

# 1

## Graficas

### 1.1 Kruskal

**Descripción:** Encuentra un árbol generador de peso mínimo para una gráfica.  
**Complejidad:**  $\mathcal{O}(E \log E)$   
**Dependencias:** Estructuras/Union Find

```
1 struct edge {
2     int u, v, w;
3     bool operator< (const edge &o) const{ return w < o.w; }
4 };
5 vector<edge> edges;
6 int kruskal() {
7     int res = 0;
8     sort(edges.begin(), edges.end());
9     for(auto e : edges) if(join(e.u, e.v))
10         res += e.w; // uv es arista del MST
11     return res;
12 }
```

### 1.2 LCA

**Descripción:** Calcula saltos de potencias de 2 en un árbol y calcula el ancestro común más cercano de dos vértices.  
**Complejidad:**  $\mathcal{O}(n \log n)$  en construcción.  $\mathcal{O}(\log n)$  por query.

```
1 const int MAXN = 1e6, LOG = 20;
2 vector<int> adj[MAXN];
3 int up[LOG][MAXN], dep[MAXN];
4 void dfs(int s, int p = 0) {
5     up[0][s] = (p != s);
6     dep[s] = (p ? dep[p] + 1 : 0);
7     for(auto v : adj[s]) if(v != s) dfs(v, s);
8 }
9 void build() {
10     dfs(1);
11     for(int l = 1; l < LOG; l++) for(int v = 1; v < MAXN; v++)
12         up[l][v] = up[l - 1][up[l - 1][v]];
13 }
14 void jmp(int &u, int v, int d) {
15     for(int l = LOG - 1; l >= 0; l--) if(d & (1 << l))
16         u = up[l][u];
17 }
18 int LCA(int u, int v) {
19     if(dep[u] < dep[v]) swap(u, v);
20     jmp(u, v, dep[u] - dep[v]);
21     if(u == v) return u;
22     for(int l = LOG - 1; l >= 0; l--)
23         if(up[l][u] != up[l][v])
24             u = up[l][u], v = up[l][v];
25     return up[0][u];
26 }
```

### 1.3 Floyd

**Descripción:** Calcula la distancia mínima entre cualesquiera dos vértices de una gráfica dirigida.  
**Complejidad:**  $\mathcal{O}(n^3)$

```
1 const int MAXN = 500, INF = 1e9;
2 int G[MAXN][MAXN], n;
3 void floyd() {
4     for(int k = 0; k < n; k++)
5         for(int i = 0; i < n; i++)
6             for(int j = 0; j < n; j++)
7                 G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
8 }
```

### 1.4 Vertices y Aristas de Corte

**Descripción:** Encuentra vértices de corte y aristas de corte (puentes) en una gráfica.  
**Complejidad:**  $\mathcal{O}(V + E)$

```
1 const int MAXN = 1e5;
2 int low[MAXN], ord[MAXN], tin;
3 vector<int> adj[MAXN];
4 int dfs(int s) {
5     low[s] = ord[s] = ++tin;
6     for(auto v : adj[s]) {
7         if(!low[v]) {
8             dfs(v);
9             if(low[v] > ord[s]) { /* uv es puente */ }
10             if(low[v] >= ord[s]) { /* u es punto de articulacion (o raiz) */}
11             low[s] = min(low[s], low[v]);
12         }
13         else if(ord[v] < ord[s])
14             low[s] = min(low[s], ord[v]);
15     }
16     return low[s];
17 }
```

### 1.5 SCC

**Descripción:** Encuentra componentes fuertemente conexas en una gráfica dirigida (Tarjan).  
**Complejidad:**  $\mathcal{O}(V + E)$

```
1 vi val, comp, sta;
2 int Time, ncomps;
3 template<class G> int dfs(int s, G &g) {
4     int low = val[s] = ++Time, x; sta.push_back(s);
5     for(auto v : g[s]) if(comp[v] < 0)
6         low = min(low, val[v] ? dfs(v, g));
7     if(low == val[s]) {
8         do {
9             x = sta.back(); sta.pop_back();
10             comp[x] = ncomps;
11         } while(x != s);
12         ncomps++;
13     }
14 }
```

## 2 Strings

### 2.1 Z Function

**Descripción:** Calcula  $z[i]$  = máximo  $L$  tal que  $s[i : i + L] = s[0 : L]$  para cada  $i$ . Útil para string matching.

**Complejidad:**  $\mathcal{O}(n)$

```

1 vi z_func(string &s) {
2     int n = s.length(), l = -1, r = -1;
3     vi z(n);
4     for(int i = 1; i < n; i++) {
5         if(i <= r) z[i] = min(z[i - l], r - i + 1);
6         while(i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
7         if(i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
8     }
9     return z;
10 }

```

### 2.2 Manacher

**Descripción:** Calcula  $p[i]$  = máximo  $L$  tal que  $s[i - L : i + L]$  es capicua.

**Complejidad:**  $\mathcal{O}(n)$

```

1 vi manacher(string &s) {
2     // Encuentra palindromos de longitud impar.
3     int n = s.length(), l = -1, r = -1;
4     vi p(n);
5     for(int i = 0; i < n; i++) {
6         if(i <= r) p[i] = min(p[l + r - i], r - i);
7         while(i + p[i] + 1 < n && i - p[i] - 1 >= 0 && s[i + p[i] + 1] == s[i - p[i] - 1]) p[i]++;
8         if(i + p[i] > r) l = i - p[i], r = i + p[i];
9     }
10    return p;
11 }

```

### 2.3 Suffix Array

**Descripción:** Crea  $SA$  tal que  $s[SA[i] : n] \leq s[SA[i + 1] : n]$  para todo  $i$ .

**Complejidad:**  $\mathcal{O}(n \log n)$

```

1 const int MAXN = 5e5 + 10;
2 string s;
3 int SA[MAXN], LCP[MAXN], val[MAXN], cnt[MAXN], n;
4
5 void csort(int l) {

```

```

6     int mx = max(300, n), sum = 0, tSA[MAXN];
7     fill(cnt, cnt + mx, 0);
8     for(int i = 0; i < n; i++)
9         cnt[(SA[i] + l < n) ? val[SA[i] + l] : 0]++;
10    for(int i = 0; i < mx; i++)
11        {int t = cnt[i]; cnt[i] = sum; sum += t;}
12    for(int i = 0; i < n; i++)
13        tSA[cnt[SA[i] + l < n ? val[SA[i] + l] : 0]++] = SA[i];
14    for(int i = 0; i < n; i++)
15        SA[i] = tSA[i];
16 }
17 void buildSA() {
18     n = s.length();
19     int nval[MAXN], rk, l = 1;
20     iota(SA, SA + n, 0);
21     for(int i = 0; i < n; i++)
22         val[SA[i]] = s[i];
23     do {
24         csort(l);
25         csort(0);
26         nval[SA[0]] = rk = 0;
27         for(int i = 1; i < n; i++) nval[SA[i]] =
28             ii{val[SA[i]], val[SA[i] + l]} == ii{val[SA[i - 1]], val[SA[i - 1] + l]} ? rk : ++rk;
29         for(int i = 0; i < n; i++)
30             val[i] = nval[i];
31         l <= 1;
32     } while(val[SA[n - 1]] != n - 1 && l < n);
33 }
34 void buildLCP() { // LCP[i] = LCP(s[SA[i - 1] : n], s[SA[i] : n])
35     int pre[MAXN], PLCP[MAXN], L = 0;
36     pre[SA[0]] = -1;
37     for(int i = 1; i < n; i++)
38         pre[SA[i]] = SA[i - 1];
39     for(int i = 0; i < n; i++) {
40         if(pre[i] == -1) {
41             PLCP[i] = -1;
42             continue;
43         }
44         while(s[i + L] == s[pre[i] + L]) L++;
45         PLCP[i] = L;
46         L = max(0, L - 1);
47     }
48     for(int i = 0; i < n; i++)
49         LCP[i] = PLCP[SA[i]];
50 }

```

### 2.4 Hashing

**Descripción:** Calcula un rolling hash de una string para comparar substrings rápidamente.

**Complejidad:**  $\mathcal{O}(n)$  para construir,  $\mathcal{O}(1)$  por query. Un poco lento en la práctica.

```

1 struct rhash {
2     ll P, Q; // P ~ cantidad de caracteres del alfabeto, Q ~ 10^9
3     vl H, po; // Se pueden usar varios hashes si las colisiones son problema
4     rhash(string &s, ll P, ll Q) : P(P), Q(Q) {
5         int n = s.length(); H.resize(n); po.resize(n);

```

```

6   po[0] = 1; H[0] = s[0];
7   for(int i = 1; i < n; i++) {
8       H[i] = (P*H[i - 1] + s[i])%Q;
9       po[i] = (po[i - 1] * P)%Q;
10  }
11  }
12  ll get(ll l, ll r) { // Hash de s[l, r]
13      if(l == 0) return H[r];
14      ll res = (H[r] - po[r - l + 1]*H[l - 1])%Q;
15      return res >= 0 ? res : res + Q;
16  }
17 };

```

## 3 Geometria

### 3.1 Punto

**Descripción:** Estructura genérica para representar puntos (vectores) en 2D.

```

1 template<class T>
2 struct pt {
3     T x, y;
4     pt(T x = 0, T y = 0) : x(x), y(y) {}
5     bool operator< (pt o) const {return (x < o.x || (x == o.x && y < o.y)); }
6     bool operator== (pt o) const {return (x == o.x && y == o.y);}
7     pt operator+ (pt o) const {return pt(x + o.x, y + o.y);}
8     pt operator- (pt o) const {return pt(x - o.x, y - o.y);}
9     pt operator* (T l) const {return pt(l*x, l*y);}
10    pt operator/ (T l) const {return pt(x/l, y/l);}
11    T dot(pt o) { return x*o.x + y*o.y; }
12    T cross(pt o) { return x*o.y - y*o.x; }
13    T cross(pt a, pt b) { return (a - *this).cross(b - *this); }
14    T normsq(pt o) { return x*x +y*y; }
15    double norm(pt o) { return hypot(x, y); }
16 };

```

### 3.2 Envolverte

**Descripción:** Encuentra la envolvente convexa de un conjunto de puntos. Si un punto está en el interior estricto de un segmento de la frontera no se considera vértice de la envolvente.

**Complejidad:**  $\mathcal{O}(n \log n)$

**Dependencias:** Geometria/Punto

```

1 vector<pt<ll>> convex_hull(vector<pt<ll>> P) {
2     int n = P.size();
3     if(n <= 2) return P;
4     vector<pt<ll>> L, U;
5     sort(P.begin(), P.end());
6     for(int i = 0; i < n; i++) {
7         while(U.size() >= 2 && U[U.size() - 2].cross(U[U.size() - 1], P[i]) >= 0)
8             U.pop_back();
9         while(L.size() >= 2 && L[L.size() - 2].cross(L[L.size() - 1], P[n-i-1]) >= 0)
10             L.pop_back();
11         U.push_back(P[i]), L.push_back(P[n - i - 1]);

```

```

12    }
13    U.insert(U.end(), L.begin() + 1, L.end() - 1);
14    return U;
15 }

```

## 4 Flujo

### 4.1 Push-Relabel

**Descripción:** Highest Label Preflow Push con gap heuristic. Calcula un flujo máximo en una red.

**Complejidad:**  $\mathcal{O}(V^2\sqrt{E})$

```

1 const int MAXN = 5000 + 5;
2 struct Edge {
3     int to, rev;
4     ll flow, cap;
5 };
6 vector<Edge> adj[MAXN];
7 void addEdge(int u, int v, int c, int r = 0) {
8     Edge a = {v, (int)adj[v].size(), 0, c};
9     Edge b = {u, (int)adj[u].size(), 0, r};
10    adj[u].push_back(a);
11    adj[v].push_back(b);
12 }
13 ll ex[MAXN];
14 vi hs[2*MAXN];
15 int H[MAXN], ptr[MAXN], n;
16 void push(Edge &e, ll f) {
17     Edge &back = adj[e.to][e.rev];
18     if(!ex[e.to] && f) hs[H[e.to]].push_back(e.to);
19     e.flow += f; e.cap -= f; ex[e.to] += f;
20     back.flow -= f; back.cap += f; ex[back.to] -= f;
21 }
22 ll maxflow(int s, int t) {
23     H[s] = n; ex[t] = 1;
24     vi cnt(2*n); cnt[0] = n - 1;
25     for(auto e : adj[s]) push(e, e.cap);
26
27     for(int ch = 0;;) {
28         while(hs[ch].empty())
29             if(!ch--) return -ex[s];
30         int u = hs[ch].back(); hs[ch].pop_back();
31         while(ex[u] > 0) {
32             if(ptr[u] == adj[u].size()) {
33                 ptr[u] = 0;
34                 H[u]++;
35                 cnt[ch]--;
36                 cnt[H[u]]++;
37                 if(!cnt[ch] && ch < n) {
38                     for(int v = 1; v <= n; v++) {
39                         if(H[v] > ch && H[v] < n) {
40                             --cnt[H[v]], H[v] = n + 1;
41                         }
42                     }

```

```

43     }
44     ch = H[u];
45 }
46 else {
47     Edge &e = adj[u][ptr[u]];
48     if(e.cap && H[u] == H[e.to] + 1)
49         push(e, min(ex[u], e.cap));
50     else ++ptr[u];
51 }
52 }
53 }
54 }

```

## 4.2 Dinic

**Descripción:** Encuentra un flujo máximo en una red. Incluye scaling para casos malvados anti-Dinic.

**Complejidad:**  $\mathcal{O}(V^2E)$  sin scaling.  $\mathcal{O}(VE \log(\max))$  con scaling.  $\mathcal{O}(E\sqrt{V})$  para matchings. Rápido en la práctica.

```

1 typedef long long ll;
2
3 const int MAXN = 5005, INF = 1e9;
4 const bool scale = false; // Magia.  $\mathcal{O}(VE \log(\max))$ , pero peor constante.
5 int dist[MAXN], ptr[MAXN], src, dst, lim = 1;
6 struct Edge {
7     int to, rev, f, cap;
8 };
9 vector<Edge> G[MAXN];
10 void addEdge(int u, int v, int c, int r = 0) {
11     Edge a = {v, (int)G[v].size(), 0, c};
12     Edge b = {u, (int)G[u].size(), 0, r};
13     G[u].push_back(a);
14     G[v].push_back(b);
15 }
16 bool bfs() {
17     queue<int> q({src});
18     memset(dist, -1, sizeof dist);
19     dist[src] = 0;
20     while(!q.empty() && dist[dst] == -1) {
21         int u = q.front();
22         q.pop();
23         for(auto e : G[u]) {
24             int v = e.to;
25             if(dist[v] == -1 && e.f < e.cap && (!scale || e.cap - e.f >= lim)) {
26                 dist[v] = dist[u] + 1;
27                 q.push(v);
28             }
29         }
30     }
31     return dist[dst] != -1;
32 }
33 int dfs(int u, int f) {
34     if(u == dst || !f) return f;
35     for(int &i = ptr[u]; i < G[u].size(); i++) {
36         Edge &e = G[u][i];

```

```

37         int v = e.to;
38         if(dist[v] != dist[u] + 1) continue;
39         if(int df = dfs(v, min(f, e.cap - e.f))) {
40             e.f += df;
41             G[v][e.rev].f -= df;
42             return df;
43         }
44     }
45     return 0;
46 }
47 long long dinic() {
48     long long mf = 0;
49     for(lim = (scale ? (1 << 30) : 1); lim > 0; lim >>= 1) {
50         while(bfs()) {
51             memset(ptr, 0, sizeof ptr);
52             while(long long pushed = dfs(src, INF))
53                 mf += pushed;
54         }
55     }
56     return mf;
57 }

```

## 5 Estructuras

### 5.1 CHT

**Descripción:** Mantiene una envolvente  $S$  con queries de agregar lineas  $y = mx + b$  y queries para  $\max_{(m,b) \in S} mx + b$ .

**Complejidad:**  $\mathcal{O}(\log n)$  amortizado por query.

```

1 bool Q;
2 struct Line {
3     mutable ll m, b, p;
4     bool operator<(const Line& o) const {
5         return Q ? p < o.p : m < o.m;
6     }
7 };
8 struct LineContainer : multiset<Line> {
9     const ll inf = LLONG_MAX;
10    ll div(ll a, ll b) {
11        return a / b - ((a ^ b) < 0 && a % b);
12    }
13    bool isect(iterator x, iterator y) {
14        if (y == end()) { x->p = inf; return false; }
15        if (x->m == y->m) x->p = x->b > y->b ? inf : -inf;
16        else x->p = div(y->b - x->b, x->m - y->m);
17        return x->p >= y->p;
18    }
19    void add(ll m, ll b) {
20        auto z = insert({m, b, 0}), y = z++, x = y;
21        while (isect(y, z)) z = erase(z);
22        if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
23        while ((y = x) != begin() && (--x)->p >= y->p)
24            isect(x, erase(y));
25    }
26    ll query(ll x) {

```

```

26     assert(!empty());
27     Q = 1; auto l = *lower_bound({0, 0, x}); Q = 0;
28     return l.m * x + l.b;
29 }
30 };

```

## 5.2 Union Find

**Descripción:** Union Find con path compression y union by size. Mantiene representantes de una unión disjunta.

**Complejidad:**  $\mathcal{O}(\alpha(n))$  amortizado.

```

1 const int MAXN = 1e6;
2 int rep[MAXN], sz[MAXN];
3 void init() {
4     fill(rep, rep + MAXN, -1), fill(sz, sz + MAXN, 1);
5 }
6 int find(int u) {
7     return (rep[u] == -1) ? u : rep[u] = find(rep[u]);
8 }
9 bool join(int u, int v) {
10    u = find(u), v = find(v);
11    if(u == v) return false;
12    if(sz[u] < sz[v]) swap(u, v);
13    return sz[u] += sz[v], rep[v] = u, true;
14 }

```

## 5.3 Treap

```

1 const int SEED = 17111999;
2 struct treap {
3     int k, p, cnt;
4     treap *l, *r;
5     treap() {}
6     treap(int k, int p) : k(k), p(p), l(nullptr), r(nullptr){}
7 };
8 typedef treap* ptreap;
9 inline int cnt(ptreap t) { return t ? t->cnt : 0; }
10 inline void upd(ptreap t) { if(t) t->cnt = 1 + cnt(t->l) + cnt(t->r); }
11 void split(ptreap t, ll k, ptreap &l, ptreap &r) {
12     if(!t)
13         l = r = nullptr;
14     else if (k < t->k)
15         split(t->l, k, l, t->l), r = t;
16     else
17         split(t->r, k, t->r, r), l = t;
18     upd(t);
19 }
20 void merge(ptreap &t, ptreap l, ptreap r) {
21     if(!l || !r)
22         t = l ? l : r;
23     else if(l.p > r.p)
24         merge(l->r, l->r, r), t = l;
25     else
26         merge(r->l, l, r->l), t = r;

```

```

27     upd(t);
28 }
29 void insert(ptreap &t, ptreap x) {
30     if(!t)
31         t = x;
32     else if(x->p > t->p)
33         split(t, x->k, x->l, x->r), t = x;
34     else
35         insert(x->k < t->k ? t->l : t->r, x);
36     upd(t);
37 }
38 void erase(ptreap &t, int k) {
39     if(!t)
40         return;
41     if(t->k == k)
42         merge(t, t->l, t->r);
43     else
44         erase(k < t->k ? t->l : t->r, k);
45     upd(t);
46 }

```

## 5.4 Wavelet Tree

**Descripción:** Árbol que mantiene subsucesiones de un arreglo formadas por elementos con valores en ciertos intervalos. Útil para range queries de k-ésimo elemento o número de elementos debajo de cierta cota, etc.

**Complejidad:**  $\mathcal{O}(\log(\max A))$  por query.

```

1 struct wnode {
2     wnode *left, *right;
3     int lo, hi;
4     vector<int> c;
5     wnode(int lo, int hi, int* st, int* en) : lo(lo), hi(hi) {
6         if(hi == lo || st == en)
7             return;
8         int mi = (lo + hi)/2;
9         auto f = [mi](int x) { return x <= mi; };
10        c.push_back(0);
11        for(auto it = st; it != en; ++it)
12            c.push_back(c.back() + f(*it));
13        auto it = stable_partition(st, en, f);
14        left = new wnode(lo, mi, st, it);
15        right = new wnode(mi + 1, hi, it, en);
16    }
17    int kth(int L, int R, int k) {
18        if(lo == hi)
19            return lo;
20        int der = c[R], izq = c[L - 1], tol = der - izq;
21        if(tol >= k)
22            return left->kth(izq + 1, der, k);
23        return right->kth(L - izq, R - der, k - tol);
24    }
25 };

```

## 5.5 Segment Tree

**Descripción:** Segment Tree recursivo para RMQ y similares. Los intervalos son cerrados.

**Complejidad:**  $\mathcal{O}(\log n)$  en updates y queries.

```

1 const int NE = 1e9;
2 struct node {
3     int mn, l, r;
4     node *left, *right;
5     node(int l, int r, int* A) : l(l), r(r) {
6         if(l == r)
7             mn = A[l];
8         else {
9             int mi = (l + r)/2;
10            left = new node(l, mi, A);
11            right = new node(mi + 1, r, A);
12            mn = min(left->mn, right->mn);
13        }
14    }
15    void upd(int p, int v) {
16        if(r < p || p < l)
17            return;
18        if(l == r) {
19            mn = v;
20            return;
21        }
22        left->upd(p, v), right->upd(p, v);
23        mn = min(left->mn, right->mn);
24    }
25    int qry(int rl, int rr) {
26        if(rr < l || r < rl)
27            return NE;
28        if(rl <= l && r <= rr)
29            return mn;
30        return min(left->qry(rl, rr), right->qry(rl, rr));
31    }
32 };
33
34 const int MAXN = 1e5 + 5;
35 int a[MAXN];

```

## 6 Mate

### 6.1 Miller-Rabin

**Descripción:** Determina si un número es primo.

**Complejidad:**  $\mathcal{O}((\log p)^k)$  para algún  $k$  razonable. Rápido en la práctica.

```

1 bool isprime(ll p) {
2     if(p == 1) return false;
3     if(p % 2 == 0) return p == 2 ? true : false;
4     ll d = p - 1;
5     while(d % 2 == 0) d >>= 1ll;
6     for(int its = 0; its < 15; its++) {
7         ll a = (rand() % (p - 1)) + 1, x = d;
8         a = mpow(a, d, p);
9         while(x != p - 1 && a != p - 1 && a != 1) {
10             a = mmul(a, a, p);
11             x *= 2ll;

```

```

12         }
13         if(a != p - 1 && x % 2 == 0) return false;
14     }
15     return true;
16 }

```

### 6.2 NTT

**Descripción:** Dadas sucesiones enteras  $\{a_k\}$  y  $\{b_k\}$ ,  $\text{conv}(a, b)$  calcula una convolucion  $c_k = \sum a_i b_{k-i} \pmod{p}$  para  $p = 2^a b + 1$  un primo con  $2^a$  mayor a la longitud de las sucesiones.

**Complejidad:**  $\mathcal{O}(n \log n)$ .

```

1 typedef vector<ll> vl;
2
3 /* Otras alternativas:
4     P = 5 << 25 + 1, 7 << 26 + 1, ambos con raiz 3. */
5 const ll MOD = (119 << 23) + 1, gen = 3;
6
7 ll mpow(ll b, ll e) {
8     ll res = 1;
9     for(ll k = 1; k <= e; k <= 1) {
10         if(k & e) res = (res * b)%MOD;
11         b = (b * b)%MOD;
12     }
13     return res;
14 }
15
16 void ntt(vl &f, vl &root, vector<int> &rev) {
17     int n = f.size();
18     for(int i = 0; i < n; i++)
19         if(i < rev[i]) swap(f[i], f[rev[i]]);
20     for(int k = 1; k < n; k *= 2) {
21         for(int i = 0; i < n; i += 2*k) {
22             for(int j = 0; j < k; j++) {
23                 ll z = (root[j + k] * f[i + j + k]) % MOD;
24                 f[i + j + k] = (f[i + j] < z ? f[i + j] - z + MOD : f[i + j] - z);
25                 f[i + j] = (f[i + j] + z >= MOD ? f[i + j] + z - MOD : f[i + j] + z);
26             }
27         }
28     }
29 }
30
31 vl conv(vl A, vl B) {
32     int sz = A.size() + B.size() - 1, L = 32 - __builtin_clz(sz), n = 1 << L;
33     vl f(A), g(B), res(n), root(n, 1);
34     f.resize(n), g.resize(n);
35     ll pr = mpow(gen, MOD/n), inv = mpow(n, MOD - 2);
36     for(int i = n/2 + 1; i < n; i++)
37         root[i] = (pr * root[i - 1])%MOD;
38     for(int i = n/2 - 1; i >= 0; i--)
39         root[i] = root[2*i];
40     vector<int> rev(n);
41     for(int i = 1; i < n; i++)
42         rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (L - 1));
43     ntt(f, root, rev), ntt(g, root, rev);
44     for(int i = 0; i < n; i++)

```

```

45     res[i] = (f[i] * g[i])%MOD;
46     reverse(res.begin() + 1, res.end());
47     ntt(res, root, rev);
48     for(int i = 0; i < sz; i++)
49         res[i] = (res[i] * inv) % MOD;
50     return {res.begin(), res.begin() + sz};
51 }

```

## 6.3 Matriz

## 6.4 CRT

**Descripción:** Encuentra  $x$  tal que  $x \equiv x_1 \pmod{m_1}, x \equiv x_2 \pmod{m_2}$ . Regresa  $-1$  si tal  $x$  no existe.

**Complejidad:**  $\mathcal{O}(\log(m_1 + m_2))$ . Rápido en la práctica.

```

1 template<typename T> void euclid(T a, T b, T &x, T &y) {
2     if(!b) {x = 1, y = 0; return;}
3     euclid(b, a % b, y, x);
4     y -= a/b * x;
5 }
6 template<typename T> T crt(T x1, T m1, T x2, T m2) {
7     T d = __gcd(m1, m2), r, s;
8     if(x1 % d != x2 % d)
9         return -1;
10    euclid(m1, m2, r, s);
11    m1 /= d, m2 /= d;
12    T mod = d*m1*m2, a1 = ((m1*r)%mod*x2)%mod, a2 = ((m2*s)%mod*x1)%mod, y = (a1 +
        a2)%mod;
13    return (y >= 0 ? y : y + mod);
14 }

```

## 6.5 FFT

**Descripción:** Dadas sucesiones reales  $\{a_k\}$  y  $\{b_k\}$ ,  $\text{conv}(a, b)$  calcula una convolucion  $c_k = \sum a_i b_{k-i}$ . Es preciso para resultados menores a  $10^{15}$ . En otro caso es mejor usar NTT y CRT o fft-mod.

**Complejidad:**  $\mathcal{O}(n \log n)$ .

```

1
2 const double T_PI = 2.0*acos(-1);
3 typedef complex<double> C;
4
5 C mul(const C &x, const C &y) {
6     double rx = x.real(), ix = x.imag(), ry = y.real(), iy = y.imag();
7     return C(rx*ry - ix*iy, rx*iy + ix*ry);
8 }
9
10 void fft(vector<C> &f, vector<C> &root, vector<int> &rev) {
11     int n = f.size();
12     for(int i = 0; i < n; i++)
13         if(i < rev[i]) swap(f[i], f[rev[i]]);
14     for(int k = 1; k < n; k <= 1) {
15         for(int i = 0; i < n; i += 2*k) {

```

```

16         for(int j = 0; j < k; j++) {
17             C z = mul(root[j + k], f[i + j + k]);
18             f[i + j + k] = f[i + j] - z;
19             f[i + j] = f[i + j] + z;
20         }
21     }
22 }
23 }
24
25 vector<double> conv(vector<double> &A, vector<double> &B) {
26     int sz = A.size() + B.size() - 1, L = 32 - __builtin_clz(sz), n = 1 << L;
27     vector<C> in(n), out(n), root(n);
28     for(int i = 0; i < A.size(); i++) in[i].real(A[i]);
29     for(int i = 0; i < B.size(); i++) in[i].imag(B[i]);
30     for(int i = 0; i < n / 2; i++)
31         root[i + n/2] = polar(1.0, T_PI*(double)i/n);
32     for(int i = n/2 - 1; i >= 0; i--)
33         root[i] = root[i << 1];
34     vector<int> rev(n);
35     for(int i = 1; i < n; i++)
36         rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (L - 1));
37     fft(in, root, rev);
38     for(int i = 0; i < n; i++) {
39         C z = in[i], zc = in[-i & (n - 1)];
40         out[i] = mul(zc, zc) - conj(mul(z, z));
41     }
42     fft(out, root, rev);
43     vector<double> res(sz);
44     for(int i = 0; i < sz; i++) res[i] = out[i].imag()/(4.0*n);
45     return res;
46 }

```

## 7 Varios

### 7.1 LIS

**Descripción:** Encuentra una subsucesión creciente (o estrictamente creciente) de longitud máxima.

**Complejidad:**  $\mathcal{O}(n \log n)$ .

```

1 vl lis(vl a) {
2     int n = a.size(), sz = 0; map<pl, pl> pre;
3     vl dp(n + 1, LLONG_MAX); dp[0] = LLONG_MIN;
4     for(int i = 0; i < n; i++) {
5         auto it = upper_bound(dp.begin(), dp.end(), a[i]); //a[i]-1 para estricta.
6         if(*it == LLONG_MAX) sz++; *it = min(*it, a[i]);
7         int pos = distance(dp.begin(), it); pre[{pos, a[i]}] = {pos - 1, dp[pos - 1]};
8     }
9     vl ans; pl cu = {sz, dp[sz]};
10    do {
11        ans.push_back(cu.second);
12        cu = pre[cu];
13    } while(cu.first);
14    return reverse(ans.begin(), ans.end()), ans;
15 }

```