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`

```
struct edge {
  int u, v, w;
  bool operator< (const edge &o) const{ return w < o.w; }
};
vector<edge> edges;
int kruskal() {
  int res = 0;
  sort(edges.begin(), edges.end());
  for(auto e : edges) if(join(e.u, e.v))
    res += e.w; // uv es arista del MST
  return res;
}
```

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.

```
const int MAXN = 1e6, LOG = 20;
vector<int> adj[MAXN];
int up[LOG][MAXN], dep[MAXN];
void dfs(int s, int p = 0) {
  up[0][s] = (p ?: s);
  dep[s] = (p ? dep[p] + 1 : s);
  for(auto v : adj[s]) if(v != s) dfs(v, s);
}
void build() {
  dfs(1);
  for(int l = 1; l < LOG; l++) for(int v = 1; v < MAXN; v++)
    up[l][v] = up[l - 1][up[l - 1][v]];
}
void jmp(int &u, int v, int d) {
  for(int l = LOG - 1; l >= 0; l--) if(d & (1 << l))
      u = up[l][u];
}
int LCA(int u, int v) {
  if(dep[u] < dep[v]) swap(u, v);
  jmp(u, v, dep[u] - dep[v]);
  if(u == v) return u;
  for(int l = LOG - 1; l >= 0; l--)
    if(up[l][u] != up[l][v])
      u = up[l][u], v = up[l][v];
  return up[0][u];
}
```

1.3 Floyd

Descripción: Calcula la distancia mínima entre cualesquiera dos vértices de una gráfica dirigida.

Complejidad: $\mathcal{O}(n^3)$

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

1.4 Vertices y Aristas de Corte

Descripción: Encuentra vértices de corte y aristas de corte (puentes) en una gráfica.

Complejidad: $\mathcal{O}(V + E)$

```
const int MAXN = 1e5;
int low[MAXN], ord[MAXN], tin;
vector<int> adj[MAXN];
int dfs(int s) {
  low[s] = ord[s] = ++tin;
  for(auto v : adj[s]) {
    if(!low[v]) {
      dfs(v);
      if(low[v] > ord[s]) { /* uv es puente */ }
      if(low[v] >= ord[s]) { /* u es punto de articulacion (o raiz) */}
      low[s] = min(low[s], low[v]);
    }
    else if(ord[v] < ord[s])
      low[s] = min(low[s], ord[v]);
  }
  return low[s];
}
```

1.5 SCC

Descripción: Encuentra componentes fuertemente conexas en una gráfica dirigida (Tarjan).

Complejidad: $\mathcal{O}(V + E)$

```
vi val, comp, sta;
int Time, ncomps;
template<class G> int dfs(int s, G &g) {
  int low = val[s] = ++Time, x; sta.push_back(s);
  for(auto v : g[s]) if(comp[v] < 0)
    low = min(low, val[v] ?: dfs(v, g));
  if(low == val[s]) {
    do {
      x = sta.back(); sta.pop_back();
      comp[x] = ncomps;
    } while(x != s);
    ncomps++;
  }
}
```

```
template<class G> int scc(G &g) {
  int n = g.size();
  comp.assign(n, -1), val.assign(0, -1);
  Time = ncomps = 0;
  for(int i = 0; i < n; i++) if(comp[i] < 0) dfs(i, g);
}
```

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: $\mathcal{O}(n)$

```
vi z_func(string &s) {
  int n = s.length(), l = -1, r = -1;
  vi z(n);
  for(int i = 1; i < n; i++) {
    if(i <= r) z[i] = min(z[i - l], r - i + 1);
    while(i + z[i] < n && s[z[i]] == s[i + z[i]]) z[i]++;
    if(i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
  }
  return z;
}
```

2.2 Manacher

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

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

```
vi manacher(string &s) {
  // Encuentra palindromos de longitud impar.
  int n = s.length(), l = -1, r = -1;
  vi p(n);
  for(int i = 0; i < n; i++) {
    if(i <= r) p[i] = min(p[l + r - i], r - i);
    while(i+p[i]+1 < n && i-p[i]-1 >= 0 && s[i+p[i]+1] == s[i-p[i]-1]) p[i]++;
    if(i + p[i] > r) l = i - p[i], r = i + p[i];
  }
  return p;
}
```

2.3 Suffix Array

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

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

```
const int MAXN = 5e5 + 10;
string s;
int SA[MAXN], LCP[MAXN], val[MAXN], cnt[MAXN], n;

void csort(int l) {
  int mx = max(300, n), sum = 0, tSA[MAXN];
  fill(cnt, cnt + mx, 0);
  for(int i = 0; i < n; i++)
    cnt[(SA[i] + l < n) ? val[SA[i] + l] : 0]++;
  for(int i = 0; i < mx; i++)
    {int t = cnt[i]; cnt[i] = sum; sum += t;}
  for(int i = 0; i < n; i++)
    tSA[cnt[SA[i] + l < n ? val[SA[i] + l] : 0]++] = SA[i];
  for(int i = 0; i < n; i++)
    SA[i] = tSA[i];
}
void buildSA() {
  n = s.length();
  int nval[MAXN], rk, l = 1;
  iota(SA, SA + n, 0);
  for(int i = 0; i < n; i++)
    val[SA[i]] = s[i];
  do {
    csort(l);
    csort(0);
    nval[SA[0]] = rk = 0;
    for(int i = 1; i < n; i++) nval[SA[i]] =
      ii{val[SA[i]],val[SA[i]+l]} == ii{val[SA[i-1]],val[SA[i - 1]+l]} ? rk:++rk;
    for(int i = 0; i < n; i++)
      val[i] = nval[i];
    l <<= 1;
  } while(val[SA[n - 1]] != n - 1 && l < n);
}
void buildLCP() { // LCP[i] = LCP(s[SA[i - 1]:n], s[SA[i]:n])
  int pre[MAXN], PLCP[MAXN], L = 0;
  pre[SA[0]] = -1;
  for(int i = 1; i < n; i++)
    pre[SA[i]] = SA[i - 1];
  for(int i = 0; i < n; i++) {
    if(pre[i] == -1) {
      PLCP[i] = -1;
      continue;
    }
    while(s[i + L] == s[pre[i] + L]) L++;
    PLCP[i] = L;
    L = max(0, L - 1);
  }
  for(int i = 0; i < n; i++)
    LCP[i] = PLCP[SA[i]];
}
```

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.

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

```
    po[0] = 1; H[0] = s[0];
    for(int i = 1; i < n; i++) {
      H[i] = (P*H[i - 1] + s[i])%Q;
      po[i] = (po[i - 1] * P)%Q;
    }
  }
  ll get(ll l, ll r) { // Hash de s[l, r]
    if(l == 0) return H[r];
    ll res = (H[r] - po[r - l + 1]*H[l - 1])%Q;
    return res >= 0 ? res : res + Q;
  }
};
```

2.5 KMP

Descripcion: KMP + Automaton. Encuentra $\pi[i] = \max\{x: s[0:x) = s(i-x:i]\}$ para cada i.

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

```

vi kmp(string s) {
  vi pi(s.size());
  for(int i = 1, j = 0; i < s.size(); i++) {
    while(j && s[i] != s[j]) j = pi[j - 1];
    pi[i] = j += (s[i] == s[j]);
  }
  return pi;
}
void build_aut(string s, vector<vi> &aut) {
  aut.assign(s.size(), vi(26));
  vi pi = kmp(s);
  for(int c = 0; c < 26; c++)
    aut[0][c] = (s[0] == 'a' + c);

  for(int i = 1; i < s.size(); i++) {
    for(int c = 0; c < 26; c++) {
      if(s[i] == 'a' + c) aut[i][c] = i + 1;
      else aut[i][c] = aut[pi[i - 1]][c];
    }
  }
}
```

3 Geometria

3.1 Punto

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

```
template<class T>
struct pt {
  T x, y;
  pt(T x = 0, T y = 0) : x(x), y(y) {}
  bool operator< (pt o) const {return (x < o.x || (x == o.x && y < o.y)); }
  bool operator== (pt o) const {return (x == o.x && y == o.y);}
  pt operator+ (pt o) const {return pt(x + o.x, y + o.y);}
  pt operator- (pt o) const {return pt(x - o.x, y - o.y);}
  pt operator* (T l) const {return pt(l*x, l*y);}
  pt operator/ (T l) const {return pt(x/l, y/l);}
  T dot(pt o) { return x*o.x + y*o.y; }
  T cross(pt o) { return x*o.y - y*o.x; }
  T cross(pt a, pt b) { return (a - *this).cross(b - *this); }
  T normsq(pt o) { return x*x +y*y; }
  double norm(pt o) { return hypot(x, y); }
};
```

3.2 Envolvente

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`

```
vector<pt<ll>> convex_hull(vector<pt<ll>> P) {
  int n = P.size();
  if(n <= 2) return P;
  vector<pt<ll>> L, U;
  sort(P.begin(), P.end());
  for(int i = 0; i < n; i++) {
    while(U.size() >= 2 && U[U.size() - 2].cross(U[U.size() - 1], P[i]) >= 0)
      U.pop_back();
    while(L.size() >= 2 && L[L.size() - 2].cross(L[L.size() - 1], P[n-i-1]) >= 0)
      L.pop_back();
    U.push_back(P[i]), L.push_back(P[n - i - 1]);
  }
  U.insert(U.end(), L.begin() + 1, L.end() - 1);
  return U;
}
```

3.3 PolyDis

```

struct pt {
  ll x, y;
  pt(ll _x = 0, ll _y = 0) : x(_x), y(_y) {}
  pt operator+ (const pt &o) const { return pt(x + o.x, y + o.y); }
  pt operator- (const pt &o) const { return pt(x - o.x, y - o.y); }
  pt operator* (const double &d) const { return pt(d * x, d * y); }
  ll dot(const pt &o) { return x * o.x + y * o.y; }
  ll cross(const pt &o) const { return x * o.y - y * o.x; }
  ll cross(const pt &A, const pt &B) const { return (A - *this).cross(B - *this);
    }
  double norm() { return hypot(x, y); }
  int quad() const {
    if(x >= 0) {
      if(y >= 0) return 1;
      return 4;
    }
    if(y >= 0) return 2;
    return 3;
  }
```

```
  bool operator< (const pt &o) const {
    if(quad() != o.quad()) return quad() < o.quad();
    return cross(o) > 0;
  }
  bool operator== (const pt &o) const { return fabs(x - o.x) + fabs(y - o.y) < 1e
      -9; }
};

ostream& operator<< (ostream& out, pt p) {
  out << "(" << p.x << ",␣" << p.y << ")";
  return out;
}

typedef vector<pt> poly;

poly mink_add(poly A, poly B) {
  int i1 = 0, i2 = 0, n = A.size(), m = B.size();
  for(int i = 1; i < n; i++)
    if(A[i].y < A[i1].y || (A[i].y == A[i1].y && A[i].x < A[i1].x)) i1 = i;
  rotate(A.begin(), A.begin() + i1, A.end());
  for(int i = 1; i < m; i++)
    if(B[i].y < B[i1].y || (B[i].y == B[i1].y && B[i].x < B[i1].x)) i2 = i;
  rotate(B.begin(), B.begin() + i2, B.end());
  vector<pt> edgA, edgB;
  for(int i = 0; i < n; i++)
    edgA.push_back(A[(i + 1) % n] - A[i]);
  for(int i = 0; i < m; i++)
    edgB.push_back(B[(i + 1) % m] - B[i]);
  poly combined;
  merge(edgA.begin(), edgA.end(), edgB.begin(), edgB.end(), back_inserter(combined
      ));
  poly res({A[0] + B[0]});
  for(int i = 0; i < combined.size() - 1; i++)
    res.push_back(res.back() + combined[i]);
  return res;
}

double dis(pt X, pt A, pt B) {
  if(A == B) return (X - A).norm();
  pt v = B - A;
  double lam = (double)(X - A).dot(v)/v.dot(v);
  if(lam > 0 && lam < 1)
    return fabs(X.cross(A, B)/(A - B).norm());
  return min((X - A).norm(), (X - B).norm());
}

bool on_seg(pt X, pt A, pt B) {
  return fabs((A - X).norm() + (B - X).normDoc() - (A - B).norm()) < 1e-9;
}

bool in_poly(pt A, poly Q) {
  int n = Q.size(), inter = 0;
  for(int i = 0; i < n; i++) {
    pt X = Q[i], Y = Q[(i + 1) % n];
    if(on_seg(A, X, Y)) return true;
    inter ^= ((A.y < X.y) - (A.y < Y.y)) * A.cross(X, Y) > 0;
  }
  return inter;
}

double dis(pt X, poly Q) {
  if(in_poly(X, Q)) return 0;
  double res = 1e18;
  int n = Q.size();
  for(int i = 0; i < Q.size(); i++)
    res = min(res, dis(X, Q[i], Q[(i + 1) % n]));
  return res;
}

double dis(poly A, poly B) {
  for(int i = 0; i < B.size(); i++)
    B[i] = pt(0, 0) - B[i];
  poly M = mink_add(A, B);
  return dis(pt(0, 0), M);
}
```

4 Flujo

4.1 Push-Relabel

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

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

```
const int MAXN = 5000 + 5;
struct Edge {
  int to, rev;
  ll flow, cap;
};
vector<Edge> adj[MAXN];
void addEdge(int u, int v, int c, int r = 0) {
  Edge a = {v, (int)adj[v].size(), 0, c};
  Edge b = {u, (int)adj[u].size(), 0, r};
  adj[u].push_back(a);
  adj[v].push_back(b);
}
ll ex[MAXN];
vi hs[2*MAXN];
int H[MAXN], ptr[MAXN], n;
void push(Edge &e, ll f) {
  Edge &back = adj[e.to][e.rev];
  if(!ex[e.to] && f) hs[H[e.to]].push_back(e.to);
  e.flow += f; e.cap -= f; ex[e.to] += f;
  back.flow -= f; back.cap += f; ex[back.to] -= f;
}
ll maxflow(int s, int t) {
  H[s] = n; ex[t] = 1;
  vi cnt(2*n); cnt[0] = n - 1;
  for(auto e : adj[s]) push(e, e.cap);

  for(int ch = 0;;) {
    while(hs[ch].empty())
      if(!ch--) return -ex[s];
    int u = hs[ch].back(); hs[ch].pop_back();
```

```
    while(ex[u] > 0) {
      if(ptr[u] == adj[u].size()) {
        ptr[u] = 0;
        H[u]++;
        cnt[ch]--;
        cnt[H[u]]++;
        if(!cnt[ch] && ch < n) {
          for(int v = 1; v <= n; v++) {
            if(H[v] > ch && H[v] < n) {
              --cnt[H[v]], H[v] = n + 1;
            }
          }
        }
        ch = H[u];
      }
      else {
        Edge &e = adj[u][ptr[u]];
        if(e.cap && H[u] ==  H[e.to] + 1)
          push(e, min(ex[u], e.cap));
        else ++ptr[u];
      }
    }
  }
}
```

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.

```
typedef long long ll;

const int MAXN = 5005, INF = 1e9;
const bool scale = false; // Magia. O(VElog(max)), pero peor constante.
int dist[MAXN], ptr[MAXN], src, dst, lim = 1;
struct Edge {
  int to, rev, f, cap;
};
vector<Edge> G[MAXN];
void addEdge(int u, int v, int c, int r = 0) {
  Edge a = {v, (int)G[v].size(), 0, c};
  Edge b = {u, (int)G[u].size(), 0, r};
  G[u].push_back(a);
  G[v].push_back(b);
}
bool bfs() {
  queue<int> q({src});
  memset(dist, -1, sizeof dist);
  dist[src] = 0;
  while(!q.empty() && dist[dst] == -1) {
    int u = q.front();
    q.pop();
    for(auto e : G[u]) {
      int v = e.to;
      if(dist[v] == -1 && e.f < e.cap && (!scale || e.cap - e.f >= lim)) {
        dist[v] = dist[u] + 1;
        q.push(v);
      }
    }
  }
  return dist[dst] != -1;
}
int dfs(int u, int f) {
  if(u == dst || !f) return f;
  for(int &i = ptr[u]; i < G[u].size(); i++) {
    Edge &e = G[u][i];
    int v = e.to;
    if(dist[v] != dist[u] + 1) continue;
    if(int df = dfs(v, min(f, e.cap - e.f))) {
      e.f += df;
      G[v][e.rev].f -= df;
      return df;
    }
  }
  return 0;
}
long long dinic() {
  long long mf = 0;
  for(lim = (scale ? (1 << 30) : 1); lim > 0; lim >>= 1) {
    while(bfs()) {
      memset(ptr, 0, sizeof ptr);
      while(long long pushed = dfs(src, INF))
        mf += pushed;
    }
  }
  return mf;
}
```

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.

```
bool Q;
struct Line {
  mutable ll m, b, p;
  bool operator<(const Line& o) const {
    return Q ? p < o.p : m < o.m;
  }
};
struct LineContainer : multiset<Line> {
  const ll inf = LLONG_MAX;
  ll div(ll a, ll b) {
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) { x->p = inf; return false; }
```

```
    if (x->m == y->m) x->p = x->b > y->b ? inf : -inf;
    else x->p = div(y->b - x->b, x->m - y->m);
    return x->p >= y->p;
  }
  void add(ll m, ll b) {
    auto z = insert({m, b, 0}), y = z++, x = y;
    while (isect(y, z)) z  = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)-> p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    Q = 1; auto l = *lower_bound({0, 0, x}); Q = 0;
    return l.m * x + l.b;
  }
};
```

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.

```
const int MAXN = 1e6;
int rep[MAXN], sz[MAXN];
void init() {
  fill(rep, rep + MAXN, -1), fill(sz, sz + MAXN, 1);
}
int find(int u) {
  return (rep[u] == -1) ? u : rep[u] = find(rep[u]);
}
bool join(int u, int v) {
  u = find(u), v = find(v);
  if(u == v) return false;
  if(sz[u] < sz[v]) swap(u, v);
  return sz[u] += sz[v], rep[v] = u, true;
}
```

5.3 Treap

```
const int SEED = 17111999;
struct treap {
  int k, p, cnt;
  treap *l, *r;
  treap() {}
  treap(int k, int p) : k(k), p(p), l(nullptr), r(nullptr){}
};
typedef treap* ptreap;
inline int cnt(ptreap t) { return t ? t->cnt : 0; }
inline void upd(ptreap t) { if(t) t->cnt = 1 + cnt(t->l) + cnt(t->r); }
void split(ptreap t, ll k, ptreap &l, ptreap &r) {
  if(!t)
    l = r = nullptr;
  else if (k < t->k)
    split(t->l, k, l, t->l), r = t;
  else
    split(t->r, k, t->r, r), l = t;
  upd(t);
}
void merge(ptreap &t, ptreap l, ptreap r) {
  if(!l || !r)
    t = l ? l : r;
  else if(l.p > r.p)
    merge(l->r, l->r, r), t = l;
  else
    merge(r->l, l, r->l), t = r;
  upd(t);
}
void insert(ptreap &t, ptreap x) {
  if(!t)
    t = x;
  else if(x->p > t->p)
    split(t, x->k, x->l, x->r), t = x;
  else
    insert(x->k < t->k ? t->l : t->r, x);
  upd(t);
}
void erase(ptreap &t, int k) {
  if(!t)
    return;
  if(t->k == k)
    merge(t, t->l, t->r);
  else
    erase(k < t->k ? t->l : t->r, k);
  upd(t);
}
```

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.

```
struct wnode {
  wnode *left, *right;
  int lo, hi;
  vector<int> c;
  wnode(int lo, int hi, int* st, int* en) : lo(lo), hi(hi) {
    if(hi == lo || st == en)
      return;
    int mi = (lo + hi)/2;
    auto f = [mi](int x) { return x <= mi; };
    c.push_back(0);
    for(auto it = st; it != en; ++it)
      c.push_back(c.back() + f(*it));
    auto it = stable_partition(st, en, f);
    left = new wnode(lo, mi, st, it);
    right = new wnode(mi + 1, hi, it, en);
  }
  int kth(int L, int R, int k) {
    if(lo == hi)
```

```
      return lo;
    int der = c[R], izq = c[L - 1], tol = der - izq;
    if(tol >= k)
      return left->kth(izq + 1, der, k);
    return right->kth(L - izq, R - der, k - tol);
  }
};
```

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.

```
const int NE = 1e9;
struct node {
  int mn, l, r;
  node *left, *right;
  node(int l, int r, int* A) : l(l), r(r) {
    if(l == r)
      mn = A[l];
    else {
      int mi = (l + r)/2;
      left = new node(l, mi, A);
      right = new node(mi + 1, r, A);
      mn = min(left->mn, right->mn);
    }
  }
  void upd(int p, int v) {
    if(r < p || p < l)
      return;
    if(l == r) {
      mn = v;
      return;
    }
    left->upd(p, v), right->upd(p, v);
    mn = min(left->mn, right->mn);
  }
  int qry(int rl, int rr) {
    if(rr < l || r < rl)
      return NE;
    if(rl <= l && r <= rr)
      return mn;
    return min(left->qry(rl, rr), right->qry(rl, rr));
  }
};

const int MAXN = 1e5 + 5;
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.

```
bool isprime(ll p) {
  if(p == 1) return false;
  if(p % 2 == 0) return p == 2 ? true : false;
  ll d = p - 1;
  while(d % 2 == 0) d >>= 1ll;
  for(int its = 0; its < 15; its++) {
    ll a = (rand() % (p - 1)) + 1, x = d;
    a = mpow(a, d, p);
    while(x != p - 1 && a != p - 1 && a != 1) {
      a = mmul(a, a, p);
      x *= 2ll;
    }
    if(a != p - 1 && x % 2 == 0) return false;
  }
  return true;
}
```

6.2 NTT

Descripción: Dadas sucesiones enteras $\{a_k\}$ y $\{b_k\}$, `conv`(a,b) calculauna 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: $\mathcal{O}(n \log n)$.

```
typedef vector<ll> vl;

/* Otras alternativas:
   P = 5 << 25 + 1, 7 << 26 + 1, ambos con raiz 3. */
const ll MOD = (119 << 23) + 1, gen = 3;

ll mpow(ll b, ll e) {
  ll res = 1;
  for(ll k = 1; k <= e; k <<= 1) {
    if(k & e) res = (res * b)%MOD;
    b = (b * b)%MOD;
  }
  return res;
}

void ntt(vl &f, vl &root, vector<int> &rev) {
  int n = f.size();
  for(int i = 0; i < n; i++)
    if(i < rev[i]) swap(f[i], f[rev[i]]);
  for(int k = 1; k < n; k *= 2) {
    for(int i = 0; i < n; i += 2*k) {
      for(int j = 0; j < k; j++) {
        ll z = (root[j + k] * f[i + j + k]) % MOD;
        f[i + j + k] = (f[i + j] < z ? f[i + j] - z + MOD : f[i + j] - z);
        f[i + j] = (f[i + j] + z >= MOD ? f[i + j] + z - MOD : f[i + j] + z);
      }
    }
  }
}

vl conv(vl A, vl B) {
  int sz = A.size() + B.size() - 1, L = 32 - __builtin_clz(sz), n = 1 << L;
```

```
  vl f(A), g(B), res(n), root(n, 1);
  f.resize(n), g.resize(n);
  ll pr = mpow(gen, MOD/n), inv = mpow(n, MOD - 2);
  for(int i = n/2 + 1; i < n; i++)
    root[i] = (pr * root[i - 1])%MOD;
  for(int i = n/2 - 1; i >= 0; i--)
    root[i] = root[2*i];
  vector<int> rev(n);
  for(int i = 1; i < n; i++)
    rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (L - 1));
  ntt(f, root, rev), ntt(g, root, rev);
  for(int i = 0; i < n; i++)
    res[i] = (f[i] * g[i])%MOD;
  reverse(res.begin() + 1, res.end());
  ntt(res, root, rev);
  for(int i = 0; i < sz; i++)
    res[i] = (res[i] * inv) % MOD;
  return {res.begin(), res.begin() + sz};
}
```

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.

```
template<typename T> void euclid(T a, T b, T &x, T &y) {
  if(!b) {x = 1, y = 0; return;}
  euclid(b, a % b, y, x);
  y -= a/b * x;
}
template<typename T> T crt(T x1, T m1, T x2, T m2) {
  T d = __gcd(m1, m2), r, s;
  if(x1 % d != x2 % d)
    return -1;
  euclid(m1, m2, r, s);
  m1 /= d, m2 /= d;
  T mod = d*m1*m2, a1 = ((m1*r)%mod*x2)%mod, a2 = ((m2*s)%mod*x1)%mod, y = (a1 +
      a2)%mod;
  return (y >= 0 ? y : y + mod);
}
```

6.5 FFT

Descripción: Dadas sucesiones reales $\{a_k\}$ y $\{b_k\}$, 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 ftt-mod.

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

```

const double T_PI = 2.0*acos(-1);
typedef complex<double> C;

C mul(const C &x, const C &y) {
  double rx = x.real(), ix = x.imag(), ry = y.real(), iy = y.imag();
  return C(rx*ry - ix*iy, rx*iy + ix*ry);
}

void fft(vector<C> &f, vector<C> &root, vector<int> &rev) {
  int n = f.size();
  for(int i = 0; i < n; i++)
    if(i < rev[i]) swap(f[i], f[rev[i]]);
  for(int k = 1; k < n; k <<= 1) {
    for(int i = 0; i < n; i += 2*k) {
      for(int j = 0; j < k; j++) {
        C z = mul(root[j + k], f[i + j + k]);
        f[i + j + k] = f[i + j] - z;
        f[i + j] = f[i + j] + z;
      }
    }
  }
}

vector<double> conv(vector<double> &A, vector<double> &B) {
  int sz = A.size() + B.size() - 1, L = 32 - __builtin_clz(sz), n = 1 << L;
  vector<C> in(n), out(n), root(n);
  for(int i = 0; i < A.size(); i++) in[i].real(A[i]);
  for(int i = 0; i < B.size(); i++) in[i].imag(B[i]);
  for(int i = 0; i < n / 2; i++)
    root[i + n/2] = polar(1.0, T_PI*(double)i/n);
  for(int i = n/2 - 1; i >= 0; i--)
    root[i] = root[i << 1];
  vector<int> rev(n);
  for(int i = 1; i < n; i++)
    rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (L - 1));
  fft(in, root, rev);
  for(int i = 0; i < n; i++) {
    C z = in[i], zc = in[-i & (n - 1)];
    out[i] = mul(zc, zc) - conj(mul(z, z));
  }
  fft(out, root, rev);
  vector<double> res(sz);
  for(int i = 0; i < sz; i++) res[i] = out[i].imag()/(4.0*n);
  return res;
}
```

7 Varios

7.1 LIS

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

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

```
vl lis(vl a) {
  int n = a.size(), sz = 0; map<pl, pl> pre;
  vl dp(n + 1, LLONG_MAX); dp[0] = LLONG_MIN;
```

```
  for(int i = 0; i < n; i++) {
    auto it = upper_bound(dp.begin(), dp.end(), a[i]); //a[i]-1 para estricta.
    if(*it == LLONG_MAX) sz++; *it = min(*it, a[i]);
    int pos = distance(dp.begin(), it); pre[{pos, a[i]}] = {pos - 1, dp[pos - 1]};
  }
  vl ans; pl cu = {sz, dp[sz]};
  do {
    ans.push_back(cu.second);
    cu = pre[cu];
  } while(cu.first);
  return reverse(ans.begin(), ans.end()), ans;
}
```

7.2 Sliding Window

Descripción: Encuentra los minimos de todos los subarreglos de longitud k.

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

```

vi mins(vi &v, int k) {
  int n = v.size();
  assert(k <= n);
  deque<ii> window;
  vi result;
  for(int i = 0, j = 0; i + k <= n; i++) {
    while(j < i + k) {
      while(window.size() && v[j] <= window.back().first)
        window.pop_back();
      window.push_back({v[j], j});
      j++;
    }
    while(window.size() && window.front().second < i)
      window.pop_front();
    result.push_back(window.front().first);
  }
  return result;
}
```