#include <iostream> // C++ I/O   
#include <fstream> // File I/O   
#include <sstream> // String stream I/O   
#include <iomanip> // C++ I/O manipulator   
   
#include <cstdlib> // C library   
#include <cstdio> // C I/O   
#include <ctime> // C time   
#include <cmath> // Math library   
#include <cstring> // C strings   
   
#include <vector> // Vector   
#include <queue> // Queue   
#include <stack> // Stack   
#include <map> // Map   
#include <set> // Set   
#include <algorithm> // Algorithms   
   
using namespace std;   
   
#define reps(\_var,\_begin,\_end,\_step) for (int \_var = (\_begin); \   
 \_var <= (\_end); \_var += (\_step))   
#define reps\_(\_var,\_end,\_begin,\_step) for (int \_var = (\_end); \   
 \_var >= (\_begin); \_var -= (\_step))   
#define rep(\_var,\_begin,\_end) reps(\_var, \_begin, \_end, 1)   
#define rep\_(\_var,\_end,\_begin) reps\_(\_var, \_end, \_begin, 1)   
#define minimize(\_var,\_targ) \_var = min(\_var, \_targ)   
#define maximize(\_var,\_targ) \_var = max(\_var, \_targ)   
   
typedef unsigned long long ull;   
typedef long long lli, ll;   
typedef double llf;   
   
template <typename typ>   
void memclr(typ p) {   
 memset(p, 0, sizeof(p)); }   
template <typename typ>   
void memclr(typ arr[], int n) {   
 memset(arr, 0, sizeof(arr[0]) \* (n + 1)); }   
template <typename typ, int dim>   
void memclr(typ arr[][dim], int n, int m) {   
 rep(i, 0, n) memset(arr[i], 0, sizeof(arr[i][0]) \* (m + 1)); }   
   
lli read(void)   
{   
 lli res = 0, sgn = 1;   
 char ch = getchar();   
 while(ch < '0' || ch > '9')   
 sgn = ch == '-' ? -1 : 1, ch = getchar();   
 while(ch >= '0' && ch <= '9')   
 res = res \* 10 + ch - '0', ch = getchar();   
 return res \* sgn;   
}   
   
const int maxn = 1010;   
   
int main(int argc, char\*\* argv)   
{   
 return 0;   
}   
   
   
// fm\_begin(maths, euclid\_gcd)   
// fm\_begin(maths, chinese\_remainder\_theorem)   
// fm\_begin(maths, fast\_exponentiation)   
// fm\_begin(maths, prime\_filter)   
// fm\_begin(maths, fermats\_little\_theorem)   
// fm\_begin(maths, miller\_rabin)   
// fm\_begin(strings, trie)   
// fm\_begin(strings, knuth\_morris\_pratt)   
// fm\_begin(strings, aho\_corasick\_automaton)   
// fm\_begin(strings, suffix\_array)   
// fm\_begin(strings, suffix\_automaton)   
// fm\_begin(strings, manacher)   
// fm\_begin(trees, disjoint\_set)   
// fm\_begin(trees, segment\_tree)   
// fm\_begin(trees, splay)   
// fm\_begin(trees, kd\_tree)   
// fm\_begin(graphs, basic\_graph)   
// fm\_begin(graphs, dijkstra)   
// fm\_begin(graphs, floyd\_warshall)   
// fm\_begin(graphs, tree\_diameter)   
// fm\_begin(graphs, tree\_center)   
// fm\_begin(graphs, heavy\_light\_decomposition)   
// fm\_begin(graphs, link\_cut\_tree)   
// fm\_begin(graphs, prim)   
// fm\_begin(graphs, kruskal)   
// fm\_begin(graphs, scc\_tarjan)   
// fm\_begin(graphs, dcc\_tarjan\_v)   
// fm\_begin(graphs, dcc\_tarjan\_e)   
// fm\_begin(graphs, 2\_sat)   
// fm\_begin(graphs, dinic)   
// fm\_begin(graphs, spfa\_costflow)   
// fm\_begin(graphs, zkw\_costflow)   
// fm\_begin(graphs, hungary\_match)   
// fm\_begin(graphs, bron\_kerbosch)   
// @chapter 1.1 Point 定义   
// @chapter 1.2 Line 定义   
// @chapter 1.3 两点间距离   
// @chapter 1.4 判断 线段相交   
// @chapter 1.5 判断 直线和线段相交   
// @chapter 1.6 点到直线距离   
// @chapter 1.7 点到线段距离   
// @chapter 1.8 计算多边形面积   
// @chapter 1.9 判断点在线段上   
// @chapter 1.10 判断点在凸多边形内   
// @chapter 1.11 判断点在任意多边形内   
// @chapter 1.12 判断多边形   
// @chapter 1.13 简单极角排序   
// @chapter 2.1 凸包   
// @chapter 3.1 平面最近点对   
// @chapter 4.1 旋转卡壳 / 平面最远点对   
// @chapter 4.2 旋转卡壳计算平面点集最大三角形面积   
// @chapter 4.3 求解两凸包最小距离   
// @chapter 5.1 半平面交   
// @chapter 6.1 三点求圆心坐标   
// @chapter 7.1 求两圆相交的面积   
   
   
fm\_begin(maths, euclid\_gcd):   
 // @desc Euclidean greatest common divisor algorithm.   
 // @complexity Time: O(Log[n]), Space: O(n)   
 // @usage gcd(a, b): calculate (a, b)   
 // @usage lcm(a, b): calculate [a, b]   
 // @usage extended\_gcd(a, b, x, y): solve equation ax + by = gcd(a, b)   
 // and store results in x, y (such x, y always exists)   
 lli gcd(lli a, lli b)   
 {   
 if (b == 0)   
 return a;   
 return gcd(b, a % b);   
 }   
 lli lcm(lli a, lli b)   
 {   
 return a / gcd(a, b) \* b;   
 }   
 lli extended\_gcd(lli a, lli b, lli& x, lli& y)   
 {   
 if (b == 0) {   
 x = 1, y = 0;   
 return a;   
 }   
 int q = extended\_gcd(b, a % b, y, x);   
 y -= lli(a / b) \* x;   
 return q;   
 }   
fm\_end(maths, euclid\_gcd);   
   
   
fm\_begin(maths, chinese\_remainder\_theorem):   
 // @desc Chinese remainder theorem   
 // @complexity Time: O(n Log[n]), Space: O(n)   
 // @usage solve(a[], m[], n): solve a series of equation, s.t.   
 // x \equiv a\_i (mod m\_i) \forall i = 1..n   
 // m\_i has to be coprime with each other   
 // @usage extended\_solve(a[], m[], n): solve a series of equation, s.t.   
 // x \equiv a\_i (mod m\_i) \forall i = 1..n   
 // m\_i doesn't have to be coprime   
 // returns -1 if no solutions available   
 fm(maths, euclid\_gcd) egcd;   
 lli solve(lli a[], lli m[], int n)   
 {   
 lli res = 0, lcm = 1, t, tg, x, y;   
 rep(i, 1, n)   
 lcm \*= m[i];   
 rep(i, 1, n) {   
 t = lcm / m[i];   
 egcd.extended\_gcd(t, m[i], x, y);   
 x = ((x % m[i]) + m[i]) % m[i];   
 res = (res + t \* x \* a[i]) % lcm;   
 }   
 return (res + lcm) % lcm;   
 }   
 lli extended\_solve(lli a[], lli m[], int n)   
 {   
 lli cm = m[1], res = a[1], x, y;   
 rep(i, 2, n) {   
 lli A = cm, B = m[i], C = (a[i] - res % B + B) % B,   
 gcd = egcd.extended\_gcd(A, B, x, y),   
 Bg = B / gcd;   
 if (C % gcd != 0)   
 return -1;   
 x = (x \* (C / gcd)) % Bg;   
 res += x \* cm;   
 cm \*= Bg;   
 res = (res % cm + cm) % cm;   
 }   
 return (res % cm + cm) % cm;   
 }   
fm\_end(maths, chinese\_remainder\_theorem);   
   
   
fm\_begin(maths, fast\_exponentiation):   
 // @desc Sped up exponential calculation   
 // @complexity Time: O(Log[n]), Space: O(n)   
 // @usage pow(a, k, m): calculate a^k % m   
 // @usage pow(a, k): calculate a^k (potential overflow)   
 lli pow(lli a, lli k, lli m = 0)   
 {   
 lli res = 1, tmp = a;   
 while (k > 0) {   
 if ((k & 1) == 1) {   
 res \*= tmp;   
 if (m > 0)   
 res %= m;   
 }   
 k >>= 1;   
 tmp \*= tmp;   
 if (m > 0)   
 tmp %= m;   
 }   
 return res;   
 }   
fm\_end(maths, fast\_exponentiation);   
   
   
fm\_begin(maths, prime\_filter):   
 // @desc Filter prime numbers   
 // @complexity Time: O(n), Space: O(n)   
 // @usage isprime[i]: true if i is a prime number, elsewise false   
 // @usage primes[0]: number of prime numbers   
 // @usage primes[i]: The i-th prime number   
 fm\_const(int, maxn, 100000000);   
 bool isprime[maxn];   
 int primes[maxn];   
 void filter(void)   
 {   
 isprime[1] = false;   
 rep(i, 2, maxn - 1)   
 isprime[i] = true;   
 primes[0] = 0;   
 rep(i, 2, maxn - 1) {   
 if (!isprime[i])   
 continue;   
 reps(j, i, maxn - 1, i)   
 isprime[j] = false;   
 primes[++primes[0]] = i;   
 }   
 return ;   
 }   
fm\_end(maths, prime\_filter);   
   
   
fm\_begin(maths, fermats\_little\_theorem):   
 // @desc Fermat's little theorem   
 // if IsPrime[p] and Gcd[a, p] == 1:   
 // a ^ (p - 1) === 1 (mod p)   
 // (a ^ (p - 1)) % p = 1   
 // @complexity Time: O(Log[n]), Space: O(1)   
 // @usage calc(a, p, k): calculate a^p % k   
 lli calc(lli a, lli p, lli k)   
 {   
 // Asserted: IsPrime[p] and Gcd[a, p] = 1   
 if (p < k - 1)   
 return lli(pow(a, p)) % k;   
 return calc(a, p % (k - 1), k);   
 }   
fm\_end(maths, fermats\_little\_theorem);   
   
   
fm\_begin(maths, miller\_rabin):   
 // @desc Miller-Rabin prime testing   
 // relies on Fermat's little theorem   
 // @complexity Time: O(k Log[n]^2), Space: O(1)   
 // @usage test(n, k): test n under modulo k   
 // @usage is\_prime(n): true if n is prime, elsewise false   
 fm(maths, fast\_exponentiation) fexp;   
 bool test(lli n, lli k)   
 {   
 if (fexp.pow(k, n - 1, n) != 1)   
 return false;   
 lli t = n - 1, tmp;   
 while (t % 2 == 0) {   
 t >>= 1;   
 tmp = fexp.pow(k, t, n);   
 if (tmp != 1 && tmp != n - 1)   
 return false;   
 if (tmp == n - 1)   
 return true;   
 }   
 return true;   
 }   
 bool is\_prime(lli n)   
 {   
 if (n == 1 || (n > 2 && n % 2 == 0))   
 return false;   
 lli samples[14] = {4,   
 2, 3, 5, 7, // n < 3.2e9   
 11, 13, 17, 19, 23, 29, 31, 37, // n < 1.8e19   
 41, // n < 3.3e25   
 };   
 rep(i, 1, samples[0]) {   
 if (n == samples[i])   
 return true;   
 if (n > samples[i] && !test(n, samples[i]))   
 return false;   
 }   
 return true; // Certain prime   
 }   
fm\_end(maths, miller\_rabin);   
   
   
fm\_begin(strings, trie):   
 // @desc Trie tree, string indexer   
 // @complexity Time: O(n), Space: O(Sum[n])   
 // @usage \_\_makenode(): creates empty node   
 // @usage \_\_insert(p, str, level): inserts str into tree, recursively   
 // returns the node marking termination of this string   
 // @usage \_\_remove(p, str, level): removes str from tree, recursively   
 // returns true if such string exists and is removed   
 // @usage \_\_query(p, str, level): returns the node marking termination of   
 // string str   
 // @usage init(): initializes tree   
 // @usage insert(str, data): inserts str into tree, if the string already   
 // exists, original data will be replaced by new one instead   
 // @usage remove(str): removes str from tree, returns true if and only if   
 // the string existed and is removed   
 // @usage query(str, data): query if str existed and its data, returns   
 // true if string existed, and its key is stored in data.   
 fm\_const(int, max\_nodes, 1001000);   
 fm\_const(int, charset\_size, 256);   
 struct node   
 {   
 int flag, children;   
 node \*child[charset\_size];   
 void \*data;   
 };   
 node npool[max\_nodes], \*root;   
 int npcnt;   
 node\* \_\_make\_node(void)   
 {   
 node \*p = &npool[++npcnt];   
 p->flag = false;   
 p->children = 0;   
 memclr(p->child);   
 p->data = nullptr;   
 return p;   
 }   
 node\* \_\_insert(node\* p, const string& str, int level)   
 {   
 if (level == str.length()) {   
 if (!p->flag)   
 p->children += 1;   
 p->flag = true;   
 return p;   
 }   
 char ch = str[level];   
 if (!p->child[ch])   
 p->child[ch] = \_\_make\_node();   
 p->children -= p->child[ch]->children;   
 node \*q = \_\_insert(p->child[ch], str, level + 1);   
 p->children += p->child[ch]->children;   
 return q;   
 }   
 bool \_\_remove(node\* p, const string& str, int level)   
 {   
 if (level == str.length()) {   
 if (p->flag) {   
 p->children -= 1;   
 p->flag = false;   
 return true;   
 }   
 return false;   
 }   
 char ch = str[level];   
 if (p->child[ch] == nullptr)   
 return false;   
 p->children -= p->child[ch]->children;   
 bool res = \_\_remove(p->child[ch], str, level + 1);   
 p->children += p->child[ch]->children;   
 if (p->child[ch]->children == 0)   
 p->child[ch] = nullptr;   
 return res;   
 }   
 node\* \_\_query(node\* p, const string& str, int level)   
 {   
 if (level == str.length())   
 return p;   
 char ch = str[level];   
 if (!p->child[ch])   
 return nullptr;   
 return \_\_query(p->child[ch], str, level + 1);   
 }   
 void init(void)   
 {   
 npcnt = 0;   
 root = \_\_make\_node();   
 return ;   
 }   
 bool insert(const string& str, void\* data = nullptr)   
 {   
 node \*p = \_\_insert(root, str, 0);   
 if (!p)   
 return false;   
 p->data = data;   
 return true;   
 }   
 bool remove(const string& str)   
 {   
 return \_\_remove(root, str, 0);   
 }   
 bool query(const string& str, void\*& data)   
 {   
 node \*p = \_\_query(root, str, 0);   
 if (p == nullptr || !p->flag)   
 return false;   
 data = p->data;   
 return true;   
 }   
 bool query(const string& str)   
 {   
 void \*data;   
 return query(str, data);   
 }   
fm\_end(strings, trie);   
   
   
fm\_begin(strings, knuth\_morris\_pratt):   
 // @desc Knuth-Morris-Pratt string matching algorithm   
 // @complexity Time: O(n+m), Space: O(m)   
 // @usage src[], n: base string and its length, indices starts from 0   
 // @usage pat[], m: pattern and length, indices starts from 0   
 // @usage get\_next(): retrieve next[] array for pattern   
 // @usage match(begin): match next occurence starting from begin   
 fm\_const(int, max\_len, 100100);   
 int n, m, src[max\_len], pat[max\_len], next[max\_len];   
 void get\_next(void)   
 {   
 next[0] = -1;   
 rep(i, 1, m - 1)   
 for (int j = next[i - 1]; ; j = next[j]) {   
 if (pat[j + 1] == pat[i]) {   
 next[i] = j + 1;   
 break;   
 } else if (j == -1) {   
 next[i] = -1;   
 break;   
 }   
 }   
 return ;   
 }   
 int match(int begin = 0)   
 {   
 int i = begin, j = 0;   
 for (; i < n && j < m; ) {   
 if (src[i] == pat[j]) {   
 i += 1;   
 j += 1;   
 } else if (j == 0) {   
 i += 1;   
 } else {   
 j = next[j - 1] + 1;   
 }   
 }   
 if (j == m)   
 return i - m;   
 return -1;   
 }   
fm\_end(strings, knuth\_morris\_pratt);   
   
   
fm\_begin(strings, aho\_corasick\_automaton):   
 // @desc Aho-Corasick automaton, match a series of patterns in a string   
 // @complexity Time: O(n+Sum[m]), Space: O(Sum[m])   
 // @usage match\_res: match result: [(position in string, pattern id)]   
 // @usage \_\_make\_node(): create new empty node   
 // @usage \_\_insert(p, str, str\_id, level): insert string, recursively   
 // @usage init(): initialize empty tree   
 // @usage insert(str, str\_id): insert str into tree, marking its id as   
 // str\_id. Only the first of same strings would appear in the tree   
 // @usage build\_tree(): construct fail pointers, no further inserts should   
 // appear after build\_tree, and no matches shall precede this   
 // @usage match(str): find all occurences of strings in tree in string   
 // and store the result in a match\_res object   
 fm\_const(int, max\_nodes, 1001000);   
 fm\_const(int, charset\_size, 256);   
 typedef vector<pair<int, int>> match\_res;   
 struct node   
 {   
 int val, flag, flag\_len, children;   
 node \*child[charset\_size], \*parent, \*fail;   
 node \*first\_child, \*next; // Adjacency list   
 };   
 node npool[max\_nodes], \*root;   
 int npcnt;   
 node\* \_\_make\_node(void)   
 {   
 node \*p = &npool[++npcnt];   
 p->val = p->flag = p->flag\_len = p->children = 0;   
 memclr(p->child);   
 p->parent = p->fail = nullptr;   
 p->first\_child = p->next = nullptr;   
 return p;   
 }   
 node\* \_\_insert(node\* p, const string& str, int str\_id, int level)   
 {   
 if (level == str.length()) {   
 if (p->flag > 0)   
 return nullptr;   
 p->flag = str\_id;   
 p->flag\_len = str.length();   
 return p;   
 }   
 char ch = str[level];   
 if (p->child[ch] == nullptr) {   
 node \*q = p->child[ch] = \_\_make\_node();   
 q->val = ch;   
 q->parent = p;   
 q->next = p->first\_child;   
 p->first\_child = q;   
 }   
 return \_\_insert(p->child[ch], str, str\_id, level + 1);   
 }   
 void init(void)   
 {   
 npcnt = 0;   
 root = \_\_make\_node();   
 root->fail = root;   
 return ;   
 }   
 void insert(const string& str, int str\_id)   
 {   
 \_\_insert(root, str, str\_id, 0);   
 return ;   
 }   
 void build\_tree(void)   
 {   
 queue<node\*> que;   
 root->fail = root;   
 for (node \*np = root->first\_child; np; np = np->next) {   
 np->fail = root;   
 for (node \*mp = np->first\_child; mp; mp = mp->next)   
 que.push(mp);   
 }   
 while (!que.empty()) {   
 node \*p = que.front();   
 que.pop();   
 p->fail = p->parent->fail->child[p->val];   
 if (p->fail == nullptr)   
 p->fail = root;   
 for (node \*np = p->first\_child; np; np = np->next)   
 que.push(np);   
 }   
 return ;   
 }   
 match\_res match(const string& str)   
 {   
 match\_res res;   
 int pos = 0;   
 node \*p = root;   
 while (pos <= str.length()) {   
 char ch = str[pos];   
 if (p->flag > 0)   
 res.push\_back(make\_pair(pos - p->flag\_len, p->flag));   
 if (pos == str.length())   
 break;   
 while (p->child[ch] == nullptr && p != root)   
 p = p->fail;   
 if (p->child[ch] != nullptr)   
 p = p->child[ch];   
 pos += 1;   
 }   
 return res;   
 }   
fm\_end(strings, aho\_corasick\_automaton);   
   
   
fm\_begin(strings, suffix\_array):   
 // @desc Suffix array   
 // @warning incompatible code style   
 // 喜欢钻研问题的 JS 同学, 最近又迷上了对加密方法的思考. 一天, 他突然想出了   
 // 一种他认为是终极的加密办法: 把需要加密的信息排成一圈, 显然, 它们有很多种   
 // 不同的读法.   
 // JSOI07 SOI07J OI07JS I07JSO 07JSOI 7JSOI0 把它们按照字符串的大小排序:   
 // 07JSOI 7JSOI0 I07JSO JSOI07OI07JS SOI07J 读出最后一列字符: I0O7SJ,   
 // 就是加密后的字符串 (其实这个加密手段实在很容易破解, 鉴于这是突然想出来   
 // 的, 那就^^). 但是, 如果想加密的字符串实在太长,...   
 fm\_const(int, maxn, 800100);   
 // This suffix array is only used to be sorted.   
 class SuffixArray   
 {   
 public:   
 int n, pwn, rank[maxn], prank[maxn];   
 int stra[maxn], strb[maxn], srta[maxn], srtb[maxn];   
 int posa[maxn], posb[maxn], cnt[maxn];   
 void init(int \_n, int\* arr)   
 {   
 n = \_n;   
 pwn = 1; while (pwn < n) pwn <<= 1;   
 for (int i = 1; i <= n; i++)   
 rank[i] = arr[i];   
 for (int i = n + 1; i <= pwn; i++)   
 rank[i] = 0;   
 swap(n, pwn);   
 return ;   
 }   
 void build(void)   
 {   
 int c = max(n, 257);   
 for (int d = 1; d <= n; d <<= 1) {   
 // Initialize "string" to be compared...   
 rep (i, 1, n) stra[i] = rank[i],   
 strb[i] = rank[i + d];   
 // Resetting counter for bit-2 to be sorted   
 memclr(cnt);   
 rep (i, 1, n) cnt[strb[i]]++;   
 rep (i, 1, c) cnt[i] += cnt[i - 1];   
 // To sort according to second position   
 rep (i, 1, n) srta[cnt[strb[i]]] = stra[i],   
 srtb[cnt[strb[i]]] = strb[i],   
 posb[cnt[strb[i]]--] = i;   
 // Resetting counter for bit-1 to be sorted   
 memclr(cnt);   
 rep (i, 1, n) cnt[srta[i]]++;   
 rep (i, 1, c) cnt[i] += cnt[i - 1];   
 rep\_ (i, n, 1) stra[cnt[srta[i]]] = srta[i],   
 strb[cnt[srta[i]]] = srtb[i],   
 posa[cnt[srta[i]]--] = posb[i];   
 // Re-updating rank array and continue   
 rep (i, 1, n) rank[posa[i]] = stra[i] == stra[i - 1] &&   
 strb[i] == strb[i - 1]   
 ? rank[posa[i - 1]] : rank[posa[i - 1]] + 1;   
 continue;   
 }   
 swap(n, pwn);   
 // Reverse rank[] to be assigned / positioned easily.   
 for (int i = 1; i <= n; i++)   
 prank[rank[i]] = i;   
 return ;   
 }   
 } sa;   
 int n, arr[maxn];   
 char str[maxn], res[maxn];   
 void download(void)   
 {   
 // To retrieve the sorting procedures and send to output pipeline.   
 int pos = 0;   
 for (int i = 1; i <= 2 \* n; i++) {   
 if (sa.prank[i] > n)   
 continue;   
 // Assign result position's value   
 res[++pos] = arr[sa.prank[i] + n - 1];   
 }   
 return ;   
 }   
 void main(void)   
 {   
 scanf("%s", str);   
 n = strlen(str);   
 rep(i, 1, n) arr[i + n] = arr[i] = (int)str[i - 1];   
 // Done copying... now suffix sorting.   
 sa.init(n \* 2, arr);   
 sa.build();   
 // Downloading final results and output...   
 download();   
 for (int i = 1; i <= n; i++)   
 printf("%c", (char)res[i]);   
 return ;   
 }   
fm\_end(strings, suffix\_array);   
   
   
fm\_begin(strings, suffix\_automaton):   
 // @desc Suffix automaton   
 // @warning incompatible code style   
 // @warning not yet understood   
 fm\_const(int, MAXN, 100100);   
 struct NODE   
 {   
 int ch[26];   
 int len,fa;   
 NODE(){memset(ch,0,sizeof(ch));len=0;}   
 }dian[MAXN<<1];   
 int las=1,tot=1;   
 void add(int c)   
 {   
 int p=las;int np=las=++tot;   
 dian[np].len=dian[p].len+1;   
 for(;p&&!dian[p].ch[c];p=dian[p].fa)dian[p].ch[c]=np;   
 if(!p)dian[np].fa=1;//以上为case 1   
 else   
 {   
 int q=dian[p].ch[c];   
 if(dian[q].len==dian[p].len+1)dian[np].fa=q;//以上为case 2   
 else   
 {   
 int nq=++tot;dian[nq]=dian[q];   
 dian[nq].len=dian[p].len+1;   
 dian[q].fa=dian[np].fa=nq;   
 for(;p&&dian[p].ch[c]==q;p=dian[p].fa)dian[p].ch[c]=nq;   
 //以上为case 3   
 }   
 }   
 }   
 char s[MAXN];int len;   
 void main()   
 {   
 scanf("%s",s);len=strlen(s);   
 for(int i=0;i<len;i++)add(s[i]-'a');   
 }   
fm\_end(strings, suffix\_automaton);   
   
   
fm\_begin(strings, manacher):   
 // @desc Manacher algorithm, calculates longest palindrome in string   
 // @complexity Time: O(n), Space: O(n)   
 // @usage eval(s, begin, length): returns the longest palindrome in s and   
 // sets begin and length as result   
 fm\_const(int, maxn, 100100);   
 int n, str[maxn], p[maxn];   
 string eval(string s, int& begin, int& length)   
 {   
 n = s.length();   
 str[0] = -1;   
 rep(i, 1, n) {   
 str[2 \* i - 1] = s[i - 1];   
 str[2 \* i] = -1; // Some unused char   
 }   
 int mx = 0, id = 0, res\_len = 0, res\_center = 0;   
 rep(i, 1, 2 \* n) {   
 p[i] = mx > i ? min(p[2 \* id - 1], mx - i) : 1;   
 while (str[i + p[i]] == str[i - p[i]])   
 p[i] += 1;   
 if (mx < i + p[i]) {   
 mx = i + p[i];   
 id = i;   
 }   
 if (res\_len < p[i]) {   
 res\_len = p[i];   
 res\_center = i;   
 }   
 }   
 begin = (res\_center - res\_len) / 2;   
 length = res\_len;   
 return s.substr(begin, length - 1);   
 }   
fm\_end(strings, manacher);   
   
   
fm\_begin(trees, disjoint\_set):   
 // @desc Disjoint set   
 // @complexity Time: O(1), Space: O(n)   
 // @usage init(n, w[]): set n objects with weight w[1..n]   
 // @usage find(p): find the component id (parent of this component)   
 // @usage join(p, q): join component p with q   
 // @usage val[find(p)]: find weight of component p   
 // @usage   
 fm\_const(int, maxn, 1001000);   
 int n, par[maxn], size[maxn];   
 lli val[maxn];   
 void init(int n, lli w[] = nullptr)   
 {   
 rep(i, 1, n) {   
 par[i] = i;   
 size[i] = 1;   
 }   
 if (w != nullptr)   
 rep(i, 1, n)   
 val[i] = w[i];   
 return ;   
 }   
 int find(int p)   
 {   
 if (par[p] != p) {   
 int q = find(par[p]);   
 par[p] = q;   
 }   
 return par[p];   
 }   
 void join(int p, int q)   
 {   
 int gp = find(p), gq = find(q);   
 if (size[gq] < size[gp])   
 swap(gq, gp);   
 par[gp] = gq;   
 val[gq] += val[gp]; // @modify   
 size[gq] += size[gp];   
 return ;   
 }   
fm\_end(trees, disjoint\_set);   
   
   
fm\_begin(trees, segment\_tree):   
 // @desc Segment tree   
 // @complexity Time: O(n Log[n]), Space: O(n)   
 // @warning incompatible code style   
 // @warning disorganized functions   
 // 给出n个数，有两个操作，第一个操作是将区间[x,y]中的数都开根号，   
 // 第二个操作是求区间[x,y]的和。   
 fm\_const(int, maxn, 200100);   
 struct node   
 {   
 node \*lc, \*rc;   
 int lb, mb, rb;   
 lli sum;   
 } \*root, npool[maxn<<1];   
 int n, ncnt;   
 node\* make\_node(void)   
 {   
 node \*p = &npool[++ncnt];   
 p->lc = p->rc = NULL;   
 p->lb = p->mb = p->rb = 0;   
 p->sum = 0;   
 return p;   
 }   
 lli query(node \*p, int l, int r)   
 {   
 if (p->lb == l && p->rb == r) {   
 return p->sum;   
 }   
 if (r <= p->mb) {   
 return query(p->lc, l, r);   
 } else if (l > p->mb) {   
 return query(p->rc, l, r);   
 } else {   
 return query(p->lc, l, p->mb) +   
 query(p->rc, p->mb + 1, r);   
 }   
 return lli();   
 }   
 lli query(int l, int r)   
 {   
 if (l > r) swap(l, r);   
 return this->query(root, l, r);   
 }   
 void change(node \*p, int l, int r)   
 {   
 if (p->lb == l && p->rb == r) {   
 if (p->rb - p->lb + 1 == p->sum)   
 return ;   
 if (p->lb == p->rb) {   
 p->sum = sqrt(p->sum);   
 return ;   
 }   
 change(p->lc, l, p->mb);   
 change(p->rc, p->mb + 1, r);   
 p->sum = p->lc->sum + p->rc->sum;   
 return ;   
 }   
 if (r <= p->mb) {   
 change(p->lc, l, r);   
 } else if (l > p->mb) {   
 change(p->rc, l, r);   
 } else {   
 change(p->lc, l, p->mb);   
 change(p->rc, p->mb + 1, r);   
 }   
 p->sum = p->lc->sum + p->rc->sum;   
 return ;   
 }   
 void change(int l, int r)   
 {   
 if (l > r) swap(l, r);   
 this->change(root, l, r);   
 return ;   
 }   
 node\* build\_tree(int l, int r, lli arr[])   
 {   
 node \*p = make\_node();   
 int mid = (l + r) >> 1;   
 p->lb = l; p->mb = mid; p->rb = r;   
 if (p->lb == p->rb) {   
 p->sum = lli(arr[mid]);   
 } else {   
 p->lc = build\_tree(l, mid, arr);   
 p->rc = build\_tree(mid + 1, r, arr);   
 p->sum = p->lc->sum + p->rc->sum;   
 }   
 return p;   
 }   
 void build(int n, lli arr[])   
 {   
 this->ncnt = 0;   
 this->n = n;   
 this->root = this->build\_tree(1, n, arr);   
 return ;   
 }   
fm\_end(trees, segment\_tree);   
   
   
fm\_begin(trees, splay):   
 // @desc Splay tree (poj3580)   
 // @complexity Time: O(n Log[n]), Space: O(n)   
 // @usage main(): test function   
 // @warning this method is still incomplete, problems may exist, please   
 // check the robustness before using this   
 // @warning code style   
 fm\_const(int, maxn, 10010);   
 fm\_const(int, infinit, 1000000007);   
 int ch[maxn][2], parent[maxn], root, ncnt, n;   
 int size[maxn], val[maxn], sum[maxn], minn[maxn];   
 int lazyadd[maxn], lazyswp[maxn];   
 #define lc(x) ch[x][lazyswp[x]]   
 #define rc(x) ch[x][!lazyswp[x]]   
 #define par(x) parent[x]   
 int makenode(int q, int v)   
 {   
 int p = ++ncnt; n++;   
 lc(p) = rc(p) = 0;   
 par(p) = q;   
 size[p] = 1;   
 val[p] = sum[p] = minn[p] = v;   
 lazyadd[p] = lazyswp[p] = 0; // Initially they aren't lazy at all   
 return p;   
 }   
 void updateminn(int p)   
 {   
 minn[p] = p > 2 ? val[p] : infinit;   
 if (lc(p)) minn[p] = min(minn[p], minn[lc(p)]);   
 if (rc(p)) minn[p] = min(minn[p], minn[rc(p)]);   
 return ;   
 }   
 void dispatchlazyadd(int p)   
 {   
 // Separate dispatched lazy values to children   
 lazyadd[lc(p)] += lazyadd[p];   
 lazyadd[rc(p)] += lazyadd[p];   
 // Update children's initial values   
 val[lc(p)] += lazyadd[p];   
 val[rc(p)] += lazyadd[p];   
 // Update children's sums   
 sum[lc(p)] += size[lc(p)] \* lazyadd[p];   
 sum[rc(p)] += size[rc(p)] \* lazyadd[p];   
 // Update minimum queried values   
 minn[lc(p)] += lazyadd[p];   
 minn[rc(p)] += lazyadd[p];   
 // Finally reset lazy value   
 lazyadd[p] = 0;   
 return ;   
 }   
 bool dispatchlazyswp(int p)   
 {   
 if (!lazyswp[p]) return false;   
 lazyswp[lc(p)] ^= 1;   
 lazyswp[rc(p)] ^= 1;   
 swap(lc(p), rc(p));   
 lazyswp[p] = 0;   
 return true;   
 }   
 void rotate(int p)   
 {   
 int q = par(p), g = par(q), x = p == rc(q);   
 // Dispatching lazy values in case something goes wrong   
 dispatchlazyadd(q);   
 dispatchlazyadd(p);   
 if (dispatchlazyswp(q)) x ^= 1; // These make no modifications to the   
 // actual values   
 dispatchlazyswp(p);   
 // Relink connexions between nodes   
 ch[q][x] = ch[p][!x], par(ch[q][x]) = q;   
 ch[p][!x] = q, par(q) = p;   
 par(p) = g;   
 if (g) ch[g][rc(g) == q] = p;   
 // Update data values   
 size[q] = size[lc(q)] + 1 + size[rc(q)];   
 size[p] = size[lc(p)] + 1 + size[rc(p)];   
 sum[q] = sum[lc(q)] + val[q] + sum[rc(q)];   
 sum[p] = sum[lc(p)] + val[p] + sum[rc(p)];   
 updateminn(p);   
 updateminn(q);   
 return ;   
 }   
 void splay(int p, int t)   
 {   
 for (int q = 0; (q = par(p)) && q != t; rotate(p))   
 if (par(q) && par(q) != t)   
 rotate((p == lc(q)) == (q == lc(par(q))) ? q : p);   
 if (t == 0) root = p;   
 return ;   
 }   
 int suc(int p)   
 {   
 if (!rc(p)) { while (p == rc(par(p))) p = par(p); p = par(p); }   
 else { p = rc(p); while (lc(p)) p = lc(p); }   
 return p;   
 }   
 int find(int x)   
 {   
 int p = root;   
 while (true) {   
 if (x <= size[lc(p)]) {   
 p = lc(p);   
 continue;   
 } x -= size[lc(p)];   
 if (x <= 1)   
 return p;   
 x -= 1;   
 p = rc(p);   
 }   
 return 0;   
 }   
 void insert(int x, int v)   
 {   
 int lp = find(x), rp = suc(lp); // Operations should be guranteed that   
 // rp is valid   
 splay(rp, 0);   
 splay(lp, root);   
 int c = makenode(lp, v);   
 rc(lp) = c;   
 size[lp]++, sum[lp] += v;   
 size[rp]++, sum[rp] += v;   
 updateminn(lp);   
 updateminn(rp);   
 return ;   
 }   
 void remove(int x)   
 {   
 int lp = find(x - 1), rp = suc(x);   
 splay(rp, 0);   
 splay(lp, root);   
 int c = rc(lp);   
 size[lp]--, sum[lp] -= val[c];   
 size[rp]--, sum[rp] -= val[c];   
 updateminn(lp);   
 updateminn(rp);   
 n--;   
 return ;   
 }   
 int query\_sum(int l, int r)   
 {   
 int lp = find(l - 1), rp = find(r + 1);   
 splay(rp, 0);   
 splay(lp, root);   
 // Return data values   
 return sum[rc(lp)];   
 }   
 int query\_min(int l, int r)   
 {   
 int lp = find(l - 1), rp = find(r + 1);   
 splay(rp, 0);   
 splay(lp, root);   
 // Return data values   
 return minn[rc(lp)];   
 }   
 void modify\_add(int l, int r, int v)   
 {   
 int lp = find(l - 1), rp = find(r + 1);   
 splay(rp, 0);   
 splay(lp, root);   
 // Update data values   
 sum[rc(lp)] += size[rc(lp)] \* v;   
 sum[lp] += size[rc(lp)] \* v;   
 sum[rp] += size[rc(lp)] \* v;   
 minn[rc(lp)] += v;   
 val[rc(lp)] += v;   
 printf("$ modify\_add: it is %d who's talking about\n", rc(lp));   
 updateminn(lp);   
 updateminn(rp);   
 lazyadd[rc(lp)] += v;   
 return ;   
 }   
 void modify\_swp(int l, int r)   
 {   
 int lp = find(l - 1), rp = find(r + 1);   
 splay(rp, 0);   
 splay(lp, root);   
 // Updating data values, which were easier   
 lazyswp[rc(lp)] ^= 1;   
 return ;   
 }   
 void buildtree()   
 {   
 n = ncnt = 0;   
 root = makenode(0, 0);   
 rc(root) = makenode(root, 0);   
 minn[1] = minn[2] = infinit;   
 par(rc(root)) = root;   
 size[root]++;   
 return ;   
 }   
 #undef lc   
 #undef rc   
 #undef par   
 void test\_main(void)   
 {   
 buildtree();   
 printf("Program begun.\n");   
 while (true)   
 {   
 string a;   
 int b, c, d;   
 cin >> a;   
 if (a == "insert") {   
 cin >> b >> c;   
 insert(b + 1, c);   
 } else if (a == "delete") {   
 cin >> b;   
 remove(b + 1);   
 } else if (a == "sum") {   
 cin >> b >> c;   
 printf("sum %d %d = %d\n", b, c, query\_sum(b + 1, c + 1));   
 } else if (a == "min") {   
 cin >> b >> c;   
 printf("min %d %d = %d\n", b, c, query\_min(b + 1, c + 1));   
 } else if (a == "add") {   
 cin >> b >> c >> d;   
 modify\_add(b + 1, c + 1, d);   
 } else if (a == "reverse") {   
 cin >> b >> c;   
 modify\_swp(b + 1, c + 1);   
 } else if (a == "revolve") {   
 cin >> b >> c;   
 d = query\_sum(c + 1, c + 1);   
 insert(b, d);   
 }   
 }   
 return ;   
 }   
fm\_end(trees, splay);   
   
   
fm\_begin(trees, kd\_tree):   
 // @desc K-D tree   
 // @warning incompatible code style   
 // @warning yet not understood   
 /\*function of this program: build a 2d tree using the input training data   
 the input is exm\_set which contains a list of tuples (x,y)   
 the output is a 2d tree pointer\*/   
 struct data   
 {   
 double x = 0;   
 double y = 0;   
 };   
 struct Tnode   
 {   
 struct data dom\_elt;   
 int split;   
 struct Tnode \* left;   
 struct Tnode \* right;   
 };   
 bool cmp1(data a, data b){   
 return a.x < b.x;   
 }   
 bool cmp2(data a, data b){   
 return a.y < b.y;   
 }   
 bool equal(data a, data b){   
 if (a.x == b.x && a.y == b.y)   
 {   
 return true;   
 }   
 else{   
 return false;   
 }   
 }   
 void ChooseSplit(data exm\_set[], int size, int &split, data &SplitChoice){   
 /\*compute the variance on every dimension. Set split as the dismension   
 that have the biggest   
 variance. Then choose the instance which is the median on this split   
 dimension.\*/   
 /\*compute variance on the x,y dimension. DX=EX^2-(EX)^2\*/   
 double tmp1,tmp2;   
 tmp1 = tmp2 = 0;   
 for (int i = 0; i < size; ++i)   
 {   
 tmp1 += 1.0 / (double)size \* exm\_set[i].x \* exm\_set[i].x;   
 tmp2 += 1.0 / (double)size \* exm\_set[i].x;   
 }   
 double v1 = tmp1 - tmp2 \* tmp2; //compute variance on the x dimension   
 tmp1 = tmp2 = 0;   
 for (int i = 0; i < size; ++i)   
 {   
 tmp1 += 1.0 / (double)size \* exm\_set[i].y \* exm\_set[i].y;   
 tmp2 += 1.0 / (double)size \* exm\_set[i].y;   
 }   
 double v2 = tmp1 - tmp2 \* tmp2; //compute variance on the y dimension   
 split = v1 > v2 ? 0:1; //set the split dimension   
 if (split == 0)   
 {   
 sort(exm\_set,exm\_set + size, cmp1);   
 }   
 else{   
 sort(exm\_set,exm\_set + size, cmp2);   
 }   
 //set the split point value   
 SplitChoice.x = exm\_set[size / 2].x;   
 SplitChoice.y = exm\_set[size / 2].y;   
 }   
 Tnode\* build\_kdtree(data exm\_set[], int size, Tnode\* T){   
 //call function ChooseSplit to choose the split dimension and split pnt   
 if (size == 0){   
 return NULL;   
 }   
 else{   
 int split;   
 data dom\_elt;   
 ChooseSplit(exm\_set, size, split, dom\_elt);   
 data exm\_set\_right [100];   
 data exm\_set\_left [100];   
 int sizeleft ,sizeright;   
 sizeleft = sizeright = 0;   
 if (split == 0)   
 {   
 for (int i = 0; i < size; ++i)   
 {   
 if (!equal(exm\_set[i],dom\_elt) &&   
 exm\_set[i].x <= dom\_elt.x)   
 {   
 exm\_set\_left[sizeleft].x = exm\_set[i].x;   
 exm\_set\_left[sizeleft].y = exm\_set[i].y;   
 sizeleft++;   
 }   
 else if (!equal(exm\_set[i],dom\_elt) &&   
 exm\_set[i].x > dom\_elt.x)   
 {   
 exm\_set\_right[sizeright].x = exm\_set[i].x;   
 exm\_set\_right[sizeright].y = exm\_set[i].y;   
 sizeright++;   
 }   
 }   
 }   
 else{   
 for (int i = 0; i < size; ++i)   
 {   
 if (!equal(exm\_set[i],dom\_elt) &&   
 exm\_set[i].y <= dom\_elt.y)   
 {   
 exm\_set\_left[sizeleft].x = exm\_set[i].x;   
 exm\_set\_left[sizeleft].y = exm\_set[i].y;   
 sizeleft++;   
 }   
 else if (!equal(exm\_set[i],dom\_elt) &&   
 exm\_set[i].y > dom\_elt.y)   
 {   
 exm\_set\_right[sizeright].x = exm\_set[i].x;   
 exm\_set\_right[sizeright].y = exm\_set[i].y;   
 sizeright++;   
 }   
 }   
 }   
 T = new Tnode;   
 T->dom\_elt.x = dom\_elt.x;   
 T->dom\_elt.y = dom\_elt.y;   
 T->split = split;   
 T->left = build\_kdtree(exm\_set\_left, sizeleft, T->left);   
 T->right = build\_kdtree(exm\_set\_right, sizeright, T->right);   
 return T;   
 }   
 }   
 double Distance(data a, data b){   
 double tmp = (a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y);   
 return sqrt(tmp);   
 }   
 void searchNearest(Tnode \* Kd, data target, data &nearestpoint,   
 double & distance){   
 //1. 如果Kd是空的，则设dist为无穷大返回   
 //2. 向下搜索直到叶子结点   
 stack<Tnode\*> search\_path;   
 Tnode\* pSearch = Kd;   
 data nearest;   
 double dist;   
 while(pSearch != NULL)   
 {   
 //pSearch加入到search\_path中;   
 search\_path.push(pSearch);   
 if (pSearch->split == 0)   
 {   
 if(target.x <= pSearch->dom\_elt.x) /\* 如果小于就进入左子树 \*/   
 {   
 pSearch = pSearch->left;   
 }   
 else   
 {   
 pSearch = pSearch->right;   
 }   
 }   
 else{   
 if(target.y <= pSearch->dom\_elt.y) /\* 如果小于就进入左子树 \*/   
 {   
 pSearch = pSearch->left;   
 }   
 else   
 {   
 pSearch = pSearch->right;   
 }   
 }   
 }   
 //取出search\_path最后一个赋给nearest   
 nearest.x = search\_path.top()->dom\_elt.x;   
 nearest.y = search\_path.top()->dom\_elt.y;   
 search\_path.pop();   
 dist = Distance(nearest, target);   
 //3. 回溯搜索路径   
 Tnode\* pBack;   
 while(search\_path.size() != 0)   
 {   
 //取出search\_path最后一个结点赋给pBack   
 pBack = search\_path.top();   
 search\_path.pop();   
 if(pBack->left == NULL && pBack->right == NULL)   
 { /\* 如果pBack为叶子结点 \*/   
 if( Distance(nearest, target) >   
 Distance(pBack->dom\_elt, target) )   
 {   
 nearest = pBack->dom\_elt;   
 dist = Distance(pBack->dom\_elt, target);   
 }   
 }   
 else   
 {   
 int s = pBack->split;   
 if (s == 0)   
 {   
 if( fabs(pBack->dom\_elt.x - target.x) < dist)   
 { /\* 如果以target为中心的圆（球或超球），半径为dist的圆与   
 分割超平面相交， 那么就要跳到另一边的子空间去搜索 \*/   
 if( Distance(nearest, target) >   
 Distance(pBack->dom\_elt, target) )   
 {   
 nearest = pBack->dom\_elt;   
 dist = Distance(pBack->dom\_elt, target);   
 }   
 if(target.x <= pBack->dom\_elt.x) /\* 如果target位于   
 pBack的左子空间，那么就要跳到右子空间去搜索 \*/   
 pSearch = pBack->right;   
 else   
 pSearch = pBack->left; /\* 如果target位于pBack的右   
 子空间，那么就要跳到左子空间去搜索 \*/   
 if(pSearch != NULL)   
 //pSearch加入到search\_path中   
 search\_path.push(pSearch);   
 }   
 }   
 else {   
 if( fabs(pBack->dom\_elt.y - target.y) < dist) /\* 如果以   
 target为中心的圆（球或超球），半径为dist的圆与分割   
 超平面相交， 那么就要跳到另一边的子空间去搜索 \*/   
 {   
 if( Distance(nearest, target) >   
 Distance(pBack->dom\_elt, target) )   
 {   
 nearest = pBack->dom\_elt;   
 dist = Distance(pBack->dom\_elt, target);   
 }   
 if(target.y <= pBack->dom\_elt.y) /\* 如果target位于   
 pBack的左子空间，那么就要跳到右子空间去搜索 \*/   
 pSearch = pBack->right;   
 else   
 pSearch = pBack->left; /\* 如果target位于pBack的   
 右子空间，那么就要跳到左子空间去搜索 \*/   
 if(pSearch != NULL)   
 // pSearch加入到search\_path中   
 search\_path.push(pSearch);   
 }   
 }   
 }   
 }   
 nearestpoint.x = nearest.x;   
 nearestpoint.y = nearest.y;   
 distance = dist;   
 }   
 void main(){   
 data exm\_set[100]; //assume the max training set size is 100   
 double x,y;   
 int id = 0;   
 cout<<"Please input the training data in the form x y." <<   
 " One instance per line. Enter -1 -1 to stop."<<endl;   
 while (cin>>x>>y){   
 if (x == -1)   
 {   
 break;   
 }   
 else{   
 exm\_set[id].x = x;   
 exm\_set[id].y = y;   
 id++;   
 }   
 }   
 struct Tnode \* root = NULL;   
 root = build\_kdtree(exm\_set, id, root);   
 data nearestpoint;   
 double distance;   
 data target;   
 cout <<"Enter search point"<<endl;   
 while (cin>>target.x>>target.y)   
 {   
 searchNearest(root, target, nearestpoint, distance);   
 cout<<"The nearest distance is "<<distance<<   
 ",and the nearest point is "<<nearestpoint.x<<","<<   
 nearestpoint.y<<endl;   
 cout <<"Enter search point"<<endl;   
 }   
 }   
fm\_end(trees, kd\_tree);   
   
   
fm\_begin(graphs, basic\_graph):   
 // @desc Basic graph   
 // @complexity Time: O(m), Space: O(m)   
 // @usage init(): clear graph   
 // @usage add\_edge(u, v, len): creates directed edge   
 // @usage add\_edge\_bi(u, v, len): creates undirected edge   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 1001000);   
 struct edge   
 {   
 int u, v;   
 lli len;   
 edge \*next, \*rev;   
 };   
 edge epool[maxm], \*edges[maxn];   
 int ecnt;   
 edge\* add\_edge(int u, int v, lli len)   
 {   
 edge \*p = &epool[++ecnt];   
 p->u = u; p->v = v; p->len = len;   
 p->next = edges[u]; edges[u] = p;   
 p->rev = nullptr;   
 return p;   
 }   
 void add\_edge\_bi(int u, int v, lli len)   
 {   
 edge \*p = add\_edge(u, v, len),   
 \*q = add\_edge(v, u, len);   
 p->rev = q; q->rev = p;   
 return ;   
 }   
 void init(void)   
 {   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
fm\_end(graphs, basic\_graph);   
   
   
fm\_begin(graphs, dijkstra):   
 // @desc Shortest Path: Dijkstra   
 // suitable for single-source multi-target positive-weight graphs   
 // @complexity Time: O(m Log[n]), Space: O(m)   
 // @usage dist[i]: the distance from source to node i   
 // @usage add\_edge(u, v, len): create directed edge   
 // @usage eval(s): calculate all distances from source s   
 fm\_const(int, maxn, 100100);   
 fm\_const(int, maxm, 1001000);   
 fm\_const(lli, infinit, 0x007f7f7f7f7f7f7fll);   
 struct edge   
 {   
 int u, v;   
 lli len;   
 edge \*next;   
 };   
 edge epool[maxm], \*edges[maxn];   
 int n, ecnt;   
 lli dist[maxn];   
 typedef pair<lli, int> pli;   
 void add\_edge(int u, int v, lli len)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v; p->len = len;   
 p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u; q->len = len;   
 q->next = edges[v]; edges[v] = q;   
 return ;   
 }   
 void eval(int s)   
 {   
 priority\_queue<pli, vector<pli>, greater<pli>> pq;   
 rep(i, 0, n)   
 dist[i] = infinit;   
 dist[s] = 0;   
 pq.push(make\_pair(dist[s], s));   
 while (!pq.empty()) {   
 pli pr = pq.top();   
 int p = pr.second;   
 pq.pop();   
 if (dist[p] < pr.first)   
 continue;   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (dist[p] + ep->len < dist[ep->v]) {   
 dist[ep->v] = dist[p] + ep->len;   
 pq.push(make\_pair(dist[ep->v], ep->v));   
 }   
 }   
 return ;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
fm\_end(graphs, dijkstra);   
   
   
fm\_begin(graphs, floyd\_warshall):   
 // @desc Shortest Path: Floyd-Warshall   
 // suitable for multi-source multi-target positive-weight graphs   
 // @complexity Time: O(n^3), Space: O(n^2)   
 // @usage dist[i][j]: the distance from node i to j   
 // @usage add\_edge(u, v, len): create directed edge   
 // @usage eval(s): calculate all distances between vertex pairs   
 fm\_const(int, maxn, 1010);   
 fm\_const(lli, infinit, 0x007f7f7f7f7f7f7fll);   
 lli dist[maxn][maxn];   
 int n;   
 void add\_edge(int u, int v, lli len)   
 {   
 dist[u][v] = len;   
 return ;   
 }   
 void eval(void)   
 {   
 rep(k, 1, n)   
 rep(i, 1, n)   
 rep(j, 1, n)   
 if (dist[i][k] + dist[k][j] < dist[i][j])   
 dist[i][j] = dist[i][k] + dist[k][j];   
 return ;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 rep(i, 1, n)   
 rep(j, 1, n)   
 dist[i][j] = i == j ? 0 : infinit;   
 return ;   
 }   
fm\_end(graphs, floyd\_warshall);   
   
   
fm\_begin(graphs, tree\_diameter):   
 // @desc Evaluate tree diameter (edges between farthest points)   
 // @complexity Time: O(n), Space: O(n)   
 // @usage add\_edge(u, v): create edge   
 // @usage bfs(s): find farthest node with s as root   
 // @usage eval(): evaluate diameter   
 // @usage init(n): reset the graph and set vertex count as n   
 fm\_const(int, maxn, 100100);   
 struct edge   
 {   
 int u, v;   
 edge \*next;   
 };   
 edge epool[2 \* maxn], \*edges[maxn];   
 int n, ecnt;   
 int depth[maxn];   
 void add\_edge(int u, int v)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v; p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u; q->next = edges[v]; edges[v] = q;   
 return ;   
 }   
 int bfs(int s)   
 {   
 queue<int> que;   
 memclr(depth);   
 depth[s] = 1;   
 que.push(s);   
 while (!que.empty()) {   
 int p = que.front();   
 que.pop();   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (!depth[ep->v]) {   
 depth[ep->v] = depth[p] + 1;   
 que.push(ep->v);   
 }   
 }   
 int maxd = 0;   
 rep(i, 1, n)   
 if (depth[i] > depth[maxd])   
 maxd = i;   
 return maxd;   
 }   
 int eval(void)   
 {   
 int p = bfs(1),   
 q = bfs(p);   
 return depth[q] - 1;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
fm\_end(graphs, tree\_diameter);   
   
   
fm\_begin(graphs, tree\_center):   
 // @desc Evaluate tree center (when root removed the vertex with most   
 // descendants has minimum descendants possible)   
 // @complexity Time: O(n), Space: O(n)   
 // @usage add\_edge(u, v): create edge   
 // @usage bfs(s): find farthest node with s as root   
 // @usage eval(): evaluate diameter   
 // @usage init(n): reset the graph and set vertex count as n   
 fm\_const(int, maxn, 100100);   
 struct edge   
 {   
 int u, v;   
 edge \*next;   
 };   
 edge epool[2 \* maxn], \*edges[maxn];   
 int n, ecnt;   
 int size[maxn];   
 void add\_edge(int u, int v)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v; p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u; q->next = edges[v]; edges[v] = q;   
 return ;   
 }   
 void dfs(int p, int par, int& min\_p, int& min\_size)   
 {   
 size[p] = 1;   
 int maxcnt = 0;   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (ep->v != par) {   
 dfs(ep->v, p, min\_p, min\_size);   
 size[p] += size[ep->v];   
 maximize(maxcnt, size[ep->v]);   
 }   
 maximize(maxcnt, n - size[p]);   
 if (maxcnt < min\_size) {   
 min\_p = p;   
 min\_size = maxcnt;   
 }   
 return ;   
 }   
 int eval(void)   
 {   
 int min\_p = 0, min\_size = n;   
 dfs(1, 0, min\_p, min\_size);   
 return min\_p;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
fm\_end(graphs, tree\_center);   
   
   
fm\_begin(graphs, heavy\_light\_decomposition):   
 // @desc Heavy-light decomposition   
 // @complexity Time: O(n Log[n]^2), Space: O(n)   
 // @warning incompatible code style   
 // 你决定设计你自己的软件包管理器。不可避免地，你要解决软件包之间的依赖   
 // 问题。如果软件包A依赖软件包B，那么安装软件包A以前，必须先安装软件包B。   
 // 同时，如果想要卸载软件包B，则必须卸载软件包A。现在你已经获得了所有的   
 // 软件包之间的依赖关系。而且，由于你之前的工作，除0号软件包以外，在你的   
 // 管理器当中的软件包都会依赖一个且仅一个软件包，而0号软件包不依赖任何一个   
 // 软件包。依赖关系不存在环（若有m(m≥2)个软件包A1,A2,A3,…,Am，其中A1依赖   
 // A2，A2依赖A3，A3依赖A4，……，Am−1依赖Am，而Am依赖A1，则称这m个软件包的   
 // 依赖关系构成环），当然也不会有一个软件包依赖自己。   
 // 现在你要为你的软件包管理器写一个依赖解决程序。根据反馈，用户希望在安装   
 // 和卸载某个软件包时，快速地知道这个操作实际上会改变多少个软件包的安装状态   
 // （即安装操作会安装多少个未安装的软件包，或卸载操作会卸载多少个已安装的   
 // 软件包），你的任务就是实现这个部分。注意，安装一个已安装的软件包，或卸载   
 // 一个未安装的软件包，都不会改变任何软件包的安装状态，即在此情况下，改变   
 // 安装状态的软件包数为0。   
 // 输入文件的第1行包含1个正整数n，表示软件包的总数。软件包从0开始编号。   
 // 随后一行包含n−1个整数，相邻整数之间用单个空格隔开，分别表示1,2,3,…,   
 // n−2,n−1号软件包依赖的软件包的编号。   
 // 接下来一行包含1个正整数q，表示询问的总数。   
 // 之后q行，每行1个询问。询问分为两种：   
 // installx：表示安装软件包x   
 // uninstallx：表示卸载软件包x   
 // 你需要维护每个软件包的安装状态，一开始所有的软件包都处于未安装状态。   
 // 对于每个操作，你需要输出这步操作会改变多少个软件包的安装状态，随后应用   
 // 这个操作（即改变你维护的安装状态）。   
 fm\_const(int, maxn, 100100);   
 fm\_const(int, maxm, 400100);   
 fm\_const(int, maxlog, 17);   
 static class SegmentTree   
 {   
 public:   
 struct interval {   
 int lc, rc; // Left and right (boundary) colours   
 int cols; // Total consecutive colours   
 void set\_colour(int col) {   
 lc = rc = col;   
 cols = 1;   
 return ; }   
 interval(void) {   
 lc = rc = 0;   
 cols = 1; }   
 interval(int col) {   
 lc = rc = col;   
 cols = 1; }   
 interval(int l, int r, int col) {   
 lc = l, rc = r, cols = col;   
 return ; }   
 };   
 interval join(const interval& a, const interval& b) const {   
 int cols = a.cols + b.cols;   
 if (a.rc == b.lc) cols -= 1;   
 return interval(a.lc, b.rc, cols);   
 }   
 struct node   
 {   
 node \*lc, \*rc;   
 int lb, mb, rb, lazy;   
 interval val;   
 } \*root, npool[maxn<<1];   
 int n, ncnt;   
 node\* make\_node(void)   
 {   
 node \*p = &npool[++ncnt];   
 p->lc = p->rc = NULL;   
 p->lb = p->mb = p->rb = 0;   
 p->lazy = -1;   
 return p;   
 }   
 void dispatch\_lazy(node \*p)   
 {   
 if (p->lazy < 0 || p->lb == p->rb)   
 return ;   
 p->lc->lazy = p->rc->lazy = p->lazy;   
 p->lc->val.set\_colour(p->lazy);   
 p->rc->val.set\_colour(p->lazy);   
 p->lazy = -1;   
 return ;   
 }   
 void change(node \*p, int l, int r, int col)   
 {   
 if (p->lb == l && p->rb == r) {   
 p->lazy = col;   
 p->val.set\_colour(col);   
 return ;   
 }   
 dispatch\_lazy(p);   
 if (r <= p->mb) {   
 change(p->lc, l, r, col);   
 } else if (l > p->mb) {   
 change(p->rc, l, r, col);   
 } else {   
 change(p->lc, l, p->mb, col);   
 change(p->rc, p->mb + 1, r, col);   
 }   
 p->val = join(p->lc->val, p->rc->val);   
 return ;   
 }   
 void change(int l, int r, int col)   
 {   
 return this->change(root, l, r, col);   
 }   
 interval query(node \*p, int l, int r)   
 {   
 if (p->lb == l && p->rb == r) {   
 return p->val;   
 }   
 dispatch\_lazy(p);   
 if (r <= p->mb) {   
 return query(p->lc, l, r);   
 } else if (l > p->mb) {   
 return query(p->rc, l, r);   
 } else {   
 return join(query(p->lc, l, p->mb),   
 query(p->rc, p->mb + 1, r));   
 }   
 return interval();   
 }   
 interval query(int l, int r)   
 {   
 return this->query(root, l, r);   
 }   
 int query(int pos)   
 {   
 node \*p = root;   
 while (p->lb < p->rb) {   
 dispatch\_lazy(p);   
 if (pos <= p->mb)   
 p = p->lc;   
 else   
 p = p->rc;   
 }   
 return p->val.lc;   
 }   
 node\* build\_tree(int l, int r, int arr[])   
 {   
 node \*p = make\_node();   
 int mid = (l + r) >> 1;   
 p->lb = l; p->rb = r; p->mb = mid;   
 if (p->lb == p->rb) {   
 p->val = interval(arr[mid]);   
 } else {   
 p->lc = build\_tree(l, mid, arr);   
 p->rc = build\_tree(mid + 1, r, arr);   
 p->val = join(p->lc->val, p->rc->val);   
 }   
 return p;   
 }   
 void build(int n, int arr[])   
 {   
 root = build\_tree(1, n, arr);   
 return ;   
 }   
 } st;   
 static class TreeChainPartition   
 {   
 public:   
 struct edge   
 {   
 int u, v;   
 edge \*next;   
 };   
 int n, root, ecnt, dcnt;   
 int alt\_arr[maxn];   
 edge \*edges[maxn], epool[maxm];   
 void add\_edge(int u, int v)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v;   
 q->u = v; q->v = u;   
 p->next = edges[u]; edges[u] = p;   
 q->next = edges[v]; edges[v] = q;   
 return ;   
 }   
 int size[maxn], par[maxn], depth[maxn];   
 int maxch[maxn], ctop[maxn], dfn[maxn];   
 int jump[maxn][maxlog+1]; // Reserved for LCA   
 void dfs1(int p)   
 {   
 size[p] = 1;   
 for (int i = 1; i < maxlog; i++) {   
 if (depth[p] < (1<<i))   
 break;   
 jump[p][i] = jump[jump[p][i-1]][i-1];   
 }   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (ep->v != par[p]) {   
 par[ep->v] = p;   
 depth[ep->v] = depth[p] + 1;   
 jump[ep->v][0] = p;   
 dfs1(ep->v);   
 size[p] += size[ep->v];   
 if (size[ep->v] > size[maxch[p]])   
 maxch[p] = ep->v;   
 }   
 return ;   
 }   
 void dfs2(int p, int chaintop)   
 {   
 dfn[p] = ++dcnt;   
 ctop[p] = chaintop;   
 if (maxch[p])   
 dfs2(maxch[p], chaintop);   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (depth[ep->v] == depth[p] + 1 && ep->v != maxch[p])   
 dfs2(ep->v, ep->v);   
 return ;   
 }   
 int lca(int x, int y)   
 {   
 if (depth[x] < depth[y])   
 swap(x, y);   
 // Ensured that x is deeper than y   
 int dist = depth[x] - depth[y];   
 // Letting x reach the depth par y   
 for (int i = 0; i < maxlog; i++)   
 if (dist & (1<<i))   
 x = jump[x][i];   
 // Syncing ancestors   
 for (int i = maxlog - 1; i >= 0; i--)   
 if (jump[x][i] != jump[y][i])   
 x = jump[x][i],   
 y = jump[y][i];   
 if (x == y)   
 return x;   
 return jump[x][0];   
 }   
 void \_\_change(int x, int y, int colour)   
 {   
 while (ctop[x] != ctop[y]) {   
 st.change(dfn[ctop[x]], dfn[x], colour);   
 x = jump[ctop[x]][0];   
 }   
 st.change(dfn[y], dfn[x], colour);   
 return ;   
 }   
 int \_\_query(int x, int y)   
 {   
 int res = 0;   
 while (ctop[x] != ctop[y]) {   
 int tmp = st.query(dfn[ctop[x]], dfn[x]).cols;   
 res += tmp;   
 if (st.query(dfn[jump[ctop[x]][0]]) == st.query(dfn[ctop[x]]))   
 res -= 1;   
 x = jump[ctop[x]][0];   
 }   
 int tmp = st.query(dfn[y], dfn[x]).cols;   
 res += tmp;   
 return res;   
 }   
 void change(int x, int y, int colour)   
 {   
 int z = lca(x, y);   
 \_\_change(x, z, colour);   
 \_\_change(y, z, colour);   
 return ;   
 }   
 int query(int x, int y)   
 {   
 int z = lca(x, y);   
 int res = \_\_query(x, z)   
 + \_\_query(y, z) - 1;   
 return res;   
 }   
 void init(int n, int arr[])   
 {   
 this->n = n;   
 dcnt = 0;   
 this->root = 1;   
 // Generating DFS sequences   
 depth[root] = 1;   
 dfs1(root);   
 dfs2(root, root);   
 // Building segment tree, with minor modifications   
 for (int i = 1; i <= n; i++)   
 alt\_arr[dfn[i]] = arr[i];   
 st.build(n, alt\_arr);   
 return ;   
 }   
 } graph;   
 int n, m;   
 int arr[maxn];   
 char str[64];   
 void main(void)   
 {   
 scanf("%d%d", &n, &m);   
 for (int i = 1; i <= n; i++)   
 scanf("%d", &arr[i]);   
 for (int i = 1, a, b; i <= n - 1; i++) {   
 scanf("%d%d", &a, &b);   
 graph.add\_edge(a, b);   
 }   
 // Building graph with integrated functions   
 graph.init(n, arr);   
 // Answering queries   
 for (int idx = 1; idx <= m; idx++) {   
 scanf("%s", str);   
 int a, b, c;   
 if (str[0] == 'C') {   
 scanf("%d%d%d", &a, &b, &c);   
 graph.change(a, b, c);   
 } else if (str[0] == 'Q') {   
 scanf("%d%d", &a, &b);   
 int res = graph.query(a, b);   
 printf("%d\n", res);   
 }   
 }   
 // Finished   
 return ;   
 }   
fm\_end(graphs, heavy\_light\_decomposition);   
   
   
fm\_begin(graphs, link\_cut\_tree):   
 // @desc Link-cut tree   
 // @complexity Time: O(n Log[n]^2), Space: O(n)   
 // @warning incompatible code style   
 // @warning missing pushdown and pushup functions   
 // 某天，Lostmonkey发明了一种超级弹力装置，为了在他的绵羊朋友面前显摆，他   
 // 邀请小绵羊一起玩个游戏。游戏一开始，Lostmonkey在地上沿着一条直线摆上n个   
 // 装置，每个装置设定初始弹力系数ki，当绵羊达到第i个装置时，它会往后弹ki   
 // 步，达到第i+ki个装置，若不存在第i+ki个装置，则绵羊被弹飞。绵羊想知道当它   
 // 从第i个装置起步时，被弹几次后会被弹飞。为了使得游戏更有趣，Lostmonkey可   
 // 以修改某个弹力装置的弹力系数，任何时候弹力系数均为正整数。   
 // 第一行包含一个整数n，表示地上有n个装置，装置的编号从0到n-1,接下来一行有   
 // n个正整数，依次为那n个装置的初始弹力系数。第三行有一个正整数m，接下来m   
 // 行每行至少有两个数i、j，若i=1，你要输出从j出发被弹几次后被弹飞，若i=2则   
 // 还会再输入一个正整数k，表示第j个弹力装置的系数被修改成k。   
 fm\_const(int, maxn, 200100);   
 int arr\_i[maxn][5];   
 #define lc(\_x) arr\_i[\_x][0]   
 #define rc(\_x) arr\_i[\_x][1]   
 #define ch(\_x,\_y) arr\_i[\_x][\_y]   
 #define par(\_x) arr\_i[\_x][2]   
 #define size(\_x) arr\_i[\_x][3]   
 #define isroot(\_x) arr\_i[\_x][4]   
 int n;   
 void update\_lazy(int p)   
 {   
 size(p) = size(lc(p)) + 1 + size(rc(p));   
 return ;   
 }   
 void rotate(int p)   
 {   
 int q = par(p), g = par(q);   
 int x = rc(q) == p, y = q == rc(g);   
 ch(q, x) = ch(p, !x); if (ch(q, x)) par(ch(q, x)) = q;   
 ch(p, !x) = q; par(q) = p;   
 par(p) = g; // if (g) ch(g, y) = p;   
 if (isroot(q)) {   
 isroot(p) = true;   
 isroot(q) = false;   
 } else {   
 ch(g, y) = p;   
 }   
 update\_lazy(q);   
 update\_lazy(p);   
 return ;   
 }   
 void splay(int p)   
 {   
 for (int q = 0; (q = par(p)) && !isroot(p); rotate(p))   
 if (par(q) && !isroot(q))   
 rotate((p == rc(q)) == (q == rc(par(q))) ? q : p);   
 return ;   
 }   
 void access(int p)   
 {   
 int q = 0;   
 while (p) {   
 splay(p);   
 isroot(rc(p)) = true;   
 isroot(q) = false;   
 rc(p) = q;   
 update\_lazy(p);   
 q = p, p = par(p);   
 }   
 return ;   
 }   
 void makeroot(int p)   
 {   
 access(p);   
 splay(p);   
 return ;   
 }   
 void link(int p, int q)   
 {   
 makeroot(p);   
 par(lc(p)) = 0;   
 isroot(lc(p)) = true;   
 lc(p) = 0;   
 par(p) = q;   
 update\_lazy(p);   
 return ;   
 }   
 void init(int n\_)   
 {   
 this->n = n\_;   
 for (int i = 1; i <= n; i++) {   
 isroot(i) = true;   
 size(i) = 1;   
 }   
 return ;   
 }   
 int query(int p)   
 {   
 makeroot(p);   
 return size(lc(p)) + 1;   
 }   
 #undef lc   
 #undef rc   
 #undef ch   
 #undef par   
 #undef size   
 #undef isroot   
 void main(void)   
 {   
 int n, m;   
 scanf("%d", &n);   
 init(n);   
 for (int i = 1, a; i <= n; i++) {   
 scanf("%d", &a);   
 if (i + a <= n)   
 link(i, i + a);   
 }   
 scanf("%d", &m);   
 for (int idx = 1; idx <= m; idx++) {   
 int a, b, c;   
 scanf("%d", &a);   
 if (a == 1) { // To query   
 scanf("%d", &b);   
 b += 1; // Due to the strange marker descripted in the problem   
 printf("%d\n", query(b));   
 } else if (a == 2) { // To modify   
 scanf("%d%d", &b, &c);   
 b += 1; // Due to the strange marker descripted in the problem   
 link(b, b+c <= n ? b+c : 0);   
 }   
 }   
 return ;   
 }   
fm\_end(graphs, link\_cut\_tree);   
   
   
fm\_begin(graphs, prim):   
 // @desc Minimum span tree: Prim   
 // suitable for dense graphs   
 // @complexity Time: O(n^2), Space: O(n^2)   
 // @usage dist[i][j]: direct weight between vertex i and j   
 // @usage vis[i]: true if i was visited   
 // @usage min\_cost[i]: the id of the closest node to i, and is already   
 // inside the minimum span tree   
 // @usage add\_edge(u, v, len): create edge   
 // @usage mst(graph): evaluate mst and store edges in graph   
 // @usage init(n): reset the graph and set vertex count as n   
 fm\_const(int, maxn, 1010);   
 fm\_const(lli, infinit, 0x007f7f7f7f7f7f7fll);   
 lli dist[maxn][maxn];   
 int n, vis[maxn], min\_cost[maxn];   
 void add\_edge(int u, int v, lli len)   
 {   
 dist[u][v] = dist[v][u] = len;   
 return ;   
 }   
 lli mst(fm(graphs, basic\_graph)& graph)   
 {   
 lli min\_span = 0;   
 graph.init();   
 rep(i, 1, n) {   
 vis[i] = false;   
 min\_cost[i] = 1;   
 }   
 vis[1] = true;   
 rep(i, 1, n) {   
 int p = 0;   
 rep(j, 1, n)   
 if (!vis[j] && dist[min\_cost[j]][j] < dist[min\_cost[p]][p])   
 p = j;   
 if (p == 0)   
 break;   
 min\_span += dist[min\_cost[p]][p];   
 graph.add\_edge\_bi(min\_cost[p], p, dist[min\_cost[p]][p]);   
 vis[p] = true;   
 rep(j, 1, n)   
 if (dist[p][j] < dist[min\_cost[j]][j])   
 min\_cost[j] = p;   
 }   
 return min\_span;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 rep(i, 0, n)   
 rep(j, 0, n)   
 dist[i][j] = infinit;   
 return ;   
 }   
fm\_end(graphs, prim);   
   
   
fm\_begin(graphs, kruskal):   
 // @desc Minimum span tree: Kruskal   
 // suitable for sparse graphs   
 // @complexity Time: O(m Log[m]), Space: O(m)   
 // @usage edges[i]: the i-th edge   
 // @usage add\_edge(u, v, len): create edge   
 // @usage mst(graph): evaluate mst and store edges in graph   
 // @usage init(n): reset the graph and set vertex count as n   
 fm\_const(int, maxn, 100100);   
 fm\_const(int, maxm, 1001000);   
 struct edge   
 {   
 int u, v;   
 lli len;   
 bool operator < (const edge& b) const   
 {   
 return this->len < b.len;   
 }   
 };   
 edge edges[maxm];   
 int n, m;   
 fm(trees, disjoint\_set) djs;   
 void add\_edge(int u, int v, lli len)   
 {   
 edge \*ep = &edges[++m];   
 ep->u = u; ep->v = v; ep->len = len;   
 return ;   
 }   
 lli mst(fm(graphs, basic\_graph)& graph)   
 {   
 lli min\_span = 0;   
 int mst\_cnt = 0;   
 djs.init(n);   
 sort(edges + 1, edges + m);   
 rep(i, 1, m) {   
 if (mst\_cnt == n - 1)   
 break;   
 int u = edges[i].u, v = edges[i].v, len = edges[i].len,   
 gu = djs.find(u), gv = djs.find(v);   
 if (gu == gv)   
 continue;   
 djs.join(gu, gv);   
 min\_span += len;   
 mst\_cnt += 1;   
 graph.add\_edge\_bi(u, v, len);   
 }   
 return min\_span;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 m = 0;   
 return ;   
 }   
fm\_end(graphs, kruskal);   
   
   
fm\_begin(graphs, scc\_tarjan):   
 // @desc Strongly connected components: Tarjan   
 // A scc is a subgraph such that all nodes can reach each other in   
 // this directed graph   
 // @complexity Time: O(n+m), Space: O(n+m)   
 // @usage add\_edge(u, v): creates edge   
 // @usage init(n): clears graph   
 // @usage eval(): how many scc(s) in graph   
 // @usage belong[i]: the id of the scc i belongs to   
 // @usage bsize[i]: the size of the i-th scc   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 20010);   
 struct edge   
 {   
 int u, v;   
 edge \*next;   
 };   
 edge epool[maxm], \*edges[maxn];   
 int n, ecnt, dcnt, bcnt;   
 stack<int> stk;   
 int instk[maxn], dfn[maxn], low[maxn];   
 int belong[maxn], bsize[maxn];   
 void add\_edge(int u, int v)   
 {   
 edge \*p = &epool[++ecnt];   
 p->u = u; p->v = v;   
 p->next = edges[u]; edges[u] = p;   
 return ;   
 }   
 void dfs(int p)   
 {   
 low[p] = dfn[p] = ++dcnt;   
 stk.push(p);   
 instk[p] = true;   
 for (edge \*ep = edges[p]; ep; ep = ep->next) {   
 int q = ep->v;   
 if (!dfn[q]) {   
 dfs(q);   
 if (low[q] < low[p])   
 low[p] = low[q];   
 } else if (instk[q] && dfn[q] < low[p]) {   
 low[p] = dfn[q];   
 }   
 }   
 if (dfn[p] == low[p]) {   
 bsize[++bcnt] = 0;   
 int q = 0;   
 do {   
 q = stk.top();   
 stk.pop();   
 instk[q] = false;   
 belong[q] = bcnt;   
 bsize[bcnt] += 1;   
 } while (q != p);   
 }   
 return ;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
 int eval(void)   
 {   
 while (!stk.empty())   
 stk.pop();   
 dcnt = bcnt = 0;   
 memclr(dfn);   
 memclr(low);   
 memclr(instk);   
 memclr(belong);   
 rep(i, 1, n)   
 if (!dfn[i])   
 dfs(i);   
 return bcnt;   
 }   
fm\_end(graphs, scc\_tarjan);   
   
   
fm\_begin(graphs, dcc\_tarjan\_v):   
 // @desc Double connected components (Vertices): Tarjan   
 // A dcc-v is a subgraph such that all nodes can reach each other in   
 // at least two paths, different in nodes   
 // @complexity Time: O(n+m), Space: O(n+m)   
 // @usage add\_edge(u, v): creates edge   
 // @usage init(n): clears graph   
 // @usage eval(): how many dcc(s) in graph   
 // @usage dcc[i]: a vector describing a dcc   
 // @usage is\_cut[i]: if vertex i is a cut, a cut is a vertex such that   
 // removing it makes the graph disconnected   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 20010);   
 struct edge   
 {   
 int u, v;   
 edge \*next;   
 };   
 edge epool[maxm], \*edges[maxn];   
 int n, ecnt, dcnt, bcnt;   
 stack<edge\*> stk;   
 int instk[maxn], dfn[maxn], low[maxn];   
 int belong[maxn];   
 vector<int> dcc[maxn];   
 bool is\_cut[maxn];   
 void add\_edge(int u, int v)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v;   
 p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u;   
 q->next = edges[v]; edges[v] = q;   
 return ;   
 }   
 void dfs(int p, int par)   
 {   
 int child = 0;   
 dfn[p] = low[p] = ++dcnt;   
 for (edge \*ep = edges[p]; ep; ep = ep->next) {   
 int q = ep->v;   
 if (!dfn[q]) {   
 stk.push(ep);   
 child += 1;   
 dfs(ep->v, p);   
 minimize(low[p], low[q]);   
 if (dfn[p] <= low[q]) {   
 is\_cut[p] = true;   
 bcnt += 1;   
 edge \*eq = nullptr;   
 do {   
 eq = stk.top();   
 stk.pop();   
 if (belong[eq->u] != bcnt) {   
 belong[eq->u] = bcnt;   
 dcc[bcnt].push\_back(eq->u);   
 }   
 if (belong[eq->v] != bcnt) {   
 belong[eq->v] = bcnt;   
 dcc[bcnt].push\_back(eq->v);   
 }   
 } while (eq->u != p || eq->v != q);   
 }   
 } else if (dfn[q] < dfn[p] && q != par) {   
 stk.push(ep);   
 minimize(low[p], dfn[q]);   
 }   
 }   
 if (par == 0 && child == 1)   
 is\_cut[p] = false;   
 return ;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
 int eval(void)   
 {   
 while (!stk.empty())   
 stk.pop();   
 dcnt = bcnt = 0;   
 memclr(dfn);   
 memclr(low);   
 memclr(instk);   
 memclr(belong);   
 rep(i, 1, n)   
 dcc[i].clear();   
 memclr(is\_cut);   
 rep(i, 1, n)   
 if (!dfn[i])   
 dfs(i, 0);   
 return bcnt;   
 }   
fm\_end(graphs, dcc\_tarjan\_v);   
   
   
fm\_begin(graphs, dcc\_tarjan\_e):   
 // @desc Double connected components (Edges): Tarjan   
 // A dcc-e is a subgraph such that at least two paths composed of   
 // distinct edges exists between two arbitrary nodes   
 // @complexity Time: O(n+m), Space: O(n+m)   
 // @usage add\_edge(u, v): creates edge   
 // @usage init(n): clears graph   
 // @usage eval(): how many dcc(s) in graph   
 // @usage belong[i]: the id of the dcc i belongs to   
 // @usage bsize[i]: the size of the i-th dcc   
 // @usage bridges: a vector of bridges, such that removing this edge makes   
 // the graph disconnected   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 20010);   
 struct edge   
 {   
 int u, v;   
 bool is\_bridge;   
 edge \*next, \*rev;   
 };   
 edge epool[maxm], \*edges[maxn];   
 int n, ecnt, dcnt, bcnt;   
 int dfn[maxn], low[maxn];   
 int belong[maxn], bsize[maxn];   
 vector<pair<int, int>> bridges;   
 void add\_edge(int u, int v)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v; p->is\_bridge = false;   
 p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u; q->is\_bridge = false;   
 q->next = edges[v]; edges[v] = q;   
 p->rev = q; q->rev = p;   
 return ;   
 }   
 void tarjan(int p, int par)   
 {   
 dfn[p] = low[p] = ++dcnt;   
 for (edge \*ep = edges[p]; ep; ep = ep->next) {   
 int q = ep->v;   
 if (!dfn[q]) {   
 tarjan(q, p);   
 minimize(low[p], low[q]);   
 if (low[q] > dfn[p])   
 ep->is\_bridge = ep->rev->is\_bridge = true;   
 } else if (dfn[q] < dfn[p] && q != par) {   
 minimize(low[p], dfn[q]);   
 }   
 }   
 return ;   
 }   
 void dfs(int p)   
 {   
 dfn[p] = true;   
 belong[p] = bcnt;   
 bsize[bcnt] += 1;   
 for (edge \*ep = edges[p]; ep; ep = ep->next) {   
 if (ep->is\_bridge) {   
 if (ep->u < ep->v)   
 bridges.push\_back(make\_pair(ep->u, ep->v));   
 continue;   
 }   
 if (!dfn[ep->v])   
 dfs(ep->v);   
 }   
 return ;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
 int eval(void)   
 {   
 dcnt = bcnt = 0;   
 memclr(dfn);   
 memclr(low);   
 memclr(belong);   
 rep(i, 1, n)   
 if (!dfn[i])   
 tarjan(i, 0);   
 memclr(dfn);   
 bridges.clear();   
 rep(i, 1, n)   
 if (!dfn[i]) {   
 bsize[++bcnt] = 0;   
 dfs(i);   
 }   
 return bcnt;   
 }   
fm\_end(graphs, dcc\_tarjan\_e);   
   
   
fm\_begin(graphs, 2\_sat):   
 // @desc 2-satisfiability, uses Tarjan engine   
 // @complexity Time: O(n+m), Space: O(n+m)   
 // @usage init(): n variables   
 // @usage constrain(c, i): unary constraint operator   
 // @usage constrain(c, i, j): binary constraint operator   
 // @usage eval(): if the situation is satisfiable   
 typedef int constraint;   
 fm\_const(int, c\_select, 1); // (unary) choose i   
 fm\_const(int, c\_deselect, 2); // (unary) don't choose i   
 fm\_const(int, c\_and, 3); // choose i or j or both   
 fm\_const(int, c\_xor, 4); // i and j not chosen together   
 fm\_const(int, c\_same, 5); // status of i, j is same   
 fm\_const(int, c\_diff, 6); // status of i, j is opposite   
 int n;   
 fm(graphs, scc\_tarjan) scc;   
 void init(int n)   
 {   
 this->n = n;   
 scc.init(2 \* n);   
 return ;   
 }   
 void constrain(constraint c, int i, int j = 0)   
 {   
 #define node(\_x) (2 \* (\_x))   
 #define rnode(\_x) (2 \* (\_x) - 1)   
 if (c == c\_select) {   
 scc.add\_edge(rnode(i), node(i));   
 } else if (c == c\_deselect) {   
 scc.add\_edge(node(i), rnode(i));   
 } else if (c == c\_and) {   
 scc.add\_edge(rnode(i), node(j));   
 scc.add\_edge(rnode(j), node(i));   
 } else if (c == c\_xor) {   
 scc.add\_edge(node(i), rnode(j));   
 scc.add\_edge(node(j), rnode(i));   
 } else if (c == c\_same) {   
 scc.add\_edge(node(i), node(j));   
 scc.add\_edge(node(j), node(i));   
 scc.add\_edge(rnode(i), rnode(j));   
 scc.add\_edge(rnode(j), rnode(i));   
 } else if (c == c\_diff) {   
 scc.add\_edge(node(i), rnode(j));   
 scc.add\_edge(node(j), rnode(i));   
 scc.add\_edge(rnode(i), node(j));   
 scc.add\_edge(rnode(j), node(i));   
 }   
 #undef node   
 #undef rnode   
 return ;   
 }   
 bool eval(void)   
 {   
 scc.eval();   
 rep(i, 1, n)   
 if (scc.belong[2 \* i] == scc.belong[2 \* i - 1])   
 return false;   
 return true;   
 }   
fm\_end(graphs, 2\_sat);   
   
   
fm\_begin(graphs, dinic):   
 // @desc Max flow: Dinic   
 // The graph's maximum flow is also the minimum cost to cut the   
 // graph such that s and t becomes disconnected   
 // @complexity Time: O(n^2 m), Space: O(n+m)   
 // @usage add\_edge(u, v, flow, rflow): create edge   
 // @usage add\_edge(u, v, flow): create edge   
 // @usage init(n, s, t): init graph size n, source s, target t   
 // @usage eval(): evaluate maximum flow   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 20010);   
 fm\_const(lli, infinit, 0x007f7f7f7f7f7f7fll);   
 struct edge   
 {   
 int u, v;   
 lli flow;   
 edge \*next, \*rev;   
 };   
 edge epool[maxm], \*edges[maxn];   
 int n, s, t, ecnt, level[maxn];   
 void add\_edge(int u, int v, lli flow, lli rflow)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v; p->flow = flow;   
 p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u; q->flow = rflow;   
 q->next = edges[v]; edges[v] = q;   
 p->rev = q; q->rev = p;   
 return ;   
 }   
 void add\_edge(int u, int v, lli flow)   
 {   
 add\_edge(u, v, flow, 0);   
 return ;   
 }   
 bool make\_level(void)   
 {   
 memclr(level);   
 queue<int> que;   
 level[s] = 1;   
 que.push(s);   
 while (!que.empty()) {   
 int p = que.front();   
 que.pop();   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (ep->flow > 0 && !level[ep->v]) {   
 level[ep->v] = level[p] + 1;   
 que.push(ep->v);   
 }   
 if (level[t])   
 return true;   
 }   
 return level[t] > 0;   
 }   
 lli find(int p, lli mn)   
 {   
 if (p == t)   
 return mn;   
 lli tmp = 0, sum = 0;   
 for (edge \*ep = edges[p]; ep && sum < mn; ep = ep->next)   
 if (ep->flow && level[ep->v] == level[p] + 1) {   
 tmp = find(ep->v, min(mn, ep->flow));   
 if (tmp > 0) {   
 sum += tmp;   
 ep->flow -= tmp;   
 ep->rev->flow += tmp;   
 return tmp;   
 }   
 }   
 if (sum == 0)   
 level[p] = 0;   
 return 0;   
 }   
 void init(int n, int s, int t)   
 {   
 this->n = n; this->s = s; this->t = t;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
 lli eval(void)   
 {   
 lli tmp, sum = 0;   
 while (make\_level()) {   
 bool found = false;   
 while (tmp = find(s, infinit)) {   
 sum += tmp;   
 found = true;   
 }   
 if (!found)   
 break;   
 }   
 return sum;   
 }   
fm\_end(graphs, dinic);   
   
   
fm\_begin(graphs, spfa\_costflow):   
 // @desc Cost flow (Maximum flow, then minimum cost): SPFA ver.   
 // @complexity Time: O(n m^2), Space: O(n+m)   
 // @usage add\_edge(u, v, flow, cost): create edge   
 // @usage init(n, s, t): init graph size n, source s, target t   
 // @usage eval(): evaluate minimum cost under maximum flow   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 20010);   
 fm\_const(lli, infinit, 0x007f7f7f7f7f7f7fll);   
 struct edge   
 {   
 int u, v;   
 lli flow, cost;   
 edge \*next, \*rev;   
 };   
 edge epool[maxm], \*edges[maxn], \*from[maxn];   
 int n, s, t, ecnt, inque[maxn];   
 lli dist[maxn];   
 typedef pair<lli, int> pli;   
 void add\_edge(int u, int v, lli flow, lli cost)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v; p->flow = flow; p->cost = cost;   
 p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u; q->flow = 0; q->cost = - cost;   
 q->next = edges[v]; edges[v] = q;   
 p->rev = q; q->rev = p;   
 return ;   
 }   
 bool spfa(void)   
 {   
 rep(i, 1, n) {   
 inque[i] = false;   
 dist[i] = infinit;   
 from[i] = nullptr;   
 }   
 priority\_queue<pli, vector<pli>, greater<pli>> pq;   
 inque[s] = true;   
 dist[s] = 0;   
 pq.push(make\_pair(dist[s], s));   
 while (!pq.empty()) {   
 pli pr = pq.top();   
 int p = pr.second;   
 pq.pop();   
 if (dist[p] < pr.first)   
 continue;   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (ep->flow && dist[p] + ep->cost < dist[ep->v]) {   
 dist[ep->v] = dist[p] + ep->cost;   
 from[ep->v] = ep;   
 if (!inque[ep->v]) {   
 inque[ep->v] = true;   
 pq.push(make\_pair(dist[ep->v], ep->v));   
 }   
 }   
 inque[p] = false;   
 }   
 return dist[t] < infinit;   
 }   
 void init(int n, int s, int t)   
 {   
 this->n = n; this->s = s; this->t = t;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
 lli eval(void)   
 {   
 lli tmp, sum = 0;   
 while (spfa()) {   
 tmp = infinit;   
 for (edge \*ep = from[t]; ep; ep = from[ep->u])   
 minimize(tmp, ep->flow);   
 for (edge \*ep = from[t]; ep; ep = from[ep->u]) {   
 ep->flow -= tmp;   
 ep->rev->flow += tmp;   
 }   
 sum += tmp \* dist[t];   
 }   
 return sum;   
 }   
fm\_end(graphs, spfa\_costflow);   
   
   
fm\_begin(graphs, zkw\_costflow):   
 // @desc Cost flow (Maximum flow, then minimum cost): ZKW ver.   
 // @complexity Time: O(n m), Space: O(n+m)   
 // @usage add\_edge(u, v, flow, cost): create edge   
 // @usage init(n, s, t): init graph size n, source s, target t   
 // @usage eval(): evaluate minimum cost under maximum flow   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 20010);   
 fm\_const(lli, infinit, 0x007f7f7f7f7f7f7fll);   
 struct edge   
 {   
 int u, v;   
 lli flow, cost;   
 edge \*next, \*rev;   
 };   
 edge epool[maxm], \*edges[maxn], \*cursor[maxn];   
 int n, s, t, ecnt, cur[maxn], vis[maxn];   
 lli dist[maxn];   
 void add\_edge(int u, int v, lli flow, lli cost)   
 {   
 edge \*p = &epool[++ecnt],   
 \*q = &epool[++ecnt];   
 p->u = u; p->v = v; p->flow = flow; p->cost = cost;   
 p->next = edges[u]; edges[u] = p;   
 q->u = v; q->v = u; q->flow = 0; q->cost = - cost;   
 q->next = edges[v]; edges[v] = q;   
 p->rev = q; q->rev = p;   
 return ;   
 }   
 lli augment(int p, lli mn)   
 {   
 if (p == t)   
 return mn;   
 vis[p] = true;   
 for (edge \*ep = cursor[p]; ep; ep = ep->next)   
 if (ep->flow && !vis[ep->v] && dist[ep->v] + ep->cost == dist[p]) {   
 lli tmp = augment(ep->v, min(mn, ep->flow));   
 if (tmp > 0) {   
 ep->flow -= tmp;   
 ep->rev->flow += tmp;   
 cursor[p] = ep;   
 return tmp;   
 }   
 }   
 return 0;   
 }   
 bool mod\_label(void)   
 {   
 lli tmp = infinit;   
 rep(i, 1, n)   
 if (vis[i])   
 for (edge \*ep = edges[i]; ep; ep = ep->next)   
 if (ep->flow && !vis[ep->v])   
 minimize(tmp, dist[ep->v] + ep->cost - dist[i]);   
 if (tmp == infinit)   
 return false;   
 rep(i, 1, n)   
 if (vis[i]) {   
 vis[i] = false;   
 dist[i] += tmp;   
 }   
 return true;   
 }   
 void init(int n, int s, int t)   
 {   
 this->n = n; this->s = s; this->t = t;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
 lli eval(void)   
 {   
 lli tmp, sum = 0;   
 do {   
 rep(i, 1, n)   
 cursor[i] = edges[i];   
 while (tmp = augment(s, infinit)) {   
 sum += tmp \* dist[s];   
 memclr(vis);   
 }   
 } while (mod\_label());   
 return sum;   
 }   
fm\_end(graphs, zkw\_costflow);   
   
   
fm\_begin(graphs, hungary\_match):   
 // @desc Bipartite maximum match: Hungary algorithm   
 // @complexity Time: O(n m), Space: O(n+m)   
 // @usage add\_edge(u, v): create edge   
 // @usage init(n): init graph size n   
 // @usage eval(): evaluate how many matches between the bipartite graph   
 // can be made (how many pairs)   
 fm\_const(int, maxn, 1010);   
 fm\_const(int, maxm, 20010);   
 struct edge   
 {   
 int u, v;   
 edge \*next;   
 };   
 edge epool[maxm], \*edges[maxn];   
 int n, ecnt, from[maxn], vis[maxn];   
 void add\_edge(int u, int v)   
 {   
 edge \*p = &epool[++ecnt];   
 p->u = u; p->v = v;   
 p->next = edges[u]; edges[u] = p;   
 return ;   
 }   
 bool find(int p)   
 {   
 for (edge \*ep = edges[p]; ep; ep = ep->next)   
 if (!vis[ep->v]) {   
 vis[ep->v] = true;   
 if (!from[ep->v] || find(from[ep->v])) {   
 from[ep->v] = p;   
 return true;   
 }   
 }   
 return false;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 ecnt = 0;   
 memclr(edges);   
 return ;   
 }   
 int eval(void)   
 {   
 int sum = 0;   
 rep(i, 1, n) {   
 memclr(vis);   
 if (!from[i])   
 sum += (int)(find(i));   
 }   
 return sum;   
 }   
fm\_end(graphs, hungary\_match);   
   
   
fm\_begin(graphs, bron\_kerbosch):   
 // @desc Max clique: Bron-Kerbosch algorithm   
 // Largest independent set   
 // @complexity Time: O(3^(n/3)), Space: O(n^2)   
 // @usage add\_edge(u, v): create edge   
 // @usage init(n): clear graph   
 // @usage eval(): yields size of max clique   
 // @usage clique[i]: the i-th vertex in its max clique   
 // @usage if largest independent set (largest subgraph that are not   
 // connected to each other) is required, send its supplementary   
 // graph to this algorithm and the max clique is the result   
 fm\_const(int, maxn, 1010);   
 bool edge[maxn][maxn];   
 int n, cnt[maxn], vis[maxn], clique[maxn];   
 void add\_edge(int u, int v)   
 {   
 edge[u][v] = edge[v][u] = true;   
 return ;   
 }   
 bool dfs(int p, int pos, int& res)   
 {   
 rep(i, p + 1, n) {   
 if (cnt[i] + pos <= res)   
 return false;   
 if (edge[p][i]) {   
 int j = 0;   
 for (; j < pos; j++)   
 if (!edge[i][vis[j]])   
 break;   
 if (j == pos) {   
 vis[j] = i;   
 if (dfs(i, pos + 1, res))   
 return true;   
 }   
 }   
 }   
 if (pos > res) {   
 res = pos;   
 rep(i, 0, res - 1)   
 clique[i + 1] = vis[i];   
 return true;   
 }   
 return false;   
 }   
 void init(int n)   
 {   
 this->n = n;   
 memclr(edge);   
 memclr(cnt);   
 memclr(vis);   
 memclr(clique);   
 return ;   
 }   
 int eval(void)   
 {   
 int res = -1;   
 rep\_(i, n, 1) {   
 vis[0] = i;   
 dfs(i, 1, res);   
 cnt[i] = res;   
 }   
 return res;   
 }   
fm\_end(graphs, bron\_kerbosch);   
   
   
// @desc Kuangbin's computational geometry templates   
// @warning these templates are expected to run smoothly, but not expected   
// to pass compiler here   
namespace kuangbin   
{   
namespace chapter\_1\_x   
{   
// @chapter 1.1 Point 定义   
const double eps = 1e-8;   
const double PI = acos(-1.0);   
int sgn(double x)   
{   
 if(fabs(x) < eps)return 0;   
 if(x < 0)return -1;   
 else return 1;   
}   
struct point   
{   
 double x,y;   
 point() {}   
 point(double \_x,double \_y)   
 {   
 x = \_x;   
 y = \_y;   
 }   
 point operator -(const point &b)const   
 {   
 return point(x - b.x,y - b.y);   
 }   
//叉积   
 double operator ^(const point &b)const   
 {   
 return x\*b.y - y\*b.x;   
 }   
//点积   
 double operator \*(const point &b)const   
 {   
 return x\*b.x + y\*b.y;   
 }   
//绕原点旋转角度B（弧度值），后x,y的变化   
 void transXY(double B)   
 {   
 double tx = x,ty = y;   
 x = tx\*cos(B) - ty\*sin(B);   
 y = tx\*sin(B) + ty\*cos(B);   
 }   
 void input()   
 {   
 scanf("%lf%lf",&x,&y);   
 }   
};   
// @chapter 1.2 Line 定义   
struct Line   
{   
 point s,e;   
 Line() {}   
 Line(point \_s,point \_e)   
 {   
 s = \_s;   
 e = \_e;   
 }   
//两直线相交求交点   
//第一个值为0表示直线重合，为1表示平行，为0表示相交,为2是相交   
//只有第一个值为2时，交点才有意义   
 pair<int,point> operator &(const Line &b)const   
 {   
 point res = s;   
 if(sgn((s-e)^(b.s-b.e)) == 0)   
 {   
 if(sgn((s-b.e)^(b.s-b.e)) == 0)   
 return make\_pair(0,res);//重合   
 else return make\_pair(1,res);//平行   
 }   
 double t = ((s-b.s)^(b.s-b.e))/((s-e)^(b.s-b.e));   
 res.x += (e.x-s.x)\*t;   
 res.y += (e.y-s.y)\*t;   
 return make\_pair(2,res);   
 }   
};   
// @chapter 1.3 两点间距离   
//\*两点间距离   
double dist(point a,point b)   
{   
 return sqrt((a-b)\*(a-b));   
}   
// @chapter 1.4 判断 线段相交   
bool inter(Line l1,Line l2)   
{   
 return   
 max(l1.s.x,l1.e.x) >= min(l2.s.x,l2.e.x) &&   
 max(l2.s.x,l2.e.x) >= min(l1.s.x,l1.e.x) &&   
 max(l1.s.y,l1.e.y) >= min(l2.s.y,l2.e.y) &&   
 max(l2.s.y,l2.e.y) >= min(l1.s.y,l1.e.y) &&   
 sgn((l2.s-l1.e)^(l1.s-l1.e))\*sgn((l2.e-l1.e)^(l1.s-l1.e)) <= 0 &&   
 sgn((l1.s-l2.e)^(l2.s-l2.e))\*sgn((l1.e-l2.e)^(l2.s-l2.e)) <= 0;   
}   
// @chapter 1.5 判断 直线和线段相交   
//判断直线和线段相交   
bool Seg\_inter\_line(Line l1,Line l2) //判断直线l1和线段l2是否相交   
{   
 return sgn((l2.s-l1.e)^(l1.s-l1.e))\*sgn((l2.e-l1.e)^(l1.s-l1.e)) <= 0;   
}   
// @chapter 1.6 点到直线距离   
//点到直线距离   
//返回为result,是点到直线最近的点   
point PointToLine(point P,Line L)   
{   
 point result;   
 double t = ((P-L.s)\*(L.e-L.s))/((L.e-L.s)\*(L.e-L.s));   
 result.x = L.s.x + (L.e.x-L.s.x)\*t;   
 result.y = L.s.y + (L.e.y-L.s.y)\*t;   
 return result;   
}   
// @chapter 1.7 点到线段距离   
point NearestPointToLineSeg(point P,Line L)   
{   
 point result;   
 double t = ((P-L.s)\*(L.e-L.s))/((L.e-L.s)\*(L.e-L.s));   
 if(t >= 0 && t <= 1)   
 {   
 result.x = L.s.x + (L.e.x - L.s.x)\*t;   
 result.y = L.s.y + (L.e.y - L.s.y)\*t;   
 }   
 else   
 {   
 if(dist(P,L.s) < dist(P,L.e))   
 result = L.s;   
 else result = L.e;   
 }   
 return result;   
}   
// @chapter 1.8 计算多边形面积   
//计算多边形面积   
//点的编号从0~n-1   
double CalcArea(point p[],int n)   
{   
 double res = 0;   
 for(int i = 0; i < n; i++)   
 res += (p[i]^p[(i+1)%n])/2;   
 return fabs(res);   
}   
// @chapter 1.9 判断点在线段上   
//\*判断点在线段上   
bool OnSeg(point P,Line L)   
{   
 return   
 sgn((L.s-P)^(L.e-P)) == 0 &&   
 sgn((P.x - L.s.x) \* (P.x - L.e.x)) <= 0 &&   
 sgn((P.y - L.s.y) \* (P.y - L.e.y)) <= 0;   
}   
// @chapter 1.10 判断点在凸多边形内   
//\*判断点在凸多边形内   
//点形成一个凸包，而且按逆时针排序（如果是顺时针把里面的<0改为>0）   
//点的编号:0~n-1   
//返回值：   
//-1:点在凸多边形外   
//0:点在凸多边形边界上   
//1:点在凸多边形内   
int inConvexPoly(point a,point p[],int n)   
{   
 for(int i = 0; i < n; i++)   
 {   
 if(sgn((p[i]-a)^(p[(i+1)%n]-a)) < 0)return -1;   
 else if(OnSeg(a,Line(p[i],p[(i+1)%n])))return 0;   
 }   
 return 1;   
}   
// @chapter 1.11 判断点在任意多边形内   
//\*判断点在任意多边形内   
//射线法，poly[]的顶点数要大于等于3,点的编号0~n-1   
//返回值   
//-1:点在凸多边形外   
//0:点在凸多边形边界上   
//1:点在凸多边形内   
int inPoly(point p,point poly[],int n)   
{   
 int cnt;   
 Line ray,side;   
 cnt = 0;   
 ray.s = p;   
 ray.e.y = p.y;   
 ray.e.x = -100000000000.0;//-INF,注意取值防止越界   
 for(int i = 0; i < n; i++)   
 {   
 side.s = poly[i];   
 side.e = poly[(i+1)%n];   
 if(OnSeg(p,side))return 0;   
//如果平行轴则不考虑   
 if(sgn(side.s.y - side.e.y) == 0)   
 continue;   
 if(OnSeg(side.s,ray))   
 {   
 if(sgn(side.s.y - side.e.y) > 0)cnt++;   
 }   
 else if(OnSeg(side.e,ray))   
 {   
 if(sgn(side.e.y - side.s.y) > 0)cnt++;   
 }   
 else if(inter(ray,side))   
 cnt++;   
 }   
 if(cnt % 2 == 1)return 1;   
 else return -1;   
}   
// @chapter 1.12 判断多边形   
//判断凸多边形   
//允许共线边   
//点可以是顺时针给出也可以是逆时针给出   
//点的编号1~n-1   
bool isconvex(point poly[],int n)   
{   
 bool s[3];   
 memset(s,false,sizeof(s));   
 for(int i = 0; i < n; i++)   
 {   
 s[sgn( (poly[(i+1)%n]-poly[i])^(poly[(i+2)%n]-poly[i]) )+1] = true;   
 if(s[0] && s[2])return false;   
 }   
 return true;   
}   
// @chapter 1.13 简单极角排序   
const int maxn=55;   
point List[maxn];   
bool \_cmp(point p1,point p2)   
{   
 double tmp = (p1-List[0])^(p2-List[0]);   
 if(sgn(tmp) > 0)return true;   
 else if(sgn(tmp) == 0 && sgn(dist(p1,List[0]) - dist(p2,List[0])) <= 0)   
 return true;   
 else return false;   
}   
// sort(List+1,List+n,\_cmp);   
};   
using namespace chapter\_1\_x;   
namespace chapter\_2\_x   
{   
// @chapter 2.1 凸包   
/\*   
\* 求凸包，Graham算法   
\* 点的编号0~n-1   
\* 返回凸包结果Stack[0~top-1]为凸包的编号   
\*/   
const int MAXN = 1010;   
point List[MAXN];   
int Stack[MAXN],top;   
//相对于List[0]的极角排序   
bool \_cmp(point p1,point p2)   
{   
 double tmp = (p1-List[0])^(p2-List[0]);   
 if(sgn(tmp) > 0)return true;   
 else if(sgn(tmp) == 0 && sgn(dist(p1,List[0]) - dist(p2,List[0])) <= 0)   
 return true;   
 else return false;   
}   
void Graham(int n)   
{   
 point p0;   
 int k = 0;   
 p0 = List[0];   
//找最下边的一个点   
 for(int i = 1; i < n; i++)   
 {   
 if( (p0.y > List[i].y) || (p0.y == List[i].y && p0.x > List[i].x) )   
 {   
 p0 = List[i];   
 k = i;   
 }   
 }   
 swap(List[k],List[0]);   
 sort(List+1,List+n,\_cmp);   
 if(n == 1)   
 {   
 top = 1;   
 Stack[0] = 0;   
 return;   
 }   
 if(n == 2)   
 {   
 top = 2;   
 Stack[0] = 0;   
 Stack[1] = 1;   
 return ;   
 }   
 Stack[0] = 0;   
 Stack[1] = 1;   
 top = 2;   
 for(int i = 2; i < n; i++)   
 {   
 while(top > 1 &&   
 sgn((List[Stack[top-1]]-List[Stack[top-2]])^   
 (List[i]-List[Stack[top-2]])) <=   
 0)top--;   
 Stack[top++] = i;   
 }   
}   
};   
namespace chapter\_3\_x   
{   
// @chapter 3.1 平面最近点对   
#include <stdio.h>   
#include <string.h>   
#include <algorithm>   
#include <iostream>   
#include <math.h>   
using namespace std;   
const double eps = 1e-6;   
const int MAXN = 100010;   
const double INF = 1e20;   
struct Point   
{   
 double x,y;   
};   
double dist(Point a,Point b)   
{   
 return sqrt((a.x-b.x)\*(a.x-b.x) + (a.y-b.y)\*(a.y-b.y));   
}   
Point p[MAXN];   
Point tmpt[MAXN];   
bool cmpxy(Point a,Point b)   
{   
 if(a.x != b.x)return a.x < b.x;   
 else return a.y < b.y;   
}   
bool cmpy(Point a,Point b)   
{   
 return a.y < b.y;   
}   
double Closest\_Pair(int left,int right)   
{   
 double d = INF;   
 if(left == right)return d;   
 if(left + 1 == right)   
 return dist(p[left],p[right]);   
 int mid = (left+right)/2;   
 double d1 = Closest\_Pair(left,mid);   
 double d2 = Closest\_Pair(mid+1,right);   
 d = min(d1,d2);   
 int k = 0;   
 for(int i = left; i <= right; i++)   
 {   
 if(fabs(p[mid].x - p[i].x) <= d)   
 tmpt[k++] = p[i];   
 }   
 sort(tmpt,tmpt+k,cmpy);   
 for(int i = 0; i <k; i++)   
 {   
 for(int j = i+1; j < k && tmpt[j].y - tmpt[i].y < d; j++)   
 {   
 d = min(d,dist(tmpt[i],tmpt[j]));   
 }   
 }   
 return d;   
}   
int main()   
{   
 int n;   
 while(scanf("%d",&n)==1 && n)   
 {   
 for(int i = 0; i < n; i++)   
 scanf("%lf%lf",&p[i].x,&p[i].y);   
 sort(p,p+n,cmpxy);   
 printf("%.2lf\n",Closest\_Pair(0,n-1)/2);   
 }   
 return 0;   
}   
};   
namespace chapter\_4\_1   
{   
// @chapter 4.1 旋转卡壳 / 平面最远点对   
struct Point   
{   
 int x,y;   
 Point(int \_x = 0,int \_y = 0)   
 {   
 x = \_x;   
 y = \_y;   
 }   
 Point operator -(const Point &b)const   
 {   
 return Point(x - b.x, y - b.y);   
 }   
 int operator ^(const Point &b)const   
 {   
 return x\*b.y - y\*b.x;   
 }   
 int operator \*(const Point &b)const   
 {   
 return x\*b.x + y\*b.y;   
 }   
 void input()   
 {   
 scanf("%d%d",&x,&y);   
 }   
};   
//距离的平方   
int dist2(Point a,Point b)   
{   
 return (a-b)\*(a-b);   
}   
//\*\*\*\*\*\*二维凸包，int\*\*\*\*\*\*\*\*\*\*\*   
const int MAXN = 50010;   
Point list[MAXN];   
int Stack[MAXN],top;   
bool \_cmp(Point p1,Point p2)   
{   
 int tmp = (p1-list[0])^(p2-list[0]);   
 if(tmp > 0)return true;   
 else if(tmp == 0 && dist2(p1,list[0]) <= dist2(p2,list[0]))   
 return true;   
 else return false;   
}   
void Graham(int n)   
{   
 Point p0;   
 int k = 0;   
 p0 = list[0];   
 for(int i = 1; i < n; i++)   
 if(p0.y > list[i].y || (p0.y == list[i].y && p0.x > list[i].x))   
 {   
 p0 = list[i];   
 k = i;   
 }   
 swap(list[k],list[0]);   
 sort(list+1,list+n,\_cmp);   
 if(n == 1)   
 {   
 top = 1;   
 Stack[0] = 0;   
 return;   
 }   
 if(n == 2)   
 {   
 top = 2;   
 Stack[0] = 0;   
 Stack[1] = 1;   
 return;   
 }   
 Stack[0] = 0;   
 Stack[1] = 1;   
 top = 2;   
 for(int i = 2; i < n; i++)   
 {   
 while(top > 1 &&   
 ((list[Stack[top-1]]-list[Stack[top-2]])^   
 (list[i]-list[Stack[top-2]])) <= 0)   
 top--;   
 Stack[top++] = i;   
 }   
}   
//旋转卡壳，求两点间距离平方的最大值   
int rotating\_calipers(Point p[],int n)   
{   
 int ans = 0;   
 Point v;   
 int cur = 1;   
 for(int i = 0; i < n; i++)   
 {   
 v = p[i]-p[(i+1)%n];   
 while((v^(p[(cur+1)%n]-p[cur])) < 0)   
 cur = (cur+1)%n;   
 ans = max(ans,max(dist2(p[i],p[cur]),dist2(p[(i+1)%n],p[(cur+1)%n])));   
 }   
 return ans;   
}   
Point p[MAXN];   
int main()   
{   
 int n;   
 while(scanf("%d",&n) == 1)   
 {   
 for(int i = 0; i < n; i++)list[i].input();   
 Graham(n);   
 for(int i = 0; i < top; i++)p[i] = list[Stack[i]];   
 printf("%d\n",rotating\_calipers(p,top));   
 }   
 return 0;   
}   
};   
namespace chapter\_4\_2   
{   
// @chapter 4.2 旋转卡壳计算平面点集最大三角形面积   
//旋转卡壳计算平面点集最大三角形面积   
int rotating\_calipers(point p[],int n)   
{   
 int ans = 0;   
 point v;   
 for(int i = 0; i < n; i++)   
 {   
 int j = (i+1)%n;   
 int k = (j+1)%n;   
 while(j != i && k != i)   
 {   
 ans = max(ans,int(abs((p[j]-p[i])^(p[k]-p[i]))));   
 while( ((p[i]-p[j])^(p[(k+1)%n]-p[k])) < 0 )   
 k = (k+1)%n;   
 j = (j+1)%n;   
 }   
 }   
 return ans;   
}   
point p[10010];   
using namespace chapter\_3\_x;   
using namespace chapter\_4\_3;   
int main()   
{   
 int n;   
 while(scanf("%d",&n) == 1)   
 {   
 if(n == -1)break;   
 for(int i = 0; i < n; i++)list[i].input();   
 Graham(n);   
 for(int i = 0; i < top; i++)p[i] = list[Stack[i]];   
 printf("%.2f\n",(double)rotating\_calipers(p,top)/2);   
 }   
 return 0;   
}   
};   
namespace chapter\_4\_3   
{   
// @chapter 4.3 求解两凸包最小距离   
const double eps = 1e-8;   
int sgn(double x)   
{   
 if(fabs(x) < eps)return 0;   
 if(x < 0)return -1;   
 else return 1;   
}   
struct Point   
{   
 double x,y;   
 Point(double \_x = 0,double \_y = 0)   
 {   
 x = \_x;   
 y = \_y;   
 }   
 Point operator -(const Point &b)const   
 {   
 return Point(x - b.x, y - b.y);   
 }   
 double operator ^(const Point &b)const   
 {   
 return x\*b.y - y\*b.x;   
 }   
 double operator \*(const Point &b)const   
 {   
 return x\*b.x + y\*b.y;   
 }   
 void input()   
 {   
 scanf("%lf%lf",&x,&y);   
 }   
};   
struct Line   
{   
 Point s,e;   
 Line() {}   
 Line(Point \_s,Point \_e)   
 {   
 s = \_s;   
 e = \_e;   
 }   
};   
//两点间距离   
double dist(Point a,Point b)   
{   
 return sqrt((a-b)\*(a-b));   
}   
//点到线段的距离，返回点到线段最近的点   
Point NearestPointToLineSeg(Point P,Line L)   
{   
 Point result;   
 double t = ((P-L.s)\*(L.e-L.s))/((L.e-L.s)\*(L.e-L.s));   
 if(t >=0 && t <= 1)   
 {   
 result.x = L.s.x + (L.e.x - L.s.x)\*t;   
 result.y = L.s.y + (L.e.y - L.s.y)\*t;   
 }   
 else   
 {   
 if(dist(P,L.s) < dist(P,L.e))   
 result = L.s;   
 else result = L.e;   
 }   
 return result;   
}   
/\*   
\* 求凸包，Graham算法   
\* 点的编号0~n-1   
\* 返回凸包结果Stack[0~top-1]为凸包的编号   
\*/const int MAXN = 10010;   
Point list[MAXN];   
int Stack[MAXN],top;   
//相对于list[0]的极角排序   
bool \_cmp(Point p1,Point p2)   
{   
 double tmp = (p1-list[0])^(p2-list[0]);   
 if(sgn(tmp) > 0)return true;   
 else if(sgn(tmp) == 0 && sgn(dist(p1,list[0]) - dist(p2,list[0])) <= 0)   
 return true;   
 else return false;   
}   
void Graham(int n)   
{   
 Point p0;   
 int k = 0;   
 p0 = list[0];   
//找最下边的一个点   
 for(int i = 1; i < n; i++)   
 {   
 if( (p0.y > list[i].y) || (p0.y == list[i].y && p0.x > list[i].x) )   
 {   
 p0 = list[i];   
 k = i;   
 }   
 }   
 swap(list[k],list[0]);   
 sort(list+1,list+n,\_cmp);   
 if(n == 1)   
 {   
 top = 1;   
 Stack[0] = 0;   
 return;   
 }   
 if(n == 2)   
 {   
 top = 2;   
 Stack[0] = 0;   
 Stack[1] = 1;   
 return ;   
 }   
 Stack[0] = 0;   
 Stack[1] = 1;   
 top = 2;   
 for(int i = 2; i < n; i++)   
 {   
 while(top > 1 &&   
 sgn((list[Stack[top-1]]-list[Stack[top-2]])^   
 (list[i]-list[Stack[top-2]])) <=   
 0)   
 top--;   
 Stack[top++] = i;   
 }   
}   
//点p0到线段p1p2的距离   
double pointtoseg(Point p0,Point p1,Point p2)   
{   
 return dist(p0,NearestPointToLineSeg(p0,Line(p1,p2)));   
}//平行线段p0p1和p2p3的距离   
double dispallseg(Point p0,Point p1,Point p2,Point p3)   
{   
 double ans1 = min(pointtoseg(p0,p2,p3),pointtoseg(p1,p2,p3));   
 double ans2 = min(pointtoseg(p2,p0,p1),pointtoseg(p3,p0,p1));   
 return min(ans1,ans2);   
}   
//得到向量a1a2和b1b2的位置关系   
double Get\_angle(Point a1,Point a2,Point b1,Point b2)   
{   
 return (a2-a1)^(b1-b2);   
}   
double rotating\_calipers(Point p[],int np,Point q[],int nq)   
{   
 int sp = 0, sq = 0;   
 for(int i = 0; i < np; i++)   
 if(sgn(p[i].y - p[sp].y) < 0)   
 sp = i;   
 for(int i = 0; i < nq; i++)   
 if(sgn(q[i].y - q[sq].y) > 0)   
 sq = i;   
 double tmp;   
 double ans = dist(p[sp],q[sq]);   
 for(int i = 0; i < np; i++)   
 {   
 while(sgn(tmp = Get\_angle(p[sp],p[(sp+1)%np],q[sq],q[(sq+1)%nq])) < 0)   
 sq = (sq+1)%nq;   
 if(sgn(tmp) == 0)   
 ans = min(ans,dispallseg(p[sp],p[(sp+1)%np],q[sq],q[(sq+1)%nq]));   
 else ans = min(ans,pointtoseg(q[sq],p[sp],p[(sp+1)%np]));   
 sp = (sp+1)%np;   
 }   
 return ans;   
}   
double solve(Point p[],int n,Point q[],int m)   
{   
 return min(rotating\_calipers(p,n,q,m),rotating\_calipers(q,m,p,n));   
}   
Point p[MAXN],q[MAXN];   
int main()   
{   
 int n,m;   
 while(scanf("%d%d",&n,&m) == 2)   
 {   
 if(n == 0 && m == 0)break;   
 for(int i = 0; i < n; i++)   
 list[i].input();   
 Graham(n);   
 for(int i = 0; i < top; i++)   
 p[i] = list[i];   
 n = top;   
 for(int i = 0; i < m; i++)   
 list[i].input();   
 Graham(m);   
 for(int i = 0; i < top; i++)   
 q[i] = list[i];   
 m = top;   
 printf("%.4f\n",solve(p,n,q,m));   
 }   
 return 0;   
}   
};   
namespace chapter\_5\_1   
{   
// @chapter 5.1 半平面交   
const double eps = 1e-8;   
const double PI = acos(-1.0);   
int sgn(double x)   
{   
 if(fabs(x) < eps) return 0;   
 if(x < 0) return -1;   
 else return 1;   
}   
struct point   
{   
 double x,y;   
 point() {}   
 point(double \_x,double \_y)   
 {   
 x = \_x;   
 y = \_y;   
 }   
 point operator -(const point &b)const   
 {   
 return point(x - b.x, y - b.y);   
 }   
 double operator ^(const point &b)const   
 {   
 return x\*b.y - y\*b.x;   
 }   
 double operator \*(const point &b)const   
 {   
 return x\*b.x + y\*b.y;   
 }   
};   
struct Line   
{   
 point s,e;   
 double k;   
 Line() {}   
 Line(point \_s,point \_e)   
 {   
 s = \_s;   
 e = \_e;   
 k = atan2(e.y - s.y,e.x - s.x);   
 }   
 point operator &(const Line &b)const   
 {   
 point res = s;   
 double t = ((s - b.s)^(b.s - b.e))/((s - e)^(b.s - b.e));   
 res.x += (e.x - s.x)\*t;   
 res.y += (e.y - s.y)\*t;   
 return res;   
 }   
};   
//半平面交，直线的左边代表有效区域   
//这个好像和给出点的顺序有关   
bool HPIcmp(Line a,Line b)   
{   
 if(fabs(a.k - b.k) > eps)return a.k < b.k;   
 return ((a.s - b.s)^(b.e - b.s)) < 0;   
}   
Line Q[110];   
//第一个位代表半平面交的直线，第二个参数代表直线条数，第三个参数是相交以后把   
//所得点压栈，第四个参数是栈有多少个元素   
void HPI(Line line[], int n, point res[], int &resn)   
{   
 int tot = n;   
 sort(line,line+n,HPIcmp);   
 tot = 1;   
 for(int i = 1; i < n; i++)   
 if(fabs(line[i].k - line[i-1].k) > eps)   
 line[tot++] = line[i];   
 int head = 0, tail = 1;   
 Q[0] = line[0];   
 Q[1] = line[1];   
 resn = 0;   
 for(int i = 2; i < tot; i++)   
 {   
 if(fabs((Q[tail].e-Q[tail].s)^(Q[tail-1].e-Q[tail-1].s)) < eps ||   
 fabs((Q[head].e-Q[head].s)^(Q[head+1].e-Q[head+1].s)) < eps)   
 return;   
 while(head < tail && (((Q[tail]&Q[tail-1]) -   
 line[i].s)^(line[i].e-line[i].s)) > eps)   
 tail--;   
 while(head < tail && (((Q[head]&Q[head+1]) -   
 line[i].s)^(line[i].e-line[i].s)) > eps)   
 head++;   
 Q[++tail] = line[i];   
 }   
 while(head < tail && (((Q[tail]&Q[tail-1]) -   
 Q[head].s)^(Q[head].e-Q[head].s)) > eps)   
 tail--;   
 while(head < tail && (((Q[head]&Q[head-1]) -   
 Q[tail].s)^(Q[tail].e-Q[tail].e)) > eps)   
 head++;   
 if(tail <= head + 1)return;   
 for(int i = head; i < tail; i++)   
 res[resn++] = Q[i]&Q[i+1];   
 if(head < tail - 1)   
 res[resn++] = Q[head]&Q[tail];   
}   
};   
namespace chapter\_6\_x\_7\_x   
{   
// @chapter 6.1 三点求圆心坐标   
//过三点求圆心坐标   
Point waixin(Point a,Point b,Point c)   
{   
 double a1 = b.x - a.x, b1 = b.y - a.y, c1 = (a1\*a1 + b1\*b1)/2;   
 double a2 = c.x - a.x, b2 = c.y - a.y, c2 = (a2\*a2 + b2\*b2)/2;   
 double d = a1\*b2 - a2\*b1;   
 return Point(a.x + (c1\*b2 - c2\*b1)/d, a.y + (a1\*c2 -a2\*c1)/d);   
}   
// @chapter 7.1 求两圆相交的面积   
//两个圆的公共部分面积   
double Area\_of\_overlap(Point c1,double r1,Point c2,double r2)   
{   
 double d = dist(c1,c2);   
 if(r1 + r2 < d + eps)return 0;   
 if(d < fabs(r1 - r2) + eps)   
 {   
 double r = min(r1,r2);   
 return PI\*r\*r;   
 }   
 double x = (d\*d + r1\*r1 - r2\*r2)/(2\*d);   
 double t1 = acos(x / r1);   
 double t2 = acos((d - x)/r2);   
 return r1\*r1\*t1 + r2\*r2\*t2 - d\*r1\*sin(t1);   
}   
};   
};   
   
   
int main(int argc, char\*\* argv)   
{   
 return 0;   
}