

# Team notebook

Universidad Francisco de Paula Santander - Semillero SILUX - Lenguaje C++

May 19, 2019



## Contents

<b>1</b>	<b>1 - Input Output</b>	<b>1</b>
1.1	cin and cout . . . . .	1
1.2	scanf and printf . . . . .	1
<b>2</b>	<b>2 - Data Structures</b>	<b>1</b>
2.1	Disjoint Set . . . . .	1
2.2	Fenwick Tree . . . . .	2
2.3	Indexed Set . . . . .	2
2.4	Segment Tree (Lazy Propagation) . . . . .	2
2.5	Sparse Table . . . . .	3
2.6	Sparse table 2D . . . . .	3
<b>3</b>	<b>3 - Dynamic Programming</b>	<b>4</b>
3.1	Knapsack . . . . .	4
3.2	Longest Common Subsequence . . . . .	4
3.3	Longest Increasing Subsequence . . . . .	5
3.4	Max Range Sum . . . . .	5
3.5	Max <sub>Range</sub> <sub>2D</sub> . . . . .	5
3.6	Max <sub>range</sub> <sub>3D</sub> . . . . .	6

<b>4</b>	<b>4 - Geometry</b>	<b>6</b>
4.1	Angle . . . . .	6
4.2	Area . . . . .	7
4.3	Collinear Points . . . . .	7
4.4	Convex Hull . . . . .	7
4.5	Euclidean Distance . . . . .	8
4.6	Geometric Vector . . . . .	8
4.7	Perimeter . . . . .	8
4.8	Point in Polygon . . . . .	8
4.9	Point . . . . .	9
4.10	Sexagesimal degrees and radians . . . . .	9
<b>5</b>	<b>5 - Graph</b>	<b>9</b>
5.1	BFS . . . . .	9
5.2	Bipartite Check . . . . .	9
5.3	DFS . . . . .	10
5.4	Dijkstra . . . . .	10
5.5	Flood Fill . . . . .	11
5.6	Floyd Warshall . . . . .	11
5.7	Init . . . . .	11
5.8	Kruskal . . . . .	12
5.9	LoopCheck . . . . .	12
5.10	Lowest Common Ancestor . . . . .	13
5.11	Maxflow . . . . .	14
5.12	Prim . . . . .	15
5.13	Puentes itmos . . . . .	15
5.14	Tarjan . . . . .	16
5.15	Topological Sort . . . . .	16

<b>6</b>	<b>6 - Math</b>	<b>17</b>
6.1	Binomial Coefficient . . . . .	17
6.2	Catalan Number . . . . .	17
6.3	Euler Totient . . . . .	17
6.4	Extended Euclides . . . . .	18
6.5	FFT . . . . .	18
6.6	Fibonacci mod m . . . . .	19
6.7	Gaussian Elimination . . . . .	19
6.8	Greatest Common Divisor . . . . .	19
6.9	Linear Recurrence . . . . .	19
6.10	Lowest Common Multiple . . . . .	20
6.11	Matrix Multiplication . . . . .	20
6.12	Miller-Rabin . . . . .	20
6.13	Modular Exponentiation . . . . .	21
6.14	Modular Inverse . . . . .	21
6.15	Modular Multiplication . . . . .	21
6.16	Pisano Period . . . . .	21
6.17	Pollard Rho . . . . .	22
6.18	Prime Factorization . . . . .	22
6.19	Sieve of Eratosthenes . . . . .	22
<b>7</b>	<b>7 - String</b>	<b>22</b>
7.1	KMP's Algorithm . . . . .	22
7.2	Manacher . . . . .	23
7.3	Prefix-Function . . . . .	23
7.4	String Hashing . . . . .	23
7.5	Suffix Array Init . . . . .	23
7.6	Suffix Array Longest Common Prefix . . . . .	24
7.7	Suffix Array Longest Common Substring . . . . .	24
7.8	Suffix Array Longest Repeated Substring . . . . .	24
7.9	Suffix Array String Matching Boolean . . . . .	24
7.10	Suffix Array String Matching . . . . .	25
7.11	Suffix Automaton . . . . .	25
7.12	Trie . . . . .	26
7.13	Z-Function . . . . .	27
<b>8</b>	<b>8 - Utilities</b>	<b>27</b>
8.1	Big Integer mod m . . . . .	27
8.2	Bit Manipulation . . . . .	27
8.3	Random Integer . . . . .	28
8.4	Split String . . . . .	28

<b>9</b>	<b>9 - Tips and formulas</b>	<b>28</b>
9.1	ASCII Table . . . . .	28
9.2	Formulas . . . . .	29
9.3	Sequences . . . . .	30
9.4	Time Complexities . . . . .	31

## 1 1 - Input Output

### 1.1 cin and cout

---

\* Optimizar I/O:

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

\* Impresin de punto flotante con d decimales (ej: d = 6)

```
cout << fixed << setprecision(6) << value << endl;
```

---

### 1.2 scanf and printf

---

\* Lectura segn 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
```

\* Impresin de punto flotante con d decimales (ej: d = 6)

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

---

## 2 2 - Data Structures

### 2.1 Disjoint Set

Estructura de datos para modelar una coleccin 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.

```
const int MAX = 100005;

struct Dsu {
    int dsu[MAX], size[MAX];
    int numSets;

    Dsu(int n) {
        for(int i = 0; i <= n; i++) {
            dsu[i] = i; size[i] = 1;
        }
        numSets = n;
    }

    int find(int u) {
        return (dsu[u] == u) ? u : (dsu[u] = find(dsu[u]));
    }

    void unite(int a, int b) {
        a = find(a);
        b = find(b);
        if(a == b) return;
        if(size[a] > size[b]) swap(a,b);
        dsu[a] = b;
        size[b] += size[a];
        numSets--;
    }

    int sizeOf(int u) {
        return size[find(u)];
    }
};
```

### 2.2 Fenwick Tree

```
const int N = 100000;
int bit[N+1];

void add(int k, int val) {
    for (; k <= N; 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 idx = 0;
    for(int i = 31-__builtin_clz(N); i >= 0; --i) {
        int nidx = idx | (1 << i);
        if(nidx <= N && bit[nidx] <= val) { /// change <= to <
            val -= bit[nidx];
            idx = nidx;
        }
    }
    return idx;
}
```

### 2.3 Indexed Set

Estructura de datos basada en polticas. Funciona como un set<> pero es indexado como un array[] y cuenta con dos mtodos adicionales.  
 .find\_by\_order(k) -> Retorna un iterador al k-simo 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> indexed_set;
```

## 2.4 Segment Tree (Lazy Propagation)

Estructura de datos que permite procesar consultas por rangos y actualizaciones individuales sobre un arreglo.  
 Recibe como parametro en el constructor un arreglo de valores.  
 IMPORTANTE: Para para procesar actualizaciones por rangos se deben descomentar las lineas de Lazy Propagation.

```
struct SegmentTree {
    vector<int> st;//, lazy;
    int n, neutro = 1 << 30;

    SegmentTree(vector<int> &arr) {
        n = arr.size();
        st.assign(n << 2, 0);
        //lazy.assign(n << 2, neutro);
        build(1, 0, n - 1, arr);
    }

    int query(int i, int j) { return query(1, 0, n - 1, i, j); }
    void update(int i, int j, int val) { update(1, 0, n - 1, i, j, val); }

    int left (int p) { return p << 1; }
    int right (int p) { return (p << 1) | 1; }

    void build(int p, int L, int R, vector<int> &arr) {
        if (L == R) st[p] = arr[L];
        else {
            int m = (L+R)/2, l = left(p), r = right(p);
            build(l, L, m, arr);
            build(r, m+1, R, arr);
            st[p] = min(st[l], st[r]);
        }
    }
    /*
    void propagate(int p, int L, int R, int val) {
        if (val == neutro) return;
        st[p] = val;
        lazy[p] = neutro;
        if (L != R) {
            lazy[left(p)] = val;
            lazy[right(p)] = val;
        }
    }
    */
};
```

```
int 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 = left(p), r = right(p);
    l = query(l, L, m, i, j);
    r = query(r, m+1, R, i, j);
    return min(l, r);
}

void update(int p, int L, int R, int i, int j, int val) {
    //propagate(p, L, R, lazy[p]);
    if (i > R || j < L) return;
    if (i <= L && j >= R) st[p] = val;//propagate(p, L, R, val);
    else {
        int m = (L+R)/2, l = left(p), r = right(p);
        update(l, L, m, i, j, val);
        update(r, m+1, R, i, j, val);
        st[p] = min(st[l], st[r]);
    }
}

};
```

## 2.5 Sparse Table

```
const int MAX_N = 1000;
const int K = log2(MAX_N)+1;
int st[MAX_N][K];
int _log2[MAX_N+1];
int A[MAX_N];
int n;

void calc_log2() {
    _log2[1] = 0;
    for (int i = 2; i <= MAX_N; i++) _log2[i] = _log2[i/2] + 1;
}

void build() {
    for (int i = 0; i < n; i++) st[i][0] = A[i];
    for (int j = 1; j <= K; j++)
        for (int i = 0; i + (1 << j) <= n; i++)
            st[i][j] = min(st[i][j-1], st[i + (1 << (j - 1))][j - 1]);
}
```

```
int rmq(int i, int j) {
    int k = _log2[j-i+1];
    return min(st[i][k], st[j - (1 << k) + 1][k]);
}
```

## 2.6 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],
            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 Knapsack

Dados N articulos, cada uno con su propio valor y peso y un tamao maximo de una mochila, se debe calcular el valor maximo de los elementos que es posible llevar.

Debe seleccionarse un subconjunto de objetos, de tal manera que quepan en la mochila y representen el mayor valor posible.

```
#include <algorithm>
```

```
const int MAX_WEIGHT = 40; //Peso maximo de la mochila
const int MAX_N = 1000; //Numero maximo de objetos
int N; //Numero de objetos
int prices[MAX_N]; //precios de cada producto
int weights[MAX_N]; //pesos de cada producto
int memo[MAX_N][MAX_WEIGHT]; //tabla dp

//El metodo debe llamarse con 0 en el id, y la capacidad de la mochila en
w
int knapsack(int id, int w) {
    if (id == N || w == 0) return 0;
    if (memo[id][w] != -1) return memo[id][w];
    if (weights[id] > w) memo[id][w] = knapsack(id + 1, w);
```

```

    else memo[id][w] = max(knapsack(id + 1, w), prices[id] + knapsack(id +
        1, w - weights[id]));
    return memo[id][w];
}

```

```

//La tabla memo debe iniciar en -1
memset(memo, -1, sizeof(memo[0][0]) * MAX_N * MAX_WEIGHT);

```

### 3.2 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];
}

```

### 3.3 Longest Increasing Subsequence

Halla la longitud de la subsecuencia creciente mas larga. MAX debe definirse en el tamao limite del array, n es el tamao del array. Si se admiten valores repetidos, cambiar el < de I[mid] <= values[i] por <=

```

const int inf = 2000000000;
const int MAX = 100000;
int n;

```

```

int values[MAX + 5];
int L[MAX + 5];
int I[MAX + 5];

int lis() {
    int i, low, high, mid;
    I[0] = -inf;
    for (i = 1; i <= n; i++) I[i] = inf;
    int ans = 0;
    for(i = 0; i < n; i++) {
        low = mid = 0;
        high = ans;
        while(low <= high) {
            mid = (low + high) / 2;
            if(I[mid] < values[i]) low = mid + 1;
            else high = mid - 1;
        }
        I[low] = values[i];
        if(ans < low) ans = low;
    }
    return ans;
}

```

### 3.4 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;
}

```

### 3.5 $\text{Max}_{\text{Range}_2D}$

---

```
#include <bits/stdc++.h>

//Cambiar infinito por el mnimo valor posible
int INF = -1000000007;
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;
}
```

---

### 3.6 $\text{Max}_{range_3D}$

---

```
#include <bits/stdc++.h>

//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 = -1000000000007;
```

---

```
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;
}

```

## 4 4 - Geometry

### 4.1 Angle

Dados 3 puntos A, B, y C, determina el valor del angulo ABC (origen en B) en radianes. IMPORTANTE: Definir la estructura point y vec (Geometric Vector). Si se desea convertir a grados sexagesimales, revisar Sexagesimal degrees and radians.

```

#include <vector>
#include <cmath>

double angle(point a, point b, point c) {
    vec ba = toVector(b, a);
    vec bc = toVector(b, c);
    return acos((ba.x * bc.x + ba.y * bc.y) / sqrt((ba.x * ba.x + ba.y *
        ba.y) * (bc.x * bc.x + bc.y * bc.y)));
}

```

### 4.2 Area

Calcula el area de un polgono representado como un vector de puntos. IMPORTANTE: Definir  $P[0] = P[n-1]$  para cerrar el polgono. El algoritmo utiliza el metodo de determinante de la matriz de puntos de la figura. IMPORTANTE: Debe definirse previamente la estructura point.

```

#include <vector>
#include <cmath>

double area(vector<point> P) {

```

```

    double result = 0.0;
    for (int i = 0; i < P.size()-1; i++) {
        result += ((P[i].x * P[i + 1].y) - (P[i + 1].x * P[i].y));
    }

    return fabs(result) / 2.0;
}

```

### 4.3 Collinear Points

Determina si el punto r est en la misma linea que los puntos p y q. IMPORTANTE: Deben incluirse las estructuras point y vec.

```

double cross(vec a, vec b) {
    return a.x * b.y - a.y * b.x;
}

bool collinear(point p, point q, point r) {
    return fabs(cross(toVector(p, q), toVector(p, r))) < 1e-9;
}

```

### 4.4 Convex Hull

Retorna el polgono convexo mas pequeno que cubre (ya sea en el borde o en el interior) un set de puntos. Recibe un vector de puntos, y retorna un vector de puntos indicando el polgono resultante. Es necesario que esten definidos previamente:

Estructuras: point y vec

Mtodos : collinear, euclideanDistance, ccw (de inPolygon) y angle.

```

#include <cmath>
#include <algorithm>
#include <vector>

point pivot;
bool angleCmp(point a, point b) {
    if (collinear(pivot, a, b)) return euclideanDistance(pivot, a) <
        euclideanDistance(pivot, b);
    double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
    double d2x = b.x - pivot.x, d2y = b.y - pivot.y;

```



```

    return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
}

vector<point> convexHull(vector<point> P) {
    int i, j, n = P.size();
    if (n <= 3) {
        if (!(P[0] == P[n-1])) P.push_back(P[0]);
        return P;
    }
    int P0 = 0;
    for (i = 1; i < n; i++){
        if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
            P0 = i;
    }
    point temp = P[0]; P[0] = P[P0]; P[P0] = temp;
    pivot = P[0];
    sort(++P.begin(), P.end(), angleCmp);
    vector<point> S;
    S.push_back(P[n-1]);
    S.push_back(P[0]);
    S.push_back(P[1]);
    i = 2;
    while (i < n) {
        j = S.size()-1;
        if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);
        else S.pop_back();
    }
    return S;
}

```

## 4.5 Euclidean Distance

Halla la distancia euclideana de 2 puntos en dos dimensiones (x,y). Para usar el primer mtodo, debe definirse previamente la estructura point

```

#include <cmath>

/*Trabajando con estructuras de tipo punto*/
double euclideanDistance(point p1, point p2) {
    return hypot(p1.x - p2.x, p1.y - p2.y);
}

/*Trabajando con los valores x y y de cada punto*/

```

```

double euclideanDistance(double x1, double y1, double x2, double y2){
    return hypot(x1 - x2, y1 - y2);
}

```

## 4.6 Geometric Vector

Dados dos puntos A y B, crea el vector A->B. IMPORTANTE: Debe definirse la estructura point. Es llamado vec para no confundirlo con el vector propio de c++.

```

struct vec {
    double x, y;
    vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVector(point a, point b) {
    return vec(b.x - a.x, b.y - a.y);
}

```

## 4.7 Perimeter

Calcula el permetro de un polgono representado como un vector de puntos. IMPORTANTE: Definir P[0] = P[n-1] para cerrar el polgono. La estructura point debe estar definida, al igual que el mtodo euclideanDistance.

```

#include <vector>

double perimeter(vector<point> P) {
    double result = 0.0;
    for (int i = 0; i < P.size()-1; i++){
        result += euclideanDistance(P[i], P[i+1]);
    }
    return result;
}

```

## 4.8 Point in Polygon

---

Determina si un punto `pt` se encuentra en el polgono `P`. Este polgono se define como un vector de puntos, donde el punto 0 y `n-1` son el mismo. IMPORTANTE: Deben incluirse las estructuras `point` y `vec`, ademas del mtodo `angle`, y el mtodo `cross` que se encuentra en `Collinear Points`.

```
#include <cmath>

bool ccw(point p, point q, point r) {
    return cross(toVector(p, q), toVector(p, r)) > 0;
}

bool inPolygon(point pt, vector<point> P) {
    if (P.size() == 0) return false;
    double sum = 0;
    for (int i = 0; i < P.size()-1; i++) {
        if (ccw(pt, P[i], P[i+1])) sum += angle(P[i], pt, P[i+1]);
        else sum -= angle(P[i], pt, P[i+1]);
    }
    return fabs(fabs(sum) - 2*acos(-1.0)) < 1e-9;
}
```

---

## 4.9 Point

---

La estructura punto ser la base sobre la cual se ejecuten otros algoritmos.

```
#include <cmath>

struct point {
    double x, y;
    point() { x = y = 0.0; }
    point(double _x, double _y) : x(_x), y(_y) {}
    bool operator == (point other) const {
        return (fabs(x - other.x) < 1e-9 && (fabs(y - other.y) <
            1e-9));
    }
};
```

---

## 4.10 Sexagesimal degrees and radians

Conversiones de grados sexagesimales a radianes y viceversa.

```
#include <cmath>

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

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

---

## 5 5 - Graph

### 5.1 BFS

---

Bsqueda 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 MAX = 100005; //Cantidad maxima de nodos
vector<int> g[MAX]; //Lista de adyacencia
long long dist[MAX]; //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() {
    for(int i = 0; i <= n; i++) {
        g[i].clear();
        dist[i] = -1;
    }
}

```

## 5.2 Bipartite Check

Algoritmo para la deteccion de grafos bipartitos. Modificacion de BFS.  
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

int v, e; //vertices, arcos
const int MAX=100005; //Cantidad mxima de nodos del grafo
vector<int> ady[MAX]; //lista de Adyacencia del grafo
int color[MAX]; //Estructura auxiliar para almacenar la distancia a cada
nodo.
bool bipartite;

/*Este mtodo se llama con el indice del nodo desde el que se desea
comenzar
el recorrido.*/
void bfs(int s){
    queue<int> q;
    q.push(s);
    color[s] = 0;
    int actual, i, next;

    while( q.size() > 0 ){
        actual = q.front();
        q.pop();

        for( i = 0; i < ady[actual].size(); i++){
            next = ady[actual][i];
            if( color[next] == -1 ){
                color[next] = 1 - color[actual];
                q.push(next);
            }else if( color[next] == color[actual] ){
                bipartite = false;
                return;
            }
        }
    }
}

```

```

}
}

```

## 5.3 DFS

Bsqueda 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 informacin de los nodos dependiendo del problema.

```

const int MAX = 100005; //Cantidad maxima de nodos
vector<int> g[MAX]; //Lista de adyacencia
bitset<MAX> 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() {
    for(int i = 0; i <= n; i++) {
        g[i].clear();
        vis[i] = false;
    }
}

```

## 5.4 Dijkstra

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

```

struct Edge {
    int v;
    long long w;

    bool operator < (const Edge &b) const {

```

```

        return w > b.w; //Orden invertido
    }
};

const int MAX = 100005; //Cantidad maxima de nodos
vector<Edge> g[MAX]; //Lista de adyacencia
bitset<MAX> vis; //Marca los nodos ya visitados
long long dist[MAX]; //Almacena la distancia a cada nodo
int par[MAX]; //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 nx : g[u]) {
                int v = nx.v;
                if (!vis[v] && dist[v] > dist[u] + nx.w) {
                    dist[v] = dist[u] + nx.w;
                    par[v] = u;
                    pq.push({v, dist[v]});
                }
            }
        }
    }
}

void init() {
    for(int i = 0; i <= n; i++) {
        g[i].clear();
        dist[i] = 1ll << 62;
        vis[i] = false;
    }
}

```

## 5.5 Flood Fill

Dado un grafo implícito en una matriz, "colorea" y cuenta el tamaño de las componentes conexas.

Este método debe ser llamado con las coordenadas (i, j) donde se inicia el recorrido, busca cada carácter c1 de la componente, los reemplaza por el carácter c2 y retorna el tamaño.

```

const int tam = 1000; //Tamaño máximo 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[tam][tam]; //Matriz de caracteres
int Y, X; //Tamaño de la matriz

int floodfill(int y, int x, char c1, char c2) {
    if (y < 0 || y >= Y || x < 0 || x >= X) 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;
}

```

## 5.6 Floyd Warshall

Algoritmo para grafos que halla la distancia mínima entre cualquier par de nodos. `ady[i][j]` guardar la distancia mínima entre el nodo i y el j. SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

int v, e; //vertices, arcos
const int MAX = 505; //Cantidad máxima de nodos del grafo
int ady[505][505]; //Matriz de adyacencia del grafo

void floydWarshall(){
    int k, i, j;

    for( k = 0; k < v; k++){
        for( i = 0; i < v; i++){
            for( j = 0; j < v; j++){
                ady[i][j] = min( ady[i][j], ( ady[i][k] + ady[k][j] ) );
            }
        }
    }
}

```

---

}

## 5.7 Init

---

Mtodo para la limpieza de TODAS las estructuras de datos utilizadas en TODOS los algoritmos de grafos.

Copiar solo las necesarias, de acuerdo al algoritmo que se este utilizando.

```
#define INF 1000000000

/*Debe llamarse al iniciar cada caso de prueba luego de haber leído la
cantidad de nodos v
Limpia todas las estructuras de datos.*/
void init() {
    long long max = LLONG_MAX;
    rta = 0; //Prim
    cont = dfsRoot = rootChildren = 0; //Puentes
    bridges.clear(); //Puentes
    topoSort.clear(); //Topological Sort
    loops = false; //Loop Check
    cantSCC = 0; //Tarjan
    bipartite = true; //Bipartite Check

    for( int j = 0; j <= v; j++ ) {
        dist[j] = -1; //Distancia a cada nodo (BFS)
        dist[j] = max; //Distancia a cada nodo (Dijkstra)
        ady[j].clear(); //Lista de Adyacencia
        marked[j] = 0; //Estructura auxiliar para marcar los nodos ya
            visitados
        previous[j] = -1; //Estructura auxiliar para almacenar las rutas
        parent[j] = j; //Estructura auxiliar para DS
        dfs_num[j] = -1;
        dfs_low[j] = 0;
        itsmos[j] = 0;
        color[j] = -1; //Bipartite Check

        for(k = 0; k < v; k++) ady[j][k] = INF; //Warshall
    }
}
```

---

## 5.8 Kruskal

---

Algoritmo para hallar el arbol cobertor mnimo de un grafo no dirigido y conexo. Utiliza la tcnica de Union-Find(Conjuntos disjuntos) para detectar que aristas generan ciclos.

Requiere de la `struct` Edge.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

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

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

const int MAXN = 100005;
const int MAXE = 200005;
int n, e;
edge arcos[MAXE];
vector<pair<int, int> > ady[MAXN];

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

void kruskall() {
    sort(arcos, arcos+e, cmp);
    edge cur;
    for (int i = 0, sz = 0; i < e && sz < n-1; i++) {
        cur = arcos[i];
        if(find(cur.u) != find(cur.v)) {
            unite(cur.u, cur.v);
            ady[cur.u].push_back({cur.v, cur.w});
            ady[cur.v].push_back({cur.u, cur.w});
            sz++;
        }
    }
}
```

---

## 5.9 LoopCheck

Determina si un Grafo DIRIGIDO tiene o no ciclos.  
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```
const int MAX = 10010; //Cantidad maxima de nodos
int v; //Cantidad de Nodos del grafo
vector<int> ady[MAX]; //Estructura para almacenar el grafo
int dfs_num[MAX];
bool loops; //Bandera de ciclos en el grafo

/* DFS_NUM STATES
    2 - Explored
    3 - Visited
    -1 - Unvisited
*/

/*
Este metodo debe ser llamado desde un nodo inicial u.
Cortara su ejecucion en el momento que encuentre algun ciclo en el grafo.
*/
void graphCheck( int u ){
    int j, next;

    if( loops ) return;

    dfs_num[u] = 2;

    for(j = 0; j < ady[u].size(); j++){
        next = ady[u][j];

        if( dfs_num[next] == -1 )    graphCheck( next );
        else if( dfs_num[next] == 2 ){
            loops = true;
            break;
        }
    }

    dfs_num[u] = 3;
}

int main(){
    for( int s = 1; s <= v && !loops; s++){ //Por si el grafo es NO
        conexo
        if( dfs_num[s] == -1 ) graphCheck(s);
    }
}
```

```
}
}
```

## 5.10 Lowest Common Ancestor

Dados los nodos a y b de un arbol determina cual es el ancestro comun mas bajo entre a y b.

\*Tambien puede determinar la arista de peso maximo entre los nodos a y b  
(Para esto quitar los "//")

SE DEBE EJECUTAR EL METODO build() ANTES DE UTILIZARSE

```
//define Node pair<int, int> //(PESO, NODO adyacente)
```

```
int n, e; //Cantidad de nodos y aristas
const int MAX = 100005; //Cantidad maxima de nodos
const int LOG2 = 17; //log2(MAX)
//vector<Node> ady[MAX]; //Lista de adyacencia
vector<int> ady[MAX]; //Lista de adyacencia
int dep[MAX]; //Almacena la profundidad de cada nodo
int par[MAX][LOG2]; //Almacena los padres para responder las consultas
//int rmq[MAX][LOG2]; //Almacena los pesos para responder las consultas
```

```
int lca(int a, int b) {
    //int ans = -1;
    if (dep[a] < dep[b]) swap(a, b);
    int diff = dep[a] - dep[b];
    for (int i = LOG2-1; i >= 0; i--) {
        if (diff & (1 << i)) {
            //ans = max(ans, rmq[a][i]);
            a = par[a][i];
        }
    }
    //if (a == b) return ans;
    if (a == b) return a;
    for (int i = LOG2-1; i >= 0; i--) {
        if (par[a][i] != par[b][i]) {
            //ans = max(ans, max(rmq[a][i], rmq[b][i]));
            a = par[a][i];
            b = par[b][i];
        }
    }
    //return max(ans, max(rmq[a][0], rmq[b][0]));
    return par[a][0];
}
```

```

}

void dfs(int u, int p, int d) {
    dep[u] = d;
    par[u][0] = p;
    for (auto v /*nx*/ : ady[u]) {
        //int v = nx.second;
        if (v != p) {
            //rmq[v][0] = nx.first;
            dfs(v, u, d + 1);
        }
    }
}

void build() {
    for(int i = 0; i < n; i++) dep[i] = -1;
    for(int i = 0; i < n; i++) {
        if(dep[i] == -1) {
            //rmq[i][0] = -1;
            dfs(i, i, 0);
        }
    }
    for(int j = 0; j < LOG2-1; j++) {
        for(int i = 0; i < n; i++) {
            par[i][j+1] = par[ par[i][j] ][j];
            //rmq[i][j+1] = max(rmq[ par[i][j] ][j], rmq[i][j]);
        }
    }
}

```

## 5.11 Maxflow

Dado un grafo, halla el mximo flujo entre una fuente s y un sumidero t.  
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

vector<int> ady [105];
int capacity [105] [105]; //Capacidad de aristas de la red
int flow [105] [105]; //Flujo de cada arista
int previous [105];

void connect(int i, int j, int cap){
    ady[i].push_back(j);
    ady[j].push_back(i);

```

```

    capacity[i][j] += cap;
    //Si el grafo es dirigido no hacer esta linea
    //capacity[j][i]+=cap;
}

int maxflow(int s, int t, int n){ //s=fuente, t=sumidero, n=numero de
    nodos
    int i, j, maxFlow, u, v, extra, start, end;
    for( i = 0; i <= n; i++){
        for( j = 0; j <= n; j++){
            flow[i][j]=0;
        }
    }

    maxFlow = 0;

    while( true ){
        for( i = 0; i <= n; i++ ) previous[i] = -1;

        queue<int> q;
        q.push(s);
        previous[s] = -2;

        while( q.size() > 0 ){
            u = q.front();
            q.pop();
            if( u == t ) break;
            for( j = 0; j < ady[u].size(); j++){
                v = ady[u][j];
                if( previous[v] == -1 && capacity[u][v] - flow[u][v] > 0 ){
                    q.push(v);
                    previous[v] = u;
                }
            }
        }
        if( previous[t] == -1 ) break;

        extra = 1 << 30;
        end = t;
        while( end != s){
            start = previous[end];
            extra = min( extra, capacity[start][end]-flow[start][end] );
            end = start;
        }
    }

```

```

    end = t;
    while( end != s){
        start = previous[end];
        flow[start][end] += extra;
        flow[end][start] = -flow[start][end];
        end = start;
    }

    maxFlow += extra;
}

return maxFlow;
}

int main(){
    //Para cada arista
    connect( s, d, f); //origen, destino, flujo
}

```

## 5.12 Prim

Algoritmo para hallar el arbol cobertor mnimo de un grafo no dirigido y conexo.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

#define Node pair < int, long long > //(Vertice adyacente, peso)

int v, e; //vertices, arcos
const int MAX = 100005;
vector < Node > ady[MAX];
bool marked[MAX];
int rta;

class cmp {
public:
    bool operator()(Node n1, Node n2) {
        return (n1.second > n2.second);
    };
};

static void prim() {
    priority_queue < Node, vector < Node > , cmp > pq;
    int u, w, i, v;

```

```

    marked[0] = true;
    for (i = 0; i < ady[0].size(); i++) {
        v = ady[0][i].first;
        if ( ! marked[v]) pq.add(Node(v, ady[u][i].second));
    }

    while ( ! pq.empty()) {
        u = pq.top().first;
        w = pq.top().second;

        pq.pop();

        if ( !marked[u]) {
            rta += w;
            marked[u] = true;
            for (i = 0; i < ady[u].size(); i++) {
                v = ady[u][i].first;
                if ( ! marked[v]) pq.add(Node(v, ady[u][i].second));
            }
        }
    }
}

```

## 5.13 Puentes itmos

Algoritmo para hallar los puentes e itsmos en un grafo no dirigido. SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

vector<int> ady[1010];
int marked[1010];
int previous[1010];
int dfs_low[1010];
int dfs_num[1010];
bool itsmos[1010];
int n, e;
int dfsRoot, rootChildren, cont;
vector< pair<int,int> > bridges;

void dfs(int u){
    dfs_low[u] = dfs_num[u] = cont;
    cont++;
    marked[u] = 1;
    int j, v;

```



```

for(j = 0; j < ady[u].size(); j++){
    v = ady[u][j];
    if( marked[v] == 0 ){
        previous[v] = u;
        //para el caso especial
        if( u == dfsRoot ) rootChildren++;
        dfs(v);

        //Itsmos
        if( dfs_low[v] >= dfs_num[u] ) itsmos[u] = 1;

        //Bridges
        if( dfs_low[v] > dfs_num[u] )
            bridges.push_back(make_pair(min(u,v),max(u,v)));

        dfs_low[u] = min(dfs_low[u], dfs_low[v]);
    }else if( v != previous[u] ) dfs_low[u] = min(dfs_low[u],
        dfs_num[v]);
    }
}

int main(){
    //Antes de ejecutar el Algoritmo
    cont = dfsRoot = rootChildren = 0;
    bridges.clear();
    dfs( dfsRoot );
    /* Caso especial */
    itmos[dfsRoot] = ( itmos[ dfsRoot ] == 1 && rootChildren > 1 ) ? 1 :
        0;
}

```

## 5.14 Tarjan

Algoritmo para hallar componentes fuertemente conexas(SCC) en grafos dirigidos.

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

int v, e;
const int MAX = 5000; // Mxima cantidad de nodos
int dfs_low[MAX];
int dfs_num[MAX];
bool marked[MAX];

```

```

vector<int> s;
int dfsCont, cantSCC;
vector<int> ady[];

void tarjanSCC( int u ){
    dfs_low[u] = dfs_num[u] = dfsCont;
    dfsCont++;
    s.push_back(u);
    marked[u] = true;

    int j, v;

    for( j = 0; j < ady[u].size(); j++ ){
        v = ady[u][j] );

        if( dfs_num[v] == -1 ){
            tarjanSCC( v );
        }

        if( marked[v] ){
            dfs_low[u] = min( dfs_low[u], dfs_low[v] );
        }
    }

    if( dfs_low[u] == dfs_num[u] ){
        cantSCC++;

        /* ***** */
        /* Esta seccion se usa para imprimir las componentes conexas */
        cout << "COMPONENTE CONEXA #" << cantSCC << "\n";
        while( true ){
            v = s.back();
            s.pop_back();
            marked[v] = false;
            cout << v << "\n";
            if( u == v ) break;
        }

        /* ***** */
    }
}

int main (){
    for( int i = 0; i < v; i++ ){ //Por si el grafo no es conexo
        if( dfs_num[i] == -1 ){
            dfsCont = 0;
            s.clear();

```

```

        tarjanSCC(i);
    }
}
}

```

## 5.15 Topological Sort

Dado un grafo aciclico dirigido, 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. SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```

int n; //Cantidad de nodos del grafo
const int MAX=100005; //Cantidad mxima de nodos del grafo
vector<int> topoSort; //Ordenamiento topologico del grafo
vector<int> ady[MAX]; //Lista de adyacencia
bool marked[MAX]; //Estructura auxiliar para marcar los nodos visitados

//Recibe un nodo inicial u
void dfs( int u ){
    int i, v;
    marked[u] = 1;
    for( i = 0; i < ady[u].size(); i++){
        v = ady[u][i];
        if( !marked[v] ) dfs(v);
    }
    topoSort.push_back(u);
}

int main(){
    for(i=0; i<n; i++){
        if( !marked[i] )    dfs(i);
    }
    //imprimir topoSort en reversa :3
}

```

## 6 6 - Math

### 6.1 Binomial Coefficient

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

```

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

```

### 6.2 Catalan Number

Guarda en el array catalan[] los numeros de Catalan hasta MAX.

```

const int MAX = 30;
long long catalan[MAX+1];

void catalanNumbers() {
    catalan[0] = 1;
    for(int i = 1; i <= MAX; i++){
        catalan[i] = (long long)(catalan[i-1]*((double)(2*((2 * i)- 1))/(i
            + 1)));
    }
}

```

### 6.3 Euler Totient

La funcin totient de Euler devuelve la cantidad de enteros positivos menores o iguales a n que son coprimos con n ( $\gcd(n, i) = 1$ )

\* Dado un valor n calcula el Euler totient de n. Debe ejecutarse primero Sieve of Eratosthenes (al menos hasta un numero mayor a la raiz cuadrada de n).

```

long long totient(long long n) {
    long long tot = n;
    for (int i = 0, p = primes[i]; p*p <= n; p = primes[++i]) {
        if (n % p == 0) {
            while (n % p == 0) n /= p;
            tot -= tot / p;
        }
    }
    if (n > 1) tot -= tot / n;
    return tot;
}

* Calcular el Euler totient para todos los numeros menores o iguales a
  MAX.

const int MAX = 100;
int totient[MAX+1];
bitset<MAX+1> marked;

void sieve_totient() {
    marked[1] = true;
    for (int i = 0; i <= MAX; i++) totient[i] = i;
    for (int i = 2; i <= MAX; i++) if (!marked[i]) {
        for (int j = i; j <= MAX; j += i) {
            totient[j] -= totient[j] / i;
            marked[j] = true;
        }
        marked[i] = false;
    }
}

```

## 6.4 Extended Euclides

El algoritmo de Euclides extendido retorna el gcd(a, b) y calcula los coeficientes enteros X y Y que satisfacen la ecuacin:  $aX + bY = \text{gcd}(a, b)$ .

```

int x, y;

int extendedEuclid(int a, int b) {
    if(b == 0) { x = 1; y = 0; return a; }
    int d = extendedEuclid(b, a % b);

```

```

    int _x = x;
    x = y;
    y = _x - (a/b)*y;
    return d;
}

```

## 6.5 FFT

Estructura y mtodos para realizar FFT

```

typedef long double lf;
const lf eps = 1e-8, pi = acos(-1);

/* COMPLEX NUMBERS */
struct pt {
    lf a, b;
    pt() {}
    pt(lf a, lf b) : a(a), b(b) {}
    pt(lf a) : a(a), b(0) {}
    pt operator + (const pt &x) const { return (pt){ a + x.a, b + x.b }; }
    pt operator - (const pt &x) const { return (pt){ a - x.a, b - x.b }; }
    pt operator * (const pt &x) const { return (pt){ a * x.a - b * x.b, a
        * x.b + b * x.a }; }
};

const int MAX = 262144; // Potencia de 2 superior al polinomio c mximo (
    10^5 + 10^5)
pt a[MAX], b[MAX]; //Polinomio a, y b a operarse

void rev( pt *a, int n ){
    int i, j, k;
    for( i = 1, j = n >> 1; i < n - 1; i++ ) {
        if( i < j ) swap( a[i], a[j] );
        for( k = n >> 1; j >= k; j -= k, k >>= 1 );
        j += k;
    }
}

/* Discrete Fourier Transform */
void dft( pt *a, int n, int flag = 1 ) {
    rev( a, n );

    int m, k, j;

```

```

for( m = 2; m <= n; m <= 1) {
    pt wm = (pt){ cos( flag * 2 * pi / m ), sin( flag * 2 * pi / m ) };
    for( k = 0; k < n; k += m ) {
        pt w = (pt){ 1.0, 0.0 };
        for( j = k; j < k + (m>>1); j++, w = w * wm ) {
            pt u = a[j], v = a[j+(m>>1)] * w;
            a[j] = u + v;
            a[j + (m>>1)] = u - v;
        }
    }
}

/* n must be a power of 2 and it is the size of resultant polynomial
   values must be in real part of pt */
void mul( pt *a, pt *b, int n ) {
    int i, x;
    dft( a, n ); dft( b, n );
    for( i = 0; i < n; i++ ) a[i] = a[i] * b[i];
    dft( a, n, -1 );
    for( i = 0; i < n; i++ ) a[i].a = abs(round(a[i].a/n));
}

void init( int n ){
    int i, j;

    // Creando los polinomios
    for( i = 0, i < s.size(); i++, j-- ){
        a[i] = pt( 1.0, 0.0 );
    }

    // Se completan con 0 los polinomios al tamaño n.
    for( i = s.size() ; i < n; i++ ){
        a[i] = pt( 0.0, 0.0 );
    }
}

int get_size(int sz1, int sz2) {
    int n = 1;
    while( n <= sz1 + sz2 ) n <= 1;
    return n;
}

```

## 6.6 Fibonacci mod m

Calcula fibonacci(n) % m.

```

long long fib(long long n, long long m) {
    long long a = 0, b = 1, c;
    for (int i = log2(n); i >= 0; i--) {
        c = a;
        a = ((c%m) * (2*(b%m) - (c%m) + m)) % m;
        b = ((c%m) * (c%m) + (b%m) * (b%m)) % m;
        if ((n >> i) & 1) != 0) {
            c = (a + b) % m;
            a = b; b = c;
        }
    }
    return a;
}

```

## 6.7 Gaussian Elimination

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

```

const int MAX_N = 100;
double mat[MAX_N][MAX_N + 1];
int n;

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 temp = 0.0;

```

```

    for (int j = i + 1; j < n; j++) temp += mat[i][j]*vec[j];
    vec[i] = (mat[i][n]-temp) / mat[i][i];
}
return vec;
}

```

## 6.8 Greatest Common Divisor

Calcula el mximo comn divisor entre a y b mediante el algoritmo de Euclides.

```

int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

```

## 6.9 Linear Recurrence

Calcula el fibonacci de n como la suma de los k terminos anteriores de la secuencia (En la secuencia comn k = 2).

IMPORTANTE: Debe agregarse Matrix Multiplication.

```

int fib(long long n, int k = 2) {
    matrix F(k, 1);
    F.m[0][0] = F.m[1][0] = 1;
    for (int i = 2; i < k; i++)
        F.m[i][0] = F.m[i-1][0] * 2;
    matrix T(k, k);
    for (int i = 0; i < k; i++)
        for (int j = 0; j < k; j++)
            if (i == k-1 || i == j-1) T.m[i][j] = 1;
    F = pow(T, n) * F;
    return F.m[0][0];
}

```

## 6.10 Lowest Common Multiple

Calculo del mnimo comn mltiplo usando el mximo comn divisor. Agregar Greatest Common Divisor.

```

int lcm (int a, int b) {
    return a*b / gcd(a, b);
}

```

## 6.11 Matrix Multiplication

Estructura para realizar operaciones de multiplicacin y exponenciacin modular sobre matrices.

```

#define mod 1000000007
const int N = 2; //tamano maximo de la matriz

struct matrix {
    int m[N][N], r, c;

    matrix(int _r = N, int _c = N, bool iden = false) {
        r = _r; c = _c;
        memset(m, 0, sizeof(m));
        if (iden) while (_c--) m[_c][_c] = 1;
    }

    matrix operator * (matrix B) {
        matrix C(r, B.c);
        for(int i = 0; i < r; i++)
            for(int j = 0; j < B.c; j++)
                for(int k = 0; k < c; k++)
                    C.m[i][j] = (1ll*C.m[i][j] + 1ll*m[i][k]*B.m[k][j]) %
                        mod;
        return C;
    }
};

matrix pow(matrix &A, long long e) {
    if (e == 0) return {A.r, A.c, true};
    if (e&1) return A * pow(A, e-1);
    matrix X = pow(A, e/2);
    return X * X;
}

```

## 6.12 Miller-Rabin

---

La funcin de Miller-Rabin determina si un nmero dado es o no un nmero primo.

IMPORTANTE: Debe agregarse los mtodos de Random Integers de Utilities, Modular Exponentiation y Modular Multiplication.

```
bool isPrime(long long p) {
    if (p < 2 || (p != 2 && p % 2 == 0)) return 0;
    long long s = p - 1;
    while (s % 2 == 0) s /= 2;
    for (int i = 0; i < 5; i++) {
        long long a = rand(1, p - 1);
        long long temp = s;
        long long mod = modpow(a, temp, p);
        while (temp != p - 1 && mod != 1 && mod != p - 1) {
            mod = modmul(mod, mod, p);
            temp *= 2;
        }
        if (mod != p - 1 && temp % 2 == 0) return 0;
    }
    return 1;
}
```

---

## 6.13 Modular Exponentiation

---

Realiza la operacin  $(a^b) \% \text{mod}$ .

```
long long modpow(long long a, long long b, long long mod) {
    if (b == 0) return 1;
    if (b&1) return (a * modpow(a, b-1, mod)) % mod;
    long long c = modpow(a, b/2, mod);
    return (c*c) % mod;
}
```

---

## 6.14 Modular Inverse

---

El inverso multiplicativo modular de  $a \% \text{mod}$  es un entero  $b$  tal que  $(a*b) \% \text{mod} = 1$ . ste existe siempre y cuando  $a$  y  $\text{mod}$  sean coprimos ( $\text{gcd}(a, \text{mod}) = 1$ ).

El inverso modular de  $a$  se utiliza para calcular  $(n/a) \% \text{mod}$  como  $(n*b) \% \text{mod}$ .

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

```
int modInverse(int a, int mod) {
    int d = extendedEuclid(a, mod);
    if (d > 1) return -1;
    return (x % mod + mod) % mod;
}
```

\* Si  $\text{mod}$  es un nmero primo, se puede calcular aplicando el pequeno teorema de Fermat. Agregar Modular Exponentiation.

```
int modInverse(int a, int mod) {
    return modpow(a, mod-2, mod);
}
```

\* Calcular el inverso modular para todos los numeros menores a un valor

```
int inv[mod];

void modInverse() {
    inv[1] = 1;
    for(int i = 2; i < mod; i++)
        inv[i] = (mod - (mod/i) * inv[mod%i] % mod) % mod;
}
```

---

## 6.15 Modular Multiplication

---

Realiza la operacin  $(a*b) \% \text{mod}$  minimizando posibles desbordamientos.

```
long long modmul(long long a, long long b, long long mod) {
    if (b == 0) return 0;
    if (b&1) return (a + modmul(a, b-1, mod)) % mod;
    long long c = modmul(a, b/2, mod);
    return (c+c) % mod;
}
```

---

## 6.16 Pisano Period

Calcula el Periodo de Pisano de  $m$ , que es el periodo con el cual se repite la Sucesin de Fibonacci modulo  $m$ .  
 IMPORTANTE: Si  $m$  es primo el algoritmo funciona (considerable) para  $m < 10^6$ . Debe agregarse Modular Exponentiation (sin el modulo) y Lowest Common Multiple (para `long long`).

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

long long pisanoPrime(long long p, long long e) {
    return modpow(p, e-1) * period(p);
}

long long pisanoPeriod(long long m) {
    long long pp = 1;
    for (long long p = 2; p*p <= m; p++) {
        if (m % p == 0) {
            long long 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;
}
```

## 6.17 Pollard Rho

La funcin Rho de Pollard calcula un divisor no trivial de  $n$ .  
 IMPORTANTE: Deben agregarse Modular Multiplication y Greatest common divisor para `long long`.

```
long long pollardRho(long long n) {
    long long i = 0, k = 2, x = 3, y = 3, d;
    while (true) {
        x = (modmul(x, x, n) + n - 1) % n;
        d = gcd(abs(y - x), n);
```

```
        if (d != 1 && d != n) return d;
        if (++i == k) y = x, k <= 1;
    }
}
```

## 6.18 Prime Factorization

Guarda en factors la lista de factores primos de  $n$  de menor a mayor.  
 IMPORTANTE: Debe ejecutarse primero Sieve of Eratosthenes (al menos hasta un numero mayor a la raiz cuadrada de  $n$ ).

```
vector<int> factors;

void primeFactors(int n) {
    factors.clear();
    for (int i = 0, p = primes[i]; p*p <= n; p = primes[++i]) {
        while (n % p == 0) {
            factors.push_back(p);
            n /= p;
        }
    }
    if (n > 1) factors.push_back(n);
}
```

## 6.19 Sieve of Eratosthenes

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

```
const int MAX = 1000000;
const int SQRT = 1000;
vector<int> primes;
bitset<MAX+1> marked;

void sieve() {
    marked[1] = true;
    int i = 2;
    for (; i <= SQRT; i++) if (!marked[i]) {
        primes.push_back(i);
        for (int j = i*i; j <= MAX; j += i) marked[j] = true;
    }
```

```

    for (; i <= MAX; i++) if (!marked[i]) primes.push_back(i);
}

```

---

## 7 7 - String

### 7.1 KMP's Algorithm

Encuentra si el string pattern se encuentra en el string cadena. Debe estar definido el mtodo prefix\_function.

```

#include <vector>

bool kmp(string cadena, string pattern) {
    int n=cadena.size();
    int m=pattern.size();
    vector<int> tab=prefix_function(pattern);

    for(int i = 0, seen = 0; i < n; i++) {
        while(seen > 0 && cadena[i] != pattern[seen]) {
            seen = tab[seen-1];
        }
        if(cadena[i] == pattern[seen]) seen++;
        if(seen == m) return true;
    }
    return false;
}

```

---

### 7.2 Manacher

```

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;
}

```

---

### 7.3 Prefix-Function

Dado un string s retorna un vector lps donde lps[i] es el largo del prefijo propio ms largo que tambien es sufixo de s[0] hasta s[i]. \*Para retornar el vector de suffix\_link quitar el comentario (//).

```

vector<int> prefix_function(string s) {
    int n = s.size(), len = 0, i = 1;
    vector<int> lps(n);
    lps[0] = 0;
    while(i < n) {
        if(s[len] != s[i]) {
            if(len) len = lps[len-1];
            else lps[i++] = len;
        } else lps[i++] = ++len;
    }
    //lps.insert(lps.begin(), -1); //Para suffix_link
    return lps;
}

```

---

### 7.4 String Hashing

Estructura para realizar operaciones de hashing.

```

long long p[] = {257, 359};
long long mod[] = {1000000007, 1000000009};
long long X = 1000000010;

struct hashing {
    vector<long long> h[2], pot[2];
    int n;

    hashing(string s) {
        n = s.size();
        for (int i = 0; i < 2; ++i) {

```



```

        h[i].resize(n + 1);
        pot[i].resize(n + 1, 1);
    }
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j < 2; ++j) {
            h[j][i] = (h[j][i-1] * p[j] + s[i-1]) %
                mod[j];
            pot[j][i] = (pot[j][i-1] * p[j]) % mod[j];
        }
    }
    //Hash del substring en el rango [i, j)
    long long hashValue(int i, int j) {
        long long a = (h[0][j] - (h[0][i] * pot[0][j-i] % mod[0])
            + mod[0]) % mod[0];
        long long b = (h[1][j] - (h[1][i] * pot[1][j-i] % mod[1])
            + mod[1]) % mod[1];
        return a*X + b;
    }
};

```

## 7.5 Suffix Array Init

Crea el suffix array. Deben inicializarse las variables s (String original), N\_MAX (Mximo size que puede tener s), y n (Size del string actual).

```

string s;
const int N_MAX;
int n;
int sa[N_MAX];
int rk[N_MAX];
long long rk2[N_MAX];

bool _cmp(int i, int j) {
    return rk2[i] < rk2[j];
}

void suffixArray() {
    for (int i = 0; i < n; i++) {
        sa[i] = i; rk[i] = s[i]; rk2[i] = 0;
    }
    for (int l = 1; l < n; l <= 1) {

```

```

        for (int i = 0; i < n; i++) {
            rk2[i] = ((long long) rk[i] << 32) + (i + 1 < n ? rk[i + 1] : -1);
        }
        sort(sa, sa + n, _cmp);
        for (int i = 0; i < n; i++) {
            if (i > 0 && rk2[sa[i]] == rk2[sa[i - 1]])
                rk[sa[i]] = rk[sa[i - 1]];
            else rk[sa[i]] = i;
        }
    }
}

```

## 7.6 Suffix Array Longest Common Prefix

Calcula el array Longest Common Prefix para todo el suffix array.

IMPORTANTE: Debe haberse ejecutado primero suffixArray(), incluido en Suffix Array Init.cpp

```

int lcp[N_MAX];

void calculateLCP() {
    for (int i = 0, l = 0; i < n; i++) {
        if (rk[i] > 0) {
            int j = sa[rk[i] - 1];
            while (s[i + l] == s[j + l]) l++;
            lcp[rk[i]] = l;
            if (l > 0) l--;
        }
    }
}

```

## 7.7 Suffix Array Longest Common Substring

Busca el substring comn mas largo entre dos strings. Retorna un par, con el size del substring y uno de los indices del suffix array. Debe ejecutarse previamente suffixArray() y calculateLCP()

```

// Los substrings deben estar concatenados de la forma
// "string1#string2$", antes de ejecutar suffixArray() y calculateLCS()
// m debe almacenar el size del string2.

```

```

pair<int, int> longestCommonSubstring() {
    int i, ind = 0, lcs = -1;
    for (i = 1; i < n; i++) {
        if (((sa[i] < n - m - 1) != (sa[i - 1] < n - m - 1)) && lcp[i] >
            lcs) {
            lcs = lcp[i]; ind = i;
        }
    }
    return make_pair(lcs, ind);
}

```

## 7.8 Suffix Array Longest Repeated Substring

Retorna un par con el size y el indice del suffix array en el cual se encuentra el substring repetido mas largo. Debe ejecutarse primero suffixArray() y calculateLCP().

```

pair<int, int> longestRepeatedSubstring() {
    int ind = -1, lrs = -1;
    for(int i = 0; i < n; i++) if(lrs < lcp[i]) lrs = lcp[i], ind = i;
    return make_pair(lrs, ind);
}

```

## 7.9 Suffix Array String Matching Boolean

Busca el string p en el string s (definido en init), y retorna **true** si se encuentra, o **false** en caso contrario. Debe inicializarse m con el tamaño de p, y debe ejecutarse previamente suffixArray() de Suffix Array Init.cpp.

```

string p;
int m;

bool stringMatching() {
    if(m - 1 > n) return false;
    char * _s = new char [s.length() + 1]; strcpy (_s, s.c_str());
    char * _p = new char [p.length() + 1]; strcpy (_p, p.c_str());
    int l = 0, h = n - 1, c = 1;
    while (l <= h) {
        c = (l + h) / 2;
        int r = strncmp(_s + sa[c], _p, m - 1);

```

```

        if(r > 0) h = c - 1;
        else if(r < 0) l = c + 1;
        else return true;
    }
    return false;
}

```

## 7.10 Suffix Array String Matching

Busca el string p en el string s (definido en init), y retorna un pair con el primer y ultimo indice del suffix array que coinciden con la busqueda. Si no se encuentra, retorna (-1, -1). Debe inicializarse m con el tamaño de p, y debe ejecutarse previamente suffixArray() de Suffix Array Init.cpp.

```

string p;
int m;

pair<int, int> stringMatching() {
    if(m - 1 > n) return make_pair(-1, -1);
    char * _s = new char [s.length() + 1]; strcpy (_s, s.c_str());
    char * _p = new char [p.length() + 1]; strcpy (_p, p.c_str());
    int l = 0, h = n - 1, c = 1;
    while (l < h) {
        c = (l + h) / 2;
        if(strncmp(_s + sa[c], _p, m - 1) >= 0) h = c;
        else l = c + 1;
    }
    if (strncmp(_s + sa[l], _p, m - 1) != 0) return make_pair(-1, -1);
    pair<int, int> ans; ans.first = l;
    l = 0; h = n - 1; c = 1;
    while (l < h) {
        c = (l + h) / 2;
        if (strncmp(_s + sa[c], _p, m - 1) > 0) h = c;
        else l = c + 1;
    }
    if (strncmp(_s + sa[h], _p, m - 1) != 0) h--;
    ans.second = h;
    return ans;
}

```

## 7.11 Suffix Automaton

```
//
struct state {
    int len, link;
    long long paths_in, paths_out;
    map<char, int> next;
    bool terminal;
};

const int MAX_N = 100001;
state sa[MAX_N<<1];
int sz, last;
long long paths;
string s;

void sa_add(char c) {
    int cur = sz++;
    sa[cur] = {sa[last].len + 1, 0, 0, 0, map<char, int>(), 0};
    for (p = last; p != -1 && !sa[p].next.count(c); p = sa[p].link) {
        sa[p].next[c] = cur;
        sa[cur].paths_in += sa[p].paths_in;
        paths += sa[p].paths_in;
    }
    if (p != -1) {
        int q = sa[p].next[c];
        if (sa[p].len + 1 != sa[q].len) {
            int clone = sz++;
            sa[clone] = {sa[p].len + 1, sa[q].link, 0, 0, sa[q].next, 0};
            for (; p != -1 && sa[p].next[c] == q; p = sa[p].link) {
                sa[p].next[c] = clone;
                sa[q].paths_in -= sa[p].paths_in;
                sa[clone].paths_in += sa[p].paths_in;
            }
            sa[q].link = sa[cur].link = clone;
        } else sa[cur].link = q;
    }
    last = cur;
}

void suffixAutomaton() {
    sz = 1; last = paths = 0;
    sa[0] = {0, -1, 1, 0, map<char, int>(), 1};
    for (char c : s) sa_add(c);
}
```

```
for(int p = last; p != 0; p = sa[p].link) sa[p].terminal = 1;
}

void sa_run(string p) {
    int n = p.size();
    for (int cur = 0, i = 0; i < n; ++i) {
        if (sa[i].next.count(p[i])) cur = sa[cur].next[p[i]];
        else cur = max(sa[cur].link, 0);
    }
}

long long sa_count_paths_out(int cur) {
    if (!sa[cur].next.size()) return 0;
    if (sa[cur].paths_out != 0) return sa[cur].paths_out;
    for (auto i : sa[cur].next)
        sa[cur].paths_out += 1 + sa_count_paths_out(i.second);
    return sa[cur].paths_out;
}

int memo[MAX_N<<1];

int sa_count_ocurrences(int cur) {
    if (sa[cur].next.empty()) memo[cur] = 1;
    if (memo[cur] != -1) return memo[cur];
    memo[cur] = sa[cur].terminal;
    for (auto i : sa[cur].next)
        memo[cur] += sa_count_ocurrences(i.second);
    return memo[cur];
}

//Para retornar booleano cambiar el primer return por false y el segundo
//por true
int sa_string_matching(string p) {
    int m = p.size(), cur = 0;
    for (int i = 0; i < m; ++i) {
        if (!sa[i].next.count(p[i])) return 0;
        else cur = sa[cur].next[p[i]];
    }
    return sa_count_ocurrences(cur);
}

//Requiere contruir el automata de (s+s)
int sa_lexico_min() {
    int n = s.size()>>1, cur = 0;
    for (int i = 0; i < n; ++i) cur = (*(sa[cur].next.begin())).second;
}
```

```
    return sa[cur].len-n;
}
```

## 7.12 Trie

(Prefix tree) Estructura de datos para almacenar un diccionario de strings. Debe ejecutarse el mtodo `init_trie`. El mtodo `dfs` hace un recorrido en orden del trie.

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

```
struct node {
    int next[MAX_L];
    bool fin;
    node() {
        memset(next, -1, sizeof(next));
        fin = 0;
    }
};
```

```
vector<node> trie;
```

```
void init_trie() {
    trie.clear();
    trie.push_back(node());
}
```

```
void add_str(string s) {
    int cur = 0;
    for (auto c : s) {
        if (trie[cur].next[c-L] == -1) {
            trie[cur].next[c-L] = trie.size();
            trie.push_back(node());
        }
        cur = trie[cur].next[c-L];
    }
    trie[cur].fin = 1;
}
```

```
bool contain(string s) {
    int cur = 0;
```

```
    for (auto c : s) {
        if (trie[cur].next[c-L] == -1) return 0;
        cur = trie[cur].next[c-L];
    }
    return trie[cur].fin;
}

void dfs(int cur){
    for (int i = 0; i < MAX_L; ++i) {
        if (trie[cur].next[i] != -1) {
            //cout << (char)(i+L) << endl;
            dfs(trie[cur].next[i]);
        }
    }
}

int main() {
    init_trie();
    string s[] = {"hello", "world", "help"};
    for (auto c : s) add(c);
    return 0;
}
```

## 7.13 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]`

```
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;
}
```

## 8 8 - Utilities

### 8.1 Big Integer mod m

Calcula  $n \% m$ . Utilizar cuando  $n$  es un nmero muy muy grande.

```
int mod(string n, int m) {
    int r = 0;
    for (char c : n)
        r = (r*10 + (c-'0')) % m;
    return r;
}
```

### 8.2 Bit Manipulation

Operaciones a nivel de bits.

$n \& (n<<k)$	-> Verifica si el $k$ -esimo bit esta encendido o no
$n \mid (1<<k)$	-> Enciende el $k$ -esimo bit
$n \& \sim(1<<k)$	-> Apaga el $k$ -esimo bit
$n \sim (1<<k)$	-> Invierte el $k$ -esimo bit
$\sim n$	-> Invierte todos los bits
$n \& -n$	-> Devuelve el bit encendido mas a la derecha
$\sim n \& (n+1)$	-> Devuelve el bit apagado mas a la derecha
$n \mid (n+1)$	-> Enciende el bit apagado mas a la derecha
$n \& (n-1)$	-> Apaga el bit encendido mas a la derecha

### 8.3 Random Integer

Genera un nmero entero aleatorio en el rango  $[a, b]$ . Para `long long` usar `"mt19937_64"`.

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

### 8.4 Split String

Divide el string  $s$  por espacios. Devuelve un `vector<>` con los substrings resultantes.

\* Tambien puede dividir el string  $s$  por cada ocurrencia de un `char`  $c$ .  
(Para esto quitar los `"//"`)

```
vector<string> split(string s/*, char c*/) {
    vector<string> v;
    istringstream iss(s);
    string sub;
    while (iss >> sub) {
        //while (getline(iss, sub, c)) {
            v.push_back(sub);
        }
    }
    return v;
}
```

## 9 9 - Tips and formulas

### 9.1 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	i
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	^
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	
109	m	125	}
110	n	126	~
111	o	127	

9.2 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
$\frac{n^r}{r!}$	
Permutación con repetición	Se tienen n elementos donde el primer elemento se repite a veces , el segundo b veces , el tercero c veces, ...
$PR_n^{a,b,c,...} = \frac{P_n}{a!b!c!...}$	
Permutaciones sin repetición	Número 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, $\alpha$ como el ángulo del arco o sector, y $(R, r)$ como radio mayor y menor respectivamente. $\begin{aligned} \text{Área} \quad A &= \pi * r^2 \\ \text{Longitud} \quad L &= 2 * \pi * r \\ \text{Longitud de un arco} \quad L &= \frac{2 * \pi * r * \alpha}{360} \\ \text{Área sector circular} \quad A &= \frac{\pi * r^2 * \alpha}{360} \\ \text{Área corona circular} \quad A &= \pi(R^2 - r^2) \end{aligned}$
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 circunferencias asociadas. $\begin{aligned} \text{Área conociendo base y altura} \quad A &= \frac{1}{2} b * h \\ \text{Área conociendo 2 lados y el ángulo que forman} \quad A &= \frac{1}{2} b * a * \sin(C) \\ \text{Área conociendo los 3 lados} \quad A &= \sqrt{p(p-a)(p-b)(p-c)} \text{ con } p = \frac{a+b+c}{2} \\ \text{Área de un triángulo circunscrito a una circunferencia} \quad A &= \frac{abc}{4r} \\ \text{Área de un triángulo inscrito a una circunferencia} \quad A &= r \left( \frac{a+b+c}{2} \right) \\ \text{Área de un triangulo equilátero} \quad A &= \frac{\sqrt{3}}{4} a^2 \end{aligned}$
Considerando un triangulo rectángulo de lados $a, b$ y $c$ , con vértices $A, B$ y $C$ (cada vértice opuesto al lado cuya letra minúscula coincide con el) y un ángulo $\alpha$ con centro en el vertice $A$ . $a$ y $b$ son catetos, $c$ es la hipotenusa: $\begin{aligned} \sin(\alpha) &= \frac{\text{cateto opuesto}}{\text{hipotenusa}} = \frac{a}{c} \\ \cos(\alpha) &= \frac{\text{cateto adyacente}}{\text{hipotenusa}} = \frac{b}{c} \\ \tan(\alpha) &= \frac{\text{cateto opuesto}}{\text{cateto adyacente}} = \frac{a}{b} \\ \sec(\alpha) &= \frac{1}{\cos(\alpha)} = \frac{c}{b} \end{aligned}$

$\begin{aligned} \csc(\alpha) &= \frac{1}{\sin(\alpha)} = \frac{c}{a} \\ \cot(\alpha) &= \frac{1}{\tan(\alpha)} = \frac{b}{a} \end{aligned}$
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 $\pi = \operatorname{acos}(-1) \approx 3.14159$ $e \approx 2.71828$ Número áureo $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61803$

9.3 Sequences

Listado de secuencias mas comunes y como hallarlas.

22cmEstrellas octangulares	$\frac{-p1.8cm-p8.6cm-}{0, 1, 14, 51, 124, 245, 426, 679, 1016, 1449, 1990, 2651, \dots}$
22cm Euler totient	$\frac{f(n) = n * (2 * n^2 - 1).}{1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6, \dots}$
22cmNúmeros de Bell	$\frac{f(n) = \text{Cantidad de números naturales } \leq n \text{ coprimos con } n.}{1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, \dots}$
22cm Números de Catalán	<p>Se inicia una matriz triangular con <math>f[0][0] = f[1][0] = 1</math>. La suma de estos dos se guarda en <math>f[1][1]</math> y se traslada a <math>f[2][0]</math>. Ahora se suman <math>f[1][0]</math> con <math>f[2][0]</math> y se guarda en <math>f[2][1]</math>. Luego se suman <math>f[1][1]</math> con <math>f[2][1]</math> y se guarda en <math>f[2][2]</math> trasladandose a <math>f[3][0]</math> y así sucesivamente. Los valores de la primera columna contienen la respuesta.</p> $\frac{f(n) = \frac{(2n)!}{(n+1)!n!}}{1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots}$
22cmNúmeros de Fermat	$\frac{3, 5, 17, 257, 65537, 4294967297, 18446744073709551617, \dots}$

$f(n) = 2^{(2^n)} + 1$	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 $n > 1$	22cm Números 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, ...
$f(0) = f(1) = 0; f(2) = 1; f(n) = f(n-1) + f(n-2) + f(n-3)$ para $n > 2$	22cm Números factoriales	1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...
$f(0) = 1; f(n) = \prod_{k=1}^n k$ para $n > 0$ .	22cm Números piramidales cuadrados	0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506, 650, ...
$f(n) = \frac{n * (n+1) * (2 * n + 1)}{6}$	22cm Números primos de Mersenne	3, 7, 31, 127, 8191, 131071, 524287, 2147483647, ...
$f(n) = 2^{p(n)} - 1$ donde $p$ representa valores primos iniciando en $p(0) = 2$ .	22cm Números tetraedrales	1, 4, 10, 20, 35, 56, 84, 120, 165, 220, 286, 364, 455, ...
$f(n) = \frac{n * (n+1) * (n+2)}{6}$	22cm Números triangulares	0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, ...
$f(n) = \frac{n(n+1)}{2}$	22cm OEIS A000127	1, 2, 4, 8, 16, 31, 57, 99, 163, 256, 386, 562, ...
$f(n) = \frac{(n^4 - 6n^3 + 23n^2 - 18n + 24)}{24}$	22cm Secuencia 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, ...
--	-----------------------------	---

$f(0) = 2; f(n+1) = f(n)^2 - f(n) + 1$	22cm Secuencia 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.

$f(n) = \frac{n(n+1)}{2} + 1$	22cm Suma 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  $n = p_1^{a_1} p_2^{a_2} \dots p_k^{a_k}$  se

tiene que:

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

## 9.4 Time Complexities

Aproximación del mayor número n de datos que pueden procesarse para cada una de las complejidades algorítmicas. 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
$O(n \log_2 n)$	$10^6$
$O(n)$	$10^8$
$O(\sqrt{n})$	$10^{16}$
$O(\log_2 n)$	-
$O(1)$	-