

# Handbook

October 15, 2023

## Contents

<b>1</b>	<b>Grafos</b>	<b>3</b>
1.1	Dinic . . . . .	3
1.2	Bridges . . . . .	4
1.3	Articulation Points . . . . .	5
<b>2</b>	<b>Strings</b>	<b>7</b>
2.1	Función Z . . . . .	7
2.2	KMP . . . . .	7
2.3	Suffix array . . . . .	7
2.4	Longest Common Prefix on Suffixs . . . . .	9
2.5	Aho Corasick . . . . .	10
<b>3</b>	<b>Búsqueda</b>	<b>12</b>
3.1	Ternary Search . . . . .	12
<b>4</b>	<b>Geometría</b>	<b>13</b>
4.1	Convex Hull . . . . .	13
4.2	Interseccion de lineas . . . . .	14
4.3	Punto en polígono . . . . .	14
4.4	Ordenamiento por ángulo polar . . . . .	16
4.5	Área de polígono . . . . .	16
<b>5</b>	<b>Matemáticas</b>	<b>17</b>
5.1	Función $\phi$ de Euler . . . . .	17
5.2	Factorial modulo m . . . . .	17
5.3	Exponenciacion binaria . . . . .	17
5.4	Inverso Modular . . . . .	18
5.5	Inverso modular del factorial modulo m . . . . .	18
5.6	Coefficientes binomiales modulo m . . . . .	18
5.6.1	$nCk \mod m$ si m es primo . . . . .	19
5.6.2	$nCk \mod m$ si m es compuesto . . . . .	20
5.7	Miller-Rabin . . . . .	20
5.8	Pollard Rho . . . . .	21

5.9	Inclusión-Exclusión . . . . .	21
-----	-------------------------------	----

# 1 Grafos

## 1.1 Dinic

Insertar utilidad del algoritmo:

C++:

```
1 struct edge{
2     int x, y, flow;
3 };
4
5 int ans;
6 vector<edge> edges;
7 vector<vector<int>> grafo;
8 vector<int> sn;
9
10 void addEdge(int x, int y, int flow){
11     grafo[x].PB(edges.size());
12     edges.PB({x, y, flow});
13
14     grafo[y].PB(edges.size());
15     edges.PB({y, x, 0});
16 }
17
18 int bfs(int &ori, int &target){
19     int x = ori, y, flow;
20
21     FOR(i, 0, target + 1) sn[i] = INF;
22
23     sn[x] = 0;
24     deque<int> q(1, x);
25
26     while(!q.empty()){
27         x = q.F(); q.P_F();
28
29         for(auto &e: grafo[x]){
30             auto &edge = edges[e];
31             y = edge.y;
32             flow = edge.flow;
33
34             if(flow <= 0) continue;
35             if(sn[y] != INF) continue;
36             sn[y] = sn[x] + 1;
37             q.PB(y);
38         }
39     }
40
41     return sn[target];
42 }
43
44 int dfs(int ori, int &target, int min_flow, vector<int> &dp){
45     int flow = INF, y, e_id;
46
47     for(int &pos = dp[ori]; pos < grafo[ori].size(); ++pos){
48         e_id = grafo[ori][pos];
49         auto &e = edges[e_id];
```

```

50     y = e.y;
51
52     if(sn[y] != 1 + sn[ori]) continue;
53     if(e.flow <= 0) continue;
54
55     if(y == target){
56         flow = min(min_flow, e.flow);
57         ans += flow;
58         edges[e_id].flow -= flow;
59         edges[e_id^1].flow += flow;
60         return flow;
61     }
62
63     flow = dfs(y, target, min(min_flow, e.flow), dp);
64
65     if(flow != INF){
66         edges[e_id].flow -= flow;
67         edges[e_id^1].flow += flow;
68         return flow;
69     }
70 }
71
72 return flow;
73 }

```

## 1.2 Bridges

Encuentra las aristas  $(u, v)$  que si son retiradas del grafo, producen dos componentes completamente aisladas.

C++:

```

1  int n;
2  vector<vector<int>>> g;
3
4  vector<bool> visited;
5  vector<int> tin, low;
6  int timer;
7
8  void dfs(int v, int p = -1) {
9      visited[v] = true;
10     tin[v] = low[v] = timer++;
11     for (int to : g[v]) {
12         if (to == p) continue;
13         if (visited[to]) {
14             low[v] = min(low[v], tin[to]);
15         } else {
16             dfs(to, v);
17             low[v] = min(low[v], low[to]);
18             if (low[to] > tin[v])
19                 cout << "bridge: " << v << " " << to << "\n";
20         }
21     }
22 }

```

```

23
24 void find_bridges() {
25     timer = 0;
26     visited.assign(n, false);
27     tin.assign(n, -1);
28     low.assign(n, -1);
29     for (int i = 0; i < n; ++i) {
30         if (!visited[i])
31             dfs(i);
32     }
33 }

```

### 1.3 Articulation Points

Encuentra los nodos que si son retirados del grafo, producen dos componentes completamente aisladas. El algoritmo puede retornar nodos repetidos, las veces que estos nodos aparecen son la cantidad de componentes del grafo a la que pertenecen - 1.

C++:

```

1  int n;
2  vector<vector<int>> g;
3
4  vector<bool> visited;
5  vector<int> tin, low;
6  int timer;
7
8  void dfs(int v, int p = -1) {
9      visited[v] = true;
10     tin[v] = low[v] = timer++;
11     int children=0;
12     for (int &to : g[v]) {
13         if (to == p) continue;
14
15         if (visited[to]) low[v] = min(low[v], tin[to]);
16         else {
17             dfs(to, v);
18             low[v] = min(low[v], low[to]);
19             if (low[to] >= tin[v] && p != -1){
20                 IS_CUTPOINT(v);
21             }
22             ++children;
23         }
24     }
25
26     if(p == -1 && children > 1){
27         IS_CUTPOINT(v);
28     }
29 }
30
31 void find_cutpoints() {
32     timer = 0;
33     visited.assign(n, false);
34     tin.assign(n, -1);

```

```
35 low.assign(n, -1);
36 for (int i = 0; i < n; ++i) {
37     if (!visited[i]) dfs (i);
38 }
39 }
```

## 2 Strings

### 2.1 Función Z

Insertar utilidad del algoritmo:

C++:

```
1 vector<int> z_function(string s) {
2     int n = (int) s.length();
3     vector<int> z(n);
4     for (int i = 1, l = 0, r = 0; i < n; ++i) {
5         if (i <= r)
6             z[i] = min (r - i + 1, z[i - l]);
7         while (i + z[i] < n && s[z[i]] == s[i + z[i]])
8             ++z[i];
9         if (i + z[i] - 1 > r)
10            l = i, r = i + z[i] - 1;
11     }
12     return z;
13 }
```

### 2.2 KMP

Insertar utilidad del algoritmo:

C++:

```
1 vector<int> z;
2
3 void kmp(string &s){
4     int j, n = s.size();
5     z.resize(n);
6
7     FOR(i, 1, n){
8         j = z[i - 1];
9         while(j > 0 and s[i] != s[j]) j = z[j - 1];
10        if(s[i] == s[j]) j++;
11        z[i] = j;
12    }
13 }
```

### 2.3 Suffix array

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

C++:

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

```

7   for(int i = 1 ; i < n ; ++i) pos[i] = pos[i - 1] + cnt[i - 1];
8   for(auto x : p){
9       int i = c[x];
10      p_new[pos[i]] = x;
11      pos[i]++;
12  }
13  p = p_new;
14 }
15 vector<int> suffix_array(string &s){
16     s+=" ";
17     int n = s.size();
18     p.resize(n);
19     c.resize(n);
20     vector<pair<char, int>> a(n);
21     for(int i = 0 ; i < n ; ++i) a[i] = {s[i], i};
22     sort(a.begin(), a.end());
23     for(int i = 0 ; i < n ; ++i) p[i] = a[i].second;
24     c[p[0]] = 0;
25     for(int i = 1 ; i < n ; ++i)
26         c[p[i]] = a[i].first == a[i - 1].first ? c[p[i - 1]] : c[p[i - 1]] + 1;
27     int k = 0, shift;
28     while( (1<<k) < n ){
29         shift = 1<<k;
30         for(int i = 0 ; i < n ; ++i)
31             p[i] = (p[i] - (1<<k) + n) % n;
32         count_sort(p,c);
33         vector<int> c_new(n);
34         c_new[p[0]] = 0;
35         for(int i = 1 ; i < n ; ++i){
36             pair<int, int> prev = {c[p[i - 1]], c[(p[i - 1] + shift) % n]};
37             pair<int, int> now = {c[p[i]], c[(p[i] + shift) % n]};
38             if(prev == now) c_new[p[i]] = c_new[p[i - 1]];
39             else c_new[p[i]] = c_new[p[i - 1]] + 1;
40         }
41         c = c_new;
42         k++;
43     }
44     return p;
45 }

```

Java:

```

1  static int[] p, c;
2  public static class Suffix implements Comparable<Suffix> {
3      int index, r, next;
4      public Suffix(int index, int rank, int next){
5          this.index = index; this.r = rank; this.next = next;
6      }
7      public int compareTo(Suffix s){
8          return r != s.r ? r - s.r : (next != s.next ? next - s.next : index - s.index);
9      }
10 }
11 public static int[] sort(int[] p, int[] c){
12     int N = p.length;
13     int[] cnt = new int[N], pos = new int[N], p_new = new int[N];
14     for(int e : c) cnt[e]++;
15     for(int i = 1 ; i < N ; ++i) pos[i] = pos[i - 1] + cnt[i - 1];
16     for(int x : p){

```



```

17         p_new[pos[c[x]]] = x; pos[c[x]]++;
18     }
19     p = p_new;
20     return p;
21 }
22 public static int[] suffixArray(String s) {
23     s+=" ";
24     int n = s.length();
25     c = new int[n];
26     p = new int[n];
27     Suffix[] su = new Suffix[n];
28     for (int i = 0; i < n; ++i) su[i] = new Suffix(i, s.charAt(i), 0);
29     Arrays.sort(su);
30     for(int i = 0 ; i < n ; ++i) p[i] = su[i].index;
31     c[p[0]] = 0;
32     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;
33     int k = 0, shift;
34     while((1<<k) < n){
35         shift = (1<<k);
36         for(int i = 0 ; i < n ; ++i) p[i] = (p[i] - shift + n) % n;
37         p = sort(p, c);
38         int[] c_new = new int[n];
39         c_new[p[0]] = 0;
40         for(int i = 1 ; i < n ; ++i)
41             c_new[p[i]] = (c[p[i]] == c[p[i-1]] && c[(p[i]+shift) % n] == c[(p[i - 1] + shift)
42                 ? c_new[p[i - 1]] : c_new[p[i - 1]] + 1;
43         c = c_new;
44         ++k;
45     }
46     return p;
47 }

```

## 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++:

```

1 vector<int> lcp(vector<int> &p, vector<int> &c, string &s){
2     int n = p.size();
3     vector<int> lcp(n);
4     int k = 0;
5     for(int i = 0 ; i < n - 1 ; ++i){
6         int pi = c[i];
7         int j = p[pi - 1];
8         while(s[i + k] == s[j + k]) k++;
9         lcp[pi] = k;
10        k = max(k - 1, 0);
11    }
12    return lcp;
13 }

```

Java:

```

1 static int[] p, c, LCP;

```

```

2
3 static int[] lcp(int[] p, int[]c, String s){
4     int n = p.length;
5     LCP = new int[n];
6     int k = 0;
7     for(int i = 0 ; i < n - 1 ; ++i){
8         int pi = c[i];
9         int j = p[pi - 1];
10        while(s.charAt(i + k) == s.charAt(j + k)) k++;
11        LCP[pi] = k;
12        k = Math.max(k - 1, 0);
13    }
14    return LCP;
15 }

```

## 2.5 Aho Corasick

C++:

```

1 int trie[MAX][26], nds = 1;
2 int fin[MAX], fail[MAX], sure_fail[MAX];
3
4 int add(string &s){
5     int cr = 0, x;
6
7     for(const auto &c: s){
8         x = c - 'a';
9
10        if(trie[cr][x] == 0) trie[cr][x] = nds++;
11        cr = trie[cr][x];
12    }
13
14    fin[cr] = 1;
15    return cr;
16 }
17
18 void build(){
19     int x, cr = 0;
20
21     deque<int> q;
22     q.PB(cr);
23
24     while(!q.empty()){
25         cr = q.F(); q.P_F();
26
27         FOR(i, 0, 26){
28             x = trie[cr][i];
29             if(x) q.PB(x);
30
31             if(cr == 0) continue;
32             if(x == 0){
33                 trie[cr][i] = trie[fail[cr]][i];
34                 continue;
35             }
36
37             fail[x] = trie[fail[cr]][i];

```

```
38     sure_fail[x] = fin[fail[x]] ? fail[x] : sure_fail[fail[x]];
39   }
40 }
41 }
```

## 3 Búsqueda

### 3.1 Ternary Search

Insertar utilidad del algoritmo:

C++:

```
1 #define ld long double
2
3 ld ternary_search(ld l, ld r) {
4     ld eps = 1e-9;
5     ld m1, m2, f1, f2;
6     while (r - l > eps) {
7         m1 = l + (r - l) / 3;
8         m2 = r - (r - l) / 3;
9         f1 = f(m1);           //evaluates the function at m1
10        f2 = f(m2);           //evaluates the function at m2
11        if (f1 < f2) l = m1;
12        else r = m2;
13    }
14
15    //return the maximum of f(x) in [l, r]
16    return f(l);
17 }
```

## 4 Geometría

### 4.1 Convex Hull

Insertar utilidad del algoritmo:

C++:

```
1 struct pt {
2     double x, y;
3 };
4
5 int orientation(pt a, pt b, pt c) {
6     double v = a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y);
7     if (v < 0) return -1; // clockwise
8     if (v > 0) return +1; // counter-clockwise
9     return 0;
10 }
11
12 bool cw(pt a, pt b, pt c, bool include_collinear) {
13     int o = orientation(a, b, c);
14     return o < 0 || (include_collinear && o == 0);
15 }
16 bool ccw(pt a, pt b, pt c, bool include_collinear) {
17     int o = orientation(a, b, c);
18     return o > 0 || (include_collinear && o == 0);
19 }
20
21 void convex_hull(vector<pt>& a, bool include_collinear = false) {
22     if (a.size() == 1)
23         return;
24
25     sort(a.begin(), a.end(), [](pt a, pt b) {
26         return make_pair(a.x, a.y) < make_pair(b.x, b.y);
27     });
28     pt p1 = a[0], p2 = a.back();
29     vector<pt> up, down;
30     up.push_back(p1);
31     down.push_back(p1);
32     for (int i = 1; i < (int)a.size(); i++) {
33         if (i == a.size() - 1 || cw(p1, a[i], p2, include_collinear)) {
34             while (up.size() >= 2){
35                 if(cw(up[up.size()-2], up[up.size()-1], a[i], include_collinear)) break;
36                 up.pop_back();
37             }
38             up.push_back(a[i]);
39         }
40         if (i == a.size() - 1 || ccw(p1, a[i], p2, include_collinear)) {
41             while (down.size() >= 2){
42                 if(ccw(down[down.size()-2], down[down.size()-1], a[i], include_collinear)) break;
43                 down.pop_back();
44             }
45             down.push_back(a[i]);
46         }
47     }
48
49     if (include_collinear && up.size() == a.size()) {
```

```

50     reverse(a.begin(), a.end());
51     return;
52 }
53 a.clear();
54 for (int i = 0; i < (int)up.size(); i++)
55     a.push_back(up[i]);
56 for (int i = down.size() - 2; i > 0; i--)
57     a.push_back(down[i]);
58 }

```

## 4.2 Interseccion de lineas

C++:

```

1 struct line {
2     double a, b, c;
3 };
4
5 const double EPS = 1e-9;
6
7 double det(double a, double b, double c, double d) {
8     return a*d - b*c;
9 }
10
11 bool intersect(line m, line n, pt & res) {
12     double zn = det(m.a, m.b, n.a, n.b);
13     if (abs(zn) < EPS)
14         return false;
15     res.x = -det(m.c, m.b, n.c, n.b) / zn;
16     res.y = -det(m.a, m.c, n.a, n.c) / zn;
17     return true;
18 }
19
20 bool parallel(line m, line n) {
21     return abs(det(m.a, m.b, n.a, n.b)) < EPS;
22 }
23
24 bool equivalent(line m, line n) {
25     return abs(det(m.a, m.b, n.a, n.b)) < EPS
26         && abs(det(m.a, m.c, n.a, n.c)) < EPS
27         && abs(det(m.b, m.c, n.b, n.c)) < EPS;
28 }

```

## 4.3 Punto en polígono

C++:

```

1 const ld EPSILON = 0.000001;
2
3 struct pt{
4     ld x, y;
5 };
6
7 int orientation(pt &a, pt &b, pt &c){
8     ld A, B, C;

```

```

9   A = -(b.y - a.y);
10  B = b.x - a.x;
11  C = b.y*a.x - a.y*b.x;
12
13  ld result = A*c.x + B*c.y + C;
14  if(result > 0.0) return 1; // Clockwise;
15  if(result < 0.0) return -1; // Counter-clockwise;
16  return 0; // Collinear.
17 }
18
19 bool coord_in_bounds(ld x, ld y, ld a){
20     ld mini, maxi;
21     mini = min(x, y);
22     maxi = max(x, y);
23
24     return (a + EPSILON >= mini and a - EPSILON <= maxi);
25 }
26
27 bool point_in_segment(pt &a, pt &b, pt &c){
28     ld A, B, C;
29     A = -(b.y - a.y);
30     B = b.x - a.x;
31     C = b.y*a.x - a.y*b.x;
32
33     ld result = A*c.x + B*c.y + C;
34     if(fabs(result) < EPSILON){
35         if(coord_in_bounds(a.x, b.x, c.x) and coord_in_bounds(a.y, b.y, c.y)) return true;
36     }
37
38     return false;
39 }
40
41 bool lines_intersect(pt &a, pt &b, pt &c, pt &d){
42     int o1, o2, o3, o4;
43     o1 = orientation(a, b, c);
44     o2 = orientation(a, b, d);
45     o3 = orientation(c, d, a);
46     o4 = orientation(c, d, b);
47
48     if(o1 != o2 and o3 != o4) return true;
49
50     if(o1 == 0 and point_in_segment(a, b, c)) return true;
51     if(o2 == 0 and point_in_segment(a, b, d)) return true;
52     if(o3 == 0 and point_in_segment(c, d, a)) return true;
53     if(o4 == 0 and point_in_segment(c, d, b)) return true;
54
55     return false;
56 }
57
58 bool point_in_polygon(pt &P, vector<pt> &pts){
59     pt aux = pt{INF, P.y};
60     int intersections = 0, duplicatedIntersections = 0;
61
62     FOR(i, 0, pts.size()){
63         pt &p1 = pts[i];
64         pt &p2 = pts[(i + 1)%pts.size()];
65

```

```

66 // Projected line pass across one point.
67 if(fabs(P.y - p1.y) < EPSILON and P.x - EPSILON < p1.x) duplicatedIntersections++;
68
69 intersections += lines_intersect(P, aux, p1, p2);
70 }
71
72 return (intersections - duplicatedIntersections) & 1;
73 }

```

## 4.4 Ordenamiento por ángulo polar

C++:

```

1 struct pt {
2     ll x, y;
3 };
4
5 pt Ori = pt{0, 0};
6
7 // Vector 0a -> 0b
8 ll cross(pt a, pt b, pt 0){
9     return a.x*(b.y-0.y)+b.x*(0.y-a.y)+0.x*(a.y-b.y);
10 }
11
12 int orientation(pt a, pt b, pt 0) {
13     ll v = cross(a, b, 0);
14     if (v < 0) return -1; // clockwise
15     if (v > 0) return +1; // counter-clockwise
16     return 0;
17 }
18
19 bool firstHalf(pt a){
20     int o = orientation(pt{1, 0}, a, pt{0, 0});
21     if(o == 0) o = a.x > 0 ? 1 : -1;
22     return o > 0;
23 }
24
25 bool comp(pt &a, pt &b){
26     if(firstHalf(a) == firstHalf(b)) return cross(a, b, Ori) > 0;
27     return firstHalf(a);
28 }

```

## 4.5 Área de polígono

C++:

```

1 ld area(const vector<pt>& fig) {
2     ld res = 0;
3     for (unsigned i = 0; i < fig.size(); i++) {
4         pt p = i ? fig[i - 1] : fig.back();
5         pt q = fig[i];
6         res += (p.x - q.x) * (p.y + q.y);
7     }
8     return fabs(res) / 2;
9 }

```



## 5 Matemáticas

### 5.1 Función $\phi$ de Euler

Permite calcular  $\phi(n)$ , que equivale a los números menores que  $n$  y coprimos a este. Para números primos,  $\phi(n) = n - 1$

C++:

```
1 int phi(int n) {
2     int result = n;
3     for (int i = 2; i * i <= n; i++) {
4         if (n % i == 0) {
5             while (n % i == 0)
6                 n /= i;
7             result -= result / i;
8         }
9     }
10    if (n > 1)
11        result -= result / n;
12    return result;
13 }
```

### 5.2 Factorial modulo $m$

Permite calcular  $n! \bmod m$

C++:

```
1 vector<ll> factorial(ll N, ll p) {
2     vector<ll> fac(N + 1);
3     fac[0] = 1;
4     for (int i = 1; i <= N; i++)
5         fac[i] = fac[i - 1] * i % p;
6     return fac;
7 }
```

### 5.3 Exponenciación binaria

Permite calcular  $c \equiv a^b \pmod{m}$ .

Para valores de  $b$  muy grandes,  $a^b \equiv a^{b \bmod \phi(m)} \pmod{m}$ .

Siguiendo lo anterior, si  $m$  es primo:  $a^b \equiv a^{b \bmod m-1} \pmod{m}$

C++:

```
1 long long binpow(long long a, long long b, long long m) {
2     a %= m;
3     long long res = 1;
4     while (b > 0) {
5         if (b & 1)
6             res = res * a % m;
7         a = a * a % m;
8     }
```

```

8     b >>= 1;
9 }
10 return res;
11 }

```

## 5.4 Inverso Modular

Permite calcular  $a^{-1} \pmod{m}$ , 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++:

```

1 //Usando binpow
2 ll inv(ll a, ll mod){
3     ll n = mod - 2;
4     ll ans = binpow(a, n, mod);
5     return ans;
6 }
7 //Usando euclides extendido
8 ll inv(ll a, ll b) {
9     pair<ll,ll> x = extend_euclid(a, b);
10    ll ans = x.first + (x.first < 0) * b;
11    return ans;
12 }

```

## 5.5 Inverso modular del factorial modulo $m$

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

C++:

```

1 vector<ll> factorial(ll N, ll p) {
2     vector<ll> fac(N + 1);
3     fac[0] = 1;
4     for (int i = 1; i <= N; i++)
5         fac[i] = fac[i - 1] * i % p;
6     return fac;
7 }

```

## 5.6 Coeficientes binomiales modulo $m$

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

### 5.6.1 $nCk \mod m$ 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++:

```

1  /** Computa nCk mod p usando DP */
2  ll binomial(int n, int k, ll p) {
3      vector<vector<ll>> dp(n + 1, vector<ll> (k + 1, 0));
4      for (int i = 0; i <= n; i++) {
5          dp[i][0] = 1;
6          if (i <= k)
7              dp[i][i] = 1;
8      }
9      for (int i = 0; i <= n; i++)
10         for (int j = 1; j <= min(i, k); j++)
11             if (i != j)
12                 dp[i][j] = (dp[i - 1][j - 1] + dp[i - 1][j]) % p;
13         /** Puede retornarse el arreglo completo
14          * con la respuesta de todos los combinatorios desde
15          * nC0 hasta nCk */
16         return dp[n][k];
17     }
18     /** Computa nCk mod p usando factoriales,
19     * que pueden ser precomputados */
20     ll binomial(int n, int k, ll p) {
21         vector<ll> fac = factorial(n, p); //Precomputarse
22         vector<ll> inv = inv_factorial(n, p); //Precomputarse
23         return fac[n] * inv[k] % p * inv[n - k] % p;
24     }

```

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

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

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

### 5.6.2 $nCk \bmod m$ si $m$ es compuesto

Se realiza la descomposición en factores primos de  $m$ , resultando

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

Por cada factor primo se computa

## 5.7 Miller-Rabin

Test de primalidad.

C++:

```
1 vector<int> a{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
2
3 ll mult(ll a, ll b, ll mod) {
4     return ((__int128)a * b) % mod;
5 }
6
7 /*
8  a es la base.
9  d es la potencia.
10 n es el modulo.
11 */
12 ll pw(ll a, ll d, ll n){ // pow in log(n)
13     vector<ll> dp(63);
14     dp[0] = a;
15     ll res;
16
17     FOR(i, 1, 63) dp[i] = mult(dp[i - 1], dp[i - 1], n);
18
19     deque<int> bits;
20
21     FOR(i, 0, 63) if(d & (ll)1 << i) bits.PB(i);
22
23     res = dp[bits.F()]%n;
24     bits.P_F();
25
26     while(!bits.empty()){
27         res = (mult(res, dp[bits.F()], n))%n;
28         bits.P_F();
29     }
30
31     return res;
32 }
33
34 bool prime(ll n){ // test de primalidad
35     ll r, x, m, d;
36     bool out;
37     r = 0;
38     m = n - 1;
39
40     while(m%2 == 0){
41         m /= 2;
```

```

42     r++;
43 }
44 d = m;
45
46 FOR(i, 0, a.size()){
47     x = pw(a[i], d, n);
48     out = false;
49     if(x == 1 or x == n - 1) continue;
50     else{
51         FOR(j, 0, r - 1){
52             x = mult(x, x, n);
53             if(x == n - 1){
54                 out = true;
55                 break;
56             }
57         }
58     }
59
60     if(out) continue;
61     return false;
62 }
63 return true;
64 }

```

## 5.8 Pollard Rho

Encontrar un divisor de P.

C++:

```

1 ll mult(ll a, ll b, ll mod) {
2     return ((__int128)a * b) % mod;
3 }
4
5 ll f(ll x, ll c, ll mod) {
6     return (mult(x, x, mod) + c) % mod;
7 }
8
9 ll rho(ll n) {
10    ll c = 1, x, y, g;
11    y = x = 2;
12    g = c;
13    while(g == 1){
14        x = f(x, c, n);
15        y = f(y, c, n);
16        y = f(y, c, n);
17        g = __gcd(abs(x - y), n);
18    }
19    return g;
20 }

```

## 5.9 Inclusión-Exclusión

C++:

```

1 ll inclusionExclusion(int pos, int size, ll res, ll x, vector<ll> &p){
2     if(res > x) return 0;
3     if(size == 0) return x/res;
4
5     ll ans = 0;
6     FOR(i, pos, p.size()){
7         ans += inclusionExclusion(i + 1, size - 1, res*p[i]/__gcd(res, p[i]), x, p);
8     }
9
10    return ans;
11 }

```