

1. Estructuras

1.1. Segment Tree

```

1 struct Max { // op = max, neutral = -INF
2     int x; Max(int _x = -INF) { x = _x; }
3     Max operator+(const Max &o) { return x > o.x ? *this : o; }
4 };
5 template<class T>
6 struct RMQ { // ops O(lg n), [0, n)
7     vector<T> t; int n;
8     T& operator[](int p) { return t[p+n]; }
9     RMQ(int sz) { n = 1 << (32-__builtin_clz(sz)), t.resize(2*n); }
10    void build() { dforn(i, n) t[i] = t[2*i] + t[2*i+1]; }
11    T get(int i, int j) { return get(i, j, 1, 0, n); }
12    T get(int i, int j, int x, int a, int b) {
13        if (j <= a || i >= b) return T();
14        if (i <= a && b <= j) return t[x];
15        int c = (a + b) / 2;
16        return get(i, j, 2*x, a, c) + get(i, j, 2*x+1, c, b);
17    }
18    void set(int p, T v) {
19        for (t[p += n] = v; p /= 2;) t[p] = t[2*p] + t[2*p+1];
20    }
21 };
22 // Use: RMQ<Max> rmq(n); forn(i, n) { int x; cin >> x; rmq[i] = x; } rmq
    .build();

```

1.2. Segment Tree (Lazy)

```

1 struct Lazy {
2     static const int C = 0; // Neutral for sum: 0
3     int val; Lazy(int v=C) : val(v) {}
4     bool dirty() { return val != C; }
5     void clear() { val = C; }
6     void update(const Lazy &o) { val += o.val; } // Update: sum
7 };
8 struct Node {
9     int val; Node(int v=INF) : val(v) {} // Neutral for min: INF
10    Node operator+(const Node &o) { return min(val, o.val); } // Query:
        min
11    void update(const Lazy &o, int sz) { val += o.val; } // Update: sum
12 };

```

```

13 template <class T, class D>
14 struct RMQ { // ops O(lg n), [0, n)
15     vector<T> t; vector<D> d; int n;
16     T& operator[](int p){ return t[p+n]; }
17     RMQ(int sz) {
18         n = 1; while (n < sz) n *= 2;
19         t.resize(2*n), d.resize(2*n);
20     }
21     void build() { dforsn(i, 1, n) t[i] = t[2*i] + t[2*i + 1]; }
22     void push(int x, int sz) {
23         if (d[x].dirty()){
24             t[x].update(d[x], sz);
25             if (sz > 1) d[2*x].update(d[x]), d[2*x + 1].update(d[x]);
26             d[x].clear();
27         }
28     }
29     T get(int i, int j) { return get(i, j, 1, 0, n); }
30     T get(int i, int j, int x, int a, int b) {
31         if (j <= a || i >= b) return T();
32         push(x, b-a);
33         if (i <= a && b <= j) return t[x];
34         int c = (a + b) / 2;
35         return get(i, j, 2*x, a, c) + get(i, j, 2*x + 1, c, b);
36     }
37     void update(int i, int j, const D &v) { update(i, j, v, 1, 0, n); }
38     void update(int i, int j, const D &v, int x, int a, int b) {
39         push(x, b-a);
40         if (j <= a || i >= b) return;
41         if (i <= a && b <= j) { d[x].update(v), push(x, b-a); return; }
42         int c = (a + b) / 2;
43         update(i, j, v, 2*x, a, c), update(i, j, v, 2*x + 1, c, b);
44         t[x] = t[2*x] + t[2*x + 1];
45     }
46 };
47 // Use: RMQ<Node, Lazy> rmq(n); forn(i, n) cin >> rmq[i].val; rmq.build
    ();

```

1.3. Fenwick Tree

```

1 // Para 2D: tratar cada columna como un Fenwick Tree,
2 // agregando un for anidado en cada operacion.
3 // Trucos:
4 // - La operacion puede no tener inverso ;)

```

```

5 // - Podemos usar unordered_map si tenemos que trabajar con numeros
   grandes
6
7 // Point update, range query:
8 template<class T>
9 struct Fenwick { // ops O(lg n), [0, n)
10     int n, h; vector<T> d;
11     Fenwick(int sz) { n = sz, d.resize(n+1), h = 1 << int(log2(n)); }
12     void upd(int i, T x) { for (++i; i <= n; i += i&-i) d[i] += x; }
13     T get(int i) { T r = 0; for (; i; i -= i&-i) r += d[i]; return r; }
14     T get(int l, int r) { return get(r) - get(l); }
15     int lower_bound(T v) {
16         int x = 0;
17         for (int p = h; p; p >>= 1)
18             if ((x|p) <= n && d[x|p] < v) v -= d[x | p];
19         return x;
20     }
21 };
22
23 // Range update, point query:
24 template<class T>
25 struct Fenwick { // ops O(lg n), [0, n)
26     vector<T> d; int n; Fenwick(int sz) { n=sz, d.resize(n+1); }
27     void upd(int l, int r, T x) { _upd(l, x), _upd(r, -x); }
28     void _upd(int i, T x) { for (++i; i <= n; i += i&-i) d[i] += x; }
29     T get(int i) { T r = 0; for (++i; i; i -= i&-i) r += d[i]; return r; }
30 };
31
32 // Range update, range query:
33 template<class T>
34 struct Fenwick { // ops O(lg n), [0, n)
35     int n; vector<T> m, a;
36     Fenwick(int sz) { n = sz, m.resize(n+1), a.resize(n+1); }
37     void upd(int l, int r, T x) {
38         _upd(l, x, -x*1), _upd(r-1, -x, x*r);
39     }
40     void _upd(int i, T x, T y) {
41         for (++i; i <= n; i += i&-i) m[i] += x, a[i] += y;
42     }
43     T get(int i) {
44         T x = 0, y = 0, s = i;
45         for (; i; i -= i&-i) x += m[i], y += a[i];

```

```

46         return x*s + y;
47     }
48     T get(int l, int r) { return get(r) - get(l); }
49 };

```

1.4. Treap

```

1 typedef pii Value; // pii(val, id)
2 typedef struct node *pnode;
3 struct node {
4     Value val, mini;
5     int dirty;
6     int prior, size;
7     pnode l, r, parent;
8     node(Value val):val(val), mini(val), dirty(0), prior(rand()), size
9         (1), l(0), r(0), parent(0) {} // usar rand piola
10 };
11
12 void push(pnode p){ // propagar dirty a los hijos (aca para lazy)
13     p->val.first += p->dirty;
14     p->mini.first += p->dirty;
15     if(p->l) p->l->dirty += p->dirty;
16     if(p->r) p->r->dirty += p->dirty;
17     p->dirty = 0;
18 }
19 static int size(pnode p){ return p ? p->size : 0; }
20 static Value mini(pnode p){ return p ? push(p), p->mini : pii(1e9, -1); }
21
22 // Update function and size from children's Value
23 void pull(pnode p){ // recalculer valor del nodo aca (para rmq)
24     p->size = 1 + size(p->l) + size(p->r);
25     p->mini = min(min(p->val, mini(p->l)), mini(p->r)); //operacion del
26     rmq!
27     p->parent = 0;
28     if(p->l) p->l->parent = p;
29     if(p->r) p->r->parent = p;
30 }
31
32 //junta dos arreglos
33 pnode merge(pnode l, pnode r){
34     if(!l || !r) return l ? l : r;
35     push(l), push(r);
36     pnode t;

```

```

34
35     if(l->prior < r->prior) l->r=merge(l->r, r), t = l;
36     else r->l=merge(l, r->l), t = r;
37
38     pull(t);
39     return t;
40 }
41
42 //parte el arreglo en dos, si(l)==tam
43 void split(pnode t, int tam, pnode &l, pnode &r){
44     if(!t) return void(l = r = 0);
45     push(t);
46
47     if(tam <= size(t->l)) split(t->l, tam, l, t->l), r = t;
48     else split(t->r, tam - 1 - size(t->l), t->r, r), l = t;
49
50     pull(t);
51 }
52
53 pnode at(pnode t, int pos){
54     if(!t) exit(1);
55     push(t);
56
57     if(pos == size(t->l)) return t;
58     if(pos < size(t->l)) return at(t->l, pos);
59
60     return at(t->r, pos - 1 - size(t->l));
61 }
62 int getpos(pnode t){ // inversa de at
63     if(!t->parent) return size(t->l);
64
65     if(t == t->parent->l) return getpos(t->parent) - size(t->r) - 1;
66
67     return getpos(t->parent) + size(t->l) + 1;
68 }
69
70 void split(pnode t, int i, int j, pnode &l, pnode &m, pnode &r){
71     split(t, i, l, t), split(t, j-i, m, r);
72 }
73 Value get(pnode &p, int i, int j){ // like rmq
74     pnode l, m, r;
75
76     split(p, i, j, l, m, r);

```

```

77     Value ret = mini(m);
78     p = merge(l, merge(m, r));
79
80     return ret;
81 }
82
83 void print(const pnode &t){ // for debugging
84     if(!t) return;
85     push(t);
86     print(t->l);
87     cout << t->val.first << '␣';
88     print(t->r);
89 }

```

1.5. Convex Hull Trick

```

1  /* Restricciones: Asume que las pendientes estan de mayor a menor
2  para calcular minimo o de menor a mayor para calcular maximo, sino
3  usar CHT online o Li-Chao Tree. Si puede haber pendientes iguales
4  agregar if y dejar la que tiene menor (mayor) termino independiente
5  para minimo (maximo). Asume que los puntos a evaluar se encuentran
6  de menor a mayor, sino hacer bb en la hull y encontrar primera
7  recta con Line.i >= x (lower_bound(x)). Si las rectas usan valores
8  reales cambiar div por a/b y las comparaciones para que use EPS.
9  Complejidad: Operaciones en O(1) amortizado. */
10 struct Line { ll a, b, i; };
11 struct CHT : vector<Line> {
12     int p = 0; // pointer to lower_bound(x)
13     ll div(ll a, ll b) { return a/b - ((a^b) < 0 && a % b); } // floor(a
14     /b)
15     void add(ll a, ll b) { // ax + b = 0
16         while (size() > 1 && div(b - back().b, back().a - a)
17             <= at(size()-2).i) pop_back();
18         if (!empty()) back().i = div(b - back().b, back().a - a);
19         pb(Line{a, b, INF});
20         if (p >= si(*this)) p = si(*this)-1;
21     }
22     ll eval(ll x) {
23         while (at(p).i < x) p++;
24         return at(p).a * x + at(p).b;
25     };

```

1.6. CHULL Trick (Dynamic)

```

1 // Default is max, change a,b to -a,-b and negate the result for min
2 // If the lines use real vals change div by a/b and the comparisons
3 struct Line {
4     ll a, b; mutable ll p;
5     bool operator<(const Line& o) const { return a < o.a; }
6     bool operator<(ll x) const { return p < x; }
7 };
8 struct CHT : multiset<Line, less<>> {
9     ll div(ll a, ll b) { return a/b - ((a^b) < 0 && a % b); } // floor(a
10     /b)
11     bool isect(iterator x, iterator y) {
12         if (y == end()) return x->p = INF, false;
13         if (x->a == y->a) x->p = x->b > y->b ? INF : -INF;
14         else x->p = div(y->b - x->b, x->a - y->a);
15         return x->p >= y->p;
16     }
17     void add(ll a, ll b) {
18         auto z = insert({a, b, 0}), y = z++, x = y;
19         while (isect(y, z)) z = erase(z);
20         if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
21         while ((y = x) != begin() && (--x)->p >= y->p) isect(x, erase(y));
22     }
23     ll eval(ll x) {
24         auto l = *lower_bound(x);
25         return l.a * x + l.b;
26     }
27 };

```

1.7. Set con índices

```

1 #include <ext/pb_ds/assoc_container.hpp>
2 using namespace __gnu_pbds; // key, mapped, comp
3 using OrderTree = tree<int, null_type, less<int>,
4 rb_tree_tag, tree_order_statistics_node_update>;
5 // use STL methods like: insert, erase, etc
6 // find_by_order(k): iterator to k-th element
7 // order_of_key(x): index of lower bound of x
8 // to use it as multiset use pair<key, timestamp>

```

2. Strings

2.1. Hash

```

1 mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
2 struct BasicHashing {
3     int mod, base; vi h, pot;
4     BasicHashing() {
5         mod = uniform_int_distribution<>(int(1e9), int(15e8))(rnd);
6         bool prime;
7         do {
8             mod++, prime = true;
9             for (ll d = 2; prime && d*d <= mod; ++d)
10                 if (mod % d == 0) prime = false;
11         } while (!prime);
12         base = uniform_int_distribution<>(256, mod-1)(rnd);
13     }
14     void process(const string &s) {
15         int n = si(s); h = vi(n+1), pot = vi(n+1);
16         h[0] = 0; forn(i, n) h[i+1] = int((h[i] * ll(base) + s[i]) % mod);
17         pot[0] = 1; forn(i, n) pot[i+1] = int(pot[i] * ll(base) % mod);
18     }
19     int hash(int i, int j) { // [ )
20         int res = int(h[j] - ll(h[i]) * pot[j-i] % mod);
21         return res < 0 ? res + mod : res;
22     }
23     int hash(const string &s) {
24         int res = 0;
25         for (char c : s) res = int((res * ll(base) + c) % mod);
26         return res;
27     }
28     int append(int a, int b, int szb) {
29         return int((ll(a) * pot[szb] + b) % mod);
30     }
31 };
32 struct Hashing {
33     BasicHashing h1, h2;
34     void process(const string &s) { h1.process(s), h2.process(s); }
35     pii hash(int i, int j) { return {h1.hash(i, j), h2.hash(i, j)}; }
36     pii hash(const string &s) { return {h1.hash(s), h2.hash(s)}; }
37     pii append(pii &a, pii &b, int szb) {
38         return {h1.append(a.fst, b.fst, szb), h2.append(a.snd, b.snd,

```

```

        szb});
39     }
40 };

```

2.2. Manacher

```

1 void manacher(string s, vi &odd, vi &even) {
2     int n = si(s);
3     s = "@" + s + "$";
4     odd = vi(n), even = vi(n);
5     int l = 0, r = -1;
6     forn(i, n) {
7         int k = i > r ? 1 : min(odd[l+r-i], r-i+1);
8         while (s[i+1-k] == s[i+1+k]) k++;
9         odd[i] = k--;
10        if (i+k > r) l = i-k, r = i+k;
11    }
12    l = 0, r = -1;
13    forn(i, n) {
14        int k = i > r ? 0 : min(even[l+r-i+1], r-i+1);
15        while (s[i-k] == s[i+1+k]) k++;
16        even[i] = k--;
17        if (i+k > r) l = i-k-1, r = i+k;
18    }
19 }

```

2.3. KMP

```

1 // pre[i] = max border of s[0..i]
2 vi prefix_function(string &s) {
3     int n = si(s), j = 0; vi pre(n);
4     forsn(i, 1, n) {
5         while (j > 0 && s[i] != s[j]) j = pre[j-1];
6         pre[i] = s[i] == s[j] ? ++j : j;
7     }
8     return pre;
9 }
10
11 vi find_occurrences(string &s, string &t) { // occurrences of s in t
12     vi pre = prefix_function(s), res;
13     int n = si(s), m = si(t), j = 0;
14     forn(i, m) {
15         while (j > 0 && t[i] != s[j]) j = pre[j-1];
16         if (t[i] == s[j]) j++;

```

```

17         if (j == n) res.pb(i-n+1), j = pre[j-1];
18     }
19     return res;
20 }
21
22 // (i chars match, next_char = c) -> (aut[i][c] chars match)
23 vector<vi> kmp_automaton(string &s) {
24     s += '#'; int n = si(s);
25     vi pre = prefix_function(s);
26     vector<vi> aut(n, vi(26)); // alphabet = lowercase letters
27     forn(i, n) forn(c, 26) {
28         if (i > 0 && 'a' + c != s[i])
29             aut[i][c] = aut[pre[i-1]][c];
30         else
31             aut[i][c] = i + ('a' + c == s[i]);
32     }
33     return aut;
34 }

```

2.4. Suffix Array (corto, nlog2n)

```

1 const int MAXN = 2e5+10;
2 pii sf[MAXN];
3 bool comp(int lhs, int rhs) {return sf[lhs] < sf[rhs];}
4 struct SuffixArray {
5     //sa guarda los indices de los sufijos ordenados
6     int sa[MAXN], r[MAXN];
7     void init(const string &a) {
8         int n = si(a);
9         forn(i,n) r[i] = a[i];
10        for(int m = 1; m < n; m <= 1) {
11            forn(i, n) sa[i]=i, sf[i] = mp(r[i], i+m<n? r[i+m]:-1);
12            stable_sort(sa, sa+n, comp);
13            r[sa[0]] = 0;
14            forsn(i, 1, n) r[sa[i]]= sf[sa[i]] != sf[sa[i - 1]] ? i : r[
15                sa[i-1]];
16        }
17    }
18 } sa;
19
20 int main(){
21     string in;
22     while(cin >> in){

```

```

22     sa.init(in, si(in));
23     forn(i, si(in)) {
24         forn(k, sa.sa[i]) cout << '␣';
25         cout << in.substr(sa.sa[i]) << '\n';
26     }
27     cout << endl;
28 }
29 return 0;
30 }
    
```

2.5. Matching Suffix Array

```

1 //returns (lowerbound, upperbound) of the search
2 pii stringMatching(string P){ //O(si(P)lgn)
3     int lo=0, hi=n-1, mid=lo;
4     while(lo<hi){
5         mid=(lo+hi)/2;
6         int res=s.compare(sa[mid], si(P), P);
7         if(res>=0) hi=mid;
8         else lo=mid+1;
9     }
10    if(s.compare(sa[lo], si(P), P)!=0) return pii(-1, -1);
11    pii ans; ans.first=lo;
12    lo=0, hi=n-1, mid;
13    while(lo<hi){
14        mid=(lo+hi)/2;
15        int res=s.compare(sa[mid], si(P), P);
16        if(res>0) hi=mid;
17        else lo=mid+1;
18    }
19    if(s.compare(sa[hi], si(P), P)!=0) hi--;
20    // para verdadero upperbound sumar 1
21    ans.second=hi;
22    return ans;
    
```

2.6. LCP (Longest Common Prefix)

```

1 //Calculates the LCP between consecutives suffixes in the Suffix Array.
2 //LCP[i] is the length of the LCP between sa[i] and sa[i-1]
3 int LCP[MAXN], phi[MAXN], PLCP[MAXN];
4 void computeLCP(){//O(n)
5     phi[sa[0]]=-1;
6     forsn(i,1,n) phi[sa[i]]=sa[i-1];
    
```

```

8     int L=0;
9     forn(i,n){
10         if (phi[i]==-1) {PLCP[i]=0; continue;}
11         while (s[i+L]==s[phi[i]+L]) L++;
12         PLCP[i]=L;
13         L=max(L-1, 0);
14     }
15     forn(i,n) LCP[i]=PLCP[sa[i]];
    
```

2.7. Aho-Corasick

```

1 const int K = 26;
2
3 // si el alfabeto es muy grande, adaptar usando map para next y go
4 // es posible almacenar los indices de las palabras en terminal usando
5     vector<int>
6 struct Vertex {
7     int next[K];
8     int terminal = 0;
9     int p = -1;
10    char pch;
11    int link = -1;
12    int go[K];
13
14    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
15        fill(begin(next), end(next), -1);
16        fill(begin(go), end(go), -1);
17    }
18 };
19 vector<Vertex> t;
20
21 void aho_init() { // INICIALIZAR!
22     t.clear(); t.pb(Vertex());
23 }
24
25 void add_string(string const& s) {
26     int v = 0;
27     for (char ch : s) {
28         int c = ch - 'a';
29         if (t[v].next[c] == -1) {
30             t[v].next[c] = si(t);
31             t.pb(v, ch);
    
```

```

32     }
33     v = t[v].next[c];
34 }
35 t[v].terminal++;
36 }
37
38 int go(int v, char ch);
39
40 int get_link(int v) {
41     if (t[v].link == -1) {
42         if (v == 0 || t[v].p == 0)
43             t[v].link = 0;
44         else
45             t[v].link = go(get_link(t[v].p), t[v].pch);
46     }
47     return t[v].link;
48 }
49
50 int go(int v, char ch) {
51     int c = ch - 'a';
52     if (t[v].go[c] == -1) {
53         if (t[v].next[c] != -1)
54             t[v].go[c] = t[v].next[c];
55         else
56             t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
57     }
58     return t[v].go[c];
59 }

```

2.8. Suffix Automaton

```

1 struct state {
2     int len, link;
3     map<char,int> next;
4     state() { }
5 };
6 const int MAXLEN = 1e5+10;
7 state st[MAXLEN*2];
8 int sz, last;
9 void sa_init() {
10     forn(i,sz) st[i].next.clear();
11     sz = last = 0;
12     st[0].len = 0;

```

```

13     st[0].link = -1;
14     ++sz;
15 }
16 // Es un DAG de una sola fuente y una sola hoja
17 // cantidad de endpos = cantidad de apariciones = cantidad de caminos de
    la clase al nodo terminal
18 // cantidad de miembros de la clase = st[v].len-st[st[v].link].len (v>0)
    = caminos del inicio a la clase
19 // El arbol de los suffix links es el suffix tree de la cadena invertida
    . La string de la arista link(v)->v son los caracteres que difieren
20 void sa_extend(char c) {
21     int cur = sz++;
22     st[cur].len = st[last].len + 1;
23     // en cur agregamos la posicion que estamos extendiendo
24     // podria agregar tambien un identificador de las cadenas a las cuales
    pertenece (si hay varias)
25     int p;
26     for (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link) // modificar
    esta linea para hacer separadores unicos entre varias cadenas (c
    ==','$')
27     st[p].next[c] = cur;
28     if (p == -1)
29         st[cur].link = 0;
30     else {
31         int q = st[p].next[c];
32         if (st[p].len + 1 == st[q].len)
33             st[cur].link = q;
34         else {
35             int clone = sz++;
36             st[clone].len = st[p].len + 1;
37             st[clone].next = st[q].next;
38             st[clone].link = st[q].link;
39             for (; p!=-1 && st[p].next.count(c) && st[p].next[c]==q; p=st[p].
    link)
40                 st[p].next[c] = clone;
41             st[q].link = st[cur].link = clone;
42         }
43     }
44     last = cur;
45 }

```

2.9. Z Function


```

1 int z[N]; // z[i] = i==0 ? 0 : max k tq s[0,k] match with s[i,i+k]
2 void z_function(string &s, int z[]) {
3     int n = si(s);
4     forn(i,n) z[i]=0;
5     for (int i = 1, l = 0, r = 0; i < n; ++i) {
6         if (i <= r) z[i] = min (r - i + 1, z[i - l]);
7         while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
8         if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
9     }
10 }

```

3. Geometría

3.1. Point

```

1 const double EPS = 1e-9;
2 struct Point {
3     double x, y;
4     Point(double _x=0, double _y=0) : x(_x),y(_y) {}
5     Point operator+(Point a) { return Point(x + a.x, y + a.y); }
6     Point operator-(Point a) { return Point(x - a.x, y - a.y); }
7     Point operator+(double a) { return Point(x + a, y + a); }
8     Point operator*(double a) { return Point(x*a, y*a); }
9     Point operator/(double a) { return Point(x/a, y/a); }
10    double norm() { return sqrt(x*x + y*y); }
11    double norm2() { return x*x + y*y; }
12    // Dot product:
13    double operator*(Point a){ return x*a.x + y*a.y; }
14    // Magnitude of the cross product (if a is less than 180 CW from b, a^
15    b > 0):
16    double operator^(Point a) { return x*a.y - y*a.x; }
17    // Returns true if this point is at the left side of line qr:
18    bool left(Point q, Point r) { return ((q - *this) ^ (r - *this)) > EPS
19    ; }
20    bool operator<(const Point &a) const {
21        return x < a.x - EPS || (abs(x - a.x) < EPS && y < a.y - EPS);
22    }
23    bool operator==(Point a) {
24        return abs(x - a.x) < EPS && abs(y - a.y) < EPS;
25    }
26 };
27 typedef Point vec;
28 double dist(Point a, Point b) { return (b-a).norm(); }

```

```

27 double dist2(Point a, Point b) { return (b-a).norm2(); }
28 double angle(Point a, Point o, Point b){ // [-pi, pi]
29     Point oa = a-o, ob = b-o;
30     return atan2(oa^ob, oa*ob);
31 }
32 // Rotate around the origin:
33 Point CCW90(Point p) { return Point(-p.y, p.x); }
34 Point CW90(Point p) { return Point(p.y, -p.x); }
35 Point CCW(Point p, double t){ // rads
36     return Point(p.x*cos(t) - p.y*sin(t), p.x*sin(t) + p.y*cos(t));
37 }
38 // Sorts points in CCW order about origin, points on neg x-axis come
39 last
40 // To change pivot to point x, just subtract x from all points and then
41 sort
42 bool half(Point &p) { return p.y == 0 ? p.x < 0 : p.y > 0; }
43 bool angularOrder(Point &x, Point &y) {
44     bool X = half(x), Y = half(y);
45     return X == Y ? (x ^ y) > 0 : X < Y;
46 }

```

3.2. Line

```

1 int sgn(ll x){return x<0? -1 : !!x;}
2 struct line{
3     line() {}
4     double a,b,c;//Ax+By=C
5     //pto MUST store float coordinates!
6     line(double a, double b, double c):a(a),b(b),c(c){}
7     line(pto p, pto q): a(q.y-p.y), b(p.x-q.x), c(a*p.x+b*p.y) {}
8     int side(pto p){return sgn(11(a) * p.x + 11(b) * p.y - c);}
9 };
10 bool parallels(line l1, line l2){return abs(11.a*12.b-12.a*11.b)<EPS;}
11 pto inter(line l1, line l2){//intersection
12     double det=11.a*12.b-12.a*11.b;
13     if(abs(det)<EPS) return pto(INF, INF);//parallels
14     return pto(12.b*11.c-11.b*12.c, 11.a*12.c-12.a*11.c)/det;
15 }

```

3.3. Segment

```

1 struct segm {
2     pto s, f;
3     segm(pto s, pto f) : s(s), f(f) {}

```



```

4   pto closest(pto p) { // use for dist to point
5       double l2 = dist2(s, f);
6       if (l2 == 0.) return s;
7       double t = ((p-s) * (f-s)) / l2;
8       if (t < 0.) return s; // don't write if its a line
9       else if (t > 1.) return f; // don't write if its a line
10      return s + ((f-s) * t);
11  }
12  bool inside(pto p) { return abs(dist(s, p) + dist(p, f) - dist(s, f)
13      ) < EPS; }
14  };
15  // Note: if the segments are collinear it only returns a point of
16  // intersection
17  pto inter(segm &s1, segm &s2){
18      if (s1.inside(s2.s)) return s2.s; // if they are collinear
19      if (s1.inside(s2.f)) return s2.f; // if they are collinear
20      pto r = inter(line(s1.s, s1.f), line(s2.s, s2.f));
21      if (s1.inside(r) && s2.inside(r)) return r;
22      return pto(INF, INF);
23  }

```

3.4. Point in Poly

```

1  //checks if v is inside of P, using ray casting
2  //works with convex and concave.
3  //excludes boundaries, handle it separately using segment.inside()
4  bool inPolygon(pto v, vector<pto>& P) {
5      bool c = 0;
6      forn(i, si(P)){
7          int j = (i+1) % si(P);
8          if((P[j].y > v.y) != (P[i].y > v.y) && (v.x < (P[i].x - P[j].x) * (v
9              .y-P[j].y) / (P[i].y - P[j].y) + P[j].x)) c = !c;
10     }
11     return c;
12 }

```

3.5. Convex Hull

```

1  // Stores convex hull of P in S in CCW order
2  // Left must return >= -EPS to delete collinear points!
3  void chull(vector<pto>& P, vector<pto> &S){
4      S.clear(), sort(all(P)); // first x, then y
5      forn(i, si(P)) { // lower hull

```

```

6          while (si(S) >= 2 && S[si(S)-1].left(S[si(S)-2], P[i])) S.pop_back()
7              ;
8          S.pb(P[i]);
9      }
10     S.pop_back();
11     int k = si(S);
12     dforn(i, si(P)) { // upper hull
13         while (si(S) >= k+2 && S[si(S)-1].left(S[si(S)-2], P[i])) S.pop_back
14             ();
15         S.pb(P[i]);
16     }
17     S.pop_back();
18 }

```

3.6. Cut Polygon

```

1  //cuts polygon Q along the line ab
2  //stores the left side (swap a, b for the right one) in P
3  void cutPolygon(pto a, pto b, vector<pto> Q, vector<pto> &P){
4      P.clear();
5      forn(i, sz(Q)){
6          double left1=(b-a)^(Q[i]-a), left2=(b-a)^(Q[(i+1)%sz(Q)]-a);
7          if(left1>=0) P.pb(Q[i]);
8          if(left1*left2<0)
9              P.pb(inter(line(Q[i], Q[(i+1)%sz(Q)]), line(a, b)));
10     }
11 }

```

3.7. Heron's formula

$$A = \sqrt{s(s-a)(s-b)(s-c)}, \quad s = \frac{a+b+c}{2}.$$

4. DP Opt

Observaciones:

$A[i][j]$ el menor k que logra la solución óptima. En Knuth y D&C la idea es aprovechar los rangos determinados por este arreglo.

4.1. Knuth

Problema de ejemplo: dado un palito de longitud l , con n puntos en los que se puede cortar, determinar el costo mínimo para partir el palito en $n + 1$ palitos unitarios (la DP se puede adaptar a k agregando un parámetro extra), donde hay un costo fijo

por partir el rango i, j que cumple la condición suficiente. Una función de costos que cumple es la distancia entre los extremos $j - i$. El problema clásico de esta pinta es el del ABB óptimo.

Recurrencia original: $dp[i][j] = \min_{i < k < j} dp[i][k] + dp[k][j] + C[i][j]$ o bien $dp[i][j] = \min_{k < j} dp[i-1][k] + C[k][j]$

Condición suficiente: $A[i, j-1] \leq A[i, j] \leq A[i+1, j]$

Es decir, si saco un elemento a derecha el óptimo se mueve a izquierda o se mantiene, y si saco un elemento a izquierda el óptimo se mueve a derecha o se mantiene.

Complejidad original: $O(n^3)$

Complejidad optimizada: $O(n^2)$

Solución: iteramos por el tamaño len del subarreglo (creciente), y para cada extremo izquierdo l , determinamos el extremo derecho $r = l + len$ e iteramos por los k entre $A[l][r-1]$ y $A[l+1][r]$, actualizando la solución del estado actual.

```

1  int cost(int l, int r); // Implementar
2
3  // Intervalos: cerrado, cerrado.
4  // Modificar tipos, comparador y neutro (INF). Revisar caso base (i, i
   +1).
5  const ll INF = 1e18;
6  ll knuth(int n) {
7      vector<vi> opt(n, vi(n));
8      vector<vll> dp(n, vll(n));
9
10     // Casos base
11     forn(i, n-2) dp[i][i+2] = cost(i, i+2), opt[i][i+2] = i+1;
12
13     // Casos recursivos
14     forsn(len, 3, n+1) {
15         forn(l, n-len) {
16             int r = l+len;
17
18             dp[l][r] = INF;
19             forsn(k, opt[l][r-1], opt[l+1][r]+1) {
20                 ll val = dp[l][k] + dp[k][r] + cost(l, r);
21                 if (val < dp[l][r]) {
22                     dp[l][r] = val;
23                     opt[l][r] = k;
24                 }
25             }
26         }
27     }
28 
```

```

29     return dp[0][n-1];
30 }

```

4.2. Divide & Conquer

Problema de ejemplo: dado un arreglo de n números con valores a_1, a_1, \dots, a_n , dividirlo en k subarreglos, tal que la suma de los cuadrados del peso total de cada subarreglo es mínimo.

Recurrencia original: $dp[i][j] = \min_{k < j} dp[i-1][k] + C[k][j]$

Condición suficiente: $A[i][j] \leq A[i][j+1]$ o (normalmente más fácil de probar) $C[a][d] + C[b][c] \geq C[a][c] + C[b][d]$, con $a < b < c < d$.

La segunda condición suficiente es la intuición de que no conviene que los intervalos se contengan.

Complejidad original: $O(kn^2)$

Complejidad optimizada: $O(kn \log(n))$

Solución: la idea es, para un i determinado, partir el rango $[j_{left}, j_{right}]$ al que pertenecen los j que queremos calcular a la mitad, determinar el óptimo y utilizarlo como límite para calcular los demás. Para implementar esto de forma sencilla, se suele utilizar la función recursiva $dp(i, j_{left}, j_{right}, opt_{left}, opt_{right})$ que se encarga de, una vez fijado el punto medio m del rango $[j_{left}, j_{right}]$ iterar por los k en $[j_{left}, j_{right}]$ para determinar el óptimo opt para m , y continuar calculando $dp(i, j_{left}, m, opt_{left}, opt)$ y $dp(i, m, j_{right}, opt, opt_{right})$.

```

1  // Modificar: tipos, operacion (max, min), neutro (INF), funcion de
   costo.
2  const ll INF = 1e18;
3
4  ll cost(int i, int j); // Implementar. Costo en rango [i, j].
5
6  vector<ll> dp_before, dp_cur;
7  // compute dp_cur[l, r)
8  void compute(int l, int r, int optl, int opttr)
9  {
10     if (l == r) return;
11     int mid = (l + r) / 2;
12     pair<ll, int> best = {INF, -1};
13
14     forsn(k, optl, min(mid, opttr))
15         best = min(best, {dp_before[k] + cost(k, mid), k});
16
17     dp_cur[mid] = best.first;
18     int opt = best.second;
19 
```

```

20     compute(l, mid, optl, opt + 1);
21     compute(mid + 1, r, opt, opt+1);
22 }
23
24 ll dc_opt(int n, int k) {
25     dp_before.assign(n+1, INF); dp_before[0] = 0;
26     dp_cur.resize(n+1); // Cuidado, dp_cur[0] = 0. No molesta porque no
        se elige.
27
28     while (k--) {
29         compute(1, n+1, 0, n); // Parametros tal que por lo menos 1 en
        cada subarreglo.
30         dp_before = dp_cur;
31     }
32
33     return dp_cur[n];
34 }

```

5. Matemática

5.1. Teoría de números

5.1.1. Funciones multiplicativas, función de Möbius

Una función $f(n)$ es **multiplicativa** si para cada par de enteros coprimos p y q se cumple que $f(pq) = f(p)f(q)$.

Si la función $f(n)$ es multiplicativa, puede evaluarse en un valor arbitrario conociendo los valores de la función en sus factores primos: $f(n) = f(p_1^{r_1})f(p_2^{r_2}) \dots f(p_k^{r_k})$.

La **función de Möbius** se define como:

$$\mu(n) = \begin{cases} 0 & d^2 \mid n, \\ 1 & n = 1, \\ (-1)^k & n = p_1 p_2 \dots p_k. \end{cases}$$

5.1.2. Teorema de Wilson

$(p-1)! \equiv -1 \pmod{p}$ Siendo p primo.

5.1.3. Pequeño teorema de Fermat

$a^p \equiv a \pmod{p}$ Siendo p primo.

5.1.4. Teorema de Euler

$$a^{\varphi(n)} \equiv 1 \pmod{n}$$

5.2. Combinatoria

5.2.1. Burnside's lemma

Sea G un grupo que actúa en un conjunto X . Para cada g en G , sea X^g el conjunto de elementos en X que son invariantes respecto a g , entonces el número de órbitas $|X/G|$ es:

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} |X^g|.$$

Por ejemplo, si el grupo G consiste de las operaciones de rotación, el conjunto X son los posibles coloreos de un tablero, entonces el número de órbitas $|X/G|$ es el número de posibles coloreos de un tablero salvo rotaciones.

5.2.2. Lucas Theorem

$$\binom{m}{n} \equiv \prod_{i=0}^k \binom{m_i}{n_i} \pmod{p}$$

where $m = m_k p^k + m_{k-1} p^{k-1} + \dots + m_1 p + m_0$,

and $n = n_k p^k + n_{k-1} p^{k-1} + \dots + n_1 p + n_0$

$\binom{m}{n} = 0$ if $m < n$.

```

1 // Calcula C(n,k) % p teniendo C[p][p] precalculado, p primo
2 ll lucas(ll n, ll k, int p) {
3     ll ans = 1;
4     while (n + k) {
5         ans = (ans * C[n % p][k % p]) % p;
6         n /= p, k /= p;
7     }
8     return ans;
9 }

```

5.2.3. Stirling

$\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ = cantidad de formas de particionar un conjunto de n elementos en m subconjuntos no vacíos.

$$\left\{ \begin{smallmatrix} n+1 \\ k \end{smallmatrix} \right\} = k \left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\} + \left\{ \begin{smallmatrix} n \\ k-1 \end{smallmatrix} \right\}$$

for $k > 0$ with initial conditions

$$\left\{ \begin{smallmatrix} 0 \\ 0 \end{smallmatrix} \right\} = 1 \quad \text{and} \quad \left\{ \begin{smallmatrix} n \\ 0 \end{smallmatrix} \right\} = \left\{ \begin{smallmatrix} 0 \\ n \end{smallmatrix} \right\} = 0 \text{ for } n > 0.$$

```

1 | const int MAXS = 1e3+1;
2 | int S[MAXS][MAXS];
3 | void stirling() {
4 |     S[0][0] = 1;
5 |     forsn(i, 1, N) S[i][0] = S[0][i] = 0;
6 |     forsn(i, 1, N) forsn(j, 1, N)
7 |         S[i][j] = add(mul(S[i-1][j], j), S[i-1][j-1]);
8 | }

```

$$\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = \frac{1}{k!} \sum_{i=0}^k (-1)^i \binom{k}{i} (k-i)^n.$$

5.2.4. Bell

B_n = cantidad de formas de particionar un conjunto de n elementos en subconjuntos no vacíos.

$$B_0 = B_1 = 1$$

$$B_{n+1} = \sum_{k=0}^n \binom{n}{k} B_k.$$

$$B_n = \sum_{k=0}^n \left\{ \begin{matrix} n \\ k \end{matrix} \right\}.$$

```

1 | const int MAXB = 1e3+1;
2 | int B[MAXB][MAXB];
3 | void bell() {
4 |     B[0] = 1;
5 |     forsn(i, 1, MAXB) forsn(k, i)
6 |         B[i] = add(B[i], mul(C[i-1][k], B[k]));
7 | }

```

5.2.5. Eulerian

$A_{n,m}$ = cantidad de permutaciones de 1 a n con m ascensos (m elementos mayores que el anterior).

$$A(n, m) = (n - m)A(n - 1, m - 1) + (m + 1)A(n - 1, m).$$

5.2.6. Catalan

C_n = cantidad de árboles binarios de $n+1$ hojas, en los que cada nodo tiene cero o dos hijos.

$$C_n = \binom{2n}{n} - \binom{2n}{n-1} \quad \text{con } n \geq 1.$$

$$C_0 = 1 \quad \text{y} \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{con } n \geq 0.$$

5.3. Euler's Phi

```

1 | /* Euler's totient function (phi) counts the positive integers
2 |    up to a given integer n that are relatively prime to n. */
3 |

```

```

4 | const int N = 1e6;
5 | vi lp(N+1), P, phi(N+1);
6 |
7 | void initPhi() { // Least prime and phi <= N in O(n)
8 |     phi[1] = 1;
9 |     forsn(i, 2, N+1) {
10 |         if (!lp[i])
11 |             lp[i] = i, P.pb(i), phi[i] = i-1;
12 |         else {
13 |             int a = i / lp[i];
14 |             phi[i] = phi[a] * (lp[i] - (lp[i] != lp[a]));
15 |         }
16 |         for (int p : P) {
17 |             if (p > lp[i] || i*p > N) break;
18 |             lp[i * p] = p;
19 |         }
20 |     }
21 | }
22 |
23 | ll eulerPhi(ll x) { // O(lg x)
24 |     ll r = x;
25 |     map<ll,int> f = fact(x);
26 |     for (auto &i : f) r -= r / i.fst;
27 |     return r;
28 | }
29 |
30 | ll eulerPhi(ll x) { // O(sqrt x)
31 |     ll r = x;
32 |     for (ll i = 2; i*i <= x; i++) {
33 |         if (x % i == 0) {
34 |             r -= r/i;
35 |             while (x % i == 0) x /= i;
36 |         }
37 |     }

```

```

38     if (x > 1) r -= r/x;
39     return r;
40 }
    
```

5.4. Euclides extendido

Dados a y b , encuentra x e y tales que $a * x + b * y = \gcd(a, b)$.

```

1 pair<ll,ll> extendedEuclid (ll a, ll b){ //a * x + b * y = gcd(a,b)
2     ll x,y;
3     if (b==0) return mp(1,0);
4     auto p=extendedEuclid(b,a%b);
5     x=p.snd;
6     y=p.fst-(a/b)*x;
7     return mp(x,y);
8 }
    
```

5.5. Inversos

```

1 const int MAXM = 15485867; // Tiene que ser primo
2 ll inv[MAXM]; //inv[i]*i=1 M M
3 void calc(int p){//0(p)
4     inv[1]=1;
5     forsn(i, 2, p) inv[i]= p-((p/i)*inv[p%i])%p;
6 }
7 // Llamar calc(MAXM);
8
9 int inv(int x){//0(log x)
10     return pot(x, eulerphi(M)-1);//si M no es primo(sacar a mano)
11     return pot(x, M-2);//si M es primo
12 }
13
14 // Inversos con euclides en O(log(x)) sin precomputo:
15 // extendedEuclid(a, -m).fst (si coprimos a y m)
    
```

5.6. Ecuaciones diofánticas

Basado en Euclides extendido. Dados a , b , y r obtiene x e y tales que $a * x + b * y = r$, suponiendo que $\gcd(a, b) | r$. Las soluciones son de la forma $(x, y) = (x_1 - b/\gcd(a, b) * k_1, x_2 + a/\gcd(a, b) * k_2)$ donde x_1 y x_2 son las soluciones particulares que obtuvo Euclides.

```

1 pair<pair<ll,ll>,pair<ll,ll> > diophantine(ll a,ll b, ll r) {
2     //a*x+b*y=r where r is multiple of gcd(a,b);
3     ll d=gcd(a,b);
    
```

```

4     a/=d; b/=d; r/=d;
5     auto p = extendedEuclid(a,b);
6     p.fst*=r; p.snd*=r;
7     assert(a*p.fst+b*p.snd==r);
8     return mp(p,mp(-b,a)); // solutions: (p.fst - b*k, p.snd + a*k)
9         //== (res.fst.fst + res.snd.fst*k, res.fst.snd + res.snd
10         .snd*k)
    
```

5.7. Teorema Chino del Resto

Dadas k ecuaciones de la forma $a_i * x \equiv a_i \pmod{n_i}$, encuentra x tal que es solución. Existe una única solución módulo $\text{lcm}(n_i)$.

```

1 #define mod(a,m) ((a)%m < 0 ? (a)%m+m) : (a)%m) // evita overflow
2     al no sumar si >= 0
3 typedef tuple<ll,ll,ll> ec;
4 pair<ll,ll> sol(ec c){ //requires inv, diophantine
5     ll a=get<0>(c), x1=get<1>(c), m=get<2>(c), d=gcd(a,m);
6     if (d==1) return mp(mod(x1*inv(a,m),m), m);
7     else return x1%d ? mp(-1LL,-1LL) : sol({a/d,x1/d,m/d});
8 }
9 pair<ll,ll> crt(vector< ec > cond) { // returns: (sol, lcm)
10     ll x1=0,m1=1,x2,m2;
11     for(auto t:cond){
12         tie(x2,m2)=sol(t);
13         if((x1-x2)%gcd(m1,m2))return mp(-1,-1);
14         if(m1==m2)continue;
15         ll k=diophantine(m2,-m1,x1-x2).fst.snd,l=m1*(m2/gcd(m1,m2));
16         x1=mod(m1*mod(k, l/m1)+x1,l);m1=l; // evita overflow con prop modulo
17     }
18     return sol(make_tuple(1,x1,m1));
19 } //cond[i]={ai,bi,mi} ai*xi=bi (mi); assumes lcm fits in ll
    
```

5.8. Matrices

```

1 struct Mat {
2     vector<vector<double> > rows;
3     Mat(int n): rows(n, vector<double>(n)) {}
4     Mat(int n, int m): rows(n, vector<double>(m)) {}
5
6     vector<double> &operator[](int f) { return rows[f]; }
    
```

```

7   int size() { return si(rows); }
8
9   Mat operator+(Mat &b) { // this de n x m entonces b de n x m
10      Mat m(si(rows), si(rows[0]));
11      forn(i, si(rows)) forn(j, si(rows[0])) m[i][j] = rows[i][j] + b[
12          i][j];
13      return m;
14  }
15  Mat operator*(Mat &b) { // this de n x m entonces b de m x t
16      int n = si(rows), m = si(rows[0]), t = si(b[0]);
17      Mat mat(n, t);
18      forn(i, n) forn(j, t) forn(k, m) mat[i][j] += rows[i][k] * b[k][
19          j];
20      return mat;
21  }
22  Mat operator^(int e) { // this debe ser matriz cuadrada
23      int n = si(rows);
24      Mat res(n); forn(i, n) res[i][i] = 1;
25
26      Mat base = *this;
27      while (e > 0) {
28          if (e % 2 == 1) res = res * base;
29          base = base * base;
30          e /= 2;
31      }
32      return res;
33  };
34  // to calculate determinants, use determinant.cpp

```

5.9. Determinante

```

1  double determinant(Mat m) { // do gaussian elimination and calculate
2      determinant
3      double det = 1;
4      int n = si(m);
5      forn(i, n) { // for each col
6          int k = i;
7          forsn(j, i+1, n) // row with largest abs val to avoid floating
8              point errors
9              if (abs(m[j][i]) > abs(m[k][i]))
10                  k = j;

```

```

9      if (abs(m[k][i]) < EPS) return 0;
10     swap(m[i], m[k]); // move pivot row
11     if (i != k) det = -det;
12     det *= m[i][i];
13     forsn(j, i+1, n) m[i][j] /= m[i][i]; // scale current row
14     forn(j, n) if (j != i && abs(m[j][i]) > EPS) // zero out other
15         rows
16         forsn(k, i+1, n)
17             m[j][k] -= m[i][k] * m[j][i];
18     }
19     return det;
20 }
21 // if mod 2, check gauss.cpp for a faster implementation

```

5.10. Sistemas de Ecuaciones Lineales - Gauss

```

1  const double EPS = 1e-9;
2  const int INF = 1e9; // it doesn't actually have to be infinity or a big
3      number
4  int gauss(Mat mat, vector<double> &ans) { // returns number of solutions
5      int n = si(mat);
6      int m = n > 0 ? si(mat[0]) - 1 : 0;
7
8      vi where(m, -1);
9      for (int col = 0, row = 0; col < m && row < n; col++) { // for each
10          col
11          int sel = row;
12          forsn(i, row, n) // row with largest abs val to avoid floating
13              point errors
14              if (abs(mat[i][col]) > abs(mat[sel][col]))
15                  sel = i;
16          if (abs(mat[sel][col]) < EPS)
17              continue;
18          swap(mat[sel], mat[row]); // move pivot row
19          where[col] = row;
20
21          forn(i, n) if (i != row) { // zero out other rows
22              double c = mat[i][col] / mat[row][col];
23              forsn(j, col, m + 1)
24                  mat[i][j] -= mat[row][j] * c;
25          }

```

```

24     row++;
25 }
26
27 ans.assign(m, 0);
28 forn(i, m)
29     if (where[i] != -1)
30         ans[i] = mat[where[i]][m] / mat[where[i]][i]; // calculate
                x_i
31 forn(i, n) { // check if the solution is valid (also possible to
                check: if a row has all zero-coefficients -> the constant term
                is also zero)
32     double sum = 0;
33     forn(j, m)
34         sum += ans[j] * mat[i][j];
35     if (abs(sum - mat[i][m]) > EPS)
36         return 0;
37 }
38
39 forn(i, m)
40     if (where[i] == -1)
41         return INF;
42 return 1;
43 }
44
45 // SPEED IMPROVEMENT IF MOD 2:
46 int gauss(vector<bitset<N>> a, int n, int m, bitset<N> &ans) {
47     vi where(m, -1);
48     for (int col = 0, row = 0; col < m && row < n; col++) {
49         forsn(i, row, n)
50             if (a[i][col]) {
51                 swap(a[i], a[row]);
52                 break;
53             }
54         if (!a[row][col])
55             continue;
56         where[col] = row;
57
58         forn(i, n)
59             if (i != row && a[i][col])
60                 a[i] ^= a[row];
61         row++;
62     }
63     // The rest of implementation is the same as above
    
```

```

64 }
    
```

5.11. FFT y NTT

NTT: es un algoritmo más lento pero más preciso para calcular la DFT, ya que trabaja con enteros módulo un primo m .

El módulo m debe ser un primo de la forma $m = c2^k + 1$. Para encontrar la raíz 2^k -ésima de la unidad r : $r = g^c$, donde g es una raíz primitiva de p (número tal que si lo elevamos a diferentes potencias recorremos todos los demás).

Valores tradicionales: $m = 998244353$ y $r = 3$, $m = 2305843009255636993$ y $r = 5$ (este último da overflow, se podría fixear).

Operaciones:

Es mucho más fácil realizar ciertas operaciones en un dominio de frecuencias:

- Multiplicar en $O(n \log(n))$: simplemente multiplicar punto a punto.
- Invertir en $O(n \log(n))$: asumiendo $B(0) \neq 0$, existe una serie infinita $C(x)$ que es inverso del polinomio. Aprovechando ciertas propiedades del producto $B(x)C(x)$ ($b_0c_0 = 1$ y el resto de los coeficientes resultantes son 0), podemos ir despejando el inverso. Es posible aplicar Divide and Conquer notando la relación entre los primeros $n/2$ términos del inverso y los siguientes $n/2$.
- Dividir en $O(n \log(n))$: resulta más fácil dividir los polinomios reversos (ya que un polinomio y su reverso son casi iguales, y no hace falta considerar resto de la división de los reversos).
- Multievaluar en $O(n \log^2(n))$: evaluar un polinomio $A(x)$ en x_1 es lo mismo que dividir $A(x)$ por $x - x_1$ y evaluar el resto $R(x)$ en x_1 . Para múltiples puntos, podemos utilizar una estrategia estilo Divide and Conquer.
- Interpolarse en $O(n \log^2(n))$: para interpolar se utilizan los polinomios de Lagrange (ver interpolación de Lagrange, $A(x) = \sum_{i=1}^n y_i \frac{1}{p_i'(x_i)} p_i(x)$ y $p_i(x) = \frac{p(x)}{x - x_i}$). Para poder computarlos rápidamente, aprovechamos que $p'(x_i) = p_i'(x_i)$ (podemos computar la derivada y evaluar con multievaluación) y utilizamos una estrategia estilo Segment Tree para generar los polinomios rápidamente (notando que si mantenemos los polinomios para dos conjuntos de puntos es fácil unirlos).

```

1 // N must be power of 2 !!!
2 // Tiene que entrar el resultado!!! (el producto, probablemente el doble
    de la entrada)
3 using tf = int;
4 using poly = vector<tf>;
5 // FFT
6 struct CD {
    
```



```

7  double r,i;
8  CD(double r=0, double i=0):r(r),i(i){
9  double real()const{return r;}
10 void operator/=(const int c){r/=c, i/=c;}
11 };
12 CD operator*(const CD& a, const CD& b){
13     return CD(a.r*b.r-a.i*b.i,a.r*b.i+a.i*b.r);}
14 CD operator+(const CD& a, const CD& b){return CD(a.r+b.r,a.i+b.i);}
15 CD operator-(const CD& a, const CD& b){return CD(a.r-b.r,a.i-b.i);}
16 const double pi=acos(-1.0);
17 // NTT
18 // M-1 needs to be a multiple of N !!
19 // tf TIENE que ser ll (si el modulo es grande)
20 // big mod and primitive root for NTT:
21 /*
22 const tf M=998244353,RT=3;
23 struct CD {
24     tf x;
25     CD(tf _x):x(_x){}
26     CD(){}
27 };
28 CD operator*(const CD& a, const CD& b){return CD(mul(a.x,b.x));}
29 CD operator+(const CD& a, const CD& b){return CD(add(a.x,b.x));}
30 CD operator-(const CD& a, const CD& b){return CD(sub(a.x,b.x));}
31 vector<tf> rts(N+9,-1);
32 CD root(int n, bool to_inv){
33     tf r=rts[n]<0?rts[n]=pot(RT,(M-1)/n):rts[n];
34     return CD(to_inv?inv(r):r);
35 }
36 */
37 CD cp1[N+9],cp2[N+9];
38 int R[N+9];
39 void dft(CD* a, int n, bool to_inv){
40     forn(i,n)if(R[i]<i)swap(a[R[i]],a[i]);
41     for(int m=2;m<=n;m*=2){
42         double z=2*pi/m*(to_inv?-1:1); // FFT
43         CD wi=CD(cos(z),sin(z)); // FFT
44         // CD wi=root(m,to_inv); // NTT
45         for(int j=0;j<n;j+=m){
46             CD w(1);
47             for(int k=j,k2=j+m/2;k2<j+m;k++,k2++){
48                 CD u=a[k];CD v=a[k2]*w;a[k]=u+v;a[k2]=u-v;w=w*wi;
49             }

```

```

50     }
51 }
52 if(to_inv)forn(i,n)a[i]/=n; // FFT
53 //if(to_inv){ // NTT
54 //     CD z(inv(n));
55 //     forn(i,n)a[i]=a[i]*z;
56 // }
57 }
58 poly multiply(poly& p1, poly& p2){
59     int n=si(p1)+si(p2)+1;
60     int m=1,cnt=0;
61     while(m<=n)m*=m,cnt++;
62     forn(i,m){R[i]=0;forn(j,cnt)R[i]=(R[i]<<1)|((i>>j)&1);}
63     forn(i,m)cp1[i]=0,cp2[i]=0;
64     forn(i,si(p1))cp1[i]=p1[i];
65     forn(i,si(p2))cp2[i]=p2[i];
66     dft(cp1,m,false);dft(cp2,m,false);
67     forn(i,m)cp1[i]=cp1[i]*cp2[i];
68     dft(cp1,m,true);
69     poly res;
70     n-=2;
71     forn(i,n)res.pb((tf)floor(cp1[i].real()+0.5)); // FFT
72     //forn(i,n)res.pb(cp1[i].x); // NTT
73     return res;
74 }

1 //Polynomial division: O(n*log(n))
2 //Multi-point polynomial evaluation: O(n*log^2(n))
3 //Polynomial interpolation: O(n*log^2(n))
4
5 //Works with NTT. For FFT, just replace add,sub,mul,inv,divide
6 poly add(poly &a, poly &b){
7     int n=si(a),m=si(b);
8     poly ans(max(n,m));
9     forn(i,max(n,m)){
10         if(i<n) ans[i]=add(ans[i],a[i]);
11         if(i<m) ans[i]=add(ans[i],b[i]);
12     }
13     while(si(ans)>1&&!ans.back())ans.pop_back();
14     return ans;
15 }

16
17 /// B(0) != 0 !!!

```

```

18 poly invert(poly &b, int d){
19     poly c = {inv(b[0])};
20     while(si(c)<=d){
21         int j=2*si(c);
22         auto bb=b; bb.resize(j);
23         poly cb=multiply(c,bb);
24         forn(i,si(cb)) cb[i]=sub(0,cb[i]);
25         cb[0]=add(cb[0],2);
26         c=multiply(c,cb);
27         c.resize(j);
28     }
29     c.resize(d+1);
30     return c;
31 }
32
33 pair<poly,poly> divslow(poly &a, poly &b){
34     poly q,r=a;
35     while(si(r)>=si(b)){
36         q.pb(divide(r.back(),b.back()));
37         if(q.back()) forn(i,si(b)){
38             r[si(r)-i-1]=sub(r[si(r)-i-1],mul(q.back(),b[si(b)-i-1]));
39         }
40         r.pop_back();
41     }
42     reverse(all(q));
43     return {q,r};
44 }
45
46 pair<poly,poly> divide(poly &a, poly &b){ //returns {quotient,remainder}
47     int m=si(a),n=si(b),MAGIC=750;
48     if(m<n) return {{0},a};
49     if(min(m-n,n)<MAGIC)return divslow(a,b);
50     poly ap=a; reverse(all(ap));
51     poly bp=b; reverse(all(bp));
52     bp=invert(bp,m-n);
53     poly q=multiply(ap,bp);
54     q.resize(si(q)+m-n-si(q)+1,0);
55     reverse(all(q));
56     poly bq=multiply(b,q);
57     forn(i,si(bq)) bq[i]=sub(0,bq[i]);
58     poly r=add(a,bq);
59     return {q,r};
60 }

```

```

61
62 vector<poly> tree;
63
64 void filltree(vector<tf> &x){
65     int k=si(x);
66     tree.resize(2*k);
67     forsn(i,k,2*k) tree[i]={sub(0,x[i-k]),1};
68     dfor sn(i,1,k) tree[i]=multiply(tree[2*i],tree[2*i+1]);
69 }
70
71 vector<tf> evaluate(poly &a, vector<tf> &x){
72     filltree(x);
73     int k=si(x);
74     vector<poly> ans(2*k);
75     ans[1]=divide(a,tree[1]).snd;
76     forsn(i,2,2*k) ans[i]=divide(ans[i>>1],tree[i]).snd;
77     vector<tf> r; forn(i,k) r.pb(ans[i+k][0]);
78     return r;
79 }
80
81 poly derivate(poly &p){
82     poly ans(si(p)-1);
83     forsn(i,1,si(p)) ans[i-1]=mul(p[i],i);
84     return ans;
85 }
86
87 poly interpolate(vector<tf> &x, vector<tf> &y){
88     filltree(x);
89     poly p=derivate(tree[1]);
90     int k=si(y);
91     vector<tf> d=evaluate(p,x);
92     vector<poly> intree(2*k);
93     forsn(i,k,2*k) intree[i]={divide(y[i-k],d[i-k])};
94     dfor sn(i,1,k) {
95         poly p1=multiply(tree[2*i],intree[2*i+1]);
96         poly p2=multiply(tree[2*i+1],intree[2*i]);
97         intree[i]=add(p1,p2);
98     }
99     return intree[1];
100 }

```

5.12. Simplex

Introducción

Permite maximizar cierta función lineal dado un conjunto de restricciones lineales.

Algoritmo

El algoritmo opera con programas lineales en la siguiente forma canónica: maximizar $z = c^T x$ sujeta a $Ax \leq b, x \geq 0$.

Por ejemplo, si $c = (2, -1)$, $A = \begin{bmatrix} 1 & 0 \end{bmatrix}$ y $b = (5)$, buscamos maximizar $z = 2x_1 - x_2$ sujeta a $x_1 \leq 5$ y $x_i \geq 0$.

Detalles implementativos

Canonizar si hace falta.

Para obtener soluciones negativas, realizar el cambio de variable $x_i = x'_i + \text{INF}$.

Si la desigualdad no incluye igual, solo menor, **no usar epsilon** al agregarla. Esto ya es considerado por el código.

```

1  const double EPS = 1e-5;
2  // if inequality is strictly less than (< vs <=), do not use EPS! this
   case is covered in the code
3  namespace Simplex {
4      vi X,Y;
5      vector<vector<double>> > A;
6      vector<double> b,c;
7      double z;
8      int n,m;
9      void pivot(int x,int y){
10         swap(X[y],Y[x]);
11         b[x]/=A[x][y];
12         for(i,m)if(i!=y)A[x][i]/=A[x][y];
13         A[x][y]=1/A[x][y];
14         for(i,n)if(i!=x&&abs(A[i][y])>EPS){
15             b[i]-=A[i][y]*b[x];
16             for(j,m)if(j!=y)A[i][j]-=A[i][y]*A[x][j];
17             A[i][y]=-A[i][y]*A[x][y];
18         }
19         z+=c[y]*b[x];
20         for(i,m)if(i!=y)c[i]-=c[y]*A[x][i];
21         c[y]=-c[y]*A[x][y];
22     }
23     pair<double,vector<double>> simplex( // maximize c^T x s.t. Ax<=b,
        x>=0
24         vector<vector<double>> > _A, vector<double> _b, vector<double>
            > _c){
25         // returns pair (maximum value, solution vector)
26         A=_A;b=_b;c=_c;
```

```

27     n=si(b);m=si(c);z=0.;
28     X=vi(m);Y=vi(n);
29     for(i,m)X[i]=i;
30     for(i,n)Y[i]=i+m;
31     while(1){
32         int x=-1,y=-1;
33         double mn=-EPS;
34         for(i,n)if(b[i]<mn)mn=b[i],x=i;
35         if(x<0)break;
36         for(i,m)if(A[x][i]<-EPS){y=i;break;}
37         assert(y>=0); // no solution to Ax<=b
38         pivot(x,y);
39     }
40     while(1){
41         int x=-1,y=-1;
42         double mx=EPS;
43         for(i,m)if(c[i]>mx)mx=c[i],y=i;
44         if(y<0)break;
45         double mn=1e200;
46         for(i,n)if(A[i][y]>EPS&&b[i]/A[i][y]<mn)mn=b[i]/A[i][y],x=i
47         ;
48         assert(x>=0); // c^T x is unbounded
49         pivot(x,y);
50     }
51     vector<double> r(m);
52     for(i,n)if(Y[i]<m)r[Y[i]]=b[i];
53     return mp(z,r);
54 };
```

6. Grafos

6.1. Teoremas y fórmulas

6.1.1. Teorema de Pick

$$A = I + \frac{B}{2} - 1$$

Donde A es el área, I es la cantidad de puntos interiores, y B la cantidad de puntos en el borde.

6.1.2. Formula de Euler

$$v - e + f = k + 1$$

Donde v es la cantidad de vértices, e la cantidad de arcos, f la cantidad de caras y k la cantidad de componentes conexas.

6.2. Bellman-Ford

```

1 vector<ii> G[MAX_N]; //ady. list with pairs (weight, dst)
2 int dist[MAX_N];
3 void bford(int src){ //O(VE)
4     dist[src]=0;
5     forn(i, N-1) forn(j, N) if(dist[j]!=INF) for(auto u: G[j])
6         dist[u.second]=min(dist[u.second], dist[j]+u.first);
7 }
8
9 bool hasNegCycle(){
10     forn(j, N) if(dist[j]!=INF) for(auto u: G[j])
11         if(dist[u.second]>dist[j]+u.first) return true;
12     //inside if: all points reachable from u.snd will have -INF distance(
13         do bfs)
14     return false;
15 }

```

6.3. 2-SAT + Tarjan SCC

```

1 // We have one node for each boolean variable and other for its negation
2 // Every edge represents an implication, to add a clause (a or b), use
3 // add_or(a, b)
4 // val[comp[i]] = value of variable i
5 struct SAT {
6     vector<vi> g;
7     stack<int> q;
8     vector<bool> val;
9     vi low, idx, comp;
10     int n, id, comps, x;
11
12     SAT(int vars) {
13         n = vars, g.resize(2*n), id = 0, comps = 0;
14         low = vi(2*n), idx = vi(2*n, -1), comp = vi(2*n, -1);
15     }
16
17     int neg(int u) { return u >= n ? u-n : u+n; }
18     void add_or(int a, int b) { g[neg(a)].pb(b), g[neg(b)].pb(a); }
19
20     void tarjan(int u) {

```

```

20         low[u] = idx[u] = id++;
21         q.push(u), comp[u] = -2;
22         for (int v : g[u]) {
23             if (idx[v] == -1 || comp[v] == -2) {
24                 if (idx[v] == -1) tarjan(v);
25                 low[u] = min(low[u], low[v]);
26             }
27         }
28         if (low[u] == idx[u]) {
29             do { x = q.top(), q.pop(), comp[x] = comps; } while (x != u);
30             val.pb(comp[neg(u)] < 0), comps++;
31         }
32     }
33
34     bool satisfiable() {
35         forn(i, 2*n) if (idx[i] == -1) tarjan(i);
36         forn(i, n) if (comp[i] == comp[neg(i)]) return false;
37         return true;
38     }
39 };

```

6.4. Articulation Points

```

1 int N;
2 vector<int> G[1000000];
3 //V[i]=node number(if visited), L[i]= lowest V[i] reachable from i
4 int qV, V[1000000], L[1000000], P[1000000];
5 void dfs(int v, int f){
6     L[v]=V[v]=++qV;
7     for(auto u: G[v])
8         if(!V[u]){
9             dfs(u, v);
10            L[v] = min(L[v], L[u]);
11            P[v] += L[u]>=V[v];
12        }
13        else if(u!=f)
14            L[v]=min(L[v], V[u]);
15    }
16    int cantart(){ //O(n)
17        qV=0;
18        zero(V), zero(P);
19        dfs(1, 0); P[1]--;

```

```

20     int q=0;
21     forn(i, N) if(P[i]) q++;
22     return q;
23 }
    
```

6.5. Comp. Biconexas y Puentes

```

1 struct bridge {
2     struct edge {
3         int u,v,comp;
4         bool bridge;
5     };
6
7     int n,t,nbc;
8     vi d,b,comp;
9     stack<int> st;
10    vector<vi> adj;
11    vector<edge> e;
12
13    bridge(int n=0): n(n) {
14        adj = vector<vi>(n);
15        e.clear();
16        initDfs();
17    }
18
19    void initDfs() {
20        d = vi(n), b = vi(n), comp = vi(n);
21        forn(i,n) d[i] = -1;
22        nbc = t = 0;
23    }
24
25    void addEdge(int u, int v) {
26        adj[u].pb(si(e)); adj[v].pb(si(e));
27        e.pb((edge){u,v,-1,false});
28    }
29
30    //d[i]=id de la dfs
31    //b[i]=lowest id reachable from i
32    void dfs(int u=0, int pe=-1) {
33        b[u] = d[u] = t++;
34        comp[u] = pe != -1;
35
36        for(int ne : adj[u]) {
    
```

```

37            if(ne == pe) continue;
38            int v = e[ne].u ^ e[ne].v ^ u;
39            if(d[v] == -1) {
40                st.push(ne);
41                dfs(v,ne);
42                if(b[v] > d[u]) e[ne].bridge = true; // bridge
43                if(b[v] >= d[u]) { // art
44                    int last;
45                    do {
46                        last = st.top(); st.pop();
47                        e[last].comp = nbc;
48                    } while(last != ne);
49                    nbc++, comp[u]++;
50                }
51                b[u] = min(b[u], b[v]);
52            }
53            else if(d[v] < d[u]) { // back edge
54                st.push(ne);
55                b[u] = min(b[u], d[v]);
56            }
57        }
58    }
59 };
    
```

6.6. LCA + Climb

```

1 #define lg(x) (31-__builtin_clz(x))
2 struct LCA {
3     vector<vi> a; vi lvl; // a[i][k] is the 2^k ancestor of i
4     void dfs(int u=0, int p=-1, int l=0) {
5         a[u][0] = p, lvl[u] = l;
6         for (int v : g[u]) if (v != p) dfs(v, u, l+1);
7     }
8     LCA(int n) : a(n, vi(lg(n)+1)), lvl(n) {
9         dfs(); forn(k, lg(n)) forn(i, n) a[i][k+1] = a[i][k] == -1 ? -1
10            : a[a[i][k]][k];
11    }
12    int climb(int x, int d) {
13        for (int i = lg(lvl[x]); d && i >= 0; i--)
14            if ((1 << i) <= d) x = a[x][i], d -= 1 << i;
15        return x;
16    }
17    int lca(int x, int y) { // O(lg n)
    
```

```

17     if (lvl[x] < lvl[y]) swap(x, y);
18     if (lvl[x] != lvl[y]) x = climb(x, lvl[x] - lvl[y]);
19     if (x != y) {
20         for (int i = lg(lvl[x]); i >= 0; i--)
21             if (a[x][i] != a[y][i]) x = a[x][i], y = a[y][i];
22         x = a[x][0];
23     }
24     return x;
25 }
26 int dist(int x, int y) { return lvl[x] + lvl[y] - 2 * lvl[lca(x, y)]; }
27 };

```

6.7. Union Find

```

1 struct DSU {
2     vi par, sz;
3     DSU(int n): par(n), sz(n, 1) { iota(all(par), 0); }
4     int find(int u) { return par[u] == u ? u : par[u] = find(par[u]); }
5     bool connected(int u, int v) { return find(u) == find(v); }
6     bool join(int u, int v) {
7         u = find(u), v = find(v);
8         if (u == v) return false;
9         if (sz[u] < sz[v]) par[u] = v, sz[v] += sz[u];
10        else par[v] = u, sz[u] += sz[v];
11        return true;
12    }
13 };

```

6.8. Heavy Light Decomposition

```

1 // For values in nodes set the flag to false and load init values in val
2 template <class T>
3 struct HLD {
4     vi par, heavy, depth, val, root, rmqPos;
5     vector<vector<pii>> g; vector<pii> e;
6     RMQ<T> rmq; // requires seg tree
7     bool valuesInEdges = true;
8     int dfs(int u = 0) {
9         int size = 1, mx = 0;
10        for (auto [v, w] : g[u]) if (v != par[u]) {
11            par[v] = u, depth[v] = depth[u] + 1;
12            if (valuesInEdges) val[v] = w;
13            int sz = dfs(v);

```

```

14            if (sz > mx) heavy[u] = v, mx = sz;
15            size += sz;
16        }
17        return size;
18    }
19    HLD(int n) : par(n, -1), heavy(n, -1), depth(n),
20                val(n), root(n), rmqPos(n), g(n), rmq(n) {}
21    void addEdge(int u, int v, int w = 0) {
22        g[u].pb(v, w), g[v].pb(u, w), e.pb(u, v);
23    }
24    void build() {
25        dfs();
26        int pos = 0, n = si(g);
27        forn(i, n) if (par[i] == -1 || heavy[par[i]] != i)
28            for (int j = i; j != -1; j = heavy[j])
29                root[j] = i, rmqPos[j] = pos++;
30        for (auto &[u, v] : e) if (par[u] != v) swap(u, v);
31        forn(i, n) rmq[rmqPos[i]] = val[i];
32        rmq.build();
33    }
34    template <class Op>
35    void processPath(int u, int v, Op op) {
36        for (; root[u] != root[v]; v = par[root[v]]) {
37            if (depth[root[u]] > depth[root[v]]) swap(u, v);
38            op(rmqPos[root[v]], rmqPos[v] + 1);
39        }
40        if (valuesInEdges && u == v) return;
41        if (depth[u] > depth[v]) swap(u, v);
42        op(rmqPos[u] + valuesInEdges, rmqPos[v] + 1);
43    }
44    T query(int u, int v) {
45        T res = T();
46        processPath(u, v, [&](int l, int r) { res = res + rmq.get(l, r); });
47        return res;
48    }
49    void set(int i, const T &x) {
50        rmq.set(rmqPos[valuesInEdges ? e[i].fst : i], x);
51    }
52    void update(int u, int v, const T &x) { // requires lazy
53        processPath(u, v, [&](int l, int r) { rmq.update(l, r, x); });
54    }
55    int lca(int u, int v) { // not needed

```

```

56     for (; root[u] != root[v]; v = par[root[v]])
57         if (depth[root[u]] > depth[root[v]]) swap(u, v);
58     return depth[u] > depth[v] ? v : u;
59 }
60 };
    
```

6.9. Centroid Decomposition

```

1 struct Centroid {
2     int n, sz[N], parent[N]; bool used[N];
3
4     int size(int u, int p=-1){
5         sz[u] = 1;
6         for(int v : tree[u])
7             if(v != p && !used[v]) sz[u] += size(v,u);
8         return sz[u];
9     }
10
11     void build(int u=0, int p=-1, int s=-1){
12         if(s == -1) s = size(u);
13         for(int v : tree[u]) if(!used[v] && sz[v] > s/2)
14             { sz[u] = 0; build(v,p,s); return; }
15         used[u] = true, parent[u] = p;
16         for(int v : tree[u]) if(!used[v]) build(v,u,-1);
17     }
    
```

7. Flujo

7.1. Trucazos generales

- **Corte mínimo:** aquellos nodos alcanzables desde S forman un conjunto, los demás forman el otro conjunto. En Dinic's: vertices con $dist[v] \geq 0$ (del lado de S) vs. $dist[v] == -1$ (del lado del T).
- **Para grafos bipartitos:** sean V_1 y V_2 los conjuntos más próximos a S y a T respectivamente.
 - **Matching:** para todo $v_1 \in V_1$ tomar las aristas a vértices en V_2 con flujo positivo ($edge.f > 0$).
 - **Min. Vertex Cover:** unión de vértices $v_1 \in V_1$ tales que son inalcanzables ($dist[v_1] == -1$), y vértices $v_2 \in V_2$ tales que son alcanzables ($dist[v_2] > 0$).
 - **Max. Independent Set:** tomar vértices no tomados por el Min. Vertex Cover.

- **Max. Clique:** construir la red G' (red complemento) y encontrar Max. Independent Set.
- **Min. Edge Cover:** tomar las aristas del Matching y para todo vértice no cubierto hasta el momento, tomar cualquier arista incidente.
- **Konig's theorem:** $|minimum\ vertex\ cover| = |maximum\ matching| \Leftrightarrow |maximum\ independent\ set| + |maximum\ matching| = |vertices|$.

7.2. Dinic

Complejidad: $O(V^2E)$ en general. $O(\min(E^{3/2}, V^{2/3}E))$ con capacidades unitarias. $O(\sqrt{V}E)$ en matching bipartito (se lo llama Hopcroft–Karp algorithm) y en cualquier otra red unitaria (indegree = outdegree = 1 para cada vértice excepto S y T).

```

1 struct Dinic {
2     struct Edge { int v, r; ll c, f=0; };
3     vector<vector<Edge>> g; vi dist, ptr;
4     static const ll INF = 1e18;
5     int n, s, t;
6     Dinic(int _n, int _s, int _t) {
7         n = _n, s = _s, t = _t;
8         g.resize(n), dist = vi(n), ptr = vi(n);
9     }
10    void addEdge(int u, int v, ll c1, ll c2=0) {
11        g[u].pb((Edge){v, si(g[v]), c1});
12        g[v].pb((Edge){u, si(g[u])-1, c2});
13    }
14    bool bfs() {
15        fill(all(dist), -1), dist[s] = 0;
16        queue<int> q({s});
17        while (si(q)) {
18            int u = q.front(); q.pop();
19            for (auto &e : g[u])
20                if (dist[e.v] == -1 && e.f < e.c)
21                    dist[e.v] = dist[u] + 1, q.push(e.v);
22        }
23        return dist[t] != -1;
24    }
25    ll dfs(int u, ll cap = INF) {
26        if (u == t) return cap;
27        for (int &i = ptr[u]; i < si(g[u]); ++i) {
28            auto &e = g[u][i];
29            if (e.f < e.c && dist[e.v] == dist[u] + 1) {
    
```



```

30         ll flow = dfs(e.v, min(cap, e.c - e.f));
31         if (flow) {
32             e.f += flow, g[e.v][e.r].f -= flow;
33             return flow;
34         }
35     }
36 }
37 return 0;
38 }
39 ll maxflow() {
40     ll res = 0;
41     while (bfs()) {
42         fill(all(ptr), 0);
43         while (ll flow = dfs(s)) res += flow;
44     }
45     return res;
46 }
47 void reset() { for (auto &v : g) for (auto &e : v) e.f = 0; }
48 };

```

7.3. Min-cost Max-flow

Algoritmo: tira camino mínimo hasta encontrar el flujo buscado. Usa SPFA (Bellman-Ford más inteligente, con mejor tiempo promedio) porque resulta en la mejor complejidad.

Complejidad: $O(V^2E^2)$.

```

1 struct MCF {
2     const ll INF = 1e18;
3     int n; vector<vi> adj;
4     vector<vll> cap, cost;
5
6     MCF(int _n) : n(_n) {
7         adj.assign(n, vi());
8         cap.assign(n, vll(n));
9         cost.assign(n, vll(n));
10    }
11
12    void addEdge(int u, int v, ll _cap, ll _cost) {
13        cap[u][v] = _cap;
14        adj[u].pb(v), adj[v].pb(u);
15        cost[u][v] = _cost, cost[v][u] = -_cost;
16    }

```

```

17
18 void shortest_paths(int s, vll &dist, vi &par) {
19     par.assign(n, -1);
20     vector<bool> inq(n);
21     queue<int> q; q.push(s);
22     dist.assign(n, INF), dist[s] = 0;
23     while (!q.empty()) {
24         int u = q.front(); q.pop();
25         inq[u] = false;
26         for (int v : adj[u]) {
27             if (cap[u][v] > 0 && dist[v] > dist[u] + cost[u][v]) {
28                 dist[v] = dist[u] + cost[u][v], par[v] = u;
29                 if (!inq[v]) inq[v] = true, q.push(v);
30             }
31         }
32     }
33 }
34
35 ll min_cost_flow(ll k, int s, int t) {
36     vll dist; vi par;
37     ll flow = 0, total = 0;
38     while (flow < k) {
39         shortest_paths(s, dist, par);
40         if (dist[t] == INF) break;
41         // find max flow on that path
42         ll f = k - flow;
43         int cur = t;
44         while (cur != s) {
45             int p = par[cur];
46             f = min(f, cap[p][cur]);
47             cur = p;
48         }
49         // apply flow
50         flow += f, total += f * dist[t], cur = t;
51         while (cur != s) {
52             int p = par[cur];
53             cap[p][cur] -= f;
54             cap[cur][p] += f;
55             cur = p;
56         }
57     }
58     return flow < k ? -1 : total;
59 }

```

```
60 |};
```

7.4. Flujo con demandas

Problema: se pide que $d(e) \leq f(e) \leq c(e)$.

Flujo arbitrario: transformar red de la siguiente forma. Agregar nueva fuente s' y nuevo sumidero t' , arcos nuevos de s' a todos los demás nodos, arcos nuevos desde todos los nodos a t' , y un arco de t a s . Definimos la nueva función de capacidad c' como:

- $c'((s', v)) = \sum_{u \in V} d((u, v))$ para cada arco (s', v) .
- $c'((v, t')) = \sum_{w \in V} d((v, w))$ para cada arco (v, t') .
- $c'((u, v)) = c((u, v)) - d((u, v))$ para cada arco (u, v) en la red original.
- $c'((t, s)) = \infty$

Flujo mínimo: hacer búsqueda binaria sobre la capacidad de la arco (t, s) , viendo que se satisfaga la demanda.

8. Template

```
1 | #include <bits/stdc++.h>
2 | using namespace std;
3 |
4 | #ifdef LOCAL
5 |     #define D(a) cerr << #a << " = " << a << endl
6 | #else
7 |     #define D(a) 8
8 | #endif
9 | #define fastio ios_base::sync_with_stdio(0); cin.tie(0)
10 | #define dforsn(i,s,n) for(int i=int(n-1);i>=int(s);i--)
11 | #define forsn(i,s,n) for(int i=int(s);i<int(n);i++)
12 | #define all(a) (a).begin(),(a).end()
13 | #define dforn(i,n) dforsn(i,0,n)
14 | #define forn(i,n) forsn(i,0,n)
15 | #define si(a) int((a).size())
16 | #define pb emplace_back
17 | #define mp make_pair
18 | #define snd second
19 | #define fst first
20 | #define endl '\n'
```

```
21 | using pii = pair<int,int>;
22 | using vi = vector<int>;
23 | using ll = long long;
24 |
25 | int main() {
26 |     fastio;
27 |
28 |     return 0;
29 | }
```

9. vimrc

```
1 | colo desert
2 | se nu
3 | se nornu
4 | se acd
5 | se ic
6 | se sc
7 | se si
8 | se cin
9 | se ts=4
10 | se sw=4
11 | se sts=4
12 | se et
13 | se spr
14 | se cb=unnamedplus
15 | se nobk
16 | se nowb
17 | se noswf
18 | se cc=80
19 | map j gj
20 | map k gk
21 | aug cpp
22 |     au!
23 |     au FileType cpp map <f9> :w<CR> :!g++ -Wno-unused-result -
24 |         D_GLIBCXX_DEBUG -Wconversion -Wshadow -Wall -Wextra -O2 -DLOCAL
25 |         -std=c++17 -g3 "%" -o "%:p:r" <CR>
26 |     au FileType cpp map <f5> :! "%:p:r" < a.in <CR>
27 |     au FileType cpp map <f6> :! "%:p:r" <CR>
28 | aug END
29 | nm <c-h> <c-w><c-h>
30 | nm <c-j> <c-w><c-j>
31 | nm <c-k> <c-w><c-k>
```

```
30 nm <c-l> <c-w><c-l>
31 vm > >gv
32 vm < <gv
33 nn <silent> [b :bp<CR>
34 nn <silent> ]b :bn<CR>
35 nn <silent> [B :bf<CR>
36 nn <silent> ]B :bl<CR>
```

10. Misc

```
1 #pragma GCC optimize ("O3")//("avx,avx2,fma")
2
3 Random numbers:
4 mt19937_64 rng(time(0)); //if TLE use 32 bits: mt19937
5 ll rnd(ll a, ll b) { return a + rng()%(b-a+1); }
6 getline(cin,str);
7 // Make an extra call if we previously read another thing from the input
  stream
8 cout << fixed << setprecision(n);
9 cout << setw(n) << setfill('0');
10
11 // #include <sys/resource.h>
12 struct rlimit rl;
13 getrlimit(RLIMIT_STACK, &rl);
14 rl.rlim_cur = rl.rlim_max;
15 setrlimit(RLIMIT_STACK, &rl);
16
17 C++11:
18 to_string(num) // returns a string with the representation of num
19 stoi,stoll,stod,stold // string to int,ll,double & long double
  respectively
```