# Handbook

September 18, 2023

# Contents

# 1 Grafos

## 1.1 Dinic

Insertar utilidad del algoritmo:

C++:

```cpp
struct edge{
  int x, y, flow;
};

int ans;
vector<edge> edges;
vector<vector<int> > grafo;
vector<int> sn;

void addEdge(int x, int y, int flow){
  grafo[x].PB(edges.size());
  edges.PB({x, y, flow});

  grafo[y].PB(edges.size());
  edges.PB({y, x, 0});
}

int bfs(int &ori, int &target){
  int x = ori, y, flow;

  FOR(i, 0, target + 1) sn[i] = INF;

  sn[x] = 0;
  deque<int> q(1, x);

  while(!q.empty()){
    x = q.F(); q.P_F();

    for(auto &e: grafo[x]){
      auto &edge = edges[e];
      y = edge.y;
      flow = edge.flow;

      if(flow <= 0) continue;
      if(sn[y] != INF) continue;
      sn[y] = sn[x] + 1;
      q.PB(y);
    }
  }

  return sn[target];
}

int dfs(int ori, int &target, int min_flow){
  int flow = INF, y;

  for(auto &e_id: grafo[ori]){
    auto &e = edges[e_id];
    y = e.y;
```

```

    if(sn[y] != 1 + sn[ori]) continue;
    if(e.flow <= 0) continue;

    if(y == target){
      flow = min(min_flow, e.flow);
      ans += flow;
      edges[e_id].flow -= flow;
      edges[e_id^1].flow += flow;
      return flow;
    }

    flow = dfs(y, target, min(min_flow, e.flow));

    if(flow != INF){
      edges[e_id].flow -= flow;
      edges[e_id^1].flow += flow;
      return flow;
    }
  }

  return flow;
}
```

# 2 Strings

## 2.1 Función Z

Insertar utilidad del algoritmo:

C++:

```
vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]);
        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 KMP

Insertar utilidad del algoritmo:

C++:

```
vector<int> z;

void kmp(string &s){
  int j, n = s.size();
  z.resize(n);

  FOR(i, 1, n){
    j = z[i - 1];
    while(j > 0 and s[i] != s[j]) j = z[j - 1];
    if(s[i] == s[j]) j++;
    z[i] = j;
  }
}
```

## 2.3 Suffix array

Devuelve un arreglo con el orden lexicográfico de los sufijos de un string S

C++:

```
vector<int> p, c;
void count_sort(vector<int> &p, vector<int> &c){
    int n = p.size();
    vector<int> cnt(n), p_new(n), pos(n);
    pos[0] = 0;
    for(auto x : c) cnt[x]++;
```

```
    for(int i = 1 ; i < n ; ++i) pos[i] = pos[i - 1] + cnt[i - 1];
    for(auto x : p){
        int i = c[x];
        p_new[pos[i]] = x;
        pos[i]++;
    }
    p = p_new;
}
vector<int> suffix_array(string &s){
    s+=" ";
    int n = s.size();
    p.resize(n);
    c.resize(n);
    vector<pair<char, int>> a(n);
    for(int i = 0 ; i < n ; ++i) a[i] = {s[i], i};
    sort(a.begin(), a.end());
    for(int i = 0 ; i < n ; ++i) p[i] = a[i].second;
    c[p[0]] = 0;
    for(int i = 1 ; i < n ; ++i)
      c[p[i]] = a[i].first == a[i - 1].first ? c[p[i - 1]] : c[p[i - 1]] + 1;
    int k = 0, shift;
    while( (1<<k) < n ){
        shift = 1<<k;
        for(int i = 0 ; i<n ; ++i)
            p[i] = (p[i] - (1<<k) +  n) % n;
        count_sort(p,c);
        vector<int> c_new(n);
        c_new[p[0]] = 0;
        for(int i = 1 ; i < n ; ++i){
            pair<int, int> prev = {c[p[i - 1]], c[ (p[i - 1] + shift) % n]};
            pair<int, int> now = {c[p[i]], c[(p[i] + shift) % n]};
            if(prev == now) c_new[p[i]] = c_new[p[i - 1]];
            else c_new[p[i]] = c_new[p[i - 1]] + 1;
        }
        c = c_new;
        k++;
    }
    return p;
}
```

Java:

```
static int[]p, c;
public static class Suffix implements Comparable<Suffix> {
    int index, r, next;
    public Suffix(int index, int rank, int next){
        this.index = index; this.r = rank; this.next = next;
        }
    public int compareTo(Suffix s){
        return r != s.r ? r - s.r : (next != s.next ? next - s.next : index - s.index);
    }
}
public static int[] sort(int[] p, int[]c){
    int N = p.length;
    int[]cnt = new int[N], pos = new int[N], p_new = new int[N];;
    for(int e : c)  cnt[e]++;
        for(int i = 1 ; i < N ; ++i) pos[i] = pos[i - 1] + cnt[i - 1];
        for(int x : p){
```

```
            p_new[pos[c[x]]] = x; pos[c[x]]++;
        }
        p = p_new;
        return p;
    }
public static int[] suffixArray(String s) {
    s+="$";
    int n = s.length();
    c = new int[n];
    p = new int[n];
    Suffix[] su = new Suffix[n];
    for (int i = 0; i < n; ++i) su[i] = new Suffix(i, s.charAt(i), 0);
    Arrays.sort(su);
    for(int i = 0 ; i < n ; ++i) p[i] = su[i].index;
    c[p[0]] = 0;
    for(int i = 1 ; i < n ; ++i) c[p[i]] = su[i].r == su[i - 1].r ?c[ p[i-1]] :c[p[i-1]] + 1;
    int k = 0, shift;
    while((1<<k) < n){
        shift = (1<<k);
        for(int i = 0 ; i < n ; ++i) p[i] = (p[i] - shift + n ) % n;
        p = sort(p, c);
        int[] c_new = new int[n];
        c_new[p[0]] = 0;
        for(int i = 1 ; i < n ; ++i)
            c_new[p[i]] = (c[p[i]] == c[p[i-1]] && c[(p[i]+shift) % n ] == c[(p[i - 1] + shift
                        ? c_new[p[i - 1]] : c_new[p[i - 1]] + 1;
        c = c_new;
        ++k;
    }
    return p;
}
```

## 2.4 Longest Common Prefix on Suffixs

Devuelve un arreglo que contiene el largo del prefijo común máximo entre 2 sufijos i e i+1

C++:

```
vector<int> lcp(vector<int> &p, vector<int> &c, string &s){
    int n = p.size();
    vector<int> lcp(n);
    int k = 0;
    for(int i = 0 ; i < n - 1 ; ++i){
        int pi = c[i];
        int j = p[pi - 1];
        while(s[i + k] == s[j + k]) k++;
        lcp[pi] = k;
        k = max(k - 1, 0);
    }
    return lcp;
}
```

Java:

```
static int[]p, c, LCP;
```

```java

static int[] lcp(int[] p, int[]c, String s){
    int n = p.length;
    LCP = new int[n];
    int k = 0;
    for(int i = 0 ; i < n - 1 ; ++i){
        int pi = c[i];
        int j = p[pi - 1];
        while(s.charAt(i + k) == s.charAt(j + k)) k++;
        LCP[pi] = k;
        k = Math.max(k - 1, 0);
    }
    return LCP;
}
```

## 2.5 Aho Corasick

C++:

```cpp
int trie[MAX][26], nds = 1;
int fin[MAX], fail[MAX], sure_fail[MAX];

int add(string &s){
  int cr = 0, x;

  for(const auto &c: s){
    x = c - 'a';

    if(trie[cr][x] == 0) trie[cr][x] = nds++;
    cr = trie[cr][x];
  }

  fin[cr] = 1;
  return cr;
}

void build(){
  int x, cr = 0;

  deque<int> q;
  q.PB(cr);

  while(!q.empty()){
    cr = q.F(); q.P_F();

    FOR(i, 0, 26){
      x = trie[cr][i];
      if(x) q.PB(x);

      if(cr == 0) continue;
      if(x == 0){
        trie[cr][i] = trie[fail[cr]][i];
        continue;
      }

      fail[x] = trie[fail[cr]][i];
```

```
        sure_fail[x] = fin[fail[x]] ? fail[x] : sure_fail[fail[x]];
      }
    }
}
```

# 3 Búsqueda

## 3.1 Ternary Search

Insertar utilidad del algoritmo:

C++:

```cpp
#define ld long double

ld ternary_search(ld l, ld r) {
    ld eps = 1e-9;
    ld m1, m2, f1, f2;
    while (r - l > eps) {
        m1 = l + (r - l) / 3;
        m2 = r - (r - l) / 3;
        f1 = f(m1);       //evaluates the function at m1
        f2 = f(m2);       //evaluates the function at m2
        if (f1 < f2) l = m1;
        else r = m2;
    }

    //return the maximum of f(x) in [l, r]
    return f(l);
}
```

# 4 Geometría

## 4.1 Convex Hull

Insertar utilidad del algoritmo:

C++:

```cpp
struct pt {
  double x, y;
};

int orientation(pt a, pt b, pt c) {
  double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
  if (v < 0) return -1; // clockwise
  if (v > 0) return +1; // counter-clockwise
  return 0;
}

bool cw(pt a, pt b, pt c, bool include_collinear) {
  int o = orientation(a, b, c);
  return o < 0 || (include_collinear && o == 0);
}
bool ccw(pt a, pt b, pt c, bool include_collinear) {
  int o = orientation(a, b, c);
  return o > 0 || (include_collinear && o == 0);
}

void convex_hull(vector<pt>& a, bool include_collinear = false) {
  if (a.size() == 1)
    return;

  sort(a.begin(), a.end(), [](pt a, pt b) {
    return make_pair(a.x, a.y) < make_pair(b.x, b.y);
  });
  pt p1 = a[0], p2 = a.back();
  vector<pt> up, down;
  up.push_back(p1);
  down.push_back(p1);
  for (int i = 1; i < (int)a.size(); i++) {
    if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
      while (up.size() >= 2){
        if(cw(up[up.size()-2], up[up.size()-1], a[i], include_collinear)) break;
        up.pop_back();
      }
      up.push_back(a[i]);
    }
    if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
      while (down.size() >= 2){
        if(ccw(down[down.size()-2], down[down.size()-1], a[i], include_collinear)) break;
        down.pop_back();
      }
      down.push_back(a[i]);
    }
  }

  if (include_collinear && up.size() == a.size()) {
```

```
    reverse(a.begin(), a.end());
    return;
  }
  a.clear();
  for (int i = 0; i < (int)up.size(); i++)
    a.push_back(up[i]);
  for (int i = down.size() - 2; i > 0; i--)
    a.push_back(down[i]);
}
```

## 4.2 Interseccion de lineas

C++:

```
struct line {
    double a, b, c;
};

const double EPS = 1e-9;

double det(double a, double b, double c, double d) {
    return a*d - b*c;
}

bool intersect(line m, line n, pt & res) {
    double zn = det(m.a, m.b, n.a, n.b);
    if (abs(zn) < EPS)
        return false;
    res.x = -det(m.c, m.b, n.c, n.b) / zn;
    res.y = -det(m.a, m.c, n.a, n.c) / zn;
    return true;
}

bool parallel(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS;
}

bool equivalent(line m, line n) {
    return abs(det(m.a, m.b, n.a, n.b)) < EPS
        && abs(det(m.a, m.c, n.a, n.c)) < EPS
        && abs(det(m.b, m.c, n.b, n.c)) < EPS;
}
```

## 4.3 Punto en polígono

C++:

```
const ld EPSILON = 0.000001;

struct pt{
  ld x, y;
};

int orientation(pt &a, pt &b, pt &c){
  ld A, B, C;
```

```
  A = -(b.y - a.y);
  B = b.x - a.x;
  C = b.y*a.x - a.y*b.x;

  ld result = A*c.x + B*c.y + C;
  if(result > 0.0) return 1; // Clockwise;
  if(result < 0.0) return -1; // Counter-clockwise;
  return 0; // Collinear.
}

bool coord_in_bounds(ld x, ld y, ld a){
  ld mini, maxi;
  mini = min(x, y);
  maxi = max(x, y);

  return (a + EPSILON >= mini and a - EPSILON <= maxi);
}

bool point_in_segment(pt &a, pt &b, pt &c){
  ld A, B, C;
  A = -(b.y - a.y);
  B = b.x - a.x;
  C = b.y*a.x - a.y*b.x;

  ld result = A*c.x + B*c.y + C;
  if(fabs(result) < EPSILON){
    if(coord_in_bounds(a.x, b.x, c.x) and coord_in_bounds(a.y, b.y, c.y)) return true;
  }

  return false;
}

bool lines_intersect(pt &a, pt &b, pt &c, pt &d){
  int o1, o2, o3, o4;
  o1 = orientation(a, b, c);
  o2 = orientation(a, b, d);
  o3 = orientation(c, d, a);
  o4 = orientation(c, d, b);

  if(o1 != o2 and o3 != o4) return true;

  if(o1 == 0 and point_in_segment(a, b, c)) return true;
  if(o2 == 0 and point_in_segment(a, b, d)) return true;
  if(o3 == 0 and point_in_segment(c, d, a)) return true;
  if(o4 == 0 and point_in_segment(c, d, b)) return true;

  return false;
}

bool point_in_polygon(pt &P, vector<pt> &pts){
  pt aux = pt{INF, P.y};
  int intersections = 0, duplicatedIntersections = 0;

  FOR(i, 0, pts.size()){
    pt &p1 = pts[i];
    pt &p2 = pts[(i + 1)%pts.size()];

```

```
    // Projected line pass across one point.
    if(fabs(P.y - p1.y) < EPSILON and P.x - EPSILON < p1.x) duplicatedIntersections++;

    intersections += lines_intersect(P, aux, p1, p2);
  }

  return (intersections - duplicatedIntersections) & 1;
}
```

### 4.4 Ordenamiento por ángulo polar

C++:

```
struct pt {
  ll x, y;
};

pt Ori = pt{0, 0};

// Vector Oa -> Ob
ll cross(pt a, pt b, pt O){
  return a.x*(b.y-O.y)+b.x*(O.y-a.y)+O.x*(a.y-b.y);
}

int orientation(pt a, pt b, pt O) {
  ll v = cross(a, b, O);
  if (v < 0) return -1; // clockwise
  if (v > 0) return +1; // counter-clockwise
  return 0;
}

bool firstHalf(pt a){
  int o = orientation(pt{1, 0}, a, pt{0, 0});
  if(o == 0) o = a.x > 0 ? 1 : -1;
  return o > 0;
}

bool comp(pt &a, pt &b){
  if(firstHalf(a) == firstHalf(b)) return cross(a, b, Ori) > 0;
  return firstHalf(a);
}
```

### 4.5 Área de polígono

C++:

```
ld area(const vector<pt>& fig) {
    ld res = 0;
    for (unsigned i = 0; i < fig.size(); i++) {
        pt p = i ? fig[i - 1] : fig.back();
        pt q = fig[i];
        res += (p.x - q.x) * (p.y + q.y);
    }
    return fabs(res) / 2;
}
```

# 5 Matemáticas

## 5.1 Factorial modulo m

Permite calcular $n! \mod m$

C++:

```
vector<ll> factorial(ll N, ll p) {
  vector<ll> fac(N + 1);
    fac[0] = 1;
  for (int i = 1; i <= N; i++)
    fac[i] = fac[i - 1] * i % p;
  return fac;
}
```

## 5.2 Exponenciacion binaria

Permite calcular $c \equiv a^b \pmod m$

C++:

```
long long binpow(long long a, long long b, long long m) {
    a %= m;
    long long res = 1;
    while (b > 0) {
        if (b & 1)
            res = res * a % m;
        a = a * a % m;
        b >>= 1;
    }
    return res;
}
```

## 5.3 Inverso Modular

Permite calcular $a^{-1} \mod m$, este número satisface $a \cdot a^{-1} \equiv 1 \pmod m$

Con el pequeño teorema de Fermat, siempre que m sea primo, se calcula $x \equiv a^{m-2} \pmod m$, siendo x su inverso modular.

Con el algoritmo de Euclides extendido, siempre y cuando $\gcd(a, m) = 1$, se calculan $x, y$ tal que $ax + my = 1$, por lo que $ax \equiv 1 \pmod m$, siendo x el inverso modular

C++:

```
//Usando binpow
ll inv(ll a, ll mod){
  ll n = mod - 2;
  ll ans = binpow(a, n, mod);
  return ans;
```

```
}
//Usando euclides extendido
ll inv(ll a, ll b) {
  pair<ll,ll> x = extend_euclid(a, b);
  ll ans = x.first + (x.first < 0) * b;
  return ans;
}
```

## 5.4 Inverso modular del factorial modulo m

Permite calcular $i!^{-1} \mod m$ para todo $1 \leq i \leq N$

C++:

```
vector<ll> factorial(ll N, ll p) {
  vector<ll> fac(N + 1);
    fac[0] = 1;
  for (int i = 1; i <= N; i++)
    fac[i] = fac[i - 1] * i % p;
  return fac;
}
```

## 5.5 Coeficientes binomiales modulo m

Calculo de $\binom{n}{k} \mod m$ de múltiples formas

### 5.5.1 nCk $\mod m$ si m es primo

Para $m \geq 10^9$, se puede emplear la fórmula recursiva

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \mod m$$

O la formula explicita mediante factoriales

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \mod m = n!\,k!^{-1}(n-k)!^{-1} \mod m$$

C++:

```
/** Computa nCk mod p usando DP */
ll binomial(int n, int k, ll p) {
  vector<vector<ll>> dp(n + 1, vector<ll> (k + 1, 0));
  for (int i = 0; i <= n; i++) {
    dp[i][0] = 1;
    if (i <= k)
      dp[i][i] = 1;
  }
  for (int i = 0; i <= n; i++)
    for (int j = 1; j <= min(i, k); j++)
      if (i != j)
```

```
        dp[i][j] = (dp[i - 1][j - 1] + dp[i - 1][j]) % p;
    /** Puede retornarse el arreglo completo
    con la respuesta de todos los combinatorios desde
    nC0 hasta nCk*/
  return dp[n][k];
}
/** Computa nCk mod p usando factoriales,
que pueden ser precomputados */
ll binomial(int n, int k, ll p) {
    vector<ll> fac = factorial(n, p); //Precomputarse
  vector<ll> inv = inv_factorial(n, p); //Precomputarse
  return fac[n] * inv[k] % p * inv[n - k] % p;
}
```

Para $m \leq 10^5$, se puede usar el teorema de Lucas que plantea

$$\binom{n}{k} \mod m = \prod_{i=1}^{\log m} \binom{n_i}{k_i}$$

Donde

$$n_i = \frac{n_{i-1}}{m}, \qquad n_0 = n$$

$$k_i = \frac{k_{i-1}}{m}, \qquad k_0 = k$$

### 5.5.2 nCk $\mod m$ si m es compuesto

Se realiza la descomposición en factores primos de m, resultando

$$k_i = \frac{k_{i-1}}{m}, \qquad k_0 = k$$

Por cada factor primo se computa

## 5.6 Miller-Rabin

Test de primalidad.

C++:

```
vector<int> a{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};

ll mult(ll a, ll b, ll mod) {
    return ((__int128)a * b) % mod;
}

/*
  a es la base.
  d es la potencia.
  n es el modulo.
*/
ll pw(ll a, ll d, ll n){ // pow in log(n)
  vector<ll> dp(63);
  dp[0] = a;
```

```cpp
  ll res;

  FOR(i, 1, 63) dp[i] = mult(dp[i - 1], dp[i - 1], n);

  deque<int> bits;

  FOR(i, 0, 63) if(d & (ll)1 << i) bits.PB(i);

  res = dp[bits.F()]%n;
  bits.P_F();

  while(!bits.empty()){
    res = (mult(res, dp[bits.F()], n))%n;
    bits.P_F();
  }

  return res;
}

bool prime(ll n){ // test de primalidad
  ll r, x, m, d;
  bool out;
  r = 0;
  m = n - 1;

  while(m%2 == 0){
    m /= 2;
    r++;
  }
  d = m;

  FOR(i, 0, a.size()){
    x = pw(a[i], d, n);
    out = false;
    if(x == 1 or x == n - 1) continue;
    else{
      FOR(j, 0, r - 1){
        x = mult(x, x, n);
        if(x == n - 1){
          out = true;
          break;
        }
      }
    }

    if(out) continue;
    return false;
  }
  return true;
}
```

## 5.7 Pollard Rho

Encontrar un divisor de P.

C++:

```cpp
ll mult(ll a, ll b, ll mod) {
    return ((__int128)a * b) % mod;
}

ll f(ll x, ll c, ll mod) {
    return (mult(x, x, mod) + c) % mod;
}

ll rho(ll n) {
  ll c = 1, x, y, g;
  y = x = 2;
  g = c;
  while(g == 1){
    x = f(x, c, n);
    y = f(y, c, n);
    y = f(y, c, n);
    g = __gcd(abs(x - y), n);
  }
  return g;
}
```

### 5.8 Inclusión-Exclusión

C++:

```cpp
ll inclusionExclusion(int pos, int size, ll res, ll x, vector<ll> &p){
  if(res > x) return 0;
  if(size == 0) return x/res;

  ll ans = 0;
  FOR(i, pos, p.size()){
    ans += inclusionExclusion(i + 1, size - 1, res*p[i]/__gcd(res, p[i]), x, p);
  }

  return ans;
}
```