

# Notebook

Para el regional elegimos nombre

6 de noviembre de 2025

## Índice

|   |          |  |           |
|---|----------|--|-----------|
| <b>1. Template</b>                      | <b>2</b> | 3.10. Gauss . . . . .                        | 9         |
| 1.1. run.sh . . . . .                   | 2        | 3.11. FFT . . . . .                          | 10        |
| 1.2. comp.sh . . . . .                  | 2        |  |           |
| 1.3. Makefile . . . . .                 | 2        |  |           |
| <b>2. Estructuras de datos</b>          | <b>3</b> | <b>4. Geometria</b>                          | <b>11</b> |
| 2.1. Sparse Table . . . . .             | 3        | 4.1. Punto . . . . .                         | 11        |
| 2.2. Segment Tree . . . . .             | 3        | 4.2. Linea . . . . .                         | 11        |
| 2.3. Segment Tree Lazy . . . . .        | 3        | 4.3. Poligono . . . . .                      | 12        |
| 2.4. Segment Tree 2D . . . . .          | 4        | 4.4. Circulo . . . . .                       | 13        |
| 2.5. Segment Tree Persistente . . . . . | 4        | 4.5. Convex Hull . . . . .                   | 14        |
| 2.6. Fenwick Tree . . . . .             | 5        | 4.6. Orden Radial . . . . .                  | 14        |
| 2.7. Treap . . . . .                    | 5        | 4.7. Par de puntos más cercano . . . . .     | 14        |
| 2.8. Union Find . . . . .               | 6        | 4.8. Arbol KD . . . . .                      | 15        |
| 2.9. Chull Trick . . . . .              | 6        | 4.9. Suma de Minkowski . . . . .             | 15        |
| 2.10. Chull Trick Dinámico . . . . .    | 6        |  |           |
| <b>3. Matemática</b>                    | <b>7</b> | <b>5. Strings</b>                            | <b>16</b> |
| 3.1. Criba Lineal . . . . .             | 7        | 5.1. Hashing . . . . .                       | 16        |
| 3.2. Phollard's Rho . . . . .           | 7        | 5.2. Suffix Array . . . . .                  | 16        |
| 3.3. Divisores . . . . .                | 8        | 5.3. String Functions . . . . .              | 17        |
| 3.4. Inversos Modulares . . . . .       | 8        | 5.4. Kmp . . . . .                           | 17        |
| 3.5. Catalan . . . . .                  | 8        | 5.5. Manacher . . . . .                      | 17        |
| 3.6. Lucas . . . . .                    | 9        | 5.6. Mínima Rotación Lexicográfica . . . . . | 17        |
| 3.7. Stirling-Bell . . . . .            | 9        | 5.7. Trie . . . . .                          | 18        |
| 3.8. DP Factoriales . . . . .           | 9        | 5.8. Suffix Automaton . . . . .              | 18        |
| 3.9. Estructura de Fracción . . . . .   | 9        |  |           |
|   |          | <b>6. Grafos</b>                             | <b>20</b> |
|   |          | 6.1. Dikjstra . . . . .                      | 20        |
|   |          | 6.2. LCA . . . . .                           | 20        |
|   |          | 6.3. Binary Lifting . . . . .                | 21        |
|   |          | 6.4. Toposort . . . . .                      | 21        |
|   |          | 6.5. Detección ciclos negativos . . . . .    | 21        |
|   |          | 6.6. Camino Euleriano . . . . .              | 22        |
|   |          | 6.7. Camino Hamiltoniano . . . . .           | 22        |
|   |          | 6.8. Tarjan SCC . . . . .                    | 22        |
|   |          | 6.9. Bellman-Ford . . . . .                  | 23        |
|   |          | 6.10. Puentes y Articulación . . . . .       | 23        |
|   |          | 6.11. Kruskal . . . . .                      | 23        |

|  |    |
|--|----|
| 6.12. Chequeo Bipartito . . . . .                        | 24 |
| 6.13. Centroid Decomposition . . . . .                   | 24 |
| 6.14. HLD . . . . .                                      | 24 |
| 6.15. Max Tree Matching . . . . .                        | 24 |
| 6.16. Min Tree Vertex Cover . . . . .                    | 24 |
| 6.17. 2-SAT . . . . .                                    | 24 |
| 6.18. K Colas . . . . .                                  | 24 |
| <b>7. Flujo</b>  |    |
| 7.1. Dinic . . . . .                                     | 24 |
| 7.2. Min Cost Max Flow . . . . .                         | 24 |
| 7.3. Hopcroft Karp . . . . .                             | 24 |
| 7.4. Kuhn . . . . .                                      | 24 |
| 7.5. Min Vertex Cover Bipartito . . . . .                | 24 |
| 7.6. Hungarian . . . . .                                 | 24 |
| <b>8. Optimización</b>                                   |    |
| 8.1. Ternary Search . . . . .                            | 24 |
| 8.2. Longest Increasing Subsequence . . . . .            | 24 |
| <b>9. Otros</b>  |    |
| 9.1. Mo . . . . .  | 24 |
| 9.2. Divide and Conquer Optimization . . . . .           | 24 |
| 9.3. Fijar el numero de decimales . . . . .              | 24 |
| 9.4. Hash Table (Unordered Map/ Unordered Set) . . . . . | 24 |
| 9.5. Indexed Set . . . . .                               | 24 |
| 9.6. Subconjuntos . . . . .                              | 24 |
| 9.7. Simpson . . . . .                                   | 24 |
| 9.8. Pragmas . . . . .                                   | 24 |
| 9.9. Random . . . . .                                    | 24 |
| 9.10. Utilidades de strings . . . . .                    | 24 |

## 1. Template

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 #define forr(i, a, b) for (int i = int(a); i < int(b); i++)
5 #define forn(i, n) forr(i,0,n)
6 #define dforr(i, a, b) for (int i = int(b)-1; i >= int(a); i--)
7 #define dforn(i, n) dforr(i,0,n)
8 #define all(v) begin(v),end(v)
9 #define sz(v) (int(size(v)))
10 #define pb push_back
11 #define fst first
12 #define snd second
13 #define mp make_pair
14 #define endl '\n'
15 #define dprint(v) cerr << __LINE__ << ":" "#v" = " << v << endl
16
17 using ll = long long;
18 using pii = pair<int,int>;
19
20 int main() {
21     ios::sync_with_stdio(0); cin.tie(0);
22 }
23
24 1.1. run.sh
25
26 clear
27 make -s $1 && ./$1 < $2
28
29 1.2. comp.sh
30
31 clear
32 make -s $1 2>&1 | head -$2

```

### 1.3. Makefile

```

1 CXXFLAGS = -std=gnu++2a -O2 -g -Wall -Wextra -Wshadow -Wconversion \
2 -fsanitize=address -fsanitize=undefined

```

## 2. Estructuras de datos

### 2.1. Sparse Table

```
1 #define oper min
2 Elemt st[K][1<<K]; // K tal que (1<<K) > n
3 void st_init(vector<Elemt>& a) {
4     int n = sz(a); // assert(K >= 31-_builtin_clz(2*n));
5     forn(i,n) st[0][i] = a[i];
6     forr(k,1,K) forn(i,n-(1<<k)+1)
7         st[k][i] = oper(st[k-1][i], st[k-1][i+(1<<(k-1))]);
8 }
9 Elemt st_query(int l, int r) { // assert(l<r);
10    int k = 31-_builtin_clz(r-l);
11    return oper(st[k][l], st[k][r-(1<<k)]);
12 }
13 // si la operacion no es idempotente
14 Elemt st_query(int l, int r) {
15     int k = 31-_builtin_clz(r-l);
16     Elemt res = st[k][l];
17     for (l+=(1<<k), k--; l<r; k--) {
18         if (l+(1<<k)<=r) {
19             res = oper(res, st[k][l]);
20             l += (1<<k);
21         }
22     }
23     return res;
24 }
```

### 2.2. Segment Tree

```
1 // Dado un array y una operacion asociativa con neutro, get(i, j)
2 // opera sobre el rango [i, j]
3 #define oper(x, y) max(x, y)
4 const int neutro=0;
5 struct RMQ{
6     int sz;
7     tipo t[4*MAXN];
8     tipo &operator[](int p){return t[sz+p];}
9     void init(int n){ // O(nlg n)
10 }
```

```
9     sz = 1 << (32-_builtin_clz(n));
10    forn(i, 2*sz) t[i]=neutro;
11 }
12 void updall(){dform(i, sz) t[i]=oper(t[2*i], t[2*i+1]);} // O(N)
13 tipo get(int i, int j){return get(i,j,1,0,sz);}
14 tipo get(int i, int j, int n, int a, int b){ // O(lgn)
15     if(j<=a || i>=b) return neutro;
16     if(i<=a && b<=j) return t[n];
17     int c=(a+b)/2;
18     return oper(get(i, j, 2*n, a, c), get(i, j, 2*n+1, c, b));
19 }
20 void set(int p, tipo val){ // O(lgn)
21     for(p+=sz; p>0 && t[p]!=val;){
22         t[p]=val;
23         p/=2;
24         val=oper(t[p*2], t[p*2+1]);
25     }
26 }
27 }rmq;
28 // Usage:
29 cin >> n; rmq.init(n); forn(i, n) cin >> rmq[i]; rmq.updall();
```

### 2.3. Segment Tree Lazy

```
1 //Dado un arreglo y una operacion asociativa con neutro, get(i, j)
2 // opera sobre el rango [i, j].
3 typedef int Elemt; //Elemt de los elementos del arreglo
4 typedef int Alt; //Elemt de la alteracion
5 #define oper(x,y) x+y
6 #define oper2(k,a,b) k*(b-a)//Aplicar actualizacion sobre [a, b)
7 const Elemt neutro=0; const Alt neutro2=-1;
8 struct RMQ{
9     int sz;
10    Elemt t[4*MAXN];
11    Alt dirty[4*MAXN];//las alteraciones pueden ser distintas a
12    Elemt &operator[](int p){return t[sz+p];}
13    void init(int n){//O(nlg n)
14        sz = 1 << (32-_builtin_clz(n));
```

```

14     forn(i, 2*sz) t[i]=neutro;
15     forn(i, 2*sz) dirty[i]=neutro2;
16 }
17 void push(int n, int a, int b){//propaga el dirty a sus hijos
18     if(dirty[n]!=0){
19         t[n]+=oper2(dirty[n], a, b); //altera el nodo
20         if(n<sz){//cambiar segun el problema
21             dirty[2*n] = dirty[n];
22             dirty[2*n+1] = dirty[n];
23         }
24         dirty[n]=0;
25     }
26 }
27 ELEM get(int i, int j, int n, int a, int b){//O(lgn)
28     if(j<=a || i>=b) return neutro;
29     push(n, a, b);
30     if(i<=a && b<=j) return t[n];
31     int c=(a+b)/2;
32     return oper(get(i, j, 2*n, a, c), get(i, j, 2*n+1, c, b));
33 }
34 ELEM get(int i, int j){return get(i,j,1,0,sz);}
35 //altera los valores en [i, j] con una alteracion de val
36 void alterar(Alt val,int i,int j,int n,int a,int b){//O(lgn)
37     push(n, a, b);
38     if(j<=a || i>=b) return;
39     if(i<=a && b<=j){
40         dirty[n]+=val;
41         push(n, a, b);
42         return;
43     }
44     int c=(a+b)/2;
45     alterar(val, i, j, 2*n, a, c);
46     alterar(val, i, j, 2*n+1, c, b);
47     t[n]=oper(t[2*n], t[2*n+1]);
48 }
49 void alterar(Alt val, int i, int j){alterar(val,i,j,1,0,sz);}
50 }rmq;

```

## 2.4. Segment Tree 2D

```

1 int n,m;
2 int a[MAXN][MAXN],st[4*MAXN][4*MAXN];
3 void build(){
4     forr(i,0,n)forr(j,0,m)st[i+n][j+m]=a[i][j];
5     forr(i,0,n)for(int j=m-1;j-->0)
6         st[i+n][j]=op(st[i+n][j<<1],st[i+n][j<<1|1]);
7     for(int i=n-1;i-->0)forr(j,0,2*m)
8         st[i][j]=op(st[i<<1][j],st[i<<1|1][j]);
9 }
10 void upd(int x, int y, int v){
11     st[x+n][y+m]=v;
12     for(int
13         j=y+m;j>1;j>>=1)st[x+n][j>>1]=op(st[x+n][j],st[x+n][j^1]);
14     for(int i=x+n;i>1;i>>=1)for(int j=y+m;j;j>>=1)
15         st[i>>1][j]=op(st[i][j],st[i^1][j]);
16 }
17 int query(int x0, int x1, int y0, int y1){
18     int r=NEUT;
19     for(int i0=x0+n,i1=x1+n;i0<i1;i0>>=1,i1>>=1){
20         int t[4],q=0;
21         if(i0&1)t[q++]=i0++;
22         if(i1&1)t[q++]=-i1;
23         forr(k,0,q)for(int j0=y0+m,j1=y1+m;j0<j1;j0>>=1,j1>>=1){
24             if(j0&1)r=op(r,st[t[k]][j0++]);
25             if(j1&1)r=op(r,st[t[k]][--j1]);
26         }
27     }
28     return r;
29 }

```

## 2.5. Segment Tree Persistente

```

1 const int LOG2N = 19; // ceil(log2(MAXN))
2 const int STLEN = 1<<LOG2N;
3
4 struct Mono {
5     // TODO agregar data
6     static Mono zero() { /* TODO */ } // neutro de la suma
7 };
8 Mono operator+ (Mono a, Mono b) { /* TODO */ } // asociativo

```

```

9
10 struct N {
11     N(Mono x_, N* l_, N* r_)
12     : x{x_}, l{l_}, r{r_} {}
13     Mono x; N* l; N* r;
14 };
15 N empty_node(Mono::zero(), &empty_node, &empty_node);
16
17 deque<N> st_alloc; // optimizacion: >30% mas rapido que 'new
18     N(x, l, r)'
19 N* make_node(Mono x, N* l, N* r) {
20     st_alloc.emplace_back(x, l, r);
21     return &st_alloc.back();
22 }
23 N* u_(N* t, int l, int r, int i, Mono x) {
24     if (i+1 <= l || r <= i) return t;
25     if (r-l == 1) return make_node(x, nullptr, nullptr);
26     int m = (l+r)/2;
27     auto lt = u_(t->l, l, m, i, x);
28     auto rt = u_(t->r, m, r, i, x);
29     return make_node(lt->x + rt->x, lt, rt);
30 }
31
32 int ql, qr;
33 Mono q_(N* t, int l, int r) {
34     if (qr <= l || r <= ql) return Mono::zero();
35     if (ql <= l && r <= qr) return t->x;
36     int m = (l+r)/2;
37     return q_(t->l, l, m) + q_(t->r, m, r);
38 }
39
40 // suma en rango: t[l,r]
41 Mono query(N* t, int l, int r) { ql = l; qr = r; return q_(t, 0,
42     STLEN); }
43
44 // asignacion en punto: t[i]=x
45 N* update(N* t, int i, Mono x) { return u_(t, 0, STLEN, i, x); }
46 /* uso:

```

```

47 auto t = &empty_node;
48 t = update(t, 0, Mono{10});
49 t = update(t, 5, Mono{5});
50 auto x = query(t, 0, 5); // devuelve Mono{10}
51 auto y = query(t, 0, 6); // devuelve Mono{10} + Mono{5}
52 auto z = query(t, 1, 6); // devuelve Mono{5}
53 */

```

## 2.6. Fenwick Tree

```

1 struct Fenwick { // 0-indexed, query [0, i), update [i]
2     int ft[MAXN+1]; // Uso: ft.u(idx, val); cout << ft.q(idx);
3     int u(int i0, int x) { for (int i=i0+1; i<=MAXN; i+=i&-i)
4         ft[i]+=x; }
5     ll q(int i0){ ll x=0; for (int i=i0; i>0; i-=i&-i) x+=ft[i];
6         return x; } };
7
8 struct RangeFT { // 0-indexed, query [0, 1), update [l, r)
9     Fenwick rate, err; // Uso: ft.u(l, r, val); cout << ft.q(l, r);
10    void u(int l, int r, int x) { // range update
11        rate.u(l, x); rate.u(r, -x); err.u(l, -x*l); err.u(r, x*r);
12    }
13    ll q(int i) { return rate.q(i) * i + err.q(i); } }; // prefix
14

```

## 2.7. Treap

```

1 // representa una lista como arbol con el orden implicito
2 struct node {
3     int val, prio, tam;
4     node *l, *r;
5 };
6 node *make(int val) {
7     return new node { val, rand(), 1, nullptr, nullptr };
8 }
9 int tam(node *n) { return n ? n->tam : 0; }
10 void recalc(node *n) { n->tam = tam(n->l) + 1 + get_tam(n->r); }
11 node* merge(node* s, node* t) {
12     if (s == nullptr) return t;
13     if (t == nullptr) return s;
14     if (s->prio > t->prio) {

```

```

15     s->r = merge(s->r, t);
16     recalc(s);
17     return s;
18 } else {
19     t->l = merge(s, t->l);
20     recalc(t);
21     return t;
22 }
23 }
24 pair<node*, node*> split(node *s, int k) {
25     if (s == nullptr) return {nullptr, nullptr};
26     if (tam(s->l) < k) {
27         if (s->l == nullptr) return {nullptr, nullptr};
28         auto [l, r] = split(s->r, k-tam(s->l)-1);
29         s->r = l;
30         recalc(s);
31         return {s, r};
32     } else {
33         auto [l, r] = split(s->l, k);
34         s->l = r;
35         recalc(s);
36         return {l, s};
37     }
38 } // usage: node *list = nullptr; list = merge(list, make(5))

```

## 2.8. Union Find

```

1 vector<int> uf(MAXN, -1);
2 int uf_find(int x) { return uf[x]<0 ? x : uf[x] = uf_find(uf[x]); }
3 bool uf_join(int x, int y){ // True si x e y estan en !=
    componentes
4     x = uf_find(x); y = uf_find(y);
5     if(x == y) return false;
6     if(uf[x] > uf[y]) swap(x, y);
7     uf[x] += uf[y]; uf[y] = x; return true;
8 }

```

## 2.9. Chull Trick

```

1 typedef ll tc;
2 struct Line{tc m,h;};

```

```

3 struct CHT { // for minimum (for maximum just change the sign of
    lines)
4     vector<Line> c;
5     int pos=0;
6     tc in(Line a, Line b){
7         tc x=b.h-a.h,y=a.m-b.m;
8         return x/y+(x%y?((x>0)?(y>0):0):0); // ==ceil(x/y)
9     }
10    void add(tc m, tc h){ // m's should be non increasing
11        Line l=(Line){m,h};
12        if(sz(c)&&m==c.back().m){
13            l.h=min(h,c.back().h);c.pop_back();if(pos)pos--;
14        }
15        while(sz(c)>1&&in(c.back(),l)<=in(c[sz(c)-2],c.back())){
16            c.pop_back();if(pos)pos--;
17        }
18        c.pb(l);
19    }
20    inline bool fbin(tc x, int m){return in(c[m],c[m+1])>x;}
21    tc eval(tc x){
22        // O(log n) query:
23        int s=0,e=c.size();
24        while(e-s>1){int m=(s+e)/2;
25            if(fbin(x,m-1))e=m;
26            else s=m;
27        }
28        return c[s].m*x+c[s].h;
29        // O(1) query (for ordered x's):
30        while(pos>0&&fbin(x,pos-1))pos--;
31        while(pos<c.size()-1&&!fbin(x,pos))pos++;
32        return c[pos].m*x+c[pos].h;
33    }
34 };

```

## 2.10. Chull Trick Dinámico

```

1 struct Entry {
2     using It = set<Entry>::iterator;
3     bool is_query;
4     ll m, b; mutable It it, end;

```

```

5     ll x;
6 };
7 bool operator< (Entry const& a, Entry const& b) {
8     if (!b.is_query) return a.m < b.m;
9     auto ni = next(a.it);
10    if (ni == a.end) return false;
11    auto const& c = *ni;
12    return (c.b-a.b) > b.x * (a.m-c.m);
13 }
14 struct ChullTrick {
15     using It = Entry::It;
16     multiset<Entry> lines;
17     bool covered(It it) {
18         auto begin = lines.begin(), end = lines.end();
19         auto ni = next(it);
20         if (it == begin && ni == end) return false;
21         if (it == begin) return ni->m==it->m && ni->b>=it->b;
22         auto pi = prev(it);
23         if (ni == end) return pi->m==it->m && pi->b>=it->b;
24         return (it->m-pi->m)*(ni->b-pi->b) >=
25             (pi->b-it->b)*(pi->m-ni->m);
26     }
27     bool add(ll m, ll b) {
28         auto it = lines.insert({false, m, b});
29         it->it = it; it->end = lines.end();
30         if (covered(it)) { lines.erase(it); return false; }
31         while (next(it) != lines.end() && covered(next(it)))
32             lines.erase(next(it));
33         while (it != lines.begin() && covered(prev(it)))
34             lines.erase(prev(it));
35         return true;
36     }
37     ll eval(ll x) {
38         auto l = *lines.lower_bound({true, -1, -1, {}, {}, x});

```

### 3. Matemática

#### 3.1. Criba Lineal

```

1 const int N = 10'000'000;
2 vector<int> lp(N+1);
3 vector<int> pr;
4 for (int i=2; i <= N; ++i) {
5     if (lp[i] == 0) lp[i] = i, pr.push_back(i);
6     for (int j = 0; i * pr[j] <= N; ++j) {
7         lp[i * pr[j]] = pr[j];
8         if (pr[j] == lp[i]) break;
9     }
10 }

```

#### 3.2. Pholland's Rho

```

1 ll mulmod(ll a, ll b, ll m) { return ll(__int128(a) * b % m); }
2
3 ll expmod(ll b, ll e, ll m) { // O(log b)
4     if (!e) return 1;
5     ll q=expmod(b,e/2,m); q=mulmod(q,q,m);
6     return e%2 ? mulmod(b,q,m) : q;
7 }
8
9 bool es_primo_prob(ll n, int a) {
10    if (n == a) return true;
11    ll s = 0, d = n-1;
12    while (d%2 == 0) s++, d/=2;
13    ll x = expmod(a,d,n);
14    if ((x == 1) || (x+1 == n)) return true;
15    forn(i,s-1){
16        x = mulmod(x,x,n);
17        if (x == 1) return false;
18        if (x+1 == n) return true;
19    }
20    return false;
21 }
22
23 bool rabin(ll n) { // devuelve true si n es primo

```

```

24     if (n == 1) return false;
25     const int ar[] = {2,3,5,7,11,13,17,19,23};
26     forn(j,9) if (!es_primo_prob(n,ar[j])) return false;
27     return true;
28 }
29
30 ll rho(ll n) {
31     if ((n & 1) == 0) return 2;
32     ll x = 2, y = 2, d = 1;
33     ll c = rand() % n + 1;
34     while (d == 1) {
35         x = (mulmod(x,x,n)+c)%n;
36         y = (mulmod(y,y,n)+c)%n;
37         y = (mulmod(y,y,n)+c)%n;
38         d=gcd(x-y,n);
39     }
40     return d==n ? rho(n) : d;
41 }
42
43 void factRho(map<ll,ll>&prim, ll n){ //O (lg n)^3. un solo numero
44     if (n == 1) return;
45     if (rabin(n)) { prim[n]++; return; }
46     ll factor = rho(n);
47     factRho(factor, prim); factRho(n/factor, prim);
48 }
49 auto fact(ll n){
50     map<ll,ll>prim;
51     factRho(prim,n);
52     return prim;
53 }

```

### 3.3. Divisores

```

1 // Usar asi: divisores(fac, divs, fac.begin()); NO ESTA ORDENADO
2 void divisores(const map<ll,ll> &f, vector<ll> &divs, auto it, ll
    n=1){
3     if (it==f.begin()) divs.clear();
4     if (it==f.end()) { divs.pb(n); return; }
5     ll p=it->fst, k=it->snd; ++it;
6     forn(_, k+1) divisores(f,divs,it,n), n*=p;

```

```

7 }
8
9 ll sumDiv (ll n){ //suma de los divisores de n
10    ll rta = 1;
11    map<ll,ll> f=fact(n);
12    for(auto it = f.begin(); it != f.end(); it++) {
13        ll pot = 1, aux = 0;
14        forn(i, it->snd+1) aux += pot, pot *= it->fst;
15        rta*=aux;
16    }
17    return rta;
18 }

```

### 3.4. Inversos Modulares

```

1 pair<ll,ll> extended_euclid(ll a, ll b) {
2     if (b == 0) return {1, 0};
3     auto [y, x] = extended_euclid(b, a%b);
4     y -= (a/b)*x;
5     if (a*x + b*y < 0) x = -x, y = -y;
6     return {x, y}; // a*x + b*y = gcd(a,b)
7 }

1 constexpr ll MOD = 1000000007; // tmb es comun 998'244'353
2 ll invmod[MAXN]; // inversos módulo MOD hasta MAXN
3 void invmods() { // todo entero en [2,MAXN] debe ser coprimo con
    MOD
4     inv[1] = 1;
5     forall(i, 2, MAXN) inv[i] = MOD - MOD/i*inv[MOD%i] %MOD;
6 }
7
8 // si MAXN es demasiado grande o MOD no es fijo:
9 // versión corta, m debe ser primo. O(log(m))
10 ll invmod(ll a, ll m) { return expmod(a,m-2,m); }
11 // versión larga, a y m deben ser coprimos. O(log(a)), en general
    más rápido
12 ll invmod(ll a, ll m) { return (extended_euclid(a,m).fst % m + m)
    % m; }

```

### 3.5. Catalan

```

1 11 Cat(int n){
2     return ((F[2*n] *FI[n+1]) %M *FI[n]) %M;
3 }

```

### 3.6. Lucas

```

1 const ll MAXP = 3e3+10; //68 MB, con 1e4 int son 380 MB
2 ll C[MAXP][MAXP], P;    //inicializar con el primo del input < MAXP
3 void llenar_C(){
4     forn(i, MAXP) C[i][0] = 1;
5     forn(i, 1, MAXP) forn(j, 1, i+1)
6         C[i][j]=addmod(C[i-1][j-1],C[i-1][j], P);
7 }
8 // Calcula nCk (mod p) con n, k arbitrariamente grandes y p primo
9 // <= 3000
10 ll lucas(ll N, ll K){ // llamar a llenar_C() antes
11     ll ret = 1;
12     while(N>K){
13         ret = ret * C[N%P][K%P] % P;
14         N /= P, K /= P;
15     }
16     return ret;
17 }

```

### 3.7. Stirling-Bell

```

1 11 STR[MAXN][MAXN], Bell[MAXN];
2 //STR[n][k] = formas de particionar un conjunto de n elementos en
3 //k conjuntos
4 //Bell[n] = formas de particionar un conjunto de n elementos
5 forn(i, 1, MAXN)STR[i][1] = 1;
6 forn(i, 2, MAXN)STR[1][i] = 0;
7 forn(i, 2, MAXN)forn(j, 2, MAXN){
8     STR[i][j] = (STR[i-1][j-1] + j*STR[i-1][j] %MOD) %MOD;
9 }
10 forn(i, MAXN){
11     Bell[i] = 0;
12     forn(j, MAXN){
13         Bell[i] = (Bell[i] + STR[i][j]) %MOD;
14     }
15 }

```

### 3.8. DP Factoriales

```

1 11 F[MAXN], INV[MAXN], FI[MAXN];
2 // ...
3 F[0] = 1; forn(i, 1, MAXN) F[i] = F[i-1]*i %M;
4 INV[1] = 1; forn(i, 2, MAXN) INV[i] = M - (ll)(M/i)*INV[M%i] %M;
5 FI[0] = 1; forn(i, 1, MAXN) FI[i] = FI[i-1]*INV[i] %M;

```

### 3.9. Estructura de Fracción

```

1 tipo mcd(tipo a, tipo b){return a?mcd(b%a, a):b;}
2 struct frac{
3     tipo p,q;
4     frac(tipo p=0, tipo q=1):p(p),q(q) {norm();}
5     void norm(){
6         tipo a = mcd(p,q);
7         if(a) p/=a, q/=a;
8         else q=1;
9         if (q<0) q=-q, p=-p;
10    }
11    frac operator+(const frac& o){
12        tipo a = mcd(q,o.q);
13        return frac(p*(o.q/a)+o.p*(q/a), q*(o.q/a));}
14    frac operator-(const frac& o){
15        tipo a = mcd(q,o.q);
16        return frac(p*(o.q/a)-o.p*(q/a), q*(o.q/a));}
17    frac operator*(frac o){
18        tipo a = mcd(q,o.p), b = mcd(o.q,p);
19        return frac((p/b)*(o.p/a), (q/a)*(o.q/b));}
20    frac operator/(frac o){
21        tipo a = mcd(q,o.q), b = mcd(o.p,p);
22        return frac((p/b)*(o.q/a), (q/a)*(o.p/b));}
23    bool operator<(const frac &o) const{return p)o.q < o.p*q;}
24    bool operator==(frac o){return p==o.p&&q==o.q;}
25 };

```

### 3.10. Gauss

```

1 double reduce(vector<vector<double>> &a){ //Devuelve determinante
2     si m == n
3         int m=sz(a), n=sz(a[0]), i=0, j=0; double r = 1.0;
4         while(i < m and j < n){
5             ...
6         }
7     }
8 }

```

```

4     int h = i;
5     forr(k, i+1, m) if(abs(a[k][j]) > abs(a[h][j])) h = k;
6     if(abs(a[h][j]) < EPS){ j++; r=0.0; continue; }
7     if(h != i){ r = -r; swap(a[i], a[h]); }
8     r *= a[i][j];
9     dforr(k, j, n) a[i][k] /= a[i][j];
10    forr(k, 0, m) if(k != i)
11        dforr(l_, j, n) a[k][l_] -= a[k][j] * a[i][l_];
12    i++; j++;
13 }
14 return r;
15 }
```

### 3.11. FFT

```

1 // MAXN must be power of 2 !!, MOD-1 needs to be a multiple of
2 // MAXN !!
3
4 typedef ll tf;
5 typedef vector<tf> poly;
6 //const tf MOD = 2305843009255636993, RT = 5;
7 const tf MOD = 998244353, RT = 3;
8 // const tf MOD2 = 897581057, RT2 = 3; // Chinese Remainder Theorem
9 /* FFT */ struct CD {
10     double r, i;
11     CD(double r_ = 0, double i_ = 0) : r(r_), i(i_) {}
12     void operator/=(const int c) { r/=c, i/=c; }
13 };
14 CD operator*(const CD& a, const CD& b){
15     return CD(a.r*b.r-a.i*b.i, a.r*b.i+a.i*b.r);}
16 CD operator+(const CD& a, const CD& b) { return CD(a.r+b.r,
17 a.i+b.i); }
18 CD operator-(const CD& a, const CD& b) { return CD(a.r-b.r,
19 a.i-b.i); }
20 /* NTT */ struct CD { tf x; CD(tf x_) : x(x_) {} CD() {} };
21 CD operator+(const CD& a, const CD& b) { return CD(addmod(a.x,
22 b.x)); } //ETC
23 vector<tf> rts(MAXN+9, -1);
24 CD root(int n, bool inv){
25     tf r = rts[n]<0 ? rts[n] = expmod(RT, (MOD-1)/n) : rts[n];
26     return CD(inv ? expmod(r, MOD-2) : r);
```

```

27 }
28 /* AMBOS */ CD cp1[MAXN+9], cp2[MAXN+9];
29 int R[MAXN+9];
30 void dft(CD* a, int n, bool inv){
31     double pi = acos(-1.0);
32     forn(i, n) if(R[i] < i) swap(a[R[i]], a[i]);
33     for(int m = 2; m <= n; m *= 2){
34         /* FFT */ double z = 2*pi/m * (inv?-1:1);
35         /* FFT */ CD wi = CD(cos(z), sin(z));
36         /* NTT */ CD wi = root(m, inv);
37         for(int j = 0; j < n; j += m){
38             CD w(1);
39             for(int k = j, k2 = j+m/2; k2 < j+m; k++, k2++){
40                 CD u = a[k]; CD v = a[k2]*w; a[k] = u+v; a[k2] =
41                 u-v; w = w*wi;
42             }
43         }
44     }
45     /* FFT */ if(inv) forn(i, n) a[i] /= n;
46     /* NTT */ if(inv){
47         CD z(expmod(n, MOD-2));
48         forn(i, n) a[i] = a[i]*z;
49     }
50     poly multiply(poly& p1, poly& p2){
51         int n = sz(p1)+sz(p2)+1;
52         int m = 1, cnt = 0;
53         while(m <= n) m *= 2, cnt++;
54         forn(i, m) { R[i] = 0; forn(j, cnt) R[i] =
55             (R[i]<<1)|(j>>1); }
56         forn(i, m) cp1[i] = 0, cp2[i] = 0;
57         forn(i, sz(p1)) cp1[i] = p1[i];
58         forn(i, sz(p2)) cp2[i] = p2[i];
59         dft(cp1, m, false); dft(cp2, m, false);
60         // fast eval: forn(i, sz(p1)) p1(expmod(RT, (MOD-1)/m*i)) ==
61         cp1[i].x
62         forn(i, m) cp1[i] = cp1[i]*cp2[i];
63         dft(cp1, m, true);
64         poly res;
65         n -= 2;
```

```

59  /* FFT */ forn(i, n) res.pb((tf)floor(cp1[i].r+0.5));
60  /* NTT */ forn(i, n)res.pb(cp1[i].x);
61  return res;
62 }

4. Geometria

4.1. Punto

1 using T = double;
2 bool iszero(T u) { return abs(u)<=EPS; }
3 struct Pt {
4     T x, y;
5     T z; // only for 3d
6     Pt() {}
7     Pt(T _x, T _y) : x(_x), y(_y) {}
8     Pt(T _x, T _y, T _z) : x(_x), y(_y), z(_z) {} // for 3d
9     T norm2(){ return *this**this; }
10    T norm(){ return sqrt(norm2()); }
11    Pt operator+(Pt o){ return Pt(x+o.x,y+o.y); }
12    Pt operator-(Pt o){ return Pt(x-o.x,y-o.y); }
13    Pt operator*(T u){ return Pt(x*u,y*u); }
14    Pt operator/(T u) {
15        if (iszero(u)) return Pt(INF,INF);
16        return Pt(x/u,y/u);
17    }
18    T operator*(Pt o){ return x*o.x+y*o.y; }
19    Pt operator^(Pt p){ // only for 3D
20        return Pt(y*p.z-z*p.y, z*p.x-x*p.z, x*p.y-y*p.x); }
21    T operator%(Pt o){ return x*o.y-y*o.x; }
22    T angle(Pt o){ return atan2(*this%o, *this*o); }
23    // T angle(Pt o){ // accurate around 90 degrees
24    //     if (*this%o>0) return acos(*this*o);
25    //     return 2*M_PI-acos(*this*o); }
26    Pt unit(){ return *this/norm(); }
27    bool left(Pt p, Pt q){ // is it to the left of directed line
28        pq?
29        return ((q-p)%(*this-p))>EPS; }
30    bool operator<(Pt p) const{ // for convex hull
31        return x<p.x-EPS||(iszero(x-p.x)&&y<p.y-EPS); }

```

```

31    bool collinear(Pt p, Pt q){
32        return iszero((p-*this)%(q-*this)); }
33    bool dir(Pt p, Pt q){ // does it have the same direction of pq?
34        return this->collinear(p, q)&&(q-p)*(*this-p)>EPS; }
35    Pt rot(Pt r){ return Pt(*this%r,*this*r); }
36    Pt rot(T a){ return rot(Pt(sin(a),cos(a))); }
37    };
38    Pt ccw90(1,0);
39    Pt cw90(-1,0);

4.2. Linea

1 using T = double;
2 int sgn2(T x){return x<0?-1:1;}
3 struct Ln {
4     Pt p,pq;
5     Ln(Pt p, Pt q):p(p),pq(q-p){}
6     Ln(){}
7     bool has(Pt r){return dist(r)<=EPS; }
8     bool seghas(Pt r){return has(r)&&(r-p)*(r-(p+pq))<=EPS; }
9     // bool operator /(Ln l){return
10        (pq.unit()^l.pq.unit()).norm()<=EPS;} // 3D
11    bool operator /(Ln l){return abs(pq.unit()^l.pq.unit())<=EPS;} // 2D
12    bool operator==(Ln l){return *this/l&&has(l.p);}
13    Pt operator^(Ln l){ // intersection
14        if(*this/l) return Pt(INF,INF);
15        T a=-pq.y, b=pq.x, c=p.x*a+p.y*b;
16        T la=-l.pq.y, lb=l.pq.x, lc=l.p.x*la+l.p.y*lb;
17        T det = a * lb - b * la;
18        Pt r((lb*c-b*lc)/det, (a*lc-c*la)/det);
19        // Pt r=l.p+l.pq*((p-l.p)^pq)/(l.pq^pq));
20        // if(!has(r)){return Pt(NAN,NAN,NAN);} // check only for 3D
21    }
22    T angle(Ln l){return pq.angle(l.pq);}
23    int side(Pt r){return has(r)?0:sgn2(pq^(r-p));} // 2D
24    Pt proj(Pt r){return p+pq*((r-p)*pq/pq.norm2());}
25    Pt segclosest(Pt r) {
26        T l2 = pq.norm2();
```

```

27     if(12==0.) return p;
28     T t =((r-p)*pq)/12;
29     return p+(pq*min(1,max(0,t)));
30 }
31 Pt ref(Pt r){return proj(r)*2-r;}
32 T dist(Pt r){return (r-proj(r)).norm();}
33 // T dist(Ln l){ // only 3D
34 //     if(*this/l) return dist(l.p);
35 //     return abs((l.p-p)*(pq^l.pq))/(pq^l.pq).norm();
36 //}
37 Ln rot(auto a){return Ln(p,p+pq.rot(a));} // 2D
38 };
39 Ln bisector(Ln l, Ln m){ // angle bisector
40     Pt p=l^m;
41     return Ln(p,p+l.pq.unit()+m.pq.unit());
42 }
43 Ln bisector(Pt p, Pt q){ // segment bisector (2D)
44     return Ln((p+q)*.5,p).rot(ccw90);
45 }


```

### 4.3. Poligono

```

1 using T = double;
2 struct Pol {
3     int n;vector<Pt> p;
4     Pol(){}
5     Pol(vector<Pt> _p){p=_p;n=p.size();}
6     T area() {
7         ll a = 0;
8         forr (i, 1, sz(p)-1) {
9             a += (p[i]-p[0])^(p[i+1]-p[0]);
10        }
11        return abs(a)/2;
12    }
13    bool has(Pt q){ // O(n), winding number
14        forr(i,0,n)if(Ln(p[i],p[(i+1)%n]).seghas(q))return true;
15        int cnt=0;
16        forr(i,0,n){
17            int j=(i+1)%n;
18            int k=sgn((q-p[j])^(p[i]-p[j]));

```

```

19                int u=sgn(p[i].y-q.y),v=sgn(p[j].y-q.y);
20                if(k>0&&u<0&&v>0)cnt++;
21                if(k<0&&v<0&&u>0)cnt--;
22            }
23            return cnt!=0;
24        }
25        void normalize(){ // (call before haslog, remove collinear
26            first)
27            if(n>3&&p[2].left(p[0],p[1]))reverse(p.begin(),p.end());
28            int pi=min_element(p.begin(),p.end())-p.begin();
29            vector<Pt> s(n);
30            forr(i,0,n)s[i]=p[(pi+i)%n];
31            p.swap(s);
32        }
33        bool haslog(Pt q){ // O(log(n)) only CONVEX. Call normalize
34            first
35            if(q.left(p[0],p[1])||q.left(p.back(),p[0]))return false;
36            int a=1,b=p.size()-1; // returns true if point on boundary
37            while(b-a>1){ // (change sign of EPS in left
38                int c=(a+b)/2; // to return false in such case)
39                if(!q.left(p[0],p[c]))a=c;
40                else b=c;
41            }
42            return !q.left(p[a],p[a+1]);
43        }
44        bool isconvex(){//O(N), delete collinear points!
45            if(n<3) return false;
46            bool isLeft=p[0].left(p[1], p[2]);
47            forr(i, 1, n)
48                if(p[i].left(p[(i+1)%n], p[(i+2)%n])!=isLeft)
49                    return false;
50            return true;
51        }
52        Pt farthest(Pt v){ // O(log(n)) only CONVEX
53            if(n<10){
54                int k=0;
55                forr(i,1,n)if(v*(p[i]-p[k])>EPS)k=i;
56                return p[k];
57            }
58            if(n==sz(p))p.pb(p[0]);

```

```

57     Pt a=p[1]-p[0];
58     int s=0,e=n,ua=v*a>EPS;
59     if(!ua&&v*(p[n-1]-p[0])<=EPS) return p[0];
60     while(1){
61         int m=(s+e)/2;Pt c=p[m+1]-p[m];
62         int uc=v*c>EPS;
63         if(!uc&&v*(p[m-1]-p[m])<=EPS) return p[m];
64         if(ua&&(!uc||v*(p[s]-p[m])>EPS)) e=m;
65         else if(ua||uc||v*(p[s]-p[m])>=-EPS)s=m,a=c,ua=uc;
66         else e=m;
67         assert(e>s+1);
68     }
69 }
70 Pol cut(Ln l){ // cut CONVEX polygon by line l
71     vector<Pt> q; // returns part at left of l.pq
72     forr(i,0,n){
73         int
74             d0=sgn(l.pq^(p[i]-l.p)),d1=sgn(l.pq^(p[(i+1)%n]-l.p));
75             if(d0>0)q.pb(p[i]);
76             Ln m(p[i],p[(i+1)%n]);
77             if(d0*d1<0&&!(1/m))q.pb(l^m);
78     }
79     return Pol(q);
80 }
81 T intercircle(circle c){ // area of intersection with circle
82     T r=0.;
83     forr(i,0,n){
84         int j=(i+1)%n;T w=c.intertriangle(p[i],p[j]);
85         if((p[j]-c.o)^(p[i]-c.o)>EPS)r+=w;
86         else r-=w;
87     }
88     return abs(r);
89 }
90 T callipers(){ // square distance of most distant points
91     T r=0; // prereq: convex, ccw, NO COLLINEAR POINTS
92     for(int i=0,j=n<2?0:1;i<j;++i){
93         for(;j=(j+1)%n{
94             r=max(r,(p[i]-p[j]).norm2());
95             if(((p[(i+1)%n]-p[i])^(p[(j+1)%n]-p[j]))<=EPS)break;
96         }
97     }
98     return r;
99 }

```

```

96     }
97     return r;
98 }
99 };

```

#### 4.4. Circulo

```

1  using T = double;
2  struct Circle {
3      Pt o;T r;
4      Circle(Pt o, T r):o(o),r(r){}
5      Circle(Pt x, Pt y, Pt
6          z){o=bisector(x,y)^bisector(x,z);r=(o-x).norm();}
7      bool has(Pt p){return (o-p).norm()<=r+EPS;}
8      vector<Pt> operator^(Circle c){ // ccw
9          vector<Pt> s;
10         T d=(o-c.o).norm();
11         if(d>r+c.r+EPS||d+min(r,c.r)+EPS<max(r,c.r))return s;
12         T x=(d*d-c.r*c.r+r*r)/(2*d);
13         T y=sqrt(r*r-x*x);
14         Pt v=(c.o-o)/d;
15         s.pb(o+v*x-v.rot(ccw90)*y);
16         if(y>EPS)s.pb(o+v*x+v.rot(ccw90)*y);
17         return s;
18     }
19     vector<Pt> operator^(Ln l){
20         vector<Pt> s;
21         Pt p=l.proj(o);
22         T d=(p-o).norm();
23         if(d-EPS>r)return s;
24         if(abs(d-r)<=EPS){s.pb(p);return s;}
25         d=sqrt(r*r-d*d);
26         s.pb(p+l.pq.unit()*d);
27         s.pb(p-l.pq.unit()*d);
28         return s;
29     }
30     vector<Pt> tang(Pt p){
31         T d=sqrt((p-o).norm2()-r*r);
32         return *this^Circle(p,d);
33     }
34 };

```

```

33     bool in(Circle c){ // non strict
34         T d=(o-c.o).norm();
35         return d+r<=c.r+EPS;
36     }
37     T intertriangle(Pt a, Pt b){ // area of intersection with oab
38         if(abs((o-a)% (o-b))<=EPS) return 0.;
39         vector<Pt> q={a},w=*this^Ln(a,b);
40         if(w.size()==2)for(auto p:w)if((a-p)*(b-p)<-EPS)q.pb(p);
41         q.pb(b);
42         if(q.size()==4&&(q[0]-q[1])*(q[2]-q[1])>EPS)swap(q[1],q[2]);
43         T s=0;
44         for(i,0,q.size()-1){
45             if(!has(q[i])||!has(q[i+1]))s+=r*r*(q[i]-o).angle(q[i+1]-o)/2;
46             else s+=abs((q[i]-o)%(q[i+1]-o)/2);
47         }
48         return s;
49     }
50 };

```

## 4.5. Convex Hull

```

1 // CCW order
2 // Includes collinear points (change sign of EPS in left to
   exclude)
3 vector<Pt> chull(vector<Pt> p){
4     if(sz(p)<3)return p;
5     vector<Pt> r;
6     sort(p.begin(),p.end()); // first x, then y
7     for(i,0,p.size()){ // lower hull
8         while(r.size()>=2&&r.back().left(r[r.size()-2],p[i]))r.pop_back();
9         r.pb(p[i]);
10    }
11    r.pop_back();
12    int k=r.size();
13    for(int i=p.size()-1;i>=0;--i){ // upper hull
14        while(r.size()>=k+2&&r.back().left(r[r.size()-2],p[i]))r.pop_back();
15        r.pb(p[i]);
16    }
17    r.pop_back();
18    return r;

```

19 }

## 4.6. Orden Radial

```

1 struct Radial {
2     Pt o;
3     Radial(Pt _o) : o(_o) {}
4     int cuad(Pt p) {
5         if (p.x>0 && p.y>0) return 1;
6         if (p.x<0 && p.y>0) return 2;
7         if (p.x<0 && p.y<0) return 3;
8         if (p.x>0 && p.y<0) return 4;
9         assert(p.x == 0 && p.y == 0);
10        return 0; // origen < todos
11    }
12    bool comp(Pt p, Pt q) {
13        int c1 = cuad(p), c2 = cuad(q);
14        if (c1 == c2) return p%q>EPS;
15        return c1 < c2;
16    }
17    bool operator()(const Pt &p, const Pt &q) const {
18        return comp(p-o,q-o);
19    }
20};

```

## 4.7. Par de puntos más cercano

```

1 #define dist(a, b) ((a-b).norm_sq())
2 bool sortx(pt a, pt b) {
3     return mp(a.x,a.y)<mp(b.x,b.y); }
4 bool sorte(pt a, pt b) {
5     return mp(a.y,a.x)<mp(b.y,b.x); }
6    closest(vector<pt> &ps, int l, int r) {
7        if (l == r-1) return INF;
8        if (l == r-2) {
9            if (sorty(ps[l+1], ps[l]))
10                swap(ps[l+1], ps[l]);
11            return dist(ps[l], ps[l+1]);
12        }
13        int m = (l+r)/2; if xm = ps[m].x;
14        min_dist = min(closest(ps, l, m), closest(ps, m, r));

```

```

15     vector<pt> left(&ps[1], &ps[m]), right(&ps[m], &ps[r]);
16     merge(all(left), all(right), &ps[1], sorty);
17     ll delta = ll(sqrt(min_dist));
18     vector<pt> strip;
19     forr (i, 1, r) if (ps[i].x>=xm-delta&&ps[i].x<=xm+delta)
20         strip.pb(ps[i]);
21     forn (i, sz(strip)) forr (j, i+1, 8) {
22         if (i+j >= sz(strip)) break;
23         min_dist = min(min_dist, dist(strip[i], strip[i+j]));
24     }
25     return min_dist;
26 }
27 ll closest(vector<pt> &ps) { // devuelve dist ~
28     sort(all(ps), sortx);
29     return closest(ps, 0, sz(ps));
30 }

```

## 4.8. Arbol KD

```

1 // given a set of points, answer queries of nearest point in
   O(log(n))
2 bool onx(pt a, pt b){return a.x<b.x;}
3 bool ony(pt a, pt b){return a.y<b.y;}
4 struct Node {
5     pt pp;
6     ll x0=INF, x1=-INF, y0=INF, y1=-INF;
7     Node *first=0, *second=0;
8     ll distance(pt p){
9         ll x=min(max(x0,p.x),x1);
10        ll y=min(max(y0,p.y),y1);
11        return (pt(x,y)-p).norm2();
12    }
13    Node(vector<pt>&& vp):pp(vp[0]){
14        for(pt p:vp){
15            x0=min(x0,p.x); x1=max(x1,p.x);
16            y0=min(y0,p.y); y1=max(y1,p.y);
17        }
18        if(sz(vp)>1){
19            sort(all(vp),x1-x0>=y1-y0?onx:ony);
20            int m=sz(vp)/2;

```

```

21                first=new Node({vp.begin(),vp.begin()+m});
22                second=new Node({vp.begin()+(m, vp.end())});
23            }
24        }
25    };
26    struct KDTree {
27        Node* root;
28        KDTree(const vector<pt>& vp):root(new Node({all(vp)})) {}
29        pair<ll,pt> search(pt p, Node *node){
30            if(!node->first){
31                //avoid query point as answer
32                //if(p==node->pp) {INF,pt()};
33                return {(p-node->pp).norm2(),node->pp};
34            }
35            Node *f=node->first, *s=node->second;
36            ll bf=f->distance(p), bs=s->distance(p);
37            if(bf>bs)swap(bf,bs),swap(f,s);
38            auto best=search(p,f);
39            if(bs<best.fst) best=min(best,search(p,s));
40            return best;
41        }
42        pair<ll,pt> nearest(pt p){return search(p,root);}
43    };

```

## 4.9. Suma de Minkowski

```

1 vector<Pt> minkowski_sum(vector<Pt> &p, vector<Pt> &q){
2     int n=sz(p),m=sz(q),x=0,y=0;
3     forr(i,0,n) if(p[i]<p[x]) x=i;
4     forr(i,0,m) if(q[i]<q[y]) y=i;
5     vector<Pt> ans={p[x]+q[y]};
6     forr(it,1,n+m){
7         Pt a=p[(x+1)%n]+q[y];
8         Pt b=p[x]+q[(y+1)%m];
9         if(b.left(ans.back(),a)) ans.pb(b), y=(y+1)%m;
10        else ans.pb(a), x=(x+1)%n;
11    }
12    return ans;
13}
14 vector<Pt> do_minkowski(vector<Pt> &p, vector<Pt> &q) {

```

```

15 normalize(p); normalize(q);
16 vector<Pt> sum = minkowski_sum(p, q);
17 return chull(sum); // no normalizado
18 }
19 // escalar poligono
20 vector<Pt> operator*(vector<Pt> &p, td u) {
21     vector<Pt> r; forn (i, sz(p)) r.pb(p[i]*u);
22     return r;
23 }

```

## 5. Strings

### 5.1. Hashing

```

1 struct StrHash { // Hash polinomial con exponentes decrecientes.
2     static constexpr ll ms[] = {1'000'000'007, 1'000'000'403};
3     static constexpr ll b = 500'000'000;
4     vector<ll> hs[2], bs[2];
5     StrHash(string const& s) {
6         int n = sz(s);
7         forn(k, 2) {
8             hs[k].resize(n+1), bs[k].resize(n+1, 1);
9             forn(i, n) {
10                 hs[k][i+1] = (hs[k][i] * b + s[i]) % ms[k];
11                 bs[k][i+1] = bs[k][i] * b % ms[k];
12             }
13         }
14     }
15     ll get(int idx, int len) const { // Hashes en `s[idx, idx+len)`.
16         ll h[2];
17         forn(k, 2) {
18             h[k] = hs[k][idx+len] - hs[k][idx] * bs[k][len] % ms[k];
19             if (h[k] < 0) h[k] += ms[k];
20         }
21         return (h[0] << 32) | h[1];
22     }
23 };

```

### 5.2. Suffix Array

```

1 #define RB(x) ((x) < n ? r[x] : 0)
2 void csort(vector<int>& sa, vector<int>& r, int k) {
3     int n = sz(sa);
4     vector<int> f(max(255, n)), t(n);
5     forn(i, n) ++f[RB(i+k)];
6     int sum = 0;
7     forn(i, max(255, n)) f[i] = (sum += f[i]) - f[i];
8     forn(i, n) t[f[RB(sa[i]+k)]++] = sa[i];
9     sa = t;
10 }
11 vector<int> compute_sa(string& s){ // O(n*log2(n))
12     int n = sz(s) + 1, rank;
13     vector<int> sa(n), r(n), t(n);
14     iota(all(sa), 0);
15     forn(i, n) r[i] = s[i];
16     for (int k = 1; k < n; k *= 2) {
17         csort(sa, r, k), csort(sa, r, 0);
18         t[sa[0]] = rank = 0;
19         forr(i, 1, n) {
20             if(r[sa[i]] != r[sa[i-1]] || RB(sa[i]+k) != RB(sa[i-1]+k)) ++rank;
21             t[sa[i]] = rank;
22         }
23         r = t;
24         if (r[sa[n-1]] == n-1) break;
25     }
26     return sa; // sa[i] = i-th suffix of s in lexicographical order
27 }
28 vector<int> compute_lcp(string& s, vector<int>& sa){
29     int n = sz(s) + 1, L = 0;
30     vector<int> lcp(n), plcp(n), phi(n);
31     phi[sa[0]] = -1;
32     forr(i, 1, n) phi[sa[i]] = sa[i-1];
33     forn(i, n) {
34         if (phi[i] < 0) { plcp[i] = 0; continue; }
35         while(s[i+L] == s[phi[i]+L]) ++L;
36         plcp[i] = L;
37         L = max(L - 1, 0);
38     }
39     forn(i, n) lcp[i] = plcp[sa[i]];

```

```

40     return lcp; // lcp[i] = longest common prefix between sa[i-1]
41     and sa[i]

```

### 5.3. String Functions

```

1 template<class Char=char>vector<int> pfun(basic_string<Char> const&
2   w) {
3     int n = sz(w), j = 0; vector<int> pi(n);
4     forn(i, 1, n) {
5       while (j != 0 && w[i] != w[j]) {j = pi[j - 1];}
6       if (w[i] == w[j]) {++j;}
7       pi[i] = j;
8     } // pi[i] = length of longest proper suffix of w[0..i] that is
      also prefix
9     return pi;
10 }
11 template<class Char=char>vector<int> zfun(const
12   basic_string<Char>& w) {
13   int n = sz(w), l = 0, r = 0; vector<int> z(n);
14   forn(i, 1, n) {
15     if (i <= r) {z[i] = min(r - i + 1, z[i - 1]);}
16     while (i + z[i] < n && w[z[i]] == w[i + z[i]]) {++z[i];}
17     if (i + z[i] - 1 > r) {l = i, r = i + z[i] - 1;}
18   } // z[i] = length of longest prefix of w that also begins at
      index i
19   return z;
20 }

```

### 5.4. Kmp

```

1 template<class Char=char>struct Kmp {
2   using str = basic_string<Char>;
3   vector<int> pi; str pat;
4   Kmp(str const& _pat): pi(move(pfun(_pat))), pat(_pat) {}
5   vector<int> matches(str const& txt) const {
6     if (sz(pat) > sz(txt)) {return {};}
7     vector<int> occs; int m = sz(pat), n = sz(txt);
8     if (m == 0) {occs.push_back(0);}
9     int j = 0;
10    forn(i, n) {

```

```

11      while (j != 0 && txt[i] != pat[j]) {j = pi[j - 1];}
12      if (txt[i] == pat[j]) {++j;}
13      if (j == m) {occs.push_back(i - j + 1);}
14    }
15  }
16 }
17 };

```

### 5.5. Manacher

```

1 struct Manacher {
2   vector<int> p;
3   Manacher(string const& s) {
4     int n = sz(s), m = 2*n+1, l = -1, r = 1;
5     vector<char> t(m); forn(i, n) t[2*i+1] = s[i];
6     p.resize(m); forn(i, 1, m) {
7       if (i < r) p[i] = min(r-i, p[l+r-i]);
8       while (p[i] <= i && i < m-p[i] && t[i-p[i]] ==
9             t[i+p[i]]) ++p[i];
10      if (i+p[i] > r) l = i-p[i], r = i+p[i];
11    }
12  } // Retorna palindromos de la forma {comienzo, largo}.
13  pii at(int i) const {int k = p[i]-1; return pair{i/2-k/2, k};}
14  pii odd(int i) const {return at(2*i+1);} // Mayor centrado en
      s[i].
15  pii even(int i) const {return at(2*i);} // Mayor centrado en
      s[i-1, i].
16 };

```

### 5.6. Mínima Rotación Lexicográfica

```

1 // única secuencia no-creciente de strings menores a sus rotaciones
2 vector<pii> lyndon(string const& s) {
3   vector<pii> fs;
4   int n = sz(s);
5   for (int i = 0, j, k; i < n;) {
6     for (k = i, j = i+1; j < n && s[k] <= s[j]; ++j)
7       if (s[k] < s[j]) k = i; else ++k;
8     for (int m = j-k; i <= k; i += m) fs.emplace_back(i, m);
9   }
10  return fs; // retorna substrings de la forma {comienzo, largo}

```

```

11 }
12
13 // último comienzo de la mínima rotación
14 int minrot(string const& s) {
15     auto fs = lyndon(s+s);
16     int n = sz(s), start = 0;
17     for (auto f : fs) if (f.fst < n) start = f.fst; else break;
18     return start;
19 }

```

## 5.7. Trie

```

1 // trie genérico. si es muy lento, se puede modificar para que los
2 // hijos sean
3 // representados con un array del tamaño del alfabeto
4 template<class Char> struct Trie {
5     struct Node {
6         map<Char, Node*> child;
7         bool term;
8     };
9     Node* root;
10    static inline deque<Node> nodes;
11    static Node* make() {
12        nodes.emplace_back();
13        return &nodes.back();
14    }
15    Trie() : root{make()} {}
16    // retorna el largo del mayor prefijo de s que es prefijo de
17    // algún string
18    // insertado en el trie
19    int find(basic_string<Char> const& s) const {
20        Node* curr = root;
21        for(i,sz(s)) {
22            auto it = curr->child.find(s[i]);
23            if (it == end(curr->child)) return i;
24            curr = it->snd;
25        }
26        return sz(s);
27    }
28    // inserta s en el trie

```

```

27    void insert(basic_string<Char> const& s) {
28        Node* curr = root;
29        for(i,sz(s)) {
30            auto it = curr->child.find(s[i]);
31            if (it == end(curr->child)) curr = curr->child[s[i]] =
32                make();
33            else curr = it->snd;
34        }
35        curr->term = true;
36    }
37    // elimina s del trie
38    void erase(basic_string<Char> const& s) {
39        auto erase = [&](auto&& me, Node* curr, int i) -> bool {
40            if (i == sz(s)) {
41                curr->term = false;
42                return sz(curr->child) == 0;
43            }
44            auto it = curr->child.find(s[i]);
45            if (it == end(curr->child)) return false;
46            if (!me(me,it->snd,i+1)) return false;
47            curr->child.erase(it);
48            return sz(curr->child) == 0;
49        };
50        erase(erase,root,0);
51    };

```

## 5.8. Suffix Automaton

```

1 /// Minimal DFA that accepts all suffixes of a string.
2 /// - Any path starting at `0` forms a substring.
3 /// - Every substring corresponds to a path starting at `0`.
4 /// - Each state corresponds to the set of all substrings that
5 ///   have the same
6 ///   ending positions in the string, that is, each state `u`
7 ///   represents an
8 ///   equivalence class according to their ending positions
9 ///   `endpos(u)`.
```

7 /// Given a state `u`, we can define the following concepts:

8 /// - `longest(u)`: longest substring corresponding to `u`.

```

9  ///`len(u)`: length of `longest(u)`.
10 //`shortest(u)`: shortest substring corresponding to `u`.
11 //`minlen(u)`: length of `shortest(u)`.
12 //Any state `u` corresponds to all suffixes of `longest(u)` no
13 //shorter
14 //than `minlen(u)`.
15 //For state `u`, `link(u)` points to the state `v` such that
16 //`longest(v)` is a suffix of `longest(u)` with `len(v) == minlen(u) - 1`.
17 //These links
18 //form a tree with the root in `0` and an inclusion
19 //relationship between
20 //all `endpos`.
21 template<class Char=char>class SuffixAutomaton {
22     using str = basic_string<Char>;
23     void extend(Char c, int& last) {
24         txt.pb(c); int p = last; last = new_state();
25         len[last] = len[p] + 1, firstpos[last] = len[p];
26         do {next[p][c] = last, p = link[p];} while (p >= 0 &&
27             !next[p].count(c));
28         if (p == -1) {link[last] = 0;} else {
29             int q = next[p][c];
30             if (len[q] == len[p] + 1) {link[last] = q;} else {
31                 int cl = copy_state(q);
32                 len[cl] = len[p] + 1; link[last] = link[q] = cl;
33                 do {next[p][c] = cl, p = link[p];} while (p >= 0 &&
34                     next[p].at(c) == q);
35             }
36         }
37     int new_state() {
38         next.pb({}), link.pb(-1), len.pb(0), firstpos.pb(-1);
39         return size++;
40     }
41     int copy_state(int state) {
42         next.pb(next[state]), link.pb(link[state]);
43         len.pb(len[state]), firstpos.pb(firstpos[state]);
44         return size++;
45     }
46     void dfs(int curr=0) {

```

```

47         terminal_paths_from[curr] = term[curr];
48         paths_from[curr] = 1;
49         fore(edge, next[curr]) {
50             int other = edge.snd;
51             if (!paths_from[other]) {dfs(other);}
52             terminal_paths_from[curr] += terminal_paths_from[other];
53             paths_from[curr] += paths_from[other];
54             substrings_from[curr] += substrings_from[other];
55         }
56         substrings_from[curr] += terminal_paths_from[curr];
57     }
58     void compute(int last) {
59         term.resize(size);
60         for (int curr = last; curr != -1; curr = link[curr])
61             {term[curr] = true;}
62         inv_link.resize(size);
63         forr(curr, 1, size) {inv_link[link[curr]].pb(curr);}
64     }
65     public:
66         vector<bool> term; // Terminal statuses.
67         vector<vector<int>> inv_link; // Inverse suffix links.
68         vector<map<Char, int>> next{{}}; // Automaton transitions.
69         vector<int> len{0}; // len[u] = length of longest(u)
70         vector<int> link{-1}; // Suffix links.
71         vector<int> firstpos{-1}; // First endpos element of each
72         // state.
73         // Number of paths starting at each state and ending in a
74         // terminal state.
75         // For `0`, this is the number of suffixes (including the
76         // empty suffix).
77         vector<int> terminal_paths_from;
78         // Number of paths starting at each state. For `0`, this is
79         // the number of
80         // distinct substrings (including the empty substring).
81         vector<ll> paths_from;
82         // Number of substrings starting at each state. For `0`, this
83         // is the number
84         // of substrings counting repetitions (including the empty
85         // substring
86         // repeated `n+1` times, where `n` is the length of the

```

```

        original string).
76    vector<ll> substrings_from;
77    int size = 1; // Number of states.
78    str txt; // Original string.
79    SuffixAutomaton(str const& _txt) {
80        int last = 0;
81        fore(c, _txt) {extend(c, last);}
82        compute(last); terminal_paths_from.resize(size);
83        paths_from.resize(size); substrings_from.resize(size);
84        dfs();
85    }
86    pair<int, int> run(str const& pat) const {
87        int curr = 0, read = 0; // curr = last visited state
88        for (
89            auto it = pat.begin();
90            it != pat.end() && next[curr].count(*it);
91            curr = next[curr].at(*it++))
92        ) {++read;} // read = number of traversed transitions
93        return {curr, read};
94    }
95    bool is_suff(str const& pat) const
96        {auto [state, read] = run(pat); return term[state] && read
97         == sz(pat);}
98    bool is_substr(str const& pat) const {return run(pat).snd ==
99         sz(pat);}
100   int num_occs(str const& pat) const {
101       auto [state, read] = run(pat);
102       return read == sz(pat) ? terminal_paths_from[state] : 0;
103   }
104   int fst_occ(str const& pat) const {
105       int m = sz(pat); auto [state, read] = run(pat);
106       return read == m ? firstpos[state] + 1 - m : -1;
107   }
108   vector<int> all_occs(str const& pat) const {
109       vector<int> occs; int m = sz(pat); auto [node, read] =
110           run(pat);
111       if (read == m) {
112           stack<int> st{{node}};
113           while (!st.empty()) {
114               int curr = st.top(); st.pop();

```

```

115                   occs.pb(firstpos[curr] + 1 - m);
116                   fore(child, inv_link[curr]) {st.push(child);}
117               }
118           }
119       }
120       // sort(all(occs)); occs.erase(unique(all(occs)),
121       // occs.end());
122       return occs; // unsorted and nonunique by default
123   }
124 };

```

## 6. Grafos

### 6.1. Dikjstra

```

1  vector<pair<int,int>> g[MAXN]; // u->[(v,cost)]
2  ll dist[MAXN];
3  // complejidad O((E+V)*log(V))
4  void dijkstra(int x){
5      memset(dist,-1,sizeof(dist));
6      priority_queue<pair<ll,int> > q;
7      dist[x]=0;q.push({0,x});
8      while(!q.empty()){
9          x=q.top().snd;ll c=-q.top().fst;q.pop();
10         if(dist[x]!=c)continue;
11         forn(i,g[x].size()){
12             int y=g[x][i].fst; ll c=g[x][i].snd;
13             if(dist[y]<0||dist[x]+c<dist[y])
14                 dist[y]=dist[x]+c,q.push({-dist[y],y});
15         }
16     }
17 }

```

### 6.2. LCA

```

1  int n;
2  vector<int> g[MAXN];
3
4  vector<int> depth, etour, vtime;
5
6  // operación de la sparse table, escribir `#define oper lca_oper`
```

```

7 int lca_oper(int u, int v) { return depth[u]<depth[v] ? u : v; }
8
9 void lca_dfs(int u) {
10    vtime[u] = sz(etour), etour.push_back(u);
11    for (auto v : g[u]) {
12        if (vtime[v] >= 0) continue;
13        depth[v] = depth[u]+1; lca_dfs(v); etour.push_back(u);
14    }
15 }
16 auto lca_init(int root) {
17    depth.assign(n,0), etour.clear(), vtime.assign(n,-1);
18    lca_dfs(root); st_init(etour);
19 }
20
21 auto lca(int u, int v) {
22    int l = min(vtime[u],vtime[v]);
23    int r = max(vtime[u],vtime[v])+1;
24    return st_query(l,r);
25 }
26 int dist(int u, int v) { return
27     depth[u]+depth[v]-2*depth[lca(u,v)]; }

```

### 6.3. Binary Lifting

```

1 vector<int> g[1<<K]; int n; // K such that  $2^K \geq n$ 
2 int F[K][1<<K], D[1<<K];
3 void lca_dfs(int x){
4    forn(i, sz(g[x])){
5        int y = g[x][i]; if(y==F[0][x]) continue;
6        F[0][y]=x; D[y]=D[x]+1;lca_dfs(y);
7    }
8 }
9 void lca_init(){
10    D[0]=0;F[0][0]=-1;
11    lca_dfs(0);
12    forall(k,1,K)forn(x,n)
13        if(F[k-1][x]<0)F[k][x]=-1;
14        else F[k][x]=F[k-1][F[k-1][x]];
15 }
16

```

```

17 int lca(int x, int y){
18    if(D[x]<D[y])swap(x,y);
19    for(int k = K-1;k>=0;--k) if(D[x]-(1<<k) >= D[y])x=F[k][x];
20    if(x==y) return x;
21    for(int k=K-1;k>=0;--k)if(F[k][x]!=F[k][y])x=F[k][x],y=F[k][y];
22    return F[0][x];
23 }
24
25 int dist(int x, int y){
26    return D[x] + D[y] - 2*D[lca(x,y)];
27 }

```

### 6.4. Toposort

```

1 vector<int> g[MAXN];int n;
2 vector<int> tsort(){ // lexicographically smallest topological sort
3    vector<int> r;priority_queue<int> q;
4    vector<int> d(2*n,0);
5    forn(i,n)forn(j,g[i].size())d[g[i][j]]++;
6    forn(i,n)if(!d[i])q.push(-i);
7    while(!q.empty()){
8        int x=-q.top();q.pop();r.pb(x);
9        forn(i,sz(g[x])){
10            d[g[x][i]]--;
11            if(!d[g[x][i]])q.push(-g[x][i]);
12        }
13    }
14    return r; // if not DAG it will have less than n elements
15 }

```

### 6.5. Detección ciclos negativos

```

1 // g[i][j]: weight of edge (i, j) or INF if there's no edge
2 // g[i][i]=0
3 ll g[MAXN][MAXN];int n;
4 void floyd(){ // O(n^3). Replaces g with min distances
5    forn(k,n)forn(i,n)if(g[i][k]<INF)forn(j,n)if(g[k][j]<INF)
6        g[i][j]=min(g[i][j],g[i][k]+g[k][j]);
7    }
8 bool inNegCycle(int v){return g[v][v]<0;}

```

```

9 bool hasNegCycle(int a, int b){ // true iff there's neg cycle in
10    between
11    forn(i,n)if(g[a][i]<INF&&g[i][b]<INF&&g[i][i]<0) return true;
12    return false;
13 }

```

## 6.6. Camino Euleriano

```

1 // Directed version (uncomment commented code for undirected)
2 struct edge {
3     int y;
4     // list<edge>::iterator rev;
5     edge(int y):y(y){}
6 };
7 list<edge> g[MAXN];
8 void add_edge(int a, int b){
9     g[a].push_front(edge(b));//auto ia=g[a].begin();
10    // g[b].push_front(edge(a));auto ib=g[b].begin();
11    // ia->rev=ib;ib->rev=ia;
12 }
13 vector<int> p;
14 void go(int x){
15     while(g[x].size()){
16         int y=g[x].front().y;
17         //g[y].erase(g[x].front().rev);
18         g[x].pop_front();
19         go(y);
20     }
21     p.push_back(x);
22 }
23 vector<int> get_path(int x){ // get a path that begins in x
24 // check that a path exists from x before calling to get_path!
25     p.clear();go(x);reverse(p.begin(),p.end());
26     return p;
27 }

```

## 6.7. Camino Hamiltoniano

```

1 constexpr int MAXN = 20;
2 int n;
3 bool adj[MAXN][MAXN];

```

```

4
5 bool seen[1<<MAXN][MAXN];
6 bool memo[1<<MAXN][MAXN];
7 // true sii existe camino simple en el conjunto s que empieza en u
8 bool hamilton(int s, int u) {
9     bool& ans = memo[s][u];
10    if (seen[s][u]) return ans;
11    seen[s][u] = true, s ^= (1<<u);
12    if (s == 0) return ans = true;
13    forn(v,n) if (adj[u][v] && (s&(1<<v)) && hamilton(s,v)) return
14        ans = true;
15    return ans = false;
16 }
17 // true sii existe camino hamiltoniano. complejidad O((1<<n)*n*n)
18 bool hamilton() {
19     forn(s,1<<n) forn(u,n) seen[s][u] = false;
20     forn(u,n) if (hamilton((1<<n)-1,u)) return true;
21     return false;
22 }

```

## 6.8. Tarjan SCC

```

1 vector<int> g[MAXN], ss;
2 int n, num, order[MAXN], lnk[MAXN], nsc, cmp[MAXN];
3 void scc(int u) {
4     order[u] = lnk[u] = ++num;
5     ss.pb(u); cmp[u] = -2;
6     for (auto v : g[u]) {
7         if (order[v] == 0) {
8             scc(v);
9             lnk[u] = min(lnk[u], lnk[v]);
10        }
11        else if (cmp[v] == -2) {
12            lnk[u] = min(lnk[u], lnk[v]);
13        }
14    }
15    if (lnk[u] == order[u]) {
16        int v;
17        do { v = ss.back(); cmp[v] = nsc; ss.pop_back(); }
18        while (v != u);

```

```

19     nsc++;
20 }
21 }
22 void tarjan() {
23     memset(order, 0, sizeof(order)); num = 0;
24     memset(cmp, -1, sizeof(cmp)); nsc = 0;
25     for (i, n) if (order[i] == 0) scc(i);
26 }
```

## 6.9. Bellman-Ford

```

1 const int INF=2e9; int n;
2 vector<pair<int,int>> g[MAXN]; // u->[(v,cost)]
3 ll dist[MAXN];
4 void bford(int src){ // O(nm)
5     fill(dist,dist+n,INF);dist[src]=0;
6     forr(_,0,n)forr(x,0,n)if(dist[x]!=INF)for(auto t:g[x]){
7         dist[t.fst]=min(dist[t.fst],dist[x]+t.snd);
8     }
9     forr(x,0,n)if(dist[x]!=INF)for(auto t:g[x]){
10         if(dist[t.fst]>dist[x]+t.snd){
11             // neg cycle: all nodes reachable from t.fst have
12             // -INF distance
13             // to reconstruct neg cycle: save "prev" of each
14             // node, go up from t.fst until repeating a node.
15             // this node and all nodes between the two
16             // occurrences form a neg cycle
17         }
18     }
19 }
```

## 6.10. Puentes y Articulacion

```

1 // solo para grafos no dirigidos
2 vector<int> g[MAXN];
3 int n, num, root, rootChildren;
4 int order[MAXN], lnk[MAXN], art[MAXN];
5 void bridge_art(int u, int p) {
6     order[u] = lnk[u] = ++num;
7     for (auto v : g[u]) if (v != p) {
8         if (u == root) rootChildren++;
9     }
10 }
```

```

9     if (order[v] == 0) {
10         bridge_art(v, u);
11         if (lnk[v] >= order[u]) // para puntos de
12             art[u] = 1;           // articulacion.
13         if (lnk[v] > order[u]) // para puentes.
14             handle_bridge(u, v);
15     }
16     lnk[u] = min(lnk[u], lnk[v]);
17 }
18 }
19 void run() {
20     memset(order, 0, sizeof(order));
21     memset(art, 0, sizeof(art)); num = 0;
22     forr (i, n) {
23         if (order[i] == 0) {
24             root = i; bridge_art(i, -1);
25             art[i] = (rootChildren > 1);
26         }
27     }
28 }
```

## 6.11. Kruskal

```

1 int uf[MAXN];
2 void uf_init(){memset(uf,-1,sizeof(uf));}
3 int uf_find(int x){return uf[x]<0?x:uf[x]=uf_find(uf[x]);}
4 bool uf_join(int x, int y){
5     x=uf_find(x);y=uf_find(y);
6     if(x==y) return false;
7     if(uf[x]>uf[y]) swap(x,y);
8     uf[x]+=uf[y];uf[y]=x;
9     return true;
10 }
11 vector<pair<ll,pair<int,int>> es; // edges (cost,(u,v))
12 ll kruskal(){ // assumes graph is connected
13     sort(es.begin(),es.end());uf_init();
14     ll r=0;
15     forr(i,0,es.size()){
16         int x=es[i].snd.fst,y=es[i].snd.snd;
17         if(uf_join(x,y))r+=es[i].fst; // (x,y,c) belongs to mst
18     }
19 }
```

```

18     }
19     return r; // total cost
20 }

```

## 6.12. Chequeo Bipartito

```

1 int n;
2 vector<int> g[MAXN];
3
4 bool color[MAXN];
5 bool bicolor() {
6     vector<bool> seen(n);
7     auto dfs = [&](auto&& me, int u, bool c) -> bool {
8         color[u] = c, seen[u] = true;
9         for (int v : g[u]) {
10             if (seen[v] && color[v] == color[u]) return false;
11             if (!seen[v] && !me(me,v,!c)) return false;
12         }
13         return true;
14     };
15     forn(u,n) if (!seen[u] && !dfs(dfs,u,0)) return false;
16     return true;
17 }

```

## 6.13. Centroid Decomposition

```

1 bool vis[MAXN]; //para centroides
2 vector<int> g[MAXN]; int size[MAXN];
3 vector<int> g1[MAXN]; //para centroides
4 void calcsz(int u, int p) {
5     size[u] = 1;
6     for (int v : g[u]) if (v != p && !vis[v]) {
7         calcsz(v, u); size[u] += size[v]; }
8 }
9 int cendfs(int u, int p, int ts) {
10     int maximo = 0, pesado, r;
11     for (int v : g[u]) if (v != p && !vis[v]) {
12         if (maximo < size[v]) {
13             maximo = size[v]; pesado = v; }
14     }
15     if (maximo <= (ts/2)) {

```

```

16         vis[u] = true;
17         for (int v : g[u]) if (!vis[v]) {
18             if (v == p) calcsz(v, u);
19             r = cendfs(v, u, hijos[v]);
20             add_edge(g1, u, r);
21         }
22         r = u;
23     }
24     else r = cendfs(pesado, u, ts);
25     return r;
26 }
27 // euler para responder en el arbol de centroides
28 int te[MAXN], ts[MAXN]; vector<Partial> euler;
29 void do_euler(int u, int p, Partial &p) {
30     te[u] = sz(euler); euler.pb(c);
31     for (int v : g[u]) if (v != p && !vis[v]) {
32         do_euler(v, u, p); } //cambiar p
33     ts[u] = sz(euler);
34 }
35 Sol oncen(int u, int p) {
36     do_euler(u, p, Partial{});
37     vis[u] = true; //no tocar visitados
38     Sol r{};
39     for (int v : g1[u]) if (v != p) {
40         r = max(r, oncen(v, u)); }
41     return r;
42 }

```

## 6.14. HLD

```

1 vector<int> g[MAXN];
2 int wg[MAXN], dad[MAXN], dep[MAXN]; // weight,father,depth
3 void dfs1(int x){
4     wg[x]=1;
5     for(int y:g[x])if(y!=dad[x]){
6         dad[y]=x;dep[y]=dep[x]+1;dfs1(y);
7         wg[x]+=wg[y];
8     }
9 }
10 int curpos, pos[MAXN], head[MAXN];

```

```

11 void hld(int x, int c){
12     if(c<0)c=x;
13     pos[x]=curpos++;head[x]=c;
14     int mx=-1;
15     for(int y:g[x])if(y!=dad[x]&&(mx<0||wg[mx]<wg[y]))mx=y;
16     if(mx>=0)hld(mx,c);
17     for(int y:g[x])if(y!=mx&&y!=dad[x])hld(y,-1);
18 }
19 void hld_init(){dad[0]=-1;dep[0]=0;dfs1(0);curpos=0;hld(0,-1);}
20 int query(int x, int y, RMQ& rmq){
21     int r=neutro; //neutro del rmq
22     while(head[x]!=head[y]){
23         if(dep[head[x]]>dep[head[y]])swap(x,y);
24         r=oper(r,rmq.get(pos[head[y]],pos[y]+1));
25         y=dad[head[y]];
26     }
27     if(dep[x]>dep[y])swap(x,y); // now x is lca
28     r=oper(r,rmq.get(pos[x],pos[y]+1));
29     return r;
30 }
31 // hacer una vez al principio hld_init() después de armar el grafo
32 // en g
33 // para querys pasar los dos nodos del camino y un stree que tiene
34 // en pos[x] el valor del nodo x
35 // for updating: rmq.set(pos[x],v);
36 // queries on edges: - assign values of edges to "child" node ()
37 // ***  

38 //           - change pos[x] to pos[x]+1 in query (line 28)
39 // *** if(dep[u] > dep[v]) rmq.upd(pos[u], w) para cada arista
40 //      (u,v)

```

## 6.15. Max Tree Matching

```

1 int n, r, p[MAXN]; // número de nodos, raíz, y lista de padres
2 vector<int> g[MAXN]; // lista de adyacencia
3
4 int match[MAXN];
5 // encuentra el max matching del árbol. complejidad O(n)
6 int maxmatch() {
7     fill(match,match+n,-1);

```

```

8     int size = 0;
9     auto dfs = [&](auto&& me, int u) -> int {
10         for (auto v : g[u]) if (v != p[u])
11             if (match[u] == me(me,v)) match[u] = v, match[v] = u;
12         size += match[u] >= 0;
13     };
14     dfs(dfs,r);
15     return size;
16 }

```

## 6.16. Min Tree Vertex Cover

```

1 int n, r, p[MAXN]; // número de nodos, raíz, y lista de padres
2 vector<int> g[MAXN]; // lista de adyacencia
3
4 bool cover[MAXN];
5 // encuentra el min vertex cover del árbol. complejidad O(n)
6 int mincover() {
7     fill(cover,cover+n,false);
8     int size = 0;
9     auto dfs = [&](auto&& me, int u) -> bool {
10         for (auto v : g[u]) if (v != p[u] && !me(me,v)) cover[u] =
11             true;
12         size += cover[u];
13     };
14     dfs(dfs,r);
15     return size;
16 }

```

## 6.17. 2-SAT

```

1 struct TwoSatSolver{
2     int n_vars;
3     int n_vertices;
4     vector<vector<int>> adj, adj_t;
5     vector<bool> used;
6     vector<int> order,comp;
7     vector<bool> assignment;
8     TwoSatSolver(int _n_vars) : n_vars(_n_vars),

```

```

9     n_vertices(2*n_vars), adj(n_vertices),
10    adj_t(n_vertices), used(n_vertices),
11    order(), comp(n_vertices, -1), assignment(n_vars){
12    order.reserve(n_vertices);
13 }
14 void dfs1(int v){
15     used[v] = true;
16     for(int u : adj[v]){
17         if(!used[u]) dfs1(u);
18     }
19     order.pb(v);
20 }
21 void dfs2(int v, int c1){
22     comp[v] = c1;
23     for(int u : adj_t[v]){
24         if(comp[u] == -1) dfs2(u, c1);
25     }
26 }
27 bool solve_2SAT(){
28     order.clear();
29     used.assign(n_vertices, false);
30     forn(i, n_vertices){
31         if(!used[i]) dfs1(i);
32     }
33     comp.assign(n_vertices, -1);
34     for(int i = 0, j = 0; i < n_vertices; ++i){
35         int v = order[n_vertices - i - 1];
36         if(comp[v] == -1) dfs2(v, j++);
37     }
38     assignment.assign(n_vars, false);
39     for(int i = 0; i < n_vertices; i+=2){
40         if(comp[i] == comp[i+1]) return false;
41         assignment[i/2] = comp[i] > comp[i+1];
42     }
43     return true;
44 }
45 void add_disjunction(int a, bool na, int b, bool nb){
46     a = 2 * a ^ na;
47     b = 2 * b ^ nb;
48     int neg_a = a ^ 1;

```

```

49     int neg_b = b ^ 1;
50     adj[neg_a].pb(b);
51     adj[neg_b].pb(a);
52     adj_t[b].pb(neg_a);
53     adj_t[a].pb(neg_b);
54 }
55 };

```

## 6.18. K Colas

```

1 const int K=9999; // en general, K = MAX_DIST+1
2 vector<Datos> colas[K];
3 int cola_actual = 0, ult_col = -1;
4 // push toma la dist actual y la siguiente
5 #define push(d,nd,args...)
6     colas[(cola_actual+nd-d)%K].emplace_back(nd, args)
7 #define pop colas[cola_actual].pop_back
8 #define top colas[cola_actual].back
9 // PUSHEAR POSICION INICIAL
10 for (; ; cola_actual = (cola_actual+1)%K) {
11     if (ult_col == cola) break; // dimos la vuelta
12     if (colas[cola_actual].size()) ult_col = cola;
13     while (colas[cola_actual].size()) {
14     }
15 }

```

## 7. Flujo

### 7.1. Dinic

```

1 // complejidad O(V^2*E)
2 struct Dinic{
3     int nodes,src,dst;
4     vector<int> dist,q,work;
5     struct edge {int to,rev,ll f,cap;};
6     vector<vector<edge>> g;
7     Dinic(int x):nodes(x),g(x),dist(x),q(x),work(x){}
8     void add_edge(int s, int t, ll cap){
9         g[s].pb((edge){t,sz(g[t]),0,cap});

```

```

10     g[t].pb((edge){s,sz(g[s])-1,0,0});
11 }
12 bool dinic_bfs(){
13     fill(all(dist),-1);dist[src]=0;
14     int qt=0;q[qt++]=src;
15     for(int qh=0;qh<qt;qh++){
16         int u=q[qh];
17         for(i,sz(g[u])){
18             edge &e=g[u][i];int v=g[u][i].to;
19             if(dist[v]<0&&e.f<e.cap)dist[v]=dist[u]+1,q[qt++]=v;
20         }
21     }
22     return dist[dst]>=0;
23 }
24 ll dinic_dfs(int u, ll f){
25     if(u==dst) return f;
26     for(int &i=work[u];i<sz(g[u]);i++){
27         edge &e=g[u][i];
28         if(e.cap<=e.f) continue;
29         int v=e.to;
30         if(dist[v]==dist[u]+1){
31             ll df=dinic_dfs(v,min(f,e.cap-e.f));
32             if(df>0){e.f+=df;g[v][e.rev].f-=df;return df;}
33         }
34     }
35     return 0;
36 }
37 ll max_flow(int _src, int _dst){
38     src=_src;dst=_dst;
39     ll result=0;
40     while(dinic_bfs()){
41         fill(all(work),0);
42         while(ll delta=dinic_dfs(src,INF))result+=delta;
43     }
44     return result;
45 }
46 };

```

## 7.2. Min Cost Max Flow

```

1  typedef ll tf;
2  typedef ll tc;
3  const tf INFFLOW=1e9;
4  const tc INFCOST=1e9;
5  // complejidad O(V^2*E*log(V))
6  struct MCF{
7      int n;
8      vector<tc> prio, pot; vector<tf> curflow; vector<int>
9          prevedge,prevnode;
10     priority_queue<pair<tc, int>, vector<pair<tc, int>>, 
11         greater<pair<tc, int>>> q;
12     struct edge{int to, rev; tf f, cap; tc cost;};
13     vector<vector<edge>> g;
14     MCF(int
15         n):n(n),prio(n),curflow(n),prevedge(n),prevnode(n),pot(n),g(n){}
16     void add_edge(int s, int t, tf cap, tc cost) {
17         g[s].pb((edge){t,sz(g[t]),0,cap,cost});
18         g[t].pb((edge){s,sz(g[s])-1,0,0,-cost});
19     }
20     pair<tf,tc> get_flow(int s, int t) {
21         tf flow=0; tc flowcost=0;
22         while(1){
23             q.push({0, s});
24             fill(all(prio),INFCOST);
25             prio[s]=0; curflow[s]=INFFLOW;
26             while(!q.empty()) {
27                 auto cur=q.top();
28                 tc d=cur.fst;
29                 int u=cur.snd;
30                 q.pop();
31                 if(d!=prio[u]) continue;
32                 for(int i=0; i<sz(g[u]); ++i) {
33                     edge &e=g[u][i];
34                     int v=e.to;
35                     if(e.cap<=e.f) continue;
36                     tc nprio=prio[u]+e.cost+pot[u]-pot[v];
37                     if(prio[v]>nprio) {
38                         prio[v]=nprio;
39                         q.push({nprio, v});
40                         prevnode[v]=u; prevedge[v]=i;
41                     }
42                 }
43             }
44         }
45     }
46 };

```

```

38         curflow[v]=min(curflow[u], e.cap-e.f);
39     }
40   }
41 }
42 if(prio[t]==INFCOST) break;
43 forr(i,0,n) pot[i]+=prio[i];
44 tf df=min(curflow[t], INFFLOW-flow);
45 flow+=df;
46 for(int v=t; v!=s; v=prevnode[v]) {
47   edge &e=g[prevnode[v]][prevedge[v]];
48   e.f+=df; g[v][e.rev].f-=df;
49   flowcost+=df*e.cost;
50 }
51 }
52 return {flow,flowcost};
53 }
54 };

```

### 7.3. Hopcroft Karp

```

1 int n, m;           // número de nodos en ambas partes
2 vector<int> g[MAXN]; // lista de adyacencia [0,n) -> [0,m)
3
4 int mat[MAXN]; // matching [0,n) -> [0,m)
5 int inv[MAXM]; // matching [0,m) -> [0,n)
6 // encuentra el max matching del grafo bipartito
7 // complejidad  $O(\sqrt{(n+m)*e})$ , donde e es el número de aristas
8 int hopkarp() {
9   fill(mat,mat+n,-1);
10  fill(inv,inv+m,-1);
11  int size = 0;
12  vector<int> d(n);
13  auto bfs = [&] {
14    bool aug = false;
15    queue<int> q;
16    forn(u,n) if (mat[u] < 0) q.push(u); else d[u] = -1;
17    while (!q.empty()) {
18      int u = q.front();
19      q.pop();
20      for (auto v : g[u]) {

```

```

21        if (inv[v] < 0) aug = true;
22        else if (d[inv[v]] < 0) d[inv[v]] = d[u] + 1,
23          q.push(inv[v]);
24      }
25    }
26  }
27  auto dfs = [&](auto&& me, int u) -> bool {
28    for (auto v : g[u]) if (inv[v] < 0) {
29      mat[u] = v, inv[v] = u;
30      return true;
31    }
32    for (auto v : g[u]) if (d[inv[v]] > d[u] && me(me,inv[v])) {
33      mat[u] = v, inv[v] = u;
34      return true;
35    }
36    d[u] = 0;
37    return false;
38  };
39  while (bfs()) forn(u,n) if (mat[u] < 0) size += dfs(dfs,u);
40  return size;
41 }

```

### 7.4. Kuhn

```

1 int n, m;           // número de nodos en ambas partes
2 vector<int> g[MAXN]; // lista de adyacencia [0,n) -> [0,m)
3
4 int mat[MAXN]; // matching [0,n) -> [0,m)
5 int inv[MAXM]; // matching [0,m) -> [0,n)
6 // encuentra el max matching del grafo bipartito
7 // complejidad  $O(n*e)$ , donde e es el número de aristas
8 int kuhn() {
9   fill(mat,mat+n,-1);
10  fill(inv,inv+m,-1);
11  int root, size = 0;
12  vector<int> seen(n,-1);
13  auto dfs = [&](auto&& me, int u) -> bool {
14    seen[u] = root;
15    for (auto v : g[u]) if (inv[v] < 0) {

```

```

16     mat[u] = v, inv[v] = u;
17     return true;
18 }
19 for (auto v : g[u]) if (seen[inv[v]] < root &&
20     me(me,inv[v])) {
21     mat[u] = v, inv[v] = u;
22     return true;
23 }
24 return false;
25 };
26 forn(u,n) size += dfs(dfs,root=u);
27 }

```

## 7.5. Min Vertex Cover Bipartito

```

1 // requisito: max matching bipartito, por defecto Hopcroft-Karp
2
3 vector<bool> cover[2]; // nodos cubiertos en ambas partes
4 // encuentra el min vertex cover del grafo bipartito
5 // misma complejidad que el algoritmo de max matching bipartito
6 // elegido
7 int konig() {
8     cover[0].assign(n,true);
9     cover[1].assign(m,false);
10    int size = hopkarp(); // alternativamente, también funciona
11        con Kuhn
12    auto dfs = [&](auto&& me, int u) -> void {
13        cover[0][u] = false;
14        for (auto v : g[u]) if (!cover[1][v]) {
15            cover[1][v] = true;
16            me(me,inv[v]);
17        }
18    };
19    forn(u,n) if (mat[u] < 0) dfs(dfs,u);
20    return size;
21 }

```

## 7.6. Hungarian

```

1 typedef long double td; typedef vector<int> vi; typedef vector<td>
2 vd;
3 const td INF=1e100; //for maximum set INF to 0, and negate costs
4 bool zero(td x){return fabs(x)<1e-9;} //change to x==0, for ints/ll
5 struct Hungarian{
6     int n; vector<vd> cs; vi L, R;
7     Hungarian(int N, int M):n(max(N,M)),cs(n,vd(n)),L(n),R(n){
8         forr(x,0,N)forr(y,0,M)cs[x][y]=INF;
9     }
10    void set(int x,int y,td c){cs[x][y]=c;}
11    td assign() {
12        int mat = 0; vd ds(n), u(n), v(n); vi dad(n), sn(n);
13        forr(i,0,n)u[i]=*min_element(all(cs[i]));
14        forr(j,0,n){
15            v[j]=cs[0][j]-u[0];
16            forr(i,1,n)v[j]=min(v[j],cs[i][j]-u[i]);
17        }
18        L=R=vi(n, -1);
19        forr(i,0,n)forr(j,0,n) {
20            if(R[j]==-1&&zero(cs[i][j]-u[i]-v[j])){
21                L[i]=j;R[j]=i;mat++;break;
22            } }
23        for(;mat<n;mat++){
24            int s=0, j=0, i;
25            while(L[s] != -1)s++;
26            fill(all(dad),-1);fill(all(sn),0);
27            forr(k,0,n)ds[k]=cs[s][k]-u[s]-v[k];
28            for(;){
29                j = -1;
30                forr(k,0,n)if(!sn[k]&&(j== -1||ds[k]<ds[j]))j=k;
31                sn[j] = 1; i = R[j];
32                if(i == -1) break;
33                forr(k,0,n)if(!sn[k]){
34                    auto new_ds=ds[j]+cs[i][k]-u[i]-v[k];
35                    if(ds[k] > new_ds){ds[k]=new_ds;dad[k]=j;}
36                }
37                forr(k,0,n)if(k!=j&&sn[k]){auto
38                    w=ds[k]-ds[j];v[k]+=w,u[R[k]]-=w;
u[s] += ds[j];

```

```

39     while(dad[j]>=0){int d =
40         dad[j];R[j]=R[d];L[R[j]]=j;j=d;}
41     R[j]=s;L[s]=j;
42 }
43 td value=0;forr(i,0,n)value+=cs[i][L[i]];
44 }
45 };

```

## 8. Optimización

### 8.1. Ternary Search

```

1 // mínimo entero de f en (l,r)
2 ll ternary(auto f, ll l, ll r) {
3     for (ll d = r-l; d > 2; d = r-l) {
4         ll a = l+d/3, b = r-d/3;
5         if (f(a) > f(b)) l = a; else r = b;
6     }
7     return l+1; // retorna un punto, no un resultado de evaluar f
8 }
9
10 // mínimo real de f en (l,r)
11 // para error < EPS, usar iters = log((r-l)/EPS)/log(1.618)
12 double golden(auto f, double l, double r, int iters) {
13     constexpr double ratio = (3-sqrt(5))/2;
14     double x1 = l+(r-l)*ratio, f1 = f(x1);
15     double x2 = r-(r-l)*ratio, f2 = f(x2);
16     while (iters--) {
17         if (f1 > f2) l=x1, x1=x2, f1=f2, x2=r-(r-l)*ratio, f2=f(x2);
18         else r=x2, x2=x1, f2=f1, x1=l+(r-l)*ratio, f1=f(x1);
19     }
20     return (l+r)/2; // retorna un punto, no un resultado de
21     evaluar f

```

### 8.2. Longest Increasing Subsequence

```

1 // subsecuencia creciente más larga
2 // para no decreciente, borrar la línea 9 con el continue

```

```

3 template<class Type> vector<int> lis(vector<Type>& a) {
4     int n = sz(a);
5     vector<int> seq, prev(n,-1), idx(n+1,-1);
6     vector<Type> dp(n+1,INF); dp[0] = -INF;
7     forn(i,n) {
8         int l = int(upper_bound(all(dp),a[i])-begin(dp));
9         if (dp[l-1] == a[i]) continue;
10        prev[i] = idx[l-1], idx[l] = i, dp[l] = a[i];
11    }
12    dforn(i,n+1) {
13        if (dp[i] < INF) {
14            for (int k = idx[i]; k >= 0; k = prev[k]) seq.pb(k);
15            reverse(all(seq));
16            break;
17        }
18    }
19    return seq;
20 }

```

## 9. Otros

### 9.1. Mo

```

1 int n,sq,nq; // array size, sqrt(array size), #queries
2 struct qu{int l,r,id;};
3 qu qs[MAXN];
4 ll ans[MAXN]; // ans[i] = answer to ith query
5 bool qcomp(const qu &a, const qu &b){
6     if(a.l/sq!=b.l/sq) return a.l<b.l;
7     return (a.l/sq)&1?a.r<b.r:a.r>b.r;
8 }
9 void mos(){
10    forn(i,nq)qs[i].id=i;
11    sq=sqrt(n)+.5;
12    sort(qs,qs+nq,qcomp);
13    int l=0,r=0;
14    init();
15    forn(i,nq){
16        qu q=qs[i];
17        while(l>q.l)add(--l);

```

```

18     while(r<q.r)add(r++);
19     while(l<q.l)remove(l++);
20     while(r>q.r)remove(--r);
21     ans[q.id]=get_ans();
22 }
23 }
```

## 9.2. Divide and Conquer Optimization

```

1 vector<ll> dp_ant, dp_curr;
2
3 void compute(int l, int r, int optl, int optr){
4     if(l == r) return;
5     int m = (l+r)/2;
6     ll dpm = 1e17;
7     int optm = -1;
8     forr(i, max(m+1, optl), optr+1){
9         ll cost = C(m, i) + (i == n ? 0 : dp_ant[i]);
10        if(cost < dpm) dpm = cost, optm = i;
11    }
12    dp_curr[m] = dpm;
13    compute(l, m, optl, optm);
14    compute(m+1, r, optm, optr);
15 }
16
17 for(i, k){
18     compute(0, n, 0, n);
19     dp_ant = dp_curr;
20 }
21 cout << dp_curr[0] << endl;
```

## 9.3. Fijar el numero de decimales

```

1 // antes de imprimir decimales, con una sola vez basta
2 cout << fixed << setprecision(DECIMAL_DIG);
```

## 9.4. Hash Table (Unordered Map/ Unordered Set)

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 using namespace __gnu_pbds;
```

```

3 template<class Key, class Val=null_type>using
4     htable=gp_hash_table<Key,Val>;
4 // como unordered_map (o unordered_set si Val es vacio), pero sin
5     metodo count
```

## 9.5. Indexed Set

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 using namespace __gnu_pbds;
3 template<class Key, class Val=null_type>
4 using indexed_set = tree<Key, Val, less<Key>, rb_tree_tag,
5                             tree_order_statistics_node_update>;
6 // indexed_set<char> s;
7 // char val = *s.find_by_order(0); // acceso por indice
8 // int idx = s.order_of_key('a'); // busca indice del valor
```

## 9.6. Subconjuntos

```

1 // iterar por mascaras O(2^n)
2 for(int bm=0; bm<(1<<n); bm++)
3     // subconjuntos de una mascara O(2^n)
4     for(int sbm=bm; sbm; sbm=(sbm-1)&bm)
5         // iterar por submascaras O(3^n)
6         for(int bm=0; bm<(1<<n); bm++)
7             for(int sbm=bm; sbm; sbm=(sbm-1)&(bm))
8                 // para superconjuntos (que contienen a bm),
9                 // negar la mascara: bm=~bm
```

## 9.7. Simpson

```

1 // integra f en [a,b] llamándola 2*n veces
2 double simpson(double f, double a, double b, int n=1e4) {
3     double h = (b-a)/2/n, s = f(a);
4     forr(i,1,2*n) s += f(a+i*h) * ((i%2)?4:2);
5     return (s+f(b))*h/3;
6 }
```

## 9.8. Pragmas

```

1 #pragma GCC target("avx2")
2 #pragma GCC optimize("O3")
3 #pragma GCC optimize("unroll-loops")
```

## 9.9. Random

```
1 unsigned seed =
2     std::chrono::steady_clock::now().time_since_epoch().count();
3 mt19937 generator(seed);
4 generator(); // generar un número aleatorio entre 0 y 4294967295
5 // existe mt19937_64 para la versión de 64 bits, que probablemente
6 // sea más rápido
7 /*
8 // también se puede hacer lo siguiente para una versión hasta 3x
9 // más rápida:
10 #include <ext/random>
11 using namespace __gnu_cxx;
12 unsigned seed =
13     std::chrono::steady_clock::now().time_since_epoch().count();
14 sfmt19937 generator(seed); // existe también sfmt19937_64
15 */
16
17 uniform_int_distribution<ll> dist_int(L, R);
18 dist_int(generator); // generar un entero en [L, R]
19 // (cerrado-cerrado) con prob uniforme
20
21 uniform_real_distribution<double> dist_real(0.0, 1.0);
22 dist_real(generator); // generar un real en [0, 1]
23 // (cerrado-abierto) con prob uniforme
```

## 9.10. Utilidades de strings

```
1 getline(cin, linea); // tomar toda la linea
2 stringstream ss(linea); // tratar una linea como stream
3 ss >> s; ss << s; // leer solo hasta un espacio, escribir a ss
4 tipo n; ss >> n; // leer de un stringstream (float, int, etc.)
5 int pos = s.find_first_of("aeiou"); // devuelve -1 si no encuentra
6 int next = s.find_first_of("aeiou", pos);
7 // s.find_first_not_of("aeiou"); s.find_last_of();
8 s.substr(pos, next-pos); // substr(pos, len)
9 s.c_str(); // devuelve un puntero de C
10 ss.str(); // devuelve el string en ss
11 // isspace(); islower(); isupper(); isdigit(); isalpha();
12 // tolower(); toupper();
```