Índice

1. Referencia

Algorítmo	**Parámetros**	**Función**
sort, stable_sort	f, l	ordena el intervalo
nth_element	f, nth, l	*void* ordena el n-esimo, y particiona el resto
fill, fill_n	f, l / n, elem	*void* llena [f, l) o [f, f+n) con elem
lower_bound, upper_bound	f, l, elem	*it* al primer / ultimo donde se puede insertar elem para que quede ordenada
binary_search	f, l, elem	*bool* esta elem en [f, l)
copy	f, l, resul	hace resul+i=f+i $\forall i$
find, find_if, find_first_of	f, l, elem / pred / f2, l2	*it* encuentra i ∈[f,l) tq. i=elem, pred(i), i∈[f2,l2)
count, count_if	f, l, elem/pred	cuenta elem, pred(i)
search	f, l, f2, l2	busca [f2,l2) ∈ [f,l)
replace, replace_if	f, l, old / pred, new	cambia old / pred(i) por new
reverse	f, l	da vuelta
partition, stable_partition	f, l, pred	pred(i) ad, !pred(i) atras
min_element, max_element	f, l, [comp]	*it* min, max de [f,l]
lexicographical_compare	f1,l1,f2,l2	*bool* con [f1,l1]¡[f2,l2]
next/prev_permutation	f,l	deja en [f,l) la perm sig, ant
set_intersection, set_difference, set_union, set_symmetric_difference,	f1, l1, f2, l2, res	[res, ...) la op. de conj
push_heap, pop_heap, make_heap	f, l, e / e /	mete/saca e en heap [f,l), hace un heap de [f,l)
is_heap	f,l	*bool* es [f,l) un heap
accumulate	f,l,i,[op]	$T = \sum$/oper de [f,l)
inner_product	f1, l1, f2, i	T = i + [f1, l1) . [f2, ...)
partial_sum	f, l, r, [op]	r+i = $\sum$/oper de [f,f+i] $\forall i$ ∈[f,l)
__builtin_ffs	unsigned int	Pos. del primer 1 desde la derecha
__builtin_clz	unsigned int	Cant. de ceros desde la izquierda.
__builtin_ctz	unsigned int	Cant. de ceros desde la derecha.
__builtin_popcount	unsigned int	Cant. de 1's en x.
__builtin_parity	unsigned int	1 si x es par, 0 si es impar.
__builtin_XXXXXXll	unsigned ll	= pero para long long's.

2. Estructuras

2.1. Sparse Table

```
#define lg(n) (31 - __builtin_clz(n))
template<class T>
struct RMQ {
    int n; vector<vector<T>> t;
    RMQ(int sz) {
        n = sz, t.assign(lg(n)+1, vector<T>(n));
    }
    T& operator[](int p) { return t[0][p]; }
    T get(int i, int j) { // O(1), [i, j)
        int p = lg(j-i);
        return max(t[p][i], t[p][j - (1 << p)]);
    }
    void build() { // O(n lg n)
        forn(p, lg(n)) forn(x, n - (1 << p))
            t[p + 1][x] = max(t[p][x], t[p][x + (1 << p)]);
    }
};
```

2.2. Segment Tree

```
struct rmax { // op = max, neutro = -INF
    int x; rmax(int _x=-INF) { x = _x; }
    rmax operator+(const rmax &o) { return x > o.x ? *this : o; }
    bool operator!=(const rmax &o) { return x != o.x; }
};
template<class T>
struct RMQ { // ops O(lg n), [0, n)
    vector<T> t; int n;
  T& operator[](int p) { return t[p+n]; }
  RMQ(int sz) { n = 1 << (32-__builtin_clz(sz)), t.resize(2*n); }
    void build() { dforn(i, n) t[i] = t[2*i] + t[2*i+1]; }
  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();
    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 set(int p, T v) {
```

```
    for (p += n; p && t[p] != v;)
      t[p] = v, p /= 2, v = t[p*2] + t[p*2+1];
  }
};
// Use: RMQ<rmax> rmq(n); forn(i, n) cin >> rmq[i].x; rmq.build();
```

2.3. Segment Tree (Iterative)

```
struct rmax { // op = max, neutro = -INF
    int x; rmax(int _x=-INF) { x = _x; }
    rmax operator+(const rmax &o) { return x > o.x ? *this : o; }
};
template<class T>
struct RMQ { // ops O(lg n), [0, n)
    vector<T> t; int n;
    T& operator[](int p) { return t[p + n]; }
    RMQ(int sz) { n = sz, t.resize(2*n); }
    void build() { dforsn(i, 1, n) t[i] = t[i<<1] + t[i<<1|1]; }
    void set(int p, T v){
        for (t[p += n] = v; p >>= 1;) t[p] = t[p<<1] + t[p<<1|1];
    }
    T get(int l, int r) {
        T a, b;
        for (l+=n, r+=n; l < r; l>>=1, r>>=1){
            if (l & 1) a = a + t[l++];
            if (r & 1) b = t[--r] + b;
        }
        return a + b;
    }
};
// Use: RMQ<rmax> rmq(n); forn(i, n) { int x; cin >> x; rmq[i] = x; }
    rmq.build();
```

2.4. 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 * sz; } // 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 << (32-__builtin_clz(sz));
      t.resize(2*n), d.resize(2*n);
    }
    void build() { dforn(i, 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 (x < n) 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
    ();
```

2.5. Segment Tree (Persistent)

```
typedef int tipo;
tipo oper(const tipo &a, const tipo &b){
    return a + b;
}
struct node {
  tipo v; node *l, *r;
  node(tipo v):v(v), l(NULL), r(NULL) {}
  node(node *l, node *r) : l(l), r(r){
    if(!l) v = r->v;
    else if(!r) v = l->v;
    else v = oper(l->v, r->v);
  }
};
node *build (tipo *a, int tl, int tr) { // modificar para tomar tipo a
  if(tl + 1 == tr) return new node(a[tl]);
  int tm = (tl + tr) >> 1;
  return new node(build(a, tl, tm), build(a, tm, tr));
}
node *upd(int pos, int new_val, node *t, int tl, int tr){
  if(tl + 1 == tr) return new node(new_val);
  int tm = (tl + tr) >> 1;
  if(pos < tm) return new node(upd(pos, new_val, t->l, tl, tm), t->r);
  else return new node(t->l, upd(pos, new_val, t->r, tm, tr));
}
tipo get(int l, int r, node *t, int tl, int tr){
  if(l == tl && tr == r) return t->v;
  int tm = (tl + tr) >> 1;
  if(r <= tm) return get(l, r, t->l, tl, tm);
  else if(l >= tm) return get(l, r, t->r, tm, tr);
  return oper(get(l, tm, t->l, tl, tm), get(tm, r, t->r, tm, tr));
}
```

2.6. Sliding Window RMQ

```
// Para max pasar less y -INF
template <class T, class Compare, T INF>
struct RMQ {
    deque<T> d; queue<T> q;
    void push(T v) {
        while (!d.empty() && Compare()(d.back(), v)) d.pop_back();
        d.pb(v), q.push(v);
    }
    void pop() {
        if (!d.empty() && d.front()==q.front()) d.pop_front();
        q.pop();
    }
    T getMax() { return d.empty() ? INF : d.front(); }
    int size() { return si(q); }
};
RMQ<ll, less<ll>, -INF> rmq;
```

2.7. Fenwick Tree

```
// Para 2D: tratar cada columna como un Fenwick Tree,
// agregando un for anidado en cada operacion.
// Trucazo para 2D: si los elementos no se repiten,
// se puede usar un ordered set para memoria O(n*log^2(n))
typedef ll tipo;
struct Fenwick {
    static const int sz = (1 << 18) + 1;
    tipo t[sz];
    void adjust(int p, tipo v){ // p en [1, sz), O(lg n)
        for(int i = p; i < sz; i += (i & -i)) t[i] += v;
    }
    tipo sum(int p){ // Suma acumulada en [1, p], O(lg n)
        tipo s = 0;
        for(int i = p; i; i -= (i & -i)) s += t[i];
        return s;
    }
    tipo sum(int a, int b){ return sum(b) - sum(a - 1); }
    int lower_bound(tipo v) { // Menor x con suma acumulada >= v, O(lg n)
        int x = 0, d = sz-1;
        if(v > t[d]) return sz;
        for(; d; d >>= 1)
            if(t[x|d] < v) v -= t[x |= d];
        return x+1;
    }
};
```

2.8. Fenwick Tree (Ranges)

```
// Point update, range query:
template<class T>
struct BIT { // ops O(lg n), [0, n)
```

```
    int n, h; vector<T> d;
    BIT(int sz) { n = sz, d.resize(n+1), h = 1 << int(log2(n)); }
    void add(int i, T x) { for (++i; i <= n; i += i&-i) d[i] += x; }
    T sum(int i) { T r = 0; for (; i; i -= i&-i) r += d[i]; return r; }
    T sum(int l, int r) { return sum(r) - sum(l); }
    int lower_bound(T v) {
        int x = 0;
        for (int p = h; p; p >>= 1)
            if ((x|p) <= n && d[x|p] < v) v -= d[x |= p];
        return x;
    }
};

// Range update, point query:
template<class T>
struct BIT { // ops O(lg n), [0, n)
    vector<T> d; int n; BIT(int sz) { n=sz, d.resize(n+1); }
    void add(int l, int r, T x) { _add(l, x), _add(r, -x); }
    void _add(int i, T x) { for (++i; i <= n; i += i&-i) d[i] += x; }
    T sum(int i) { T r = 0; for (++i; i; i -= i&-i) r += d[i]; return r;
     }
};

// Range update, range query:
template<class T>
struct BIT { // ops O(lg n), [0, n)
    int n; vector<T> m, a;
    BIT(int sz) { n = sz, m.resize(n+1), a.resize(n+1); }
    void add(int l, int r, T x) {
        _add(l, x, -x*l), _add(r-1, -x, x*r);
    }
    void _add(int i, T x, T y) {
        for (++i; i <= n; i += i&-i) m[i] += x, a[i] += y;
    }
    T sum(int i) {
        T x = 0, y = 0, s = i;
        for (; i; i -= i&-i) x += m[i], y += a[i];
        return x*s + y;
    }
    T sum(int l, int r) { return sum(r) - sum(l); }
};
```

2.9. Union Find

```
struct UF {
    vi par, sz;
    UF(int n): par(n), sz(n, 1) { iota(all(par), 0); }
    int find(int u) { return par[u] == u ? u : par[u] = find(par[u]); }
    bool connected(int u, int v) { return find(u) == find(v); }
    bool join(int u, int v) {
        if (connected(u, v)) return false;
        u = find(u), v = find(v);
        if (sz[u] < sz[v]) par[u] = v, sz[v] += sz[u];
        else par[v] = u, sz[u] += sz[v];
        return true;
    }
};
```

2.10. Disjoint Intervals

```
// Guarda intervalos como [first, second]
// En caso de colision, los une en un solo intervalo
bool operator <(const pii &a, const pii &b){ return a.first < b.first; }
struct disjoint_intervals {
  set<pii> segs;
  void insert(pii v){ // O(lg n)
    if(v.second - v.first == 0.0) return; // Cuidado!
    set<pii>::iterator it, at;
    at = it = segs.lower_bound(v);
    if(at != segs.begin() && (--at)->second >= v.first){
      v.first = at->first;
      --it;
    }
    for(; it!=segs.end() && it->first <= v.second; segs.erase(it++))
      v.second = max(v.second, it->second);
    segs.insert(v);
  }
};
```

2.11. Segment Tree (2D)

```
struct RMQ2D { // n filas, m columnas
  int sz;
  RMQ t[4*MAXN]; // t[i][j] = i fila, j columna
  RMQ &operator [](int p){ return t[sz/2 + p]; }
  void init(int n, int m){ // O(n*m)
    sz = 1 << (32 - __builtin_clz(n));
    forn(i, 2*sz) t[i].init(m);
```

```
  }
  void set(int i, int j, tipo val){ // O(lg(m)*lg(n))
    for(i += sz; i > 0;){
      t[i].set(j, val);
      i /= 2;
      val = operacion(t[i*2][j], t[i*2 + 1][j]);
    }
  }
  tipo get(int i1, int j1, int i2, int j2){
    return get(i1, j1, i2, j2, 1, 0, sz);
  }
  // O(lg(m)*lg(n)), rangos cerrado abierto
  int get(int i1, int j1, int i2, int j2, int n, int a, int b){
    if(i2 <= a || i1 >= b) return 0;
    if(i1 <= a && b <= i2) return t[n].get(j1, j2);
    int c = (a + b)/2;
    return operacion(get(i1, j1, i2, j2, 2*n, a, c),
                     get(i1, j1, i2, j2, 2*n + 1, c, b));
  }
} rmq;
// Ejemplo para inicializar una matriz de n filas por m columnas
RMQ2D rmq; rmq.init(n, m);
forn(i, n) forn(j, m){
  int v; cin >> v; rmq.set(i, j, v);
}
```

2.12. Big Int

```
#define BASE 10
#define LMAX 1000
int pad(int x){
    x--; int c = 0;
    while(x) x /= 10, c++;
    return c;
}
const int PAD = pad(BASE);
struct bint {
    int l;
    ll n[LMAX];
    bint(ll x = 0){
        l = 1;
        forn(i,LMAX){
          if(x) l = i+1;
          n[i] = x % BASE;
          x /= BASE;
        }
    }
    bint(string x){
        int sz = si(x);
        l = (sz-1)/PAD + 1;
        fill(n, n+LMAX, 0);
        ll r = 1;
        forn(i,sz){
            if(i % PAD == 0) r = 1;
            n[i/PAD] += r*(x[sz-1-i]-'0');
            r *= 10;
        }
    }
    void out() const {
        cout << n[l-1] << setfill('0');
        dforn(i,l-1) cout << setw(PAD) << n[i];
    }
    void invar(){
        fill(n+l, n+LMAX, 0);
        while(l > 1 && !n[l-1]) l--;
    }
};
bint operator+(const bint &a, const bint &b){
    bint c;
    c.l = max(a.l, b.l);
    ll q = 0;
    forn(i,c.l){
        q += a.n[i] + b.n[i];
        c.n[i] = q % BASE;
        q /= BASE;
    }
    if(q) c.n[c.l++] = q;
    c.invar();
    return c;
}
pair<bint,bool> lresta(const bint &a, const bint &b){ // c = a - b
    bint c;
    c.l = max(a.l, b.l);
    ll q = 0;
    forn(i,c.l){
        q += a.n[i] - b.n[i];
```

```
        c.n[i] = (q + BASE) % BASE;
        q = (q + BASE)/BASE - 1;
    }
    c.invar();
    return {c,!q};
}
bint &operator -=(bint &a, const bint &b){ return a = lresta(a, b).fst;
    }
bint operator -(const bint &a, const bint &b){ return lresta(a, b).fst;
    }
bool operator <(const bint &a, const bint &b){ return !lresta(a, b).snd;
     }
bool operator <=(const bint &a, const bint &b){ return lresta(b, a).snd;
     }
bool operator ==(const bint &a, const bint &b){ return a <= b && b <= a;
     }
bool operator !=(const bint &a, const bint &b){ return a < b || b < a; }
bint operator *(const bint &a, ll b){
    bint c;
    ll q = 0;
    forn(i,a.l){
        q += a.n[i]*b;
        c.n[i] = q % BASE;
        q /= BASE;
    }
    c.l = a.l;
    while(q){
        c.n[c.l++] = q % BASE;
        q /= BASE;
    }
    c.invar();
    return c;
}
bint operator *(const bint &a, const bint &b){
    bint c;
    c.l = a.l+b.l;
    fill(c.n, c.n+b.l, 0);
    forn(i,a.l){
        ll q = 0;
        forn(j,b.l){
            q += a.n[i]*b.n[j] + c.n[i+j];
            c.n[i + j] = q % BASE;
            q /= BASE;
        }
        c.n[i+b.l] = q;
    }
    c.invar();
    return c;
}
pair<bint,ll> ldiv(const bint &a, ll b){ // c = a / b ; rm = a % b
  bint c;
  ll rm = 0;
  dforn(i,a.l){
        rm = rm*BASE + a.n[i];
        c.n[i] = rm/b;
        rm %= b;
    }
    c.l = a.l;
    c.invar();
    return {c,rm};
}
bint operator /(const bint &a, ll b){ return ldiv(a, b).fst; }
ll operator %(const bint &a, ll b){ return ldiv(a, b).snd; }
pair<bint,bint> ldiv(const bint &a, const bint &b){
    bint c, rm = 0;
    dforn(i,a.l){
        if(rm.l == 1 && !rm.n[0]) rm.n[0] = a.n[i];
        else {
            dforn(j,rm.l) rm.n[j+1] = rm.n[j];
            rm.n[0] = a.n[i], rm.l++;
        }
        ll q = rm.n[b.l]*BASE + rm.n[b.l-1];
        ll u = q / (b.n[b.l-1] + 1);
        ll v = q / b.n[b.l-1] + 1;
        while(u < v-1){
            ll m = (u + v)/2;
            if(b*m <= rm) u = m;
            else v = m;
        }
        c.n[i] = u, rm -= b*u;
    }
    c.l = a.l;
    c.invar();
    return {c,rm};
}
bint operator /(const bint &a, const bint &b){ return ldiv(a, b).fst; }
```

```
bint operator %(const bint &a, const bint &b){ return ldiv(a, b).snd; }
bint gcd(bint a, bint b){
    while(b != bint(0)){
        bint r = a % b;
        a = b, b = r;
    }
    return a;
}
```

2.13. Modnum

```
struct num {
    int a;
    num(int _a = 0) : a(_a) {} // o tambien num(ll _a=0) : a((_a%M+M)%M)
        {}
    operator int(){ return a; }
    num operator +(num b){ return a+b.a >= M ? a+b.a-M : a+b.a; }
    num operator -(num b){ return a-b.a < 0 ? a-b.a+M : a-b.a; }
    num operator *(num b){ return int((ll)a*b.a % M); }
    num operator ^(ll e){
    if(!e) return 1;
        num q = (*this)^(e/2);
    return e & 1 ? q*q*(*this) : q*q;
    }
    num operator ++(int x){ return a++; }
};
int norm(ll x){ return x < 0 ? int(x % M + M) : int(x % M); }
num inv(num x){ return x^(M-2); } // M must be prime
num operator /(num a, num b){ return a*inv(b); }
num neg(num x){ return x.a ? -x.a+M : 0; }
istream& operator >>(istream &i, num &x){ i >> x.a; return i; }
ostream& operator <<(ostream &o, const num &x){ o << x.a; return o; }
// Cast integral values to num in arithmetic expressions!
```

2.14. Treap

Definición: estructura de datos que combina los conceptos de *binary search tree* (para las claves) y *heap* (para las prioridades), y asigna las prioridades de forma aleatoria para asegurar una altura de $O(\log n)$ en promedio.

Operaciones básicas:

- $split(T, X)$: separa al árbol T en 2 subárboles T_L y T_R tales que T_L contiene a todos los elementos con claves menores a X y T_R a los demás.
- $merge(T_1, T_2)$: combina dos subárboles T_1 y T_2 y retorna un nuevo árbol, asume que las claves en T_1 son menores que las claves en T_2.

Operaciones avanzadas:

- $insert(T, X)$: inserta una nueva clave al árbol. Resulta trivial de implementar a partir de las anteriores: $(T_1, T_2) = split(T, X)$ y $T_3 = merge(merge(T_1, X), T_2)$.

2.14.1. Treap set

```
typedef int Key;
typedef struct node *pnode;
struct node {
    Key key;
    int prior, size;
    pnode l, r;
    node(Key key = 0): key(key), prior(rand()), size(1), l(0), r(0) {}
        // usar rand piola
};
static int size(pnode p){ return p ? p->size : 0; }
void push(pnode p){
    // modificar y propagar el dirty a los hijos aca (para lazy)
}
// 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);
}
//junta dos sets
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 set en dos, l < key <= r
void split(pnode t, Key key, pnode &l, pnode &r){
    if(!t) return void(l = r = 0);
    push(t);

    if(key <= t->key) split(t->l, key, l, t->l), r = t;
```

```
    else split(t->r, key, t->r, r), l = t;

    pull(t);
}
//junta dos sets, sin asunciones
pnode unite(pnode l, pnode r){
    if(!l || !r) return l ? l : r;
    push(l), push(r);
    pnode t;

    if (l->prior > r->prior) swap(l, r);

    pnode rl, rr;
    split(r, l->key, rl, rr);
    l->l = unite(l->l, rl);
    l->r = unite(l->r, rr);

    pull(l);
    return l;
}

void erase(pnode &t, Key key){
    if(!t) return;
    push(t);

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

    if(t) pull(t);
}

pnode find(pnode t, Key key){
    if(!t) return 0;

    if(key == t->key) return t;
    if(key < t->key) return find(t->l, key);

    return find(t->r, key);
}

ostream& operator<<(ostream &out, const pnode &t){
    if(!t) return out;
    return out << t->l << t->key << ' ' << t->r;
}

struct treap {
    pnode root;
    treap(pnode root = 0): root(root) {}
    int size(){ return ::size(root); }
    void insert(Key key){
        pnode t1, t2; split(root, key, t1, t2);
        t1 = ::merge(t1, new node(key));
        root = ::merge(t1,t2);
    }
    void erase(Key key1, Key key2){
        pnode t1, t2, t3;
        split(root, key1, t1, t2);
        split(t2, key2, t2, t3);
        root = merge(t1, t3);
    }
    void erase(Key key){ ::erase(root, key); }
    pnode find(Key key){ return ::find(root, key); }
    Key &operator[](int pos){ return find(pos)->key; }//ojito
};
treap merge(treap a, treap b){ return treap(merge(a.root, b.root)); }
```

2.14.2. Treap array

Explicación treap implícito: permite insertar, borrar, hacer queries y updates (incluyendo reverse) en rangos en un arreglo. La idea es usar a los índices como claves, pero en vez de almacenarlos (sería difícil actualizar en ese caso), aprovechamos que la clave de un nodo es la cantidad de elementos menores a ese nodo (cuidado, no son solo los del subárbol izquierdo).

```
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);
}
```

2.15. Convex Hull Trick

```
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 el <= 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); }
    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;
    }
};
```

2.16. Convex Hull Trick (Dynamic)

```
using ld = long double;
const ld EPS = 1e-9, INF = 2e18;
#define less_equal(a,b) ((a) < (b) + EPS)
bool query, qmax; // qmax = 1: queries return max
struct line {
    ll a,b; ld i;
    line(ll _a, ll _b, ld _i) { a=_a, b=_b, i=_i; }
    bool operator<(const line &l) const {
        return query ? i < l.i : (qmax ? a < l.a : a > l.a);
    }
};
struct CHT {
    set<line> s;
    ld inter(const line &x, const line &y) {
        return ld(x.b - y.b) / (y.a - x.a);
    }
    bool improves(const line &x, const line &y, const line &z) {
        return less_equal(inter(x, z), inter(x, y));
    }
    void insert(ll a, ll b) { // ax + b = 0
        query = false;
        line v = {a,b,0};
        auto l = s.lower_bound(v);
        bool add = l == s.end() || (a == l->a && (qmax ? b > l->b : b <
            l->b)) ||
             (a != l->a && (l == s.begin() || !improves(*prev(l), v, *l)
                 ));
        if (!add) return;
        bool first = l == s.begin();
        auto f = l, p = l;
        if (l != s.end() && a == l->a) l++;
        if (!first) {
            p = prev(p);
            while (p != s.begin() && improves(*prev(p), *p, v)) f = p, p
                = prev(p);
        }
        if (l != s.end()) {
            auto n = next(l);
            while (n != s.end() && improves(v, *l, *n)) l = n, n = next(
                n);
        }
        if (f != l) s.erase(f, l);
        if (l != s.end()) {
            ld i = inter(v, *l);
            ll c = l->a, d = l->b;
            s.erase(l), s.emplace(c, d, i);
        }
        if (first) s.emplace(a, b, -INF);
        else s.emplace(a, b, inter(*p, v));
    }
    ll eval(ld x) {
        query = true;
        auto &l = *prev(s.upper_bound({0,0,x}));
        return l.a * x + l.b;
    }
} cht;
```

2.17. Gain-Cost Set

```
//esta estructura mantiene pairs(beneficio, costo)
//de tal manera que en el set quedan ordenados
//por beneficio Y COSTO creciente. (va borrando los que no son optimos)
struct V{
```

```
  int gain, cost;
  bool operator<(const V &b)const{return gain<b.gain;}
};
set<V> s;
void add(V x){
  set<V>::iterator p=s.lower_bound(x);//primer elemento mayor o igual
  if(p!=s.end() && p->cost <= x.cost) return;//ya hay uno mejor
  p=s.upper_bound(x);//primer elemento mayor
  if(p!=s.begin()){//borro todos los peores (<=beneficio  y >=costo)
    --p;//ahora es ultimo elemento menor o igual
    while(p->cost >= x.cost){
      if(p==s.begin()){s.erase(p); break;}
      s.erase(p--);
    }
  }
  s.insert(x);
}
int get(int gain){//minimo costo de obtener tal ganancia
  set<V>::iterator p=s.lower_bound((V){gain, 0});
  return p==s.end()? INF : p->cost;}
```

2.18. Set con índices

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

3. Algoritmos varios

3.1. Longest Increasing Subsecuence

```
int lis(const vi &a) { // O(n lg n)
    int n = si(a), INF = 2e9, r = 0;
    vi v(n+1,INF); v[0] = -INF;
    forn(i,n){
        int j = int(upper_bound(all(v), a[i]) - v.begin());
        if(v[j-1] < a[i] && a[i] < v[j]) v[j] = a[i], r = max(r,j);
    }
    return r;
}

---------------------------------------------------------------------

vi path;
int lis(const vi &a) { // O(n lg n)
    int n = si(a), INF = 2e9, r = 0;
    vi v(n+1,INF),id(n+1),p(n);
    v[0] = -INF;

    forn(i,n){
        int j = int(upper_bound(all(v), a[i]) - v.begin());
        if(v[j-1] < a[i] && a[i] < v[j])
            v[j] = a[i], r = max(r,j), id[j] = i, p[i] = id[j-1];
    }

    path = vi(r); int c = id[r];
    forn(i,r) path[r-i-1] = a[c], c = p[c];
    return r;
}
```

3.2. Alpha-Beta prunning

```
ll alphabeta(State &s, bool player = true, int depth = 1e9, ll alpha = -
    INF, ll beta = INF) { //player = true -> Maximiza
    if(s.isFinal()) return s.score;
  //~ if (!depth) return s.heuristic();
    vector<State> children;
    s.expand(player, children);
    int n = children.size();
    forn(i, n) {
        ll v = alphabeta(children[i], !player, depth-1, alpha, beta);
        if(!player) alpha = max(alpha, v);
        else beta = min(beta, v);
        if(beta <= alpha) break;
    }
    return !player ? alpha : beta;}
```

3.3. Mo's algorithm

```
const int Q = 2e5, SQ = 200;

struct Query { // [l, r)
```

```
    int l,r,id;
    bool operator<(const Query &q){
        if(l/SQ != q.l/SQ) return l < q.l;
        return l/SQ & 1 ? r < q.r : r > q.r;
    }
} qs[Q];

int ans[Q],res,pl,pr; // ans[i] = ans to ith query

void mo(int m){ // O( (n+q) * sqrt(n) * (add() + remove()) )
    forn(i,m) qs[i].id = i;
    sort(qs, qs + m);
    pl = 0, pr = 0, res = 0;
    forn(i,m){
        Query &q = qs[i];
        while(pl > q.l) add(--pl);
        while(pr < q.r) add(pr++);
        while(pl < q.l) remove(pl++);
        while(pr > q.r) remove(--pr);
        ans[q.id] = res;
    }
}
```

3.4. Parallel binary search

Descripción: permite reutilizar información cuando se necesitan realizar múltiples búsquedas binarias sobre la misma información.
Explicación algoritmo: imaginarse un árbol binario de rangos de búsqueda binaria $(lo, hi]$ y queries asignadas a cada nodo, que implican que esa query está en ese rango de la búsqueda binaria. El algoritmo aprovecha que para cada nivel del árbol las queries están ordenadas, y se puede procesar la información hasta el *mid* de cada query en orden, resultando en un tiempo de $O(N + Q_{nivel})$ por nivel (más un *log* extra por ordenar).
Observación: se puede implementar de forma recursiva, dependiendo del problema. Esto puede mejorar la complejidad ya que se evita el ordenamiento.

```
using QueryInRange = tuple<int, int, int>;

void init(); // reset values to start
void add(int k); // work that is common to multiple queries
bool can(int q); // usual check

vi ans; // resize to q
void binary_search(int start, int end, vi query_ids) {
    vector<QueryInRange> queries;
    for (int id : query_ids) queries.pb(start, end, id);

    while (!queries.empty()) {
        vector<QueryInRange> next_queries;

        int progress = 0;
        init();

        for (auto &query : queries) {
            int lo, hi, id; tie(lo, hi, id) = query;
            if (lo + 1 == hi) continue;

            int mid = (lo + hi) / 2;
            while (progress < mid) add(progress++);

            if (can(id)) ans[id] = mid, next_queries.pb(lo, mid, id);
            else next_queries.pb(mid, hi, id);
        }

        sort(all(next_queries));

        queries = next_queries;
    }
}
```

4. Strings

4.1. Hash

```
mt19937 rng;
struct basicHashing {
    int mod,mul; vi h,pot;

    bool prime(int n) {
        for (int d = 2; d*d <= n; d++) if (n%d == 0) return false;
        return true;
    }

    void randomize() {
        rng.seed(chrono::steady_clock::now().time_since_epoch().count())
            ;
```

```
        mod = uniform_int_distribution<>(0, (int) 5e8)(rng) + int(1e9);
        while (!prime(mod)) mod++;
        mul = uniform_int_distribution<>(2,mod-2)(rng);
    }
    basicHashing() { randomize(); }

    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(( ll(h[i])*mul + s[i]) % mod);
        pot[0] = 1; forn(i,n) pot[i+1] = int( ll(pot[i]) * mul % mod);
    }

    int hash(int i, int j) { // [ )
        int res = int(h[j] - ll(h[i])*pot[j-i] % mod);
        if (res < 0) res += mod;
        return res;
    }
    int hash(const string &s) {
        int res = 0;
        for (char c : s) res = int(( ll(res)*mul + 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)
            };
    }
};
```

4.2. 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;
    }
}
```

4.3. KMP

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

vi find_occurrences(string &s, string &t) { //apariciones de t en s
    vi pre = prefix_function(t), res;
```

```
    int n = si(s), m = si(t), j = 0;
    forn(i, n) {
        while (j > 0 && s[i] != t[j]) j = pre[j-1];
        if (s[i] == t[j]) j++;
        if (j == m) {
            res.pb(i-j+1);
            j = pre[j-1];
        }
    }
    return res;
}

// aut[i][c] = (next o failure function) al intentar matchear s[i] con c
void compute_automaton(string s, vector<vi>& aut) {
    s += '#'; // separador!
    int n = si(s);
    vi pi = prefix_function(s);
    aut.assign(n, vi(26));

    forn(i, n) forn(c, 26)
        if (i > 0 && 'a' + c != s[i])
            aut[i][c] = aut[pi[i-1]][c];
        else
            aut[i][c] = i + ('a' + c == s[i]);
}
```

4.4. Trie

```
struct trie {
    int p = 0, w = 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->p++;
        }
        x->w++;
    }
    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->w;
    }
    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->p--;
            else return;
            if(!y->p){
                x->c.erase(s[i]);
                return;
            }
            x = y;
        }
        x->w--;
    }
  void print(string tab = "") {
    for(auto &i : c) {
      cerr << tab << i.fst << endl;
      i.snd->print(tab + "--");
    }
  }
};
```

4.5. 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;
}
```

4.6. Suffix Array (largo, nlogn)

```
const int MAXN = 1e3+10;
#define rBOUND(x) (x<n? r[x] : 0)
//sa will hold the suffixes in order.
int sa[MAXN], r[MAXN], n;
string s; //input string, n=si(s)

int f[MAXN], tmpsa[MAXN];
void countingSort(int k){
    fill(f, f+MAXN, 0);
  forn(i, n) f[rBOUND(i+k)]++;
  int sum=0;
  forn(i, max(255, n)){
    int t=f[i]; f[i]=sum; sum+=t;}
  forn(i, n)
    tmpsa[f[rBOUND(sa[i]+k)]++]=sa[i];
  memcpy(sa, tmpsa, sizeof(sa));
}
void constructsa(){//O(n log n)
  n=si(s);
  forn(i, n) sa[i]=i, r[i]=s[i];
  for(int k=1; k<n; k<<=1){
    countingSort(k), countingSort(0);
    int rank, tmpr[MAXN];
    tmpr[sa[0]]=rank=0;
    forsn(i, 1, n)
      tmpr[sa[i]]=(r[sa[i]]==r[sa[i-1]] && r[sa[i]+k]==r[sa[i-1]+k] )?
          rank : ++rank;
    memcpy(r, tmpr, sizeof(r));
    if(r[sa[n-1]]==n-1) break;
  }
}
void print(){//for debug
  forn(i,n){
    cout << i << ' ';
    s.substr(sa[i], s.find( '$', sa[i])-sa[i]) << endl;
    }
}

//returns (lowerbound, upperbound) of the search
```

4.7. 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;
```

4.8. 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]];
```

4.9. Aho-Corasick

Definición El automáta Aho-Corasick es un autómata A que reconoce un conjunto de cadenas S.

Conceptos importantes

- Cada nodo del autómata se asocia con (al menos) un prefijo de una cadena en S.
- Un *suffix link* para un vértice p es un arco que apunta al sufijo propio más largo de la cadena correspondiente al vértice p.
- Estando en un estado p que corresponde a una palabra t, se pueden definir arcos de dos tipos:
  - Transiciones tipo trie: dado un caracter c tal que $t+c$ pertenece al autómata, el arco apunta a $t+c$.
  - Transiciones tipo suffix link: dado un caracter c tal que $t+c$ no pertenece al autómata, el arco apunta al máximo sufijo propio de $t+c$ que pertenece al árbol.
- Implementación:
  - Cada nodo mantiene:
    - Un indicador de la cantidad de cadenas que terminan en ese nodo: *terminal*.
    - El padre p y el caracter desde el que transicionó *pch*.
    - Las transiciones tipo trie en *next*.
    - El suffix link en *link*.
    - Todas las transiciones (tipo trie y tipo suffix link) en *go*.
  - El algoritmo se divide en:
    - *add_string*: agrega una cadena s al autómata.
    - *go*: calcula el nodo destino de la transición (v, ch).
    - *get_link*: calcula el suffix link de la cadena correspondiente al nodo v.

Problemas clásicos

- Encontrar todas las cadenas de un conjunto en un texto: mantener *exit link* (nodo terminal más cercano alcanzable mediante suffix links), recorrer autómata con el texto como entrada y transicionar por exit links para encontrar matches.
- Cadena lexicográficamente mínima de longitud *len* que no matchea ninguna cadena de un conjunto S: DFS sobre autómata para encontrar camino de longitud L evitando entrar en nodos terminales.
- Mínima cadena que contiene todas las cadenas de un conjunto S: BFS sobre autómata manteniendo máscara de cadenas matcheadas (y máscara de terminales, incluyendo alcanzables por suffix link, en cada nodo). **Recordatorio importante:** un nodo solo mantiene los matches para la cadena completa. Para mantener todos los matches (incluyendo sufijos) estando en un nodo v, hay que usar la información que propagan los suffix links.
- Cadena lexicogrficamente mínima de longitud *len* que contiene k cadenas de un conjunto S: DFS sobre grafo (v, len, cnt).

```
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);
    }
};

```

```cpp
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];
}
```

4.10. Suffix Automaton

Definición Un suffix automaton A es un autómata minimal que reconoce los sufijos de una cadena s.

Conceptos importantes

- A *reconoce* a una cadena s si comenzando desde el nodo inicial llegamos a un terminal.
- Dada una subcadena t de s, definimos $endpos(t)$ como el conjunto de las posiciones en s en las que terminan las apariciones de t.
- Dos subcadenas u y v de s son *equivalentes* si recorrer el autómata con u y con v nos lleva al mismo nodo. Esto es equivalente a $endpos(u) = endpos(v)$. Los nodos del automáta se corresponden al conjunto de cadenas de las *clases de equivalencia* bajo la relación anterior.
- Las cadenas en una clase de equivalencia son sufijos de la cadena de mayor tamaño de la clase, y forman un intervalo contiguo de tamaños. El *suffix link* nos lleva al primer sufijo que no pertenece a esta clase.
- *Suffix tree* implícito (de s'): el suffix link saliente de un nodo nos lleva al padre en el suffix tree de s' y los suffix links entrantes de un nodo provienen de los hijos del suffix tree de s'.

Algoritmo para construcción

- Agregamos un caracter a la vez. Sea c el caracter a agregar.
- Sea *last* el estado que corresponde a la cadena entera antes de agregar a c.
- Creamos un nuevo estado *cur*, que corresponde a la cadena luego de agregar a c.
- Agregamos transiciones a través de c a los sufijos de la cadena (recorriendo suffix links a partir de *last*), hasta encontrar un estado de un sufijo que ya tenga una transición con c.
    - Si no encontramos un estado, el suffix link de *cur* es t_0.
    - Si la transición lleva a un estado q que representa una cadena con un solo caracter más, el suffix link de *cur* es q.
    - Si no, es necesario dividir el estado q, ya que debemos usarlo como suffix link pero tiene sufijos extra. Después de esto hace falta actualizar los estados que tenían transiciones a q.

Problemas clásicos

- Determinar si w es subcadena de s: simplemente correr el autómata.

- Determinar si w es sufijo de s: correr el autómata y ver si caemos en un terminal.
- Contar cantidad de subcadenas distintas de s: esto es igual a la cantidad de caminos en el autómata y se calcula mediante una DP.
- Contar cantidad de apariciones de w en s: correr autómata con w. Llamemos u al nodo en el que terminamos, la cantidad de apariciones es la cantidad de caminos en A que comienzan en u y llegan a un terminal.
- Encontrar dónde aparece w por primera vez en s: correr autómata con w. Llamemos u al nodo en el que terminamos, esto equivale a calcular el camino más largo del autómata a partir del nodo u. Otra solución: mantener $firstpos(v)$, la primera aparición de una subcadena en la cadena (se actualiza cuando se crea un nuevo nodo y cuando se clonan nodos).
- Encontrar las posiciones de todas las apariciones de w en s: encontrar el nodo u que corresponde a w, armar el *suffix tree* (mantener los suffix links invertidos), encontrar todos los nodos en el subárbol con raíz en u, cada nodo corresponde a por lo menos una aparición y cada aparición corresponde a un nodo y su clon (utilizar $firstpos(v)$ para saber la posición, saltear nodos clonados; o bien agregar un \$ al comienzo de la cadena y encontrar todas las hojas, la posición es la longitud).
- Subcadena común más larga de un conjunto de cadenas: dadas k cadenas S_i, elegimos k separadores distintos entre sí D_i, formamos $T = S_1 + D_1 + \cdots + S_k + D_k$ y construimos el autómata de esa cadena. Saber si una subcadena pertenece a una cadena S_i en particular corresponde a verificar que existe un camino a D_i sin pasar por los demás separadores. Si calculamos para cada nodo a qué separadores puede llegar, la respuesta es la máxima de las cadenas más largas de las clases correspondientes a estados v que puede llegar a todos los separadores.

```
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;
}
```

4.11. Z Function

Definición La función Z para una string s de longitud n es un arreglo a de la misma longitud tal que $a[i]$ es la *máxima cantidad de caracteres* comenzando desde la posición i que coinciden con los primeros caracteres de s. Es decir, es el *máximo prefijo común*.

Observación $z[0]$ no está bien definido, pero se asume igual a 0.
Algoritmo La idea es mantener el máximo *match* (es decir, el segmento $[l, r]$ con máximo r tal que se sabe que $s[0..r-l] = s[l..r]$).
Siendo i el índice actual (del que queremos calcular la función Z), el algoritmo se divide en dos casos:

- $i > r$: la posición está fuera de lo que hemos procesado. Se corre el *algoritmo trivial*.
- $i <= r$: la posición está dentro del *match actual*, por lo que se puede utilizar como aproximación inicial $z[i] = min(r-i+1, z[i-l])$, y luego correr el *algoritmo trivial*.

Problemas clásicos

- Buscar una subcadena: concatenamos p con t (utilizando un separador). Hay una aparición si la función Z matcheó tantos caracteres como la longitud de p.

```
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;
    }
}
```

4.12. Palindrome

```
bool palindrome(ll x){
  string s = to_string(x); int n = si(s);
  forn(i,n/2) if(s[i] != s[n-i-1]) return 0;
  return 1;
}
```

5. Geometría

5.1. 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)
```

5.2. Point

```
const double EPS = 1e-9;
struct pto {
  double x, y;
  pto(double _x=0, double _y=0) : x(_x),y(_y) {}
  pto operator+(pto a) { return pto(x + a.x, y + a.y); }
  pto operator-(pto a) { return pto(x - a.x, y - a.y); }
  pto operator+(double a) { return pto(x + a, y + a); }
  pto operator*(double a) { return pto(x*a, y*a); }
  pto operator/(double a) { return pto(x/a, y/a); }
  double norm() { return sqrt(x*x + y*y); }
  double norm2() { return x*x + y*y; }
    // Dot product:
  double operator*(pto 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^(pto a) { return x*a.y - y*a.x; }
  // Returns true if this point is at the left side of line qr:
  bool left(pto q, pto r) { return ((q - *this) ^ (r - *this)) > 0; }
  bool operator<(const pto &a) const {
        return x < a.x - EPS || (abs(x - a.x) < EPS && y < a.y - EPS);
    }
    bool operator==(pto a) {
        return abs(x - a.x) < EPS && abs(y - a.y) < EPS;
    }
};
typedef pto vec;
double dist(pto a, pto b) { return (b-a).norm(); }
double dist2(pto a, pto b) { return (b-a).norm2(); }
double angle(pto a, pto o, pto b){ // [-pi, pi]
  pto oa = a-o, ob = b-o;
  return atan2(oa^ob, oa*ob);
}
// Rotate around the origin:
pto CCW90(pto p) { return pto(-p.y, p.x); }
pto CW90(pto p) { return pto(p.y, -p.x); }
pto CCW(pto p, double t){ // rads
  return pto(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(pto &p) { return p.y == 0 ? p.x < 0 : p.y > 0; }
bool angularOrder(pto &x, pto &y) {
  bool X = half(x), Y = half(y);
  return X == Y ? (x ^ y) > 0 : X < Y;
}
```

5.3. 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);
    }
};
```

5.4. Line

```
int sgn(ll x){return x<0? -1 : !!x;}
struct line{
  line() {}
  double a,b,c;//Ax+By=C
//pto MUST store float coordinates!
  line(double a, double b, double c):a(a),b(b),c(c){}
  line(pto p, pto q): a(q.y-p.y), b(p.x-q.x), c(a*p.x+b*p.y) {}
  int side(pto p){return sgn(ll(a) * p.x + ll(b) * p.y - c);}
};
bool parallels(line l1, line l2){return abs(l1.a*l2.b-l2.a*l1.b)<EPS;}
pto inter(line l1, line l2){//intersection
  double det=l1.a*l2.b-l2.a*l1.b;
  if(abs(det)<EPS) return pto(INF, INF);//parallels
  return pto(l2.b*l1.c-l1.b*l2.c, l1.a*l2.c-l2.a*l1.c)/det;
}
```

5.5. Segment

```
struct segm {
  pto s, f;
  segm(pto s, pto f) : s(s), f(f) {}
  pto closest(pto p) { // use for dist to point
     double l2 = dist2(s, f);
     if (l2 == 0.) return s;
     double t = ((p-s) * (f-s)) / l2;
     if (t < 0.) return s; // don't write if its a line
     else if (t > 1.) return f; // don't write if its a line
     return s + ((f-s) * t);
  }
    bool inside(pto p) { return abs(dist(s, p) + dist(p, f) - dist(s, f)
        ) < EPS; }
};

// Note: if the segments are collinear it only returns a point of intersection
pto inter(segm &s1, segm &s2){
    if (s1.inside(s2.s)) return s2.s; // if they are collinear
    if (s1.inside(s2.f)) return s2.f; // if they are collinear
  pto r = inter(line(s1.s, s1.f), line(s2.s, s2.f));
    if (s1.inside(r) && s2.inside(r)) return r;
  return pto(INF, INF);
}
```

5.6. Rectangle

```
struct rect { pto lw, up; }; // lower-left and upper-right corners
// Returns if there's an intersection and stores it in r
bool inter(rect a, rect b, rect &r){
  r.lw = pto(max(a.lw.x, b.lw.x), max(a.lw.y, b.lw.y));
  r.up = pto(min(a.up.x, b.up.x), min(a.up.y, b.up.y));
    // check case when only a edge is common
  return r.lw.x < r.up.x && r.lw.y < r.up.y;
}
```

5.7. Polygon Area

```
double area(vector<pto> &p) { // O(n)
  double area = 0; int n = si(p);
  forn(i, n) area += p[i] ^ p[(i+1) % n];
  // if points are in CW order then area is negative
  return abs(area) / 2;
}
// Area ellipse = M_PI*a*b where a and b are the semi axis lengths
// Area triangle = sqrt(s*(s-a)(s-b)(s-c)) where s = (a+b+c)/2
```

5.8. Circle

```
vec perp(vec v){return vec(-v.y, v.x);}
line bisector(pto x, pto y){
  line l=line(x, y); pto m=(x+y)/2;
  return line(-l.b, l.a, -l.b*m.x+l.a*m.y);
}
struct Circle{
  pto o;
  double r;
  Circle(pto x, pto y, pto z){
    o=inter(bisector(x, y), bisector(y, z));
    r=dist(o, x);
  }
  pair<pto, pto> ptosTang(pto p){
    pto m=(p+o)/2;
    tipo d=dist(o, m);
    tipo a=r*r/(2*d);
    tipo h=sqrt(r*r-a*a);
    pto m2=o+(m-o)*a/d;
    vec per=perp(m-o)/d;
    return make_pair(m2-per*h, m2+per*h);
  }
};
//finds the center of the circle containing p1 and p2 with radius r
//as there may be two solutions swap p1, p2 to get the other
bool circle2PtsRad(pto p1, pto p2, double r, pto &c){
        double d2=(p1-p2).norm2(), det=r*r/d2-0.25;
        if(det<0) return false;
        c=(p1+p2)/2+perp(p2-p1)*sqrt(det);
        return true;
}
#define sqr(a) ((a)*(a))
#define feq(a,b) (fabs((a)-(b))<EPS)
pair<tipo, tipo> ecCuad(tipo a, tipo b, tipo c){//a*x*x+b*x+c=0
  tipo dx = sqrt(b*b-4.0*a*c);
  return make_pair((-b + dx)/(2.0*a),(-b - dx)/(2.0*a));
}
pair<pto, pto> interCL(Circle c, line l){
  bool sw=false;
  if((sw=feq(0,l.b))){
  swap(l.a, l.b);
  swap(c.o.x, c.o.y);
  }
  pair<tipo, tipo> rc = ecCuad(
  sqr(l.a)+sqr(l.b),
  2.0*l.a*l.b*c.o.y-2.0*(sqr(l.b)*c.o.x+l.c*l.a),
  sqr(l.b)*(sqr(c.o.x)+sqr(c.o.y)-sqr(c.r))+sqr(l.c)-2.0*l.c*l.b*c.o.y
  );
  pair<pto, pto> p( pto(rc.first, (l.c - l.a * rc.first) / l.b),
            pto(rc.second, (l.c - l.a * rc.second) / l.b) );
  if(sw){
  swap(p.first.x, p.first.y);
  swap(p.second.x, p.second.y);
  }
  return p;
}

-----------------------------------------------------------------

struct circle {
    pto p; double r;
    bool contains(pto a) { return leq(dist(p, a), r); }
};

vector<pto> interCC(circle &a, circle &b) {
    vector<pto> r;
    double d = dist(a.p, b.p);
    if (gr(d, a.r + b.r) || le(d + min(a.r, b.r), max(a.r, b.r))) return
         r;
    double x = (d*d + a.r*a.r - b.r*b.r) / (2*d);
    double y = sqrt(a.r*a.r - x*x);
    pto v = (b.p - a.p) / d;
    r.pb(a.p + v*x + CCW90(v)*y);
    if (gr(y, 0)) r.pb(a.p + v*x - CCW90(v)*y);
```

```
    return r;
}
```

5.9. 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;
}
```

5.10. Point in Convex Poly log(n)

```
void normalize(vector<pto> &pt){ //delete collinear points first!
  //this makes it clockwise:
    if(pt[2].left(pt[0], pt[1])) reverse(pt.begin(), pt.end());
  int n=si(pt), pi=0;
  forn(i, n)
    if(pt[i].x<pt[pi].x || (pt[i].x==pt[pi].x && pt[i].y<pt[pi].y))
      pi=i;
  vector<pto> shift(n); //puts pi as first point
    forn(i, n) shift[i]=pt[(pi+i)%n];
    pt.swap(shift);
}
bool inPolygon(pto p, const vector<pto> &pt){
  //call normalize first!
  if(p.left(pt[0], pt[1]) || p.left(pt[si(pt)-1], pt[0])) return 0;
  int a=1, b=si(pt)-1;
  while(b-a>1){
    int c=(a+b)/2;
    if(!p.left(pt[0], pt[c])) a=c;
    else b=c;
  }
  return !p.left(pt[a], pt[a+1]);
}
```

5.11. Convex Check CHECK

```
bool isConvex(vi &p) { //O(N), delete collinear points!
  int n = sz(p);
  if (n < 3) return false;
  bool isLeft = p[0].left(p[1], p[2]);
  forsn(i, 1, n)
    if (p[i].left(p[(i+1) %n], p[(i+2) %n]) != isLeft)
      return false;
  return true;
}
```

5.12. 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();
}
```

5.13. 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)));
  }
}
```

5.14. Bresenham

```cpp
//plot a line approximation in a 2d map
void bresenham(pto a, pto b){
  pto d=b-a; d.x=abs(d.x), d.y=abs(d.y);
  pto s(a.x<b.x? 1: -1, a.y<b.y? 1: -1);
  int err=d.x-d.y;
  while(1){
    m[a.x][a.y]=1;//plot
    if(a==b) break;
    int e2=err;
    if(e2 >= 0) err-=2*d.y, a.x+=s.x;
    if(e2 <= 0) err+= 2*d.x, a.y+= s.y;
  }
}
```

5.15. Rotate Matrix

```cpp
//rotates matrix t 90 degrees clockwise
//using auxiliary matrix t2(faster)
void rotate(){
  forn(x, n) forn(y, n)
    t2[n-y-1][x]=t[x][y];
  memcpy(t, t2, sizeof(t));
}
```

5.16. Interseccion de Circulos en n3log(n)

```cpp
struct event {
    double x; int t;
    event(double xx, int tt) : x(xx), t(tt) {}
    bool operator <(const event &o) const { return x < o.x; }
};
typedef vector<Circle> VC;
typedef vector<event> VE;
int n;
double cuenta(VE &v, double A,double B) {
    sort(v.begin(), v.end());
    double res = 0.0, lx = ((v.empty())?0.0:v[0].x);
    int contador = 0;
    forn(i,sz(v)) {
        //interseccion de todos (contador == n), union de todos (contador > 0)
        //conjunto de puntos cubierto por exacta k Circulos (contador == k)
        if (contador == n) res += v[i].x - lx;
        contador += v[i].t, lx = v[i].x;
    }
    return res;
}
// Primitiva de sqrt(r*r - x*x) como funcion double de una variable x.
inline double primitiva(double x,double r) {
    if (x >= r) return r*r*M_PI/4.0;
    if (x <= -r) return -r*r*M_PI/4.0;
    double raiz = sqrt(r*r-x*x);
    return 0.5 * (x * raiz + r*r*atan(x/raiz));
}
double interCircle(VC &v) {
    vector<double> p; p.reserve(v.size() * (v.size() + 2));
    forn(i,sz(v))  p.push_back(v[i].c.x + v[i].r), p.push_back(v[i].c.x - v[i].r);
    forn(i,sz(v)) forn(j,i) {
        Circle &a = v[i], b = v[j];
        double d = (a.c - b.c).norm();
        if (fabs(a.r - b.r) < d && d < a.r + b.r) {
            double alfa = acos((sqr(a.r) + sqr(d) - sqr(b.r)) / (2.0 * d * a.r));
            pto vec = (b.c - a.c) * (a.r / d);
            p.pb((a.c + rotate(vec, alfa)).x), p.pb((a.c + rotate(vec, -alfa)).x);
        }
    }
    sort(p.begin(), p.end());
    double res = 0.0;
    forn(i,sz(p)-1) {
        const double A = p[i], B = p[i+1];
        VE ve; ve.reserve(2 * v.size());
        forn(j,sz(v)) {
            const Circle &c = v[j];
            double arco = primitiva(B-c.c.x,c.r) - primitiva(A-c.c.x,c.r);
            double base = c.c.y * (B-A);
            ve.push_back(event(base + arco,-1));
            ve.push_back(event(base - arco, 1));
        }
        res += cuenta(ve,A,B);
```

```
    }
    return res;
}
```

5.17. Cayley-Menger

Permite calcular el volumen de un simplex (triángulo en k dimensiones) mediante el cálculo de una determinante.

```
double d[5][5];

double sqr(double x) { return x*x; }

double init_cayley_menger() { // en los demas d[i][j] va la longitud del lado del vertice i al vertice j
    for (int i = 0; i < 4; i++) d[i][4] = d[4][i] = 1;
}
double cayley_menger(vector<int> idx) { // idx = indices de vertices, por ej {0, 1, 2, 3} en 3d
    idx.push_back(4);
    int n = (int) idx.size();

    Mat mat(n, n);
    forn(i,n) forn(j,n) mat[i][j] = sqr(d[idx[i]][idx[j]]);

    double ans = mat.determinant();
    forn(i,n-2) ans /= -2*(i+1)*(i+1);
    return sqrt(-ans);
}
```

5.18. Heron's formula

It states that the area of a triangle whose sides have lengths a, b, and c is $A = \sqrt{s(s-a)(s-b)(s-c)}$, where s is the semiperimeter of the triangle; that is, $s = \dfrac{a+b+c}{2}$.

6. 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.

6.1. 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];
}
```

6.2. Chull

Problema de ejemplo:
Recurrencia original:
Condición suficiente:
Complejidad original:
Complejidad optimizada:
Solución:

6.3. 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];
}
```

7. Matemática

7.1. Teoría de números

7.1.1. 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:

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

7.1.2. Teorema de Wilson

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

7.1.3. Pequeño teorema de Fermat

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

7.1.4. Teorema de Euler

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

7.2. Combinatoria

7.2.1. 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.

7.2.2. 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]);
    }
}
```

7.2.3. 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;
}
```

7.2.4. 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

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

```
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]);
}
```

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

7.2.5. 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]));
}
```

7.2.6. Eulerian

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

7.2.7. Catalan

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

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

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

7.3. Sumatorias conocidas

$\sum_{i=0}^{n} \binom{n}{i} = 2^n$
$\sum_{i=0}^{n} i\binom{n}{i} = n * 2^{n-1}$
$\sum_{i=m}^{n} i = \frac{n(n+1)}{2} - \frac{m(m-1)}{2} = \frac{(n+1-m)(n+m)}{2}$
$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$
$\sum_{i=0}^{n} i(i-1) = \frac{8}{6}(\frac{n}{2})(\frac{n}{2}+1)(n+1)$ (doubles) $\rightarrow$ Sino ver caso impar y par
$\sum_{i=0}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^4}{4} + \frac{n^3}{2} + \frac{n^2}{4} = \left[\sum_{i=1}^{n} i\right]^2$
$\sum_{i=0}^{n} i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30} = \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30}$
$\sum_{i=0}^{n} i^p = \frac{(n+1)^{p+1}}{p+1} + \sum_{k=1}^{p} \frac{B_k}{p-k+1} \binom{p}{k} (n+1)^{p-k+1}$

7.4. Ec. Característica

$a_0T(n) + a_1T(n-1) + ... + a_kT(n-k) = 0$
$p(x) = a_0x^k + a_1x^{k-1} + ... + a_k$
Sean $r_1, r_2, ..., r_q$ las raíces distintas, de mult. $m_1, m_2, ..., m_q$
$T(n) = \sum_{i=1}^{q} \sum_{j=0}^{m_i-1} c_{ij} n^j r_i^n$
Las constantes c_{ij} se determinan por los casos base.

7.5. Aritmetica Modular

```
const int M = 1e9 + 7;
int add(int a, int b){ return a+b < M ? a+b : a+b-M; }
int sub(int a, int b){ return a-b >= 0 ? a-b : a-b+M; }
int mul(int a, int b){ return int(ll(a)*b % M); }
int pot(int b, int e){ // O(log e)
  int r=1;
    while (e) {
        if (e&1) r = mul(r,b);
        e >>= 1; b = mul(b,b);
    }
    return r;
}
int inv(int x){ return pot(x, M-2); } // Change M-2 for Phi(M)-1 if M
    isn't prime
int divide(int a, int b) { return mul(a, inv(b)); }
int neg(int a){ return add(-a, M); }
int normal(int a){ return ((a % M) + M) % M;} // For neg numbers
```

7.6. Exp. de Numeros Mod.

```
ll pot(ll b, ll e){ // O(log e)
  ll r=1;
    while (e) {
        if (e&1) r *= b;
        e >>= 1; b *= b;
    }
    return r;
}
```

7.7. Exp. de Matrices y Fibonacci en log(n)

```
const int S = 2;
int temp[S][S];
void mul(int a[S][S], int b[S][S]){
    forn(i, S) forn(j, S) temp[i][j] = 0;
    forn(i, S) forn(j, S) forn(k, S) temp[i][j]+=a[i][k]*b[k][j];
    forn(i, S) forn(j, S) a[i][j]=temp[i][j];
}
void powmat(int a[S][S], ll n, int res[S][S]){
    forn(i, S) forn(j, S) res[i][j]=(i==j);
    while (n) {
        if (n&1) mul(res, a);
        n >>= 1; mul(a, a);
    }
}
```

7.8. Matrices y determinante $O(n^3)$

```
struct Mat {
    vector<vector<double> > vec;
    Mat(int n): vec(n, vector<double>(n) ) {}
    Mat(int n, int m): vec(n, vector<double>(m) ) {}
    vector<double> &operator[](int f){return vec[f];}
    const vector<double> &operator[](int f) const {return vec[f];}
    int size() const {return si(vec);}
    Mat operator+(Mat &b) { ///this de n x m entonces b de n x m
        Mat m(si(b),si(b[0]));
        forn(i,si(vec)) forn(j,si(vec[0])) m[i][j] = vec[i][j] + b[i][j];
        return m;    }
    Mat operator*(const Mat &b) { ///this de n x m entonces b de m x t
        int n = si(vec), m = si(vec[0]), t = si(b[0]);
        Mat mat(n,t);
        forn(i,n) forn(j,t) forn(k,m) mat[i][j] += vec[i][k] * b[k][j];
        return mat;    }
    double determinant(){//sacado de e maxx ru
        double det = 1;
        int n = si(vec);
        Mat m(*this);
        forn(i, n){//para cada columna
            int k = i;
            forsn(j, i+1, n)//busco la fila con mayor val abs
                if(abs(m[j][i])>abs(m[k][i])) k = j;
            if(abs(m[k][i])<EPS) return 0;
            m[i].swap(m[k]);//la swapeo
            if(i!=k) det = -det;
            det *= m[i][i];
            forsn(j, i+1, n) m[i][j] /= m[i][i];
            //hago 0 todas las otras filas
            forn(j, n) if (j!= i && abs(m[j][i])>EPS)
                forsn(k, i+1, n) m[j][k]-=m[i][k]*m[j][i];
        }
        return det;
    }
};
```

7.9. Primos

```
// P keeps primes until N. Check if a number is prime with lp[x] == x.
const int N = 1e6;
vi lp(N+1), P;

void sieve() { // O(n)
  forsn(i, 2, N+1) {
    if (!lp[i]) lp[i] = i, P.pb(i);
        for (int p : P) {
            if (p > lp[i] || i*p > N) break;
            lp[i * p] = p;
        }
  }
}

void eratosthenes() { // O(n * log log n)
    forsn(i, 2, N+1) lp[i] = i & 1 ? i : 2;
    for (int i = 3; i*i <= N; i += 2) if (lp[i] == i) {
        for (int j = i*i; j <= N; j += 2*i) if (lp[j] == j) lp[j] = i;
        P.pb(i);
    }
}

bool prime(int x) { // O(sqrt x)
    if (x < 2 || x % 2 == 0) return false;
    for (int i = 3; i*i <= x; i += 2)
        if (x % i == 0) return false;
  return true;
```

```
}
```

7.10. Factorizacion

Sea $n = \prod p_i^{k_i}$, fact(n) genera un map donde a cada p_i le asocia su k_i

```
// Both functions require sieve to work

map<ll,int> fact(int x) { // O(lg x), x <= N
    map<ll,int> f;
    while (x > 1) f[lp[x]]++, x /= lp[x];
    return f;
}

map<ll,int> fact(ll x) { // O(sqrt x), x <= N*N
    map<ll,int> f;
    for (int p : P) {
        if (ll(p)*p > x) break;
        while (x % p == 0) f[p]++, x /= p;
    }
    if(x > 1) f[x]++;
    return f;
}
```

7.11. Divisores

```
const int N = 1e6;
vi C(N+1), D[N+1]; // D[x] contains all the divisors of x

void generateDivisors() { // O(n lg n) because of the harmonic series
    forsn(i, 1, N+1) for (int j = i; j <= N; j += i) C[j]++;
    forsn(i, 1, N+1) D[i] = vi(C[i]), C[i] = 0;
    forsn(i, 1, N+1) for (int j = i; j <= N; j += i) D[j][C[j]++] = i;
}

typedef vector<ll> vll;

vll divisors(ll x) { // O(sqrt x)
    vll r;
    for (ll i = 1; i*i <= x; i++) {
        ll d = x/i;
        if (d*i == x) {
            r.pb(i);
            if (d != i) r.pb(d);
        }
    }
    return r;
}

vll divisors(const map<ll,int> &f) { // O(num of divs)
    vll d = {1}; // divs are unordered
    for (auto &i : f) {
        ll b = 1, n = si(d);
        forn(_, i.snd) {
            b *= i.fst;
            forn(j, n) d.pb(b * d[j]);
        }
    }
    return d;
}

ll sumDivisors(ll x) { // O(lg x)
    ll r = 1;
    map<ll,int> f = fact(x);
    for (auto &i : f) {
      ll pow = 1, s = 0;
      forn(j, i.snd + 1)
            s += pow, pow *= i.fst;
      r *= s;
    }
    return r;
}
```

7.12. Euler's Phi

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

const int N = 1e6;
vi lp(N+1), P, phi(N+1);

void initPhi() { // Least prime and phi <= N in O(n)
    phi[1] = 1;
    forsn(i, 2, N+1) {
        if (!lp[i])
            lp[i] = i, P.pb(i), phi[i] = i-1;
```

```
        else {
            int a = i / lp[i];
            phi[i] = phi[a] * (lp[i] - (lp[i] != lp[a]));
        }
        for (int p : P) {
            if (p > lp[i] || i*p > N) break;
            lp[i * p] = p;
        }
    }
}

ll eulerPhi(ll x) { // O(lg x)
    ll r = x;
    map<ll,int> f = fact(x);
    for (auto &i : f) r -= r / i.fst;
    return r;
}

ll eulerPhi(ll x) { // O(sqrt x)
    ll r = x;
    for (ll i = 2; i*i <= x; i++) {
        if (x % i == 0) {
            r -= r/i;
            while (x % i == 0) x /= i;
        }
    }
    if (x > 1) r -= r/x;
    return r;
}
```

7.13. Phollard's Rho - Miller-Rabin

```
ll gcd(ll a, ll b){return b?__gcd(a,b):a;}

typedef unsigned long long ull;
ull mulmod(ull a, ull b, ull m){ // 0 <= a, b < m
   long double x; ull c; ll r;
   x = a; c = x * b / m;
   r = (ll)(a * b - c * m) % (ll)m;
   return r < 0 ? r + m : r;
}

ll expmod(ll b, ll e, ll m){ // O(log(b))
  ll ans = 1;
  while(e){
        if(e&1)ans = mulmod(ans, b, m);
        b = mulmod(b, b, m); e >>= 1;
  }
  return ans;
}

bool es_primo_prob (ll n, int a)
{
  if (n == a) return true;
  ll s = 0,d = n-1;
  while (d % 2 == 0) s++,d/=2;

  ll x = expmod(a,d,n);
  if ((x == 1) || (x+1 == n)) return true;

  forn (i, s-1){
    x = mulmod(x, x, n);
    if (x == 1) return false;
    if (x+1 == n) return true;
  }
  return false;
}

bool rabin (ll n){ //devuelve true si n es primo O(n^0.25)
  if (n == 1) return false;
  const int ar[] = {2,3,5,7,11,13,17,19,23};
  forn (j,9)
    if (!es_primo_prob(n,ar[j]))
      return false;
  return true;
}

ll rho(ll n){
    if(!(n&1))return 2;
    ll x = 2, y = 2, d = 1;
    ll c = rand()%n + 1;
    while(d == 1){
        x = (mulmod(x,x, n)+c)%n;
        y = (mulmod(y,y, n)+c)%n;
        y = (mulmod(y,y, n)+c)%n;
        if(x >= y)d = gcd(x-y, n);
```

```
        else d = gcd(y-x, n);
    }
    return d == n ? rho(n) : d;
}
void fact(ll n, map<ll,int>& f){ //O (lg n)^3
  if(n == 1)return;
  if(rabin(n)){ f[n]++; return; }
  ll q = rho(n); fact(q, f); fact(n/q, f);
}
```

7.14. GCD

```
// Predefined in C++17: gcd(a, b)
template<class T> T gcd(T a, T b) { return b ? __gcd(a,b) : a; }
```

7.15. LCM

```
// Predefined in C++17: lcm(a, b)
template<class T> T lcm(T a, T b) { return a * (b / gcd(a,b)); }
```

7.16. 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);
}
```

7.17. 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)
```

7.18. 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)
}
```

7.19. 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
```

7.20. 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.;}
```

7.21. 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;}
};
```

7.22. Polinomio, Ruffini e interpolación de Lagrange

Interpolación de Lagrange: dados $n+1$ pares (x_i, y_i) permite encontrar el polinomio de grado n tal que $f(x_i) = y_i$.
Explicación: computa $P(x) = y_1 * f_1(x) + y_2 * f_2(x) + \ldots + y_{n+1} * f_{n+1}(x)$ donde $f_i(x) = \frac{g_i(x)}{g_i(x_i)}$, $g_i(x) = \frac{h(x)}{x-x_i}$ y $h(x) = (x-x_1)*(x-x_2)*\ldots*(x-x_{n+1})$. Usa Ruffini para la división de polinomios.
Trucazo para computar en $O(n)$: $x_{i+1} - x_i = x_{j+1} - x_j$ para todo $i, j < n$.
Ejemplo de problema: tenés que calcular una respuesta que depende de un n y parece ser polinomial, conseguís un par de puntos e intentás armar el polinomio (usando el algoritmo online u offline).

```
using tp = int; // type of polynomial
template<class T=tp>
struct poly {  // poly<> : 1 variable, poly<poly<>>: 2 variables, etc.
  vector<T> c;
  T& operator[](int k){return c[k];}
  poly(vector<T>& c):c(c){}
  poly(initializer_list<T> c):c(c){}
  poly(int k):c(k){}
  poly(){}
  poly operator+(poly<T> o){
    int m=si(c),n=si(o.c);
    poly res(max(m,n));
    forn(i,m)res[i]=res[i]+c[i];
    forn(i,n)res[i]=res[i]+o.c[i];
    return res;
  }
  poly operator*(tp k){
    poly res(si(c));
    forn(i,si(c))res[i]=c[i]*k;
    return res;
  }
  poly operator*(poly o){
    int m=si(c),n=si(o.c);
    poly res(m+n-1);
    forn(i,m)forn(j,n)res[i+j]=res[i+j]+c[i]*o.c[j];
    return res;
  }
  poly operator-(poly<T> o){return *this+(o*-1);}
  T operator()(tp v){
    T sum(0);
    dforn(i, si(c)) sum=sum*v+c[i];
    return sum;
```

```
  }
};
// example: p(x,y)=2*x^2+3*x*y-y+4
// poly<poly<>> p={{4,-1},{0,3},{2}}
// printf("%d\n",p(2)(3)) // 27 (p(2,3))
set<tp> roots(poly<> p){ // only for integer polynomials
  set<tp> r;
  while(!p.c.empty()&&!p.c.back())p.c.pop_back();
  if(!p(0))r.insert(0);
  if(p.c.empty())return r;
  tp a0=0,an=abs(p[si(p.c)-1]);
  for(int k=0;!a0;a0=abs(p[k++]));
  vector<tp> ps,qs;
  forsn(i,1,sqrt(a0)+1)if(a0%i==0)ps.pb(i),ps.pb(a0/i);
  forsn(i,1,sqrt(an)+1)if(an%i==0)qs.pb(i),qs.pb(an/i);
  for(auto pt:ps)for(auto qt:qs)if(pt%qt==0){
    tp x=pt/qt;
    if(!p(x))r.insert(x);
    if(!p(-x))r.insert(-x);
  }
  return r;
}
pair<poly<>,tp> ruffini(poly<> p, tp r){ // returns pair (result,rem)
  int n=si(p.c)-1;
  vector<tp> b(n);
  b[n-1]=p[n];
  dforn(k, n-1) b[k]=p[k+1]+r*b[k+1];
  return mp(poly<>(b),p[0]+r*b[0]);
}
// only for double polynomials
pair<poly<>,poly<> > polydiv(poly<> p, poly<> q){ // returns pair (result,rem)
  int n=si(p.c)-si(q.c)+1;
  vector<tp> b(n);
  dforn(k, n) {
    b[k]=p.c.back()/q.c.back();
    forn(i,si(q.c))p[i+k]-=b[k]*q[i];
    p.c.pop_back();
  }
  while(!p.c.empty()&&abs(p.c.back())<EPS)p.c.pop_back();
  return mp(poly<>(b),p);
}
// for double polynomials
// O(n^2), constante aaaalta
poly<> interpolate(vector<tp> x, vector<tp> y){
  poly<> q={1},S={0};
  for(tp a:x)q=poly<>({-a,1})*q;
  forn(i,si(x)){
    poly<> Li=ruffini(q,x[i]).fst;
    Li=Li*(1.0/Li(x[i])); // change for int polynomials
    S=S+Li*y[i];
  }
  return S;
}
// for int polynomials
// O(n), rapido, la posta
int evalInterpolation(const vector<int> &y, int x) { // {0, y[0]}, ...
    int ans = 0;
    int k = 1;
    forsn(j, 1, si(y)) {
        if (x == j) return y[j];
        k = mul(k, normal(x - j));
        k = div(k, normal(0 - j));
    }
    forn(i, si(y)) {
        ans = add(ans, mul(y[i], k));
        if (i + 1 >= si(y)) break;
        k = mul(k, div(normal(x - i), normal(x - (i + 1))));
        k = mul(k, div(normal(i - (si(y) - 1)), normal(i + 1))); // TODO: terminar de explicar esta linea
    }
    return ans;
}
```

7.23. Ec. Lineales

```
bool resolver_ev(Mat a, Vec y, Vec &x, Mat &ev){
  int n = a.size(), m = n?a[0].size():0, rw = min(n, m);
  vector<int> p; forn(i,m) p.push_back(i);
  forn(i, rw) {
    int uc=i, uf=i;
    forr(f, i, n) forr(c, i, m) if(fabs(a[f][c])>fabs(a[uf][uc])) {uf=f;uc=c;}
    if (feq(a[uf][uc], 0)) { rw = i; break; }
    forn(j, n) swap(a[j][i], a[j][uc]);
```

```
    swap(a[i], a[uf]); swap(y[i], y[uf]); swap(p[i], p[uc]);
    tipo inv = 1 / a[i][i]; //aca divide
    forr(j, i+1, n) {
      tipo v = a[j][i] * inv;
      forr(k, i, m) a[j][k]-=v * a[i][k];
      y[j] -= v*y[i];
    }
  } // rw = rango(a), aca la matriz esta triangulada
  forr(i, rw, n) if (!feq(y[i],0)) return false; // checkeo de compatibilidad
  x = vector<tipo>(m, 0);
  dforn(i, rw){
    tipo s = y[i];
    forr(j, i+1, rw) s -= a[i][j]*x[p[j]];
    x[p[i]] = s / a[i][i]; //aca divide
  }
  ev = Mat(m-rw, Vec(m, 0)); // Esta parte va SOLO si se necesita el ev
  forn(k, m-rw) {
    ev[k][p[k+rw]] = 1;
    dforn(i, rw){
      tipo s = -a[i][k+rw];
      forr(j, i+1, rw) s -= a[i][j]*ev[k][p[j]];
      ev[k][p[i]] = s / a[i][i]; //aca divide
    }
  }
  return true;
}
```

7.24. 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];
}
```

7.25. 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);
    }
};
```

7.26. Tablas y cotas (Primos, Divisores, Factoriales, etc)

Factoriales

$0! = 1$	$11! = 39.916.800$
$1! = 1$	$12! = 479.001.600$ ($\in$ `int`)
$2! = 2$	$13! = 6.227.020.800$
$3! = 6$	$14! = 87.178.291.200$
$4! = 24$	$15! = 1.307.674.368.000$
$5! = 120$	$16! = 20.922.789.888.000$
$6! = 720$	$17! = 355.687.428.096.000$
$7! = 5.040$	$18! = 6.402.373.705.728.000$
$8! = 40.320$	$19! = 121.645.100.408.832.000$
$9! = 362.880$	$20! = 2.432.902.008.176.640.000$ ($\in$ `tint`)
$10! = 3.628.800$	$21! = 51.090.942.171.709.400.000$

max signed tint = 9.223.372.036.854.775.807
max unsigned tint = 18.446.744.073.709.551.615

Primos

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093 1097 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427 1429 1433 1439 1447 1451 1453 1459 1471 1481 1483 1487 1489 1493 1499 1511 1523 1531 1543 1549 1553 1559 1567 1571 1579 1583 1597 1601 1607 1609 1613 1619 1621 1627 1637 1657 1663 1667 1669 1693 1697 1699 1709 1721 1723 1733 1741 1747 1753 1759 1777 1783 1787 1789 1801 1811 1823 1831 1847 1861 1867 1871 1873 1877 1879 1889 1901 1907 1913 1931 1933 1949 1951 1973 1979 1987 1993 1997 1999 2003 2011 2017 2027 2029 2039 2053 2063 2069 2081

Primos cercanos a 10^n

9941 9949 9967 9973 10007 10009 10037 10039 10061 10067 10069 10079
99961 99971 99989 99991 100003 100019 100043 100049 100057 100069
999959 999961 999979 999983 1000003 1000033 1000037 1000039
9999943 9999971 9999973 9999991 10000019 10000079 10000103 10000121
99999941 99999959 99999971 99999989 100000007 100000037 100000039 100000049
999999893 999999929 999999937 1000000007 1000000009 1000000021 1000000033

Cantidad de primos menores que 10^n

$\pi(10^1) = 4$; $\pi(10^2) = 25$; $\pi(10^3) = 168$; $\pi(10^4) = 1229$; $\pi(10^5) = 9592$
$\pi(10^6) = 78.498$; $\pi(10^7) = 664.579$; $\pi(10^8) = 5.761.455$; $\pi(10^9) = 50.847.534$
$\pi(10^{10}) = 455.052{,}511$; $\pi(10^{11}) = 4.118.054.813$; $\pi(10^{12}) = 37.607.912.018$
Observación: Una buena aproximación es $x/ln(x)$.

Divisores

Cantidad de divisores (σ_0) para *algunos* $n/\neg\exists n' < n, \sigma_0(n') \geqslant \sigma_0(n)$
Referencias: $\sigma_0(10^9) = 1344$ y $\sigma_0(10^{18}) = 103680$
$\sigma_0(60) = 12$; $\sigma_0(120) = 16$; $\sigma_0(180) = 18$; $\sigma_0(240) = 20$; $\sigma_0(360) = 24$
$\sigma_0(720) = 30$; $\sigma_0(840) = 32$; $\sigma_0(1260) = 36$; $\sigma_0(1680) = 40$; $\sigma_0(10080) = 72$
$\sigma_0(15120) = 80$; $\sigma_0(50400) = 108$; $\sigma_0(83160) = 128$; $\sigma_0(110880) = 144$
$\sigma_0(498960) = 200$; $\sigma_0(554400) = 216$; $\sigma_0(1081080) = 256$; $\sigma_0(1441440) = 288$
$\sigma_0(4324320) = 384$; $\sigma_0(8648640) = 448$
Observación: Una buena aproximación es $x^{1/3}$.

Suma de divisores (σ_1) para *algunos* $n/\neg\exists n' < n, \sigma_1(n') \geqslant \sigma_1(n)$
$\sigma_1(96) = 252$; $\sigma_1(108) = 280$; $\sigma_1(120) = 360$; $\sigma_1(144) = 403$; $\sigma_1(168) = 480$
$\sigma_1(960) = 3048$; $\sigma_1(1008) = 3224$; $\sigma_1(1080) = 3600$; $\sigma_1(1200) = 3844$
$\sigma_1(4620) = 16128$; $\sigma_1(4680) = 16380$; $\sigma_1(5040) = 19344$; $\sigma_1(5760) = 19890$
$\sigma_1(8820) = 31122$; $\sigma_1(9240) = 34560$; $\sigma_1(10080) = 39312$; $\sigma_1(10920) = 40320$
$\sigma_1(32760) = 131040$; $\sigma_1(35280) = 137826$; $\sigma_1(36960) = 145152$; $\sigma_1(37800) = 148800$
$\sigma_1(60480) = 243840$; $\sigma_1(64680) = 246240$; $\sigma_1(65520) = 270816$; $\sigma_1(70560) = 280098$
$\sigma_1(95760) = 386880$; $\sigma_1(98280) = 403200$; $\sigma_1(100800) = 409448$
$\sigma_1(491400) = 2083200$; $\sigma_1(498960) = 2160576$; $\sigma_1(514080) = 2177280$
$\sigma_1(982800) = 4305280$; $\sigma_1(997920) = 4390848$; $\sigma_1(1048320) = 4464096$
$\sigma_1(4979520) = 22189440$; $\sigma_1(4989600) = 22686048$; $\sigma_1(5045040) = 23154768$
$\sigma_1(9896040) = 44323200$; $\sigma_1(9959040) = 44553600$; $\sigma_1(9979200) = 45732192$

8. Grafos

8.1. Teoremas y fórmulas

8.1.1. 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.

8.1.2. 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.

8.2. Dijkstra

```
const ll N = 2e5, INF = 1e18;
typedef pair<ll,int> pli;
ll dist[N]; int par[N];
vector<pii> g[N];
bool seen[N];

ll dijkstra(int n, int s=0, int t=-1) { // O(E lg V)
    forn(i, n) dist[i] = INF, seen[i] = 0, par[i] = -1;
  priority_queue<pli, vector<pli>, greater<pli>> q;
  q.emplace(0, s), dist[s] = 0;

  while (!q.empty()){
    int u = q.top().snd; q.pop();
        if (seen[u]) continue;
        seen[u] = true;
    if (u == t) break;
    for (auto &e : g[u]) {
            int v, w; tie(v, w) = e;
      if (dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        par[v] = u;
        q.emplace(dist[v], v);
            }
        }
  }
  return t != -1 ? dist[t] : 0;
}

// Path generator:
vi path;
if (dist[t] != INF) {
    for (int u = t; u != -1; u = par[u]) path.pb(u);
    reverse(all(path));
}
```

8.3. Bellman-Ford

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

8.4. Floyd-Warshall

```
// if i != j, g[i][j] = weight of edge (i,j) or INF, else g[i][i] = 0
// For multigraphs: remember to keep the shortest direct paths
const int INF = 1e9, N = 200;
int g[N][N];
void floyd_warshall(int n) { // O(n^3)
    forn(k, n)
        forn(i, n) if (g[i][k] != INF)
            forn(j, n) if (g[k][j] != INF)
              g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
}

bool inNegCycle(int u) { return g[u][u] < 0; }

// Checks if there's a negative cycle in path from a to b (precomputable
    ):
bool hasNegCycle(int n, int a, int b) {
  forn(i, n) if (g[i][i] < 0 && g[a][i] != INF && g[i][b] != INF)
    return true;
  return false;
}
```

8.5. Kruskal

```
struct Edge {
    int u, v, c;
    Edge(int _u, int _v, int _c) : u(_u), v(_v), c(_c) {}
    bool operator < (const Edge &o) const { return c < o.c; }
```

```
};

struct Kruskal {
    vector<Edge> edges;
    int n;

    Kruskal(int _n) : n(_n) {}
    void addEdge(int u, int v, int c) { edges.pb(u, v, c); }

    ll build() {
        sort(all(edges));

        UF uf(n);
        ll cost = 0;
        for (Edge &edge : edges) {
            if (uf.join(edge.u, edge.v)) {
                cost += edge.c;
            }
        }
        return cost;
    }
};
```

8.6. Prim

```
bool taken[MAXN];
priority_queue<ii, vector<ii>, greater<ii> > pq;//min heap
void process(int v){
    taken[v]=true;
    forall(e, G[v])
        if(!taken[e->second]) pq.push(*e);
}

ll prim(){
    zero(taken);
    process(0);
    ll cost=0;
    while(sz(pq)){
        ii e=pq.top(); pq.pop();
        if(!taken[e.second]) cost+=e.first, process(e.second);
    }
    return cost;
}
```

8.7. 2-SAT + Tarjan SCC

```
//We have a vertex representing a var and other for his negation.
//Every edge stored in G represents an implication. To add an equation
    of the form a||b, use addor(a, b)
//N=max cant var, n=cant var
struct SAT {
    const static int N = 1e5;

    vector<int> adj[N*2];
    //idx[i]=index assigned in the dfs
    //lw[i]=lowest index(closer from the root) reachable from i
    int lw[N*2], idx[N*2], qidx;
    stack<int> q;
    int qcmp, cmp[N*2];
    //value[cmp[i]]=valor de la variable i
    bool value[N*2+1];
    int n;

    //remember to CALL INIT!!!
    void init(int _n) {
        n = _n;
        forn(u, 2*n) adj[u].clear();
    }

    int neg(int x) { return x >=n ? x-n : x+n; }
    void addor(int a, int b) { adj[neg(a)].pb(b), adj[neg(b)].pb(a); }

    void tjn(int v){
        lw[v]=idx[v]=++qidx;
        q.push(v), cmp[v]=-2;
        for (auto u : adj[v]){
            if (!idx[u] || cmp[u]==-2){
                if (!idx[u]) tjn(u);
                lw[v]=min(lw[v], lw[u]);
            }
        }
        if (lw[v]==idx[v]){
            int x;
            do { x=q.top(); q.pop(); cmp[x]=qcmp; } while (x!=v);
            value[qcmp]=(cmp[neg(v)]<0);
            qcmp++;
        }
```

```
    }

    bool satisf(){ //O(n)
        memset(idx, 0, sizeof(idx)), qidx=0;
        memset(cmp, -1, sizeof(cmp)), qcmp=0;
        forn(i, n){
            if (!idx[i]) tjn(i);
            if (!idx[neg(i)]) tjn(neg(i));
        }
        forn(i, n) if (cmp[i]==cmp[neg(i)]) return false;
        return true;
    }
};
```

8.8. Kosaraju

```
struct Kosaraju {
  static const int default_sz = 1e5+10;
  int n;
  vector<vi> G, revG, C, ady; // ady is the condensed graph
  vi used, where;
  Kosaraju(int sz = default_sz){
    n = sz;
    G.assign(sz, vi());
    revG.assign(sz, vi());
    used.assign(sz, 0);
    where.assign(sz, -1);
  }
  void addEdge(int a, int b){ G[a].pb(b); revG[b].pb(a); }
  void dfsNormal(vi &F, int u){
    used[u] = true;
    for (int v : G[u]) if(!used[v])
      dfsNormal(F, v);
    F.pb(u);
  }
  void dfsRev(vi &F, int u){
    used[u] = true;
    for (int v : revG[u]) if(!used[v])
      dfsRev(F, v);
    F.pb(u);
  }
  void build(){
    vi T;
    fill(all(used), 0);
    forn(i, n) if(!used[i]) dfsNormal(T, i);
    reverse(all(T));
    fill(all(used), 0);
    for (int u : T)
      if(!used[u]){
        vi F;
        dfsRev(F, u);
        for (int v : F) where[v] = si(C);
        C.pb(F);
      }
    ady.resize(si(C)); // Create edges between condensed nodes
    forn(u, n) for(int v : G[u]){
      if(where[u] != where[v]){
        ady[where[u]].pb(where[v]);
      }
    }
    forn(u, si(C)){
      sort(all(ady[u]));
      ady[u].erase(unique(all(ady[u])), ady[u].end());
    }
  }
};
```

8.9. Articulation Points

```
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;
}
```

8.10. 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]);
      }
    }
  }
};
```

8.11. LCA + Climb

```
#define lg(x) (31-__builtin_clz(x))
struct LCA {
    static const int L = 20;
    int n, a[N][L], lvl[N]; // a[i][k] is the 2^k ancestor of i

    void dfs(int u=0, int p=-1, int d=0){
        a[u][0] = p, lvl[u] = d;
        for(int v : tree[u]) if(v != p) dfs(v,u,d+1);
    }

    void init(int m){
        n = m; dfs(); forn(k, L-1) forn(i,n) if(a[i][k] != -1) a[i][k+1]
             = a[a[i][k]][k];
    }

```

```
    int climb(int x, int d){
        if(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(lgn)
        if(lvl[x] < lvl[y]) swap(x,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)]; }
} lca;
```

8.12. Heavy Light Decomposition

```
// Usa RMQ Dynamic
// ATENCION: valores en nodos. Ver comments para valores en arcos.
template <int V, class T>
class HeavyLight {
    int parent[V], heavy[V], depth[V];
    int root[V], treePos[V];
    RMQ<V, T, T> tree;

    template <class G>
        int dfs(const G& graph, int v) {
            int size = 1, maxSubtree = 0;
            for (int u : graph[v]) if (u != parent[v]) {
                parent[u] = v;
                depth[u] = depth[v] + 1;
                int subtree = dfs(graph, u);
                if (subtree > maxSubtree) heavy[v] = u, maxSubtree =
                    subtree;
                size += subtree;
            }
            return size;
        }

    template <class BinaryOperation>
        void processPath(int u, int v, BinaryOperation op) {
            for (; root[u] != root[v]; v = parent[root[v]]) {
                if (depth[root[u]] > depth[root[v]]) swap(u, v);
                op(treePos[root[v]], treePos[v] + 1);
            }
            if (depth[u] > depth[v]) swap(u, v);
            // ATENCION: para valores almacenados en arcos: cambiar por
                op(treePos[u]+1, treePos[v]+1)
            op(treePos[u], treePos[v] + 1);
        }

    public:
    // ATENCION: grafo como vector<vector<int>>
    template <class G>
        void init(const G& graph) {
            int n = si(graph);
            fill_n(heavy, n, -1);
            parent[0] = -1;
            depth[0] = 0;
            dfs(graph, 0);
            for (int i = 0, currentPos = 0; i < n; ++i)
                if (parent[i] == -1 || heavy[parent[i]] != i)
                    for (int j = i; j != -1; j = heavy[j]) {
                        root[j] = i;
                        treePos[j] = currentPos++;
                    }
            tree.init(n);
        }

    void set(int v, const T& value) {
        tree.modify(treePos[v], treePos[v]+1, value);
    }

    void modifyPath(int u, int v, const T& value) {
        processPath(u, v, [this, &value](int l, int r) { tree.modify(
            value, l, r); });
    }

    T queryPath(int u, int v) {
        T res = T();
        processPath(u, v, [this, &res](int l, int r) { res += tree.get(l
```

```
            , r); });
        return res;
    }
};
```

8.13. Centroid Decomposition

```
struct Centroid {
    int n, sz[N], parent[N]; bool used[N];

    int size(int u, int p=-1){
        sz[u] = 1;
        for(int v : tree[u])
            if(v != p && !used[v]) sz[u] += size(v,u);
        return sz[u];
    }

    void build(int u=0, int p=-1, int s=-1){
        if(s == -1) s = size(u);
        for(int v : tree[u]) if(!used[v] && sz[v] > s/2)
            { sz[u] = 0; build(v,p,s); return; }
        used[u] = true, parent[u] = p;
        for(int v : tree[u]) if(!used[v]) build(v,u,-1);
    }
```

8.14. Euler Cycle

```
int n,m,ars[MAXE], eq;
vector<int> G[MAXN];//fill G,n,m,ars,eq
list<int> path;
int used[MAXN];
bool usede[MAXE];
queue<list<int>::iterator> q;
int get(int v){
  while(used[v]<sz(G[v]) && usede[ G[v][used[v]] ]) used[v]++;
  return used[v];
}
void explore(int v, int r, list<int>::iterator it){
  int ar=G[v][get(v)]; int u=v^ars[ar];
  usede[ar]=true;
  list<int>::iterator it2=path.insert(it, u);
  if(u!=r) explore(u, r, it2);
  if(get(v)<sz(G[v])) q.push(it);
}
void euler(){
  zero(used), zero(usede);
  path.clear();
  q=queue<list<int>::iterator>();
  path.push_back(0); q.push(path.begin());
  while(sz(q)){
    list<int>::iterator it=q.front(); q.pop();
    if(used[*it]<sz(G[*it])) explore(*it, *it, it);
  }
  reverse(path.begin(), path.end());
}
void addEdge(int u, int v){
  G[u].pb(eq), G[v].pb(eq);
  ars[eq++]=u^v;
}
```

8.15. Diametro árbol

```
int n;
vi adj[N];

pii farthest(int u, int p = -1) {
    pii ans = {-1, u};

    for (int v : adj[u])
        if (v != p)
            ans = max(ans, farthest(v, u));

    ans.fst++;
    return ans;
}

int diam(int r) {
    return farthest(farthest(r).snd).fst;
}

bool path(int s, int e, vi &p, int pre = -1) {
    p.pb(s);
    if (s == e) return true;

    for (int v : adj[s])
        if (v != pre && path(v, e, p, s))
            return true;
```

```
    p.pop_back();
    return false;
}

int center(int r) {
    int s = farthest(r).snd, e = farthest(s).snd;
    vi p; path(s, e, p);
    return p[si(p)/2];
}
```

8.16. Chu-liu

```
void visit(graph &h, int v, int s, int r,
  vector<int> &no, vector< vector<int> > &comp,
  vector<int> &prev, vector< vector<int> > &next, vector<weight> &mcost,
  vector<int> &mark, weight &cost, bool &found) {
  if (mark[v]) {
    vector<int> temp = no;
    found = true;
    do {
      cost += mcost[v];
      v = prev[v];
      if (v != s) {
        while (comp[v].size() > 0) {
          no[comp[v].back()] = s;
          comp[s].push_back(comp[v].back());
          comp[v].pop_back();
        }
      }
    } while (v != s);
    forall(j,comp[s]) if (*j != r) forall(e,h[*j])
      if (no[e->src] != s) e->w -= mcost[ temp[*j] ];
  }
  mark[v] = true;
  forall(i,next[v]) if (no[*i] != no[v] && prev[no[*i]] == v)
    if (!mark[no[*i]] || *i == s)
      visit(h, *i, s, r, no, comp, prev, next, mcost, mark, cost, found)
          ;
}
weight minimumSpanningArborescence(const graph &g, int r) {
    const int n=sz(g);
  graph h(n);
  forn(u,n) forall(e,g[u]) h[e->dst].pb(*e);
  vector<int> no(n);
  vector<vector<int> > comp(n);
  forn(u, n) comp[u].pb(no[u] = u);
  for (weight cost = 0; ;) {
    vector<int> prev(n, -1);
    vector<weight> mcost(n, INF);
    forn(j,n) if (j != r) forall(e,h[j])
      if (no[e->src] != no[j])
        if (e->w < mcost[ no[j] ])
          mcost[ no[j] ] = e->w, prev[ no[j] ] = no[e->src];
    vector< vector<int> > next(n);
    forn(u,n) if (prev[u] >= 0)
      next[ prev[u] ].push_back(u);
    bool stop = true;
    vector<int> mark(n);
    forn(u,n) if (u != r && !mark[u] && !comp[u].empty()) {
      bool found = false;
      visit(h, u, u, r, no, comp, prev, next, mcost, mark, cost, found);
      if (found) stop = false;
    }
    if (stop) {
      forn(u,n) if (prev[u] >= 0) cost += mcost[u];
      return cost;
    }
  }
}
```

8.17. Hungarian

```
//Dado un grafo bipartito completo con costos no negativos, encuentra el
     matching perfecto de minimo costo.
tipo cost[N][N], lx[N], ly[N], slack[N]; //llenar: cost=matriz de
     adyacencia
int n, max_match, xy[N], yx[N], slackx[N],prev2[N];//n=cantidad de nodos
bool S[N], T[N]; //sets S and T in algorithm
void add_to_tree(int x, int prevx) {
  S[x] = true, prev2[x] = prevx;
  forn(y, n) if (lx[x] + ly[y] - cost[x][y] < slack[y] - EPS)
    slack[y] = lx[x] + ly[y] - cost[x][y], slackx[y] = x;
}
void update_labels(){
  tipo delta = INF;
```

```
  forn (y, n) if (!T[y]) delta = min(delta, slack[y]);
  forn (x, n) if (S[x]) lx[x] -= delta;
  forn (y, n) if (T[y]) ly[y] += delta; else slack[y] -= delta;
}
void init_labels(){
  zero(lx), zero(ly);
  forn (x,n) forn(y,n) lx[x] = max(lx[x], cost[x][y]);
}
void augment() {
  if (max_match == n) return;
  int x, y, root, q[N], wr = 0, rd = 0;
  memset(S, false, sizeof(S)), memset(T, false, sizeof(T));
  memset(prev2, -1, sizeof(prev2));
  forn (x, n) if (xy[x] == -1){
    q[wr++] = root = x, prev2[x] = -2;
    S[x] = true; break; }
  forn (y, n) slack[y] = lx[root] + ly[y] - cost[root][y], slackx[y] =
      root;
  while (true){
    while (rd < wr){
      x = q[rd++];
      for (y = 0; y < n; y++) if (cost[x][y] == lx[x] + ly[y] && !T[y]){
        if (yx[y] == -1) break; T[y] = true;
        q[wr++] = yx[y], add_to_tree(yx[y], x); }
      if (y < n) break; }
    if (y < n) break;
    update_labels(), wr = rd = 0;
    for (y = 0; y < n; y++) if (!T[y] && slack[y] == 0){
      if (yx[y] == -1){x = slackx[y]; break;}
      else{
        T[y] = true;
        if (!S[yx[y]]) q[wr++] = yx[y], add_to_tree(yx[y], slackx[y]);
      }}
    if (y < n) break; }
  if (y < n){
    max_match++;
    for (int cx = x, cy = y, ty; cx != -2; cx = prev2[cx], cy = ty)
      ty = xy[cx], yx[cy] = cx, xy[cx] = cy;
    augment(); }
}
tipo hungarian(){
  tipo ret = 0; max_match = 0, memset(xy, -1, sizeof(xy));
  memset(yx, -1, sizeof(yx)), init_labels(), augment(); //steps 1-3
  forn (x,n) ret += cost[x][xy[x]]; return ret;
}
```

8.18. Dynamic Conectivity

Definición: permite realizar queries sobre un grafo dinámico al que se le pueden agregar y quitar aristas.
Explicación: procesa las queries (y los updates) offline, con una estrategia muy similar a la de la búsqueda binaria en paralelo: pensar que los arcos están presentes en cierto intervalo de tiempo, y que solo incluimos los arcos que contienen totalmente al intervalo que estamos considerando (a medida que se mueven los extremos). Al igual que en la búsqueda binaria en paralelo, se puede ver que se forma un árbol binario en el que se realiza una cantidad de operaciones lineal en cada nivel.

```
struct UF {
    int n, comp;
    vi par, size, c;
    UF(int n = 0): n(n), comp(n), par(n), size(n, 1) { iota(all(par), 0)
        ; }
    int find(int u) { return u == par[u] ? u : find(par[u]); }
    bool connected(int u, int v) { return find(u) == find(v); }
    bool merge(int u, int v) {
        if (connected(u, v)) return false;
        u = find(u), v = find(v);

        if (size[u] < size[v]) swap(u, v);
        size[u] += size[v], par[v] = u, comp--, c.pb(v);
        return true;
    }
    int snap() { return si(c); }
    void rollback(int snap){
        while (si(c) > snap) {
            int v = c.back(); c.pop_back();
            size[par[v]] -= size[v], par[v] = v, comp++;
        }
    }
};
enum { ADD, DEL, QUERY };
struct Query { int type, u, v; };
struct DynCon {
    vector<Query> q;
    UF uf;
    vi match; // match[i] = remove j asociado al add i (y viceversa)
    map<pii, int> last; // last[{u, v}] = i tal que add i agrega {u, v}
```

```
    vi res;
    DynCon(int n=0): uf(n) {}
    void add(int u, int v) {
        if (u > v) swap(u, v);
        q.pb((Query){ADD, u, v}), match.pb(-1), last[{u, v}] = si(q) -
            1;
    }
    void remove(int u, int v) {
        if (u > v) swap(u, v);
        q.pb((Query){DEL, u, v});
        int prev = last[{u, v}]; match[prev] = si(q) - 1; match.pb(prev)
            ;
    }
    void query() {
        q.pb((Query){QUERY, -1, -1}), match.pb(-1);
    }
    void process() { // answers all queries in order
        if (q.empty()) return;
        forn(i, si(q)) if (q[i].type == ADD && match[i] == -1) match[i]
            = si(q);
        go(0, si(q));
    }
    void go(int l, int r) { // divide intervalo al medio y procesa por
        partes, O(k log k)
        if (l+1 == r) {
            if (q[l].type == QUERY) // answer query using UF
                res.pb(uf.comp); // aqui query=cantidad de componentes
                    conexas
            return;
        }
        int m = (l+r) / 2;

        int s = uf.snap();
        dforsn(i, m, r) if (match[i] != -1 && match[i] < l) uf.merge(q[i
            ].u, q[i].v);
        go(l, m); uf.rollback(s);

        s = uf.snap();
        dforsn(i, l, m) if (match[i] != -1 && match[i] >= r) uf.merge(q[
            i].u, q[i].v);
        go(m, r); uf.rollback(s);
    }
};

// Primero agregar queries, adds y removes, luego llamar a process
```

9. Flujo

9.1. 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**: $|\text{minimum vertex cover}| = |\text{maximum matching}| \Leftrightarrow |\text{maximum independent set}| + |\text{maximum matching}| = |\text{vertices}|$.

9.2. Ford Fulkerson

Complejidad: $O(fE)$. **Algoritmo:** cambiar BFS por DFS en Edmonds Karp.

9.3. Edmonds Karp

Complejidad: $O(VE^2)$.

```
const int INF = 1e9;
template<class T>
struct EK {
    int n;
    vector<vi> adj;
    vector<vector<T>> capacity;
```

```

    EK(int _n) : n(_n) {
        adj = vector<vi>(n);
        capacity = vector<vector<T>>(n, vector<T>(n));
    }

    void addEdge(int u, int v, T c) {
        adj[u].pb(v), adj[v].pb(u);
        capacity[u][v] = c;
    }

    T bfs(int s, int t, vi &parent) {
        fill(all(parent), -1);
        parent[s] = s;
        queue<pair<int, T>> q;
        q.emplace(s, INF);
        while (!q.empty()) {
            int u = q.front().fst;
            T flow = q.front().snd;
            q.pop();
            for (int v : adj[u]) {
                if (parent[v] == -1 && capacity[u][v]) {
                    parent[v] = u;
                    T new_flow = min(flow, capacity[u][v]);
                    if (v == t) return new_flow;
                    q.emplace(v, new_flow);
                }
            }
        }
        return 0;
    }

    T maxflow(int s, int t) {
        T flow = 0, new_flow;
        vi parent(n);
        while ((new_flow = bfs(s, t, parent))) {
            flow += new_flow;
            int cur = t;
            while (cur != s) {
                int prev = parent[cur];
                capacity[prev][cur] -= new_flow;
                capacity[cur][prev] += new_flow;
                cur = prev;
            }
        }
        return flow;
    }
};
```

9.4. 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).

```
template<int MAXN>
struct dinic {
    struct edge {
        int u,v; ll c,f;
        ll r() { return c-f; }
    };

    static const ll INF = 1e18;

    int N,S,T;
    vector<edge> e;
    //edge red[MAXN][MAXN];
    vi adjG[MAXN];

    void reset() {
        forn(u,N) for (auto ind : adjG[u]) {
            auto &ei = e[ind];
            ei.f = 0;
        }
    }

    void initGraph(int n, int s, int t) {
        N = n; S = s; T = t;
        e.clear();
        forn(u,N) adjG[u].clear();
    }

    void addEdge(int u, int v, ll c) {
        adjG[u].pb(si(e)); e.pb((edge){u,v,c,0});
        adjG[v].pb(si(e)); e.pb((edge){v,u,0,0});
    }
```

```

    int dist[MAXN];
    bool dinic_bfs() {
        forn(u,N) dist[u] = -1;
        queue<int> q; q.push(S); dist[S] = 0;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            for (auto ind : adjG[u]) {
                auto &ei = e[ind];
                int v = ei.v;
                if (dist[v] != -1 || ei.r() == 0) continue;
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
        return dist[T] != -1;
    }

    ll dinic_dfs(int u, ll cap) {
        if (u == T) return cap;
        ll res = 0;
        for (auto ind : adjG[u]) {
            auto &ei = e[ind], &ej = e[ind^1];
            int v = ei.v;
            if (ei.r() && dist[v] == dist[u] + 1) {
                ll send = dinic_dfs(v,min(cap, ei.r()));
                ei.f += send; ej.f -= send;
                res += send; cap -= send;
                if (cap == 0) break;
            }
        }
        if (res == 0) dist[u] = -1;
        return res;
    }

    ll flow() {
        ll res = 0;
        while (dinic_bfs()) res += dinic_dfs(S,INF);
        return res;
    }

    vi cut() {
        dinic_bfs();
        vi ans;
        for (auto u : adjG[S]) if (dist[e[u].v] == -1) ans.pb(e[u].v);
        for (auto u : adjG[T]) if (dist[e[u].v] != -1) ans.pb(e[u].v);
        return ans;
    }

    vi indep() {
        dinic_bfs();
        vi ans;
        for (auto u : adjG[S]) if (dist[e[u].v] != -1) ans.pb(e[u].v);
        for (auto u : adjG[T]) if (dist[e[u].v] == -1) ans.pb(e[u].v);
        return ans;
    }
};
```

9.5. Maximum matching

Complejidad: $O(VE)$.

```
struct matching {
    // Indicate whether each node is on the left or call bipartition
    int n;
    vi match;
    vector<vi> g;
    vector<bool> vis, left;

    void addEdge(int u, int v) { g[u].pb(v), g[v].pb(u); }

    matching(int _n) { n = _n, match = vi(n, -1), g.resize(n), left.
        resize(n); }

    bool dfs(int u) {
        if (vis[u]) return false;
        vis[u] = true;
        for (int v : g[u])
            if (match[v] == -1 || dfs(match[v]))
                return match[v] = u, match[u] = v, true;
        return false;
    }

    int max_matching() { // O(N * M)
        int flow = 0;
        forn(i, n) if (left[i])
```

```
            vis.assign(n, 0), flow += dfs(i);
        return flow;
    }

    bool bipartition() {
        queue<int> q;
        vi dist(n, -1);
        forn(i, n) if (dist[i] == -1) {
            q.push(i), dist[i] = 0;
            while (!q.empty()) {
                int u = q.front(); q.pop();
                if (dist[u] & 1) left[u] = 1;
                for (int v : g[u]) {
                    if (dist[v] == -1)
                        dist[v] = dist[u] + 1, q.push(v);
                    else if ((dist[u] & 1) == (dist[v] & 1))
                        return false; // graph isn't bipartite
                }
            }
        }
        return true;
    }
};
```

9.6. 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;
    }
};
```

9.7. 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.

10. 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;
}
```

11. 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>
```

12. misc

```
#include <bits/stdc++.h> // Library that includes the most used libraries
using namespace std;  // It avoids the use of std::func(), instead we can simply use func()

ios_base::sync_with_stdio(0); cin.tie(0); // Speeds up considerably the read speed, very convenient when the input is large

#pragma GCC optimize ("O3") // Asks the compiler to apply more optimizations, that way speeding up the program very much!

Math:
max(a,b);  // Returns the largest of a and b
min(a,b);  // Returns the smallest of a and b
abs(a,b);  // Returns the absolute value of x (integral value)
fabs(a,b); // Returns the absolute value of x (double)
sqrt(x);   // Returns the square root of x.
pow(base,exp); // Returns base raised to the power exp
ceil(x); // Rounds x upward, returning the smallest integral value that is not less than x
floor(x); // Rounds x downward, returning the largest integral value that is not greater than x
exp(x); // Returns the base-e exponential function of x, which is e raised to the power x
log(x); // Returns the natural logarithm of x
log2(x); // Returns the binary (base-2) logarithm of x
log10(x); // Returns the common (base-10) logarithm of x
modf(double x, double *intpart); /* Breaks x into an integral and a fractional part. The integer part is stored in the object pointed by intpart, and the fractional part is returned by the function. Both parts have the same sign as x. */
sin(),cos(),tan(); asin(),acos(),atan(); sinh(),cosh(),tanh(); // Trigonometric functions
// See http://www.cplusplus.com/reference/cmath/ for more useful math functions!

Strings:
s.replace(pos,len,str); // Replaces the portion of the string that begins at character pos and spans len characters by str
s.replace(start,end,str); // or the part of the string in the range between [start,end)
s.substr(pos = 0,len = npos); // Returns the substring starting at character pos that spans len characters (or until the end of the string, whichever comes first).
// A value of string::npos indicates all characters until the end of the string.
s.insert(pos,str); // Inserts str right before the character indicated by pos
s.erase(pos = 0, len = npos); erase(first,last); erase(iterator p); // Erases part of the string
s.find(str,pos = 0); // Searches the string for the first occurrence of the sequence specified by its arguments after position pos
toupper(char x); // Converts lowercase letter to uppercase. If no such conversion is possible, the value returned is x unchanged.
tolower(char x); // Converts uppercase letter to lowercase. If no such conversion is possible, the value returned is x unchanged.

Constants:
INT_MAX, INT_MIN, LLONG_MIN, LLONG_MAX, ULLONG_MAX
const int maxn = 1e5; // 1e5 means 1x10^5, C++ features scientific notation. e.g.: 4.56e6 = 4.560.000, 7.67e-5 = 0.0000767.
const double pi = acos(-1); // Compute Pi

Algorithms:
swap(a,b); // Exchanges the values of a and b
minmax(a,b); // Returns a pair with the smallest of a and b as first element, and the largest as second.
minmax({1,2,3,4,5}); // Returns a pair with the smallest of all the elements in the list as first element and the largest as second
next_permutation(a,a+n); // Rearranges the elements in the range [first, last) into the next lexicographically greater permutation.
reverse(first,last); // Reverses the order of the elements in the range
```

```
[first,last)
rotate(first,middle,last) // Rotates the order of the elements in the range [first,last), in such a way that the element pointed by middle becomes the new first element
remove_if(first,last,func) // Returns an iterator to the element that follows the last element not removed. The range between first and this iterator includes all the elements in the sequence for which func does not return true.
// See http://www.cplusplus.com/reference/algorithm/ for more useful algorithms!

Binary search:
int a[] = {1, 2, 4, 7, 10, 12}, x = 5;
int *l = lower_bound(a,a+6,x); // lower_bound: Returns the first element that is not less than x
cout << (l == a+5 ? -1 : *l) << endl;
cout << x << (binary_search(a,a+6,x)?"␣is\n":"␣isn't\n"); // binary_search: Returns true if any element in the range [first,last) is equivalent to x, and false otherwise.
vi v(a,a+6);
auto i = upper_bound(v.begin(),v.end(),x) // upper_bound: Returns the first element that is greater than x

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); }
Unhackable seed (Codeforces):
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
random_shuffle(a,a+n,rng); // Rearranges the elements in the range [first,last) randomly

Sorting:
sort(a,a+n,comp); /* Sorts the elements in the range [first,last) into ascending order.
The third parameter is optional, if greater<Type> is passed then the array is sorted in descending order.
comp: Binary function that accepts two elements in the range as arguments, and returns a value convertible to bool. The value returned
indicates whether the element passed as first argument is considered to go before the second in the specific strict weak ordering
it defines. The function shall not modify any of its arguments. This can either be a function pointer or a function object. */
stable_sort(a,a+n); // Sorts the elements in the range [first,last) into ascending order, like sort, but stable_sort preserves the relative order of the elements with equivalent values.
sort(a.begin(),a.end()); // Sort using container ranges
sort(a,a+n,[](const node &a, const node &b){ // Custom sort with a "lambda expression": an unnamed function object capable of capturing variables in scope.
  return a.x < b.x || (a.x == b.x && a.y < b.y);   // Custom sort
}); // see https://en.cppreference.com/w/cpp/language/lambda for more details
bool myfunction(const edge &a, const edge &b){ return a.w < b.w; }
sort(myvector.begin()+4, myvector.end(), myfunction); // Using a function as a comparator
struct comp{ bool operator()(const edge &a, const edge &b){ return a.w < b.w; } };
multiset<edge,comp> l; // Using a function object as comparator:
bool operator<(const edge &a, const edge &b){ return a.w < b.w; } // Operator definition (it can be inside or outside the class)

Input/output handling:
freopen("input.txt","r",stdin); // Sets the standard input stream (keyboard) to the file input.txt
freopen("output.txt","w",stdout); // Sets the standard output stream (screen) to the file output.txt
getline(cin,str); // Reads until an end of line is reached from the input stream into str. If we use cin >> str it would read until it finds a whitespace
// Make an extra call if we previously read another thing from the input stream (otherwise it wouldn't work as expected)
cout << fixed << setprecision(n); // Sets the decimal precision to be used to format floating-point values on output operations to n
cout << setw(n); // Sets the field width to be used on output operations to n
cout << setfill('0'); // Sets c as the stream's fill character

Increment stack size to the maximum (Linux):
// #include <sys/resource.h>
struct rlimit rl;
getrlimit(RLIMIT_STACK, &rl);
rl.rlim_cur = rl.rlim_max;
setrlimit(RLIMIT_STACK, &rl);

String to int and vice versa (might be very useful to parse odd things):
```

```
template <typename T> string to_str(T str) { stringstream s; s << str; return s.str(); }
template <typename T> int to_int(T n) { int r; stringstream s; s << n; s >> r; return r; }
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

Print structs with cout:
ostream& operator << (ostream &o, pto &p) {
    o << p.x << ' ' << p.y;
    return o;
}
```

13. Ayudamemoria

Cant. decimales

```
#include <iomanip>
cout << setprecision(2) << fixed;
```

Rellenar con espacios(para justificar)

```
#include <iomanip>
cout << setfill(' ') << setw(3) << 2 << endl;
```

Comparación de Doubles

```
const double EPS = 1e-9;
x == y  <=> fabs(x-y) < EPS
x >  y  <=> x > y + EPS
x >= y  <=> x > y - EPS
```

Limites

```
#include <limits>
numeric_limits<T>
  ::max()
  ::min()
  ::epsilon()
```

Muahaha

```
#include <signal.h>
void divzero(int p){
  while(true);}
void segm(int p){
  exit(0);}
//in main
signal(SIGFPE, divzero);
signal(SIGSEGV, segm);
```

Mejorar velocidad 2

```
//Solo para enteros positivos
inline void Scanf(int& a){
  char c = 0;
  while(c<33) c = getc(stdin);
  a = 0;
  while(c>33) a = a*10 + c - '0', c = getc(stdin);
}
```

Leer del teclado

```
freopen("/dev/tty", "a", stdin);
```

Iterar subconjunto

```
for(int sbm=bm; sbm; sbm=(sbm-1)&bm)
```

File setup

```
// tambien se pueden usar comas: {a, x, m, l}
touch {a..l}.in; tee {a..l}.cpp < template.cpp
```

Releer String

```
string s; int n;
getline(cin, s);
stringstream leer(s);
while(leer >> n){
  // do something ...
}
```