1. Estructuras

1.1. Segment Tree (Lazy)

```
struct Lazy {
    static const int C = 0; // Neutral for sum: 0
    int val; Lazy(int v=C) : val(v) {}
    bool dirty() { return val != C; }
    void clear() { val = C; }
    void update(const Lazy &o) { val += o.val; } // Update: sum
};
struct Node {
    int val; Node(int v=INF) : val(v) {} // Neutral for min: INF
    Node operator+(const Node &o) { return min(val, o.val); } // Query:
        min
    void update(const Lazy &o, int sz) { val += o.val; } // Update: sum
};
template <class T, class D>
struct RMQ { // ops O(lg n), [0, n)
    vector<T> t; vector<D> d; int n;
  T& operator[](int p){ return t[p+n]; }
  RMQ(int sz) {
        n = 1; while (n < sz) n *= 2;
        t.resize(2*n), d.resize(2*n);
    }
    void build() { dforsn(i, 1, n) t[i] = t[2*i] + t[2*i + 1]; }
  void push(int x, int sz) {
    if (d[x].dirty()){
            t[x].update(d[x], sz);
      if (sz > 1) d[2*x].update(d[x]), d[2*x + 1].update(d[x]);
        d[x].clear();
    }
  }
  T get(int i, int j) { return get(i, j, 1, 0, n); }
  T get(int i, int j, int x, int a, int b) {
    if (j <= a || i >= b) return T();
    push(x, b-a);
    if (i <= a && b <= j) return t[x];
    int c = (a + b) / 2;
        return get(i, j, 2*x, a, c) + get(i, j, 2*x + 1, c, b);
  }
  void update(int i, int j, const D &v) { update(i, j, v, 1, 0, n); }
  void update(int i, int j, const D &v, int x, int a, int b) {
    push(x, b-a);
    if (j <= a || i >= b) return;
    if (i <= a && b <= j) { d[x].update(v), push(x, b-a); return; }
    int c = (a + b) / 2;
    update(i, j, v, 2*x, a, c), update(i, j, v, 2*x + 1, c, b);
        t[x] = t[2*x] + t[2*x + 1];
  }
};
// Use: RMQ<Node, Lazy> rmq(n); forn(i, n) cin >> rmq[i].val; rmq.build
    ();
```

1.2. Treap

```
typedef pii Value; // pii(val, id)
typedef struct node *pnode;
struct node {
    Value val, mini;
    int dirty;
    int prior, size;
    pnode l, r, parent;
    node(Value val):val(val), mini(val), dirty(0), prior(rand()), size
        (1), l(0), r(0), parent(0) {} // usar rand piola
};

void push(pnode p){ // propagar dirty a los hijos (aca para lazy)
    p->val.first += p->dirty;
    p->mini.first += p->dirty;
    if(p->l) p->l->dirty += p->dirty;
    if(p->r) p->r->dirty += p->dirty;
    p->dirty = 0;
}
static int size(pnode p){ return p ? p->size : 0; }
static Value mini(pnode p){ return p ? push(p), p->mini : pii(1e9, -1);
    }
// Update function and size from children's Value
void pull(pnode p){ // recalcular valor del nodo aca (para rmq)
    p->size = 1 + size(p->l) + size(p->r);
    p->mini = min(min(p->val, mini(p->l)), mini(p->r));//operacion del
        rmq!
    p->parent = 0;
    if(p->l) p->l->parent = p;
    if(p->r) p->r->parent = p;
}
```

```

//junta dos arreglos
pnode merge(pnode l, pnode r){
    if(!l || !r) return l ? l : r;
    push(l), push(r);
    pnode t;

    if(l->prior < r->prior) l->r=merge(l->r, r), t = l;
    else r->l=merge(l, r->l), t = r;

    pull(t);
    return t;
}

//parte el arreglo en dos, si(l)==tam
void split(pnode t, int tam, pnode &l, pnode &r){
    if(!t) return void(l = r = 0);
    push(t);

    if(tam <= size(t->l)) split(t->l, tam, l, t->l), r = t;
    else split(t->r, tam - 1 - size(t->l), t->r, r), l = t;

    pull(t);
}

pnode at(pnode t, int pos){
    if(!t) exit(1);
    push(t);

    if(pos == size(t->l)) return t;
    if(pos < size(t->l)) return at(t->l, pos);

    return at(t->r, pos - 1 - size(t->l));
}
int getpos(pnode t){ // inversa de at
    if(!t->parent) return size(t->l);

    if(t == t->parent->l) return getpos(t->parent) - size(t->r) - 1;

    return getpos(t->parent) + size(t->l) + 1;
}

void split(pnode t, int i, int j, pnode &l, pnode &m, pnode &r){
    split(t, i, l, t), split(t, j-i, m, r);
}
Value get(pnode &p, int i, int j){ // like rmq
    pnode l, m, r;

    split(p, i, j, l, m, r);
    Value ret = mini(m);
    p = merge(l, merge(m, r));

    return ret;
}

void print(const pnode &t){ // for debugging
    if(!t) return;
    push(t);
    print(t->l);
    cout << t->val.first << ' ';
    print(t->r);
}
```

1.3. Convex Hull Trick

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

```
    ll eval(ll x) {
        while (at(p).i < x) p++;
        return at(p).a * x + at(p).b;
    }
};
```

1.4. Convex Hull Trick (Dynamic)

```
// Default is max, change a,b to -a,-b and negate the result for min
// If the lines use real vals change div by a/b and the comparisons
struct Line {
    ll a, b; mutable ll p;
    bool operator<(const Line& o) const { return a < o.a; }
    bool operator<(ll x) const { return p < x; }
};
struct CHT : multiset<Line, less<>> {
    ll div(ll a, ll b) { return a/b - ((a^b) < 0 && a % b); } // floor(a/b)
    bool isect(iterator x, iterator y) {
        if (y == end()) return x->p = INF, false;
        if (x->a == y->a) x->p = x->b > y->b ? INF : -INF;
        else x->p = div(y->b - x->b, x->a - y->a);
        return x->p >= y->p;
    }
    void add(ll a, ll b) {
        auto z = insert({a, b, 0}), y = z++, x = y;
        while (isect(y, z)) z = erase(z);
        if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
        while ((y = x) != begin() && (--x)->p >= y->p) isect(x, erase(y));
    }
    ll eval(ll x) {
        auto l = *lower_bound(x);
        return l.a * x + l.b;
    }
};
```

2. Strings

2.1. Hash

```
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());
struct BasicHashing {
    int mod, base; vi h, pot;
    BasicHashing() {
        mod = uniform_int_distribution<>(int(1e9), int(15e8))(rnd);
        bool prime;
        do {
            mod++, prime = true;
            for (ll d = 2; prime && d*d <= mod; ++d)
                if (mod % d == 0) prime = false;
        } while (!prime);
        base = uniform_int_distribution<>(256, mod-1)(rnd);
    }
    void process(const string &s) {
        int n = si(s); h = vi(n+1), pot = vi(n+1);
        h[0] = 0; forn(i, n) h[i+1] = int((h[i] * ll(base) + s[i]) % mod);
        pot[0] = 1; forn(i, n) pot[i+1] = int(pot[i] * ll(base) % mod);
    }
    int hash(int i, int j) { // [ )
        int res = int(h[j] - ll(h[i]) * pot[j-i] % mod);
        return res < 0 ? res + mod : res;
    }
    int hash(const string &s) {
        int res = 0;
        for (char c : s) res = int((res * ll(base) + c) % mod);
        return res;
    }
    int append(int a, int b, int szb) {
        return int((ll(a) * pot[szb] + b) % mod);
    }
};
struct Hashing {
    BasicHashing h1, h2;
    void process(const string &s) { h1.process(s), h2.process(s); }
    pii hash(int i, int j) { return {h1.hash(i, j), h2.hash(i, j)}; }
    pii hash(const string &s) { return {h1.hash(s), h2.hash(s)}; }
    pii append(pii &a, pii &b, int szb) {
        return {h1.append(a.fst, b.fst, szb), h2.append(a.snd, b.snd, szb)};
    }
};
```

2.2. Manacher

Definición: permite calcular todas las substrings de una string s que son palíndromos. Para ello, mantiene un arreglo *odd* tal que $odd[i]$ almacena la longitud del palíndromo impar maximal con centro en i. Análogamente mantiene un arreglo *even* tal que $even[i]$ guarda la longitud del palíndromo par maximal con centro derecho en i.
Explicación del algoritmo: muy similar al algoritmo para calcular la *función Z*, mantiene el palíndromo que termina más a la derecha entre todos los palíndromos ya detectados con rango $[l, r]$. Utiliza la información ya calculada si i está dentro de $[l, r]$, y luego corre el algoritmo trivial. Cada vez que se corre el algoritmo trivial, r se incrementa en 1 y r jamás decrece.

```
void manacher(string s, vi &odd, vi &even) {
    int n = si(s);
    s = "@" + s + "$";
    odd = vi(n), even = vi(n);
    int l = 0, r = -1;
    forn(i, n) {
        int k = i > r ? 1 : min(odd[l+r-i], r-i+1);
        while (s[i+1-k] == s[i+1+k]) k++;
        odd[i] = k--;
        if (i+k > r) l = i-k, r = i+k;
    }
    l = 0, r = -1;
    forn(i, n) {
        int k = i > r ? 0 : min(even[l+r-i+1], r-i+1);
        while (s[i-k] == s[i+1+k]) k++;
        even[i] = k--;
        if (i+k > r) l = i-k-1, r = i+k;
    }
}
```

2.3. KMP

```
// pre[i] = max border of s[0..i]
vi prefix_function(string &s) {
    int n = si(s), j = 0; vi pre(n);
    forsn(i, 1, n) {
        while (j > 0 && s[i] != s[j]) j = pre[j-1];
        pre[i] = s[i] == s[j] ? ++j : j;
    }
    return pre;
}

vi find_occurrences(string &s, string &t) { // occurrences of s in t
    vi pre = prefix_function(s), res;
    int n = si(s), m = si(t), j = 0;
    forn(i, m) {
        while (j > 0 && t[i] != s[j]) j = pre[j-1];
        if (t[i] == s[j]) j++;
        if (j == n) res.pb(i-n+1), j = pre[j-1];
    }
    return res;
}

// (i chars match, next_char = c) -> (aut[i][c] chars match)
vector<vi> kmp_automaton(string &s) {
    s += '#'; int n = si(s);
    vi pre = prefix_function(s);
    vector<vi> aut(n, vi(26)); // alphabet = lowercase letters
    forn(i, n) forn(c, 26) {
        if (i > 0 && 'a' + c != s[i])
            aut[i][c] = aut[pre[i-1]][c];
        else
            aut[i][c] = i + ('a' + c == s[i]);
    }
    return aut;
}
```

2.4. Trie

```
struct Trie {
    int u = 0, ws = 0;
    map<char, Trie*> c;
    Trie() {}
    void add(const string &s) {
        Trie *x = this;
        forn(i, si(s)){
            if(!x->c.count(s[i])) x->c[s[i]] = new Trie();
            x = x->c[s[i]];
            x->u++;
        }
        x->ws++;
    }
    int find(const string &s) {
```

```
        Trie *x = this;
        forn(i, si(s)){
            if (x->c.count(s[i])) x = x->c[s[i]];
            else return 0;
        }
        return x->ws;
    }
    void erase(const string &s) {
        Trie *x = this, *y;
        forn(i, si(s)){
            if (x->c.count(s[i])) y = x->c[s[i]], y->u--;
            else return;
            if (!y->u){
                x->c.erase(s[i]);
                return;
            }
            x = y;
        }
        x->ws--;
    }
  void print(string tab = "") {
    for (auto &i : c) {
      cerr << tab << i.fst << endl;
      i.snd->print(tab + "--");
    }
  }
};
```

2.5. Suffix Array (corto, nlog2n)

```
const int MAXN = 2e5+10;
pii sf[MAXN];
bool comp(int lhs, int rhs) {return sf[lhs] < sf[rhs];}
struct SuffixArray {
  //sa guarda los indices de los sufijos ordenados
    int sa[MAXN], r[MAXN];
    void init(const string &a) {
        int n = si(a);
        forn(i,n) r[i] = a[i];
        for(int m = 1; m < n; m <<= 1) {
      forn(i, n) sa[i]=i, sf[i] = mp(r[i], i+m<n? r[i+m]:-1);
            stable_sort(sa, sa+n, comp);
            r[sa[0]] = 0;
            forsn(i, 1, n) r[sa[i]]= sf[sa[i]] != sf[sa[i - 1]] ? i : r[
                sa[i-1]];
        }
    }
} sa;

int main(){
    string in;
  while(cin >> in){
    sa.init(in, si(in));
    forn(i, si(in)) {
            forn(k, sa.sa[i]) cout << '␣';
            cout << in.substr(sa.sa[i]) << '\n';
        }
        cout << endl;
  }
  return 0;
}
```

2.6. String Matching With Suffix Array

```
//returns (lowerbound, upperbound) of the search
pii stringMatching(string P){ //O(si(P)lgn)
  int lo=0, hi=n-1, mid=lo;
  while(lo<hi){
    mid=(lo+hi)/2;
    int res=s.compare(sa[mid], si(P), P);
    if(res>=0) hi=mid;
    else lo=mid+1;
  }
  if(s.compare(sa[lo], si(P), P)!=0) return pii(-1, -1);
  pii ans; ans.first=lo;
  lo=0, hi=n-1, mid;
  while(lo<hi){
    mid=(lo+hi)/2;
    int res=s.compare(sa[mid], si(P), P);
    if(res>0) hi=mid;
    else lo=mid+1;
  }
  if(s.compare(sa[hi], si(P), P)!=0) hi--;
    // para verdadero upperbound sumar 1
  ans.second=hi;
  return ans;
```

2.7. LCP (Longest Common Prefix)

```

//Calculates the LCP between consecutives suffixes in the Suffix Array.
//LCP[i] is the length of the LCP between sa[i] and sa[i-1]
int LCP[MAXN], phi[MAXN], PLCP[MAXN];
void computeLCP(){//O(n)
  phi[sa[0]]=-1;
  forsn(i,1,n) phi[sa[i]]=sa[i-1];
  int L=0;
  forn(i,n){
    if (phi[i]==-1) {PLCP[i]=0; continue;}
    while (s[i+L]==s[phi[i]+L]) L++;
    PLCP[i]=L;
    L=max(L-1, 0);
  }
  forn(i,n) LCP[i]=PLCP[sa[i]];
```

2.8. Aho-Corasick

```
const int K = 26;

// si el alfabeto es muy grande, adaptar usando map para next y go
// es posible almacenar los indices de las palabras en terminal usando
    vector<int>
struct Vertex {
    int next[K];
    int terminal = 0;
    int p = -1;
    char pch;
    int link = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t;

void aho_init() { // INICIALIZAR!
    t.clear(); t.pb(Vertex());
}

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = si(t);
            t.pb(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].terminal++;
}

int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}
```

2.9. Suffix Automaton

```
struct state {
  int len, link;
  map<char,int> next;
```

```
  state() { }
};
const int MAXLEN = 1e5+10;
state st[MAXLEN*2];
int sz, last;
void sa_init() {
  forn(i,sz) st[i].next.clear();
  sz = last = 0;
  st[0].len = 0;
  st[0].link = -1;
  ++sz;
}
// Es un DAG de una sola fuente y una sola hoja
// cantidad de endpos = cantidad de apariciones = cantidad de caminos de la clase al nodo terminal
// cantidad de miembros de la clase = st[v].len-st[st[v].link].len (v>0) = caminos del inicio a la clase
// 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
void sa_extend (char c) {
  int cur = sz++;
  st[cur].len = st[last].len + 1;
  // en cur agregamos la posicion que estamos extendiendo
  // podria agregar tambien un identificador de las cadenas a las cuales pertenece (si hay varias)
  int p;
  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=='$')
    st[p].next[c] = cur;
  if (p == -1)
    st[cur].link = 0;
  else {
    int q = st[p].next[c];
    if (st[p].len + 1 == st[q].len)
      st[cur].link = q;
    else {
      int clone = sz++;
      st[clone].len = st[p].len + 1;
      st[clone].next = st[q].next;
      st[clone].link = st[q].link;
      for (; p!=-1 && st[p].next.count(c) && st[p].next[c]==q; p=st[p].link)
        st[p].next[c] = clone;
      st[q].link = st[cur].link = clone;
    }
  }
  last = cur;
}
```

2.10. Z Function

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

3. Geometría

3.1. Epsilon

```
const double EPS = 1e-9;
#define less(a,b)          ((a) < (b) - EPS)
#define greater(a,b)       ((a) > (b) + EPS)
#define less_equal(a,b)    ((a) < (b) + EPS)
#define greater_equal(a,b) ((a) > (b) - EPS)
#define equal(a,b)         (abs((a) - (b)) < EPS)
```

3.2. Point

```
const double EPS = 1e-9;
struct Point {
  double x, y;
  Point(double _x=0, double _y=0) : x(_x),y(_y) {}
  Point operator+(Point a) { return Point(x + a.x, y + a.y); }
  Point operator-(Point a) { return Point(x - a.x, y - a.y); }
  Point operator+(double a) { return Point(x + a, y + a); }
  Point operator*(double a) { return Point(x*a, y*a); }
  Point operator/(double a) { return Point(x/a, y/a); }
  double norm() { return sqrt(x*x + y*y); }
  double norm2() { return x*x + y*y; }
```

```
    // Dot product:
  double operator*(Point a){ return x*a.x + y*a.y; }
  // Magnitude of the cross product (if a is less than 180 CW from b, a^
      b > 0):
  double operator^(Point a) { return x*a.y - y*a.x; }
  // Returns true if this point is at the left side of line qr:
  bool left(Point q, Point r) { return ((q - *this) ^ (r - *this)) > EPS
      ; }
  bool operator<(const Point &a) const {
        return x < a.x - EPS || (abs(x - a.x) < EPS && y < a.y - EPS);
    }
    bool operator==(Point a) {
        return abs(x - a.x) < EPS && abs(y - a.y) < EPS;
    }
};
typedef Point vec;
double dist(Point a, Point b) { return (b-a).norm(); }
double dist2(Point a, Point b) { return (b-a).norm2(); }
double angle(Point a, Point o, Point b){ // [-pi, pi]
  Point oa = a-o, ob = b-o;
  return atan2(oa^ob, oa*ob);
}
// Rotate around the origin:
Point CCW90(Point p) { return Point(-p.y, p.x); }
Point CW90(Point p) { return Point(p.y, -p.x); }
Point CCW(Point p, double t){ // rads
  return Point(p.x*cos(t) - p.y*sin(t), p.x*sin(t) + p.y*cos(t));
}
// Sorts points in CCW order about origin, points on neg x-axis come
    last
// To change pivot to point x, just substract x from all points and then
     sort
bool half(Point &p) { return p.y == 0 ? p.x < 0 : p.y > 0; }
bool angularOrder(Point &x, Point &y) {
  bool X = half(x), Y = half(y);
  return X == Y ? (x ^ y) > 0 : X < Y;
}
```

3.3. Orden radial de puntos

```
// Absolute order around point r
struct RadialOrder {
  pto r;
  RadialOrder(pto _r) : r(_r) {}
  int cuad(const pto &a) const {
    if(a.x > 0 && a.y >= 0) return 0;
    if(a.x <= 0 && a.y > 0) return 1;
    if(a.x < 0 && a.y <= 0) return 2;
    if(a.x >= 0 && a.y < 0) return 3;
    return -1;
  }
  bool comp(const pto &p1, const pto &p2) const {
    int c1 = cuad(p1), c2 = cuad(p2);
    if (c1 == c2) return (p1 ^ p2) > 0;
        else return c1 < c2;
  }
    bool operator()(const pto &p1, const pto &p2) const {
        return comp(p1 - r, p2 - r);
    }
};
```

3.4. Point in Poly

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

3.5. Convex Hull

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

```
  S.pop_back();
  int k = si(S);
  dforn(i, si(P)) { // upper hull
    while (si(S) >= k+2 && S[si(S)-1].left(S[si(S)-2], P[i])) S.pop_back
        ();
    S.pb(P[i]);
  }
  S.pop_back();
}
```

3.6. Cut Polygon

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

3.7. Heron's formula

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

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.

```
int cost(int l, int r); // Implementar

// Intervalos: cerrado, cerrado.
// Modificar tipos, comparador y neutro (INF). Revisar caso base (i, i
    +1).
const ll INF = 1e18;
ll knuth(int n) {
    vector<vi> opt(n, vi(n));
    vector<vll> dp(n, vll(n));

    // Casos base
    forn(i, n-2) dp[i][i+2] = cost(i, i+2), opt[i][i+2] = i+1;

    // Casos recursivos
    forsn(len, 3, n+1) {
        forn(l, n-len) {
            int r = l+len;

            dp[l][r] = INF;
            forsn(k, opt[l][r-1], opt[l+1][r]+1) {
                ll val = dp[l][k] + dp[k][r] + cost(l, r);
                if (val < dp[l][r]) {
                    dp[l][r] = val;
                    opt[l][r] = k;
                }
            }
        }
    }

    return dp[0][n-1];
}
```

4.2. Divide & Conquer

Problema de ejemplo: dado un arreglo de n números con valores $a_1, a_1, \ldots, a_n$, 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 $dp(i, m, j_{right}, opt, opt_{right})$.

```
// Modificar: tipos, operacion (max, min), neutro (INF), funcion de
    costo.
const ll INF = 1e18;

ll cost(int i, int j); // Implementar. Costo en rango [i, j).

vector<ll> dp_before, dp_cur;
// compute dp_cur[l, r)
void compute(int l, int r, int optl, int optr)
{
    if (l == r) return;
    int mid = (l + r) / 2;
    pair<ll, int> best = {INF, -1};

    forsn(k, optl, min(mid, optr))
        best = min(best, {dp_before[k] + cost(k, mid), k});

    dp_cur[mid] = best.first;
    int opt = best.second;

    compute(l, mid, optl, opt + 1);
    compute(mid + 1, r, opt, optr);
}

ll dc_opt(int n, int k) {
    dp_before.assign(n+1, INF); dp_before[0] = 0;
    dp_cur.resize(n+1); // Cuidado, dp_cur[0] = 0. No molesta porque no
        se elige.

    while (k--) {
        compute(1, n+1, 0, n); // Parametros tal que por lo menos 1 en
            cada subarreglo.
        dp_before = dp_cur;
    }

    return dp_cur[n];
}
```

5. Matemática

5.1. Teoría de números

5.1.1. Funciones multiplicativas, función de Möbius

Una funcion $f(n)$ 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: $f(n) = f(p_1^{r_1})f(p_2^{r_2})\ldots f(p_k^{r_k})$.

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

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

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. Combinatorios

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

```
int combinations(int n, int k){
    return divide(fact[n], mul(fact[n-k], fact[k]));
}

const int MAXC = 1e3+1;
int C[MAXC][MAXC];
void combinations() {
    forn(i, MAXC) {
        C[i][0] = C[i][i] = 1;
        forsn(k, 1, i) C[i][k] = add(C[i-1][k], C[i-1][k-1]);
    }
}
```

5.2.3. Lucas Theorem

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

where $m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0$,
and $n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$
$\binom{m}{n} = 0$ if $m < n$.

```
// Calcula C(n,k) % p teniendo C[p][p] precalculado, p primo
ll lucas(ll n, ll k, int p) {
    ll ans = 1;
    while (n + k) {
        ans = (ans * C[n % p][k % p]) % p;
        n /= p, k /= p;
    }
    return ans;
}
```

5.2.4. Stirling

$\left\{ {n \atop k} \right\}$ = cantidad de formas de particionar un conjunto de n elementos en m subconjuntos no vacíos.
$\left\{ {n+1 \atop k} \right\} = k \left\{ {n \atop k} \right\} + \left\{ {n \atop k-1} \right\}$
for $k > 0$ with initial conditions
$\left\{ {0 \atop 0} \right\} = 1$ and $\left\{ {n \atop 0} \right\} = \left\{ {0 \atop n} \right\} = 0$ for $n > 0$.

```
const int MAXS = 1e3+1;
int S[MAXS][MAXS];
void stirling() {
  S[0][0] = 1;
  forsn(i, 1, N) S[i][0] = S[0][i] = 0;
  forsn(i, 1, N) forsn(j, 1, N)
    S[i][j] = add(mul(S[i-1][j], j), S[i-1][j-1]);
}
```

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

5.2.5. Bell

B_n = cantidad de formas de particionar un conjunto de n elementos en subconjuntos no vacíos.
$B_0 = B_1 = 1$

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

$$B_n = \sum_{k=0}^{n} \left\{ {n \atop k} \right\}.$$

```
const int MAXB = 1e3+1;
int B[MAXB][MAXB];
void bell() {
    B[0] = 1;
    forsn(i, 1, MAXB) forn(k, i)
        B[i] = add(B[i], mul(C[i-1][k], B[k]));
```

```
}
```

5.2.6. Eulerian

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

5.2.7. 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. Euclides extendido

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

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

5.4. Inversos

```
const int MAXM = 15485867; // Tiene que ser primo
ll inv[MAXM]; //inv[i]*i=1 M M
void calc(int p){//O(p)
  inv[1]=1;
  forsn(i, 2, p) inv[i]= p-((p/i)*inv[p%i])%p;
}
// Llamar calc(MAXM);

int inv(int x){//O(log x)
  return pot(x, eulerphi(M)-1);//si M no es primo(sacar a mano)
  return pot(x, M-2);//si M es primo
}

// Inversos con euclides en O(log(x)) sin precomputo:
// extendedEuclid(a, -m).fst (si coprimos a y m)
```

5.5. 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.

```
pair<pair<ll,ll>,pair<ll,ll> > diophantine(ll a,ll b, ll r) {
  //a*x+b*y=r where r is multiple of gcd(a,b);
  ll d=gcd(a,b);
  a/=d; b/=d; r/=d;
  auto p = extendedEuclid(a,b);
  p.fst*=r; p.snd*=r;
  assert(a*p.fst+b*p.snd==r);
  return mp(p,mp(-b,a)); // solutions: (p.fst - b*k, p.snd + a*k)
                //== (res.fst.fst + res.snd.fst*k, res.fst.snd + res.snd
                    .snd*k)
}
```

5.6. 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 $lcm(n_i)$.

```
#define mod(a,m) ((a)%(m) < 0 ? (a)%(m)+(m) : (a)%(m)) // evita overflow
     al no sumar si >= 0
typedef tuple<ll,ll,ll> ec;
pair<ll,ll> sol(ec c){ //requires inv, diophantine
    ll a=get<0>(c), x1=get<1>(c), m=get<2>(c), d=gcd(a,m);
    if (d==1) return mp(mod(x1*inv(a,m),m), m);
    else return x1%d ? mp(-1LL,-1LL) : sol({a/d,x1/d,m/d});
}
pair<ll,ll> crt(vector< ec > cond) { // returns: (sol, lcm)
  ll x1=0,m1=1,x2,m2;
  for(auto t:cond){
    tie(x2,m2)=sol(t);
    if((x1-x2)%gcd(m1,m2))return mp(-1,-1);
    if(m1==m2)continue;
```

```
    ll k=diophantine(m2,-m1,x1-x2).fst.snd,l=m1*(m2/gcd(m1,m2));
    x1=mod(m1*mod(k, l/m1)+x1,l);m1=l; // evita overflow con prop modulo
  }
  return sol(make_tuple(1,x1,m1));
} //cond[i]={ai,bi,mi} ai*xi=bi (mi); assumes lcm fits in ll
```

5.7. Simpson

```
double integral(double a, double b, int n=10000) {//O(n), n=cantdiv
  double area=0, h=(b-a)/n, fa=f(a), fb;
  forn(i, n){
    fb=f(a+h*(i+1));
    area+=fa+ 4*f(a+h*(i+0.5)) +fb, fa=fb;
  }
  return area*h/6.;}
```

5.8. Fraction

```
struct frac{
  int p,q;
  frac(int p=0, int q=1):p(p),q(q) {norm();}
  void norm(){
    int a = gcd(p,q);
    p/=a, q/=a;
    if(q < 0) q=-q, p=-p;}
  frac operator+(const frac& o){
    int a = gcd(q,o.q);
    return frac(add(mul(p,o.q/a), mul(o.p,q/a)), mul(q,o.q/a));}
  frac operator-(const frac& o){
    int a = gcd(q,o.q);
    return frac(sub(mul(p,o.q/a), mul(o.p,q/a)), mul(q,o.q/a));}
  frac operator*(frac o){
    int a = gcd(q,o.p), b = gcd(o.q,p);
    return frac(mul(p/b,o.p/a), mul(q/a,o.q/b));}
  frac operator/(frac o){
    int a = gcd(q,o.q), b = gcd(o.p,p);
    return frac(mul(p/b,o.q/a), mul(q/a,o.p/b));}
  bool operator<(const frac &o) const{return ll(p)*o.q < ll(o.p)*q;}
  bool operator==(frac o){return p==o.p && q==o.q;}
  bool operator!=(frac o){return p!=o.p || q!=o.q;}
};
```

5.9. Matrices

```
struct Mat {
    vector<vector<double> > rows;
    Mat(int n): rows(n, vector<double>(n)) {}
    Mat(int n, int m): rows(n, vector<double>(m)) {}

    vector<double> &operator[](int f) { return rows[f]; }
    int size() { return si(rows); }

    Mat operator+(Mat &b) { // this de n x m entonces b de n x m
        Mat m(si(rows), si(rows[0]));
        forn(i, si(rows)) forn(j, si(rows[0])) m[i][j] = rows[i][j] + b[
            i][j];
        return m;
    }
    Mat operator*(Mat &b) { // this de n x m entonces b de m x t
        int n = si(rows), m = si(rows[0]), t = si(b[0]);
        Mat mat(n, t);
        forn(i, n) forn(j, t) forn(k, m) mat[i][j] += rows[i][k] * b[k][
            j];
        return mat;
    }
    Mat operator^(int e) { // this debe ser matriz cuadrada
        int n = si(rows);
        Mat res(n); forn(i, n) res[i][i] = 1;

        Mat base = *this;
        while (e > 0) {
            if (e % 2 == 1) res = res * base;
            base = base * base;
            e /= 2;
        }

        return res;
    }
};
// to calculate determinants, use determinant.cpp
```

5.10. Determinante

```
double determinant(Mat m) { // do gaussian elimination and calculate
    determinant
    double det = 1;
    int n = si(m);
```

```
    forn(i, n) { // for each col
        int k = i;
        forsn(j, i+1, n) // row with largest abs val to avoid floating
            point errors
            if (abs(m[j][i]) > abs(m[k][i]))
                k = j;
        if (abs(m[k][i]) < EPS) return 0;
        swap(m[i], m[k]); // move pivot row
        if (i != k) det = -det;
        det *= m[i][i];
        forsn(j, i+1, n) m[i][j] /= m[i][i]; // scale current row
        forn(j, n) if (j != i && abs(m[j][i]) > EPS) // zero out other
            rows
            forsn(k, i+1, n)
                m[j][k] -= m[i][k] * m[j][i];
    }
    return det;
}

// if mod 2, check gauss.cpp for a faster implementation
```

5.11. Sistemas de Ecuaciones Lineales - Gauss

```
const double EPS = 1e-9;
const int INF = 1e9; // it doesn't actually have to be infinity or a big
     number

int gauss(Mat mat, vector<double> &ans) { // returns number of solutions
    int n = si(mat);
    int m = n > 0 ? si(mat[0]) - 1 : 0;

    vi where(m, -1);
    for (int col = 0, row = 0; col < m && row < n; col++) { // for each
        col
        int sel = row;
        forsn(i, row, n) // row with largest abs val to avoid floating
            point errors
            if (abs(mat[i][col]) > abs(mat[sel][col]))
                sel = i;
        if (abs(mat[sel][col]) < EPS)
            continue;
        swap(mat[sel], mat[row]); // move pivot row
        where[col] = row;

        forn(i, n) if (i != row) { // zero out other rows
            double c = mat[i][col] / mat[row][col];
            forsn(j, col, m + 1)
                mat[i][j] -= mat[row][j] * c;
        }
        row++;
    }

    ans.assign(m, 0);
    forn(i, m)
        if (where[i] != -1)
            ans[i] = mat[where[i]][m] / mat[where[i]][i]; // calculate
                x_i
    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)
        double sum = 0;
        forn(j, m)
            sum += ans[j] * mat[i][j];
        if (abs(sum - mat[i][m]) > EPS)
            return 0;
    }

    forn(i, m)
        if (where[i] == -1)
            return INF;
    return 1;
}

// SPEED IMPROVEMENT IF MOD 2:
int gauss(vector<bitset<N>> a, int n, int m, bitset<N> &ans) {
    vi where(m, -1);
    for (int col = 0, row = 0; col < m && row < n; col++) {
        forsn(i, row, n)
            if (a[i][col]) {
                swap(a[i], a[row]);
                break;
            }
        if (!a[row][col])
            continue;
        where[col] = row;

```

```
        forn(i, n)
            if (i != row && a[i][col])
                a[i] ^= a[row];
        row++;
    }
    // The rest of implementation is the same as above
}
```

5.12. FFT y NTT

Base teórica (e intuición):
La **transformada de Fourier** mapea una función temporal a un dominio de frecuencias.
Podemos pensar que rotamos la función temporal alrededor de un círculo a diferentes frecuencias y calculamos la magnitud del centro de masa de la figura resultante; la función del dominio de frecuencias representa este mapeo.

$$F_n = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \xi_n & \xi_n^2 & \dots & \xi_n^{n-1} \\ 1 & \xi_n^2 & \xi_n^4 & \dots & \xi_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \xi_n^{n-1} & \xi_n^{2(n-1)} & \dots & \xi_n^{(n-1)(n-1)} \end{bmatrix}$$

$$y = F_n x$$

Donde ω_n es una raíz primitiva n-ésima de la unidad y $\xi_n = w_n^{n-1}$. La **transformada rápida de Fourier** se basa en que las raíces de la unidad cumplen la propiedad $\omega_{2n}^2 = \omega_n$. Por lo tanto:

$$F_n = \begin{bmatrix} F_{n/2} & D_{n/2}F_{n/2} \\ F_{n/2} & -D_{n/2}F_{n/2} \end{bmatrix} P_n$$

donde:

$$D_{n/2} = \begin{bmatrix} 1 & & & & \\ & \xi_n & & & \\ & & \xi_n^2 & & \\ & & & \ddots & \\ & & & & \xi_n^{n/2-1} \end{bmatrix}$$

y

$$P_n^T = \begin{bmatrix} e_0 & e_2 & e_4 & \dots & e_{n-2} & e_1 & e_3 & \dots & e_{n-1} \end{bmatrix}$$

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.
- Interpolar 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).

```
// N must be power of 2 !!!
// Tiene que entrar el resultado!!! (el producto, probablemente el doble
    de la entrada)
using tf = int;
using poly = vector<tf>;
// FFT
struct CD {
  double r,i;
  CD(double r=0, double i=0):r(r),i(i){}
  double real()const{return r;}
  void operator/=(const int c){r/=c, i/=c;}
};
CD operator*(const CD& a, const CD& b){
  return CD(a.r*b.r-a.i*b.i,a.r*b.i+a.i*b.r);}
```

```
CD operator+(const CD& a, const CD& b){return CD(a.r+b.r,a.i+b.i);}
CD operator-(const CD& a, const CD& b){return CD(a.r-b.r,a.i-b.i);}
const double pi=acos(-1.0);
// NTT
// M-1 needs to be a multiple of N !!
// tf TIENE que ser ll (si el modulo es grande)
// big mod and primitive root for NTT:
/*
const tf M=998244353,RT=3;
struct CD {
  tf x;
  CD(tf _x):x(_x){}
  CD(){}
};
CD operator*(const CD& a, const CD& b){return CD(mul(a.x,b.x));}
CD operator+(const CD& a, const CD& b){return CD(add(a.x,b.x));}
CD operator-(const CD& a, const CD& b){return CD(sub(a.x,b.x));}
vector<tf> rts(N+9,-1);
CD root(int n, bool to_inv){
  tf r=rts[n]<0?rts[n]=pot(RT,(M-1)/n):rts[n];
  return CD(to_inv?inv(r):r);
}
*/
CD cp1[N+9],cp2[N+9];
int R[N+9];
void dft(CD* a, int n, bool to_inv){
  forn(i,n)if(R[i]<i)swap(a[R[i]],a[i]);
  for(int m=2;m<=n;m*=2){
    double z=2*pi/m*(to_inv?-1:1); // FFT
    CD wi=CD(cos(z),sin(z)); // FFT
    // CD wi=root(m,to_inv); // NTT
    for(int j=0;j<n;j+=m){
      CD w(1);
      for(int k=j,k2=j+m/2;k2<j+m;k++,k2++){
        CD u=a[k];CD v=a[k2]*w;a[k]=u+v;a[k2]=u-v;w=w*wi;
      }
    }
  }
  if(to_inv)forn(i,n)a[i]/=n; // FFT
  //if(to_inv){ // NTT
  //  CD z(inv(n));
  //  forn(i,n)a[i]=a[i]*z;
  //}
}
poly multiply(poly& p1, poly& p2){
  int n=si(p1)+si(p2)+1;
  int m=1,cnt=0;
  while(m<=n)m+=m,cnt++;
  forn(i,m){R[i]=0;forn(j,cnt)R[i]=(R[i]<<1)|((i>>j)&1);}
  forn(i,m)cp1[i]=0,cp2[i]=0;
  forn(i,si(p1))cp1[i]=p1[i];
  forn(i,si(p2))cp2[i]=p2[i];
  dft(cp1,m,false);dft(cp2,m,false);
  forn(i,m)cp1[i]=cp1[i]*cp2[i];
  dft(cp1,m,true);
  poly res;
  n-=2;
  forn(i,n)res.pb((tf)floor(cp1[i].real()+0.5)); // FFT
  //forn(i,n)res.pb(cp1[i].x); // NTT
  return res;
}
```

```
//Polynomial division: O(n*log(n))
//Multi-point polynomial evaluation: O(n*log^2(n))
//Polynomial interpolation: O(n*log^2(n))

//Works with NTT. For FFT, just replace add,sub,mul,inv,divide
poly add(poly &a, poly &b){
    int n=si(a),m=si(b);
    poly ans(max(n,m));
    forn(i,max(n,m)){
        if(i<n) ans[i]=add(ans[i],a[i]);
        if(i<m) ans[i]=add(ans[i],b[i]);
    }
    while(si(ans)>1&&!ans.back())ans.pop_back();
    return ans;
}

/// B(0) != 0 !!!
poly invert(poly &b, int d){
    poly c = {inv(b[0])};
    while(si(c)<=d){
        int j=2*si(c);
        auto bb=b; bb.resize(j);
        poly cb=multiply(c,bb);
        forn(i,si(cb)) cb[i]=sub(0,cb[i]);
```

```
        cb[0]=add(cb[0],2);
        c=multiply(c,cb);
        c.resize(j);
    }
    c.resize(d+1);
    return c;
}

pair<poly,poly> divslow(poly &a, poly &b){
    poly q,r=a;
    while(si(r)>=si(b)){
        q.pb(divide(r.back(),b.back()));
        if(q.back()) forn(i,si(b)){
            r[si(r)-i-1]=sub(r[si(r)-i-1],mul(q.back(),b[si(b)-i-1]));
        }
        r.pop_back();
    }
    reverse(all(q));
    return {q,r};
}

pair<poly,poly> divide(poly &a, poly &b){ //returns {quotient,remainder}
    int m=si(a),n=si(b),MAGIC=750;
    if(m<n) return {{0},a};
    if(min(m-n,n)<MAGIC)return divslow(a,b);
    poly ap=a; reverse(all(ap));
    poly bp=b; reverse(all(bp));
    bp=invert(bp,m-n);
    poly q=multiply(ap,bp);
    q.resize(si(q)+m-n-si(q)+1,0);
    reverse(all(q));
    poly bq=multiply(b,q);
    forn(i,si(bq)) bq[i]=sub(0,bq[i]);
    poly r=add(a,bq);
    return {q,r};
}

vector<poly> tree;

void filltree(vector<tf> &x){
    int k=si(x);
    tree.resize(2*k);
    forsn(i,k,2*k) tree[i]={sub(0,x[i-k]),1};
    dforsn(i,1,k) tree[i]=multiply(tree[2*i],tree[2*i+1]);
}

vector<tf> evaluate(poly &a, vector<tf> &x){
    filltree(x);
    int k=si(x);
    vector<poly> ans(2*k);
    ans[1]=divide(a,tree[1]).snd;
    forsn(i,2,2*k) ans[i]=divide(ans[i>>1],tree[i]).snd;
    vector<tf> r; forn(i,k) r.pb(ans[i+k][0]);
    return r;
}

poly derivate(poly &p){
    poly ans(si(p)-1);
    forsn(i,1,si(p)) ans[i-1]=mul(p[i],i);
    return ans;
}

poly interpolate(vector<tf> &x, vector<tf> &y){
    filltree(x);
    poly p=derivate(tree[1]);
    int k=si(y);
    vector<tf> d=evaluate(p,x);
    vector<poly> intree(2*k);
    forsn(i,k,2*k) intree[i]={divide(y[i-k],d[i-k])};
    dforsn(i,1,k) {
        poly p1=multiply(tree[2*i],intree[2*i+1]);
        poly p2=multiply(tree[2*i+1],intree[2*i]);
        intree[i]=add(p1,p2);
    }
    return intree[1];
}
```

5.13. Programación lineal: 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.

```
const double EPS = 1e-5;
// if inequality is strictly less than (< vs <=), do not use EPS! this case is covered in the code
namespace Simplex {
    vi X,Y;
    vector<vector<double> > A;
    vector<double> b,c;
    double z;
    int n,m;
    void pivot(int x,int y){
        swap(X[y],Y[x]);
        b[x]/=A[x][y];
        forn(i,m)if(i!=y)A[x][i]/=A[x][y];
        A[x][y]=1/A[x][y];
        forn(i,n)if(i!=x&&abs(A[i][y])>EPS){
            b[i]-=A[i][y]*b[x];
            forn(j,m)if(j!=y)A[i][j]-=A[i][y]*A[x][j];
            A[i][y]=-A[i][y]*A[x][y];
        }
        z+=c[y]*b[x];
        forn(i,m)if(i!=y)c[i]-=c[y]*A[x][i];
        c[y]=-c[y]*A[x][y];
    }
    pair<double,vector<double> > simplex( // maximize c^T x s.t. Ax<=b, x>=0
            vector<vector<double> > _A, vector<double> _b, vector<double> _c){
        // returns pair (maximum value, solution vector)
        A=_A;b=_b;c=_c;
        n=si(b);m=si(c);z=0.;
        X=vi(m);Y=vi(n);
        forn(i,m)X[i]=i;
        forn(i,n)Y[i]=i+m;
        while(1){
            int x=-1,y=-1;
            double mn=-EPS;
            forn(i,n)if(b[i]<mn)mn=b[i],x=i;
            if(x<0)break;
            forn(i,m)if(A[x][i]<-EPS){y=i;break;}
            assert(y>=0); // no solution to Ax<=b
            pivot(x,y);
        }
        while(1){
            int x=-1,y=-1;
            double mx=EPS;
            forn(i,m)if(c[i]>mx)mx=c[i],y=i;
            if(y<0)break;
            double mn=1e200;
            forn(i,n)if(A[i][y]>EPS&&b[i]/A[i][y]<mn)mn=b[i]/A[i][y],x=i;
            assert(x>=0); // c^T x is unbounded
            pivot(x,y);
        }
        vector<double> r(m);
        forn(i,n)if(Y[i]<m)r[Y[i]]=b[i];
        return mp(z,r);
    }
};
```

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

```
vector<ii> G[MAX_N];//ady. list with pairs (weight, dst)
int dist[MAX_N];
```

```cpp
void bford(int src){//O(VE)
  dist[src]=0;
  forn(i, N-1) forn(j, N) if(dist[j]!=INF) for(auto u: G[j])
    dist[u.second]=min(dist[u.second], dist[j]+u.first);
}

bool hasNegCycle(){
  forn(j, N) if(dist[j]!=INF) for(auto u: G[j])
    if(dist[u.second]>dist[j]+u.first) return true;
  //inside if: all points reachable from u.snd will have -INF distance(
      do bfs)
  return false;
}
```

6.3. Kruskal

```cpp
struct Edge {
    int u, v, c;
    bool operator<(const Edge &o) const { return c < o.c; }
};
struct Kruskal {
    vector<Edge> edges;
    void addEdge(int u, int v, int c) {
        edges.pb(Edge{u, v, c});
    }
    ll build(int n) {
        sort(all(edges));
        ll cost = 0;
        UF uf(n);
        for (Edge &e : edges)
            if (uf.join(e.u, e.v)) cost += e.c;
        return cost;
    }
};
```

6.4. 2-SAT + Tarjan SCC

```cpp
// We have one node for each boolean variable and other for its negation
// Every edge represents an implication, to add a clause (a or b), use
    add_or(a, b)
// val[comp[i]] = value of variable i
struct SAT {
    vector<vi> g;
    stack<int> q;
    vector<bool> val;
    vi low, idx, comp;
    int n, id, comps, x;

    SAT(int vars) {
        n = vars, g.resize(2*n), id = 0, comps = 0;
        low = vi(2*n), idx = vi(2*n, -1), comp = vi(2*n, -1);
    }

    int neg(int u) { return u >= n ? u-n : u+n; }
    void add_or(int a, int b) { g[neg(a)].pb(b), g[neg(b)].pb(a); }

    void tarjan(int u) {
        low[u] = idx[u] = id++;
        q.push(u), comp[u] = -2;
        for (int v : g[u]) {
            if (idx[v] == -1 || comp[v] == -2) {
                if (idx[v] == -1) tarjan(v);
                low[u] = min(low[u], low[v]);
            }
        }
        if (low[u] == idx[u]) {
            do { x = q.top(), q.pop(), comp[x] = comps; } while (x != u)
                ;
            val.pb(comp[neg(u)] < 0), comps++;
        }
    }

    bool satisfiable() {
        forn(i, 2*n) if (idx[i] == -1) tarjan(i);
        forn(i, n) if (comp[i] == comp[neg(i)]) return false;
        return true;
    }
};
```

6.5. Articulation Points

```cpp
int N;
vector<int> G[1000000];
//V[i]=node number(if visited), L[i]= lowest V[i] reachable from i
int qV, V[1000000], L[1000000], P[1000000];
void dfs(int v, int f){
  L[v]=V[v]=++qV;
```

```
  for(auto u: G[v])
    if(!V[u]){
      dfs(u, v);
      L[v] = min(L[v], L[u]);
      P[v]+= L[u]>=V[v];
    }
    else if(u!=f)
      L[v]=min(L[v], V[u]);
}
int cantart(){ //O(n)
  qV=0;
  zero(V), zero(P);
  dfs(1, 0); P[1]--;
  int q=0;
  forn(i, N) if(P[i]) q++;
return q;
}
```

6.6. Comp. Biconexas y Puentes

```
struct bridge {
  struct edge {
    int u,v,comp;
    bool bridge;
  };

  int n,t,nbc;
  vi d,b,comp;
  stack<int> st;
    vector<vi> adj;
  vector<edge> e;

  bridge(int n=0): n(n) {
    adj = vector<vi>(n);
    e.clear();
    initDfs();
  }

  void initDfs() {
        d = vi(n), b = vi(n), comp = vi(n);
        forn(i,n) d[i] = -1;
        nbc = t = 0;
  }

  void addEdge(int u, int v) {
    adj[u].pb(si(e)); adj[v].pb(si(e));
    e.pb((edge){u,v,-1,false});
  }

    //d[i]=id de la dfs
    //b[i]=lowest id reachable from i
  void dfs(int u=0, int pe=-1) {
    b[u] = d[u] = t++;
        comp[u] = pe != -1;

    for(int ne : adj[u]) {
      if(ne == pe) continue;
      int v = e[ne].u ^ e[ne].v ^ u;
      if(d[v] == -1) {
        st.push(ne);
        dfs(v,ne);
        if(b[v] > d[u]) e[ne].bridge = true; // bridge
        if(b[v] >= d[u]) { // art
          int last;
          do {
            last = st.top(); st.pop();
            e[last].comp = nbc;
          } while(last != ne);
          nbc++, comp[u]++;
        }
        b[u] = min(b[u], b[v]);
      }
      else if(d[v] < d[u]) { // back edge
        st.push(ne);
        b[u] = min(b[u], d[v]);
      }
    }
  }
};
```

6.7. LCA + Climb

```
#define lg(x) (31-__builtin_clz(x))
struct LCA {
    vector<vi> a; vi lvl; // a[i][k] is the 2^k ancestor of i
    void dfs(int u=0, int p=-1, int l=0) {
```

```
        a[u][0] = p, lvl[u] = l;
        for (int v : g[u]) if (v != p) dfs(v, u, l+1);
    }
    LCA(int n) : a(n, vi(lg(n)+1)), lvl(n) {
        dfs(); forn(k, lg(n)) forn(i, n) a[i][k+1] = a[i][k] == -1 ? -1
            : a[a[i][k]][k];
    }
    int climb(int x, int d) {
        for (int i = lg(lvl[x]); d && i >= 0; i--)
            if ((1 << i) <= d) x = a[x][i], d -= 1 << i;
        return x;
    }
    int lca(int x, int y) { // O(lg n)
        if (lvl[x] < lvl[y]) swap(x, y);
        if (lvl[x] != lvl[y]) x = climb(x, lvl[x] - lvl[y]);
        if (x != y) {
            for (int i = lg(lvl[x]); i >= 0; i--)
                if (a[x][i] != a[y][i]) x = a[x][i], y = a[y][i];
            x = a[x][0];
        }
        return x;
    }
    int dist(int x, int y) { return lvl[x] + lvl[y] - 2 * lvl[lca(x, y)
        ]; }
};
```

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 $dist[v] >= 0$ (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).

```
struct Dinic {
    struct Edge { int v, r; ll c, f=0; };
    vector<vector<Edge>> g; vi dist, ptr;
    static const ll INF = 1e18;
    int n, s, t;
    Dinic(int _n, int _s, int _t) {
        n = _n, s = _s, t = _t;
        g.resize(n), dist = vi(n), ptr = vi(n);
    }
    void addEdge(int u, int v, ll c1, ll c2=0) {
        g[u].pb((Edge){v, si(g[v]), c1});
        g[v].pb((Edge){u, si(g[u])-1, c2});
    }
    bool bfs() {
        fill(all(dist), -1), dist[s] = 0;
        queue<int> q({s});
        while (si(q)) {
            int u = q.front(); q.pop();
            for (auto &e : g[u])
                if (dist[e.v] == -1 && e.f < e.c)
                    dist[e.v] = dist[u] + 1, q.push(e.v);
        }
        return dist[t] != -1;
    }
    ll dfs(int u, ll cap = INF) {
        if (u == t) return cap;
        for (int &i = ptr[u]; i < si(g[u]); ++i) {
            auto &e = g[u][i];
            if (e.f < e.c && dist[e.v] == dist[u] + 1) {
```

```
                ll flow = dfs(e.v, min(cap, e.c - e.f));
                if (flow) {
                    e.f += flow, g[e.v][e.r].f -= flow;
                    return flow;
                }
            }
        }
        return 0;
    }
    ll maxflow() {
        ll res = 0;
        while (bfs()) {
            fill(all(ptr), 0);
            while (ll flow = dfs(s)) res += flow;
        }
        return res;
    }
    void reset() { for (auto &v : g) for (auto &e : v) e.f = 0; }
};
```

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

```
struct MCF {
    const ll INF = 1e18;
    int n; vector<vi> adj;
    vector<vll> cap, cost;

    MCF(int _n) : n(_n) {
        adj.assign(n, vi());
        cap.assign(n, vll(n));
        cost.assign(n, vll(n));
    }

    void addEdge(int u, int v, ll _cap, ll _cost) {
        cap[u][v] = _cap;
        adj[u].pb(v), adj[v].pb(u);
        cost[u][v] = _cost, cost[v][u] = -_cost;
    }

    void shortest_paths(int s, vll &dist, vi &par) {
        par.assign(n, -1);
        vector<bool> inq(n);
        queue<int> q; q.push(s);
        dist.assign(n, INF), dist[s] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            inq[u] = false;
            for (int v : adj[u]) {
                if (cap[u][v] > 0 && dist[v] > dist[u] + cost[u][v]) {
                    dist[v] = dist[u] + cost[u][v], par[v] = u;
                    if (!inq[v]) inq[v] = true, q.push(v);
                }
            }
        }
    }

    ll min_cost_flow(ll k, int s, int t) {
        vll dist; vi par;
        ll flow = 0, total = 0;
        while (flow < k) {
            shortest_paths(s, dist, par);
            if (dist[t] == INF) break;
            // find max flow on that path
            ll f = k - flow;
            int cur = t;
            while (cur != s) {
                int p = par[cur];
                f = min(f, cap[p][cur]);
                cur = p;
            }
            // apply flow
            flow += f, total += f * dist[t], cur = t;
            while (cur != s) {
                int p = par[cur];
                cap[p][cur] -= f;
                cap[cur][p] += f;
                cur = p;
            }
        }
        return flow < k ? -1 : total;
    }
```

```
};
```

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

```cpp
#include <bits/stdc++.h>
using namespace std;

#ifdef LOCAL
    #define D(a) cerr << #a << " = " << a << endl
#else
    #define D(a) 8
#endif
#define fastio ios_base::sync_with_stdio(0); cin.tie(0)
#define dforsn(i,s,n) for(int i=int(n-1);i>=int(s);i--)
#define forsn(i,s,n) for(int i=int(s);i<int(n);i++)
#define all(a) (a).begin(),(a).end()
#define dforn(i,n) dforsn(i,0,n)
#define forn(i,n) forsn(i,0,n)
#define si(a) int((a).size())
#define pb emplace_back
#define mp make_pair
#define snd second
#define fst first
#define endl '\n'
using pii = pair<int,int>;
using vi = vector<int>;
using ll = long long;

int main() {
    fastio;

    return 0;
}
```

9. vimrc

```
colo desert
se nu
se nornu
se acd
se ic
se sc
se si
se cin
se ts=4
se sw=4
se sts=4
se et
se spr
se cb=unnamedplus
se nobk
se nowb
se noswf
se cc=80
map j gj
map k gk
aug cpp
    au!
    au FileType cpp map <f9> :w<CR> :!g++ -Wno-unused-result -
        D_GLIBCXX_DEBUG -Wconversion -Wshadow -Wall -Wextra -O2 -DLOCAL
        -std=c++17 -g3 "%" -o "%:p:r" <CR>
    au FileType cpp map <f5> :!"%:p:r" < a.in <CR>
    au FileType cpp map <f6> :!"%:p:r" <CR>
aug END
nm <c-h> <c-w><c-h>
nm <c-j> <c-w><c-j>
nm <c-k> <c-w><c-k>
```

```
nm <c-l> <c-w><c-l>
vm > >gv
vm < <gv
nn <silent> [b :bp<CR>
nn <silent> ]b :bn<CR>
nn <silent> [B :bf<CR>
nn <silent> ]B :bl<CR>
```

10. Misc

```
#pragma GCC optimize ("O3")//("avx,avx2,fma")

Random numbers:
mt19937_64 rng(time(0)); //if TLE use 32 bits: mt19937
ll rnd(ll a, ll b) { return a + rng()%(b-a+1); }
getline(cin,str);
// Make an extra call if we previously read another thing from the input
     stream
cout << fixed << setprecision(n);
cout << setw(n) << setfill('0');

// #include <sys/resource.h>
struct rlimit rl;
getrlimit(RLIMIT_STACK, &rl);
rl.rlim_cur = rl.rlim_max;
setrlimit(RLIMIT_STACK, &rl);

C++11:
to_string(num) // returns a string with the representation of num
stoi,stoll,stod,stold // string to int,ll,double & long double
    respectively
```