         }
52
53         /*
54          * 增加一个父子关系, now的父亲是pre
55          * 复杂度: O(1)
56          */
57         void add(int pre, int now) {
58             son[pre].push_back(now);

```

```

59         son[now].push_back(pre);
60     }
61
62     /*
63     * 完整建树，需要在add完所有父子关系才可以调用
64     * 复杂度:  $O(n \log(n))$ 
65     */
66     void build() {
67         this->dfs(root_);
68         this->build_father();
69     }
70
71     /*
72     * 查询u和v的最近公共祖先
73     * 复杂度:  $O(\log(n))$ 
74     */
75     int query_lca(int u, int v) {
76         if(depth[u] > depth[v]) {
77             swap(u, v);
78         }
79         int depth_diff = depth[v] - depth[u];
80         for(int i = 0; i < lg; i++) {
81             if(depth_diff & (1<<i)) {
82                 v = father[i][v];
83             }
84         }
85         if(u == v) {
86             return u;
87         }
88         for(int i = lg - 1; i >= 0; i--) {
89             if(father[i][u] != father[i][v]) {
90                 u = father[i][u];
91                 v = father[i][v];
92             }
93         }
94         return father[0][u];
95     }
96 };
97 }

```

树同构

```

1  #include <vector>
2  #include <map>
3  #include <algorithm>

```

```

4 namespace TreeIsomorphism {
5     /*
6      * 判断有根树是否同构，限制：
7      * 1. root只能为0或者1
8      * 2. 复杂度  $O(n \cdot \log(n))$ 
9      */
10    using namespace std;
11    class Tree {
12    private:
13        int n_, root_;
14        vector<vector<int>> son;
15        map<vector<int>, int> mp;
16        vector<int> t;
17        /*
18         * 建树过程
19         * 每次会从root开始遍历，记录每个子节点的编号，并从map中判断是否已
20         *   ↳ 经存在同构结构
21         * 为了保证同构，所以需要有一个sort来保证子节点编号不影响结果
22         */
23        void dfs(int root, int pre) {
24            vector<int> v(1, 0);
25            for(auto i : son[root]) {
26                if(i==pre) continue;
27                dfs(i, root);
28                v.push_back(t[i]);
29            }
30            sort(v.begin(), v.end());
31            if(mp.find(v) == mp.end()) {
32                mp[v] = mp.size();
33            }
34            t[root] = mp[v];
35        }
36    public:
37        Tree(int n, int root) {
38            n_ = n;
39            root_ = root;
40            t = vector<int>(n+(root==1), 0);
41            son.resize(n+(root==1));
42        }
43
44        /*
45         * 添加边
46         * 这里只要是无向边就行了，不需要保证u是v的父亲
47         */
48        void add_edge(int u, int v) {

```

```

49         son[u].push_back(v);
50         son[v].push_back(u);
51     }
52
53     /*
54     * 构建树
55     * 这里会计算出每个节点的结构，并给一个编号，保证编号相同的节点对应
56     *   ↳ 的子树是同构的
57     */
58     void build() {
59         dfs(root_, -1);
60     }
61
62     /*
63     * 返回有多少个不同构的子树
64     */
65     int get_size() {
66         return mp.size();
67     }
68
69     /*
70     * 返回每个子树的编号
71     * Input: 每个节点的编号
72     * Output: 每个子树的结构编号
73     * 仅保证相同结构的子树返回的编号一致，不保证节点更多的子树编号更大
74     */
75     int get_mask(int i) {
76         return t[i];
77     }
78 };

```

图论

单源最短路

```
1  #include <vector>
2  #include <set>
3  #include <algorithm>
4  namespace ShortestPath {
5      /*
6       * 单源最短路
7       * 限制:
8       *   - 复杂度  $O(N+M)$ 
9       *   - 无负权
10      *   - 最长路径不超过  $1e18$ 
11      *   - 有向图 / 无向图都可以
12      *   - 点的标号从0开始
13      */
14      class Graph {
15      private:
16          int N, M; // n point, m edge
17          std::vector<std::vector<std::pair<long long, int>>>> sons; //
18          ↪ first = value, second = to
19          std::vector<long long> distance; // 记录每个点的最近距离
20          std::set<std::pair<long long, int>> q; // 一个临时的 queue
21          std::vector<int> per; // 记录从s开始到当前位置的最短路径中, 当前节
22          ↪ 点的前一个节点是什么
23
24      public:
25          Graph(int n, int m) : N(n), M(m) {
26              sons = std::vector<std::vector<std::pair<long long,
27              ↪ int>>>>(N);
28          }
29
30          void add_edge(int s, int t, long long v) {
31              sons[s].push_back({v, t});
32          }
33
34          /*
35           * 单源最短路, 复杂度  $O(N+M)$ 
36           * 会从s开始建图, 构建每个从s可达的点的距离 & 路径
37           */
38          void build(int s) {
39              per = std::vector<int>(N, -1);
40              distance = std::vector<long long>(N, (long long) 1e18);
41              q.clear();
```

```

39
40     distance[s] = 0;
41     q.insert({0, s});
42     while(!q.empty()) {
43         auto [v, p] = *q.begin();
44         q.erase(q.begin());
45         if(distance[p] < v) continue;
46
47         for(auto [v2, p2] : sons[p]) {
48             if(distance[p2] > distance[p] + v2) {
49                 q.erase({distance[p2], p2});
50                 per[p2] = p;
51                 distance[p2] = distance[p] + v2;
52                 q.insert({distance[p2], p2});
53             }
54         }
55     }
56 }
57
58 /*
59  * build完成后才可以调用，并且二者s需要相同，纯找路径
60  * 复杂度O(path_length)
61  */
62 std::pair<long long, std::vector<int>>> shortest_path(int s, int
↪ t) {
63     std::pair<long long, std::vector<int>>> ret;
64     ret.first = distance[t];
65     if(ret.first == 1e18) return ret; // 不存在最短路
66     ret.second.push_back(t);
67     while(ret.second[ret.second.size()-1] != s) {
68
69         ↪ ret.second.push_back(per[ret.second[ret.second.size()-1]]);
70     }
71     std::reverse(ret.second.begin(), ret.second.end());
72     return ret;
73 }
74 }

```

有向图-强连通分量

```

1  #include <cassert>
2  #include <vector>
3  namespace SCC {
4      /*

```



```

5      * 找出有向图的强连通分量
6      * 限制:
7      *     - 点的标号从0开始
8      *     - 复杂度  $O(N+M)$ 
9      */
10     class Graph{
11     private:
12         int N;
13         std::vector<std::vector<int>> g; // 有向图
14         std::vector<std::vector<int>> g2; // 反向图
15         std::vector<int> s; // 第一次dfs的时候记录dfs序, 然后反向dfs
16         std::vector<bool> vis;
17
18         void dfs(int u) {
19             vis[u] = true;
20             for(auto v : g[u]) {
21                 if(!vis[v]) dfs(v);
22             }
23             s.push_back(u);
24         }
25         void dfs2(int a, int c) {
26             vis[a] = true;
27             sccs[c].push_back(a);
28             for(auto v : g2[a]) {
29                 if(!vis[v]) dfs2(v, c);
30             }
31         }
32     public:
33         std::vector<std::vector<int>> sccs; // 保存了有哪些强连通分量, 每
        ↳ 一个list是一个强连通分量
34
35         Graph(int n) : N(n) {
36             g.resize(n);
37             g2.resize(n);
38             vis = std::vector<bool>(n, false);
39             s = std::vector<int>(0);
40             sccs = std::vector<std::vector<int>>(0);
41         }
42         void add_edge(int s, int t) {
43             g[s].push_back(t);
44             g2[t].push_back(s);
45         }
46         void build() {
47             for(int i = 0; i < N; i++) {
48                 if(!vis[i]) dfs(i);
49             }

```

```

50         vis = std::vector<bool>(N, false);
51         for(int i = s.size()-1; i>=0; i--) {
52             if(!vis[s[i]]) {
53                 sccs.push_back(std::vector<int>());
54                 dfs2(s[i], sccs.size()-1);
55             }
56         }
57     };
58 };
59 }

```

计算几何

vector重排

```
1  #include <vector>
2  #include <algorithm> // sort函数需要
3  #include <numeric> // iota函数调用, 你可以可以去掉这个, 然后手动赋值 (但是
    ↳ 不够优雅)
4  namespace Rerange {
5      /*
6       * 任意类型vector排序
7       * 需要重载 <, ==, =
8       * 下标从0开始
9       * 按照升序排列
10     */
11     template <typename T>
12     std::vector<int> argsort(const std::vector<T> &A) { // 返回重排下标,
        ↳ 表示当前这个位置应该用A[ret[i]]填充后就是有序的了
13         std::vector<int> ids(A.size());
14         std::iota(ids.begin(), ids.end(), 0);
15         sort(ids.begin(), ids.end(), [&](int i, int j) { return (A[i] ==
            ↳ A[j] ? i < j : A[i] < A[j]); });
16         return ids;
17     }
18
19     template <typename T>
20     std::vector<T> rearrange(const std::vector<T> &before, const
        ↳ std::vector<int> &index) { // 利用重排下标进行排序, 返回排序后的数
        ↳ 组
21         std::vector<T> after(before.size());
22         for(int i = 0; i < before.size(); i++) {
23             after[i] = before[index[i]];
24         }
25         return after;
26     }
27 }
```

平面最近点对

```
1  #include <vector>
2  namespace ClosestPair {
3      /*
4       * 平面最近点对
5       * 复杂度:  $O(n \log n)$ 
```

```

6      * 限制:
7      *      - 坐标需要是整数, 范围 INT_MIN ~ INT_MAX
8      */
9      struct Point {
10         long long x, y;
11         Point() : x(0), y(0) {}
12         Point(long long x, long long y) : x(x), y(y) {}
13         Point(std::pair<long long, long long> p) : x(p.first),
14             ↪ y(p.second) {}
15         Point operator+(Point p) const { return {x + p.x, y + p.y}; }
16         Point operator-(Point p) const { return {x - p.x, y - p.y}; }
17         bool operator==(Point p) const { return x == p.x && y == p.y; }
18         bool operator!=(Point p) const { return x != p.x || y != p.y; }
19         Point operator-() const { return {-x, -y}; }
20         Point operator*(long long t) const { return {x * t, y * t}; }
21         Point operator/(long long t) const { return {x / t, y / t}; }
22         bool operator<(Point p) const {
23             if (x != p.x) return x < p.x;
24             return y < p.y;
25         }
26         long long dot(const Point& other) { return x * other.x + y *
27             ↪ other.y; }
28         long long det(const Point& other) { return x * other.y - y *
29             ↪ other.x; }
30     };
31
32     std::pair<int, int> closest_pair_dc(std::vector<Point> point) {
33         int N = point.size();
34
35         auto I = Rerange::argsort(point); // 按照x,y排序
36         point = Rerange::rearrange(point, I); // 重排后的point列表
37
38         long long best = -1; // 记录最佳距离
39         std::pair<int, int> best_pair = {-1, -1}; // 记录最近点对
40         auto upd = [&](int i, int j) -> void { // 无脑更新函数
41             Point p = point[i] - point[j];
42             long long d = p.dot(p);
43             if (best == -1 || best > d) { best = d, best_pair = {I[i],
44                 ↪ I[j]}; }
45         };
46
47         upd(0, 1); // 用(0,1)这两个点来初始化答案
48
49         auto dfs = [&](auto &dfs, int L, int R) -> std::vector<int> { //
50             ↪ 分治

```

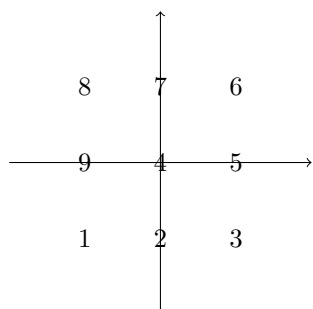
```

47         if (R == L + 1) return {L};
48         int M = (L + R) / 2;
49         auto I0 = dfs(dfs, L, M);
50         auto I1 = dfs(dfs, M, R);
51         std::vector<int> I;
52         std::vector<int> near;
53         int a = 0, b = 0;
54         for (int i = 0; i < R-L; i++) {
55             auto idx = [&]() -> int {
56                 if (a == I0.size()) return I1[b++];
57                 if (b == I1.size()) return I0[a++];
58                 int i = I0[a], j = I1[b];
59                 if (point[i].y < point[j].y) {
60                     ++a;
61                     return i;
62                 }
63                 ++b;
64                 return j;
65             }();
66             I.emplace_back(idx);
67             auto dx = point[M].x - point[idx].x;
68             if (dx * dx > best) { continue; }
69             for (int k = near.size()-1; k >= 0; k--) {
70                 int j = near[k];
71                 long long dy = point[idx].y - point[j].y;
72                 if (best == 0 || dy * dy > best) break;
73                 upd(idx, j);
74             }
75             near.emplace_back(idx);
76         }
77         return I;
78     };
79
80     dfs(dfs, 0, N);
81     return best_pair;
82 }
83 }
84

```

点集排序-逆时针排序

排序顺序如下图:



注意，这里的(0,0)被特判了，默认原点在x轴正半轴上。此外，如果两点斜率相同的情况下，排序结果会任意

```

1 namespace OrderedPoint {
2     /*
3      * 可以对点进行排序，逆时针排序
4      * 注意：可以支持任意点集的排序，包括  $x = 0, y = 0$  的 case
5      * 因为这里用了乘法，所以  $x/y$  坐标不要超过  $INT\_MAX$ ，否则你需要转
        ↳ 成  $\_\_int128$  来算乘法
6      */
7     class Point {
8     public:
9         long long x, y;
10        int sign() const {
11            if (y < 0) return -1;
12            if (y == 0 && x >= 0) return 0;
13            return 1;
14        }
15        bool operator<(const Point& other) const {
16            if (sign() != other.sign()) return sign() < other.sign();
17            return x * other.y - y * other.x > 0;
18        }
19    };
20 };

```

凸包

```

1 #include <vector>
2 #include <numeric>
3 #include <algorithm>
4 namespace ConvexHull {
5     /*
6      * 计算凸包
7      * 限制:
8      *     - 点的范围必须要  $INT\_MIN \sim INT\_MAX$  范围内

```

```

9      *      - 复杂度:  $O(n \log n)$  // 因为有一次排序逻辑
10     *      - 可以有重复点, 可以有斜率为0的点
11     */
12     std::vector<int> argsort(const std::vector<std::pair<long long, long
    ↪ long long>> &points) {
13         std::vector<int> ids(points.size());
14         std::iota(ids.begin(), ids.end(), 0);
15         std::sort(ids.begin(), ids.end(), [&](int i, int j) { return
    ↪ (points[i] == points[j] ? i < j : points[i] < points[j]); });
16         return ids;
17     }
18     /*
19     * 凸包实际的逻辑
20     * 输入:
21     *      - points: 点集, 下标从0开始
22     *      - inclusize: 是否包含重复点 / 是否包含多点共边的情况下的中间点
23     * 输出:
24     *      - 表示哪些点是凸包上的点的下标
25     */
26     std::vector<int> ConvexHull(const std::vector<std::pair<long long,
    ↪ long long>> &points, bool inclusize=false) {
27         long long N = points.size();
28         if(N==0) return {};
29         if(N==1) return {0};
30         if(N==2) {
31             if(points[0]<points[1]) return {0, 1};
32             if(points[1]<points[0]) return {1, 0};
33             if(inclusize) return {0, 1};
34             return {0};
35         }
36         auto idx = argsort(points);
37
38         auto check = [&](long long i, long long j, long long k) -> bool {
39             auto xi = points[i].first, yi = points[i].second;
40             auto xj = points[j].first, yj = points[j].second;
41             auto xk = points[k].first, yk = points[k].second;
42             auto dx1 = xj - xi, dy1 = yj - yi;
43             auto dx2 = xk - xj, dy2 = yk - yj;
44             auto det = dx1 * dy2 - dy1 * dx2;
45             return (inclusize ? det >= 0 : det > 0);
46         };
47
48         auto calc = [&]() {
49             std::vector<int> P;
50             for(auto &k: idx) {
51                 if(P.size() && points[P.back()] == points[k]) continue;

```

```

52         while(P.size()>1) {
53             auto i = P[P.size()-2];
54             auto j = P[P.size()-1];
55             if(check(i, j, k)) break;
56             P.pop_back();
57         }
58         P.emplace_back(k);
59     }
60     return P;
61 };
62
63 auto P = calc();
64 if(!P.empty()) P.pop_back();
65 std::reverse(idx.begin(), idx.end());
66 auto Q = calc();
67 P.insert(P.end(), Q.begin(), Q.end());
68 if(P.size()>=2 && points[P[0]] == points[P.back()]) P.pop_back();
69 return P;
70 }
71 }

```