

Sale con dp y fritas



Contents

1	Template	2
2	Data structures	2
2.1	Disjoint Set	2
2.2	MinHeap	3
3	Grafos	3
3.1	BFS	3
3.2	DFS	4
3.3	Bellman-Ford	5
3.4	Dijkstra	5
3.5	Floyd-Warshall	6
3.6	Dantzig	6
3.7	Kruskal	6
3.8	Toposort	7
4	Searching	7
4.1	Binary search	7
4.2	Integer ternary search	8
4.3	Ternary search	8
4.4	Intervals	8
5	Matematicas	10
5.1	Integrador Numerico Simpson	10
5.2	Euclides extendido	10
5.3	Ecuaciones diofanticas lineales	10
5.4	GCD - Maximo comun divisor	10
5.5	LCM - Minimo comun multiplo	10

5.6	Criba de Eratostenes	10
5.7	Fibonacci mod m	11
5.8	Teorema Chino del Resto	11
6	Other	11
6.1	Enumerar abecedario	11
6.2	Funciones utiles	11

1 Template

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 typedef long long ll;
5 typedef long double ld;
6
7 const ll UNDEFINED = -1;
8 const int MAX_N = 1e5 + 1;
9 const int MOD = 1e9 + 7;
10 const int INF = 1e9;
11 const ll LINF = 1e18;
12 const ll zero = 0;
13 const ld EPSILON = 1e-10;
14 const double PI = acos(-1.0);
15
16 #define pb push_back
17 #define fst first
18 #define snd second
19 #define esta(x,c) ((c).find(x) != (c).end()) // Devuelve true si x es
    un elemento de c.
20 #define all(c) (c).begin(),(c).end()
21
22 #define DBG(x) cerr << #x << " = " << (x) << endl
23 #define RAYA cerr << "-----" << endl
24
25 #define forn(i,n) for (int i=0;i<(int)(n);i++)
26 #define forsn(i,s,n) for (int i=(s);i<(int)(n);i++)
27 #define dforn(i,n) for(int i=(int)((n)-1);i>=0;i--)
28 #define dforsn(i,s,n) for(int i=(int)((n)-1);i>=(int)(s);i--)
29
30 // Show vector
31 template <typename T>
32 ostream & operator <<(ostream &os, const vector<T> &v) {
33     os << "[";
34     forn(i, v.size()) {
35         if (i > 0) os << ",";
36         os << v[i];
37     }
38     return os << "]";
39 }

```

```

40
41 // Show pair
42 template <typename T1, typename T2>
43 ostream & operator <<(ostream &os, const pair<T1, T2> &p) {
44     os << "{" << p.first << "," << p.second << "}";
45     return os;
46 }
47
48 // Show set
49 template <typename T>
50 ostream & operator <<(ostream &os, const set<T> &s) {
51     os << "{";
52     for(auto it = s.begin(); it != s.end(); it++){
53         if(it != s.begin()) os << ",";
54         os << *it;
55     }
56     return os << "}";
57 }
58
59 // ##### //
60
61 int main()
62 {
63     cin.tie(0);
64     cin.sync_with_stdio(0);
65
66     return 0;
67 }

```

2 Data structures

2.1 Disjoint Set

```

1 struct DisjointSet{
2     vector<ll> parent, rnk;
3     ll numOfComponents;
4
5     DisjointSet(ll n){
6         rnk.assign(n, 0);
7         for (ll i = 0; i < n; i++)
8             parent.push_back(i);
9         numOfComponents = n;
10    }

```

```

11 ll findSet(ll x){
12     if(parent[x]==x) return x;
13     parent[x] = findSet(parent[x]);
14     return parent[x];
15 }
16
17 void unionSet(ll x, ll y){
18     // Encontrar los representantes del conjunto.
19     x = findSet(x);
20     y = findSet(y);
21
22     // Si los conjuntos son disjuntos:
23     if (x != y){
24         // Pongo al que tiene menos rango por debajo del de mayor
25         // rango.
26         if (rnk[x] < rnk[y]){
27             parent[x] = y;
28         } else if (rnk[x] > rnk[y]){
29             parent[y] = x;
30         } else { // Si tienen el mismo rango, incremento del rango.
31             (rnk[x] == rnk[y])
32             parent[y] = x;
33             rnk[x]++;
34         }
35         numOfComponents--;
36     }
37 }
38
39 bool same(ll x, ll y){
40     return findSet(x) == findSet(y);
41 }

```

2.2 MinHeap

```

1 template <typename T>
2 using min_heap = priority_queue<T, vector<T>, greater<T>>;
3 // Uso: min_heap<pair<int, int>> q;

```

3 Grafos

3.1 BFS

```

1 // BFS Normal
2 void bfs(ll inicio, vector<vector<ll>> &ady, vector<bool> &vis, vector<
3     ll> &parent){
4     queue<ll> q;
5     q.push(inicio);
6     vis[inicio] = true;
7     while(!q.empty()){
8         ll v = q.front(); // v es el vertice que estoy procesando
9         q.pop();
10        for (ll u : ady[v]){
11            if (!vis[u]){
12                vis[u] = true;
13                parent[u] = v;
14                q.push(u);
15            }
16        }
17    }
18
19    // BFS que calcula los padres de los vertices
20    void calculatingParents(ll v, vector<vector<ll>> &adjList, vector<bool>
21        &visited, vector<ll> &parents){
22        ll n = adjList.size();
23        visited[v] = true;
24        parents[v] = v;
25
26        queue<ll> q;
27        q.push(v);
28
29        while (!q.empty()){
30            ll u = q.front();
31            q.pop();
32
33            for (int i = 0; i < adjList[u].size(); i++){
34                ll w = adjList[u][i];
35                if (!visited[w]){
36                    parents[w] = u;
37                    visited[w] = true;
38                    q.push(w);
39                }
40            }
41        }

```

```

42
43 // BFS que chequea si un grafo es bipartito
44 bool isBipartite(ll v, vector<vector<ll>> &adjList, vector<ll> &teams){
45     bool res = true;
46     ll n = adjList.size();
47     teams[v] = 1;
48
49     queue<ll> q;
50     q.push(v);
51
52     while (!q.empty()){
53         ll u = q.front();
54         q.pop();
55
56         for (int i = 0; i < adjList[u].size(); i++){
57             ll w = adjList[u][i];
58             if (teams[w] == 0){ // If the node isn't painted, paint it.
59                 if (teams[u] == 1){
60                     teams[w] = 2;
61                 } else {
62                     teams[w] = 1;
63                 }
64                 q.push(w);
65             } else {
66                 if (teams[w] == teams[u]){ // If the node is painted, I
67                     // check that its color is different from the v.
68                     res = false;
69                     break;
70                 }
71             }
72         }
73     }
74     return res;
75 }
76
77 // BFS que calcula el numero de componentes conexas
78 ll numberOfConnectedComponents(vector<vector<ll>> &adjList){
79     ll n = adjList.size();
80     vector<bool> visited(n, false);
81     vector<ll> parents(n, UNDEFINED);
82     ll res = 0;
83

```

```

84     for (int i = 0; i < n; i++){
85         if (!visited[i]){
86             bfs(i, adjList, visited, parents);
87         }
88     }
89
90     return res;
91 }
92
93 // BFS que calcula el numero de vertices en una componente conexas
94 ll numberOfVerticesInConnectedComponent(ll v, vector<vector<ll>> &
95     adjList){
96     ll n = adjList.size();
97     vector<bool> visited(n, false);
98     visited[v] = true;
99     ll res = 1;
100
101     queue<ll> q;
102     q.push(v);
103
104     while (!q.empty()){
105         ll w = q.front();
106         q.pop();
107
108         for (int i = 0; i < adjList[w].size(); i++){
109             ll u = adjList[w][i];
110
111             if (!visited[u]){
112                 q.push(u);
113                 visited[u] = true;
114                 res++;
115             }
116         }
117     }
118     return res;
119 }

```

3.2 DFS

```

1 // DFS simple
2 void dfs(ll v, vector<vector<ll>> &ady, vector<bool> &vis){
3     vis[v] = true;

```

```

4     for(ll u : adj[v]){
5         if (!vis[u]){
6             dfs(u, adj, vis);
7         }
8     }
9 }
10
11 // DFS que me dice si existe un ciclo en un grafo (no dirigido) y en
    caso de existir, retorna la back-edge
12 pair<bool, edge> hasCycle(int v, vector<vector<int>> &adjList, vector<
    bool> &visited, vector<int> &parents){
13     visited[v] = true;
14     edge e = {UNDEFINED, UNDEFINED};
15     pair<bool, edge> res = {false, e};
16
17     for (int i = 0; i < adjList[v].size(); i++){
18         int w = adjList[v][i];
19         if (!visited[w]){
20             parents[w] = v;
21             res = hasCycle(w, adjList, visited, parents);
22             if (res.first){
23                 break;
24             }
25         } else if (visited[w] && parents[v] != w && parents[v] !=
            UNDEFINED){
26             edge backEdge = {w,v};
27             return {true, backEdge};
28         }
29     }
30
31     return res;
32 }
    
```

3.3 Bellman-Ford

```

1 using peso = ll;
2 using indice_nodo = ll;
3 using nodo_pesado = pair<peso, indice_nodo>;
4
5 struct Edge {
6     ll a, b, cost;
7
8     Edge(ll desde, ll hasta, ll c) : a(desde), b(hasta), cost(c) {}
    
```

```

9 };
10
11 // #####
12
13 // Asume grafo representado como lista de aristas.
14
15 bool bellman_ford(ll n, indice_nodo inicio, vector<Edge> &edges, vector<
    peso> &dist){
16     // Devuelve true sii existe un ciclo de longitud negativa.
17     // Calcula SSSP en dist.
18
19     // Obtiene las distancias mas cortas desde inicio hacia todos.
20     dist[inicio] = 0;
21     for(ll i=0; i<n; i++){
22         for(Edge e : edges){
23             if(-LINF < dist[e.a] && dist[e.a] < LINF){
24                 dist[e.b] = min(dist[e.b], dist[e.a] + e.cost);
25             }
26         }
27     }
28
29     // Detectar ciclo de longitud negativa.
30     for(Edge e : edges){
31         if(-LINF < dist[e.a] && dist[e.a] < LINF){
32             if (dist[e.a] + e.cost < dist[e.b]){
33                 return true;
34             }
35         }
36     }
37     return false;
38 }
    
```

3.4 Dijkstra

```

1 using peso = ll;
2 using indice_nodo = ll;
3 using nodo_pesado = pair<peso, indice_nodo>;
4
5 // #####
6
7 // Devuelve el vector de distancias desde inicio al i-esimo vertice.
8 vector<ll> dijkstra(const indice_nodo inicio, const vector<vector<
    nodo_pesado>> &ady){
    
```

```

9     vector<ll> distancia(ady.size(), LINF);
10    // vector<ll> parent(ady.size(), UNDEFINED);
11    vector<bool> vis(ady.size(), false);
12
13    distancia[inicio] = 0;
14    set<nodo_pesado> q;
15
16    q.insert({0, inicio});
17
18    while(!q.empty()){
19        ll v = q.begin() -> second;
20        q.erase(q.begin());
21
22        if (vis[v]) {continue;}
23        vis[v] = true;
24        for(const auto& [longitud, u] : ady[v]){
25            // Longitud del camino de v hacia u.
26            // Relax:
27            if(distancia[v] + longitud < distancia[u]){
28                q.erase({distancia[u], u});
29                distancia[u] = distancia[v] + longitud;
30                // parent[u] = v;
31                q.insert({distancia[u], u});
32            }
33        }
34    }
35    return distancia;
36 }
```

3.5 Floyd-Warshall

```

1 // La matriz tuvo que ser inicializada como:
2 // d(s,v) = w(s,v) si existe
3 // d(s,v) = 0    si s = v
4 // d(s,v) = INF si no
5
6 void floyd_warshall (ll n, vector<vector<ll>> &matrizDist){
7     // matrizDist en la entrada era la matriz de distancias directas,
8     // luego del algoritmo queda calculada en ella APSP. (Distancias
9     // minimas i->j)
10    for (ll k = 0; k < n; ++k) {
11        for (ll i = 0; i < n; ++i) {
12            for (ll j = 0; j < n; ++j) {

```

```

11        if (matrizDist[i][k] < LINF && matrizDist[k][j] < LINF){
12            matrizDist[i][j] = min(matrizDist[i][j], matrizDist[
13                i][k] + matrizDist[k][j]);
14        }
15    }
16 }
17 }
```

3.6 Dantzig

```

1 void dantzig(ll n, vector<vector<peso>> &matrizDist){
2     // matrizDist en la entrada era la matriz de distancias directas,
3     // luego del algoritmo queda calculada en ella APSP.
4     for(ll k=0; k<n; k++){
5         for(ll i=0; i<k; i++){
6             for(ll j=0; j<k; j++){
7                 matrizDist[i][k] = min(matrizDist[i][k], matrizDist[i][j]
8                     + matrizDist[j][k]);
9                 matrizDist[k][i] = min(matrizDist[k][i], matrizDist[k][j]
10                    + matrizDist[j][i]);
11             }
12         }
13         for(ll i=0; i<k; i++){
14             for(ll j=0; j<k; ++j){
15                 matrizDist[i][j] = min(matrizDist[i][j], matrizDist[i][k]
16                    + matrizDist[k][j]);
17             }
18         }
19     }
20 }
```

3.7 Kruskal

```

1 // Asume grafo representado como lista de aristas.
2
3 ll kruskal(ll n, vector<pair<ll, pair<ll, ll>>> &lista_edges){
4     // Devuelve el costo del AGM. En caso de que no sea conexo, devuelve
5     // -1.
6     sort(lista_edges.begin(), lista_edges.end());
7     DisjointSet dsu(n+1);
8     ll res = 0;

```

```

8     for(auto e : lista_edges){
9         ll peso = e.first;
10        ll x = (e.second).first;
11        ll y = (e.second).second;
12
13        if (dsu.findSet(x) != dsu.findSet(y)){
14            dsu.unionSet(x, y);
15            res += peso;
16            n--; // Para verificar luego si es posible visitar todos.
17        }
18    }
19
20    if(n!=1){res = -1;} // No era conexo.
21    return res;
22 }
```

3.8 Toposort

```

1  enum Color {WHITE, GREY, BLACK}; // Sin visitar / en proceso /
   Procesado.
2
3  // Devuelve true si encuentra un ciclo.
4  bool tdfs(ll v, const vector<vector<ll>> &ady, vector<Color> &color,
   vector<ll> &orden){
5      color[v] = GREY;
6
7      for(auto u : ady[v]){
8          if(color[u] == GREY){ // Si encuentra un nodo en proceso, hay
   un ciclo.
9              return true;
10         }
11         else if (color[u] == WHITE){ // Si encuentra un nodo no
   visitado, realiza DFS.
12             if(tdfs(u, ady, color, orden)) return true;
13         }
14     }
15
16     orden.pb(v);
17     color[v] = BLACK;
18     return false;
19 }
20
21 // Devuelve true sii existe un ciclo en G.
```

```

22 // Si no existe ciclo, en orden queda almacenado un orden topologico de
   G.
23 bool toposort(const vector<vector<ll>> &ady, vector<ll> &orden){
24     vector<Color> color(ady.size(), WHITE);
25     orden.clear();
26
27     for(ll v=0; v < ady.size(); v++){
28         if (color[v] == WHITE){
29             if(tdfs(v, ady, color, orden)) return true;
30         }
31     }
32
33     reverse(orden.begin(), orden.end());
34     return false;
35 }
```

4 Searching

4.1 Binary search

```

1  // Asumiento que quiero hacer una busqueda binaria en el rango [0, n)
2  // Importante: chequear el indice porque en caso de que no exista
   elemento que cumple P(X) puede haber problemas
3
4  // Calcular extremo derecho que cumple P(X)
5  int l = -1; // extremo izquierdo del rango de busqueda -1
6  int r = n; // extremo derecho del rango de busqueda +1
7  while(r - l > 1) { // mientras que la distancia entre las fronteras sea
   >1 (es decir, mientras que no esten contiguas)
8      int mid = (l + r) / 2;
9      if(P(mid)) {
10         l = mid;
11     } else {
12         r = mid;
13     }
14 }
15 // l es el ultimo elemento que cumple P(X)
16
17 // Calcular extremo izquierdo que cumple P(X)
18 int l = -1;
19 int r = n;
20 while(r - l > 1) {
21     int mid = (l + r) / 2;
```

```

22     if(!P(mid)) {
23         l = mid;
24     } else {
25         r = mid;
26     }
27 }
28
29 // r es el primer elemento que cumple P(X)

```

4.2 Integer ternary search

```

1 // Busqueda ternaria entera sobre [lower, high].
2
3 ll l = lower; // extremo izquierdo del rango de busqueda.
4 ll r = high; // extremo derecho del rango de busqueda.
5 while(l < r) {
6     ll mid = (l + r) / 2;
7     if(f(mid) < f(mid+1)) { // (<) Busca el minimo | (>) Busca el
8         // maximo.
9         r = mid;
10    } else {
11        l = mid+1;
12    }
13 }
14 // Respuesta: f(l).

```

4.3 Ternary search

```

1 // Retorna el valor minimo de una funcion entre l y r. Se recomienda
2 // usar de 50 a 90 iteraciones.
3
4 double f(double x) {
5     double y = x; // funcion a evaluar que depende de x
6     return y;
7 }
8
9 double ternary_search(double l, double r, int it) {
10    double a = (2.0*l + r)/3.0;
11    double b = (l + 2.0*r)/3.0;
12    if (it == 0) return f(a);
13    if (f(a) < f(b)) return ternary_search(l, b, it-1);
14    return ternary_search(a, r, it-1);
15 }

```

4.4 Intervals

```

1 // We suppose that the intervals are correct: p.first <= p.second
2 // The intervals describe this: [p.first, p.second]
3
4 // This function checks if the intervals p1 and p2 are disjoint
5 bool areDisjoint(pair<ll, ll> &p1, pair<ll, ll> &p2) {
6     ll startingTimeP1 = p1.first;
7     ll endingTimeP1 = p1.second;
8     ll startingTimeP2 = p2.first;
9     ll endingTimeP2 = p2.second;
10
11     // This option checks supposing that p1 starts before than p2
12     bool option1 = endingTimeP1 < startingTimeP2;
13     // This option checks supposing that p2 starts before than p1
14     bool option2 = endingTimeP2 < startingTimeP1;
15
16     return option1 || option2;
17 }
18
19 // This function checks if p1 is included in p2
20 bool isIncluded(pair<ll, ll> &p1, pair<ll, ll> &p2){
21     ll startingTimeP1 = p1.first;
22     ll endingTimeP1 = p1.second;
23     ll startingTimeP2 = p2.first;
24     ll endingTimeP2 = p2.second;
25
26     bool res = startingTimeP2 <= startingTimeP1 && endingTimeP1 <=
27         endingTimeP2;
28     return res;
29 }
30
31 bool belongsToTheInterval(pair<ll, ll> &p1, ll x){
32     ll startingTimeP1 = p1.first;
33     ll endingTimeP1 = p1.second;
34
35     return startingTimeP1 <= x && x <= endingTimeP1;
36 }
37
38 // This function helps me to sort the intervals to have first those
39 // which finish earlier. In case of a tie, I choose first the interval
40 // which starts earlier.
41 bool customCompare(pair<ll, ll> &p1, pair<ll, ll> &p2){

```



```

39     ll endingTimeP1 = p1.second;
40     ll endingTimeP2 = p2.second;
41
42     if (endingTimeP1 < endingTimeP2){
43         return true;
44     } else if (endingTimeP1 > endingTimeP2){
45         return false;
46     }
47
48     ll startingTimeP1 = p1.first;
49     ll startingTimeP2 = p2.first;
50
51     return startingTimeP1 < startingTimeP2;
52 }
53
54 // This function helps me to sort the intervals to have first those
55 // which start earlier. In case of a tie, I choose first the interval
56 // which ends later.
57 bool customCompare2(pair<ll,ll> &p1, pair<ll,ll> &p2){
58     ll startingTimeP1 = p1.first;
59     ll startingTimeP2 = p2.first;
60
61     if (startingTimeP1 < startingTimeP2){
62         return true;
63     } else if (startingTimeP1 > startingTimeP2){
64         return false;
65     }
66
67     ll endingTimeP1 = p1.second;
68     ll endingTimeP2 = p2.second;
69
70     return endingTimeP1 > endingTimeP2;
71 }
72
73 ll maximumNumberOfDisjointIntervals(vector<pair<ll, ll>> &intervals){
74     int n = intervals.size();
75     if (n == 0) return 0;
76     sort(all(intervals), customCompare);
77
78     pair<ll, ll> lastInterval = intervals[0];
79     int res = 1;
80
81     forsn(i,1,n){

```

```

80         pair<ll, ll> currentInterval = intervals[i];
81
82         if (areDisjoint(currentInterval, lastInterval)){
83             lastInterval = currentInterval;
84             res++;
85         }
86     }
87
88     return res;
89 }
90
91 pair<ll, ll> redundantInterval(vector<pair<ll, ll>> &intervals){
92     int n = intervals.size();
93     pair<ll, ll> res = {UNDEFINED, UNDEFINED};
94
95     sort(all(intervals), customCompare2);
96
97     forn(i, n-1){
98         pair<ll, ll> currentInterval = intervals[i];
99         pair<ll, ll> nextInterval = intervals[i+1];
100
101         if (isIncluded(nextInterval, currentInterval)){
102             res = nextInterval;
103             break;
104         } else if (i+2 <= n-1){
105             pair<ll, ll> laterInterval = intervals[i+2];
106
107             if (belongsToTheInterval(currentInterval, nextInterval.first
108                                     )){
109                 pair<ll, ll> newInterval = {currentInterval.second + 1,
110                                             nextInterval.second};
111
112                 if (isIncluded(newInterval, laterInterval)){
113                     res = nextInterval;
114                     break;
115                 }
116             }
117         }
118     }
119
120     return res;

```

121 | }

5 Matematicas

5.1 Integrador Numerico Simpson

```

1 typedef double Funcion(double);
2 double integrar(Funcion *f, double a, double b, int n)
3 {
4     double h = (b-a)/(double)(n);
5     double res = 0.0;
6     double x0 = a;
7     double fx0 = f(x0);
8     const double h2 = h/2.0;
9     forn(i,n) {
10         double fx0h = f(x0+h);
11         res += fx0 + fx0h + 4.0 * f(x0+h2);
12         x0 += h;
13         fx0 = fx0h;
14     }
15     return res * h / 6.0;
16 }
```

5.2 Euclides extendido

```

1 // El algoritmo de Euclides extendido retorna el gcd(a, b) y calcula los
   // coeficientes enteros X y Y que satisfacen la ecuacion: a*X + b*Y =
   // gcd(a, b).
2 int x, y;
3 // O(log(max(a, b)))
4 int euclid(int a, int b) {
5     if (b == 0) { x = 1; y = 0; return a; }
6     int d = euclid(b, a%b);
7     int aux = x;
8     x = y;
9     y = aux - a/b*y;
10    return d;
11 }
```

5.3 Ecuaciones diofanticas lineales

```

1 // Encuentra x, y en la ecuacion de la forma ax + by = c
2 // Agregar Extended Euclides.
3 ll g;
```

```

4 bool diophantine(ll a, ll b, ll c) {
5     x = y = 0;
6     if (!a && !b) return (!c); // solo hay solucion con c = 0
7     g = euclid(abs(a), abs(b));
8     if (c % g) return false;
9     a /= g; b /= g; c /= g;
10    if (a < 0) x *= -1;
11    x = (x % b) * (c % b) % b;
12    if (x < 0) x += b;
13    y = (c - a*x) / b;
14    return true;
15 }
```

5.4 GCD - Maximo comun divisor

```

1 // Calcula el maximo comun divisor entre a y b mediante el algoritmo de
   // Euclides. Tambien se puede usar __gcd(a, b).
2 // O(log(max(a, b)))
3 int gcd(int a, int b) {
4     return b == 0 ? a : gcd(b, a%b);
5 }
```

5.5 LCM - Minimo comun multiplo

```

1 // Calculo del minimo comun multiplo usando el maximo comun divisor.
2 // O(log(max(a, b)))
3 int lcm(int a, int b) {
4     return a / __gcd(a, b) * b;
5 }
```

5.6 Criba de Eratostenes

```

1 // Guarda en primes los numeros primos menores o iguales a MX. Para
   // saber si p es un numero primo, hacer: if (!marked[p]).
2
3 const int MX = 1e6;
4 bool marked[MX+1];
5 vector<int> primes;
6 // O(MX log(log(MX)))
7 void sieve() {
8     marked[0] = marked[1] = true;
9     for (int i = 2; i <= MX; i++) {
10         if (marked[i]) continue;
11         primes.pb(i);
```

```

12     for (ll j = 1ll*i*i; j <= MX; j += i) marked[j] = true;
13 }
14 }
    
```

5.7 Fibonacci mod m

```

1 // Calcula fibonacci(n) % m.
2
3 // O(log(n))
4 int fibmod(ll n, int m) {
5     int a = 0, b = 1, c;
6     for (int i = 63-__builtin_clzll(n); i >= 0; i--) {
7         c = a;
8         a = (1ll*c*(2ll*b-c+m)) % m;
9         b = (1ll*c*c + 1ll*b*b) % m;
10        if ((n>>i) & 1) {
11            c = (a+b) % m;
12            a = b; b = c;
13        }
14    }
15    return a;
16 }
    
```

5.8 Teorema Chino del Resto

```

1 // Encuentra un x tal que para cada i : x es congruente con A_i mod M_i
2 // Devuelve {x, lcm}, donde x es la solucion con modulo lcm (lcm = LCM(
3 //   M_0, M_1, ...)). Dado un k : x + k*lcm es solucion tambien.
4 // Si la solucion no existe o la entrada no es valida devuelve {-1, -1}
5 // Agregar Extended Euclides.
6 pii crt(vector<int> A, vector<int> M) {
7     int n = A.size(), ans = A[0], lcm = M[0];
8     for (int i = 1; i < n; i++) {
9         int d = euclid(lcm, M[i]);
10        if ((A[i] - ans) % d) return {-1, -1};
11        int mod = lcm / d * M[i];
12        ans = (ans + x * (A[i] - ans) / d % (M[i] / d) * lcm) % mod;
13        if (ans < 0) ans += mod;
14        lcm = mod;
15    }
16    return {ans, lcm};
    }
    
```

6 Other

6.1 Enumerar abecedario

```

1 // 'A' es 65 | 'a' es 97.
2 int transformarLetra(char x) {return (int) (x - 97);}
3 char transformarNumero(int x) {return (char) (x + 97);}
    
```

6.2 Funciones utiles

```

1 cout << setprecision(numeric_limits<long double>::digits10 + 1);
2 string stri = to_string(myInt); int myint1 = stoi(myString); //
3     Funcionan con int y float.
4 // lower_bound: find the first pos in which val could be inserted
5 //   without changing the order.
6 // upper_bound: find last position in which val could be inserted without
7 //   changing the order.
8 lower_bound(v.begin(), v.end(), 30);
9 *max_element(a.begin(), a.end()); *min_element(a.begin(), a.end());
    
```

Aproximación del mayor número n de datos que pueden procesarse para cada una de las complejidades algoritmicas. Tomar esta tabla solo como referencia.

Complexity	n
$O(n!)$	11
$O(n^5)$	50
$O(2^n * n^2)$	18
$O(2^n * n)$	22
$O(n^4)$	100
$O(n^3)$	500
$O(n^2 \log_2 n)$	1.000
$O(n^2)$	10.000
$O(n \log_2 n)$	10^6
$O(n)$	10^8
$O(\sqrt{n})$	10^{16}
$O(\log_2 n)$	-
$O(1)$	-