Team notebook

Universidad Fracisco de Paula Santander

February 28, 2022

Contents

1 1 - Input Output

1.1 cin and cout

```
* Optimizar I/O:

ios::sync_with_stdio(0);
cin.tie(0);

* Impresion de punto flotante con d decimales. Ejemplo 6 decimales:

cout << fixed << setprecision(6) << value << '\n';
```

1.2 scanf and printf

```
* Lectura segun el tipo de dato (Se usan las mismas para imprimir):

scanf("%d", &value); //int
scanf("%ld", &value); //long y long int
scanf("%c", &value); //char
scanf("%f", &value); //float
scanf("%lf", &value); //double
scanf("%s", &value); //char*
scanf("%lld", &value); //long long int
scanf("%x", &value); //int hexadecimal
scanf("%o", &value); //int octal

* Impresion de punto flotante con d decimales, ejemplo 6 decimales:

printf("%.6lf", value);
```

2 2 - Data Structures

2.1 Centroid Decomposition

```
const int MX = 1e5+5;
vector<int> g[MX];
int par[MX], dep[MX], sz[MX];

int pre(int u, int p) {
    sz[u] = 1;
    for (auto &v : g[u])
        if (!dep[v] && v != p)
            sz[u] += pre(v, u);
    return sz[u];
}

int centroid(int u, int p, int k) {
    for (auto &v : g[u])
        if (!dep[v] && v != p && sz[v] > k)
            return centroid(v, u, k);
    return u;
}

void build(int u, int p = -1, int d = 1) {
    int k = pre(u, p);
    int c = centroid(u, p, k/2);
    par[c] = p; dep[c] = d;
    // dfs(c, p, ...);
    for (auto &v : g[c])
        if (!dep[v]) build(v, c, d+1);
}

int lca(int u, int v) {
    for (; u != v; u = par[u])
        if (dep[v] > dep[u]) swap(u, v);
    return u;
}
```

2.2 Convex Hull Trick

```
Agregar lineas en orden no-creciente por pendiente m. Permite consultar
    el minimo f(x). Para maximo, cambiar el signo de las lineas: -m, -h.

typedef ll T;
struct line { T m, h; };
struct cht {
    vector<line> c;
    int pos = 0;

    T inter(line a, line b){
        T x = b.h-a.h, y = a.m-b.m;
        return x/y + (x%y ? (x>0)==(y>0) : 0); // == ceil(x/y)
    }

    void add(T m, T h) {
        line l = {m, h};
        if (c.size() && m == c.back().m) {
            l.h = min(h, c.back().h);
            c.pop_back(); if (pos) pos--;
        }
        while (c.size() > 1 && inter(c.back(), l) <= inter(c[c.size()-2],
            c.back())) {
            c.pop_back(); if (pos) pos--;
        }
        c.pb(l);
    }

    inline bool fbin(T x, int m) { return inter(c[m], c[m+1]) > x; }
```

```
    T query(T x) {
        // O(log n) query:
        int l = 0, r = c.size();
        while (r-l > 1) {
            int m = (l+r)/2;
            if (fbin(x, m-1)) r = m;
            else l = m;
        }
        return c[l].m*x + c[l].h;
        // O(1) query (para x's ordenadas):
        while (pos > 0 && fbin(x, pos-1)) pos--;
        while (pos < (int)c.size()-1 && !fbin(x, pos)) pos++;
        return c[pos].m*x + c[pos].h;
    }
};
```

2.3 Disjoint Set

```
Estructura de datos para modelar una coleccion de conjuntos disyuntos.
Permite determinar de manera eficiente a que conjunto pertenece un
    elemento, si dos elementos se encuentran en un mismo conjunto y unir
    dos conjuntos en un uno.

struct dsu {
    vector<int> par, sz;
    int size; //Cantidad de conjuntos

    dsu(int n) : par(n), sz(n, 1) {
        size = n;
        iota(par.begin(), par.end(), 0);
    }
    //Busca el nodo representativo del conjunto de u
    int find(int u) {
        return par[u] == u ? u : (par[u] = find(par[u]));
    }
    //Une los conjuntos de u y v
    void unite(int u, int v) {
        u = find(u), v = find(v);
        if (u == v) return;
        if (sz[u] > sz[v]) swap(u,v);
        par[u] = v;
        sz[v] += sz[u];
        size--;
    }
    //Retorna la cantidad de elementos del conjunto de u
    int count(int u) { return sz[find(u)]; }
};
```

2.4 Fenwick Tree

```
Estructura de datos que permite procesar consultas por rangos y
    actualizaciones individuales sobre un arreglo.

const int MX = 1e5;
int bit[MX+1];

void add(int k, int val) {
    for (; k <= MX; k += k&-k) bit[k] += val;
}

int rsq(int k) {
    int sum = 0;
    for (; k >= 1; k -= k&-k) sum += bit[k];
    return sum;
}

int rsq(int i, int j) { return rsq(j) - rsq(i-1); }

int lower_find(int val) { /// last value < or <= to val
    int id = 0;
    for (int i = 31-__builtin_clz(MX); i >= 0; --i) {
        int nid = id | (1<<i);
        if (nid <= MX && bit[nid] <= val) { /// change <= to <
            val -= bit[nid];
            id = nid;
        }
    }
    return idx;
}
```

2.5 Heavy Light Decomposition

```
Para inicializar llamar build(). Agregar Segment Tree con un constructor
    vacio, actualizaciones puntuales y declarar el valor neutro de forma
```

```
    global.
Para consultas sobre aristas guardar el valor de cada arista en su nodo
    hijo y cambiar pos[u] por pos[u]+1 en la linea 54.

typedef int T; //tipo de dato del segtree
const int MX = 1e5+5;
vector<int> g[MX];
int par[MX], dep[MX], sz[MX];
int pos[MX], top[MX], value[MX];
vector<T> arr;
int idx;

int pre(int u, int p, int d) {
    par[u] = p; dep[u] = d;
    int aux = 1;
    for (auto &v : g[u]) {
        if (v != p) {
            aux += pre(v, u, d+1);
            if (sz[v] >= sz[g[u][0]]) swap(v, g[u][0]);
        }
    }
    return sz[u] = aux;
}

void hld(int u, int p, int t) {
    arr[idx] = value[u]; //vector para inicializar el segtree
    pos[u] = idx++;
    top[u] = t < 0 ? t = u : t;
    for (auto &v : g[u]) {
        if (v != p) {
            hld(v, u, t);
            t = -1;
        }
    }
}

segtree sgt;

void build(int n, int root) {
    idx = 0;
    arr.resize(n);
    pre(root, root, 0);
    hld(root, root, -1);
    sgt = segtree(arr);
}

T query(int u, int v) {
    T ans = neutro;
    while (top[u] != top[v]) {
        if (dep[top[u]] > dep[top[v]]) swap(u, v);
        ans = min(ans, sgt.query(pos[top[v]], pos[v]));
        v = par[top[v]];
    }
    if (dep[u] > dep[v]) swap(u, v);
    ans = min(ans, sgt.query(pos[u], pos[v]));
    return ans;
}

void upd(int u, T val) {
    sgt.upd(pos[u], val);
}
```

2.6 Min queue

```
Permite hallar el elemento minimo para todos los subarreglos de un largo
    fijo en O(n). Para Max queue cambiar el > por <.

struct mnque {
    deque<int> dq, mn;

    void push(int x) {
        dq.push_back(x);
        while (mn.size() && mn.back() > x) mn.pop_back();
        mn.push_back(x);
    }

    void pop() {
        if (dq.front() == mn.front()) mn.pop_front();
        dq.pop_front();
    }

    int min() { return mn.front(); }
};
```

2.7 Ordered Set

```
Estructura de datos basada en politicas. Funciona como un set<> pero es
    internamente indexado, cuenta con dos funciones adicionales.
.find_by_order(k) -> Retorna un iterador al k-esimo elemento, si k >=
    size() retona .end()
.order_of_key(x) -> Retorna cuantos elementos hay menores (<) que x

#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
   tree_order_statistics_node_update> ordered_set;
```

2.8 Sack (dsu on tree)

```
Tecnica basada en disjoint set, util para responer queries sobre arboles
    del tipo "cuantos vertices en el subarbol de u cumplen con alguna
    propiedad" en O(n log(n)) para todas las queries.

const int MX = 1e5+5;
vector<int> g[MX];
int sz[MX];
bool big[MX];
int cnt[MX];

void pre(int u, int p) {
   sz[u] = 1;
   for (auto &v : g[u]) {
      if (v != p) {
         pre(v, u);
         sz[u] += sz[v];
      }
   }
}

void upd(int u, int p, int x) {
   cnt[u] += x;
   for (auto &v : g[u])
      if (v != p && !big[v])
         upd(v, u, x);
}

void dfs(int u, int p, bool keep) {
   int mx = -1, id = -1;
   for (auto &v : g[u])
      if (v != p && sz[v] > mx)
         mx = sz[v], id = v;
   for (auto &v : g[u])
      if (v != p && v != id)
         dfs(v, u, false);
   if (id != -1) {
      dfs(id, u, true);
      big[id] = true;
   }
   upd(u, p, 1);
   /*Aqui se responden las queries. cnt[u] es el numero de
   vertices en el subarbol de u que cumplen la propiedad.*/
   if (id != -1)
      big[id] = false;
   if (!keep)
      upd(u, p, -1);
}
```

2.9 Segment Tree (Lazy Propagation)

```
Dado un vector de valores permite hacer consultas sobre rangos y
    actualizaciones individuales en O(log n). Construccion en O(n).
Para hacer actualizaciones sobre rangos se deben descomentar las lineas
    de Lazy Propagation.
El valor neutro depende del tipo de consulta. Para sumas: 0, minimos:
    infinito, maximos: -infinito, etc.

typedef int T; //tipo de dato del segtree
struct segtree {
   vector<T> st;//, lazy;
   int n; T neutro = 1e9; // "infinito"

   segtree(const vector<int> &v) {
      n = v.size();
      st.assign(n*4, 0);
      //lazy.assign(n*4, neutro);
      build(1, 0, n-1, v);
   }

   void build(int p, int L, int R, const vector<int> &v) {
      if (L == R) st[p] = v[L];
```

```
        else {
            int m = (L+R)/2, l = p*2, r = l+1;
            build(l, L, m, v);
            build(r, m+1, R, v);
            st[p] = min(st[l], st[r]);
        }
    }
    /*
    void propagate(int p, int L, int R, T val) {
        if (val == neutro) return;
        st[p] = val;
        lazy[p] = neutro;
        if (L == R) return;
        int l = p*2, r = l+1;
        lazy[l] = lazy[r] = val;
    }
    */
    T query(int i, int j) { return query(1, 0, n-1, i, j); }
    void upd(int i, int j, T val) { upd(1, 0, n-1, i, j, val); }

    T query(int p, int L, int R, int i, int j) {
        //propagate(p, L, R, lazy[p]);
        if (i > R || j < L) return neutro;
        if (i <= L && j >= R) return st[p];
        int m = (L+R)/2, l = p*2, r = l+1;
        T lf = query(l, L, m, i, j);
        T rg = query(r, m+1, R, i, j);
        return min(lf, rg);
    }

    void upd(int p, int L, int R, int i, int j, T val) {
        //propagate(p, L, R, lazy[p]);
        if (i > R || j < L) return;
        if (i <= L && j >= R) st[p] = val;//cambiar por propagate(p, L, R,
            val);
        else {
            int m = (L+R)/2, l = p*2, r = l+1;
            upd(l, L, m, i, j, val);
            upd(r, m+1, R, i, j, val);
            st[p] = min(st[l], st[r]);
        }
    }
};
```

2.10 Segment Tree 2D

```
struct segtree {
        int n, m;
        T neutro = {1, 0, 0, true};
        vector<vector<T>> st;

        segtree(int &n, int &m, vector<vector<T>> &a) : n(n), m(m){
                st = vector<vector<T>>(2*n, vector<T>(2*m, neutro));
                build(n, m, a);
        }

        T get(T a, T b){
                return max(a, b);
        }

        void build(int &n, int &m, vector<vector<T>> &a){
                for(int i = 0; i < n; i++)
                for(int j = 0; j < m; j++)
                st[i + n][j + m] = a[i][j];

                for(int i = 0; i < n; i++)
                for(int j = m - 1; j; j--)
                st[i + n][j] = get(st[i + n][j << 1], st[i + n][j << 1 |
                    1]);

                for(int i = n - 1; i; i--)
                for(int j = 0; j < 2*m; j++)
                st[i][j] = get(st[i << 1][j], st[i << 1 | 1][j]);
        }

        T query(int x1, int y1, int x2, int y2){
                T ans = neutro;
                vector<int> pos(2, 0);
                int node;
                for(x1 += n, x2 += n + 1; x1 < x2; x1 >>= 1, x2 >>= 1){ //
                    rows
                        node = 0;
                        if(x1&1) pos[node++] = x1++;
                        if(x2&1) pos[node++] = --x2;
                        for(int it = 0; it < node; it++){
                                for(int l = y1 + m, r = y2 + m + 1; l < r; l
                                    >>= 1, r >>= 1){ // cols
                                        if(l&1) ans = get(ans,
                                            st[pos[it]][l++]);
```

```
                        if(r&1) ans = get(ans,
                            st[pos[it]][--r]);
                    }
                }
            }
            return ans;
        }

        void upd(int l, int r, T val){
            st[l + n][r + m] = val;
            for(int j = r + m; j; j >>= 1)
                st[l][j >> 1] = get(st[l][j], st[l][j + 1]);

            for(int i = l + n; i; i >>= 1)
                for(int j = r + m; j; j >>= 1)
                    st[i >> 1][j] = get(st[i][j], st[i + 1][j]);
        }
};
```

2.11 Sparse Table

```
Estructura de datos que permite procesar consultas por rangos.

const int MX = 1e5+5;
const int LG = log2(MX)+1;
int spt[LG][MX];
int arr[MX];
int n;

void build() {
    for (int i = 0; i < n; i++) spt[0][i] = arr[i];
    for (int j = 0; j < LG-1; j++)
        for (int i = 0; i+(1<<(j+1)) <= n; i++)
            spt[j+1][i] = min(spt[j][i], spt[j][i+(1<<j)]);
}

int rmq(int i, int j) {
    int k = 31-__builtin_clz(j-i+1);
    return min(spt[k][i], spt[k][j-(1<<k)+1]);
}
```

2.12 Sparse table 2D

```
const int MAX_N = 100;
const int MAX_M = 100;
const int KN = log2(MAX_N)+1;
const int KM = log2(MAX_M)+1;
int table[KN][MAX_N][KM][MAX_M];
int _log2N[MAX_N+1];
int _log2M[MAX_M+1];

int MAT[MAX_N][MAX_M];
int n, m, ic, ir, jc, jr;

void calc_log2() {
    _log2N[1] = 0;
    _log2M[1] = 0;
    for (int i = 2; i <= MAX_N; i++) _log2N[i] = _log2N[i/2] + 1;
    for (int i = 2; i <= MAX_M; i++) _log2M[i] = _log2M[i/2] + 1;
}

void build() {
    for(ir = 0; ir < n; ir++){
        for(ic = 0; ic < m; ic++)
            table[ 0 ][ ir ][ 0 ][ ic ] = MAT[ ir ][ ic ];

        for(jc = 1; jc < KM; jc++)
            for(ic = 0; ic + (1 <<(jc-1)) < m; ic++)
                table[0 ][ir ][jc ][ic ] = min(table[0 ][ir ][jc-1 ][ic
                    ],table[0 ][ir ][ jc-1 ][ ic + (1 << (jc-1)) ]);

    }

    for(jr = 1; jr < KN; jr++)
        for(ir = 0; ir < n; ir++)
            for(jc = 0; jc < KM; jc++)
                for(ic = 0; ic < m; ic++)
                    table[jr ][ir ][jc ][ic ] = min(table[jr-1 ][ir ][jc
                        ][ic ],table[jr-1 ][ir+(1<<(jr-1)) ][jc ][ic ]);
}

int rmq(int x1, int y1, int x2, int y2) {
    int lenx = x2-x1+1;
    int kx = _log2N[lenx];
    int leny = y2-y1+1;
    int ky = _log2M[leny];
```

```
    int min_R1 = min ( table[kx ][x1 ][ky ][y1 ] , table[kx ][x1 ][ky ][
        y2 + 1 - (1<<ky) ] );
    int min_R2 = min ( table[kx ][x2+1-(1<<kx) ][ky ][y1 ], table[kx
        ][x2+1- (1<<kx) ][ky ][y2 + 1 - (1<<ky)] );
    return min ( min_R1, min_R2 );
}
```

3 3 - Dynamic Programming

3.1 Coin Change

```
Problemas clsicos de moneda con DP

const int MAX_COINS = 1005;
const int MAX_VALUE = 1005;
const int INF = (int) (2e9);
int coins[MAX_COINS];
int dp[MAX_VALUE];
vector<int> rb;

//Calcula el nmero de formas para valores entre 1 y value. SIN CONTAR
    PERMUTACIONES
void ways(int value){
        memset(dp, 0, sizeof(dp));
        dp[0] = 1;
        for(auto c: coins){
                for(int i = 1; i <= value; i++){
                        if(i - c >= 0) dp[i] += dp[i - c];
                }
        }
}

//Calcula el nmero de formas para valores entre 1 y value. CONTANDO
    PERMUTACIONES
void ways_perm(int value){
        memset(dp, 0, sizeof(dp));
        dp[0] = 1;
        for(int i = 1; i <= value; i++){
                for(auto c: coins){
                        if(i - c >= 0) dp[i] += dp[i - c];
                }
        }
}

//Calcula el minimo numero de monedas necesarias para los valores entre 1
    y value.
void min_coin(int value){
        memset(dp, 0, sizeof(dp));
        for(int i = 1; i <= value; i++){
                dp[i] = INF;
                for(auto c: coins){
                        if(i - c >= 0) dp[i] = min(dp[i], dp[i - c] + 1);
                }
        }
}

//Guarda en el vector rb las monedas usadas en min_coin.
void build_ways(int value){
        rb.clear();
        for(int c = MAX_COINS - 1; c >= 0; c--){
                if(value == 0) return;
                while(value - coins[c] >= 0 && dp[value] == dp[value -
                    coins[c]] + 1){
                        rb.push_back(coins[c]);
                        value -= coins[c];
                }
        }
}
```

3.2 Directed Acyclic Graph

```
Problemas clasicos con DAG

const int INF = 1e9;
const int MAX = 1000;
int init, fin;
int dp[MAX];
vector<int> g[MAX]; //USADO PARA ARISTAS NO PONDERADAS
vector<pair<int, int>> gw[MAX]; //PARA ARISTAS PONDERADAS First: Nodo
    vecino. Second = Peso de la arista

//Funcion para calcular el numero de formas de ir del nodo u al nodo end
//LLamar para nodo inicial (init)
int ways(int u){
```

```
	if(u == fin) return 1;
	int &ans = dp[u];
	if(ans != -1) return ans;
	ans = 0;
	for(auto v: g[u]){
		ans += ways(v);
	}
	return ans;
}

//MINIMO CAMINO DESDE U HASTA END. LLAMAR PARA INIT
int min_way(int u){
	if(u == fin) return 0;
	int &ans = dp[u];
	if(ans != -1) return ans;
	ans = INF;
	for(auto v: gw[u]){
		ans = min(ans, min_way(v.first) + v.second);
	}
	return ans;
}
```

3.3 Knapsack

```
const int MAX_N = 1000;
const int MAX_W = 10000;
const int INF = (int) (2e9);

int dp[MAX_N + 5][MAX_W + 5];
int gold[MAX_N];
int weight[MAX_N];
int N;
vector<int> rb;

//mochila TOP_DOWN. NECESITA INICIALIZARSE ANTES DP EN -1
int f(int i, int w){
	if(w < 0) return -INF;
	if(i == N) return 0;
	int &ans = dp[i][w];
	if(ans != -1) return ans;
	ans = max(f(i + 1, w), f(i + 1, w - weight[i]) + gold[i]);
	return ans;
}

//BOTTOM_UP MOCHILA. ACCEDER COMO dp[0][W]
void mochila(){
	for(int i = 0; i <= MAX_W; i++) dp[N][i] = 0;
	for(int i = N - 1; i >= 0; i--){
		for(int w = 0; w <= MAX_W; w++){
			dp[i][w] = dp[i + 1][w];
			if(w - weight[i] >= 0) dp[i][w] = max(dp[i][w],
			    dp[i + 1][w - weight[i]] + gold[i]);
		}
	}
}

//MOCHILA OPTIMIZANDO MEMORIA. ACCEDER COMO dp_opt[0][W]
int dp_opt[2][MAX_W + 5];
void mochila_opt(){
	for(int i = 0; i <= MAX_W; i++) dp[N%2][i] = 0;
	for(int i = N - 1; i >= 0; i--){
		for(int w = 0; w <= MAX_W; w++){
			dp_opt[i%2][w] = dp_opt[(i + 1)%2][w];
			if(w - weight[i] >= 0) dp_opt[i%2][w] =
			    max(dp_opt[i%2][w], dp_opt[(i + 1)%2][w -
			    weight[i]] + gold[i]);
		}
	}
}

//RECONSTRUIR SOLUCION. GUARDA LOS INDICES DE LOS ELEMENTOS USADOS. NO
    FUNCIONA CON MOCHILA OPTIMIZADA.
//ADVERTENCIA: Si existen multiples soluciones reconstruye la que primero
    aparezca. Para la ultima recorrer al revs
void build(int W){
	rb.clear();
	for(int i = 0; i < N && W > 0; i++){
		if(W - weight[i] >= 0 && dp[i][W] == dp[i + 1][W -
		    weight[i]] + gold[i])
			rb.push_back(i);
	}
}
```

3.4 Longest Common Subsequence

```
Dados dos Strings, encuentra el largo de la subsecuencia en comn mas
    larga entre ellas.

const int M_MAX = 20; // Mximo size del String 1
const int N_MAX = 20; // Mximo size del String 2
int m, n; // Size de Strings 1 y 2
string X; // String 1
string Y; // String 2
int memo[M_MAX + 1][N_MAX + 1];

int lcs (int m, int n) {
  for (int i = 0; i <= m; i++) {
    for (int j = 0; j <= n; j++) {
      if (i == 0 || j == 0) memo[i][j] = 0;
      else if (X[i - 1] == Y[j - 1]) memo[i][j] = memo[i - 1][j - 1] + 1;
      else memo[i][j] = max(memo[i - 1][j], memo[i][j - 1]);
    }
  }
  return memo[m][n];
}
```

3.5 Longest Increasing Subsequence

```
//METODO PARA CALCULAR EL LIS en O(n^2) y O(nlog(n)). La ventaja de tener
    a mano O(n^2) es porque es mas facil de codear, entender y modificar

const int MAX = 1e5+1;

int A[MAX];
int dp[MAX];
int N = MAX;
vector<int> LIS; //PARA Lis_opt


//LIS O(n^2).Si es non-decreasing cambiar (i > j) por (i >= j)
int lis(){
	int best = -1;
	for(int i = 0; i < N; i++){
		dp[i] = 1;
		for(int j = 0; j < i; j++){
			if(A[i] > A[j]) dp[i] = max(dp[i], dp[j] + 1);

		}
		best = max(best, dp[i]);
	}
	return best;
}


//LIS O(nlog(n)) Para longest non-decreasing cambiar lower_bound por
    upper_bound
int lis_opt(){
	LIS.clear();
	for(int i = 0; i < N; i++){
		auto id = lower_bound(LIS.begin(), LIS.end(), A[i]);
		if(id == LIS.end()){
			LIS.push_back(A[i]);
			dp[i] = LIS.size();
		}
		else{
			int idx = id - LIS.begin();
			LIS[idx] = A[i];
			dp[i] = idx + 1;
		}
	}
	return LIS.size();
}

//METODO PARA RECONSTRUIR LIS. Para non-decreasing cambiar < por <=
stack<int> rb;
void build(){
	int k = LIS.size();
	int cur = oo;
	for(int i = N - 1; i >= 0, k; i--){
		if(A[i] < cur && k == dp[i]){
			cur = A[i];
			rb.push(A[i]);
			k--;
		}
	}
}
```

3.6 Max Range 2D

```
//Cambiar infinito por el mnimo valor posible
int INF = -100000007;
```

```
int n, m; //filas y columnas
const int MAX_N = 105, MAX_M = 105;
int values[MAX_N][MAX_M];

int max_range_sum2D(){
	for(int i=0; i<n;i++){
		for(int j=0; j<m; j++){
			if(i>0) values[i][j] += values[i-1][j];
			if(j>0) values[i][j] += values[i][j-1];
			if(i>0 && j>0) values[i][j] -= values[i-1][j-1];
		}
	}
	int max_mat = INF;
	for(int i=0; i<n;i++){
		for(int j=0; j<m; j++){
			for(int h = i; h<n; h++){
				for(int k = j; k<m; k++){
					int sub_mat = values[h][k];
					if(i>0) sub_mat -= values[i-1][k];
					if(j>0) sub_mat -= values[h][j-1];
					if(i>0 && j>0) sub_mat +=
					    values[i-1][j-1];
					max_mat = max(sub_mat, max_mat);
				}
			}
		}
	}
	return max_mat;
}
```

3.7 Max Range 3D

```
//Cambir valores a, b, c por lmites correspondientes
long long a=20, b=20, c=20;
long long acum[a][b][c];
long long INF = -100000000007;

long long max_range_3D(){
	for(int x=0; x<a; x++){
		for(int y = 0; y<b; y++){
			for(int z = 0; z<c; z++){
				if(x>0) acum[x][y][z] += acum[x-1][y][z];
				if(y>0) acum[x][y][z] += acum[x][y-1][z];
				if(z>0) acum[x][y][z] += acum[x][y][z-1];
				if(x>0 && y>0) acum[x][y][z] -=
				    acum[x-1][y-1][z];
				if(x>0 && z>0) acum[x][y][z] -=
				    acum[x-1][y][z-1];
				if(y>0 && z>0) acum[x][y][z] -=
				    acum[x][y-1][z-1];
				if(x>0 && y>0 && z>0) acum[x][y][z] +=
				    acum[x-1][y-1][z-1];
			}
		}
	}
	long long max_value = INF;
	for(int x=0; x<a; x++){
		for(int y = 0; y<b; y++){
			for(int z = 0; z<c; z++){
				for(int h = x; h<a; h++){
					for(int k = y; k<b; k++){
						for(int l = z; l<c; l++){
							long long aux =
							    acum[h][k][l];
							if(x>0) aux -=
							    acum[x-1][k][l];
							if(y>0) aux -=
							    acum[h][y-1][l];
							if(z>0) aux -=
							    acum[x][k][z-1];
							if(x>0 && y>0) aux +=
							    acum[x-1][y-1][l];
							if(x>0 && z>0) aux +=
							    acum[x-1][k][z-1];
							if(z>0 && y>0) aux +=
							    acum[h][y-1][z-1];
							if(x>0 && y>0 && z>0)
							    aux -=
							    acum[x-1][y-1][z-1];
							max_value =
							    max(max_value,
							    aux);
						}
					}
				}
			}
		}
	}
}
```

```
	return max_value;
}
```

3.8 Max Range Sum

```
Dada una lista de enteros, retorna la mxima suma de un rango de la lista.

#include <algorithm>

int maxRangeSum(vector<int> a){
	int sum = 0, ans = 0;
	for (int i = 0; i < a.size(); i++){
		if (sum + a[i] >= 0) {
			sum += a[i];
		   ans = max(ans, sum);
	   } else sum = 0;
	}
	return ans;
}
```

3.9 Min Max Range

```
Devuelve el el minimo de los maximos entre pares de rangos consecutivos
    haciendo cortes en el Array.

const int MAX = 1005;
long long dp[MAX][MAX];
long long sum_ran[MAX][MAX];
int N;

long long f(int i, int cuts){
	if(cuts == 0) return sum_ran[i][N-1];
	if(i == N) return 0;
	long long &ans = dp[i][cuts];
	if(ans != - 1) return ans;
	for(int j = i; j < N; j++){
		ans = min(ans, max(sum_ran[i][j], f(i + 1, cuts - 1)));
	}
}
```

3.10 Traveling Salesman Problem

```
Problema del viajero. Devuelve la ruta minima haciendo un tour visitando
    todas los nodos (ciudades) una unica vez.

const int MAX = 18;
int target; // Inicializarlo para (1<<N) - 1
int dist[MAX][MAX]; //Distancia entre cada par de nodos
int N;
int dp[(1<<MAX) + 2][MAX];
vector<int> rb;
const int INF = (int) (2e9);


//Llamar para TSP(0, -1) Si no empieza de ninguna ciudad especificia
//De lo contrario llamar TSP(0, 0)
int TSP(int mask, int u){
	if(mask == target){
		return 0;
		//0 en su defecto el costo extra tras haber visitado todas
		    las ciudades. EJ: Volver a la ciudad principal
	}
	if(u == -1){
		int ans = INF;
		for(int i = 0; i < N; i++){
			ans = min(ans, TSP(mask | (1<<i), i));
			//Agregar costo Extra desde el punto de partida si
			    es necesario
		}
		return ans;

	}
	int &ans = dp[mask][u];
	if(ans != -1) return ans;
	ans = INF;
	for(int i = 0; i < N; i++){
		if(!(mask & (1<<i)))
			ans = min(ans, TSP(mask | (1<<i), i) + dist[u][i]);
	}
	return ans;
}

void build(int mask, int u){
	if(mask == target) return; //Acaba el recorrido
	if(u == -1){
```

```
            for(int i = 0; i < N; i++){
                if(TSP(mask, u) == TSP(mask | (1<<i), i)){
                    rb.push_back(i);
                    build(mask | (1<<i), i);
                    return;
                }
            }
        }else{
            for(int i = 0; i < N; i++){
                if(!(mask & (1<<i))){
                    if(TSP(mask, u) == TSP(mask | (1<<i), i) +
                        dist[u][i]){
                        rb.push_back(i);
                        build(mask | (1<<i), i);
                        return;
                    }
                }
            }
        }
}
```

4 4 - Geometry

4.1 Angle

```
Calcula el angulo de una linea con respecto a otra.

lf get_ang(pt a, pt b) {
    lf ang = acos(max(lf(-1.0), min(lf(1.0),
        lf(dot(a,b))/abs(a)/abs(b))));
    ang = ang * 180.0 / acos(-1.0);
    if (b.y < 0) ang = lf(360) - ang;
    return ang;
}

lf angle(pt a, pt b) {
    pt xo = {1, 0};
    lf ang = get_ang(xo, b) - get_ang(xo, a);
    if (ang < 0) ang += 360;
    return ang;
}

double DegToRad(double d) {
        return d * acos(-1.0) / 180.0;
}

double RadToDeg(double r) {
        return r * 180.0 / acos(-1.0);
}
```

4.2 Point

```
typedef double lf;
const lf eps = 1e-9;
typedef double T;
struct pt {
    T x, y;
    pt operator + (pt p) { return {x+p.x, y+p.y}; }
    pt operator - (pt p) { return {x-p.x, y-p.y}; }
    pt operator * (pt p) { return {x*p.x-y*p.y, x*p.y+y*p.x}; }
    pt operator * (T d) { return {x*d, y*d}; }
    pt operator / (lf d) { return {x/d, y/d}; } /// only for floating
        point
    bool operator == (pt b) { return x == b.x && y == b.y; }
    bool operator != (pt b) { return !(*this == b); }
    bool operator < (const pt &o) const { return y < o.y || (y == o.y &&
        x < o.x); }
    bool operator > (const pt &o) const { return y > o.y || (y == o.y &&
        x > o.x); }
};
int cmp (lf a, lf b) { return (a + eps < b ? -1 :(b + eps < a ? 1 : 0));
    } //double comparator

T norm(pt a) { return a.x*a.x + a.y*a.y; }
lf abs(pt a) { return sqrt(norm(a)); }
lf arg(pt a) { return atan2(a.y, a.x); }
pt unit(pt a) { return a/abs(a); }

T dot(pt a, pt b) { return a.x*b.x + a.y*b.y; } // x = 90 -> cos = 0
T cross(pt a, pt b) { return a.x*b.y - a.y*b.x; } // x = 180 -> sin = 0
T orient(pt a, pt b, pt c) { return cross(b-a,c-a); }// clockwise = -
pt rot(pt p, lf a) { return {p.x*cos(a) - p.y*sin(a), p.x*sin(a) +
    p.y*cos(a)}; }
pt rotate_to_b(pt a, pt b, lf ang) { return rot(a-b, ang)+b; } // rotate
    by ang center b
```

```
pt rot90ccw(pt p) { return {-p.y, p.x}; }
pt rot90cw(pt p) { return {p.y, -p.x}; }
pt translate(pt p, pt v) { return p+v; }
pt scale(pt p, double f, pt c) { return c + (p-c)*f; } // c-center
bool are_perp(pt v, pt w) { return dot(v,w) == 0; }
int sign(T x) { return (T(0) < x) - (x < T(0)); }

bool in_angle(pt a, pt b, pt c, pt x) { // x inside angle abc (center in
     a)
    assert(orient(a,b,c) != 0);
    if (orient(a,b,c) < 0) swap(b,c);
    return orient(a,b,x) >= 0 && orient(a,c,x) <= 0;
}
//angle bwn 2 vectors
lf angle(pt a, pt b) { return acos(max(-1.0, min(1.0,
     dot(a,b)/abs(a)/abs(b)))); }
lf angle(pt a, pt b) { return atan2(cross(a, b), dot(a, b)); }
/// returns vector to transform points
pt get_linear_transformation(pt p, pt q, pt r, pt fp, pt fq) {
    pt pq = q-p, num{cross(pq, fq-fp), dot(pq, fq-fp)};
    return fp + pt{cross(r-p, num), dot(r-p, num)} / norm(pq);
}
bool half(pt p) { /// true if is in (0, 180] (line is x axis)
    assert(p.x != 0 || p.y != 0); /// the argument of (0,0) is undefined
    return p.y > 0 || (p.y == 0 && p.x < 0);
}
bool half_from(pt p, pt v = {1, 0}) { //line is v (above v is true)
    return cross(v,p) < 0 || (cross(v,p) == 0 && dot(v,p) < 0);
}
bool polar_cmp(const pt &a, const pt &b) {//polar sort
    return make_tuple(half(a), 0) < make_tuple(half(b), cross(a,b));
//  return make_tuple(half(a), 0, sq(a)) < make_tuple(half(b), cross(a,
     b), sq(b)); // further ones appear later
}
```

4.3 circle

```
struct circle {
    pt c; T r;
};
// (x-xo)^2 + (y-yo)^2 = r^2
//circle that passes through abc
circle center(pt a, pt b, pt c) {
    b = b-a, c = c-a;
    assert(cross(b,c) != 0); /// no circumcircle if A,B,C aligned
    pt cen = a + rot90ccw(b*norm(c) - c*norm(b))/cross(b,c)/2;
    return {cen, abs(a-cen)};
}
//centers of the circles that pass through ab and has radius r
vector<pt> centers(pt a, pt b, T r) {
    if (abs(a-b) > 2*r + eps) return {};
    pt m = (a+b)/2;
    double f = sqrt(r*r/norm(a-m) - 1);
    pt c = rot90ccw(a-m)*f;
    return {m-c, m+c};
}
int inter_cl(circle c, line l, pair<pt, pt> &out) {
    lf h2 = c.r*c.r - l.sq_dist(c.c);
    if(h2 >= 0) { // line touches circle
        pt p = l.proj(c.c);
        pt h = l.v*sqrt(h2)/abs(l.v); // vector of len h parallel to line
        out = {p-h, p+h};
    }
    return 1 + sign(h2); // if 1 -> out.F == out.S
}
int inter_cc(circle c1, circle c2, pair<pt, pt> &out) {
    pt d = c2.c-c1.c;
    double d2 = norm(d);
    if(d2 == 0) { assert(c1.r != c2.r); return 0; } // concentric circles
         (identical)
    double pd = (d2 + c1.r*c1.r - c2.r*c2.r)/2; // = |O_1P| * d
    double h2 = c1.r*c1.r - pd*pd/d2; // = h^2
    if(h2 >= 0) {
        pt p = c1.c + d*pd/d2, h = rot90ccw(d)*sqrt(h2/d2);
        out = {p-h, p+h};
    }
    return 1 + sign(h2);
}
//circle-line inter = 1
int tangents(circle c1, circle c2, bool inner, vector<pair<pt,pt>> &out) {
    if(inner) c2.r = -c2.r; // inner tangent
    pt d = c2.c-c1.c;
    double dr = c1.r-c2.r, d2 = norm(d), h2 = d2-dr*dr;
    if(d2 == 0 || h2 < 0) { assert(h2 != 0); return 0; } // (identical)
    for(double s : {-1,1}) {
        pt v = (d*dr + rot90ccw(d)*sqrt(h2)*s)/d2;
        out.push_back({c1.c + v*c1.r, c2.c + v*c2.r});
    }
```

```
    return 1 + (h2 > 0); // if 1: circle are tangent
}
//circle targent passing through pt p
int tangent_through_pt(pt p, circle c, pair<pt, pt> &out) {
    double d = abs(p - c.c);
    if(d < c.r) return 0;
    pt base = c.c-p;
    double w = sqrt(norm(base) - c.r*c.r);
    pt a = {w, c.r}, b = {w, -c.r};
    pt s = p + base*a/norm(base)*w;
    pt t = p + base*b/norm(base)*w;
    out = {s, t};
    return 1 + (abs(c.c-p) == c.r);
}
```

4.4 halfplanes

```
struct halfplane{
    double angle;
    pt p, pq;
    halfplane(){}
    halfplane(pt a, pt b): p(a), pq(b - a) {
        angle = atan2(pq.y,pq.x);
    }
    bool operator < (halfplane b)const{return angle < b.angle;}
    bool out(pt q){return cross(pq, (q-p)) < -eps;} // checks if p is
        inside the half plane
};

const lf inf = 1e100;
// intersection pt of the lines of 2 halfplanes
pt inter(halfplane& h1, halfplane& h2){
    if(abs(cross(unit(h1.pq), unit(h2.pq))) <= eps)return {inf, inf};
    lf alpha = cross((h2.p - h1.p), h2.pq) / cross(h1.pq, h2.pq);
    return h1.p + (h1.pq * alpha);
}

// intersection of halfplanes
vector<pt> intersect(vector<halfplane>& b){
    vector<pt> box = { {inf, inf}, {-inf, inf}, {-inf, -inf}, {inf, -inf}
        };
    for(int i = 0; i < 4; i++){
        b.push_back({box[i], box[(i + 1) % 4]});
    }
    sort(b.begin(), b.end());
    int n = b.size(), q = 1, h = 0;
    vector<halfplane> c(n + 10);
    for(int i = 0; i < n; i++){
        while(q < h && b[i].out(inter(c[h], c[h-1]))) h--;
        while(q < h && b[i].out(inter(c[q], c[q+1]))) q++;
        c[++h] = b[i];
        if(q < h && abs(cross(c[h].pq, c[h-1].pq)) < eps){
            if(dot(c[h].pq, c[h-1].pq) <= 0) return {};
            h--;
            if(b[i].out(c[h].p)) c[h] = b[i];
        }
    }
    while(q < h-1 && c[q].out(inter(c[h], c[h-1]))) h--;
    while(q < h-1 && c[h].out(inter(c[q], c[q+1]))) q++;
    if(h - q <= 1)return {};
    c[h+1] = c[q];
    vector<pt> s;
    for(int i = q; i < h+1; i++) s.pb(inter(c[i], c[i+1]));
    return s;
}
```

4.5 line

```
struct line {
    pt v; T c; // v:direction c: pos in y axis
    line(pt v, T c) : v(v), c(c) {}
    line(T a, T b, T c) : v({b,-a}), c(c) {} // ax + by = c
    line(pt p, pt q) : v(q-p), c(cross(v,p)) {}
    T side(pt p) { return cross(v,p)-c; }
    lf dist(pt p) { return abs(side(p)) / abs(v); }
    lf sq_dist(pt p) { return side(p)*side(p) / (lf)norm(v); }
    line perp_through(pt p) { return {p, p + rot90ccw(v)}; } // line perp
        to v passing through p
    bool cmp_proj(pt p, pt q) { return dot(v,p) < dot(v,q); } // order
        for points over the line
    line translate(pt t) { return {v, c + cross(v,t)}; }
    line shift_left(double d) { return {v, c + d*abs(v)}; }
    pt proj(pt p) { return p - rot90ccw(v)*side(p)/norm(v); } // pt
        proyected on the line
    pt refl(pt p) { return p - rot90ccw(v)*2*side(p)/norm(v); } // pt
        reflected on the other side of the line
```

```
};

bool inter_ll(line l1, line l2, pt &out) {
    T d = cross(l1.v, l2.v);
    if (d == 0) return false;
    out = (l2.v*l1.c - l1.v*l2.c) / d; // floating points
    return true;
}
//bisector divides the angle in 2 equal angles
//interior line goes on the same direction as l1 and l2
line bisector(line l1, line l2, bool interior) {
    assert(cross(l1.v, l2.v) != 0); /// l1 and l2 cannot be parallel!
    lf sign = interior ? 1 : -1;
    return {l2.v/abs(l2.v) + l1.v/abs(l1.v) * sign,
            l2.c/abs(l2.v) + l1.c/abs(l1.v) * sign};
}
```

4.6 polygon

```
enum {IN, OUT, ON};
struct polygon {
    vector<pt> p;
    polygon(int n) : p(n) {}
    int top = -1, bottom = -1;
    void delete_repetead() {
        vector<pt> aux;
        sort(p.begin(), p.end());
        for(pt &i : p)
            if(aux.empty() || aux.back() != i)
              aux.push_back(i);
        p.swap(aux);
    }
    bool is_convex() {
        bool pos = 0, neg = 0;
        for (int i = 0, n = p.size(); i < n; i++) {
            int o = orient(p[i], p[(i+1)%n], p[(i+2)%n]);
            if (o > 0) pos = 1;
            if (o < 0) neg = 1;
        }
        return !(pos && neg);
    }
    lf area(bool s = false) { // better on clockwise order
        lf ans = 0;
        for (int i = 0, n = p.size(); i < n; i++)
            ans += cross(p[i], p[(i+1)%n]);
        ans /= 2;
        return s ? ans : abs(ans);
    }
    lf perimeter() {
        lf per = 0;
        for(int i = 0, n = p.size(); i < n; i++)
           per += abs(p[i] - p[(i+1)%n]);
        return per;
    }
    bool above(pt a, pt p) { return p.y >= a.y; }
    bool crosses_ray(pt a, pt p, pt q) { // pq crosses ray from a
        return (above(a,q)-above(a,p))*orient(a,p,q) > 0;
    }
    int in_polygon(pt a) {
        int crosses = 0;
        for(int i = 0, n = p.size(); i < n; i++) {
            if(on_segment(p[i], p[(i+1)%n], a)) return ON;
            crosses += crosses_ray(a, p[i], p[(i+1)%n]);
        }
        return (crosses&1 ? IN : OUT);
    }
    void normalize() { /// polygon is CCW
        bottom = min_element(p.begin(), p.end()) - p.begin();
        vector<pt> tmp(p.begin()+bottom, p.end());
        tmp.insert(tmp.end(), p.begin(), p.begin()+bottom);
        p.swap(tmp);
        bottom = 0;
        top = max_element(p.begin(), p.end()) - p.begin();
    }
    int in_convex(pt a) {
        assert(bottom == 0 && top != -1);
        if(a < p[0] || a > p[top]) return OUT;
        T orientation = orient(p[0], p[top], a);
        if(orientation == 0) {
            if(a == p[0] || a == p[top]) return ON;
            return top == 1 || top + 1 == p.size() ? ON : IN;
        } else if (orientation < 0) {
            auto it = lower_bound(p.begin()+1, p.begin()+top, a);
            T d = orient(*prev(it), a, *it);
            return d < 0 ? IN : (d > 0 ? OUT: ON);
        } else {
            auto it = upper_bound(p.rbegin(), p.rend()-top-1, a);
            T d = orient(*it, a, it == p.rbegin() ? p[0] : *prev(it));
```

```
        return d < 0 ? IN : (d > 0 ? OUT: ON);
    }
}
polygon cut(pt a, pt b) { // cuts polygon on line ab
    line l(a, b);
    polygon new_polygon(0);
    for(int i = 0, n = p.size(); i < n; ++i) {
        pt c = p[i], d = p[(i+1)%n];
        lf abc = cross(b-a, c-a), abd = cross(b-a, d-a);
        if(abc >= 0) new_polygon.p.push_back(c);
        if(abc*abd < 0) {
          pt out; inter_ll(l, line(c, d), out);
          new_polygon.p.push_back(out);
        }
    }
    return new_polygon;
}
void convex_hull() {
    sort(p.begin(), p.end());
    vector<pt> ch;
    ch.reserve(p.size()+1);
    for(int it = 0; it < 2; it++) {
        int start = ch.size();
        for(auto &a : p) {
            /// if colineal are needed, use < and remove repeated
                points
            while(ch.size() >= start+2 && orient(ch[ch.size()-2],
                ch.back(), a) <= 0)
                ch.pop_back();
            ch.push_back(a);
        }
        ch.pop_back();
        reverse(p.begin(), p.end());
    }
    if(ch.size() == 2 && ch[0] == ch[1]) ch.pop_back();
    /// be careful with CH of size < 3
    p.swap(ch);
}
vector<pii> antipodal() {
    vector<pii> ans;
    int n = p.size();
    if(n == 2) ans.push_back({0, 1});
    if(n < 3) return ans;
    auto nxt = [&](int x) { return (x+1 == n ? 0 : x+1); };
    auto area2 = [&](pt a, pt b, pt c) { return cross(b-a, c-a); };
    int b0 = 0;
    while(abs(area2(p[n - 1], p[0], p[nxt(b0)])) > abs(area2(p[n - 1],
        p[0], p[b0]))) ++b0;
    for(int b = b0, a = 0; b != 0 && a <= b0; ++a) {
        ans.push_back({a, b});
        while (abs(area2(p[a], p[nxt(a)], p[nxt(b)])) >
            abs(area2(p[a], p[nxt(a)], p[b]))) {
            b = nxt(b);
            if(a != b0 || b != 0) ans.push_back({ a, b });
            else return ans;
        }
        if(abs(area2(p[a], p[nxt(a)], p[nxt(b)])) == abs(area2(p[a],
            p[nxt(a)], p[b]))) {
            if(a != b0 || b != n-1) ans.push_back({ a, nxt(b) });
            else ans.push_back({ nxt(a), b });
        }
    }
    return ans;
}
pt centroid() {
    pt c{0, 0};
    lf scale = 6. * area(true);
    for(int i = 0, n = p.size(); i < n; ++i) {
        int j = (i+1 == n ? 0 : i+1);
        c = c + (p[i] + p[j]) * cross(p[i], p[j]);
    }
    return c / scale;
}
ll pick() {
    ll boundary = 0;
    for(int i = 0, n = p.size(); i < n; i++) {
        int j = (i+1 == n ? 0 : i+1);
        boundary += __gcd((ll)abs(p[i].x - p[j].x), (ll)abs(p[i].y -
            p[j].y));
    }
    return area() + 1 - boundary/2;
}
pt& operator[] (int i){ return p[i]; }
};
```

4.7 segment

```
bool in_disk(pt a, pt b, pt p) { // pt p inside ab disk
```

```cpp
    return dot(a-p, b-p) <= 0;
}
bool on_segment(pt a, pt b, pt p) { // p on ab
    return orient(a,b,p) == 0 && in_disk(a,b,p);
}
// ab crossing cd
bool proper_inter(pt a, pt b, pt c, pt d, pt &out) {
    T oa = orient(c,d,a),
    ob = orient(c,d,b),
    oc = orient(a,b,c),
    od = orient(a,b,d);
    /// Proper intersection exists iff opposite signs
    if (oa*ob < 0 && oc*od < 0) {
        out = (a*ob - b*oa) / (ob-oa);
        return true;
    }
    return false;
}
// intersection bwn segments
set<pt> inter_ss(pt a, pt b, pt c, pt d) {
    pt out;
    if (proper_inter(a,b,c,d,out)) return {out}; //if cross -> 1
    set<pt> s;
    if (on_segment(c,d,a)) s.insert(a); // a in cd
    if (on_segment(c,d,b)) s.insert(b); // b in cd
    if (on_segment(a,b,c)) s.insert(c); // c in ab
    if (on_segment(a,b,d)) s.insert(d); // d in ab
    return s; // 0, 2
}
lf pt_to_seg(pt a, pt b, pt p) { // p to ab
    if(a != b) {
        line l(a,b);
        if (l.cmp_proj(a,p) && l.cmp_proj(p,b)) /// if closest to
            projection = (a, p, b)
            return l.dist(p); /// output distance to line
    }
    return min(abs(p-a), abs(p-b)); /// otherwise distance to A or B
}
lf seg_to_seg(pt a, pt b, pt c, pt d) {
  pt dummy;
  if (proper_inter(a,b,c,d,dummy)) return 0; // ab intersects cd
  return min({pt_to_seg(a,b,c), pt_to_seg(a,b,d), pt_to_seg(c,d,a),
      pt_to_seg(c,d,b)}); // try the 4 pts
}
```

5 5 - Graph

5.1 2-satisfiability

```cpp
struct sat2 {
    int n;
    vector<vector<vector<int>>> g;
    vector<bool> vis, val;
    vector<int> comp;
    stack<int> st;

    sat2(int n) : n(n), g(2, vector<vector<int>>(2*n)), vis(2*n),
        val(2*n), comp(2*n) { }

    int neg(int x) { return 2*n-x-1; }
    void make_true(int u) { add_edge(neg(u), u); }
    void make_false(int u) { make_true(neg(u)); }
    void add_or(int u, int v) { implication(neg(u), v); }
    void diff(int u, int v) { eq(u, neg(v)); }
    void eq(int u, int v) {
        implication(u, v);
        implication(v, u);
    }
    void implication(int u, int v) {
        add_edge(u, v);
        add_edge(neg(v), neg(u));
    }

    void add_edge(int u, int v) {
        g[0][u].push_back(v);
        g[1][v].push_back(u);
    }

    void dfs(int id, int u, int t = 0) {
        vis[u] = true;
        for (auto &v : g[id][u])
            if (!vis[v]) dfs(id, v, t);
        if (id) comp[u] = t;
        else st.push(u);
    }

    void kosaraju() {
        for (int u = 0; u < n; u++) {
            if (!vis[u]) dfs(0, u);
```

```
            if (!vis[neg(u)]) dfs(0, neg(u));
        }
        vis.assign(2*n, false);
        int t = 0;
        while (!st.empty()) {
            int u = st.top(); st.pop();
            if (!vis[u]) dfs(1, u, t++);
        }
    }

    bool check() {
        kosaraju();
        for (int i = 0; i < n; i++) {
            if (comp[i] == comp[neg(i)]) return false;
            val[i] = comp[i] > comp[neg(i)];
        }
        return true;
    }
};
```

5.2 Articulation Bridges Biconnected

```
Dado un grafo no dirigido halla los puntos de articulacin, puentes y
    componentes biconexas. Para construir el block cut tree quitar los
    comentarios.

struct edge {
    int u, v, comp; //A que componente biconexa pertenece
    bool bridge; //Si la arista es un puente
};

const int MX = 1e5+5; //Cantidad maxima de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<edge> e; //Lista de aristas
stack<int> st;
int low[MX], num[MX], cont;
bool art[MX]; //Si el nodo es un punto de articulacion
//vector<set<int>> comps; //Componentes biconexas
//vector<vector<int>> tree; //Block cut tree
//vector<int> id; //Id del nodo en el block cut tree
int BCC; //Cantidad de componentes biconexas
int n, m; //Cantidad de nodos y aristas

void add_edge(int u, int v){
    g[u].push_back(e.size());
    g[v].push_back(e.size());
    e.push_back({u, v, -1, false});
}

void dfs(int u, int p = -1) {
    low[u] = num[u] = cont++;
    for (auto &i : g[u]) {
        edge &ed = e[i];
        int v = ed.u^ed.v^u;
        if (num[v] == -1) {
            st.push(i);
            dfs(v, i);
            if (low[v] > num[u])
                ed.bridge = true; //bridge
            if (low[v] >= num[u]) {
                art[u] = (num[u] > 0 || num[v] > 1); //articulation
                int last; //start biconnected
                //comps.push_back({});
                do {
                    last = st.top(); st.pop();
                    e[last].comp = BCC;
                    //comps.back().insert(e[last].u);
                    //comps.back().insert(e[last].v);
                } while (last != i);
                BCC++; //end biconnected
            }
            low[u] = min(low[u], low[v]);
        } else if (i != p && num[v] < num[u]) {
            st.push(i);
            low[u] = min(low[u], num[v]);
        }
    }
}

void build_tree() {
    tree.clear(); id.resize(n);
    for (int u = 0; u < n; u++) {
        if (art[u]) {
            id[u] = tree.size();
            tree.push_back({});
        }
    }
    for (auto &comp : comps) {
```

```
        int node = tree.size();
        tree.push_back({});
        for (auto &u : comp) {
            if (art[u]) {
                tree[id[u]].push_back(node);
                tree[node].push_back(id[u]);
            } else id[u] = node;
        }
    }
}

void init() {
    cont = BCC = 0;
    //comps.clear();
    e.clear();
    for (int i = 0; i <= n; i++) {
        g[i].clear();
        num[i] = -1; //no visitado
        art[i] = false;
    }
}
```

5.3 BFS

```
Busqueda en anchura sobre grafos. Recibe un nodo inicial u y visita todos
    los nodos alcanzables desde u.
BFS tambien halla la distancia mas corta entre el nodo inicial u y los
    demas nodos si todas las aristas tienen peso 1.

const int MX = 1e5+5; //Cantidad maxima de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<ll> dist; //Almacena la distancia a cada nodo
int n, m; //Cantidad de nodos y aristas

void bfs(int u) {
    queue<int> q;
    q.push(u);
    dist[u] = 0;

    while (q.size()) {
        u = q.front();
        q.pop();
        for (auto &v : g[u]) {
            if (dist[v] == -1) {
                dist[v] = dist[u] + 1;
                q.push(v);
            }
        }
    }
}

void init() {
    dist.assign(n, -1);
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.4 Bellman Ford

```
Dado un grafo con pesos, positivos o negativos, halla la ruta de costo
    minimo entre un nodo inicial u y todos los demas nodos.
Tambien halla ciclos negativos.

const ll inf = 1e18;
const int MX = 1e5+5; //Cantidad maxima de nodos
vector<pii> g[MX]; //Lista de adyacencia, u->[(v, cost)]
vector<ll> dist; //Almacena la distancia a cada nodo
//vector<int> cycle; //Para construir el ciclo negativo
int n, m; //Cantidad de nodos y aristas

/// O(n*m)
void bellmanFord(int src) {
    dist.assign(n, inf);
    dist[src] = 0;
    for (int i = 0; i < n-1; i++)
        for (int u = 0; u < n; u++)
            if (dist[u] != inf)
                for (auto &v : g[u]) {
                    dist[v.F] = min(dist[v.F], dist[u] + v.S);
                }
    //Encontrar ciclos negativos
    //cycle.clear();
    for (int u = 0; u < n; u++)
        if (dist[u] != inf)
            for (auto &v : g[u])
```

```
            if (dist[v.F] > dist[u] + v.S) { //Ciclo negativo
                dist[v.F] = -inf;
                //cycle.pb(v.F); //Para reconstruir
            }
}

void init() {
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.5 Bipartite Check

Modificacion del BFS para detectar si un grafo es bipartito.

```
const int MX = 1e5+5; //Cantidad maxima de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<int> color; //Almacena el color de cada nodo
bool bipartite; //true si el grafo es bipartito
int n, m; //Cantidad de nodos y aristas

void bfs(int u) {
    queue<int> q;
    q.push(u);
    color[u] = 0;

    while (q.size()) {
        u = q.front();
        q.pop();
        for (auto &v : g[u]) {
            if (color[v] == -1) {
                color[v] = 1-color[u];
                q.push(v);
            } else if (color[v] == color[u]) {
                bipartite = false;
                return;
            }
        }
    }
}

void init() {
    bipartite = true;
    color.assign(n, -1);
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.6 Cycle Detection

Determina si un grafo dirigido tiene o no ciclos (si es un DAG o no).

```
const int MX = 1e5+5; //Cantidad maxima de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<int> vis; //Marca el estado de los nodos ya visitados
bool cycle; //true si el grafo tiene ciclos
int n, m; //Cantidad de nodos y aristas

void dfs(int u) {
    if (cycle) return;
    vis[u] = 1;
    for (auto &v : g[u]) {
        if (!vis[v]) dfs(v);
        else if (vis[v] == 1) {
            cycle = true;
            break;
        }
    }
    vis[u] = 2;
}

void init() {
    vis.assign(n, 0);
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.7 DFS

Busqueda en profundidad sobre grafos. Recibe un nodo inicial u y visita todos los nodos alcanzables desde u.

```
DFS puede ser usado para contar la cantidad de componentes conexas en un
    grafo y puede ser modificado para que retorne informacion de los
    nodos dependiendo del problema.

const int MX = 1e5+5; //Cantidad maxima de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<bool> vis; //Marca los nodos ya visitados
int n, m; //Cantidad de nodos y aristas

void dfs(int u) {
    vis[u] = true;
    for (auto &v : g[u]) {
        if (!vis[v]) dfs(v);
    }
}

void init() {
    vis.assign(n, false);
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.8 Dijkstra

```
Dado un grafo con pesos no negativos halla la ruta de costo minimo entre
    un nodo inicial u y todos los demas nodos.

struct edge {
    int v; ll w;

    bool operator < (const edge &o) const {
        return o.w < w; //invertidos para que la pq ordene de < a >
    }
};

const ll inf = 1e18;
const int MX = 1e5+5; //Cantidad maxima de nodos
vector<edge> g[MX]; //Lista de adyacencia
vector<bool> vis; //Marca los nodos ya visitados
vector<ll> dist; //Almacena la distancia a cada nodo
int pre[MX]; //Almacena el nodo anterior para construir las rutas
int n, m; //Cantidad de nodos y aristas

void dijkstra(int u) {
    priority_queue<edge> pq;
    pq.push({u, 0});
    dist[u] = 0;

    while (pq.size()) {
        u = pq.top().v; pq.pop();
        if (!vis[u]) {
            vis[u] = true;
            for (auto &ed : g[u]) {
                int v = ed.v;
                if (!vis[v] && dist[v] > dist[u] + ed.w) {
                    dist[v] = dist[u] + ed.w;
                    pre[v] = u;
                    pq.push({v, dist[v]});
                }
            }
        }
    }
}

void init() {
    vis.assign(n, false);
    dist.assign(n, inf);
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.9 Flood Fill

```
Dado un grafo implicito como matriz, "colorea" y cuenta el tamao de las
    componentes conexas.
Esta funcion debe ser llamada con las coordenadas (i, j) donde se inicia
    el recorrido, busca cada caracter c1 de la componente, los remplaza
    por el caracter c2 y retorna el tamao.

const int MX = 1e3; //Tamanio maximo de la matriz
int dy[] = {1,1,0,-1,-1,-1, 0, 1}; //Posibles movimientos:
int dx[] = {0,1,1, 1, 0,-1,-1,-1}; // (8 direcciones)
char grid[MX][MX]; //Matriz de caracteres
int n, m; //Tamanio de la matriz
```

```
int floodfill(int y, int x, char c1, char c2) {
    if (y < 0 || y >= n || x < 0 || x >= m) return 0;
    if (grid[y][x] != c1) return 0;
    grid[y][x] = c2;
    int ans = 1;
    for (int i = 0; i < 8; i++) {
        ans += floodfill(y + dy[i], x + dx[i], c1, c2);
    }
    return ans;
}
```

5.10 Floyd Warshall

```
Dado un grafo halla la distancia minima entre cualquier par de nodos.
    g[i][j] guardara la distancia minima entre el nodo i y el j.

const int inf = 1e9;
const int MX = 505; //Cantidad maxima de nodos
int g[MX][MX]; //Matriz de adyacencia
int n, m; //Cantidad de nodos y aristas

void floydWarshall() {
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
}

void init() {
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= n; j++) {
            g[i][j] = (i == j ? 0 : inf);
        }
    }
}
```

5.11 Kruskal

```
Dado un grafo con pesos halla su arbol cobertor minimo. Debe agregarse
    Disjoint Set.

struct edge { int u, v, w; };

bool cmp(edge &a, edge &b) {
    return a.w < b.w;
}

const int MX = 1e5+5; //Cantidad maxima de nodos
vector<pair<int, int>> g[MX]; //Lista de adyacencia
vector<edge> e; //Lista de aristas
int n, m; //Cantidad de nodos y aristas

void kruskall() {
    sort(e.begin(), e.end(), cmp);
    dsu ds(n);
    int cnt = 0;
    for (auto &ed : e) {
        if (ds.find(ed.u) != ds.find(ed.v)) {
            ds.unite(ed.u, ed.v);
            g[ed.u].push_back({ed.v, ed.w});
            g[ed.v].push_back({ed.u, ed.w});
            if (++cnt == n-1) break;
        }
    }
}

void init() {
    e.clear();
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.12 Lowest Common Ancestor

```
Dados los nodos u y v de un arbol determina cual es el ancestro comun mas
    bajo entre u y v.
*Tambien puede determinar la arista de peso maximo/minimo entre los nodos
    u y v (Para esto quitar los "//")
Se debe ejecutar la funcion dfs() primero, el padre de la raiz es s
    mismo, w es el valor a almacenar del padre.

const int N = 4e5+2, inf = 1e9, LOG2 = 20;
```

```
int dep[N]; // Profundidad de cada nodo
int par[LOG2][N]; // Sparse table para guardar los padres
//int rmq[LOG2][N]; // Sparse table para guardar pesos

struct edge { int v, w; };
vector<edge> g[N];

void dfs(int u, int p, int d, int w){
    dep[u] = d;
    par[0][u] = p;
    // rmq[0][u] = w;
    for(int j = 1; j < LOG2; j++){
        par[j][u] = par[j-1][par[j-1][u]];
        // rmq[j][u] = max(rmq[j-1][u], rmq[j-1][par[j-1][u]]);
    }
    for(auto &ed: g[u]){
        int v = ed.v;
        int val = ed.w;
        if(v == p)continue;
        dfs(v, u, d+1, val);
    }
}

int lca(int u, int v){
    // int ans = -1;
    if(dep[v] < dep[u])swap(u, v);
    int d = dep[v]-dep[u];
    for(int j = LOG2-1; j >= 0; j--){
        if(d >> j & 1){
            // ans = max(ans, rmq[j][v]);
            v = par[j][v];
        }
    }
    // if(u == v)return ans;
    if(u == v)return u;
    for(int j = LOG2-1; j >= 0; j--){
        if(par[j][u] != par[j][v]){
            // ans = max({ans, rmq[j][u], rmq[j][v]});
            u = par[j][u];
            v = par[j][v];
        }
    }
    // return max({ans, rmq[1][u], rmq[0][v]}); // si la info es de los
        nodos
    // return max({ans, rmq[0][u], rmq[0][v]}); // si la info es de las
        aristas
    return par[0][u];
}
```

5.13 Prim

Dado un grafo halla el costo total de su arbol cobertor minimo.

```
struct edge {
    int v; ll w;

    bool operator < (const edge &o) const {
        return o.w < w; //invertidos para que la pq ordene de < a >
    }
};

const int MX = 1e5+5; //Cantidad maxima de nodos
vector<edge> g[MX]; //Lista de adyacencia
vector<bool> vis; //Marca los nodos ya visitados
ll ans; //Costo total del arbol cobertor minimo
int n, m; //Cantidad de nodos y aristas

void prim() {
    priority_queue<edge> pq;
    vis[0] = true;
    for (auto &ed : g[0]) {
        int v = ed.v;
        if (!vis[v]) pq.push({v, ed.w});
    }

    while (pq.size()) {
        int u = pq.top().v;
        ll w = pq.top().w;
        pq.pop();
        if (!vis[u]) {
            ans += w;
            vis[u] = true;
            for (auto &ed : g[u]) {
                int v = ed.v;
                if (!vis[v]) pq.push({v, ed.w});
            }
        }
```

```
    }
}

void init() {
    ans = 0;
    vis.assign(n, false);
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

5.14 Tarjan

Dado un grafo dirigido halla las componentes fuertemente conexas (SCC).

```
const int inf = 1e9;
const int MX = 1e5+5; //Cantidad maxima de nodos
vector<int> g[MX]; //Lista de adyacencia
stack<int> st;
int low[MX], pre[MX], cnt;
int comp[MX]; //Almacena la componente a la que pertenece cada nodo
int SCC; //Cantidad de componentes fuertemente conexas
int n, m; //Cantidad de nodos y aristas

void tarjan(int u) {
    low[u] = pre[u] = cnt++;
    st.push(u);

    for (auto &v : g[u]) {
        if (pre[v] == -1) tarjan(v);
        low[u] = min(low[u], low[v]);
    }
    if (low[u] == pre[u]) {
        while (true) {
            int v = st.top(); st.pop();
            low[v] = inf;
            comp[v] = SCC;
            if (u == v) break;
        }
        SCC++;
    }
}

void init() {
    cnt = SCC = 0;
    for (int i = 0; i <= n; i++) {
        g[i].clear();
        pre[i] = -1; //no visitado
    }
}
```

5.15 Topological Sort

Dado un grafo aciclico dirigido (DAG), ordena los nodos linealmente de tal manera que si existe una arista entre los nodos u y v entonces u aparece antes que v.
Este ordenamiento es una manera de poner todos los nodos en una linea recta de tal manera que las aristas vayan de izquierda a derecha.

```
const int MX = 1e5+5; //Cantidad maxima de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<bool> vis; //Marca los nodos ya visitados
deque<int> order; //Orden topologico del grafo
int n, m; //Cantidad de nodos y aristas

void toposort(int u) {
    vis[u] = true;
    for (auto &v : g[u]) {
        if (!vis[v]) toposort(v);
    }
    order.push_front(u);
}

void init() {
    order.clear();
    vis.assign(n, false);
    for (int i = 0; i <= n; i++) {
        g[i].clear();
    }
}
```

6 6 - Math

6.1 Basis of a Vector Space (mod 2 Field)

```
Dado un arreglo A con n numeros calcula en basis[] las mascaras con las
    cuales se pueden generar todos los diferentes xor que se generan al
    hacer xor entre los elementos de cualquier subconjunto A. La cantidad
    de xor diferentes es 2^sz.

const int D = 30; //maxima cantidad de bits
int basis[D];
int sz; //cantidad de mascaras en la base

/// O(n*D)
void insertVector(int mask) {
    for (int i = 0; i < D; ++i) {
        if (mask & (1<<i)) {
            if (!basis[i]) {
                basis[i] = mask;
                ++sz;
                break;
            }
            mask ^= basis[i];
        }
    }
}
```

6.2 Binomial Coefficient

```
Calcula el coeficiente binomial nCr, entendido como el numero de
    subconjuntos de r elementos escogidos de un conjunto con n elementos.

/// O(min(r, n-r))
ll ncr(ll n, ll r) {
    if (r < 0 || n < r) return 0;
    r = min(r, n-r);
    ll ans = 1;
    for (int i = 1; i <= r; i++) {
        ans = ans * (n-i+1) / i;
    }
    return ans;
}
```

6.3 Chinese Remainder Theorem

```
Encuentra un x tal que para cada i : x es congruente con A_i mod M_i
Devuelve {x, lcm}, donde x es la solucion con modulo lcm (lcm = LCM(M_0,
    M_1, ...)). Dado un k : x + k*lcm es solucion tambien.
Si la solucion no existe o la entrada no es valida devuelve {-1, -1}
Agregar Extended Euclides.

pair<int, int> crt(vector<int> A, vector<int> M) {
    int n = A.size(), ans = A[0], lcm = M[0];
    for (int i = 1; i < n; i++) {
        int d = euclid(lcm, M[i]);
        if ((A[i] - ans) % d) return {-1, -1};
        int mod = lcm / d * M[i];
        ans = (ans + x * (A[i] - ans) / d % (M[i] / d) * lcm) % mod;
        if (ans < 0) ans += mod;
        lcm = mod;
    }
    return {ans, lcm};
}
```

6.4 Diophantine Ecuations

```
Encuentra x, y en la ecuacin de la forma ax + by = c
Agregar Extended Euclides.

ll g;
bool diophantine(ll a, ll b, ll c) {
        x = y = 0;
        if(!a && !b) return (!c); // slo hay solucin con c = 0
        g = euclid(abs(a), abs(b));
        if(c % g) return false;
        a /= g; b /= g; c /= g;
        if(a < 0) x *= -1;
        x = (x % b) * (c % b) % b;
        if(x < 0) x += b;
        y = (c - a*x) / b;
        return true;
}
```

6.5 Discrete Logarithm

```
Devuelve un entero x tal que a^x = b (mod m) or -1 si no existe tal x.
    Agregar Modular Exponentiation.

ll discrete_log(ll a, ll b, ll m) {
    a %= m, b %= m;
    if (b == 1) return 0;
    int cnt = 0;
    ll tmp = 1;
    for (int g = __gcd(a, m); g != 1; g = __gcd(a, m)) {
        if (b%g) return -1;
        m /= g, b /= g;
        tmp = tmp*a / g % m;
        ++cnt;
        if (b == tmp) return cnt;
    }
    map<ll, int> w;
    int s = ceil(sqrt(m));
    ll base = b;
    for (int i = 0; i < s; i++) {
        w[base] = i;
        base = base*a % m;
    }
    base = expmod(a, s, m);
    ll key = tmp;
    for (int i = 1; i <= s+1; i++) {
        key = key*base % m;
        if (w.count(key)) return i*s - w[key] + cnt;
    }
    return -1;
}
```

6.6 Divisors

```
* Calcula la suma de los divisores de n. Agregar Prime Factorization y
    Modular Exponentiation (sin el modulo).

ll sumDiv(ll n) {
    map<ll, int> f;
    fact(n, f);
    ll ans = 1;
    for (auto p : f) {
        ans *= (exp(p.first, p.second+1)-1)/(p.first-1);
    }
    return ans;
}

* Calcula la cantidad de divisores de n. Agregar Prime Factorization.

ll cantDiv(ll n) {
    map<ll, int> f;
        fact(n, f);
    ll ans = 1;
    for (auto p : f) ans *= (p.second + 1);
    return ans;
}

* Calcular la cantidad de divisores para todos los numeros menores o
    iguales a MX con Sieve of Eratosthenes.

const int MX = 1e6;
bool marked[MX+1];
vector<int> cnt;

void sieve() {
    cnt.assign(MX+1, 1);
    marked[0] = marked[1] = true;
    for (int i = 2; i <= MX; i++) {
        if (marked[i]) continue;
        cnt[i]++;
        for (int j = i*2; j <= MX ; j += i) {
            int n = j, c = 1;
            while (n%i == 0) n /= i, c++;
            cnt[j] *= c;
            marked[j] = true;
        }
    }
}
```

6.7 Euler Totient

```
La funcion phi de Euler devuelve la cantidad de enteros positivos menores
    o iguales a n que son coprimos con n (gcd(n, i) = 1)

/// O(sqrt(n))
```

```
ll phi(ll n) {
    ll ans = n;
    for (int p = 2; p <= n/p; ++p) {
        if (n % p == 0) ans -= ans / p;
        while (n % p == 0) n /= p;
    }
    if (n > 1) ans -= ans / n;
    return ans;
}
```

```
* Calcular el Euler totient para todos los numeros menores o iguales a MX
    con Sieve of Eratosthenes.
```

```
const int MX = 1e6;
bool marked[MX+1];
int phi[MX+1];
/// O(MX log(log(MX)))
void sieve() {
    iota(phi, phi+MX+1, 0);
    marked[0] = marked[1] = true;
    for (int i = 2; i <= MX; i++) {
        if (marked[i]) continue;
        for (int j = i; j <= MX ; j += i) {
            phi[j] -= phi[j] / i;
            marked[j] = true;
        }
        marked[i] = false;
    }
}
```

6.8 Extended Euclides

```
El algoritmo de Euclides extendido retorna el gcd(a, b) y calcula los
    coeficientes enteros X y Y que satisfacen la ecuacion: a*X + b*Y =
    gcd(a, b).
```

```
int x, y;
/// O(log(max(a, b)))
int euclid(int a, int b) {
    if(b == 0) { x = 1; y = 0; return a; }
    int d = euclid(b, a%b);
    int aux = x;
    x = y;
    y = aux - a/b*y;
    return d;
}
```

6.9 Fast Fourier Transform

```
Multiplicacion de polinomios en O(n log n)
```

```
const double PI = acos(-1.0);

namespace fft {
    struct pt {
        double r, i;
        pt(double r = 0.0, double i = 0.0) : r(r), i(i) {}
        pt operator + (const pt &b) { return pt(r+b.r, i+b.i); }
        pt operator - (const pt &b) { return pt(r-b.r, i-b.i); }
        pt operator * (const pt &b) { return pt(r*b.r - i*b.i, r*b.i +
            i*b.r); }
    };
    vector<int> rev;

    void fft(vector<pt> &y, int on) {
        int n = y.size();
        for (int i = 1; i < n; i++)
            if (i < rev[i]) swap(y[i], y[rev[i]]);
        for (int m = 2; m <= n; m <<= 1) {
            double ang = -on * 2 * PI / m;
            pt wm(cos(ang), sin(ang));
            for (int k = 0; k < n; k += m) {
                pt w(1, 0);
                for (int j = 0; j < m / 2; j++) {
                    pt u = y[k + j];
                    pt t = w * y[k + j + m / 2];
                    y[k + j] = u + t;
                    y[k + j + m / 2] = u - t;
                    w = w * wm;
                }
            }
        }
        if (on == -1) for (int i = 0; i < n; i++) y[i].r /= n;
    }

    vector<ll> mul(vector<ll> &a, vector<ll> &b) {
```

```
        int n = 1, t = 0, la = a.size(), lb = b.size();
        for (; n <= (la+lb+1); n <<= 1, t++); t = 1<<(t-1);
        vector<pt> x1(n), x2(n);
        rev.assign(n, 0);
        for (int i = 0; i < n; i++) rev[i] = rev[i >> 1] >> 1 | (i & 1 ? t
            : 0);
        for (int i = 0; i < la; i++) x1[i] = pt(a[i], 0);
        for (int i = 0; i < lb; i++) x2[i] = pt(b[i], 0);
        fft(x1, 1); fft(x2, 1);
        for (int i = 0; i < n; i++) x1[i] = x1[i] * x2[i];
        fft(x1, -1);
        vector<ll> ans(n);
        for (int i = 0; i < n; i++) ans[i] = x1[i].r + 0.5;
        return ans;
    }
}
```

6.10 Fibonacci mod m

```
Calcula fibonacci(n) % m.

/// O(log(n))
int fibmod(ll n, int m) {
    int a = 0, b = 1, c;
    for (int i = 63-__builtin_clzll(n); i >= 0; i--) {
        c = a;
        a = (1ll*c*(2ll*b-c+m)) % m;
        b = (1ll*c*c + 1ll*b*b) % m;
        if ((n>>i) & 1) {
            c = (a+b) % m;
            a = b; b = c;
        }
    }
    return a;
}
```

6.11 Gauss Jordan

```
Algoritmo de eliminacin Gauss-Jordan O(N ^ 3)
Soluciona un sistema de ecuaciones de la forma:
a11 x1 + a12 x2 + ... + a1m xm = b1
a21 x1 + a22 x2 + ... + a2m xm = b2
an x1 + an2 x2 + ... + anm xm = bn

El vector a contiene los valores de la matriz, cada fila es una ecuacin,
    la ltima columna contiene los valores b.

const int EPS = 1;
int gauss (vector<vector<int>>& a, vector<int> &ans) {
  int n = a.size(), m = a[0].size()-1;
  vector<int> where(m, -1);
  for(int col = 0, row = 0; col < m && row < n; ++col) {
    int sel = row;
    for(int i = row; i < n; ++i)
      if(abs(a[i][col]) > abs(a[sel][col])) sel = i;
    if(abs(a[sel][col]) < EPS) continue;
    swap(a[sel], a[row]);
    where[col] = row;
    for(int i = 0; i < n; ++i)
      if(i != row) {
        int c = divide(a[i][col], a[row][col]);
        for(int j = col; j <= m; ++j)
          a[i][j] = sub(a[i][j], mul(a[row][j], c));
      }
    ++row;
  }
  ans.assign(m, 0);
  for(int i = 0; i < m; ++i)
    if(where[i] != -1) ans[i] = divide(a[where[i]][m], a[where[i]][i]);
  for(int i = 0; i < n; ++i) {
    int sum = 0;
    for(int j = 0; j < m; ++j)
      sum = add(sum, mul(ans[j], a[i][j]));
    if(sum != a[i][m]) return 0;
  }
  for(int i = 0; i < m; ++i)
    if(where[i] == -1) return -1;
  return 1;
}


Gauss jordan para operaciones de xor

int gauss (vector<bitset<N>> &a, vector<bitset<N>> &b, int n, int m,
    vector<bitset<N>> &ans) {
  vector<int> where(m, -1);
```

```
  for(int col = 0, row = 0; col < m && row < n; ++col) {
    for(int i = row; i < n; ++i){
      if(a[i][col]){
        swap(a[i], a[row]);
        swap(b[i], b[row]);
        break;
      }
    }
    if(!a[row][col])continue;
    where[col] = row;
    for(int i = 0; i < n; ++i)
      if(i != row && a[i][col]) {
        a[i] ^= a[row];
        b[i] ^= b[row];
      }
    ++row;
  }
  ans.assign(m, 0);
  for(int i = 0; i < m; ++i)
    if(where[i] != -1) ans[i] = b[where[i]];
  for(int i = 0; i < n; ++i) {
    if(ans[i] == 0) return 0;
  }
  for(int i = 0; i < m; ++i)
    if(where[i] == -1) return -1; /// infinite solutions
  return 1;
}
```

6.12 Gaussian Elimination

```
Resuelve sistemas de ecuaciones lineales por eliminacion Gaussiana.
    mat[][] contiene los valores de la matriz cuadrada y los resultados
    de las ecuaciones en la ultima columna. Retorna un vector<> con el
    valor de las n incongnitas. Los resultados pueden necesitar redondeo.
```

```
const int MX = 100;
double mat[MX][MX+1];
int n;
/// O(n^3)
vector<double> gauss() {
   vector<double> vec(n-1);
   for (int i = 0; i < n-1; i++) {
      int pivot = i;
      for (int j = i+1; j < n; j++)
         if (abs(mat[j][i]) > abs(mat[pivot][i])) pivot = j;
      for (int j = i; j <= n; j++)
         swap(mat[i][j], mat[pivot][j]);
      for (int j = i+1; j < n; j++)
         for (int k = n; k >= i; k--)
            mat[j][k] -= mat[i][k]*mat[j][i] / mat[i][i];
   }
   for (int i = n-1; i >= 0; i--) {
      double tmp = 0.0;
      for (int j = i+1; j < n; j++) tmp += mat[i][j]*vec[j];
      vec[i] = (mat[i][n]-tmp) / mat[i][i];
   }
   return vec;
}
```

6.13 Greatest Common Divisor

```
Calcula el maximo comun divisor entre a y b mediante el algoritmo de
    Euclides. Tambien se puede usar __gcd(a, b).
```

```
/// O(log(max(a, b)))
int gcd(int a, int b) {
   return b == 0 ? a : gcd(b, a%b);
}
```

6.14 Linear Recurrence

```
Calcula el n-esimo termino de una recurrencia lineal (que depende de los
    k terminos anteriores).
* Llamar init(k) en el main una unica vez si no es necesario inicializar
    las matrices multiples veces.
Este ejemplo calcula el fibonacci de n como la suma de los k terminos
    anteriores de la secuencia (En la secuencia comun k es 2).
Agregar Matrix Multiplication con un construcctor vacio.
```

```
matrix F, T;

void init(int k) {
   F = {k, 1}; // primeros k terminos
   F[k-1][0] = 1;
```

```
    T = {k, k}; // fila k-1 = coeficientes: [c_k, c_k-1, ..., c_1]
    for (int i = 0; i < k-1; i++) T[i][i+1] = 1;
    for (int i = 0; i < k; i++) T[k-1][i] = 1;
}
/// O(k^3 log(n))
int fib(ll n, int k = 2) {
    init(k);
    matrix ans = pow(T, n+k-1) * F;
    return ans[0][0];
}
```

6.15 Lowest Common Multiple

Calculo del minimo comun multiplo usando el maximo comun divisor.

```
/// O(log(max(a, b)))
int lcm(int a, int b) {
    return a / __gcd(a, b) * b;
}
```

6.16 Matrix Multiplication

Estructura para realizar operaciones de multiplicacion y exponenciacion modular sobre matrices.

```
const int mod = 1e9+7;

struct matrix {
    vector<vector<int>> v;
    int n, m;

    matrix(int n, int m, bool o = false) : n(n), m(m), v(n,
         vector<int>(m)) {
        if (o) while (n--) v[n][n] = 1;
    }

    matrix operator * (const matrix &o) {
        matrix ans(n, o.m);
        for (int i = 0; i < n; i++)
            for (int k = 0; k < m; k++) if (v[i][k])
                for (int j = 0; j < o.m; j++)
                    ans[i][j] = (1ll*v[i][k]*o.v[k][j] + ans[i][j]) % mod;
        return ans;
    }

    vector<int>& operator[] (int i) { return v[i]; }
};

matrix pow(matrix b, ll e) {
    matrix ans(b.n, b.m, true);
    while (e) {
        if (e&1) ans = ans*b;
        b = b*b;
        e /= 2;
    }
    return ans;
}
```

6.17 Miller Rabin

El algoritmo de Miller-Rabin determina si un numero es primo o no. Agregar Modular Exponentiation (para m ll) y Modular Multiplication.

```
/// O(log^3(n))
bool test(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;
    for (int i = 0; i < s-1; i++) {
        x = mulmod(x, x, n);
        if (x == 1) return false;
        if (x+1 == n) return true;
    }
    return false;
}

bool is_prime(ll n) {
    if (n == 1) return false;
    int ar[] = {2,3,5,7,11,13,17,19,23};
    for (auto &p : ar) if (!test(n, p)) return false;
    return true;
}
```

6.18 Mobius

```
La funcion mu de Mobius devuelve 0 si n es divisible por algun cuadrado
    (x^2).
Si n es libre de cuadrados entonces devuelve 1 o -1 si n tiene un numero
    par o impar de factores primos distintos.
* Calcular Mobius para todos los numeros menores o iguales a MX con Sieve
    of Eratosthenes.

const int MX = 1e6;
short mu[MX+1] = {0, 1};
/// O(MX log(log(MX)))
void mobius() {
    for (int i = 1; i <= MX; i++) {
        if (!mu[i]) continue;
        for (int j = i*2; j <= MX; j += i) {
            mu[j] -= mu[i];
        }
    }
}
```

6.19 Modular Exponentiation

```
* Calcula (b^e) % m (e puede ser ll). b debe estar ya con modulo m.
Si m es ll se debe cambiar todo a ll, agregar Modular Multiplication y
    calcular las multiplicaciones con mulmod().
/// O(log(e))
int expmod(int b, int e, int m) {
    int ans = 1;
    while (e) {
        if (e&1) ans = (1ll*ans*b) % m;
        b = (1ll*b*b) % m;
        e /= 2;
    }
    return ans;
}
```

6.20 Modular Inverse

```
El inverso multiplicativo modular de a % m es un entero b tal que (a*b) %
    m = 1. Este existe siempre y cuando a y m sean coprimos (gcd(a, m) =
    1).
El inverso modular de a se utiliza para calcular (n/a) % m como (n*b) % m.

* Se puede calcular usando el algoritmo de Euclides extendido. Agregar
    Extended Euclides.

/// O(log(max(a, m)))
int invmod(int a, int m) {
    int d = euclid(a, m);
    if (d > 1) return -1;
    return (x % m + m) % m;
}

* Si m es un numero primo, se puede calcular aplicando el pequeo teorema
    de Fermat. Agregar Modular Exponentiation.

/// O(log(m))
int invmod(int a, int m) {
    return expmod(a, m-2, m);
}

* Calcular el inverso modulo m para todos los numeros menores o iguales a
    MX

const int MX = 1e6;
ll inv[MX+1];
/// O(MX)
void invmod(ll m) {
    inv[1] = 1;
    for(int i = 2; i <= MX; i++)
        inv[i] = m - (m/i) * inv[m%i] % m;
}
```

6.21 Modular Multiplication

```
* Calcula (a*b) % m sin overflow cuando m es ll.
/// O(1)
ll mulmod(ll a, ll b, ll m) {
    ll r = a*b-(ll)((long double)a*b/m+.5)*m;
    return r < 0 ? r+m : r;
}
```

6.22 Pisano Period

```
Calcula el Periodo de Pisano de m, que es el periodo con el cual se
    repite la Sucesion de Fibonacci modulo m.
Si m es primo el algoritmo funciona (considerable) para m < 10^6. Aregar
    Modular Exponentiation (sin el modulo) y Lowest Common Multiple (para
    ll).

ll period(ll m) {
    ll a = 0, b = 1, c, pp = 0;
    do {
        c = (a+b) % m;
        a = b; b = c; pp++;
    } while (a != 0 || b != 1);
    return pp;
}

ll pisanoPrime(ll p, int e) {
    return expmod(p, e-1) * period(p);
}

ll pisanoPeriod(ll m) {
    ll pp = 1;
    for (ll p = 2; p <= m/p; p++) {
        if (m%p == 0) {
            int e = 0;
            while (m%p == 0) e++, m /= p;
            pp = lcm(pp, pisanoPrime(p, e));
        }
    }
    if (m > 1) pp = lcm(pp, period(m));
    return pp;
}
```

6.23 Pollard Rho

```
La funcion Rho de Pollard calcula un divisor no trivial de n. Agregar
    Modular Multiplication.

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

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;
        d = gcd(abs(x-y), n);
    }
    return d == n ? rho(n) : d;
}

* Version optimizada

ll add(ll a, ll b, ll m) { return (a += b) < m ? a : a-m; }

ll rho(ll n) {
    static ll s[MX];
    while (1) {
        ll x = rand()%n, y = x, c = rand()%n;
        ll *px = s, *py = s, v = 0, p = 1;
        while (1) {
            *py++ = y = add(mulmod(y, y, n), c, n);
            *py++ = y = add(mulmod(y, y, n), c, n);
            if ((x = *px++) == y) break;
            ll t = p;
            p = mulmod(p, abs(y-x), n);
            if (!p) return gcd(t, n);
            if (++v == 26) {
                if ((p = gcd(p, n)) > 1 && p < n) return p;
                v = 0;
            }
        }
        if (v && (p = gcd(p, n)) > 1 && p < n) return p;
    }
}
```

6.24 Prime Factorization

```
Tres funciones diferentes que guardan en el map f los pares <primo,
    exponente> de la descomposicion en factores primos de n.

1.1) Iterando hasta sqrt(n)
/// O(sqrt(n))
```

```
void fact(ll n, map<ll, int> &f) {
    for (int p = 2; 1ll*p*p <= n; p++)
        while (n%p == 0) f[p]++, n /= p;
    if (n > 1) f[n]++;
}

1.2) Version optimizada. Precalcular los primos <= sqrt(n) para iterarlos
    en el for.
/// O(sqrt(n)/log(sqrt(n)))

2.1) Utilizando Pollard Rho y Miller Rabin (agregar funciones).
/// O(log(n)^3) aprox
void fact(ll n, map<ll, int> &f) {
    if (n == 1) return;
    if (is_prime(n)) { f[n]++; return; }
    ll q = rho(n);
    fact(q, f); fact(n/q, f);
}

2.2) Version optimizada. Usar Pollard Rho optimizado y sieve() del metodo
    3.

void fact(ll n, map<ll, int> &f) {
    for (auto &p : f) while (n%p.F == 0) { p.S++; n /= p.F; }
    if (n <= MX) while (n > 1) { f[prime[n]]++; n /= prime[n]; }
    else if (is_prime(n)) f[n]++;
    else { ll q = rho(n); fact(q, f); fact(n/q, f); }
}

3) Precalculando un divisor primo para cada n (solo para n <= 1e6 aprox).

const int MX = 1e6;
int prime[MX+1];

void sieve() {
    for (int i = 2; i <= MX; i++) {
        if (prime[i]) continue;
        for (int j = i; j <= MX; j += i) {
            prime[j] = i;
        }
    }
}
/// O(log(n))
void fact(int n, map<int, int> &f) {
    while (n > 1) {
        f[prime[n]]++;
        n /= prime[n];
    }
}
```

6.25 Sieve of Eratosthenes

```
Guarda en primes los numeros primos menores o iguales a MX. Para saber si
    p es un nmero primo, hacer: if (!marked[p]).

const int MX = 1e6;
bool marked[MX+1];
vector<int> primes;
/// O(MX log(log(MX)))
void sieve() {
    marked[0] = marked[1] = true;
    for (int i = 2; i <= MX; i++) {
        if (marked[i]) continue;
        primes.push_back(i);
        for (ll j = 1ll*i*i; j <= MX; j += i) marked[j] = true;
    }
}
```

6.26 Simplex

```
Maximizar la ecuacin x1 + x2 + x3 ...
sujeta a restricciones x1 + x2 <= 2, x2 + x3 <= 5 ...
A: matriz de ecuaciones, contiene los coeficientes de cada variable
B: vector con los coeficientes de las restricciones
C: costos de las variables

const double EPS = 1e-6;

struct simplex {
    vector<int> 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];
        for(int i = 0; i < m; i++){
            if(i != y) a[x][i] /= a[x][y];
        }
        a[x][y] = 1 / a[x][y];
        for(int i = 0; i < n; i++){
            if(i != x && abs(a[i][y]) > EPS) {
                b[i] -= a[i][y] * b[x];
                for(int j = 0; j < m; j++){
                    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];
        for(int i = 0; i < m; i++){
            if(i != y) c[i] -= c[y] * a[x][i];
        }
        c[y] =- c[y] * a[x][y];
    }

    simplex(vector<vector<double>> &A, vector<double> &B, vector<double>
         &C) {
        a = A; b = B; c = C;
        n = b.size(); m = c.size(); z = 0.0;
        X.resize(m); iota(X.begin(), X.end(), 0);
        Y.resize(n); iota(Y.begin(), Y.end(), m);
    }

    pair<double, vector<double>> maximize() {
        while(true) {
            int x = -1, y = -1;
            double mn = -EPS;
            for(int i = 0; i < n; i++){
                if(b[i] < mn) mn = b[i], x = i;
            }
            if(x < 0) break;
            for(int i = 0; i < m; i++){
                if(a[x][i] < -EPS) {
                    y = i;
                    break;
                }
            }
            assert(y >= 0); // no hay solucion para Ax <= b
            pivot(x, y);
        }
        while(true) {
            double mx = EPS;
            int x = -1, y = -1;
            for(int i = 0; i < m; i++){
                if(c[i] > mx) mx = c[i], y = i;
            }
            if(y < 0) break;
            double mn = 1e200;
            for(int i = 0; i < n; i++){
                if(a[i][y] > EPS && b[i] / a[i][y] < mn)
                    mn = b[i] / a[i][y], x = i;
            }
            assert(x >= 0); // unbounded
            pivot(x, y);
        }
        vector<double> r(m);
        for(int i = 0; i < n; i++){
            if(Y[i] < m) r[ Y[i] ] = b[i];
        }
        return {z, r};
    }
};
```

6.27 Ternary Search

```
Retorna el valor minimo de una funcion entre l y r. Se recomienda usar de
    50 a 90 iteraciones.

double f(double x) {
    double y = x; //funcion a evaluar que depende de x
    return y;
}

double ternary_search(double l, double r, int it) {
    double a = (2.0*l + r)/3.0;
    double b = (l + 2.0*r)/3.0;
    if (it == 0) return f(a);
    if (f(a) < f(b)) return ternary_search(l, b, it-1);
    return ternary_search(a, r, it-1);
}
```

7 6 - Network Flows

7.1 Blossom

```
Halla el mximo match en un grafo general O(E * v ^2)

struct network {
    struct struct_edge {
        int v; struct_edge * n;
    };

    typedef struct_edge* edge;

    int n;
    struct_edge pool[MAXE]; ///2*n*n;
    edge top;
    vector<edge> adj;
    queue<int> q;
    vector<int> f, base, inq, inb, inp, match;
    vector<vector<int>> ed;

    network(int n) : n(n), match(n, -1), adj(n), top(pool), f(n), base(n),
                    inq(n), inb(n), inp(n), ed(n, vector<int>(n)) {}

    void add_edge(int u, int v) {
        if(ed[u][v]) return;
        ed[u][v] = 1;
        top->v = v, top->n = adj[u], adj[u] = top++;
        top->v = u, top->n = adj[v], adj[v] = top++;
    }

    int get_lca(int root, int u, int v) {
        fill(inp.begin(), inp.end(), 0);
        while(1) {
            inp[u = base[u]] = 1;
            if(u == root) break;
            u = f[ match[u] ];
        }
        while(1) {
            if(inp[v = base[v]]) return v;
            else v = f[ match[v] ];
        }
    }

    void mark(int lca, int u) {
        while(base[u] != lca) {
            int v = match[u];
            inb[ base[u ]] = 1;
            inb[ base[v] ] = 1;
            u = f[v];
            if(base[u] != lca) f[u] = v;
        }
    }

    void blossom_contraction(int s, int u, int v) {
        int lca = get_lca(s, u, v);
        fill(inb.begin(), inb.end(), 0);
        mark(lca, u); mark(lca, v);
        if(base[u] != lca) f[u] = v;
        if(base[v] != lca) f[v] = u;
        for(int u = 0; u < n; u++){
            if(inb[base[u]]) {
                base[u] = lca;
                if(!inq[u]) {
                    inq[u] = 1;
                    q.push(u);
                }
            }
        }
    }

    int bfs(int s) {
        fill(inq.begin(), inq.end(), 0);
        fill(f.begin(), f.end(), -1);
        for(int i = 0; i < n; i++) base[i] = i;
        q = queue<int>();
        q.push(s);
        inq[s] = 1;
        while(q.size()) {
            int u = q.front(); q.pop();
            for(edge e = adj[u]; e; e = e->n) {
                int v = e->v;
                if(base[u] != base[v] && match[u] != v) {
                    if((v == s) || (match[v] != -1 && f[match[v]] != -1)){
                        blossom_contraction(s, u, v);
                    }else if(f[v] == -1) {
                        f[v] = u;
                        if(match[v] == -1) return v;
                        else if(!inq[match[v]]) {
```

```
                    inq[match[v]] = 1;
                    q.push(match[v]);
                }
            }
        }
    }
    }
    return -1;
}

int doit(int u) {
    if(u == -1) return 0;
    int v = f[u];
    doit(match[v]);
    match[v] = u; match[u] = v;
    return u != -1;
}

/// (i < net.match[i]) => means match
int maximum_matching() {
    int ans = 0;
    for(int u = 0; u < n; u++)
        ans += (match[u] == -1) && doit(bfs(u));
    return ans;
}
};
```

7.2 Dinic

```
Halla el flujo mximo O(E * V ^ 2)

struct edge { int v, cap, inv, flow; };

struct network {
    int n, s, t;
    vector<int> lvl;
    vector<vector<edge>> g;

    network(int n) : n(n), lvl(n), g(n) {}

    void add_edge(int u, int v, int c) {
        g[u].push_back({v, c, (int)g[v].size(), 0});
        g[v].push_back({u, 0, (int)g[u].size()-1, c});
    }

    bool bfs() {
        fill(lvl.begin(), lvl.end(), -1);
        queue<int> q;
        lvl[s] = 0;
        for(q.push(s); q.size(); q.pop()) {
            int u = q.front();
            for(auto &e : g[u]) {
                if(e.cap > 0 && lvl[e.v] == -1) {
                    lvl[e.v] = lvl[u]+1;
                    q.push(e.v);
                }
            }
        }
        return lvl[t] != -1;
    }

    int dfs(int u, int nf) {
        if(u == t) return nf;
        int res = 0;
        for(auto &e : g[u]) {
            if(e.cap > 0 && lvl[e.v] == lvl[u]+1) {
                int tf = dfs(e.v, min(nf, e.cap));
                res += tf; nf -= tf; e.cap -= tf;
                g[e.v][e.inv].cap += tf;
                g[e.v][e.inv].flow -= tf;
                e.flow += tf;
                if(nf == 0) return res;
            }
        }
        if(!res) lvl[u] = -1;
        return res;
    }

    int max_flow(int so, int si, int res = 0) {
        s = so; t = si;
        while(bfs()) res += dfs(s, INT_MAX);
        return res;
    }
};
```

7.3 Hungarian

```
Halla el mximo match en un grafo bipartito con pesos (min cost) O(V ^ 3)

typedef ll T;
const T inf = 1e18;

struct hung {
    int n, m;
    vector<T> u, v; vector<int> p, way;
    vector<vector<T>> g;

    hung(int n, int m):
        n(n), m(m), g(n+1, vector<T>(m+1, inf-1)),
        u(n+1), v(m+1), p(m+1), way(m+1) {}

    void set(int u, int v, T w) { g[u+1][v+1] = w; }

    T assign() {
        for (int i = 1; i <= n; ++i) {
            int j0 = 0; p[0] = i;
            vector<T> minv(m+1, inf);
            vector<char> used(m+1, false);
            do {
                used[j0] = true;
                int i0 = p[j0], j1; T delta = inf;
                for (int j = 1; j <= m; ++j) if (!used[j]) {
                    T cur = g[i0][j] - u[i0] - v[j];
                    if (cur < minv[j]) minv[j] = cur, way[j] = j0;
                    if (minv[j] < delta) delta = minv[j], j1 = j;
                }
                for (int j = 0; j <= m; ++j)
                    if (used[j]) u[p[j]] += delta, v[j] -= delta;
                    else minv[j] -= delta;
                j0 = j1;
            } while (p[j0]);
            do {
                int j1 = way[j0]; p[j0] = p[j1]; j0 = j1;
            } while (j0);
        }
        return -v[0];
    }
};
```

7.4 Maximum Bipartite Matching

```
Halla el maximo match en un grafo bipartito O(|E|*|V|)

struct mbm {
    int l, r;
    vector<vector<int>> g;
    vector<int> match, vis;

    mbm(int l, int r) : l(l), r(r), g(l) {}

    void add_edge(int l, int r) {
        g[l].push_back(r);
    }

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

    int max_matching() {
        int ans = 0;
        match.assign(r, -1);
        for (int u = 0; u < l; ++u) {
            vis.assign(r, 0);
            ans += dfs(u);
        }
        return ans;
    }
};

Hopcroft Karp: O(E * sqrt(V))

const int INF = INT_MAX;

struct mbm {
    vector<vector<int>> g;
    vector<int> d, match;
    int nil, l, r;
```

```
/// u -> 0 to l, v -> 0 to r
mbm(int l, int r) : l(l), r(r), nil(l+r), g(l+r),
                  d(1+l+r, INF), match(l+r, l+r) {}

void add_edge(int a, int b) {
   g[a].push_back(l+b);
   g[l+b].push_back(a);
}

bool bfs() {
   queue<int> q;
   for(int u = 0; u < l; u++) {
      if(match[u] == nil) {
         d[u] = 0;
         q.push(u);
      } else {
         d[u] = INF;
      }
   }
   d[nil] = INF;
   while(q.size()) {
      int u = q.front(); q.pop();
      if(u == nil) continue;
      for(auto v : g[u]) {
         if(d[ match[v] ] == INF) {
            d[ match[v] ] = d[u]+1;
            q.push(match[v]);
         }
      }
   }
   return d[nil] != INF;
}

bool dfs(int u) {
   if(u == nil) return true;
   for(int v : g[u]) {
      if(d[ match[v] ] == d[u]+1 && dfs(match[v])) {
         match[v] = u; match[u] = v;
         return true;
      }
   }
   d[u] = INF;
   return false;
}
```

```
   int max_matching() {
      int ans = 0;
      while(bfs()) {
         for(int u = 0; u < l; u++) {
            ans += (match[u] == nil && dfs(u));
         }
      }
      return ans;
   }
};
```

7.5 MinCost MaxFlow

```
Dado un grafo, halla el flujo maximo y el costo minimo entre el source s
    y el sink t.

struct edge {
   int u, v, cap, flow, cost;
   int rem() { return cap - flow; }
};

const int inf = 1e9;
const int MX = 405; //Cantidad maxima TOTAL de nodos
vector<int> g[MX]; //Lista de adyacencia
vector<edge> e; //Lista de aristas
vector<bool> in_queue; //Marca los nodos que estan en cola
vector<int> pre, dist, cap; //Almacena el nodo anterior, la distancia y
    el flujo de cada nodo
int mxflow, mncost; //Flujo maximo y costo minimo
int N; //Cantidad TOTAL de nodos

void add_edge(int u, int v, int cap, int cost) {
   g[u].push_back(e.size());
   e.push_back({u, v, cap, 0, cost});
   g[v].push_back(e.size());
   e.push_back({v, u, 0, 0, -cost});
}

void flow(int s, int t) {
   mxflow = mncost = 0;
   in_queue.assign(N, false);
   while (true) {
      dist.assign(N, inf); dist[s] = 0;
```

```
        cap.assign(N, 0); cap[s] = inf;
        pre.assign(N, -1); pre[s] = 0;
        queue<int> q; q.push(s);
        in_queue[s] = true;

        while (q.size()) {
            int u = q.front(); q.pop();
            in_queue[u] = false;
            for (int &id : g[u]) {
                edge &ed = e[id];
                int v = ed.v;
                if (ed.rem() && dist[v] > dist[u]+ed.cost) {
                    dist[v] = dist[u]+ed.cost;
                    cap[v] = min(cap[u], ed.rem());
                    pre[v] = id;
                    if (!in_queue[v]) {
                        q.push(v);
                        in_queue[v] = true;
                    }
                }
            }
        }
        if (pre[t] == -1) break;
        mxflow += cap[t];
        mncost += cap[t] * dist[t];
        for (int v = t; v != s; v = e[pre[v]].u) {
            e[pre[v]].flow += cap[t];
            e[pre[v]^1].flow -= cap[t];
        }
    }
}

void init() {
    e.clear();
    for (int i = 0; i <= N; i++) {
        g[i].clear();
    }
}

// O(V * E * 2 * log(E))
template <class type>
struct mcmf {
    struct edge { int u, v, cap, flow; type cost; };

    int n;
    vector<edge> ed;
    vector<vector<int>> g;
    vector<int> p;
    vector<type> d, phi;

    mcmf(int n) : n(n), g(n), p(n), d(n), phi(n) {}

    void add_edge(int u, int v, int cap, type cost) {
        g[u].push_back(ed.size());
        ed.push_back({u, v, cap, 0, cost});
        g[v].push_back(ed.size());
        ed.push_back({v, u, 0, 0, -cost});
    }

    bool dijkstra(int s, int t) {
        fill(d.begin(), d.end(), INF);
        fill(p.begin(), p.end(), -1);
        set<pair<type, int>> q;
        d[s] = 0;
        for(q.insert({d[s], s}); q.size();) {
            int u = (*q.begin()).second; q.erase(q.begin());
            for(auto v : g[u]) {
                auto &e = ed[v];
                type nd = d[e.u]+e.cost+phi[e.u]-phi[e.v];
                if(0 < (e.cap-e.flow) && nd < d[e.v]) {
                    q.erase({d[e.v], e.v});
                    d[e.v] = nd; p[e.v] = v;
                    q.insert({d[e.v], e.v});
                }
            }
        }
        for(int i = 0; i < n; i++) phi[i] = min(INF, phi[i]+d[i]);
        return d[t] != INF;
    }
    pair<int, type> max_flow(int s, int t) {
        type mc = 0;
        int mf = 0;
        fill(phi.begin(), phi.end(), 0);
        while(dijkstra(s, t)) {
            int flow = INF;
            for(int v = p[t]; v != -1; v = p[ ed[v].u ])
                flow = min(flow, ed[v].cap-ed[v].flow);
            for(int v = p[t]; v != -1; v = p[ ed[v].u ]) {
                edge &e1 = ed[v];
                edge &e2 = ed[v^1];
```

```
            mc += e1.cost*flow;
            e1.flow += flow;
            e2.flow -= flow;
        }
        mf += flow;
    }
    return {mf, mc};
  }
};
```

7.6 Stoer Wagner

Halla el corte mnimo en un grafo no dirigido y con pesos O(V ^ 3)

```
struct stoer_wagner {
  int n;
  vector<vector<int>> g;

  stoer_wagner(int n) : n(n), g(n, vector<int>(n)) {}

  void add_edge(int a, int b, int w) {
    g[a][b] = g[b][a] = w;
  }

  pair<int, vector<int>> min_cut() {
    vector<int> used(n);
    vector<int> cut, best_cut;
    int best_weight = -1;
    for(int p = n-1; p >= 0; --p) {
      vector<int> w = g[0];
      vector<int> added = used;
      int prv, lst = 0;
      for(int i = 0; i < p; ++i) {
        prv = lst; lst = -1;
        for(int j = 1; j < n; ++j){
          if(!added[j] && (lst == -1 || w[j] > w[lst])) lst = j;
        }
        if(i == p-1) {
          for(int j = 0; j < n; j++)
            g[prv][j] += g[lst][j];
          for(int j = 0; j < n; j++)
            g[j][prv] = g[prv][j];
          used[lst] = true;
          cut.push_back(lst);
          if(best_weight == -1 || w[lst] < best_weight) {
            best_cut = cut;
            best_weight = w[lst];
          }
        } else {
          for(int j = 0; j < n; j++)
            w[j] += g[lst][j];
          added[lst] = true;
        }
      }
    }
    return {best_weight, best_cut}; /// best_cut contains all nodes in
        the same set
  }
};
```

7.7 Weighted matching

Halla el mximo match con pesos O(V ^ 3)

```
typedef int type;
struct matching_weighted {
  int l, r;
  vector<vector<type>> c;
  matching_weighted(int l, int r) : l(l), r(r), c(l, vector<type>(r)) {
    assert(l <= r);
  }

  void add_edge(int a, int b, type cost) { c[a][b] = cost; }

  type matching() {
    vector<type> v(r), d(r); // v: potential
    vector<int> ml(l, -1), mr(r, -1); // matching pairs
    vector<int> idx(r), prev(r);
    iota(idx.begin(), idx.end(), 0);
    auto residue = [&](int i, int j) { return c[i][j]-v[j]; };
    for(int f = 0; f < l; ++f) {
      for(int j = 0; j < r; ++j) {
        d[j] = residue(f, j);
        prev[j] = f;
      }
      type w;
```

```
    int j, l;
    for (int s = 0, t = 0;;) {
        if(s == t) {
            l = s;
            w = d[ idx[t++] ];
            for(int k = t; k < r; ++k) {
                j = idx[k];
                type h = d[j];
                if (h <= w) {
                    if (h < w) t = s, w = h;
                    idx[k] = idx[t];
                    idx[t++] = j;
                }
            }
            for (int k = s; k < t; ++k) {
                j = idx[k];
                if (mr[j] < 0) goto aug;
            }
        }
        int q = idx[s++], i = mr[q];
        for (int k = t; k < r; ++k) {
            j = idx[k];
            type h = residue(i, j) - residue(i, q) + w;
            if (h < d[j]) {
                d[j] = h;
                prev[j] = i;
                if(h == w) {
                    if(mr[j] < 0) goto aug;
                    idx[k] = idx[t];
                    idx[t++] = j;
                }
            }
        }
    }
    aug:
    for (int k = 0; k < l; ++k)
        v[ idx[k] ] += d[ idx[k] ] - w;
    int i;
    do {
        mr[j] = i = prev[j];
        swap(j, ml[i]);
    } while (i != f);
}
type opt = 0;
for (int i = 0; i < l; ++i)
        opt += c[i][ml[i]]; // (i, ml[i]) is a solution
    return opt;
  }
};
```

8 7 - String

8.1 Aho Corasick (Trie)

```
El trie (o prefix tree) guarda un diccionario de strings como un arbol
    enraizado.
Aho corasick permite encontrar las ocurrencias de todos los strings del
    trie en un string s.

const int alpha = 26; //cantidad de letras del lenguaje
const char L = 'a'; //primera letra del lenguaje

struct node {
    int next[alpha], end;
    //int link, exit, cnt; //para aho corasick
    int& operator[](int i) { return next[i]; }
};

vector<node> trie = {node()};

void add_str(string &s, int id = 1) {
    int u = 0;
    for (auto ch : s) {
        int c = ch-L;
        if (!trie[u][c]) {
            trie[u][c] = trie.size();
            trie.push_back(node());
        }
        u = trie[u][c];
    }
    trie[u].end = id; //con id > 0
    //trie[u].cnt++; //para aho corasick
}

// aho corasick
void build_ac() {
    queue<int> q; q.push(0);
```

```
    while (q.size()) {
        int u = q.front(); q.pop();
        for (int c = 0; c < alpha; ++c) {
            int v = trie[u][c];
            if (!v) trie[u][c] = trie[trie[u].link][c];
            else q.push(v);
            if (!u || !v) continue;
            trie[v].link = trie[trie[u].link][c];
                    trie[v].exit = trie[trie[v].link].end ?
                        trie[v].link : trie[trie[v].link].exit;
            trie[v].cnt += trie[trie[v].link].cnt;
        }
    }
}

vector<int> cnt; //cantidad de ocurrencias en s para cada patron

void run_ac(string &s) {
    int u = 0, sz = s.size();
    for (int i = 0; i < sz; ++i) {
        int c = s[i]-L;
        while (u && !trie[u][c]) u = trie[u].link;
        u = trie[u][c];
        int x = u;
        while (x) {
            int id = trie[x].end;
            if (id) cnt[id-1]++;
            x = trie[x].exit;
        }
    }
}
```

8.2 Hashing

```
Convierte el string en un polinomio, en O(n), tal que podemos comparar
    substrings como valores numericos en O(1).
Primero llamar calc_xpow() (una unica vez) con el largo maximo de los
    strings dados.

inline int add(int a, int b, const int &mod) { return a+b >= mod ?
    a+b-mod : a+b; }
inline int sbt(int a, int b, const int &mod) { return a-b < 0 ? a-b+mod :
    a-b; }
inline int mul(int a, int b, const int &mod) { return 1ll*a*b % mod; }

const int X[] = {257, 359};
const int MOD[] = {(int)1e9+7, (int)1e9+9};
vector<int> xpow[2];

struct hashing {
    vector<int> h[2];

    hashing(string &s) {
        int n = s.size();
        for (int j = 0; j < 2; ++j) {
            h[j].resize(n+1);
            for (int i = 1; i <= n; ++i) {
                h[j][i] = add(mul(h[j][i-1], X[j], MOD[j]), s[i-1],
                    MOD[j]);
            }
        }
    }
    //Hash del substring en el rango [i, j)
    ll value(int l, int r) {
        int a = sbt(h[0][r], mul(h[0][l], xpow[0][r-l], MOD[0]), MOD[0]);
        int b = sbt(h[1][r], mul(h[1][l], xpow[1][r-l], MOD[1]), MOD[1]);
        return (ll(a)<<32) + b;
    }
};

void calc_xpow(int mxlen) {
    for (int j = 0; j < 2; ++j) {
        xpow[j].resize(mxlen+1, 1);
        for (int i = 1; i <= mxlen; ++i) {
            xpow[j][i] = mul(xpow[j][i-1], X[j], MOD[j]);
        }
    }
}
```

8.3 KMP Automaton

```
const int MAXN = 1e5 + 5, alpha = 26;
const char L = 'A';
int aut[MAXN][alpha]; //aut[i][j] = a donde vuelvo si estoy en i y pongo
    una j
```

```
void build(string &s){
    int lps = 0;
    aut[0][s[0]-L] = 1;
  int n = s.size();
    for(int i = 1; i < n+1; i++){
        for(int j = 0; j < alpha; j++) aut[i][j] = aut[lps][j];
        if(i < n){
            aut[i][s[i]-L] = i + 1;
            lps = aut[lps][s[i]-L];
        }
    }
}
```

8.4 KMP

```
Cuenta las ocurrencias del string p en el string s. Agregar Prefix
    Function.

/// O(n+m)
int kmp(string &s, string &p) {
    int n = s.size(), m = p.size(), cnt = 0;
    vector<int> pf = prefix_function(p);
    for(int i = 0, j = 0; i < n; i++) {
        while(j && s[i] != p[j]) j = pf[j-1];
        if(s[i] == p[j]) j++;
        if(j == m) {
            cnt++;
            j = pf[j-1];
        }
    }
    return cnt;
}
```

8.5 Manacher

```
Devuelve un vector p donde, para cada i, p[i] es igual al largo del
    palindromo mas largo con centro en i.
Tener en cuenta que el string debe tener el siguiente formato:
    %#s[0]#s[1]#...#s[n-1]#$ (s es el string original y n es el largo del
    string)

vector<int> manacher(string s) {
    int n = s.size();
    vector<int> p(n, 0);
    int c = 0, r = 0;
    for (int i = 1; i < n-1; i++) {
        int j = c - (i-c) ;
        if (r > i) p[i] = min(r-i , p[j]);
        while (s[i+1+p[i]] == s[i-1-p[i]])
            p[i]++;
        if (i+p[i] > r) {
            c = i;
            r = i+p[i];
        }
    }
    return p;
}
```

8.6 Minimum Expression

```
Dado un string s devuelve el indice donde comienza la rotacin
    lexicograficamente menor de s.

/// O(n)
int minimum_expression(string s) {
    s = s+s; // si no se concatena devuelve el indice del sufijo menor
    int len = s.size(), i = 0, j = 1, k = 0;
    while (i+k < len && j+k < len) {
        if (s[i+k] == s[j+k]) k++;
        else if (s[i+k] > s[j+k]) i = i+k+1, k = 0; // cambiar por < para
            maximum
        else j = j+k+1, k = 0;
        if (i == j) j++;
    }
    return min(i, j);
}
```

8.7 Palindromic tree

```
const int alfa = 26;
const char L = 'a';
```

```
struct node {
  int next[alfa], link, len;
  ll cnt;
  node(int x, int l = 0, ll c = 1): len(x), link(l), cnt(c){
    memset(next, 0, sizeof next);
  }
  int& operator[](int i) { return next[i]; }
};

struct palindromic_tree {
  vector<node> tree;
  string s;
  int n;
  int last;
  palindromic_tree(string t = ""){
    n = last = 0;
    tree.pb(node(-1));
    tree.pb(node(0));
    for(auto &c: t)add_char(c);
  }

  int getlink(int p){
    while(s[n - tree[p].len - 1] != s[n])p = tree[p].link;
    return p;
  }

  void add_char(char ch){
    s.pb(ch);
    int p = getlink(last), c = ch - L;
    if(!tree[p][c]){
      int link = getlink(tree[p].link);
      link = max(1, tree[link][c]);
      tree[p][c] = SZ(tree);
      tree.pb(node(tree[p].len + 2,link, 0));
    }
    last = tree[p][c];
    tree[last].cnt++;
    n++;
  }
};
```

8.8 Prefix Function

```
Dado un string s retorna un vector pf donde pf[i] es el largo del prefijo
    propio mas largo que tambien es sufijo de s[0] hasta s[i].

/// O(n)
vector<int> prefix_function(string &s) {
    int n = s.size();
    vector<int> pf(n);
    pf[0] = 0;
    for (int i = 1, j = 0; i < n; i++) {
        while (j && s[i] != s[j]) j = pf[j-1];
        if (s[i] == s[j]) j++;
        pf[i] = j;
    }
    return pf;
}
```

8.9 Suffix Array

```
const int MAXL = 300;

struct suffixArray {
    string s;
    int n, MX;
    vector<int> ra, tra, sa, tsa, lcp;

    suffixArray(string &_s) {
        s = _s+"$";
        n = s.size();
        MX = max(MAXL, n)+2;
        ra = tra = sa = tsa = lcp = vector<int>(n);
        build();
    }

    void radix_sort(int k) {
        vector<int> cnt(MX, 0);
        for(int i = 0; i < n; i++)
            cnt[(i+k < n) ? ra[i+k]+1 : 1]++;
        for(int i = 1; i < MX; i++)
            cnt[i] += cnt[i-1];
        for(int i = 0; i < n; i++)
            tsa[cnt[(sa[i]+k < n) ? ra[sa[i]+k] : 0]++] = sa[i];
        sa = tsa;
    }
```

```cpp
    void build() {
        for (int i = 0; i < n; i++)
            ra[i] = s[i], sa[i] = i;
        for (int k = 1, r; k < n; k <<= 1) {
            radix_sort(k);
            radix_sort(0);
            tra[sa[0]] = r = 0;
            for (int i = 1; i < n; i++) {
                if (ra[sa[i]] != ra[sa[i-1]] || ra[sa[i]+k] !=
                    ra[sa[i-1]+k]) ++r;
                tra[sa[i]] = r;
            }
            ra = tra;
            if (ra[sa[n-1]] == n-1) break;
        }
    }

    int& operator[] (int i) { return sa[i]; }

    void build_lcp() {
        lcp[0] = 0;
        for (int i = 0, k = 0; i < n; i++) {
            if (!ra[i]) continue;
            while (s[i+k] == s[sa[ra[i]-1]+k]) k++;
            lcp[ra[i]] = k;
            if (k) k--;
        }
    }
    //Longest Common Substring: construir el suffixArray s = s1 + "#" +
        s2 + "$" y m = s2.size()
    pair<int, int> lcs() {
        int mx = -1, ind = -1;
        for (int i = 1; i < n; i++) {
            if (((sa[i] < n-m-1) != (sa[i-1] < n-m-1)) && mx < lcp[i]) {
                mx = lcp[i]; ind = i;
            }
        }
        return {mx, ind};
    }
};
```

8.10 Suffix Automaton

```cpp
struct suffixAutomaton {
    struct node {
        int len, link; bool end;
        map<char, int> next;
        int cnt; ll in, out;
    };

    vector<node> sa;
    int last; ll substrs = 0;

    suffixAutomaton() {}
    suffixAutomaton(string &s) {
        sa.reserve(s.size()*2);
        last = add_node();
        sa[0].link = -1;
        sa[0].in = 1;
        for (char &c : s) add_char(c);
        for (int p = last; p; p = sa[p].link) sa[p].end = 1;
    }

    int add_node() { sa.pb({}); return sa.size()-1; }

    void add_char(char c) {
        int u = add_node(), p = last;
        sa[u].len = sa[last].len + 1;
        while (p != -1 && !sa[p].next.count(c)) {
            sa[p].next[c] = u;
            sa[u].in += sa[p].in;
            substrs += sa[p].in;
            p = sa[p].link;
        }
        if (p != -1) {
            int q = sa[p].next[c];
            if (sa[p].len + 1 != sa[q].len) {
                int clone = add_node();
                sa[clone] = sa[q];
                sa[clone].len = sa[p].len + 1;
                sa[clone].in = 0;
                sa[q].link = sa[u].link = clone;
                while (p != -1 && sa[p].next[c] == q) {
                    sa[p].next[c] = clone;
                    sa[q].in -= sa[p].in;
                    sa[clone].in += sa[p].in;
                    p = sa[p].link;
```

```
            }
        } else sa[u].link = q;
    }
    last = u;
}

void run(string &s) {
    int u = 0;
    for (int i = 0; i < s.size(); ++i) {
        while (u && !sa[u].next.count(s[i])) u = sa[u].link;
        if (sa[u].next.count(s[i])) u = sa[u].next[s[i]];
    }
}

int match_str(string &s) {
    int u = 0, n = s.size();
    for (int i = 0; i < n; ++i) {
        if (!sa[u].next.count(s[i])) return 0;
        u = sa[u].next[s[i]];
    }
    return count_occ(u);
}

int count_occ(int u) {
    if (sa[u].cnt != 0) return sa[u].cnt;
    sa[u].cnt = sa[u].end;
    for (auto &v : sa[u].next)
        sa[u].cnt += count_occ(v.S);
    return sa[u].cnt;
}

ll count_paths(int u) {
    if (sa[u].out != 0) return sa[u].out;
    for (auto &v : sa[u].next)
        sa[u].out += count_paths(v.S) + 1;
    return sa[u].out;
}

node& operator[](int i) { return sa[i]; }
};
```

8.11 Z Function

```
Dado un string s retorna un vector z donde z[i] es igual al mayor numero
    de caracteres desde s[i] que coinciden con los caracteres desde s[0]

/// O(n)
vector<int> z_function (string &s) {
    int n = s.size();
    vector<int> z(n);
    for (int i = 1, x = 0, y = 0; i < n; i++) {
        z[i] = max(0, min(z[i-x], y-i+1));
        while (i+z[i] < n && s[z[i]] == s[i+z[i]]) {
            x = i, y = i+z[i], z[i]++;
        }
    }
    return z;
}
```

9 8 - Utilities

9.1 Bits Manipulation

```
* Operaciones a nivel de bits. Si n es ll usar 1ll<< en los corrimientos.

x & 1          -> Verifica si x es impar
x & (1<<i)     -> Verifica si el i-esimo bit esta encendido
x = x | (1<<i) -> Enciende el i-esimo bit
x = x & ~(1<<i) -> Apaga el i-esimo bit
x = x ^ (1<<i) -> Invierte el i-esimo bit
x = ~x         -> Invierte todos los bits
x & -x         -> Devuelve el bit encendido mas a la derecha (potencia de
    2, no el indice)
~x & (x+1)     -> Devuelve el bit apagado mas a la derecha (potencia de
    2, no el indice)
x = x | (x+1)  -> Enciende el bit apagado mas a la derecha
x = x & (x-1)  -> Apaga el bit encendido mas a la derecha
x = x & ~y     -> Apaga en x los bits encendidos de y

* Funciones del compilador gcc. Si n es ll agregar el sufijo ll, por ej:
    __builtin_clzll(n).

__builtin_clz(x)    -> Cantidad de bits apagados por la izquierda
__builtin_ctz(x)    -> Cantidad de bits apagados por la derecha. Indice
    del bit encendido mas a la derecha
```

```
__builtin_popcount(x) -> Cantida de bits encendidos

* Logaritmo en base 2 (entero). Indice del bit encendido mas a la
    izquierda. Si x es ll usar 63 y clzll(x).
/// O(1)
int lg2(const int &x) { return 31-__builtin_clz(x); }

* Itera, con indices, los bits encendidos de una mascara.
/// O(#bits_encendidos)
for (int x = mask; x; x &= x-1) {
   int i = __builtin_ctz(x);

}

* Itera todas las submascaras de una mascara. (Iterar todas las
    submascaras de todas las mascaras es O(3^n)).
/// O(2^(#bits_encendidos))
for (int sub = mask; sub; sub = (sub-1)&mask) {

}
```

9.2 Random Integer

```
Genera un numero entero aleatorio en el rango [a, b]. Para ll usar
    "mt19937_64" y cambiar todo a ll.

mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
int rand(int a, int b) { return uniform_int_distribution<int>(a, b)(rng);
    }
```

9.3 Split String

```
Divide el string s por cada espacio ' ' y devuelve un vector<> con los
    substrings resultantes.
Para dividir el string por un caracter especifico, agregar el parametro c
    y cambiar el while.

vector<string> split(const string &s/*, char c*/) {
   vector<string> v;
   stringstream ss(s);
   string sub;
   while (ss >> sub) v.pb(sub);
   //while (getline(ss, sub, c)) v.pb(sub);
   return v;
}
```

10 9 - Tips and formulas

10.1 ASCII Table

Caracteres ASCII con sus respectivos valores numéricos.

No.	ASCII	No.	ASCII
0	NUL	16	DLE
1	SOH	17	DC1
2	STX	18	DC2
3	ETX	19	DC3
4	EOT	20	DC4
5	ENQ	21	NAK
6	ACK	22	SYN
7	BEL	23	ETB
8	BS	24	CAN
9	TAB	25	EM
10	LF	26	SUB
11	VT	27	ESC
12	FF	28	FS
13	CR	29	GS
14	SO	30	RS
15	SI	31	US

No.	ASCII	No.	ASCII
32	(space)	48	0
33	!	49	1
34	”	50	2
35	#	51	3
36	$	52	4
37	%	53	5
38	&	54	6
39	’	55	7
40	(	56	8
41	)	57	9

42	*	58	:
43	+	59	;
44	,	60	¡
45	-	61	=
46	.	62	¿
47	/	63	?

No.	**ASCII**	**No.**	**ASCII**
64	@	80	P
65	A	81	Q
66	B	82	R
67	C	83	S
68	D	84	T
69	E	85	U
70	F	86	V
71	G	87	W
72	H	88	X
73	I	89	Y
74	J	90	Z
75	K	91	[
76	L	92	\
77	M	93	]
78	N	94	ˆ
79	O	95	_

No.	**ASCII**	**No.**	**ASCII**
96	‘	112	p
97	a	113	q
98	b	114	r
99	c	115	s
100	d	116	t
101	e	117	u
102	f	118	v
103	g	119	w
104	h	120	x
105	i	121	y
106	j	122	z
107	k	123	{
108	l	124	$\vert$
109	m	125	}
110	n	126	˜
111	o	127	

10.2 Formulas

—p2.2cm—p8.2cm—

Combinación (Coeficiente Binomial) Número de subconjuntos de k elementos escogidos de un conjunto con n elementos.

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

Combinación con repetición Número de grupos formados por n elementos, partiendo de m tipos de elementos.

$$CR_m^n = \binom{m+n-1}{n} = \frac{(m+n-1)!}{n!(m-1)!}$$

Permutación Número de formas de agrupar n elementos, donde importa el orden y sin repetir elementos

$$P_n = n!$$

Permutación múltiple Elegir r elementos de n posibles con repetición

$$n^r$$

Permutación con repetición Se tienen n elementos donde el primer elemento se repite a veces , el segundo b veces , el tercero c veces, ...

$$PR_n^{a,b,c\ldots} = \frac{P_n}{a!b!c!\ldots}$$

Permutaciones sin repetición Núumero de formas de agrupar r elementos de n disponibles, sin repetir elementos

$$\frac{n!}{(n-r)!}$$

Distancia Euclideana $d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

Distancia Manhattan $d_M(P_1, P_2) = |x_2 - x_1| + |y_2 - y_1|$

Considerando r como el radio, α como el ángulo del arco o sector, y (R, r) como radio mayor y menor respectivamente.

Área $A = \pi * r^2$

Longitud $L = 2 * \pi * r$

Longitud de un arco $L = \dfrac{2 * \pi * r * \alpha}{360}$

Área sector circular	$A = \dfrac{\pi * r^2 * \alpha}{360}$
Área corona circular	$A = \pi(R^2 - r^2)$

Considerando b como la longitud de la base, h como la altura, letras minúsculas como la longitud de los lados, letras mayúsculas como los ángulos, y r como el radio de círcunferencias asociadas.

Área conociendo base y altura	$A = \dfrac{1}{2} b * h$
Área conociendo 2 lados y el ángulo que forman	$A = \dfrac{1}{2} b * a * sin(C)$
Área conociendo los 3 lados	$A = \sqrt{p(p-a)(p-b)(p-c)}$ con $p = \dfrac{a+b+c}{2}$
Área de un triángulo circunscrito a una circunferencia	$A = \dfrac{abc}{4r}$
Área de un triángulo inscrito a una circunferencia	$A = r(\dfrac{a+b+c}{2})$
Área de un triangulo equilátero	$A = \dfrac{\sqrt{3}}{4} a^2$

Considerando un triangulo rectángulo de lados a, b y c, con vértices A, B y C (cada vértice opuesto al lado cuya letra minuscula coincide con el) y un ángulo α con centro en el vertice A. a y b son catetos, c es la hipotenusa:

$sin(\alpha) = \dfrac{cateto\ opuesto}{hipotenusa} = \dfrac{a}{c}$
$cos(\alpha) = \dfrac{cateto\ adyacente}{hipotenusa} = \dfrac{b}{c}$
$tan(\alpha) = \dfrac{cateto\ opuesto}{cateto\ adyacente} = \dfrac{a}{b}$
$sec(\alpha) = \dfrac{1}{cos(\alpha)} = \dfrac{c}{b}$
$csc(\alpha) = \dfrac{1}{sin(\alpha)} = \dfrac{c}{a}$
$cot(\alpha) = \dfrac{1}{tan(\alpha)} = \dfrac{b}{a}$

Propiedad neutro	(a % b) % b = a % b
Propiedad asociativa en multiplicación	(ab) % c = ((a % c)(b % c)) % c
Propiedad asociativa en suma	(a + b) % c = ((a % c) + (b % c)) % c

Pi	$\pi = acos(-1) \approx 3.14159$
e	$e \approx 2.71828$
Número áureo	$\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61803$

10.3 Sequences

Listado de secuencias mas comunes y como hallarlas.

—p1.8cm—p8.6cm—

22cmEstrellas octangulares	0, 1, 14, 51, 124, 245, 426, 679, 1016, 1449, 1990, 2651, ...

$f(n) = n * (2 * n^2 - 1)$.

22cm Euler totient	1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6,...

$f(n)$ = Cantidad de números naturales $\leq n$ coprimos con n.

22cmNúmeros de Bell	1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, ...

Se inicia una matriz triangular con f[0][0] = f[1][0] = 1. La suma de estos dos se guarda en f[1][1] y se traslada a f[2][0]. Ahora se suman f[1][0] con f[2][0] y se guarda en f[2][1]. Luego se suman f[1][1] con f[2][1] y se guarda en f[2][2] trasladandose a f[3][0] y así sucesivamente. Los valores de la primera columna contienen la respuesta.

22cm Números de Catalán	1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, ...

$$f(n) = \frac{(2n)!}{(n+1)!n!}$$

22cmNúmeros de Fermat	3, 5, 17, 257, 65537, 4294967297, 18446744073709551617, ...

$$f(n) = 2^{(2^n)} + 1$$

22cm Números de Fibonacci	0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...

$f(0) = 0;\ f(1) = 1;\ f(n) = f(n-1) + f(n-2)$ para $n > 1$

22cm Números de Lucas	2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, ...

$f(0) = 2;\ f(1) = 1;\ f(n) = f(n-1) + f(n-2)$ para $n > 1$

22cmNúmeros de Pell	0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, 13860, ...

$f(0) = 0; f(1) = 1; f(n) = 2f(n-1) + f(n-2)$ para $n > 1$

22cm Números de Tribonacci	0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, ...

$$f(0) = f(1) = 0; f(2) = 1; f(n) = f(n-1) + f(n-2) + f(n-3) \text{ para } n > 2$$

22cmNúmeros factoriales 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...

$$f(0) = 1; f(n) = \prod_{k=1}^{n} k \text{ para } n > 0.$$

22cmNúmeros piramidales cuadrados 0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506, 650, ...

$$f(n) = \frac{n * (n+1) * (2 * n + 1)}{6}$$

22cmNúmeros primos de Mersenne 3, 7, 31, 127, 8191, 131071, 524287, 2147483647, ...

$f(n) = 2^{p(n)} - 1$ donde p representa valores primos iniciando en $p(0) = 2$.

22cmNúmeros tetraedrales 1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286, 364, 455, ...

$$f(n) = \frac{n * (n+1) * (n+2)}{6}$$

22cmNúmeros triangulares 0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, ...

$$f(n) = \frac{n(n+1)}{2}$$

22cmOEIS A000127 1, 2, 4, 8, 16, 31, 57, 99, 163, 256, 386, 562, ...

$$f(n) = \frac{(n^4 - 6n^3 + 23n^2 - 18n + 24)}{24}.$$

22cmSecuencia de Narayana 1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88, 129, ...

$f(0) = f(1) = f(2) = 1; f(n) = f(n-1) + f(n-3)$ para todo $n > 2$.

22cm Secuencia de Silvestre 2, 3, 7, 43, 1807, 3263443, 10650056950807, ...

$$f(0) = 2; f(n+1) = f(n)^2 - f(n) + 1$$

22cmSecuencia de vendedor perezoso 1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, ...

Equivale al triangular(n) + 1. Máxima número de piezas que se pueden formar al hacer n cortes a un disco.

$$f(n) = \frac{n(n+1)}{2} + 1$$

22cmSuma de los divisores de un número 1, 3, 4, 7, 6, 12, 8, 15, 13, 18, 12, 28, 14, 24, ...

Para todo $n > 1$ cuya descomposición en factores primos es $n = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$ se tiene que:

$$f(n) = \frac{p_1^{a_1+1} - 1}{p_1 - 1} * \frac{p_2^{a_2+1} - 1}{p_2 - 1} * ... * \frac{p_k^{a_k+1} - 1}{p_k - 1}$$

10.4 Time Complexities

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

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