|  |
| --- |
|  |
|  |
| **QUETZAL.H**  **Reference**  **Indice**   |  |  | | --- | --- | | **Arboles** | **3** | | **Estructuras** | **5** | | **Flujos** | **8** | | **Geometría** | **20** | | **Grafos** | **38** | | **Matemáticas** | **48** | | **Strings** | **56** | |  |  |   **Arboles**  // Definiciones iniciales. |
|  |
| typedef vector<int> Lista; |
|  |
| // Arbol con Heavy-Light Decomposition. |
| // Los nodos estan indexados de 0 a n - 1. |
|  |
| struct HeavyLight { |
|  |
| int n, conteo; |
| Lista nivel, tamano, up; |
| Lista indice, super, top; |
| vector<Lista> aristas; |
|  |
| HeavyLight(int N) : n(N), conteo(), |
| top(N), nivel(N), tamano(N), up(N), |
| indice(N), super(N), aristas(N) {} |
|  |
| void AgregarArista(int u, int v) { |
| aristas[u].push\_back(v); |
| aristas[v].push\_back(u); |
| } |
|  |
| void CalcularNivel(int u, int p) { |
| for (int i = 0; i < aristas[u].size(); ++i) { |
| int v = aristas[u][i]; if (p == v) continue; |
| if (super[u] == super[v]) nivel[v] = nivel[u]; |
| else nivel[v] = nivel[u] + 1; |
| CalcularNivel(v, u); |
| } |
| } |
|  |
| // Construir realiza todas las operaciones para |
| // trabajar con Heavy-Light. Por defecto, la raiz del |
| // arbol se establece como el nodo 0. Si quieren definir |
| // una raiz diferente, llamen Construir(r) donde el |
| // parametro r indica cual sera la raiz del arbol. |
|  |
| int Construir(int u = 0, int p = -1) { |
| int tam\_subarbol = 0; |
| up[u] = p, super[u] = -1; |
|  |
| for (int i = 0; i < aristas[u].size(); ++i) { |
| int v = aristas[u][i]; if (p == v) continue; |
| tam\_subarbol += Construir(v, u); |
| } |
| for (int i = 0; i < aristas[u].size(); ++i) { |
| int v = aristas[u][i]; if (p == v) continue; |
| if (tamano[v] > tam\_subarbol / 2) |
| indice[u] = indice[v] + 1, |
| super[u] = super[v], |
| top[super[v]] = u; |
| } |
| if (super[u] == -1) super[u] = conteo, |
| top[conteo++] = u; |
| if (p == -1) CalcularNivel(u, p); |
| return tamano[u] = tam\_subarbol + 1; |
| } |
|  |
| int LCA(int u, int v) { |
| if (nivel[v] > nivel[u]) swap(u, v); |
| while (nivel[u] > nivel[v]) u = up[top[super[u]]]; |
| while (super[u] != super[v]) u = up[top[super[u]]], |
| v = up[top[super[v]]]; |
| return (indice[u] > indice[v])? u: v; |
| } |
| }; |
| |  | | --- | | Estructuras | | const int INF = 1 << 30; | |  | | // Segment Tree version dinamica. Para generar el | | // arbol completo deben llamar a la funcion Construir. | | // CUIDADO: Para usarlo deben especificar el tipo de | | // dato a utilizar; SegTree<int> por ejemplo. | |  | | template<class T> | | struct SegTree { | |  | | T dato; int i, d; | | SegTree\* izq, \*der; | |  | | SegTree(int I, int D) | | : izq(NULL), der(NULL), | | i(I), d(D), dato() {} | |  | | ~SegTree() { | | if (izq) delete izq; | | if (der) delete der; | | } | |  | | T Construir() { | | if (i == d) return dato = T(); | | int m = (i + d) >> 1; | | izq = new SegTree(i, m); | | der = new SegTree(m + 1, d); | | return dato = izq->Construir() + | | der->Construir(); | | } | |  | | T Actualizar(int a, T v) { | | if (a < i || d < a) return dato; | | if (a == i && d == a) return dato = v; | | if (!izq) { | | int m = (i + d) >> 1; | | izq = new SegTree(i, m); | | der = new SegTree(m + 1, d); | | } | | return dato = izq->Actualizar(a, v) + | | der->Actualizar(a, v); | | } | |  | | T Query(int a, int b) { | | if (b < i || d < a) return T(); | | if (a <= i && d <= b) return dato; | | return izq? izq->Query(a, b) + | | der->Query(a, b): T(); | | } | | }; | |  | | // A continuación se ejemplifica como sobrecargar | | // el operador + dentro de una estructura para poder | | // reutilizar el codigo del Segment Tree facilmente. | | // El ejemplo sobrecarga el + por la funcion de maximo. | | // Es MUY IMPORTANTE tener un constructor por defecto. | |  | | struct MaxInt { | | int d; MaxInt(int D) : d(D) {} | | MaxInt() : d(-INF) {} // IMPORTANTE! | | MaxInt operator+(const MaxInt& o) { | | return MaxInt(max(d, o.d)); | | } | | }; | |  | | // Fenwick Tree. Indices de 1 a n. | |  | | struct FenTree { | | vector<int> tree; | | FenTree(int n) : tree(n + 1) {} | |  | | void Actualizar(int i, int v) { | | while (i < tree.size()) { | | tree[i] += v; | | i += i & -i; | | } | | } | |  | | int Query(int i) { | | int sum = 0; | | while (i > 0) { | | sum += tree[i]; | | i -= i & -i; | | } | | return sum; | | } | |  | | int Rango(int i, int j) { | | return Query(j) - | | Query(i - 1); | | } | | }; | |  |   FLUJOS   |  | | --- | |  | | typedef vector<int> Lista; | | typedef pair<int, int> Par; | | typedef vector<Flujo> Flujo1D; | | typedef vector<Flujo1D> Flujo2D; | |  | | const Flujo FINF = 1 << 30; | |  | | // EMPAREJAMIENTO BIPARTITO | | // Nodos indexados de 0 a n - 1. | |  | | struct Bipartito { | |  | | int n; Lista pareja; | | vector<Lista> aristas; | | vector<bool> lado, visitado; | |  | | Bipartito(int N) : lado(N), pareja(N), | | visitado(N), aristas(N), n(N) {} | |  | | void AgregarArista(int u, int v) { | | aristas[u].push\_back(v); | | aristas[v].push\_back(u); | | } | |  | | void AgregarIzq(int u) { lado[u] = true; } | | void AgregarDer(int u) { lado[u] = false; } | |  | | int CaminoIncremental(int u) { | | visitado[u] = true; | | for (int i = 0; i < aristas[u].size(); ++i) | | if (pareja[aristas[u][i]] == -1) | | return pareja[aristas[u][i]] = u; | | for (int i = 0; i < aristas[u].size(); ++i) { | | int v = aristas[u][i]; | | if (visitado[pareja[v]]) continue; | | if (CaminoIncremental(pareja[v]) != -1) | | return pareja[v] = u; | | } | | return -1; | | } | |  | | vector<Par> MaxEmparejamiento() { | | fill(pareja.begin(), pareja.end(), -1); | | for (int i = 0; i < n; ++i) { | | if (!lado[i]) continue; CaminoIncremental(i); | | fill(visitado.begin(), visitado.end(), false); | | } | | vector<Par> pares; | | for (int i = 0; i < n; ++i) | | if (!lado[i] && pareja[i] != -1) | | pares.push\_back(Par(pareja[i], i)); | | return pares; // Cardinalidad = pares.size() | | } | | }; | |  | | // FLUJO MAXIMO | | // Nodos indexados de 0 a n - 1. | |  | | struct GrafoFlujo { | |  | | int n; vector<Lista> aristas; | | Flujo2D cap, flujo; Lista padre, dist; | |  | | GrafoFlujo(int N) : dist(N), padre(N), aristas(N), | | cap(N, Flujo1D(N)), flujo(N, Flujo1D(N)), n(N) {} | |  | | void AgregarArista(int u, int v, Flujo c) { | | flujo[v][u] += c; // Solo dirigidas! | | cap[u][v] += c, cap[v][u] += c; | | aristas[u].push\_back(v); | | aristas[v].push\_back(u); | | } | |  | | // Flujo maximo mediante Edmonds-Karp O(VE^2). | |  | | Flujo ActualizarFlujo(int u, Flujo f) { | | int p = padre[u]; | | if (p == u) return f; | | f = ActualizarFlujo(p, min(f, | | cap[p][u] - flujo[p][u])); | | flujo[p][u] += f; | | flujo[u][p] -= f; | | return f; | | } | |  | | Flujo AumentarFlujo(int s, int t) { | | fill(padre.begin(), padre.end(), -1); | | queue<int> q; q.push(s); padre[s] = s; | | while (!q.empty()) { | | int u = q.front(); | | q.pop(); if (u == t) break; | | for (int i = 0; i < aristas[u].size(); ++i) { | | int v = aristas[u][i]; | | if (flujo[u][v] == cap[u][v] || | | padre[v] != -1) continue; | | padre[v] = u, q.push(v); | | } | | } | | if (padre[t] == -1) return 0; | | return ActualizarFlujo(t, FINF); | | } | |  | | Flujo EdmondsKarp(int s, int t) { | | Flujo flujo\_maximo = 0, f; | | while (f = AumentarFlujo(s, t)) | | flujo\_maximo += f; | | return flujo\_maximo; | | } | |  | | // Flujo maximo mediante Dinic O(V^2E). | |  | | Flujo FlujoBloqueante(int u, int t, Flujo f) { | | if (u == t) return f; Flujo fluido = 0; | | for (int i = 0; i < aristas[u].size(); ++i) { | | if (fluido == f) break; int v = aristas[u][i]; | | if (dist[u] + 1 > dist[v]) continue; | | Flujo fv = FlujoBloqueante(v, t, | | min(f - fluido, cap[u][v] - flujo[u][v])); | | flujo[u][v] += fv, fluido += fv; | | flujo[v][u] -= fv; | | } | | return fluido; | | } | |  | | Flujo Dinic(int s, int t) { | | Flujo flujo\_maximo = dist[t] = 0; | | while (dist[t] < INT\_MAX) { | | fill(dist.begin(), dist.end(), INT\_MAX); | | queue<int> q; q.push(s); dist[s] = 0; | | while (!q.empty()) { | | int u = q.front(); q.pop(); | | for (int i = 0; i < aristas[u].size(); ++i) { | | int v = aristas[u][i]; | | if (flujo[u][v] == cap[u][v] || | | dist[v] <= dist[u] + 1) continue; | | dist[v] = dist[u] + 1, q.push(v); | | } | | } | | if (dist[t] < INT\_MAX) flujo\_maximo += | | FlujoBloqueante(s, t, FINF); | | } | | return flujo\_maximo; | | } | | }; | |  | | // Definiciones adicionales. | |  | | typedef int Costo; // Ajustable. | |  | | typedef vector<Costo> Costo1D; | | typedef vector<Costo1D> Costo2D; | | typedef pair<Costo, int> CostoNodo; | | typedef pair<Flujo, Costo> FlujoCosto; | |  | | const double ERROR = 1e-9; | | const Costo CINF = 1 << 30; | |  | | // Tolerancia en flotantes. | |  | | bool Igual(double a, double b) { | | return fabs(a - b) < ERROR; | | } | |  | | // EMPAREJAMIENTO BIPARTITO DE COSTO MAX/MIN | | // Nodos indexados de 0 a n - 1, diferencia | | // entre nodos en el conjunto izquierdo y derecho. | | // Es posible que alguna variable se desborde y se | | // cicle, para evitarlo cambien Dato a long long. | |  | | struct BipartitoCosto { | |  | | Lista pareja, retorno; vector<bool> visitado; | | int n, s; Costo1D slack, etiqueta; Costo2D costo; | |  | | // Emparejamiento de costo maximo S = 1 | | // Emparejamiento de costo minimo S = -1 | |  | | BipartitoCosto(int N, int S = 1) | | : costo(N, Costo1D(N, S \* -CINF)), s(S), | | slack(2 \* N), etiqueta(2 \* N), pareja(2 \* N), | | retorno(2 \* N), visitado(2 \* N), n(N) {} | |  | | void AgregarArista(int u, int v, | | Costo c) { costo[u][v] = c \* s; } | |  | | vector<Par> EmparejamientoOptimo() { | | fill(pareja.begin(), pareja.end(), -1); | | fill(etiqueta.begin(), etiqueta.end(), 0); | | for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j) | | etiqueta[i] = max(etiqueta[i], costo[i][j]); | |  | | for (int i = 0; i < n; ++i) { | | for (int j = 0; j < n; ++j) | | slack[j + n] = etiqueta[i] + | | etiqueta[j + n] - costo[i][j]; | | fill(visitado.begin(), visitado.end(), false); | | fill(retorno.begin(), retorno.end(), i); | | visitado[i] = true; | |  | | bool emparejado = false; | | for (int j = 0; !emparejado; ++j) { | | int t = n; for (; t < 2 \* n; ++t) { | | if (visitado[t]) continue; | | if (Igual(slack[t], 0)) break; | | } | | if (t < 2 \* n) { | | visitado[t] = true; | | if (pareja[t] == -1) { | | emparejado = true; | | for (int p; ; t = p) { | | pareja[t] = retorno[t]; | | p = pareja[retorno[t]]; | | pareja[retorno[t]] = t; | | if (retorno[t] == i) break; | | } | | } else { | | visitado[t = pareja[t]] = true; | | for (int k = 0; k < n; ++k) { | | Costo new\_slack = etiqueta[t] + | | etiqueta[k + n] - costo[t][k]; | | if (!Igual(new\_slack, slack[k + n]) | | && new\_slack < slack[k + n]) { | | slack[k + n] = new\_slack; | | retorno[k + n] = t; | | } | | } | | } | | } else { | | Costo d = CINF; | | for (int k = n; k < 2 \* n; ++k) | | if (!Igual(slack[k], 0)) | | d = min(d, slack[k]); | | for (int k = 0; k < n; ++k) | | if (visitado[k]) etiqueta[k] -= d; | | for (int k = n; k < 2 \* n; ++k) | | if (!visitado[k]) slack[k] -= d; | | else etiqueta[k] += d; | | } | | } | | } | | vector<Par> pares; | | for (int i = 0; i < n; ++i) | | if (!Igual(costo[i][pareja[i] - n], s \* -CINF)) | | pares.push\_back(Par(i, pareja[i] - n)); | | return pares; // Emparejamiento optimo. | | } | | }; | |  | | // FLUJO MEMORIA OPTIMIZADA Y | | // FLUJO MAXIMO DE COSTO MINIMO | | // Nodos indexados de 0 a n - 1. | | // No utiliza matrices de adyacencia. | |  | | struct GrafoFlujoCosto { | |  | | struct AristaFlujo { | |  | | int dst; AristaFlujo\* residual; | | Flujo cap, flujo; Costo peso, npeso; | |  | | AristaFlujo(int d, Flujo f, Flujo c) | | : dst(d), flujo(f), cap(c) {} | |  | | Costo AumentarFlujo(Flujo f) { | | residual->flujo -= f; | | this->flujo += f; | | return peso \* f; | | } | | }; | |  | | int n; vector<Par> prv; Lista dist; | | vector< vector<AristaFlujo\*> > aristas; | |  | | GrafoFlujoCosto(int N) : n(N), | | aristas(N), prv(N), dist(N) {} | |  | | ~GrafoFlujoCosto() { for (int i = 0; i < n; ++i) | | for (int j = 0; j < aristas[i].size(); ++j) | | delete aristas[i][j]; // NO OMITIR!!! | | } | |  | | // Para aristas bidireccionales agreguen dos aristas | | // dirigidas. Si las aristas no son ponderadas dejen | | // el ultimo parametro con el valor por defecto. | |  | | void AgregarArista(int u, int v, Flujo c, Costo p = 0) { | | AristaFlujo\* uv = new AristaFlujo(v, 0, c); | | AristaFlujo\* vu = new AristaFlujo(u, c, c); | | uv->residual = vu, vu->residual = uv; | | uv->peso = uv->npeso = p; | | vu->peso = vu->npeso = -p; | | aristas[u].push\_back(uv); | | aristas[v].push\_back(vu); | | } | |  | | // Dinic para flujo maximo con memoria optimizada. | | // Prefieran esta version solo cuando n > 5,000. | |  | | Flujo FlujoBloqueante(int u, int t, Flujo f) { | | if (u == t) return f; Flujo fluido = 0; | | for (int i = 0; i < aristas[u].size(); ++i) { | | if (fluido == f) break; | | AristaFlujo\* v = aristas[u][i]; | | if (dist[u] + 1 == dist[v->dst]) { | | Flujo fv = FlujoBloqueante(v->dst, t, | | min(f - fluido, v->cap - v->flujo)); | | v->AumentarFlujo(fv), fluido += fv; | | } | | } | | return fluido; | | } | |  | | Flujo Dinic(int s, int t) { | | Flujo flujo\_maximo = dist[t] = 0; | | while (dist[t] < INT\_MAX) { | | fill(dist.begin(), dist.end(), INT\_MAX); | | queue<int> q; q.push(s); dist[s] = 0; | | while (!q.empty()) { | | int u = q.front(); q.pop(); | | for (int i = 0; i < aristas[u].size(); ++i) { | | AristaFlujo\* v = aristas[u][i]; | | if (dist[v->dst] < INT\_MAX) continue; | | if (v->flujo == v->cap) continue; | | dist[v->dst] = dist[u] + 1; | | q.push(v->dst); | | } | | } | | if (dist[t] < INT\_MAX) flujo\_maximo += | | FlujoBloqueante(s, t, FINF); | | } | | return flujo\_maximo; | | } | |  | | // Flujo de costo minimo en O(VElogV \* flow). Si dejan el | | // valor por defecto del parametro k saca el flujo maximo. | |  | | void RecalcularCosto(const Costo1D& pi) { | | for (int u = 0; u < n; ++u) { | | for (int i = 0; i < aristas[u].size(); ++i) { | | AristaFlujo\* v = aristas[u][i]; | | v->npeso = v->npeso + pi[u] - pi[v->dst]; | | } | | } | | } | |  | | FlujoCosto ActualizarFlujo(int u, Flujo f) { | | int p = prv[u].first, i = prv[u].second; | | if (p == -1) return FlujoCosto(f, 0); | | AristaFlujo\* pu = aristas[p][i]; | |  | | FlujoCosto res = ActualizarFlujo( | | p, min(f, pu->cap - pu->flujo)); | | res.second += pu->AumentarFlujo( | | res.first); return res; | | } | |  | | FlujoCosto AumentarFlujo(int s, int t, Flujo f) { | | Costo1D dist(n, CINF); | | fill(prv.begin(), prv.end(), Par(-1, -1)); | | priority\_queue<CostoNodo, vector<CostoNodo>, | | greater<CostoNodo> > pq; | | pq.push(FlujoCosto(0, s)); dist[s] = 0; | |  | | while (!pq.empty()) { | | int u = pq.top().second; | | Costo p = pq.top().first; pq.pop(); | | if (!Igual(dist[u], p)) continue; | | for (int i = 0; i < aristas[u].size(); ++i) { | | AristaFlujo\* v = aristas[u][i]; | | if (v->flujo == v->cap) continue; | | Costo ndist = dist[u] + v->npeso; | | if (!Igual(ndist, dist[v->dst]) && | | ndist < dist[v->dst]) { | | dist[v->dst] = dist[u] + v->npeso; | | pq.push(CostoNodo(ndist, v->dst)); | | prv[v->dst].second = i; | | prv[v->dst].first = u; | | } | | } | | } | | if (Igual(dist[t], CINF)) | | return FlujoCosto(0, 0); | | RecalcularCosto(dist); | | return ActualizarFlujo(t, f); | | } | |  | | FlujoCosto FlujoCostoMin(int s, int t, Flujo k = FINF) { | | Costo1D dist(n, CINF); dist[s] = 0; | | for (int i = 0; i < n; ++i) { | | for (int u = 0; u < n; ++u) { | | if (Igual(dist[u], CINF)) continue; | | for (int j = 0; j < aristas[u].size(); ++j) { | | AristaFlujo\* v = aristas[u][j]; | | if (v->flujo < v->cap) dist[v->dst] = | | min(dist[v->dst], dist[u] + v->npeso); | | } | | } | | } | | RecalcularCosto(dist); | |  | | FlujoCosto flujo\_costo(0, 0); | | while (flujo\_costo.first < k) { | | FlujoCosto fc = AumentarFlujo( | | s, t, k - flujo\_costo.first); | | flujo\_costo.second += fc.second; | | flujo\_costo.first += fc.first; | | if (!fc.first) break; | | } | | return flujo\_costo; | | } | | };  Geometría   |  | | --- | | // Definiciones iniciales.  typedef long long Long; | |  | | const double ERROR = 1e-9; | | const double M\_2PI = 2 \* M\_PI; | |  | | // Tolerancia en flotantes. | | bool Igual(double a, double b) { | | return fabs(a - b) < ERROR; | | } | |  | | // Punto en 2D. | | struct Punto { | | double x, y; | |  | | Punto() : x(), y() {} | | Punto(double X, double Y) : x(X), y(Y) {} | |  | | // Izquierda a derecha, abajo a arriba. | | bool operator<(const Punto& cmp) const { | | if (!Igual(x, cmp.x)) return x < cmp.x; | | return Igual(y, cmp.y)? false: y < cmp.y; | | } | |  | | bool operator==(const Punto& cmp) const { | | return Igual(x, cmp.x) && Igual(y, cmp.y); | | } | | }; | |  | | // Distancia entre dos puntos p y q. | | double Distancia(const Punto& p, const Punto& q) { | | return hypot(p.x - q.x, p.y - q.y); | | } | |  | | // Magnitud de un vector v. | | double Magnitud(const Punto& v) { | | return hypot(v.x, v.y); | | } | |  | | // Producto punto entre vectores v y w. | | double Dot(const Punto& v, const Punto& w) { | | return v.x \* w.x + v.y \* w.y; | | } | |  | | // Producto cruz entre vectores v y w. | | double Cruz(const Punto& v, const Punto& w) { | | return v.x \* w.y - v.y \* w.x; | | } | |  | | // Conversion de grados a radianes. | | double GradARad(double grd) { | | return (grd \* M\_PI) / 180; | | } | |  | | // Conversion de radianes a grados. | | double RadAGrad(double rad) { | | return (rad \* 180) / M\_PI; | | } | |  | | // Rotar un punto respecto al origen. | | // La rotación se hace en orden CCW, para | | // rotar en CW llamar Rotar(p, M\_2PI - rad). | | Punto Rotar(const Punto& p, double rad) { | | return Punto(p.x\*cos(rad) - p.y\*sin(rad), | | p.x\*sin(rad) + p.y\*cos(rad)); | | } | |  | | // Trasladar p tomando como origen al punto o . | | Punto Trasladar(const Punto& o, const Punto& p) { | | return Punto(p.x - o.x, p.y - o.y); | | } | |  | | // Escalar un vector v por un factor s. | | Punto Escalar(const Punto& v, double s) { | | return Punto(v.x \* s, v.y \* s); | | } | |  | | // Obtener vector opuesto. | | Punto Opuesto(const Punto& v) { | | return Punto(-v.x, -v.y); | | } | |  | | // Angulo entre vectores v y w. | | double Angulo(const Punto& v, const Punto& w) { | | return acos(Dot(v, w) / (Magnitud(v) \* Magnitud(w))); | | } | |  | | // Test de mano derecha: CCW = 1, CW = -1, Colineal = 0. | | int ManoDerecha(const Punto& o, const Punto& p, const Punto& q) { | | double ccw = Cruz(Trasladar(o, p), Trasladar(o, q)); | | return Igual(ccw, 0)? 0: (ccw < 0)? -1: 1; | | } | |  | | // Par de puntos mas cercanos en un conjunto de puntos P. | | pair<Punto, Punto> ParMasCercano(vector<Punto> P) { | | // Si ya esta ordenado, no usar sort. | | sort(P.begin(), P.end()); | |  | | set<Punto> rect; | | pair<Punto, Punto> par; | | int prev = 0; double delta = 1e9; | | for (int i = 0; i < P.size(); ++i) { | | while (P[i].x - P[prev].x > delta) | | rect.erase(Punto(P[prev].y, P[prev++].x)); | |  | | set<Punto>::iterator it = rect.lower\_bound( | | Punto(P[i].y - delta, P[0].x)); | |  | | for (; it != rect.end() && it->x <= P[i].y + delta; ++it) { | | double dist = hypot(P[i].x - it->y, P[i].y - it->x); | | if (dist < delta) delta = dist, par = make\_pair( | | Punto(it->y, it->x), P[i]); | | } | | rect.insert(Punto(P[i].y, P[i].x)); | | } | | return par; // Alternativamente puede devolver delta. | | } | |  | | // Linea en 2D. | | // Si los puntos no aseguran coordenadas | | // enteras usar version double. ¡CUIDADO! | | // Verifiquen los tags <comment> <uncomment> | |  | | struct Linea { | | Punto p, q; | | Long a, b, c; // <comment/> | | //double a, b, c; // <uncomment/> | |  | | Linea() : p(), q(), a(), b(), c() {} | |  | | Linea(Long A, Long B, Long C) | | : p(), q(), a(A), b(B), c(C) { | | if (Igual(a, 0)) { | | c /= -b; b = -1; | | p = Punto(0, c); | | q = Punto(1, c); | | } else if (Igual(b, 0)) { | | c /= -a; a = -1; | | p = Punto(c, 0); | | q = Punto(c, 1); | | } else { | | p = Punto(-c/a, 0); | | q = Punto(-(b+c)/a, 1); | | } if (q < p) swap(p, q); | | } | |  | | Linea(const Punto& P, const Punto& Q) | | : p(P), q(Q), a(), b(), c() { | | // Asegura p como punto menor. | | if (q < p) swap(p, q); | | a = q.y - p.y; | | b = p.x - q.x; | | if (!a) c = p.y, b = -1; | | else if (!b) c = p.x, a = -1; | | else { | | // <comment> | | c = abs(\_\_gcd(a, b)); | | a /= c, b /= c; | | // </comment> | | c = -a\*p.x - b\*p.y; | | } | | } | |  | | // ¡PELIGRO! Ordena por ecuacion de recta. | | bool operator<(const Linea& cmp) const { | | if (!Igual(a, cmp.a)) return a < cmp.a; | | if (!Igual(b, cmp.b)) return b < cmp.b; | | return Igual(c, cmp.c)? false: c < cmp.c; | | } | | }; | |  | | // Saber si un punto p esta en la recta r. | | bool PuntoEnRecta(const Punto& p, const Linea& r) { | | return !ManoDerecha(r.p, r.q, p); | | } | |  | | // Saber si un punto p esta en el segmento s. | | bool PuntoEnSegmento(const Punto& p, const Linea& s) { | | return PuntoEnRecta(p, s) && !(p < s.p || s.q < p); | | } | |  | | // Saber si dos lineas l y m son paralelas. | | bool LineasParalelas(const Linea& l, const Linea& m) { | | return l.a == m.a && l.b == m.b; // <comment/> | | // <uncomment> | | //if (Igual(l.b, 0) || Igual(m.b, 0)) | | // return Igual(l.a, m.a) && Igual(l.b, m.b); | | //return Igual(l.a/l.b, m.a/m.b); | | // </uncomment> | | } | |  | | // Saber si dos lineas l y m son iguales. | | bool LineasIguales(const Linea& l, const Linea& m) { | | return LineasParalelas(l, m) && Igual(l.c, m.c); | | } | |  | | // Saber si dos lineas l y m son perpendiculares. | | bool LineasPerpendiculares(const Linea& l, const Linea& m) { | | return (l.a == m.b && l.b == -m.a) || | | (m.a == l.b && m.b == -l.a); // <comment/> | | // <uncomment> | | //if (Igual(l.b, 0) || Igual(l.a, 0)) | | // return Igual(l.a, m.b) && Igual(l.b, m.a); | | //return Igual(-l.a/l.b, m.b/m.a); | | // </uncomment> | | } | |  | | // Obtener una linea paralela a l que pase por p. | | Linea ParalelaEnPunto(const Linea& l, const Punto& p) { | | return Linea(p, Punto(p.x - l.b, p.y + l.a)); | | } | |  | | // Obtener una linea perpendicular a l que pase por p. | | Linea PerpendicularEnPunto(const Linea& l, const Punto& p) { | | return Linea(p, Punto(p.x + l.a, p.y + l.b)); | | } | |  | | // Saber si dos rectas r y s se intersectan. | | // No intersectan = 0, Interseccion en un punto = 1, | | // Interseccion paralela en infinitos puntos = -1. | | int InterseccionRectas(const Linea& r, const Linea& s) { | | if (LineasIguales(r, s)) return -1; | | return LineasParalelas(r, s)? 0: 1; | | } | |  | | // Saber si una recta r y un segmento s se intersectan. | | // No intersectan = 0, Interseccion en un punto = 1, | | // Interseccion paralela en infinitos puntos = -1. | | int IntersecRectaSegmen(const Linea& r, const Linea& s) { | | if (LineasIguales(r, s)) return -1; | | if (LineasParalelas(r, s)) return 0; | | int t1 = ManoDerecha(r.p, r.q, s.p); | | int t2 = ManoDerecha(r.p, r.q, s.q); | | return (t1 != t2)? 1: 0; | | } | |  | | // Saber si dos segmentos s y t se intersectan. | | // No intersectan = 0, Interseccion en un punto = 1, | | // Interseccion paralela en infinitos puntos = -1. | | int InterseccionSegmentos(const Linea& s, const Linea& t) { | | int t1 = ManoDerecha(s.p, s.q, t.p); | | int t2 = ManoDerecha(s.p, s.q, t.q); | |  | | if (t1 == t2) return t1? 0: | | (PuntoEnSegmento(s.p, t) || | | PuntoEnSegmento(s.q, t) || | | PuntoEnSegmento(t.p, s) || | | PuntoEnSegmento(t.q, s))? -1: 0; | |  | | return (ManoDerecha(t.p, t.q, s.p) != | | ManoDerecha(t.p, t.q, s.q))? 1: 0; | | } | |  | | // Obtener punto de interseccion entre lineas l y m. | | Punto PuntoInterseccion(const Linea& l, const Linea& m) { | | assert(!LineasParalelas(l, m)); // Si son paralelas KABOOM! | | if (Igual(l.a, 0)) return Punto((double)(l.c\*m.b + m.c) / -m.a, l.c); | | double y = (double)(m.a\*l.c - l.a\*m.c) / (m.b\*l.a - m.a\*l.b); | | return Punto((double)(l.c + l.b \* y) / -l.a, y); | | } | |  | | // Obtener proyeccion del vector v en la recta r. | | Punto ProyeccionEnRecta(const Punto& v, const Linea& r) { | | Punto a = Trasladar(r.p, v), b = Trasladar(r.p, r.q); | | return Trasladar(Opuesto(r.p), Escalar(b, | | Dot(a, b) / pow(Magnitud(b), 2))); | | } | |  | | // Distancia entre un punto p y una recta r. | | double DistanciaPuntoRecta(const Punto& p, const Linea& r) { | | return Distancia(ProyeccionEnRecta(p, r), p); | | } | |  | | // Distancia entre un punto p y un segmento s. | | double DistanciaPuntoSegmento(const Punto& p, const Linea& s) { | | Punto proy = ProyeccionEnRecta(p, s); | | if (proy < s.p) return Distancia(s.p, p); | | if (s.q < proy) return Distancia(s.q, p); | | return Distancia(proy, p); | | } | |  | | // Distancia entre dos lineas l y m. | | double DistanciaRectaRecta(const Linea& l, const Linea& m) { | | return LineasParalelas(l, m)? DistanciaPuntoRecta(l.p, m): 0; | | } | |  | | // Distancia entre dos segmentos s y r. | | double DistanciaSegmenSegmen(const Linea& s, const Linea& r) { | | if (InterseccionSegmentos(s, r)) return 0; | | return min(min(DistanciaPuntoSegmento(s.p, r), | | DistanciaPuntoSegmento(s.q, r)), | | min(DistanciaPuntoSegmento(r.p, s), | | DistanciaPuntoSegmento(r.q, s))); | | } | |  | | // Un poligono es una serie de | | // vertices conectados por aristas. | | // P = p1 -> p2 -> p3 -> ... -> pn -> p1. | | typedef vector<Punto> Poligono; | |  | | // Saber si un punto esta en el perimetro de un poligono. | | bool PuntoEnPerimetro(const Punto& p, const Poligono& P) { | | for (int i = 1; i < P.size(); ++i) { | | Punto l = min(P[i - 1], P[i]); | | Punto r = max(P[i - 1], P[i]); | | if (ManoDerecha(l, r, p) == 0 && | | !(p < l || r < p)) return true; | | } | | return false; | | } | |  | | // Prueba de punto en poligono convexo. | | // En el perimetro = -1, Fuera = 0, Dentro = 1. | | int PuntoEnConvexo(const Punto& p, const Poligono& P) { | | if (PuntoEnPerimetro(p, P)) return -1; | | int dir = ManoDerecha(P[0], P[1], p); | | for (int i = 2; i < P.size(); ++i) | | if (ManoDerecha(P[i - 1], P[i], p) | | != dir) return 0; // Fuera. | | return 1; // Dentro. | | } | |  | | // Punto en poligono concavo por ray casting. | | // En el perimetro = -1, Fuera = 0, Dentro = 1. | | int RayCasting(const Punto& p, const Poligono& P) { | | if (PuntoEnPerimetro(p, P)) return -1; | | Punto o = \*min\_element(P.begin(), P.end()); | | Linea rayo(p, Punto(o.x - M\_PI, o.y - M\_E)); | |  | | int cruces = 0; | | for (int i = 1; i < P.size(); ++i) | | if (InterseccionSegmentos(rayo, | | Linea(P[i - 1], P[i]))) ++cruces; | | return cruces & 1; | | } | |  | | // Punto en poligono concavo por angle summation. | | // En el perimetro = -1, Fuera = 0, Dentro = 1. | | int AngleSummation(const Punto& p, const Poligono& P) { | | if (PuntoEnPerimetro(p, P)) return -1; | | double angulo = 0; | | for (int i = 1; i < P.size(); ++i) | | angulo += ManoDerecha(p, P[i - 1], P[i]) \* Angulo( | | Trasladar(p, P[i - 1]), Trasladar(p, P[i])); | | return (fabs(angulo) > M\_PI)? 1: 0; | | } | |  | | // Area de un poligono. | | double Area(const Poligono& P) { | | double area = 0; | | for (int i = 1; i < P.size(); ++i) | | area += Cruz(P[i - 1], P[i]); | | return fabs(area) / 2.0; | | } | |  | | // Perimetro de un poligono. | | double Perimetro(const Poligono& P) { | | double perimetro = 0; | | for (int i = 1; i < P.size(); ++i) | | perimetro += Distancia(P[i - 1], P[i]); | | return perimetro; | | } | |  | | // Cerco convexo de un conjunto de puntos. | | Poligono CercoConvexo(vector<Punto> P){ | | // Si ya esta ordenado, no usar sort. | | sort(P.begin(), P.end()); | | Poligono arriba, abajo; | | for (int i = 0; i < P.size(); ++i) { | | while (arriba.size() > 1) { | | int p = arriba.size() - 1; | | // Permitir colineales: <= | | if (ManoDerecha(arriba[p - 1], | | arriba[p], P[i]) < 0) break; | | arriba.pop\_back(); | | } | | arriba.push\_back(P[i]); | | } | | arriba.pop\_back(); | | for (int i = P.size() - 1; i >= 0; --i) { | | while (abajo.size() > 1) { | | int p = abajo.size() - 1; | | // Permitir colineales: <= | | if (ManoDerecha(abajo[p - 1], | | abajo[p], P[i]) < 0) break; | | abajo.pop\_back(); | | } | | abajo.push\_back(P[i]); | | } | | arriba.insert(arriba.end(), | | abajo.begin(), abajo.end()); | | return arriba; // Convex hull. | | } | |  | | // Centroide de un poligono. | | Punto Centroide(const Poligono& P) { | | double x = 0, y = 0, k = 0; | | for (int i = 1; i < P.size(); ++i) { | | double cruz = Cruz(P[i - 1], P[i]); | | x += cruz \* (P[i - 1].x + P[i].x); | | y += cruz \* (P[i - 1].y + P[i].y); | | k += cruz \* 3; | | } | | return Punto(x/k, y/k); | | } | |  | | // Saber si una recta corta un poligono. | | bool RectaCortaPoligono( | | const Linea& r, const Poligono& P) { | | for (int i = 0, prim = 0; i < P.size(); ++i) { | | int lado = ManoDerecha(r.p, r.q, P[i]); | | if (!lado) continue; if (!prim) prim = lado; | | else if (lado != prim) return true; | | } | | return false; | | } | |  | | // Obtiene los poligonos resultantes de | | // cortar un poligono convexo con una recta. | | vector<Poligono> CortarPoligono( | | const Poligono& P, const Linea& r) { | | if (!RectaCortaPoligono(r, P)) | | return vector<Poligono>(1, P); | | int ind = 0; | | vector<Poligono> Ps(2); | | for (int i = 1; i < P.size(); ++i) { | | Linea s(P[i - 1], P[i]); | | if (IntersecRectaSegmen(r, s)) { | | Punto p = PuntoInterseccion(r, s); | | if (P[i - 1] == p) continue; | | Ps[ind].push\_back(P[i - 1]); | | Ps[1 - ind].push\_back(p); | | Ps[ind].push\_back(p); | | ind = 1 - ind; | | } | | else Ps[ind].push\_back(P[i - 1]); | | } | | Ps[0].push\_back(Ps[0][0]); | | Ps[1].push\_back(Ps[1][0]); | | return Ps; | | } | |  | | // Circulo en 2D. | | struct Circulo { | | Punto c; double r; Circulo() : c(), r() {} | | Circulo(const Punto& C, double R) : c(C), r(R) {} | |  | | bool operator<(const Circulo& cmp) const { | | if (!(c == cmp.c)) return c < cmp.c; | | return Igual(r, cmp.r)? false: r < cmp.r; | | } | | }; | |  | | // Circunferencia de un circulo. | | double Circuferencia(const Circulo& c) { | | return M\_2PI \* c.r; | | } | |  | | // Area de un circulo. | | double Area(const Circulo& c) { | | return M\_PI \* c.r \* c.r; | | } | |  | | // Saber si un punto esta dentro de un circulo. | | // En circunferencia = -1, Fuera = 0, Dentro = 1. | | int PuntoEnCirculo(const Punto& p, const Circulo& c) { | | double dist = Distancia(p, c.c); | | if (Igual(dist, c.r)) return -1; | | return (dist < c.r)? 1: 0; | | } | |  | | // Distancia de un punto p a un circulo c | | double DistanciaPuntoCirculo(const Punto& p, const Circulo& c) { | | double dist = Distancia(p, c.c) - c.r; | | return (dist < 0)? 0: dist; | | } | |  | | // Proyecta un punto fuera de un circulo en su circunferencia. | | Punto ProyPuntoCircunferencia(const Punto& p, const Circulo& c) { | | Punto v = Trasladar(p, c.c); | | double prop = DistanciaPuntoCirculo(p, c) / Magnitud(v); | | return Trasladar(Opuesto(p), Escalar(v, prop)); | | } | |  | | // Obtiene dos puntos que, desde el punto p, forman | | // lineas tangentes a la circunferencia del circulo c. | | Linea ProyTangentes(const Punto& p, const Circulo& c) { | | double a = acos(c.r / Distancia(p, c.c)); | | Punto p\_ = Trasladar(c.c, ProyPuntoCircunferencia(p, c)); | | return Linea(Trasladar(Opuesto(c.c), Rotar(p\_, M\_2PI - a)), | | Trasladar(Opuesto(c.c), Rotar(p\_, a))); | | } | |  | | // Saber si se intersecta un circulo c y una recta r. | | // Tangente = -1, No se intersectan = 0, Cuerda = 1. | | int IntersecCirculoRecta(const Circulo& c, const Linea& r) { | | double dist = DistanciaPuntoRecta(c.c, r); | | if (Igual(dist, c.r)) return -1; | | return (dist < c.r)? 1: 0; | | } | |  | | // Soluciones a un sistema de ecuaciones cuadraticas. | | Punto Chicharronera(double a, double b, double c) { | | double sq = sqrt(b\*b - 4\*a\*c); | | return Punto((-b + sq) / (2\*a), | | (-b - sq) / (2\*a)); | | } | |  | | // Cuerda de interseccion entre un circulo y una recta. | | Linea CuerdaInterseccion(const Circulo& c, const Linea& r) { | | assert(IntersecCirculoRecta(c, r)); // KABOOM! | | Punto p, q; | | if (!Igual(r.b, 0)) { | | Linea R = Linea(Trasladar(c.c, r.p), | | Trasladar(c.c, r.q)); | | p = Chicharronera(R.a\*R.a + R.b\*R.b, 2\*R.a\*R.c, | | R.c\*R.c - R.b\*R.b\*c.r\*c.r); | | q = Punto(p.y, (R.c + R.a\*p.y) / -R.b); | | p.y = (R.c + R.a\*p.x) / -R.b; | | p = Trasladar(Opuesto(c.c), p); | | q = Trasladar(Opuesto(c.c), q); | | } | | else { | | double sq = sqrt(c.r\*c.r - | | pow(r.p.x - c.c.x, 2)); | | p = Punto(r.p.x, c.c.y + sq); | | q = Punto(r.p.x, c.c.y - sq); | | } | | return Linea(p, q); | | } | |  | | // Saber si un circulo c esta dentro de un circulo d. | | bool CirculoEnCirculo(const Circulo& c, const Circulo& d) { | | return Distancia(c.c, d.c) < d.r - c.r; | | } | |  | | // Saber si el circulo c intersecta con el circulo d. | | // Uno dentro del otro = -1, Disjuntos = 0, Intersectan = 1. | | int IntersecCirculoCirculo(const Circulo& c, const Circulo& d) { | | double dist = Distancia(c.c, d.c); | | if (dist < fabs(c.r - d.r)) return -1; | | return (dist > c.r + d.r)? 0: 1; | | } | |  | | // Obtiene tangentes exteriores (las que NO se cruzan) entre dos círculos. | | int TangenteExtCirculoCirculo(const Circulo& a, const Circulo& b, Linea &s, Linea &t) { | | // Circulos identicos. Tangentes infinitas (o ninguna a discrecion) | | if (Igual(a.r, b.r) && a.c == b.c) { | | return 0; // O etiqueta de caso especial. | | } | | // Uno es círculo interior del otro. Comparten una tangente. | | // EL CALCULO PUEDE COPIARSE A TangenteInt SI SE REQUIERE. | | Punto u; | | bool unico = false; | | if (b.r < a.r && Igual(Distancia(a.c, b.c) + b.r, a.r)) { | | u = Trasladar(Opuesto(a.c), Escalar(Trasladar(a.c, b.c), a.r / Distancia(a.c, b.c))); | | unico = true; | | } | | if (a.r < b.r && Igual(Distancia(a.c, b.c) + a.r, b.r)) { | | u = Trasladar(Opuesto(b.c), Escalar(Trasladar(b.c, a.c), b.r / Distancia(a.c, b.c))); | | unico = true; | | } | | if (unico) { | | s = t = PerpendicularEnPunto(Linea(a.c, b.c), u); // Recta de tangencia; un punto es referencia. | | //s = t = Linea(u, u); // Punto de tangencia. | | return 1; | | } | | // Circulo en circulo. No hay tangentes. | | if (CirculoEnCirculo(a, b) || CirculoEnCirculo(b, a)) { | | return 0; | | } | | // Calcular las 2 rectas tangentes. | | Linea proy; | | Punto v; | | if (Igual(a.r, b.r)) { | | proy = Linea(a.c, a.c); | | Linea perp = PerpendicularEnPunto(Linea(a.c, b.c), a.c); | | u = Escalar(Trasladar(perp.q, perp.p), b.r / Distancia(perp.p, perp.q)); | | v = Opuesto(u); | | } else { | | Circulo c(a.c, abs(a.r - b.r)); | | proy = ProyTangentes(b.c, c); | | u = Escalar(Trasladar(a.c, proy.p), b.r / (a.r - b.r)); | | v = Escalar(Trasladar(a.c, proy.q), b.r / (a.r - b.r)); | | } | | s = Linea(Trasladar(Opuesto(proy.p), u), Trasladar(Opuesto(b.c), u)); | | t = Linea(Trasladar(Opuesto(proy.q), v), Trasladar(Opuesto(b.c), v)); | | return 2; | | } | |  | | // Obtiene tangentes interiores (las que SI se cruzan) entre dos círculos. | | int TangenteIntCirculoCirculo(const Circulo& a, const Circulo& b, Linea &s, Linea &t) { | | // Circulos identicos. Tangentes infinitas (o ninguna a discrecion) | | if (Igual(a.r, b.r) && a.c == b.c) { | | return 0; // O etiqueta de caso especial. | | } | | // Uno es círculo interior del otro. Comparten una tangente. | | // CALCULO HECHO EN TangenteInt. Copiar de este si se requiere. | | Punto u; | | // Circulos tangentes. Obtener recta tangente unica. | | if (Igual(Distancia(a.c, b.c), a.r + b.r)) { | | u = Trasladar(Opuesto(b.c), Escalar(Trasladar(b.c, a.c), b.r / Distancia(a.c, b.c))); | | s = t = PerpendicularEnPunto(Linea(a.c, b.c), u); // Recta de tangencia. | | //s = t = Linea(u, u); // Punto de tangencia. | | return 1; | | } | | // Circulos se traslapan. no hay tangentes. | | if (!(a.r + b.r < Distancia(a.c, b.c))) { | | return 0; | | } | | // Obtener 2 rectas tangentes. | | Linea proy; | | Punto v; | | Circulo c(a.c, a.r + b.r); | | proy = ProyTangentes(b.c, c); | | u = Escalar(Trasladar(a.c, proy.p), b.r / (a.r + b.r)); | | v = Escalar(Trasladar(a.c, proy.q), b.r / (a.r + b.r)); | | s = Linea(Trasladar(Opuesto(proy.p), Opuesto(u)), Trasladar(Opuesto(b.c), Opuesto(u))); | | t = Linea(Trasladar(Opuesto(proy.q), Opuesto(v)), Trasladar(Opuesto(b.c), Opuesto(v))); | | return 2; | | } | |  | | vector< Punto > PuntosInterseccionCirculos( const Circulo& c, const Circulo& d ){ | | int mano; | | vector<Punto> ret; | | double angulo = 0.0, dist, X, Y; | | Circulo C = Circulo( Punto( 0, 0 ), c.r ); | | Circulo D = Circulo( Trasladar( c.c, d.c ), d.r ); | | mano = ManoDerecha( Punto(0, 0), Punto( 1, 0 ), D.c ); | |  | | if( mano == 1 ) | | angulo = M\_2PI - Angulo( Punto( 1, 0 ), D.c ); | | else if( mano == -1 ) | | angulo = Angulo( Punto( 1, 0 ), D.c ); | |  | | D.c = Rotar( D.c, angulo ); | | dist = Distancia( D.c, C.c ); | |  | | if( Igual( dist, C.r + D.r ) ){ | | ret.push\_back( Punto( C.r, 0 ) ); | | ret[0] = Rotar( ret[0], M\_2PI - angulo ); | | ret[0] = Trasladar( Punto( -c.c.x, -c.c.y ), ret[0] ); | | } | | else if( dist < (C.r + D.r) && dist > fabs(C.r - D.r)) { | | X = (dist\*dist - D.r\*D.r + C.r\*C.r) / (2\*dist); | | Y = sqrt( C.r\*C.r - X\*X ); | | ret.push\_back( Punto( X, Y ) ); | | ret.push\_back( Punto( X, -Y ) ); | | for( int i = 0; i < 2; i++ ){ | | ret[i] = Rotar( ret[i], M\_2PI - angulo ); | | ret[i] = Trasladar( Punto( -c.c.x, -c.c.y ), ret[i] ); | | } | | } | | return ret; | | } | |  | | |

Grafos

|  |
| --- |
| typedef pair<int, int> Arista; |
| typedef int Costo; |
| const Costo INF = 1 << 30; |
|  |
| // Grafos no ponderados. |
| // Nodos indexados de 0 a n - 1. |
| // Grafo(n, true) -> Bidireccional. |
| // Grafo(n, false) -> Dirigido. |
|  |
| struct Grafo { |
|  |
| int n; bool bi; |
| vector<vector<int>> ady; |
| Grafo(int N, bool B = true) |
| : n(N), bi(B), ady(N) {} |
|  |
| void AgregarArista(int u, int v) { |
| if (bi) ady[v].push\_back(u); |
| ady[u].push\_back(v); |
| } |
|  |
| // Detecta ciclos en un grafo o multigrafo. |
| // Llamar a DetectarCiclos() devuelve un |
| // vector de vectores; cada vector interno es |
| // una lista que representa un ciclo del grafo. |
| // NOTA: Solo detecta un ciclo por componente. |
|  |
| vector<int> ciclo; |
| vector<char> color; |
|  |
| void DetectarCiclo(int u, int p) { |
| int retorno = bi? 0: 1; |
| color[u] = ciclo.empty()? 'G': 'N'; |
| for (int v : ady[u]) { |
| if (v == p && !retorno++) continue; |
| if (ciclo.empty() && color[v] == 'G') { |
| color[v] = 'A', ciclo.push\_back(v); |
| if (u != v) color[u] = 'R', |
| ciclo.push\_back(u); |
| } |
| if (color[v] != 'B') continue; |
|  |
| DetectarCiclo(v, u); |
| if (color[u] == 'G' && color[v] == 'R') |
| color[u] = 'R', ciclo.push\_back(u); |
| } |
| if (color[u] == 'G') color[u] = 'N'; |
| } |
|  |
| vector<vector<int>> DetectarCiclos() { |
| vector<vector<int>> ciclos; |
| color = vector<char>(n, 'B'); |
| for (int u = 0; u < n; ++u) { |
| if (color[u] != 'B') continue; |
| ciclo.clear(); DetectarCiclo(u, n); |
| reverse(ciclo.begin(), ciclo.end()); |
| if (!ciclo.empty()) |
| ciclos.push\_back(ciclo); |
| } |
| return ciclos; |
| } |
|  |
| // Deteccion de puentes y puntos de articulacion |
| // en un grafo o multigrafo bidireccional. Los puentes |
| // quedan guardados en un vector de aristas. Lo puntos |
| // de articulacion son marcados como true o false |
| // respectivamente en un vector de booleanos. |
|  |
| int tiempo; |
| vector<int> label, low; |
| vector<Arista> puentes; // <- Resultado |
| vector<bool> articulacion; // <- Resultado |
|  |
| int PuentesArticulacion(int u, int p) { |
| label[u] = low[u] = ++tiempo; |
|  |
| int hijos = 0, retorno = 0; |
| for (int v : ady[u]) { |
| if (v == p && !retorno++) continue; |
| if (!label[v]) { ++hijos; |
| PuentesArticulacion(v, u); |
| if (label[u] <= low[v]) |
| articulacion[u] = true; |
| if (label[u] < low[v]) |
| puentes.push\_back(Arista(u, v)); |
| low[u] = min(low[u], low[v]); |
| } |
| low[u] = min(low[u], label[v]); |
| } |
| return hijos; |
| } |
|  |
| void PuentesArticulacion() { |
| low = vector<int>(n); |
| label = vector<int>(n); |
| tiempo = 0, puentes.clear(); |
| articulacion = vector<bool>(n); |
| for (int u = 0; u < n; ++u) |
| if (!label[u]) articulacion[u] = |
| PuentesArticulacion(u, n) > 1; |
| } |
|  |
| // Deteccion de componentes fuertemente conexas |
| // en un grafo dirigido. Las componentes quedan |
| // guardadas en un vector de vectores, donde |
| // cada vector interno contiene los nodos |
| // de una componente fuertemente conexa. |
|  |
| vector<vector<int>> scc; // <- Resultado |
| int top; vector<int> pila; |
|  |
| void FuertementeConexo(int u) { |
| label[u] = low[u] = ++tiempo; |
|  |
| pila[++top] = u; |
| for (int v : ady[u]) { |
| if (!label[v]) FuertementeConexo(v); |
| low[u] = min(low[u], low[v]); |
| } |
| if (label[u] == low[u]) { |
| vector<int> componente; |
| while (pila[top] != u) { |
| componente.push\_back(pila[top]); |
| low[pila[top--]] = n + 1; |
| } |
| componente.push\_back(pila[top--]); |
| scc.push\_back(componente); |
| low[u] = n + 1; |
| } |
| } |
|  |
| void FuertementeConexo() { |
| low = vector<int>(n); |
| label = vector<int>(n); |
| tiempo = 0, scc.clear(); |
| top = -1, pila = vector<int>(n); |
| for (int u = 0; u < n; ++u) |
| if (!label[u]) FuertementeConexo(u); |
| } |
|  |
| // Determina si un grafo bidireccional |
| // es bipartito (o bien, es bicoloreable). |
|  |
| bool EsBipartito() { |
| vector<char> color(n, -1); |
| for (int u = 0; u < n; ++u) { |
| if (color[u] >= 0) continue; |
|  |
| color[u] = 0; |
| queue<int> q; q.push(u); |
| while (!q.empty()) { |
| int u = q.front(); q.pop(); |
| for (int v : ady[u]) { |
| if (color[v] < 0) q.push(v), |
| color[v] = 1 - color[u]; |
| if (color[u] == color[v]) |
| return false; |
| } |
| } |
| } |
| return true; |
| } |
|  |
| // REVISIT |
| // Obtiene el orden topologico de los nodos |
| // de un grafo dirigido. Orden ascendente |
| // respecto al numero de dependencias. |
|  |
| vector<bool> vis; |
| vector<int> ordenados; |
|  |
| void OrdenTopologico(int u) { |
| vis[u] = true; |
| for (int v : ady[u]) |
| if (!vis[v]) OrdenTopologico(v); |
| ordenados.push\_back(u); |
| } |
|  |
| void OrdenTopologico() { |
| ordenados.clear(); |
| vis = vector<bool>(n); |
| for (int u = 0; u < n; ++u) |
| if (!vis[u]) OrdenTopologico(u); |
| } |
|  |
| // Busqueda en amplitud desde el nodo s. |
| // Devuelve el vector de distancias a todos |
| // los nodos desde s. Un valor INF indica que |
| // no es posible ir de s al respectivo nodo. |
|  |
| vector<Costo> BFS(int s) { |
| queue<int> q; |
| vector<Costo> d(n, INF); |
| q.push(s), d[s] = 0; |
|  |
| while (!q.empty()) { |
| int u = q.front(); q.pop(); |
| for (int v : ady[u]) |
| if (d[u] + 1 < d[v]) |
| d[v] = d[u] + 1, |
| q.push(v); |
| } |
| return d; |
| } |
| }; |
|  |
| // Conjuntos disjuntos con Union-Find. |
|  |
| struct UnionFind { |
|  |
| int n; vector<int> padre, tam; |
|  |
| UnionFind(int N) : n(N), |
| tam(N, 1), padre(N) { |
| while (--N) padre[N] = N; |
| } |
|  |
| int Raiz(int u) { |
| if (padre[u] == u) return u; |
| return padre[u] = Raiz(padre[u]); |
| } |
|  |
| bool SonConexos(int u, int v) { |
| return Raiz(u) == Raiz(v); |
| } |
|  |
| void Unir(int u, int v) { |
| int Ru = Raiz(u); |
| int Rv = Raiz(v); |
| if (Ru == Rv) return; |
| --n, padre[Ru] = Rv; |
| tam[Rv] += tam[Ru]; |
| } |
|  |
| int Tamano(int u) { |
| return tam[Raiz(u)]; |
| } |
| }; |
|  |
| typedef pair<Costo, int> CostoNodo; |
| typedef pair<Costo, Arista> Ponderada; |
|  |
| // Grafos con ponderacion. |
| // Nodos indexados de 0 a n - 1. |
| // GrafoPonderado(n, true) -> Bidireccional. |
| // GrafoPonderado(n, false) -> Dirigido. |
|  |
| struct GrafoPonderado { |
|  |
| int n; bool bi; |
| vector<vector<CostoNodo>> ady; |
| GrafoPonderado(int N, bool B = true) |
| : n(N), bi(B), ady(N) {} |
|  |
| void AgregarArista(int u, int v, Costo c) { |
| if (bi) ady[v].push\_back(CostoNodo(c, u)); |
| ady[u].push\_back(CostoNodo(c, v)); |
| } |
|  |
| // Obtiene el arbol de expansion minima de un |
| // grafo bidireccional. Para obtener el arbol |
| // de expansion maxima descomentar el reverse. |
| // En caso de tener varias componentes conexas, |
| // obtiene el bosque de expansion minima. |
|  |
| vector<Ponderada> Kruskal() { |
| vector<Ponderada> todas; |
| for (int u = 0; u < n; ++u) |
| for (CostoNodo arista : ady[u]) |
| todas.push\_back( |
| Ponderada(arista.first, |
| Arista(u, arista.second))); |
| sort(todas.begin(), todas.end()); |
| // reverse(todas.begin(), todas.end()); |
|  |
| vector<Ponderada> mst; |
| UnionFind componentes(n); |
| for (Ponderada arista : todas) { |
| int u = arista.second.first; |
| int v = arista.second.second; |
| if (!componentes.SonConexos(u, v)) |
| componentes.Unir(u, v), |
| mst.push\_back(arista); |
| } |
| return mst; |
| } |
|  |
| // Algoritmo de dijkstra desde el nodo s. |
| // Devuelve el vector de distancias a todos |
| // los nodos desde s. Un valor INF indica que |
| // no es posible ir de s al respectivo nodo. |
|  |
| vector<Costo> Dijkstra(int s) { |
| vector<Costo> dist(n, INF); |
| priority\_queue<CostoNodo> pq; |
| pq.push(CostoNodo(0, s)), dist[s] = 0; |
|  |
| while (!pq.empty()) { |
| Costo p = -pq.top().first; |
| int u = pq.top().second; pq.pop(); |
| if (dist[u] < p) continue; |
|  |
| for (CostoNodo arista : ady[u]) { |
| int v = arista.second; |
| p = dist[u] + arista.first; |
| if (p < dist[v]) dist[v] = p, |
| pq.push(CostoNodo(-p, v)); |
| } |
| } |
| return dist; |
| } |
|  |
| // Algoritmo de Bellman-Ford optimizado, desde |
| // el nodo s. Devuelve un booleano indicando si |
| // existe un ciclo negativo en un digrafo. |
| // Obtiene el vector de distancias a todos. |
|  |
| vector<Costo> dist; // <- Resultado |
|  |
| bool BellmanFerrari(int s) { |
| queue<int> q; |
| vector<bool> en\_cola(n); |
| vector<int> procesado(n); |
| dist = vector<Costo>(n, INF); |
| q.push(s), dist[s] = 0; |
|  |
| while (!q.empty()) { |
| int u = q.front(); |
| q.pop(), en\_cola[u] = false; |
| if (++procesado[u] == n) return true; |
| for (CostoNodo arista : ady[u]) { |
| int v = arista.second; |
| Costo p = arista.first; |
| if (dist[u] + p < dist[v]) { |
| if (!en\_cola[v]) q.push(v); |
| dist[v] = dist[u] + p; |
| en\_cola[v] = true; |
| } |
| } |
| } |
| return false; |
| } |
| }; |

Matemáticas

|  |
| --- |
| // Factores primos de un numero a. |
| typedef pair<int, int> Factor; |
|  |
| vector<Factor> FactoresPrimos(int a) { |
| int conteo = 0; |
| vector<Factor> factores; |
| while (!(a & 1)) ++conteo, a >>= 1; |
| if (conteo) factores.push\_back( |
| Factor(2, conteo)), conteo = 0; |
|  |
| int raiz = sqrt(a); |
| for (int i = 3; i <= raiz; i += 2) { |
| while (!(a % i)) ++conteo, a /= i; |
| if (conteo) factores.push\_back( |
| Factor(i, conteo)), conteo = 0; |
| } |
| if (a > 1) factores.push\_back( |
| Factor(a, 1)); |
| return factores; |
| } |
|  |
| // Criba de Eratostenes de 1 a n. |
|  |
| vector<int> Criba(int n) { |
| int raiz = sqrt(n); vector<int> criba(n + 1); |
| for (int i = 4; i <= n; i += 2) criba[i] = 2; |
| for (int i = 3; i <= raiz; i += 2) if (!criba[i]) |
| for (int j = i \* i; j <= n; j += i) |
| if (!criba[j]) criba[j] = i; |
| return criba; |
| } |
|  |
| // Factores primos de n factorial (n!). |
| // El vector de primos debe estar ordenado. |
|  |
| vector<Factor> FactoresFactorial( |
| int n, const vector<int>& primos) { |
|  |
| vector<Factor> factores; |
| for (int i = 0; i < primos.size(); ++i) { |
| if (n < primos[i]) break; int p = primos[i]; |
| int reps = n / p; while (primos[i] <= n / p) |
| p \*= primos[i], reps += n / p; |
| factores.push\_back(Factor(primos[i], reps)); |
| } |
| return factores; |
| } |
|  |
| // Exponenciacion binaria a^n mod m. |
|  |
| typedef long long Long; |
|  |
| Long Exponenciar(Long a, Long n, Long m) { |
| Long resultado = 1; |
| for (; n; n >>= 1) { |
| if (n & 1) resultado = |
| (resultado \* a) % m; |
| a = (a \* a) % m; |
| } |
| return resultado; |
| } |
|  |
| {  // Multiplicacion binaria a\*b mod m. |
|  |
| Long Multiplicar(Long a, Long b, Long m) { |
| Long resultado = 0; |
| for (; b; b >>= 1) { |
| if (b & 1) resultado = |
| (resultado + a) % m; |
| a = (a + a) % m; |
| } |
| return resultado; |
| } |
|  |
| // Algoritmo de Euclides extendido entre a y b. |
| // Además de devolver el gcd(a, b), resuelve la |
| // identidad de Bezout con el par (x, y). Si el |
| // parametro mod no es especificado, se resuelve |
| // con aritmetica normal; si mod se especifica, |
| // la identidad se resuelve modulo mod. |
|  |
| Long Euclides(Long a, Long b, |
| Long& x, Long& y, Long mod = 0) { |
| if (!b) { x = 1, y = 0; return a; } |
| Long gcd = Euclides(b, a % b, x, y, mod); |
|  |
| x = !mod? x - y \* (a / b): (mod + |
| x - (y \* (a / b)) % mod) % mod; |
| swap(x, y); return gcd; |
| } |
|  |
| // Tipo de dato para operar fracciones. |
|  |
| struct Fraccion { |
| Long num, den; |
| Fraccion() : num(0), den(1) {} |
| Fraccion(Long n, Long d) { |
| if (d < 0) n = -n, d = -d; |
| Long gcd = \_\_gcd(abs(n), abs(d)); |
| num = n / gcd, den = d / gcd; |
| } |
|  |
| Fraccion operator-() const { |
| return Fraccion(-num, den); |
| } |
|  |
| Fraccion operator+(const Fraccion& f) { |
| Long gcd = \_\_gcd(den, f.den); |
| return Fraccion( |
| num \* (f.den / gcd) + |
| f.num \* (den / gcd), |
| den \* (f.den / gcd) |
| ); |
| } |
|  |
| Fraccion operator-(const Fraccion& f) { |
| return \*this + -f; // a - b = a + (-b) |
| } |
|  |
| Fraccion operator\*(const Fraccion& f) { |
| return Fraccion(num \* f.num, den \* f.den); |
| } |
|  |
| Fraccion operator/(const Fraccion& f) { |
| return Fraccion(num \* f.den, den \* f.num); |
| } |
|  |
| bool operator<(const Fraccion& cmp) { |
| Long gcd = \_\_gcd(den, cmp.den); |
| return num \* (cmp.den / gcd) < |
| cmp.num \* (den / gcd); |
| } |
|  |
| bool operator==(const Fraccion& cmp) { |
| Long gcd = \_\_gcd(den, cmp.den); |
| return num \* (cmp.den / gcd) == |
| cmp.num \* (den / gcd); |
| } |
| }; |
|  |
| // Eliminacion Gaussiana de matrices. |
| // Definiciones iniciales para Gauss-Jordan. |
|  |
| typedef vector<double> Vector; |
| typedef vector<Vector> Matriz; |
|  |
| // Para eliminacion con fracciones. |
|  |
| Fraccion fabs(const Fraccion& f) { |
| return Fraccion(abs(f.num), f.den); |
| } |
|  |
| bool EsCero(const Fraccion& f) { |
| return f.num == 0; |
| } |
|  |
| // Para eliminacion con flotantes. |
|  |
| const double ERROR = 1e-9; |
|  |
| bool EsCero(double a) { |
| return fabs(a) < ERROR; |
| } |
|  |
| // En caso de no permitir el pivoteo (eg. cuando |
| // requieran sacar una matriz inversa) simplemente |
| // comenten o borren la seccion <comment>. |
|  |
| void EliminacionGaussiana(Matriz& m) { |
| for (int i = 0; i < m.size(); ++i) { |
| // <comment> |
| int fila\_mayor = i; |
| for (int j = i + 1; j < m.size(); ++j) |
| if (fabs(m[fila\_mayor][i]) < |
| fabs(m[j][i])) fila\_mayor = j; |
| swap(m[i], m[fila\_mayor]); |
| // </comment> |
|  |
| if (EsCero(m[i][i])) continue; |
| for (int j = m[i].size() - 1; j >= i; --j) |
| m[i][j] = m[i][j] / m[i][i]; |
| for (int j = 0; j < m.size(); ++j) { |
| if (i == j || EsCero(m[j][i])) continue; |
| for (int k = m[j].size() - 1; k >= i; --k) |
| m[j][k] = m[j][k] - m[i][k] \* m[j][i]; |
| } |
| } |
| } |
|  |
| // Tipo de dato para operar numeros complejos. |
|  |
| struct Complejo { |
| double real, imag; |
| Complejo() : real(), imag() {} |
| Complejo(double r, double i) : real(r), imag(i) {} |
|  |
| Complejo operator+(const Complejo& c) { |
| return Complejo(real + c.real, imag + c.imag); |
| } |
| Complejo operator-(const Complejo& c) { |
| return Complejo(real - c.real, imag - c.imag); |
| } |
| Complejo operator\*(const Complejo& c) { |
| return Complejo(real \* c.real - imag \* c.imag, |
| real \* c.imag + imag \* c.real); |
| } |
| }; |
|  |
| // Transformada rapida de Fourier. |
| // Se tiene que garantizar que el numero de |
| // elementos en el vector sea una potencia de 2. |
|  |
| const double M\_2PI = 2 \* M\_PI; |
|  |
| vector<Complejo> FastAndFourier( |
| const vector<Complejo>& a, int k = 1) { |
|  |
| int n = a.size(); |
| if (n == 1) return a; |
| vector<Complejo> par, impar; |
| for (int i = 0; i < n; ++i) |
| if (i & 1) par.push\_back(a[i]); |
| else impar.push\_back(a[i]); |
| impar = FastAndFourier(impar, k); |
| par = FastAndFourier(par, k); |
|  |
| vector<Complejo> fourier(n); |
| Complejo w(1, 0), wn(cos(-k \* M\_2PI/n), |
| sin(-k \* M\_2PI/n)); |
| for (int i = 0; i < n/2; w = w \* wn, ++i) { |
| fourier[i + n/2] = impar[i] - w \* par[i]; |
| fourier[i] = impar[i] + w \* par[i]; |
| } |
| return fourier; |
| } |
|  |
| // Transformada inversa de Fourier. |
| // Se tiene que garantizar que el numero de |
| // elementos en el vector sea una potencia de 2. |
|  |
| vector<Complejo> InvFastAndFourier( |
| const vector<Complejo>& a) { |
|  |
| vector<Complejo> ifft = FastAndFourier(a, -1); |
| for (int i = 0; i < ifft.size(); ++i) |
| ifft[i].real /= ifft.size(), |
| ifft[i].imag /= ifft.size(); |
| return ifft; |
| } |
|  |
| // Convolucion discreta de dos vectores usando |
| // transformada rapida de Fourier O(n log n). |
| // Multiplica eficientemente dos polinomios. |
|  |
| Vector ConvolucionDiscreta( |
| const Vector& a, const Vector& b) { |
|  |
| int n = a.size() + b.size() - 1; |
| int p = 1; while (p < n) p <<= 1; |
|  |
| vector<Complejo> A(p), B(p), C(p); |
| for (int i = 0; i < a.size(); ++i) |
| A[i] = Complejo(a[i], 0); |
| for (int i = 0; i < b.size(); ++i) |
| B[i] = Complejo(b[i], 0); |
|  |
| A = FastAndFourier(A); |
| B = FastAndFourier(B); |
| for (int i = 0; i < p; ++i) |
| C[i] = A[i] \* B[i]; |
| C = InvFastAndFourier(C); |
|  |
| Vector convolucion(n); |
| for (int i = 0; i < n; ++i) |
| convolucion[i] = C[i].real; |
| return convolucion; |
| } |

Strings

|  |
| --- |
| void BucketSort(vector<int>& sa,const vector<int>& rank, int ranks) { |
|  |
| vector<int> bucket(ranks, 0); |
| vector<int> tmp\_sa(sa.size()); |
| for (int i = 0; i < sa.size(); ++i) |
| ++bucket[rank[sa[i]]]; |
|  |
| for (int i = 0, sum = 0; i < ranks; ++i) { |
| swap(bucket[i], sum); sum += bucket[i]; |
| } for (int i = 0; i < sa.size(); ++i) |
| tmp\_sa[bucket[rank[sa[i]]]++] = sa[i]; |
| swap(sa, tmp\_sa); |
| } |
|  |
| // Recuerden poner '$' al final de la cadena. |
|  |
| vector<int> SuffixArray(const string& str) { |
| int ranks = 255; |
| vector<int> sa(str.size()); |
| vector<int> nrank(str.size()); |
| vector<int> rank(str.size(), 0); |
| vector<int> tmp\_rank(str.size()); |
| for (int i = 0; i < str.size(); ++i) |
| nrank[i] = str[i], sa[i] = i; |
|  |
| for (int p = 0; true; ++p) { |
| BucketSort(sa, nrank, ranks + 1); |
| BucketSort(sa, rank, ranks + 1); |
|  |
| tmp\_rank[0] = ranks = 0; |
| for (int i = 1; i < str.size(); ++i) |
| if (rank[sa[i]] != rank[sa[i - 1]] || |
| nrank[sa[i]] != nrank[sa[i - 1]]) |
| tmp\_rank[i] = ++ranks; |
| else tmp\_rank[i] = ranks; |
|  |
| if (ranks + 1 == str.size()) break; |
|  |
| for (int i = 1; i <= 1 << p; ++i) |
| nrank[str.size() - i] = 0; |
| for (int i = 0; i < str.size(); ++i) { |
| int prv = sa[i] - (1 << p); |
| if (prv >= 0) nrank[prv] = tmp\_rank[i]; |
| rank[sa[i]] = tmp\_rank[i]; |
| } |
| } |
| return sa; |
| } |
|  |
| vector<int> LCP(const string& str, |
| const vector<int>& sa) { |
|  |
| vector<int> lcp(str.size()); |
| vector<int> plcp(str.size()); |
| vector<int> phi(str.size(), -1); |
| for(int i = 1; i < str.size(); ++i) |
| phi[sa[i]] = sa[i - 1]; |
|  |
| int len = 0; |
| for(int i = 0; i < str.size(); ++i) { |
| if (phi[i] == -1) continue; |
| for (; str[phi[i] + len] == |
| str[i + len]; ++len) {} |
| plcp[i] = len; len = max(len-1, 0); |
| } |
| for (int i = 0; i < str.size(); ++i) |
| lcp[i] = plcp[sa[i]]; |
| return lcp; |
| } |