

Resumen de algoritmos para torneos de programación

Andrés Mejía

9 de noviembre de 2012

Índice

| | | | |
|---|----------|---|-----------|
| 1. Plantilla | 2 | 5. Programación dinámica | 15 |
| 2. Teoría de números | 3 | 5.1. Longest common subsequence | 15 |
| 2.1. Big mod | 3 | 5.2. Longest increasing subsequence | 15 |
| 2.2. Criba de Eratóstenes | 3 | 5.3. Partición de troncos | 15 |
| 2.3. Divisores de un número | 4 | 6. Strings | 16 |
| 2.4. Propiedades modulares y algunas identidades | 4 | 6.1. Algoritmo de Knuth-Morris-Pratt (KMP) | 16 |
| 2.5. Inverso modular | 4 | 6.2. Algoritmo Z | 17 |
| 2.6. Teorema chino del residuo | 4 | 6.3. Algoritmo de Aho-Corasick | 17 |
| 3. Combinatoria | 5 | 6.4. Suffix array y longest common prefix | 19 |
| 3.1. Cuadro resumen | 5 | 6.5. Hashing dinámico | 21 |
| 3.2. Combinaciones, coeficientes binomiales, triángulo de Pascal | 5 | 6.6. Mínima rotación lexicográfica de una string | 22 |
| 3.3. Permutaciones con elementos indistinguibles | 5 | 6.7. Algoritmo de Manacher: Hallar las subcadenas palindrómicas más largas | 22 |
| 3.4. Desordenes, desarreglos o permutaciones completas | 5 | 7. Geometría | 23 |
| 4. Grafos | 5 | 7.1. Identidades trigonométricas | 23 |
| 4.1. Algoritmo de Dijkstra | 5 | 7.2. Triángulos | 23 |
| 4.2. Minimum spanning tree: Algoritmo de Prim | 6 | 7.2.1. Circuncentro, incentro, baricentro y ortocentro | 23 |
| 4.3. Minimum spanning tree: Algoritmo de Kruskal + Union-Find | 7 | 7.2.2. Centro del círculo que pasa por 3 puntos (circuncentro) | 24 |
| 4.4. Algoritmo de Floyd-Warshall | 7 | 7.2.3. Ley del seno y ley del coseno | 24 |
| 4.5. Algoritmo de Bellman-Ford | 8 | 7.3. Área de un polígono | 25 |
| 4.6. Puntos de articulación | 9 | 7.4. Centro de masa de un polígono | 25 |
| 4.7. LCA: Lowest Common Ancestor | 9 | 7.5. Convex hull: Graham Scan | 25 |
| 4.8. Máximo flujo: Método de Ford-Fulkerson, algoritmo de Edmonds-Karp | 10 | 7.6. Convex hull: Andrew's monotone chain | 26 |
| 4.9. Máximo flujo para grafos dispersos usando Ford-Fulkerson | 11 | 7.7. Mínima distancia entre un punto y un segmento | 27 |
| 4.10. Maximum bipartite matching | 13 | 7.8. Mínima distancia entre un punto y una recta | 27 |
| 4.10.1. Teorema de König | 13 | 7.9. Determinar si un polígono es convexo | 27 |
| 4.11. Componentes fuertemente conexas: Algoritmo de Tarjan | 14 | 7.10. Determinar si un punto está dentro de un polígono convexo | 28 |
| 4.12. 2-Satisfiability | 14 | 7.11. Determinar si un punto está dentro de un polígono cualquiera | 28 |
| | | 7.12. Hallar la intersección de dos rectas | 29 |
| | | 7.13. Hallar la intersección de dos segmentos de recta | 29 |
| | | 7.14. Determinar si dos segmentos de recta se intersectan o no | 30 |

| | |
|--|-----------|
| 7.15. Cortar un polígono convexo por una recta infinita | 31 |
| 7.16. Proyección de un vector en otro | 32 |
| 7.17. Reflejar un rayo de luz en un espejo | 32 |
| 7.18. Reflejar un punto al otro lado de una recta | 32 |
| 7.19. Hallar la intersección de un segmento y una superficie cuádrica (esfera, cono, elipsoide, hiperboloide o paraboloide) | 33 |
| 7.20. Hallar el área de la unión de varios rectángulos | 34 |
| 7.21. Hallar el volumen de la unión de varios paralelepípedos | 36 |
| 7.22. Mínima distancia entre dos puntos en la superficie de la Tierra . | 36 |
| 7.23. Distancias más cortas en 3D (punto a segmento, segmento a seg- mento y punto a triángulo) | 37 |
| 7.24. Punto más cercano de aproximación | 39 |
| 7.25. Círculo más pequeño que envuelve una lista de puntos | 39 |
| 8. Estructuras de datos | 40 |
| 8.1. Árboles de Fenwick ó Binary indexed trees | 40 |
| 8.2. Segment tree | 40 |
| 8.3. Treap | 41 |
| 8.4. Rope | 42 |
| 8.5. Range Minimum Query | 44 |
| 8.5.1. Con tabla | 44 |
| 8.5.2. Con segment tree | 45 |
| 9. Misceláneo | 46 |
| 9.1. Problema de Josephus | 46 |
| 9.2. Distancia más corta para un caballo en un tablero de ajedrez infinito | 46 |
| 9.3. Trucos con bits | 46 |
| 9.3.1. Iterar sobre los subconjuntos de una máscara | 46 |
| 9.4. El <i>parser</i> más rápido del mundo | 47 |
| 9.5. Checklist para corregir un Wrong Answer | 47 |
| 9.6. Redondeo de dobles | 48 |
| 9.6.1. Convertir un doble al entero más cercano | 48 |
| 9.6.2. Redondear un doble a cierto número de cifras de precisión | 49 |
| 10. Java | 49 |
| 10.1. Entrada desde entrada estándar | 49 |
| 10.2. Entrada desde archivo | 50 |
| 10.3. Mapas y sets | 50 |
| 10.4. Colas de prioridad | 51 |

| | |
|---|-----------|
| 11. C++ | 52 |
| 11.1. Entrada desde archivo | 52 |
| 11.2. Strings con caracteres especiales | 52 |
| 11.3. Imprimir un doble con cout con cierto número de cifras de precisión | 53 |

1. Plantilla

```

#define _GLIBCXX_DEBUG
using namespace std;
#include <algorithm>
#include <iostream>
#include <iterator>
#include <sstream>
#include <fstream>
#include <cassert>
#include <cstdlib>
#include <cstring>
#include <string>
#include <cstdio>
#include <vector>
#include <cmath>
#include <queue>
#include <stack>
#include <map>
#include <set>

template <class T> string toStr(const T &x)
{ stringstream s; s << x; return s.str(); }
template <class T> int toInt(const T &x)
{ stringstream s; s << x; int r; s >> r; return r; }

#define For(i, a, b) for (int i=(a); i<(b); ++i)
#define foreach(x, v) for (typeof (v).begin() x = (v).begin(); \
                           x != (v).end(); ++x)
#define D(x) cout << #x " = " << (x) << endl

const double EPS = 1e-9;
int cmp(double x, double y = 0, double tol = EPS){
    return( x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}

#define INPUT_FILE "problemname"

```

```
int main(){
    freopen(INPUT_FILE ".in", "r", stdin); // Read from file

    return 0;
}
```

2. Teoría de números

2.1. Big mod

```
//retorna (b^p)mod(m)
// 0 <= b,p <= 2147483647
// 1 <= m <= 46340
int bigmod(int b, int p, int m){
    int mask = 1;
    int pow2 = b % m;
    int r = 1;
    while (mask){
        if (p & mask) r = (r * pow2) % m;
        pow2 = (pow2 * pow2) % m;
        mask <<= 1;
    }
    return r;
}

// Si se cambian los int por long longs los
// valores de entrada deben cumplir:
// 0 <= b,p <= 9223372036854775807
// 1 <= m <= 3037000499
// Si se cambian por unsigned long longs:
// 0 <= b,p <= 18446744073709551615
// 1 <= m <= 4294967295

// Versión recursiva
int bigMod(int b, int p, int m){
    if(p == 0) return 1;
    if (p % 2 == 0){
        int x = bigMod(b, p / 2, m);
        return (x * x) % m;
    }
}
```

```
return ((b % m) * bigMod(b, p-1, m)) % m;
}
```

2.2. Criba de Eratóstenes

Field-testing:

- *SPOJ* - 2912 - Super Primes
- *Live Archive* - 3639 - Prime Path

Marca los números primos en un arreglo. Algunos tiempos de ejecución:

| SIZE | Tiempo (s) |
|-----------|------------|
| 100000 | 0.007 |
| 1000000 | 0.032 |
| 10000000 | 0.253 |
| 100000000 | 2.749 |

// Complejidad: $O(n)$

```
const int LIMIT = 31622779;
```

```
int sieve[LIMIT + 1]; // Inicializar con 0's.
int primes[LIMIT + 1];
```

```
int primeCount = 1;
for (int i = 2; i <= LIMIT; ++i) {
    if (!sieve[i]) {
        primes[primeCount] = i;
        sieve[i] = primeCount;
        primeCount++;
    }

    for (int j = 1; j <= sieve[i] && i * primes[j] <= LIMIT; j++){
        sieve[ i * primes[j] ] = j;
    }
}
```

```
// primes contiene la lista de primos <= LIMIT, en los índices
// 1 a primeCount.
// ¡El primer primo está en la posición 1 y no 0!
```

```
// sieve[i] contiene el índice en el arreglo primes del primo
// más pequeño que divide a i. Con esta información se puede
// saber si un número es primo o descomponerlo en primos si es
// compuesto.
```

```
// i es primo si primes[sieve[i]] == i, y compuesto si no.
```

2.3. Divisores de un número

Imprime todos los divisores de un número (en desorden) en $O(\sqrt{n})$. Hasta 4294967295 (máximo *unsigned int*) responde instantáneamente. Se puede forzar un poco más usando *unsigned long long* pero más allá de 10^{12} empieza a responder muy lento.

```
for (int i=1; i*i<=n; i++) {
    if (n%i == 0) {
        cout << i << endl;
        if (i*i<n) cout << (n/i) << endl;
    }
}
```

Si sólo se requiere contar los divisores, usando la criba de Eratóstenes se puede usar esta propiedad:

Si la descomposición prima de $n > 1$ es

$$n = p_0^{e_0} \times p_1^{e_1} \times \cdots \times p_n^{e_n}$$

entonces el número de divisores (positivos) de n es

$$(e_0 + 1) \times (e_1 + 1) \times \cdots \times (e_n + 1).$$

2.4. Propiedades modulares y algunas identidades

- $a \mid c \wedge b \mid c \wedge \gcd(a, b) = 1 \rightarrow ab \mid c$
- **Euclid's Lemma:** $a \mid bc \wedge \gcd(a, b) = 1 \rightarrow a \mid c$
- $x^n - 1 = (x - 1)(x^{n-1} + x^{n-2} + \cdots + x + 1)$

2.5. Inverso modular

El inverso modular de a mód n es un entero denotado por a^{-1} tal que $a(a^{-1}) \equiv 1 \pmod{n}$. Existe si y sólo si $\gcd(a, n) = 1$.

Si el inverso módulo n existe, la operación de dividir por a mód n se puede definir como multiplicar por el inverso.

```
// Assumes gcd(a, n) == 1 and a > 0.
// Returns v such that a * v == 1 (mod n).
long long mod_inverse(long long a, long long n) {
    long long i = n, v = 0, d = 1;
    while (a > 0) {
        long long t = i / a, x = a;
        a = i % x;
        i = x;
        x = d;
        d = v - t * x;
        v = x;
    }
    v %= n;
    if (v < 0) v += n;
    return v;
}
```

2.6. Teorema chino del residuo

Sean b_1, b_2, \dots, b_r enteros positivos primos relativos dos a dos. Llamemos $B = b_1 b_2 \cdots b_r$. El sistema de congruencias lineales

$$\begin{aligned} x &\equiv a_1 \pmod{b_1} \\ x &\equiv a_2 \pmod{b_2} \\ &\vdots \\ x &\equiv a_r \pmod{b_r} \end{aligned}$$

tiene una solución simultánea que es única módulo N .

La solución está dada por

$$a \equiv (a_1 c_1 + a_2 c_2 + \cdots + a_r c_r) \pmod{N}$$

donde $c_i = m_i(m_i^{-1} \pmod{b_i})$ y $m_i = \frac{B}{b_i}$.

3. Combinatoria

3.1. Cuadro resumen

Fórmulas para combinaciones y permutaciones:

| Tipo | ¿Se permite la repetición? | Fórmula |
|--------------------|----------------------------|-----------------------------|
| r -permutaciones | No | $\frac{n!}{(n-r)!}$ |
| r -combinaciones | No | $\frac{n!}{r!(n-r)!}$ |
| r -permutaciones | Sí | n^r |
| r -combinaciones | Sí | $\frac{(n+r-1)!}{r!(n-1)!}$ |

Tomado de *Matemática discreta y sus aplicaciones*, Kenneth Rosen, 5^{ta} edición, McGraw-Hill, página 315.

3.2. Combinaciones, coeficientes binomiales, triángulo de Pascal

Complejidad: $O(n^2)$

$$\binom{n}{k} = \begin{cases} 1 & k = 0 \\ 1 & n = k \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{en otro caso} \end{cases}$$

```
const int N = 30;
long long choose[N+1][N+1];
/* Binomial coefficients */
for (int i=0; i<=N; ++i) choose[i][0] = choose[i][i] = 1;
for (int i=1; i<=N; ++i)
    for (int j=1; j<i; ++j)
        choose[i][j] = choose[i-1][j-1] + choose[i-1][j];
.....
```

Nota: $\binom{n}{k}$ está indefinido en el código anterior si $n > k$. ¡La tabla puede estar llena con cualquier basura del compilador!

3.3. Permutaciones con elementos indistinguibles

El número de permutaciones diferentes de n objetos, donde hay n_1 objetos indistinguibles de tipo 1, n_2 objetos indistinguibles de tipo 2, ..., y n_k objetos indistinguibles de tipo k , es

$$\frac{n!}{n_1!n_2!\cdots n_k!}$$

Ejemplo: Con las letras de la palabra PROGRAMAR se pueden formar $\frac{9!}{2! \cdot 3!} = 30240$ permutaciones diferentes.

3.4. Desordenes, desarreglos o permutaciones completas

Un desarreglo es una permutación donde ningún elemento i está en la posición i -ésima. Por ejemplo, 4213 es un desarreglo de 4 elementos pero 3241 no lo es porque el 2 aparece en la posición 2.

Sea D_n el número de desarreglos de n elementos, entonces:

$$D_n = \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n-1)(D_{n-1} + D_{n-2}) & n \geq 2 \end{cases}$$

Usando el principio de inclusión-exclusión, también se puede encontrar la fórmula

$$D_n = n! \left[1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \cdots + (-1)^n \frac{1}{n!} \right] = n! \sum_{i=0}^n \frac{(-1)^i}{i!}$$

4. Grafos

4.1. Algoritmo de Dijkstra

El peso de todas las aristas debe ser no negativo.

```
// //Complejidad: O(E log V)
// ¡Si hay ciclos de peso negativo, el algoritmo se queda
// en un ciclo infinito!
// Usar Bellman-Ford en ese caso.
struct edge{
    int to, weight;
    edge() {}
    edge(int t, int w) : to(t), weight(w) {}
    bool operator < (const edge &that) const {
        return weight > that.weight;
    }
}
```

```

};

vector<edge> g[MAXNODES];
// g[i] es la lista de aristas salientes del nodo i. Cada una
// indica hacia que nodo va (to) y su peso (weight). Para
// aristas bidireccionales se deben crear 2 aristas dirigidas.

// encuentra el camino más corto entre s y todos los demás
// nodos.
int d[MAXNODES]; //d[i] = distancia más corta desde s hasta i
int p[MAXNODES]; //p[i] = predecesor de i en la ruta más corta
int dijkstra(int s, int n){
    //s = nodo inicial, n = número de nodos
    for (int i=0; i<n; ++i){
        d[i] = INT_MAX;
        p[i] = -1;
    }
    d[s] = 0;
    priority_queue<edge> q;
    q.push(edge(s, 0));
    while (!q.empty()){
        int node = q.top().to;
        int dist = q.top().weight;
        q.pop();

        if (dist > d[node]) continue;
        if (node == t){
            //dist es la distancia más corta hasta t.
            //Para reconstruir la ruta se pueden seguir
            //los p[i] hasta que sea -1.
            return dist;
        }

        for (int i=0; i<g[node].size(); ++i){
            int to = g[node][i].to;
            int w_extra = g[node][i].weight;

            if (dist + w_extra < d[to]){
                d[to] = dist + w_extra;
                p[to] = node;
                q.push(edge(to, d[to]));
            }
        }
    }
}

```

```

}
return INT_MAX;
}
.....

```

4.2. Minimum spanning tree: Algoritmo de Prim

```

//Complejidad:  $O(E \log V)$ 
//¡El grafo debe ser no dirigido!
typedef string node;
typedef pair<double, node> edge;
//edge.first = peso de la arista, edge.second = nodo al que se
//dirige
typedef map<node, vector<edge> > graph;

double prim(const graph &g){
    double total = 0.0;
    priority_queue<edge, vector<edge>, greater<edge> > q;
    q.push(edge(0.0, g.begin()->first));
    set<node> visited;
    while (q.size()){
        node u = q.top().second;
        double w = q.top().first;
        q.pop(); //!!
        if (visited.count(u)) continue;
        visited.insert(u);
        total += w;
        vector<edge> &vecinos = g[u];
        for (int i=0; i<vecinos.size(); ++i){
            node v = vecinos[i].second;
            double w_extra = vecinos[i].first;
            if (visited.count(v) == 0){
                q.push(edge(w_extra, v));
            }
        }
    }
    return total; //suma de todas las aristas del MST
}
.....

```

4.3. Minimum spanning tree: Algoritmo de Kruskal + Union-Find

```
//Complejidad:  $O(E \log V)$ 
struct edge{
    int start, end, weight;
    bool operator < (const edge &that) const {
        //Si se desea encontrar el árbol de recubrimiento de
        //máxima suma, cambiar el < por un >
        return weight < that.weight;
    }
};

////////// Empieza Union find //////////
//Complejidad:  $O(m \log n)$ , donde m es el número de operaciones
//y n es el número de objetos. En la práctica la complejidad
//es casi que  $O(m)$ .
int p[MAXNODES], rank[MAXNODES];
void make_set(int x){ p[x] = x, rank[x] = 0; }
void link(int x, int y){
    if (rank[x] > rank[y]) p[y] = x;
    else{ p[x] = y; if (rank[x] == rank[y]) rank[y]++; }
}
int find_set(int x){
    return x != p[x] ? p[x] = find_set(p[x]) : p[x];
}
void merge(int x, int y){ link(find_set(x), find_set(y)); }
////////// Termina Union find //////////

//e es un vector con todas las aristas del grafo ;El grafo
//debe ser no digirido!
long long kruskal(const vector<edge> &e){
    long long total = 0;
    sort(e.begin(), e.end());
    for (int i=0; i<=n; ++i){
        make_set(i);
    }
    for (int i=0; i<e.size(); ++i){
        int u = e[i].start, v = e[i].end, w = e[i].weight;
        if (find_set(u) != find_set(v)){
            total += w;
            merge(u, v);
        }
    }
}
```

```
    }
}
return total;
}
```

.....

4.4. Algoritmo de Floyd-Warshall

```
//Complejidad:  $O(V^3)$ 
//No funciona si hay ciclos de peso negativo
// g[i][j] = Distancia entre el nodo i y el j.
unsigned long long g[MAXNODES][MAXNODES];
void floyd(int n){
    //Llenar g antes
    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]);
            }
        }
    }
    //Acá se cumple que g[i][j] = Longitud de la ruta más corta
    //de i a j.
}

/**
 * Aplicaciones de FW.
 *
 * Maxi-min: Encontrar el camino donde la arista más pequeña del
             camino es la más grande (entre las más pequeñas)
             que el sitio más ruidoso sea el más tranquilo
             g[i][j] = max(g[i][j], min(g[i][k], g[k][j]))
 *
 * Mini-max: Encontrar el camino donde la arista más grande
             del camino es la más pequeña (entre las más
             grandes) - que el peaje más caro, sea el más
             barato.
             g[i][j] = min(g[i][j], max(g[i][k], g[k][j]))
 *
 * Ciclo Neg: Si hay un ciclo negativo, g[i][i] < 0 para algún i
 *
 * Safest Path: Maximizar la probabilidad de "sobrevivir" por
```

```

        algún camino. Los valores son porcentajes!
        g[i][j] = max(g[i][j], g[i][k] * g[k][j]);

* Clausura Transitiva: Decir si se puede llegar de un nodo
  al otro:
*
  g[i][j] = g[i][j] || (g[i][k] && g[k][j])
**/

.....

```

4.5. Algoritmo de Bellman-Ford

Si no hay ciclos de coste negativo, encuentra la distancia más corta entre un nodo y todos los demás. Si sí hay, permite saberlo. El coste de las aristas sí puede ser negativo (*Debería*, si no es así se puede usar Dijkstra o Floyd).

```

//Complejidad: O(V*E)

const int oo = 1000000000;
struct edge{
    int v, w; edge(){} edge(int v, int w) : v(v), w(w) {}
};
vector<edge> g[MAXNODES];

int d[MAXNODES];
int p[MAXNODES];
// Retorna falso si hay un ciclo de costo negativo alcanzable
// desde s. Si retorna verdadero, entonces d[i] contiene la
// distancia más corta para ir de s a i. Si se quiere
// determinar la existencia de un costo negativo que no
// necesariamente sea alcanzable desde s, se crea un nuevo
// nodo A y nuevo nodo B. Para todo nodo original u se crean
// las aristas dirigidas (A, u) con peso 1 y (u, B) con peso
// 1. Luego se corre el algoritmo de Bellman-Ford iniciando en
// A.
bool bellman(int s, int n){
    for (int i=0; i<n; ++i){
        d[i] = oo;
        p[i] = -1;
    }

    d[s] = 0;
    for (int i=0, changed = true; i<n-1 && changed; ++i){

```

```

        changed = false;
        for (int u=0; u<n; ++u){
            for (int k=0; k<g[u].size(); ++k){
                int v = g[u][k].v, w = g[u][k].w;
                if (d[u] + w < d[v]){
                    d[v] = d[u] + w;
                    p[v] = u;
                    changed = true;
                }
            }
        }
    }
}

for (int u=0; u<n; ++u){
    for (int k=0; k<g[u].size(); ++k){
        int v = g[u][k].v, w = g[u][k].w;
        if (d[u] + w < d[v]){
            //Negative weight cycle!

            //Finding the actual negative cycle. If not needed
            //return false immediately.
            vector<bool> seen(n, false);
            deque<int> cycle;
            int cur = v;
            for (; !seen[cur]; cur = p[cur]){
                seen[cur] = true;
                cycle.push_front(cur);
            }
            cycle.push_front(cur);
            //there's a negative cycle that goes from
            //cycle.front() until it reaches itself again
            printf("Negative weight cycle reachable from s:\n");
            int i = 0;
            do{
                printf("%d ", cycle[i]);
                i++;
            }while(cycle[i] != cycle[0]);
            printf("\n");
            // Negative weight cycle found

            return false;
        }
    }
}

```



```

    }
    return true;
}

```

4.6. Puntos de articulación

// Complejidad: $O(E + V)$

```

typedef string node;
typedef map<node, vector<node> > graph;
typedef char color;
const color WHITE = 0, GRAY = 1, BLACK = 2;
graph g;
map<node, color> colors;
map<node, int> d, low;

set<node> cameras; //contendrá los puntos de articulación
int timeCount;

// Uso: Para cada nodo u:
// colors[u] = WHITE, g[u] = Aristas salientes de u.
// Funciona para grafos no dirigidos.

void dfs(node v, bool isRoot = true){
    colors[v] = GRAY;
    d[v] = low[v] = ++timeCount;
    const vector<node> &neighbors = g[v];
    int count = 0;
    for (int i=0; i<neighbors.size(); ++i){
        if (colors[neighbors[i]] == WHITE){
            //(v, neighbors[i]) is a tree edge
            dfs(neighbors[i], false);
            if (!isRoot && low[neighbors[i]] >= d[v]){
                //current node is an articulation point
                cameras.insert(v);
            }
            low[v] = min(low[v], low[neighbors[i]]);
            ++count;
        }else{ //(v, neighbors[i]) is a back edge
            low[v] = min(low[v], d[neighbors[i]]);
        }
    }
}

```

```

    }
    if (isRoot && count > 1){
        //Is root and has two neighbors in the DFS-tree
        cameras.insert(v);
    }
    colors[v] = BLACK;
}

```

4.7. LCA: Lowest Common Ancestor

Field-testing:

- *Live Archive* - 4805 - Ants Colony

```

// Usage:
// Preprocessing in  $O(n \log n)$ :
//   - LCA::clear(number_of_nodes);
//   - LCA::add_edge(u, v) for every edge
//   - LCA::preprocess(number_of_nodes, root);
// Queries in  $O(\log n)$ :
//   - LCA::lca(u, v);

// Assumes the tree is connected. If it's not,
// you might want to run the algorithm on every
// component (but remember to check if the two
// nodes of a query are on the same component,
// otherwise there's no LCA).

namespace LCA {
    int tin[MAXN], tout[MAXN];
    vector<int> up[MAXN];
    vector<int> g[MAXN];
    // alternatively, you can use a global graph to save space.
    // This graph is only used inside LCA::dfs below.

    int L, timer;

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

    void add_edge(int u, int v) {

```

```

        g[u].push_back(v);
        g[v].push_back(u);
    }
    void dfs(int v, int p) {
        tin[v] = ++timer;
        up[v][0] = p;
        for (int i = 1; i <= L; ++i)
            up[v][i] = up[up[v][i-1]][i-1];
        for (int i = 0; i < g[v].size(); ++i) {
            int to = g[v][i];
            if (to != p) dfs(to, v);
        }
        tout[v] = ++timer;
    }
    inline bool upper(int a, int b) {
        return tin[a] <= tin[b] && tout[b] <= tout[a];
    }
    int lca(int a, int b) {
        if (upper(a, b)) return a;
        if (upper(b, a)) return b;
        for (int i = L; i >= 0; --i)
            if (!upper(up[a][i], b))
                a = up[a][i];
        return up[a][0];
    }
    void preprocess(int n, int root) {
        L = 1;
        while ((1 << L) <= n) ++L;
        for (int i = 0; i < n; ++i) up[i].resize(L + 1);
        timer = 0;
        dfs(root, root);
    }
};

```

4.8. Máximo flujo: Método de Ford-Fulkerson, algoritmo de Edmonds-Karp

El algoritmo de Edmonds-Karp es una modificación al método de Ford-Fulkerson. Este último utiliza DFS para hallar un camino de aumentación, pero la sugerencia de Edmonds-Karp es utilizar BFS que lo hace más eficiente en algunos grafos.

```

/*
    cap[i][j] = Capacidad de la arista (i, j).
    prev[i] = Predecesor del nodo i en un camino de aumentación.
*/
int cap[MAXN+1][MAXN+1], prev[MAXN+1];

vector<int> g[MAXN+1]; //Vecinos de cada nodo.
inline void link(int u, int v, int c)
{ cap[u][v] = c; g[u].push_back(v), g[v].push_back(u); }
/*
    Notar que link crea las aristas (u, v) && (v, u) en el grafo
    g. Esto es necesario porque el algoritmo de Edmonds-Karp
    necesita mirar el "back-edge" (j, i) que se crea al bombear
    flujo a través de (i, j). Sin embargo, no modifica
    cap[v][u], porque se asume que el grafo es dirigido. Si es
    no-dirigido, hacer cap[u][v] = cap[v][u] = c.
*/

/*
    Método 1:

    Mantener la red residual, donde residual[i][j] = cuánto
    flujo extra puedo inyectar a través de la arista (i, j).

    Si empujo k unidades de i a j, entonces residual[i][j] -= k
    y residual[j][i] += k (Puedo "desempujar" las k unidades de
    j a i).

    Se puede modificar para que no utilice extra memoria en la
    tabla residual, sino que modifique directamente la tabla
    cap.
*/

int residual[MAXN+1][MAXN+1];
int fordFulkerson(int n, int s, int t){
    memcpy(residual, cap, sizeof cap);

    int ans = 0;
    while (true){
        fill(prev, prev+n, -1);
        queue<int> q;
        q.push(s);

```

```

while (q.size() && prev[t] == -1){
    int u = q.front();
    q.pop();
    vector<int> &out = g[u];
    for (int k = 0, m = out.size(); k<m; ++k){
        int v = out[k];
        if (v != s && prev[v] == -1 && residual[u][v] > 0)
            prev[v] = u, q.push(v);
    }
}

if (prev[t] == -1) break;

int bottleneck = INT_MAX;
for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
    bottleneck = min(bottleneck, residual[u][v]);
}
for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
    residual[u][v] -= bottleneck;
    residual[v][u] += bottleneck;
}
ans += bottleneck;
}
return ans;
}

```

/*
Método 2:

Mantener la red de flujos, donde $\text{flow}[i][j]$ = Flujo que, err, fluye de i a j . Notar que $\text{flow}[i][j]$ puede ser negativo. Si esto pasa, es lo equivalente a decir que i "absorbe" flujo de j , o lo que es lo mismo, que hay flujo positivo de j a i .

En cualquier momento se cumple la propiedad de skew symmetry, es decir, $\text{flow}[i][j] = -\text{flow}[j][i]$. El flujo neto de i a j es entonces $\text{flow}[i][j]$.

*/

```
int flow[MAXN+1][MAXN+1];
```

```

int fordFulkerson(int n, int s, int t){
    //memset(flow, 0, sizeof flow);
    for (int i=0; i<n; ++i) fill(flow[i], flow[i]+n, 0);
    int ans = 0;
    while (true){
        fill(prev, prev+n, -1);
        queue<int> q;
        q.push(s);
        while (q.size() && prev[t] == -1){
            int u = q.front();
            q.pop();
            vector<int> &out = g[u];
            for (int k = 0, m = out.size(); k<m; ++k){
                int v = out[k];
                if (v != s && prev[v] == -1 && cap[u][v] > flow[u][v])
                    prev[v] = u, q.push(v);
            }
        }

        if (prev[t] == -1) break;

        int bottleneck = INT_MAX;
        for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
            bottleneck = min(bottleneck, cap[u][v] - flow[u][v]);
        }
        for (int v = t, u = prev[v]; u != -1; v = u, u = prev[v]){
            flow[u][v] += bottleneck;
            flow[v][u] = -flow[u][v];
        }
        ans += bottleneck;
    }
    return ans;
}

```

.....

4.9. Máximo flujo para grafos dispersos usando Ford-Fulkerson

Field-testing:

- UVa - 563 - Crimewave

```

////////// Maximum flow for sparse graphs //////////
////////// Complexity:  $O(V * E^2)$  //////////

/*
Usage:
Call initialize_max_flow();
Create graph using add_edge(u, v, c);
max_flow(source, sink);

WARNING: The algorithm writes on the cap array. The capacity
is not the same after having run the algorithm. If you need
to run the algorithm several times on the same graph, backup
the cap array.
*/

namespace Flow {
    // Maximum number of nodes
    const int MAXN = 100;
    // Maximum number of edges
    // IMPORTANT: Remember to consider the backedges. For
    // every edge we actually need two! That's why we have
    // to multiply by two at the end.
    const int MAXE = MAXN * (MAXN + 1) / 2 * 2;
    const int oo = INT_MAX / 4;
    int first[MAXN], next[MAXE], adj[MAXE], cap[MAXE];
    int current_edge;

    /*
    Builds a directed edge (u, v) with capacity c.
    Note that actually two edges are added, the edge
    and its complementary edge for the backflow.
    */
    int add_edge(int u, int v, int c){
        adj[current_edge] = v;
        cap[current_edge] = c;
        next[current_edge] = first[u];
        first[u] = current_edge++;

        adj[current_edge] = u;
        cap[current_edge] = 0;
        next[current_edge] = first[v];
        first[v] = current_edge++;
    }
}

```

```

void initialize_max_flow(){
    current_edge = 0;
    memset(next, -1, sizeof next);
    memset(first, -1, sizeof first);
}

int q[MAXN];
int incr[MAXN];
int arrived_by[MAXN];
//arrived_by[i] = The last edge used to reach node i
int find_augmenting_path(int src, int snk){
    /*
    Make a BFS to find an augmenting path from the source
    to the sink. Then pump flow through this path, and
    return the amount that was pumped.
    */
    memset(arrived_by, -1, sizeof arrived_by);
    int h = 0, t = 0;
    q[t++] = src;
    arrived_by[src] = -2;
    incr[src] = oo;
    while (h < t && arrived_by[snk] == -1){ //BFS
        int u = q[h++];
        for (int e = first[u]; e != -1; e = next[e]){
            int v = adj[e];
            if (arrived_by[v] == -1 && cap[e] > 0){
                arrived_by[v] = e;
                incr[v] = min(incr[u], cap[e]);
                q[t++] = v;
            }
        }
    }

    if (arrived_by[snk] == -1) return 0;

    int cur = snk;
    int neck = incr[snk];
    while (cur != src){
        //Remove capacity from the edge used to reach
        //node "cur", and add capacity to the backedge
        cap[arrived_by[cur]] -= neck;
        cap[arrived_by[cur] ^ 1] += neck;
    }
}

```

```

        //move backwards in the path
        cur = adj[arrived_by[cur] ^ 1];
    }
    return neck;
}

int max_flow(int src, int snk){
    int ans = 0, neck;
    while ((neck = find_augmenting_path(src, snk)) != 0){
        ans += neck;
    }
    return ans;
}
}

```

4.10. Maximum bipartite matching

```

// Maximum Bipartite Matching
// Complexity: O(VE)

// Finds a maximum bipartite matching for two sets of
// size L and R.

// How to use:
// Set g[i][j] to true if element i of the Left set can
// be paired with element j of the Right set.
// Fill the table for all 0 <= i < L and 0 <= j < R.

// matchL[i] will contain i's match in the Right set
// and matchR[j] will contain j's match in the Left set.

bool g[MAXN][MAXN], seen[MAXN];
int L, R, matchL[MAXN], matchR[MAXN];

bool assign(int i) {
    for (int j = 0; j < R; ++j) if (g[i][j] and !seen[j]) {
        seen[j] = true;
        if (matchR[j] < 0 or assign(matchR[j]))
            return matchL[i] = j, matchR[j] = i, true;
    }
    return false;
}

```

```

}

int maxBipartiteMatching() {
    for (int i = 0; i < L; ++i) matchL[i] = -1;
    for (int j = 0; j < R; ++j) matchR[j] = -1;
    int ans = 0;
    for (int i = 0; i < L; ++i) {
        for (int j = 0; j < R; ++j) seen[j] = false; // or memset
        if (assign(i)) ans++;
    }
    return ans;
}

```

4.10.1. Teorema de Kőnig

Un **minimum vertex cover** es un subconjunto de vértices lo más pequeño posible tal que cualquier arista del grafo toque algún vértice del subconjunto.

Teorema de Kőnig. En un grafo bipartito, el tamaño del maximum bipartite matching es igual al tamaño del minimum vertex cover.

Para encontrar los nodos que componen el minimum vertex cover, hacer un DFS como el que se usa para encontrar el maximum bipartite matching¹ pero empezando únicamente en los vértices de L que no tienen pareja en el lado R (empezando en los i tales que `matchL[i] == -1`). Adicionalmente, marcar cuales nodos fueron visitados tanto en L como en R.

El minimum vertex cover estará formado por los nodos de L que no fueron visitados y los nodos de R que sí fueron visitados.

```

#include <bitset>
int L, R, int matchL[MAXN], matchR[MAXN];
bitset<MAXN> seenL, seenR;
vector<int> g[MAXN];

void dfs(int u) { // u is on L
    if (u == -1 or seenL[u]) return;
    seenL[u] = true;
    foreach(out, g[u]) {
        int v = *out;
        if (!seenR[v]) {

```

¹Es decir, un DFS que usa aristas que no están en el matching cuando va de L a R y aristas que sí están en el matching cuando va de R a L.

```

        seenR[v] = true;
        dfs(matchR[v]);
    }
}

// Build the maximum bipartite matching first!
void minimumVertexCover() {
    int size = 0;
    for (int i = 0; i < L; ++i) {
        if (matchL[i] == -1) dfs(i);
        else size++;
    }
    // size == size of the minimum vertex cover.
    // The actual minimum vertex cover is formed by:
    //   + nodes in L such that seenL[i] == false
    //   + nodes in R such that seenR[i] == true
}

```

4.11. Componentes fuertemente conexas: Algoritmo de Tarjan

```

/* Complexity:  $O(E + V)$ 
Tarjan's algorithm for finding strongly connected
components.

*d[i] = Discovery time of node i. (Initialize to -1)
*low[i] = Lowest discovery time reachable from node
i. (Doesn't need to be initialized)
*scc[i] = Strongly connected component of node i. (Doesn't
need to be initialized)
*s = Stack used by the algorithm (Initialize to an empty
stack)
*stacked[i] = True if i was pushed into s. (Initialize to
false)
*ticks = Clock used for discovery times (Initialize to 0)
*current_scc = ID of the current_scc being discovered
(Initialize to 0)
*/
vector<int> g[MAXN];
int d[MAXN], low[MAXN], scc[MAXN];

```

```

bool stacked[MAXN];
stack<int> s;
int ticks, current_scc;
void tarjan(int u){
    d[u] = low[u] = ticks++;
    s.push(u);
    stacked[u] = true;
    const vector<int> &out = g[u];
    for (int k=0, m=out.size(); k<m; ++k){
        const int &v = out[k];
        if (d[v] == -1){
            tarjan(v);
            low[u] = min(low[u], low[v]);
        }else if (stacked[v]){
            low[u] = min(low[u], low[v]);
        }
    }
    if (d[u] == low[u]){
        int v;
        do{
            v = s.top();
            s.pop();
            stacked[v] = false;
            scc[v] = current_scc;
        }while (u != v);
        current_scc++;
    }
}

```

4.12. 2-Satisfiability

Dada una ecuación lógica de conjunciones de disyunciones de 2 términos, se pretende decidir si existen valores de verdad que puedan asignarse a las variables para hacer cierta la ecuación.

Por ejemplo, $(b_1 \vee \neg b_2) \wedge (b_2 \vee b_3) \wedge (\neg b_1 \vee \neg b_2)$ es verdadero cuando b_1 y b_3 son verdaderos y b_2 es falso.

Solución: Se sabe que $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$. Entonces se traduce cada disyunción en una implicación y se crea un grafo donde los nodos son cada variable y su negación. Cada implicación es una arista en este grafo. Existe solución si nunca se cumple que una variable y su negación están en la misma componenete fuertemente conexas (Se usa el algoritmo de Tarjan, 4.11).

5. Programación dinámica

5.1. Longest common subsequence

```
#define MAX(a,b) ((a>b)?(a):(b))
int dp[1001][1001];

int lcs(const string &s, const string &t){
    int m = s.size(), n = t.size();
    if (m == 0 || n == 0) return 0;
    for (int i=0; i<=m; ++i)
        dp[i][0] = 0;
    for (int j=1; j<=n; ++j)
        dp[0][j] = 0;
    for (int i=0; i<m; ++i)
        for (int j=0; j<n; ++j)
            if (s[i] == t[j])
                dp[i+1][j+1] = dp[i][j]+1;
            else
                dp[i+1][j+1] = MAX(dp[i+1][j], dp[i][j+1]);
    return dp[m][n];
}
.....
```

5.2. Longest increasing subsequence

```
// Longest Increasing Subsequence.
// O(n log n)

const int INF = 1 << 30 - 1;

int main(){
    int n;
    while(scanf("%d", &n) == 1){
        vector<long> S(n);
        vector<long> M(n+1, INF);
        for (int i = 0; i < n; ++i) scanf("%ld", &S[i]);
        M[0] = 0;
        int m = 0;
        for (int i = 0; i < S.size(); ++i){
            int d = upper_bound(M.begin(), M.begin()+n, S[i])-M.begin();
            if (S[i] != M[d-1]){
                M[d] = S[i];
            }
        }
    }
}
.....
```

```
        m = max(m,d);
        parent[S[i]] = M[d-1];
    }
}
printf("%d\n", max(1, m));
}
return 0;
}
```

5.3. Partición de troncos

Este problema es similar al problema de *Matrix Chain Multiplication*. Se tiene un tronco de longitud n , y m puntos de corte en el tronco. Se puede hacer un corte a la vez, cuyo costo es igual a la longitud del tronco. ¿Cuál es el mínimo costo para partir todo el tronco en pedacitos individuales?

Ejemplo: Se tiene un tronco de longitud 10. Los puntos de corte son 2, 4, y 7.

El mínimo costo para partirlo es 20, y se obtiene así:

- Partir el tronco (0, 10) por 4. Vale 10 y quedan los troncos (0, 4) y (4, 10).
- Partir el tronco (0, 4) por 2. Vale 4 y quedan los troncos (0, 2), (2, 4) y (4, 10).
- No hay que partir el tronco (0, 2).
- No hay que partir el tronco (2, 4).
- Partir el tronco (4, 10) por 7. Vale 6 y quedan los troncos (4, 7) y (7, 10).
- No hay que partir el tronco (4, 7).
- No hay que partir el tronco (7, 10).
- El costo total es $10 + 4 + 6 = 20$.

El algoritmo es $O(n^3)$, pero optimizable a $O(n^2)$ con una tabla adicional:

```
/*
    O(n^3)
    dp[i][j] = Mínimo costo de partir la cadena entre las
    particiones i e j, ambas incluidas.
*/
int dp[1005][1005];
int p[1005];
```

```

int cubic(){
    int n, m;
    while (scanf("%d %d", &n, &m)==2){
        p[0] = 0;
        for (int i=1; i<=m; ++i){
            scanf("%d", &p[i]);
        }
        p[m+1] = n;
        m += 2;

        for (int i=0; i<m; ++i){
            dp[i][i+1] = 0;
        }

        for (int i=m-2; i>=0; --i){
            for (int j=i+2; j<m; ++j){
                dp[i][j] = p[j]-p[i];
                int t = INT_MAX;
                for (int k=i+1; k<j; ++k){
                    t = min(t, dp[i][k] + dp[k][j]);
                }
                dp[i][j] += t;
            }
        }

        printf("%d\n", dp[0][m-1]);
    }
    return 0;
}

/*
    0(n^2)

    dp[i][j] = Mínimo costo de partir la cadena entre las
    particiones i e j, ambas incluidas. pivot[i][j] = Índice de
    la partición que usé para lograr dp[i][j].
*/
int dp[1005][1005], pivot[1005][1005];
int p[1005];

int quadratic(){

```

```

int n, m;
while (scanf("%d %d", &n, &m)==2){
    p[0] = 0;
    for (int i=1; i<=m; ++i){
        scanf("%d", &p[i]);
    }
    p[m+1] = n;
    m += 2;

    for (int i=0; i<m-1; ++i){
        dp[i][i+1] = 0;
    }
    for (int i=0; i<m-2; ++i){
        dp[i][i+2] = p[i+2] - p[i];
        pivot[i][i+2] = i+1;
    }

    for (int d=3; d<m; ++d){ //d = longitud
        for (int j, i=0; (j = i + d) < m; ++i){
            dp[i][j] = p[j] - p[i];
            int t = INT_MAX, s;
            for (int k=pivot[i][j-1]; k<=pivot[i+1][j]; ++k){
                int x = dp[i][k] + dp[k][j];
                if (x < t) t = x, s = k;
            }
            dp[i][j] += t, pivot[i][j] = s;
        }
    }

    printf("%d\n", dp[0][m-1]);
}
return 0;
}

```

6. Strings

6.1. Algoritmo de Knuth-Morris-Pratt (KMP)

Computa el arreglo *border* que contiene la longitud del borde más largo de todos los prefijos de una cadena.

Un borde de una cadena es otra cadena más corta que es a la vez prefijo y sufijo de la original (por ejemplo, *aba* es un borde de *abacaba* porque es más corta que *abacaba* y es al mismo tiempo prefijo y sufijo de *abacaba*. *a* también es un borde de *abacaba*. *abac* no es un borde de *abacaba* porque no es un sufijo).

En el código, `border[i]` contiene el borde más grande del prefijo de *needle* que termina en la posición *i* (*needle* es el patrón que se quiere buscar en la otra cadena). Una ejemplo del arreglo `border` es:

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|
| <i>needle</i> | a | b | a | c | a | b | a | c | a | b | a | d | a | b |
| <i>border</i> | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 0 | 1 | 2 |

```
// Knuth-Morris-Pratt algorithm for string matching
// Complexity: O(n + m)

// Reports all occurrences of 'needle' in 'haystack'.
void kmp(const string &needle, const string &haystack) {
    // Precompute border function
    int m = needle.size();
    vector<int> border(m);
    border[0] = 0;
    for (int i = 1; i < m; ++i) {
        border[i] = border[i - 1];
        while (border[i] > 0 and needle[i] != needle[border[i]]) {
            border[i] = border[border[i] - 1];
        }
        if (needle[i] == needle[border[i]]) border[i]++;
    }

    // Now the actual matching
    int n = haystack.size();
    int seen = 0;
    for (int i = 0; i < n; ++i){
        while (seen > 0 and haystack[i] != needle[seen]) {
            seen = border[seen - 1];
        }
        if (haystack[i] == needle[seen]) seen++;

        if (seen == m) {
            printf("Needle occurs from %d to %d\n", i - m + 1, i);
            seen = border[m - 1];
        }
    }
}
```

}

6.2. Algoritmo Z

$Z[i]$ = longitud del prefijo propio más largo de $S[i, n)$ que también es prefijo de $S[0, n)$.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---------------|---|---|---|---|---|---|---|---|---|---|----|----|----|
| <i>string</i> | a | b | a | c | a | b | a | c | a | d | a | b | a |
| Z_i | 0 | 0 | 1 | 0 | 5 | 0 | 1 | 0 | 1 | 0 | 3 | 0 | 1 |

```
// Find z function
int n = s.size();
vector<int> z(n);
z[0] = 0;
for (int i = 1, l = 0, r = 0; i < n; ++i) {
    z[i] = 0;
    if (i <= r) z[i] = min(z[i - l], r - i + 1);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
    if (i + z[i] - 1 > r) {
        l = i;
        r = i + z[i] - 1;
    }
}
}
```

6.3. Algoritmo de Aho-Corasick

Sirve para buscar muchos patrones en una cadena. Por ejemplo, dada la cadena *ahishers* y los patrones *{he, she, hers, his}*, encuentra que *his* aparece en la posición 1, *he* aparece en la posición 4, *she* aparece en la posición 3 y *hers* aparece en la posición 4.

Complejidad: $O(n + m)$ donde n es la longitud de la cadena en la que hay que buscar y m es la suma de las longitudes de todos los patrones.

```
////////////////////////////////////
//      Aho-Corasick's algorithm, as explained in      //
//      http://dx.doi.org/10.1145/360825.360855          //
////////////////////////////////////
```

```

// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 6 * 50 + 10;

// Number of characters in the alphabet.
const int MAXC = 26;

// Output for each state, as a bitwise mask.
// Bit i in this mask is on if the keyword with index i
// appears when the machine enters this state.
int out[MAXS];

// Used internally in the algorithm.
int f[MAXS]; // Failure function
int g[MAXS][MAXC]; // Goto function, or -1 if fail.

// Builds the string matching machine.
//
// words - Vector of keywords. The index of each keyword is
//         important:
//         "out[state] & (1 << i)" is > 0 if we just found
//         word[i] in the text.
// lowestChar - The lowest char in the alphabet.
//              Defaults to 'a'.
// highestChar - The highest char in the alphabet.
//              Defaults to 'z'.
//              "highestChar - lowestChar" must be <= MAXC,
//              otherwise we will access the g matrix outside
//              its bounds and things will go wrong.
//
// Returns the number of states that the new machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(const vector<string> &words,
                        char lowestChar = 'a',
                        char highestChar = 'z') {
    memset(out, 0, sizeof out);
    memset(f, -1, sizeof f);
    memset(g, -1, sizeof g);

    int states = 1; // Initially, we just have the 0 state

    for (int i = 0; i < words.size(); ++i) {
        const string &keyword = words[i];

```

```

        int currentState = 0;
        for (int j = 0; j < keyword.size(); ++j) {
            int c = keyword[j] - lowestChar;
            if (g[currentState][c] == -1) {
                // Allocate a new node
                g[currentState][c] = states++;
            }
            currentState = g[currentState][c];
        }
        // There's a match of keywords[i] at node currentState.
        out[currentState] |= (1 << i);
    }

    // State 0 should have an outgoing edge for all characters.
    for (int c = 0; c < MAXC; ++c) {
        if (g[0][c] == -1) {
            g[0][c] = 0;
        }
    }

    // Now, let's build the failure function
    queue<int> q;
    // Iterate over every possible input
    for (int c = 0; c <= highestChar - lowestChar; ++c) {
        // All nodes s of depth 1 have f[s] = 0
        if (g[0][c] != -1 and g[0][c] != 0) {
            f[g[0][c]] = 0;
            q.push(g[0][c]);
        }
    }
    while (q.size()) {
        int state = q.front();
        q.pop();
        for (int c = 0; c <= highestChar - lowestChar; ++c) {
            if (g[state][c] != -1) {
                int failure = f[state];
                while (g[failure][c] == -1) {
                    failure = f[failure];
                }
                failure = g[failure][c];
                f[g[state][c]] = failure;
            }
        }
        // Merge out values

```



```

//Usage: Call SuffixArray::compute(s), where s is the
//      string you want the Suffix Array for.
//
// * * * IMPORTANT: The last character of s must compare less
//      than any other character (for example, do s = s + '\1';
//      before calling this function).

//Output:
// sa   = The suffix array. Contains the n suffixes of s sorted
//      in lexicographical order. Each suffix is represented
//      as a single integer (the position in the string
//      where it starts).
// rank = The inverse of the suffix array. rank[i] = the index
//      of the suffix s[i..n) in the pos array. (In other
//      words, sa[i] = k <=> rank[k] = i).
//      With this array, you can compare two suffixes in O(1):
//      Suffix s[i..n) is smaller than s[j..n) if and
//      only if rank[i] < rank[j].
// lcp  = The length of the longest common prefix between two
//      consecutive suffixes:
//      lcp[i] = lcp(s + sa[i], s + sa[i-1]). lcp[0] = 0.

namespace SuffixArray {
    int t, rank[MAXN], sa[MAXN], lcp[MAXN];

    bool compare(int i, int j){
        return rank[i + t] < rank[j + t];
    }

    void build(const string &s){
        int n = s.size();
        int bc[256];
        for (int i = 0; i < 256; ++i) bc[i] = 0;
        for (int i = 0; i < n; ++i) ++bc[s[i]];
        for (int i = 1; i < 256; ++i) bc[i] += bc[i-1];
        for (int i = 0; i < n; ++i) sa[--bc[s[i]]] = i;
        for (int i = 0; i < n; ++i) rank[i] = bc[s[i]];
        for (t = 1; t < n; t <= 1){
            for (int i = 0, j = 1; j < n; i = j++){
                while (j < n && rank[sa[j]] == rank[sa[i]]) j++;
                if (j - i == 1) continue;
                int *start = sa + i, *end = sa + j;
                sort(start, end, compare);
            }
        }
    }
}

```

```

        int first = rank[*start + t], num = i, k;
        for(; start < end; rank[*start++] = num){
            k = rank[*start + t];
            if (k != first and (i > first or k >= j))
                first = k, num = start - sa;
        }
    }
}

// Remove this part if you don't need the LCP
int size = 0, i, j;
for(i = 0; i < n; i++) if (rank[i] > 0) {
    j = sa[rank[i] - 1];
    while(s[i + size] == s[j + size]) ++size;
    lcp[rank[i]] = size;
    if (size > 0) --size;
}
lcp[0] = 0;
}
};

////////////////////////////////////

// Applications:

// lcp(x,y) = min(lcp(x,x+1), lcp(x+1, x+2), ... , lcp(y-1, y))

void number_of_different_substrings(){
    // If you have the i-th smaller suffix, Si,
    // it's length will be |Si| = n - sa[i]
    // Now, lcp[i] stores the number of
    // common letters between Si and Si-1
    // (s.substr(sa[i]) and s.substr(sa[i-1]))
    // so, you have |Si| - lcp[i] different strings
    // from these two suffixes => n - lcp[i] - sa[i]
    for(int i = 0; i < n; ++i) ans += n - sa[i] - lcp[i];
}

void number_of_repeated_substrings(){
    // Number of substrings that appear at least twice in the text.
    // The trick is that all 'spare' substrings that can give us
    // Lcp(i - 1, i) can be obtained by Lcp(i - 2, i - 1)
    // due to the ordered nature of our array.
    // And the overall answer is

```

```

// Lcp(0, 1) +
// Sum(max[0, Lcp(i, i - 1) - Lcp(i - 2, i - 1)])
// for 2 <= i < n
// File Recover
int cnt = lcp[1];
for(int i=2; i < n; ++i){
    cnt += max(0, lcp[i] - lcp[i-1]);
}
}

void repeated_m_times(int m){
    // Given a string s and an int m, find the size
    // of the biggest substring repeated m times (find the rightmost pos)
    // if a string is repeated m+1 times, then it's repeated m times too
    // The answer is the maximum, over i, of the longest common prefix
    // between suffix i+m-1 in the sorted array.
    int length = 0, position = -1, t;
    for (int i = 0; i <= n-m; ++i){
        if ((t = getLcp(i, i+m-1, n)) > length){
            length = t;
            position = sa[i];
        } else if (t == length) { position = max(position, sa[i]); }
    }
    // Here you'll get the rightmost position
    // (that means, the last time the substring appears)
    for (int i = 0; i < n; ){
        if (sa[i] + length > n) { ++i; continue; }
        int ps = 0, j = i+1;
        while (j < n && lcp[j] >= length){
            ps = max(ps, sa[j]);
            j++;
        }
        if(j - i >= m) position = max(position, ps);
        i = j;
    }
    if(length != 0)
        printf("%d %d\n", length, position);
    else
        puts("none");
}

```

```

void smallest_rotation(){
    // Reads a string of lenght k. Then just double it (s = s+s)
    // and find the suffix array.
    // The answer is the smallest i for which s.size() - sa[i] >= k
    // If you want the first appearence (and not the string)
    // you'll need the second cycle
    int best = 0;
    for (int i=0; i < n; ++i){
        if (n - sa[i] >= k){
            //Find the first appearence of the string
            while (n - sa[i] >= k){
                if(sa[i] < sa[best] && sa[i] != 0) best = i;
                i++;
            }
            break;
        }
    }
    if (sa[best] == k) puts("0");
    else printf("%d\n", sa[best]);
}

```

6.5. Hashing dinámico

Usando una modificación de un Fenwick Tree, se puede hacer una estructura de datos que soporta las siguientes operaciones en $O(\lg n)$:

- Encontrar el hash de la subcadena $[i..j]$.
- Modificar el caracter en la posición i .

```

// N = size of the array. It is assumed that elements are
// indexed from 1 to N, inclusive.
// B = the base for the hash. Must be > 0.
// P = The modulo for the hash. Must be > 0. Doesn't need
// to be prime.

```

```

int N, B, P;

int tree[MAXN], base[MAXN];

```

```

void precomputeBases() {
    base[0] = 1;
    for (int i = 1; i <= N + 1; ++i) {

```

```

        base[i] = (1LL * base[i - 1] * B) % P;
    }
}

inline int mod(long long a) {
    int ans = a % P;
    if (ans < 0) ans += P;
    return ans;
}

// Usually you don't want to use this function directly,
// use 'put' below instead.
void add(int at, int what) {
    what = mod(what);
    int seen = 0;
    for (at++; at <= N + 1; at += at & -at) {
        tree[at] += (1LL * what * base[seen]) % P;
        tree[at] = mod(tree[at]);
        seen += at & -at;
    }
}

// Returns the hash for subarray [1..at].
int query(int at) {
    int ans = 0, seen = 0;
    for (at++; at > 0; at -= at & -at) {
        ans += (1LL * tree[at] * base[seen]) % P;
        ans = mod(ans);
        seen += at & -at;
    }
    return ans;
}

// Returns the hash for subarray [i..j]. That hash is:
// a[i]*B^(j-i+1) + a[i+1]*B^(j-i) + a[i+2]*B^(j-i-1) + ...
// + a[j-2]*B^2 + a[j-1]*B^1 + a[j]*B^0 (mod P)
int hash(int i, int j) {
    assert(i <= j);
    int ans = query(j) - (1LL * query(i-1) * base[j-i+1]) % P;
    return mod(ans);
}

// Changes the number or char at position 'at' for 'what'.

```

```

void put(int at, int what) {
    add(at, -hash(at, at) + what);
}

```

6.6. Mínima rotación lexicográfica de una string

Field-testing:

- *SPOJ* - MINMOVE - Minimum Rotations

```

// Finds the lexicographically smallest rotation of string s.
// Returns the index that should be moved to the first position
// to achieve the smallest rotation.
// If there are two or more smallest rotations, returns the
// smallest index.
int minimum_rotation(string s) {
    int n = s.size();
    s = s + s;
    int mini = 0, p = 1, k = 0;
    while (p < n && mini + k + 1 < n) {
        if (s[mini + k] == s[p + k]) {
            k++;
        } else if (s[mini + k] < s[p + k]) {
            p = p + k + 1;
            k = 0;
        } else if (s[mini + k] > s[p + k]) {
            mini = max(mini + k + 1, p);
            p = mini + 1;
            k = 0;
        }
    }
    // the actual minimum rotated string is s.substr(mini, n)
    return mini;
}

```

6.7. Algoritmo de Manacher: Hallar las subcadenas palindrómicas más largas

```

// Complejidad: O(n)
void manacher(const string &s) {

```

```

int n = s.size();

vector<int> d1(n);
int l=0, r=-1;
for (int i=0; i<n; ++i) {
    int k = (i>r ? 0 : min (d1[l+r-i], r-i)) + 1;
    while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) ++k;
    d1[i] = --k;
    if (i+k > r) l = i-k, r = i+k;
}
vector<int> d2(n);
l=0, r=-1;
for (int i=0; i<n; ++i) {
    int k = (i>r ? 0 : min (d2[l+r-i+1], r-i+1)) + 1;
    while (i+k-1 < n && i-k >= 0 && s[i+k-1] == s[i-k]) ++k;
    d2[i] = --k;
    if (i+k-1 > r) l = i-k, r = i+k-1;
}

// d1[i] = piso de la mitad de la longitud del palíndromo
//         impar más largo cuyo centro es i.
// d2[i] = mitad de la longitud del palíndromo par más
//         largo cuyo centro de la derecha es i.

for (int i = 0; i < n; ++i) {
    assert(is_palindrome( s.substr(i - d1[i], 2*d1[i] + 1) ));
    assert(is_palindrome( s.substr(i - d2[i], 2*d2[i])      ));
}
}

```

7. Geometría

7.1. Identidades trigonométricas

$$\sin(\alpha \pm \beta) = \sin \alpha \cos \beta \pm \cos \alpha \sin \beta$$

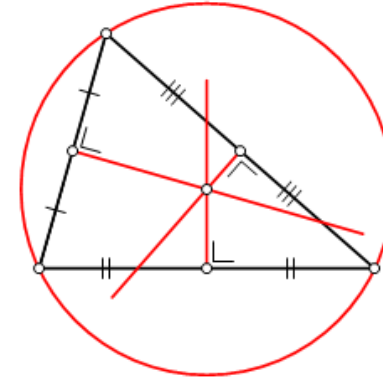
$$\cos(\alpha \pm \beta) = \cos \alpha \cos \beta \mp \sin \alpha \sin \beta$$

$$\tan(\alpha \pm \beta) = \frac{\tan \alpha \pm \tan \beta}{1 \mp \tan \alpha \tan \beta}$$

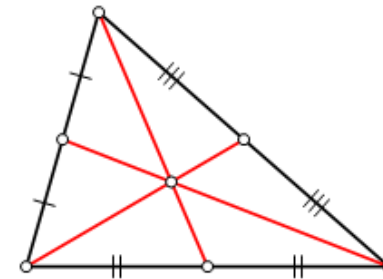
7.2. Triángulos

7.2.1. Circuncentro, incentro, baricentro y ortocentro

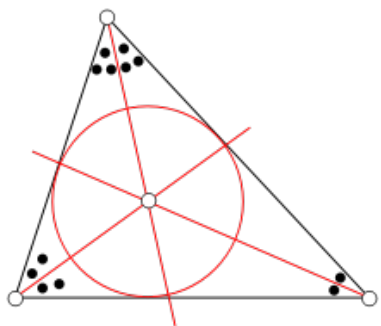
- El circuncentro (círculo que contiene el triángulo) es el punto de intersección de las mediatrices (recta perpendicular al lado en su punto medio).



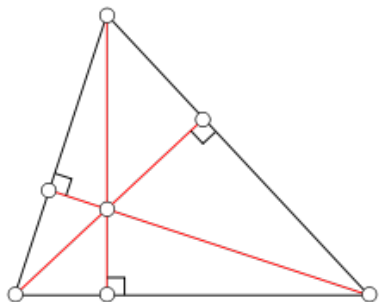
- El baricentro (centro de gravedad) es el punto de intersección de las medianas (recta de un vértice al punto medio del lado opuesto).



- El incentro (centro del círculo inscrito) es el punto de intersección de las bisectrices (recta que divide el ángulo en dos ángulos iguales).



- El ortocentro es el punto donde se cortan las tres alturas (recta desde un vértice y perpendicular al lado opuesto). El ortocentro se encuentra dentro del triángulo si éste es acutángulo, coincide con el vértice del ángulo recto si es rectángulo, y se halla fuera del triángulo si es obtusángulo.



7.2.2. Centro del círculo que pasa por 3 puntos (circuncentro)

Field-testing:

- *Live Archive* - 4807 - Cocircular Points

```
point center(const point &p, const point &q, const point &r) {
    double ax = q.x - p.x;
    double ay = q.y - p.y;
    double bx = r.x - p.x;
    double by = r.y - p.y;
    double d = ax*by - bx*ay;

    if (cmp(d, 0) == 0) {
        printf("Points are collinear!\n");
        assert(false);
    }
}
```

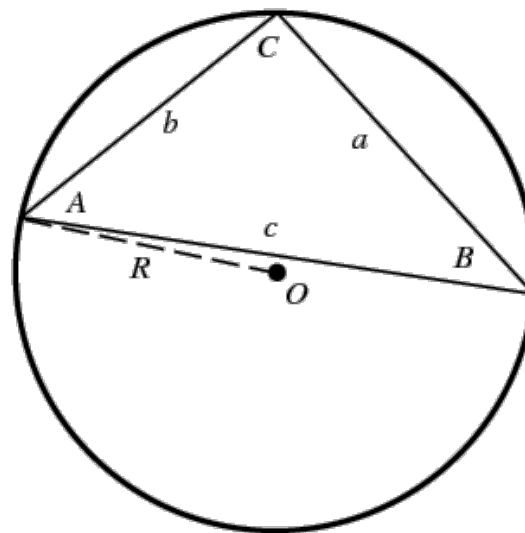
```
double cx = (q.x + p.x) / 2;
double cy = (q.y + p.y) / 2;
double dx = (r.x + p.x) / 2;
double dy = (r.y + p.y) / 2;

double t1 = bx*dx + by*dy;
double t2 = ax*cx + ay*cy;

double x = (by*t2 - ay*t1) / d;
double y = (ax*t1 - bx*t2) / d;

return point(x, y);
}
```

7.2.3. Ley del seno y ley del coseno



Sean a , b y c las longitudes de los lados de un triángulo con ángulos opuestos A , B y C . La ley del seno dice:

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C} = 2R$$

donde R es el radio del circuncírculo.

Otros resultados son:

$$a(\sin B - \sin C) + b(\sin C - \sin A) + c(\sin A - \sin B) = 0$$

$$a = b \cos C + c \cos B$$

La ley del coseno:

$$\cos A = \frac{c^2 + b^2 - a^2}{2bc}$$

y la ley de tangentes:

$$\frac{a+b}{a-b} = \frac{\tan[\frac{1}{2}(A+B)]}{\tan[\frac{1}{2}(A-B)]}.$$

7.3. Área de un polígono

Si P es un polígono simple (no se intersecta a sí mismo) su área está dada por:

$$A(P) = \frac{1}{2} \sum_{i=0}^{n-1} (x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

```
//P es un polígono ordenado anticlockwise.
//Si es clockwise, retorna el area negativa.
//Si no esta ordenado retorna pura mierda.
//P[0] != P[n-1]
double PolygonArea(const vector<point> &p){
    double r = 0.0;
    for (int i=0; i<p.size(); ++i){
        int j = (i+1) % p.size();
        r += p[i].x*p[j].y - p[j].x*p[i].y;
    }
    return r/2.0;
}
```

7.4. Centro de masa de un polígono

Si P es un polígono simple (no se intersecta a sí mismo) su centro de masa está dado por:

$$\bar{C}_x = \frac{\iint_R x dA}{M} = \frac{1}{6M} \sum_{i=1}^n (y_{i+1} - y_i)(x_{i+1}^2 + x_{i+1} \cdot x_i + x_i^2)$$

$$\bar{C}_y = \frac{\iint_R y dA}{M} = \frac{1}{6M} \sum_{i=1}^n (x_i - x_{i+1})(y_{i+1}^2 + y_{i+1} \cdot y_i + y_i^2)$$

Donde M es el área del polígono.

Otra posible fórmula equivalente:

$$\bar{C}_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

$$\bar{C}_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

7.5. Convex hull: Graham Scan

Complejidad: $O(n \log_2 n)$

```
//Graham scan: Complexity: O(n log n)
struct point{
    int x,y;
    point() {}
    point(int X, int Y) : x(X), y(Y) {}
};

point pivot;

inline int distsq(const point &a, const point &b){
    return (a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y);
}

inline double dist(const point &a, const point &b){
    return sqrt(distsq(a, b));
}
```

```

//retorna > 0 si c esta a la izquierda del segmento AB
//retorna < 0 si c esta a la derecha del segmento AB
//retorna == 0 si c es colineal con el segmento AB
inline
int cross(const point &a, const point &b, const point &c){
    return (b.x-a.x)*(c.y-a.y) - (c.x-a.x)*(b.y-a.y);
}

//Self < that si esta a la derecha del segmento Pivot-That
bool angleCmp(const point &self, const point &that){
    int t = cross(pivot, that, self);
    if (t < 0) return true;
    if (t == 0){
        //Self < that si está más cerquita
        return (distsqr(pivot, self) < distsqr(pivot, that));
    }
    return false;
}

vector<point> graham(vector<point> p){
    //Metemos el más abajo más a la izquierda en la posición 0
    for (int i=1; i<p.size(); ++i){
        if (p[i].y < p[0].y ||
            (p[i].y == p[0].y && p[i].x < p[0].x))
            swap(p[0], p[i]);
    }

    pivot = p[0];
    sort(p.begin(), p.end(), angleCmp);

    //Ordenar por ángulo y eliminar repetidos.
    //Si varios puntos tienen el mismo angulo el más lejano
    //queda después en la lista
    vector<point> hull(p.begin(), p.begin()+3);

    //Ahora sí!!!
    for (int i=3; i<p.size(); ++i){
        while (hull.size() >= 2 &&
            cross(hull[hull.size()-2],
                hull[hull.size()-1],
                p[i]) <= 0){
            hull.erase(hull.end() - 1);
        }
    }
}

```

```

    }
    hull.push_back(p[i]);
}
//hull contiene los puntos del convex hull ordenados
//anti-clockwise. No contiene ningún punto repetido. El
//primer punto no es el mismo que el último, i.e, la última
//arista va de hull[hull.size()-1] a hull[0]
return hull;
}
.....

```

7.6. Convex hull: Andrew's monotone chain

Complejidad: $O(n \log_2 n)$

// Convex Hull: Andrew's Monotone Chain Convex Hull
// Complexity: $O(n \log n)$ (But lower constant than Graham Scan)

```
typedef long long CoordType;
```

```

struct Point {
    CoordType x, y;
    bool operator <(const Point &p) const {
        return x < p.x || (x == p.x && y < p.y);
    }
};

```

// 2D cross product. Returns a positive value, if OAB makes a
// counter-clockwise turn, negative for clockwise turn, and zero
// if the points are collinear.

```

CoordType cross(const Point &O, const Point &A, const Point &B){
    return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

```

// Returns a list of points on the convex hull in
// counter-clockwise order. Note: the last point in the returned
// list is the same as the first one.

```

vector<Point> convexHull(vector<Point> P){
    int n = P.size(), k = 0;
    vector<Point> H(2*n);
    // Sort points lexicographically
    sort(P.begin(), P.end());
    // Build lower hull

```

```

for (int i = 0; i < n; i++) {
    while (k >= 2 && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
}
// Build upper hull
for (int i = n-2, t = k+1; i >= 0; i--) {
    while (k >= t && cross(H[k-2], H[k-1], P[i]) <= 0) k--;
    H[k++] = P[i];
}
H.resize(k);
return H;
}

```

7.7. Mínima distancia entre un punto y un segmento

```

/*
Returns the closest distance between point pnt and the segment
that goes from point a to b
Idea by:
http://local.wasp.uwa.edu.au/~pbourke/geometry/pointline/
*/
double distance_point_to_segment(const point &a, const point &b,
                                const point &pnt){
    double u =
        ((pnt.x - a.x)*(b.x - a.x) + (pnt.y - a.y)*(b.y - a.y))
        /distsqr(a, b);
    point intersection;
    intersection.x = a.x + u*(b.x - a.x);
    intersection.y = a.y + u*(b.y - a.y);
    if (u < 0.0 || u > 1.0){
        return min(dist(a, pnt), dist(b, pnt));
    }
    return dist(pnt, intersection);
}

```

7.8. Mínima distancia entre un punto y una recta

```

/*
Returns the closest distance between point pnt and the line
that passes through points a and b

```

```

Idea by:
http://local.wasp.uwa.edu.au/~pbourke/geometry/pointline/
*/
double distance_point_to_line(const point &a, const point &b,
                              const point &pnt){
    double u =
        ((pnt.x - a.x)*(b.x - a.x) + (pnt.y - a.y)*(b.y - a.y))
        /distsqr(a, b);
    point intersection;
    intersection.x = a.x + u*(b.x - a.x);
    intersection.y = a.y + u*(b.y - a.y);
    return dist(pnt, intersection);
}

```

7.9. Determinar si un polígono es convexo

```

/*
Returns positive if a-b-c makes a left turn.
Returns negative if a-b-c makes a right turn.
Returns 0.0 if a-b-c are colinear.
*/
double turn(const point &a, const point &b, const point &c){
    double z = (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
    if (fabs(z) < 1e-9) return 0.0;
    return z;
}

/*
Returns true if polygon p is convex.
False if it's concave or it can't be determined
(For example, if all points are colinear we can't
make a choice).
*/
bool isConvexPolygon(const vector<point> &p){
    int mask = 0;
    int n = p.size();
    for (int i=0; i<n; ++i){
        int j=(i+1)%n;
        int k=(i+2)%n;
        double z = turn(p[i], p[j], p[k]);
        if (z < 0.0){

```

```

    mask |= 1;
} else if (z > 0.0){
    mask |= 2;
}
if (mask == 3) return false;
}
return mask != 0;
}

```

.....

7.10. Determinar si un punto está dentro de un polígono convexo

```

/*
Returns true if point a is inside convex polygon p. Note
that if point a lies on the border of p it is considered
outside.

We assume p is convex! The result is useless if p is
concave.
*/
bool insideConvexPolygon(const vector<point> &p,
                        const point &a){

    int mask = 0;
    int n = p.size();
    for (int i=0; i<n; ++i){
        int j = (i+1)%n;
        double z = turn(p[i], p[j], a);
        if (z < 0.0){
            mask |= 1;
        } else if (z > 0.0){
            mask |= 2;
        } else if (z == 0.0) return false;
        if (mask == 3) return false;
    }
    return mask != 0;
}

```

.....

7.11. Determinar si un punto está dentro de un polígono cualquiera

Field-testing:

- *TopCoder* - SRM 187 - Division 2 Hard - PointInPolygon
- *UVa* - 11665 - Chinese Ink

```

//Point
//Choose one of these two:
struct P {
    double x, y; P(){}; P(double q, double w) : x(q), y(w){}
};
struct P {
    int x, y; P(){}; P(int q, int w) : x(q), y(w){}
};

// Polar angle
// Returns an angle in the range [0, 2*Pi) of a given
// Cartesian point. If the point is (0,0), -1.0 is returned.

// REQUIRES:
// include math.h
// define EPS 0.000000001, or your choice
// P has members x and y.
double polarAngle( P p )
{
    if(fabs(p.x) <= EPS && fabs(p.y) <= EPS) return -1.0;
    if(fabs(p.x) <= EPS) return (p.y > EPS ? 1.0 : 3.0) * acos(0);
    double theta = atan(1.0 * p.y / p.x);
    if(p.x > EPS)
        return (p.y >= -EPS ? theta : (4*acos(0) + theta));
    else
        return(2 * acos( 0 ) + theta);
}

//Point inside polygon
// Returns true iff p is inside poly.
// PRE: The vertices of poly are ordered (either clockwise or
//       counter-clockwise.
// POST: Modify code inside to handle the special case of "on
//       an edge".
// REQUIRES:
// polarAngle()

```

```

// include math.h
// include vector
// define EPS 0.000000001, or your choice
bool pointInPoly( P p, vector< P > &poly )
{
    int n = poly.size();
    double ang = 0.0;
    for(int i = n - 1, j = 0; j < n; i = j++){
        P v( poly[i].x - p.x, poly[i].y - p.y );
        P w( poly[j].x - p.x, poly[j].y - p.y );
        double va = polarAngle(v);
        double wa = polarAngle(w);
        double xx = wa - va;
        if(va < -0.5 || wa < -0.5 || fabs(fabs(xx)-2*acos(0)) < EPS){
            // POINT IS ON THE EDGE
            assert( false );
            ang += 2 * acos( 0 );
            continue;
        }
        if( xx < -2 * acos( 0 ) ) ang += xx + 4 * acos( 0 );
        else if( xx > 2 * acos( 0 ) ) ang += xx - 4 * acos( 0 );
        else ang += xx;
    }
    return( ang * ang > 1.0 );
}

```

7.12. Hallar la intersección de dos rectas

```

/*
    Finds the intersection between two lines (Not segments!
    Infinite lines)
    Line 1 passes through points (x0, y0) and (x1, y1).
    Line 2 passes through points (x2, y2) and (x3, y3).

    Handles the case when the 2 lines are the same (infinite
    intersections),
    parallel (no intersection) or only one intersection.
*/
void line_line_intersection(double x0, double y0,
                           double x1, double y1,

```

```

                           double x2, double y2,
                           double x3, double y3){
#ifdef EPS
#define EPS 1e-9
#endif

    double t0 = (y3-y2)*(x0-x2)-(x3-x2)*(y0-y2);
    double t1 = (x1-x0)*(y2-y0)-(y1-y0)*(x2-x0);
    double det = (y1-y0)*(x3-x2)-(y3-y2)*(x1-x0);
    if (fabs(det) < EPS){
        //parallel
        if (fabs(t0) < EPS || fabs(t1) < EPS){
            //same line
            printf("LINE\n");
        }else{
            //just parallel
            printf("NONE\n");
        }
    }else{
        t0 /= det;
        t1 /= det;
        double x = x0 + t0*(x1-x0);
        double y = y0 + t0*(y1-y0);
        //intersection is point (x, y)
        printf("POINT %.21f %.21f\n", x, y);
    }
}

```

7.13. Hallar la intersección de dos segmentos de recta

Field-testing:

- UVa - 11665 - Chinese Ink

```

/*
    Returns true if point (x, y) lies inside (or in the border)
    of box defined by points (x0, y0) and (x1, y1).
*/
bool point_in_box(double x, double y,
                  double x0, double y0,
                  double x1, double y1){
    return

```


(Can be modified to find the intersection between a segment and a line)

Handles the case when the 2 segments are:

- *Parallel but don't lie on the same line (No intersection)
- *Parallel and both lie on the same line (Infinite intersections or no intersections)
- *Not parallel (One intersection or no intersections)

Returns true if the segments do intersect in any case.

```

*/
bool segment_segment_intersection(int x1, int y1,
                                int x2, int y2,
                                int x3, int y3,
                                int x4, int y4){

    int d1 = direction(x3, y3, x4, y4, x1, y1);
    int d2 = direction(x3, y3, x4, y4, x2, y2);
    int d3 = direction(x1, y1, x2, y2, x3, y3);
    int d4 = direction(x1, y1, x2, y2, x4, y4);
    bool b1 = d1 > 0 and d2 < 0 or d1 < 0 and d2 > 0;
    bool b2 = d3 > 0 and d4 < 0 or d3 < 0 and d4 > 0;
    if (b1 and b2) return true;
    /* point_in_box is on segment_segmet_intersection */
    if (d1 == 0 and point_in_box(x1, y1, x3, y3, x4, y4))
        return true;

    if (d2 == 0 and point_in_box(x2, y2, x3, y3, x4, y4))
        return true;

    if (d3 == 0 and point_in_box(x3, y3, x1, y1, x2, y2))
        return true;

    if (d4 == 0 and point_in_box(x4, y4, x1, y1, x2, y2))
        return true;

    return false;
}

```

7.15. Cortar un polígono convexo por una recta infinita

Field-testing:

- UVa - 12514 - Cookie

```

typedef long double Double;
typedef vector<Point> Polygon;

// This is not standard intersection because it returns false
// when the intersection point is exactly the t=1 endpoint of
// the segment. This is OK for this algorithm but not for general
// use.
bool segment_line_intersection(Double x0, Double y0,
                               Double x1, Double y1, Double x2, Double y2,
                               Double x3, Double y3, Double &x, Double &y){

    Double t0 = (y3-y2)*(x0-x2) - (x3-x2)*(y0-y2);
    Double t1 = (x1-x0)*(y2-y0) - (y1-y0)*(x2-x0);
    Double det = (y1-y0)*(x3-x2) - (y3-y2)*(x1-x0);

    if (fabs(det) < EPS){
        //Paralelas
        return false;
    }else{
        t0 /= det;
        t1 /= det;
        if (cmp(0, t0) <= 0 and cmp(t0, 1) < 0){
            x = x0 + t0 * (x1-x0);
            y = y0 + t0 * (y1-y0);
            return true;
        }
        return false;
    }
}

// Returns the polygons that result of cutting the CONVEX
// polygon p by the infinite line that passes through (x0, y0)
// and (x1, y1).
// The returned value has either 1 element if this line
// doesn't cut the polygon at all (or barely touches it)
// or 2 elements if the line does split the polygon.
vector<Polygon> split(const Polygon &p, Double x0, Double y0,
                    Double x1, Double y1) {

```

```

int hits = 0, side = 0;
Double x, y;
vector<Polygon> ans(2);

for (int i = 0; i < p.size(); ++i) {
    int j = (i + 1) % p.size();
    if (segment_line_intersection(p[i].x, p[i].y,
        p[j].x, p[j].y, x0, y0, x1, y1, x, y)) {
        hits++;
        ans[side].push_back(p[i]);
        if (cmp(p[i].x, x) != 0 or cmp(p[i].y, y) != 0) {
            ans[side].push_back(Point(x, y));
        }
        side ^= 1;
        ans[side].push_back(Point(x, y));
    } else {
        ans[side].push_back(p[i]);
    }
}
return hits < 2 ? vector<Polygon>(1, p) : ans;
}

```

7.16. Proyección de un vector en otro

$$\vec{a} \text{ proyectado en } \vec{b} = \vec{c} = \frac{\vec{a} \cdot \vec{b}}{|\vec{b}|^2} \vec{b} = \frac{\vec{a} \cdot \vec{b}}{\vec{b} \cdot \vec{b}} \vec{b}.$$

\vec{c} es un vector nulo ó paralelo a \vec{b} cuya magnitud es la “sombra” (la proyección ortogonal) proyectada por el vector \vec{a} sobre una recta infinita paralela a \vec{b} . Si θ es el ángulo entre \vec{a} y \vec{b} entonces:

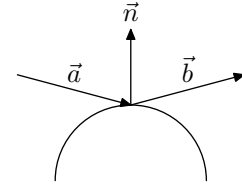
- $\vec{c} = \vec{0}$ si $\theta = 90^\circ$,
- \vec{c} tiene la misma dirección que \vec{b} si $0 \leq \theta < 90^\circ$,
- \vec{c} tiene dirección contraria a \vec{b} si $90 < \theta \leq 180^\circ$.

7.17. Reflejar un rayo de luz en un espejo

Field-testing:

- *Live Archive* - 5929 - Laser Tag

Dado un rayo \vec{a} incidente en una superficie con vector normal unitario \vec{n} se quiere encontrar el rayo \vec{b} reflejado en ese punto:



$$\vec{b} = \vec{a} - 2(\vec{a} \cdot \vec{n})\vec{n}$$

donde \cdot es producto punto. ¡Para que la fórmula funcione \vec{n} tiene que ser unitario! Notar que no importa cuál de las dos direcciones tiene \vec{n} porque si lo reemplazamos por $-\vec{n}$ da el mismo resultado.

```

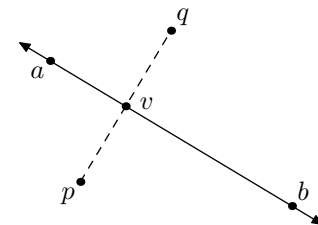
// Reflects the ray that goes in direction dir on the infinite
// line that passes through a and b.
Point reflect_ray(const Point &dir, const Point &a, const Point &b){
    Point n(b.y - a.y, a.x - b.x); // normal
    long double length = sqrt(n.x * n.x + n.y * n.y);
    n.x /= length, n.y /= length;
    long double dot = 2 * (dir.x * n.x + dir.y * n.y);
    Point new_dir = dir;
    new_dir.x -= dot * n.x, new_dir.y -= dot * n.y;
    return new_dir;
}

```

7.18. Reflejar un punto al otro lado de una recta

Field-testing:

- *Live Archive* - 5929 - Laser Tag



```

// Reflects point p to the other side of the infinite line that
// passes through a and b. It's assumed that a != b.
Point reflect(Point p, const Point &a, const Point &b) {

```



```

double dx = b.x - a.x;
double dy = b.y - a.y;
double u = (dx * (p.x - a.x) + (p.y - a.y) * dy) /
            (dx * dx + dy * dy);

Point v; // Intersection
v.x = a.x + u * dx;
v.y = a.y + u * dy;

p.x += 2*(v.x - p.x);
p.y += 2*(v.y - p.y);
return p;
}

```

7.19. Hallar la intersección de un segmento y una superficie cuádrica (esfera, cono, elipsoide, hiperboloide o paraboloides)

Field-testing:

- *UVa - 11334 - Antenna in the Cinoc Mountains*

La ecuación genérica para describir una superficie cuádrica es

$$ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j = 0$$

Se tiene un segmento de P a Q y se quiere hallar la intersección (si existe):

```

// Find the intersection of the segment that goes from P to Q with
// the quadric defined by
// ax^2 + by^2 + cz^2 + dxy + exz + fyz + gx + hy + iz + j = 0

```

```

Point P, Q; // Initialize to the two endpoints of the segment

```

```

// Initialize to the coefficients of the quadric
long double a, b, c, d, e, f, g, h, i, j;

```

```

long double xd = Q.x - P.x, yd = Q.y - P.y, zd = Q.z - P.z,
            xo = P.x, yo = P.y, zo = P.z;

```

```

double A = a*xd*xd + b*yd*yd + c*zd*zd + d*xd*yd + e*xd*zd +
            f*yd*zd;

```

```

double B = 2*(a*xo*xd + b*yo*yd + c*zo*zd) + d*(xo*yd + yo*xd) +
            e*(xo*zd + xd*zo) + f*(yo*zd + yd*zo) + g*xd +
            h*yd + i*zd;

```

```

double C = a*xo*xo + b*yo*yo + c*zo*zo + d*xo*yo + e*xo*zo +
            f*yo*zo + g*xo + h*yo + i*zo + j;

```

```

// Now solve the quadratic equation defined by A, B and C. This
// will give 0, 1, 2 or infinite solutions for t, the parameter
// of the parametric equation of the segment.
// If 0 <= t <= 1, then the intersection lies inside the segment.

```

```

// Remember to check these cases:
// if A == B == C == 0: Infinite solutions: The line coincides
// with the surface of the quadric
// else if A == B == 0 and C != 0: No solutions
// else if A == 0 and B != 0: One solution: t = -C / B
// else if B^2 - 4AC < 0: No solutions.
// else if B^2 - 4AC == 0: One solution: t = -B / 2A
// else: Two solutions: t = (-B +/- sqrt(B^2 - 4AC)) / 2A

```

Estas son las ecuaciones de las cónicas más comunes:

- *Elipsoide:*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1.$$

a , b y c son los “radios” en cada dirección (cuando $a = b = c$ se tiene una esfera).

- *Paraboloides elíptico:*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - z = 0.$$

(Cuando $a = b$ se tiene un paraboloides circular).

- *Paraboloides hiperbólico* (silla de montar o papa Pringles):

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} - z = 0.$$

.

- *Hiperboloide de una hoja:*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 1.$$

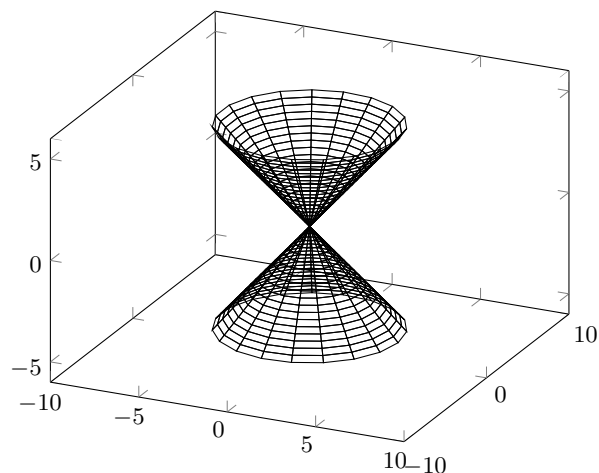
- *Hiperboloide de dos hojas:*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = -1.$$

- *Cono:*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 0.$$

(Cuando $a = b$ se tiene un cono circular).



Para mover el cono se puede usar la ecuación

$$\frac{(x - x_0)^2}{a^2} + \frac{(y - y_0)^2}{b^2} - \frac{(z - z_0)^2}{c^2} = 0$$

aunque generalmente es más fácil trasladar todos los objetos tal que el ápice del cono quede en el origen. Para cambiar el eje en el que se abre el cono, se cambia el signo menos en la ecuación y se pone en frente del eje sobre el que se quiere abrir. Para un cono que se abre en Z y que tiene radio r a la altura h (como en *Antenna in the Cinoc Mountains*) se usa la ecuación $x^2 + y^2 = (\frac{r}{h}z)^2$.

- *Cilindro elíptico:*

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1.$$

(Cuando $a = b$ se tiene un cilindro circular).

- *Planos que se intersectan:*

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 0.$$

7.20. Hallar el área de la unión de varios rectángulos

Field-testing:

- *TopCoder* - SRM 444 Div 1 Hard - PaperAndPaint

// Complexity: $O(n \log n)$, where n is the number
// of rectangles.

// Usage:

// - Set the MAXN variable below to be big enough.
// - Run `RectangleUnion::get_area(vector<Box> r)` with
// the rectangles. This will return the area of their
// union.

```
struct Box {
    long long x1, x2, y1, y2, z1, z2;

    Box(){}
    Box(long long _x1, long long _x2, long long _y1,
        long long _y2, long long _z1 = 0, long long _z2 = 0) {
        x1 = min(_x1, _x2);
        x2 = max(_x1, _x2);
        y1 = min(_y1, _y2);
        y2 = max(_y1, _y2);
        z1 = min(_z1, _z2);
        z2 = max(_z1, _z2);
    }
};
```

```
namespace RectangleUnion {
    struct Event {
        enum { OUT, IN };
        int at, y1, y2, type;
        Event(int at, int y1, int y2, int type) : at(at),
            y1(y1), y2(y2), type(type) {}

        bool operator < (const Event &t) const {
            if (at != t.at) return at < t.at;
            return type < t.type;
        }
    };
```

```

vector<int> y;

namespace SegmentTree {
    // For every rectangle we need two places in the
    // segment tree. If the segment tree has N places,
    // its size must be at least twice the first power
    // of two greater than or equal to N.
    // A safe value is 8 times the maximum number of
    // rectangles, because this always satisfies the
    // constraints above.
    const int MAXN = 8 * 55;
    long long sum[MAXN];
    int add[MAXN];

    void clear() {
        memset(sum, 0, sizeof sum);
        memset(add, 0, sizeof add);
    }

    void refresh(int u, int l, int r) {
        sum[u] = 0;
        if (add[u] > 0) sum[u] = y[r + 1] - y[l];
        else if (l < r) sum[u] = sum[2*u+1] + sum[2*u+2];
    }

    void update(int u, int l, int r, int p, int q, int what) {
        if (q < l or r < p) return; // outside
        p = max(p, l);
        q = min(q, r);
        if (l == p and q == r) {
            add[u] += what;
        } else {
            int m = (l + r) / 2;
            update(2*u+1, l, m, p, q, what);
            update(2*u+2, m+1, r, p, q, what);
        }
        refresh(u, l, r);
    }
};

// Returns the area covered by the given rectangles.
// All z values are ignored; we only consider the
// rectangles formed by the x and y values in each

```

```

// box (hence, we are finding the area covered by
// all the projections of each box onto the z = 0
// plane).
long long get_area(const vector<Box> &r) {
    y.clear();
    for (int i = 0; i < r.size(); ++i) {
        y.push_back(r[i].y1);
        y.push_back(r[i].y2);
    }
    sort(y.begin(), y.end());
    y.resize(unique(y.begin(), y.end()) - y.begin());

    vector<Event> events;
    for (int i = 0; i < r.size(); ++i) {
        // discard empty rectangle
        if (r[i].x1 >= r[i].x2 or r[i].y1 >= r[i].y2) continue;
        events.push_back(Event(r[i].x1, r[i].y1, r[i].y2,
                               Event::IN));
        events.push_back(Event(r[i].x2, r[i].y1, r[i].y2,
                               Event::OUT));
    }
    sort(events.begin(), events.end());

    SegmentTree::clear();

    long long previous_length = 0, ans = 0;
    for (int i = 0; i < events.size(); ++i) {
        if (i > 0) {
            int advanced = events[i].at - events[i - 1].at;
            ans += previous_length * advanced;
        }

        int p = lower_bound(y.begin(), y.end(), events[i].y1)
                - y.begin();
        int q = lower_bound(y.begin(), y.end(), events[i].y2)
                - y.begin();
        assert(p < q);

        if (events[i].type == Event::IN) {
            SegmentTree::update(0, 0, y.size()-2, p, q-1, +1);
        } else {
            SegmentTree::update(0, 0, y.size()-2, p, q-1, -1);
        }
    }
}

```

```

        // update to current value for next iteration
        previous_length = SegmentTree::sum[0];
    }
    return ans;
}
}
}
.....

7.21. Hallar el volumen de la unión de varios paralelepípedos

Field-testing:
    ■ Live Archive - 4969 - World of cubes

Necesita también el código de la unión de rectángulos (sección 7.20).

// Complexity:  $O(n^2 \log n)$ , where  $n$  is the number
// of boxes.
namespace BoxUnion {
    long long get_volume(const vector<Box> &r) {
        vector<int> z;
        for (int i = 0; i < r.size(); ++i) {
            z.push_back(r[i].z1);
            z.push_back(r[i].z2);
        }
        sort(z.begin(), z.end());
        z.resize(unique(z.begin(), z.end()) - z.begin());

        long long ans = 0;
        for (int i = 1; i < z.size(); ++i) {
            vector<Box> boxes;
            for (int j = 0; j < r.size(); ++j) {
                if (r[j].z1 < z[i] and z[i] <= r[j].z2) {
                    boxes.push_back(r[j]);
                }
            }
            ans += RectangleUnion::get_area(boxes)*(z[i]-z[i-1]);
        }
        return ans;
    }
}
}
.....

```

7.22. Mínima distancia entre dos puntos en la superficie de la Tierra

Field-testing:

- *UVa* - 535 - Globetrotter

```

// Returns the shortest distance on the surface of a sphere with
// radius R from point A to point B.
// Both points are described as a pair of latitude and longitude
// angles, in degrees.
//
// -90 (South Pole) <= latitude <= +90 (North Pole)
// -180 (West of meridian) <= longitude <= +180 (East of meridian)
//
double greatCircle(double laa, double loa,
                   double lab, double lob,
                   double R = 6378.0){
    const double PI = acos(-1.0);
    // Convert to radians
    laa *= PI / 180.0; loa *= PI / 180.0;
    lab *= PI / 180.0; lob *= PI / 180.0;

    double u[3] = { cos(laa) * sin(loa), cos(laa) * cos(loa),
                    sin(laa) };
    double v[3] = { cos(lab) * sin(lob), cos(lab) * cos(lob),
                    sin(lab) };
    double dot = u[0] * v[0] + u[1] * v[1] + u[2] * v[2];
    bool flip = false;
    if (dot < 0.0){
        flip = true;
        for (int i = 0; i < 3; i++) v[i] = -v[i];
    }
    double cr[3] = { u[1] * v[2] - u[2] * v[1],
                    u[2] * v[0] - u[0] * v[2],
                    u[0] * v[1] - u[1] * v[0] };
    double theta = asin(sqrt(cr[0] * cr[0] +
                             cr[1] * cr[1] +
                             cr[2] * cr[2]));

    double len = theta * R;
    if (flip) len = PI * R - len;
    return len;
}
}
.....

```

7.23. Distancias más cortas en 3D (punto a segmento, segmento a segmento y punto a triángulo)

Field-testing:

- UVa - 11836 - Star War

```
typedef long double Double;
```

```
struct Point {
    Double x, y, z;
    Point(){}
    Point(Double x, Double y, Double z) : x(x), y(y), z(z) {}

    Point operator + (const Point &p) const {
        return Point(x + p.x, y + p.y, z + p.z);
    }

    Point operator - (const Point &p) const {
        return Point(x - p.x, y - p.y, z - p.z);
    }

    Double length() const {
        return sqrtl(*this * *this);
    }

    // Dot product
    Double operator * (const Point &p) const {
        return x*p.x + y*p.y + z*p.z;
    }

    // Cross product
    Point operator ^ (const Point &p) const {
        return Point(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
    }

    void normalize() {
        Double d = length();
        x /= d; y /= d; z /= d;
    }
};

// Scalar times vector product
Point operator * (Double t, const Point &p) {
```

```
    return Point(t * p.x, t * p.y, t * p.z);
}

// Returns the shortest distance from point p to the segment
// from a to b.
// This works on 2D and 3D.
Double distance_point_to_segment(const Point &p, const Point &a,
                                const Point &b) {
    Point u = b - a;
    Point v = p - a;
    Double t = (u * v) / (u * u);
    if (t < 0.0) t = 0.0;
    if (t > 1.0) t = 1.0;
    // Actual closest point is a + t*(b - a).
    Point boom = a + t*u;
    return (boom - p).length();
}

// Returns the shortest distance between any point in the segment
// from a1 to b1 and any point in the segment from a2 to b2.
// Works in 2D and 3D.
Double distance_between_segments(const Point &a1, const Point &b1,
                                const Point &a2, const Point &b2){
    Point u = b1 - a1;
    Point v = b2 - a2;
    Point w = a1 - a2;

    Double a = u * u;
    Double b = u * v;
    Double c = v * v;
    Double d = u * w;
    Double e = v * w;

    Double den = a*c - b*b;
    Double t1, t2;

    if (cmp(den, 0.0) == 0) { // the lines are parallel
        t1 = 0;
        t2 = d / b;
    } else {
        t1 = (b * e - c * d) / den;
        t2 = (a * e - b * d) / den;
    }
}
```

```

// The shortest distance between the two infinite lines happens
// from:
// - On segment 1: p = a1 + t1 * (b1 - a1)
// - On segment 2: q = a2 + t2 * (b2 - a2)
if (0 <= t1 and t1 <= 1 and 0 <= t2 and t2 <= 1) {
    // We were lucky, the closest distance between the two
    // infinite lines happens right inside both segments.
    Point p = a1 + t1 * u;
    Point q = a2 + t2 * v;
    return (p - q).length();
} else {
    Double ans = distance_point_to_segment(a2, a1, b1);
    ans = min(ans, distance_point_to_segment(b2, a1, b1));
    ans = min(ans, distance_point_to_segment(a1, a2, b2));
    ans = min(ans, distance_point_to_segment(b1, a2, b2));
    return ans;
}
}

// Returns the shortest distance from point p to the plane defined
// by the three points [a, b, c].
// The distance is signed, and is == 0 if the point lies on the
// plane, > 0 if the point is on one side of the plane and < 0 if
// the point is on the other side.
// Intended to use on 3D.
Double signed_distance_point_to_plane(const Point &p,
    const Point &a, const Point &b, const Point &c) {
    Point n = (b - a) ^ (c - a);
    n.normalize();
    // If n * n == 0, at least two of the
    // triangle points are collinear and there's
    // no single plane defined by them.
    assert(cmp(n * n, 0) > 0);
    return (n * (p - a)) / (n * n);
}

// Returns the shortest distance from point p to the triangle
// defined by points t1, t2 and t3.
// Works in 2D and 3D.
Double distance_point_to_triangle(const Point &p,
    const Point &t1, const Point &t2, const Point &t3) {
    Point u = t2 - t1;
    Point v = t3 - t1;

```

```

    Point n = u ^ v; // plane normal

    // If n * n == 0, at least two of the
    // triangle points are collinear and there's
    // no single plane defined by them.
    assert( cmp(n * n, 0) > 0 );

    Double s = -(n * (p - t1)) / (n * n);
    // The intersection point between the plane and
    // its perpendicular line passing through p is at
    // p + s * n. This is the closest point to p that
    // lies on the plane.
    Point boom = p + s * n;

    // Let's check if boom lies inside the triangle.
    Point w = boom - t1;
    Point nv = n ^ v;
    Double a2 = (w * nv) / (u * nv);
    Point nu = n ^ u;
    Double a3 = (w * nu) / (v * nu);
    Double a1 = 1 - a2 - a3;

    // Here we have found the barycentric coordinates of point
    // boom on the plane.
    // That means that boom == a1*t1 + a2*t2 + a3*t3.
    if (0 <= a1 and a1 <= 1 and 0 <= a2 and a2 <= 1 and
        0 <= a3 and a3 <= 1) {
        // The point is strictly inside the triangle or on its
        // border, so just return the distance from p to the
        // intersection point.
        return (boom - p).length();
    } else {
        // The point is on the plane, but outside the triangle,
        // so the shortest distance is with one of the borders
        // of the triangle.
        Double ans = distance_point_to_segment(p, t1, t2);
        ans = min(ans, distance_point_to_segment(p, t2, t3));
        ans = min(ans, distance_point_to_segment(p, t3, t1));
        return ans;
    }
}

```

7.24. Punto más cercano de aproximación

Supongamos que en el tiempo $t = 0$ el avión 1 está en P_0 y el avión 2 está en Q_0 y sus vectores de velocidad por unidad de tiempo son \vec{u} y \vec{v} respectivamente.

Las posiciones de cada avión en el tiempo t están dadas por $P(t) = P_0 + t\vec{u}$ y $Q(t) = Q_0 + t\vec{v}$.

Llamemos $\vec{w}_0 = P_0 - Q_0$. Entonces el instante de tiempo en que los dos aviones están lo más cerca posible está dado por:

$$t_c = \frac{-\vec{w}_0 \cdot (\vec{u} - \vec{v})}{|\vec{u} - \vec{v}|^2}$$

siempre y cuando $|\vec{u} - \vec{v}|$ no sea 0. Si $|\vec{u} - \vec{v}| = 0$, los dos aviones viajan en la misma dirección a la misma velocidad y siempre van a estar a la misma distancia, entonces se puede simplemente usar $t_c = 0$.

Cuando $t_c < 0$, los aviones ya estuvieron lo más cerca posible en el pasado y se están alejando cada vez más a medida que avanzan.

7.25. Círculo más pequeño que envuelve una lista de puntos

Field-testing:

- *SPOJ* - ALIENS - Aliens

// Complexity is $O(n)$ (expected)

```
const int MAXN = 100005;
```

```
struct Point {
    double x,y; Point(){ } Point(double x, double y) : x(x), y(y){ }
    double length() const { return sqrt(x*x + y*y); }
};
```

```
Point operator + (const Point &a, const Point &b) {
    return Point(a.x + b.x, a.y + b.y);
}
```

```
Point operator - (const Point &a, const Point &b) {
    return Point(a.x - b.x, a.y - b.y);
}
```

```
Point operator / (const Point &a, double s) {
    return Point(a.x / s, a.y / s);
}
```

```
double operator ^ (const Point &a, const Point &b) {
    return a.x * b.y - a.y * b.x;
```

```
}
double operator * (const Point &a, const Point &b) {
    return a.x * b.x + a.y * b.y;
}

// Real algorithm starts below
typedef pair<Point, double> Circle;

bool in_circle(Circle c, Point p){
    return cmp((p - c.first).length(), c.second) <= 0;
}
```

```
Point circumcenter(Point p, Point q, Point r) {
    Point a = p - r, b = q - r;
    Point c = Point(a * (p + r) / 2, b * (q + r) / 2);
    double x = c ^ Point(a.y, b.y);
    double y = Point(a.x, b.x) ^ c;
    double d = a ^ b;
    return Point(x, y) / d;
}
```

```
Point T[MAXN];
```

```
// Fill T with the points and call spanning_circle(n).
```

```
Circle spanning_circle(int n) {
    random_shuffle(T, T + n);
    Circle ans(Point(0, 0), -INFINITY);
    for (int i = 0; i < n; i++) {
        if (!in_circle(ans, T[i])) {
            ans = Circle(T[i], 0);
            for (int j = 0; j < i; j++) {
                if (!in_circle(ans, T[j])) {
                    ans = Circle( (T[i] + T[j]) / 2,
                                   (T[i] - T[j]).length() / 2);
                    for (int k = 0; k < j; k++) {
                        if (!in_circle(ans, T[k])) {
                            Point o = circumcenter(T[i], T[j], T[k]);
                            ans = Circle(o, (o - T[k]).length());
                        }
                    }
                }
            }
        }
    }
}
```

```

    }
}
return ans;
}

```

8. Estructuras de datos

8.1. Árboles de Fenwick ó Binary indexed trees

Se tiene un arreglo $\{a_0, a_1, \dots, a_{n-1}\}$. Los árboles de Fenwick permiten encontrar $\sum_{k=i}^j a_k$ en orden $O(\log_2 n)$ para parejas de (i, j) con $i \leq j$. De la misma manera, permiten sumarle una cantidad a un a_i también en tiempo $O(\log_2 n)$.

```

class FenwickTree{
    vector<long long> v;
    int maxSize;

public:
    FenwickTree(int _maxSize) : maxSize(_maxSize+1) {
        v = vector<long long>(maxSize, 0LL);
    }

    void add(int where, long long what){
        for (where++; where <= maxSize; where += where & -where){
            v[where] += what;
        }
    }

    long long query(int where){
        long long sum = v[0];
        for (where++; where > 0; where -= where & -where){
            sum += v[where];
        }
        return sum;
    }

    long long query(int from, int to){
        return query(to) - query(from-1);
    }
}

```

```
};
```

8.2. Segment tree

```

class SegmentTree{
public:
    vector<int> arr, tree;
    int n;

    SegmentTree(){}
    SegmentTree(const vector<int> &arr) : arr(arr) {
        initialize();
    }

    //must be called after assigning a new arr.
    void initialize(){
        n = arr.size();
        tree.resize(4*n + 1);
        initialize(0, 0, n-1);
    }

    int query(int query_left, int query_right) const{
        return query(0, 0, n-1, query_left, query_right);
    }

    void update(int where, int what){
        update(0, 0, n-1, where, what);
    }

private:
    int initialize(int node, int node_left, int node_right);
    int query(int node, int node_left, int node_right,
               int query_left, int query_right) const;
    void update(int node, int node_left, int node_right,
                int where, int what);
};

int SegmentTree::initialize(int node,
                             int node_left, int node_right){
    if (node_left == node_right){

```



```

    tree[node] = node_left;
    return tree[node];
}
int half = (node_left + node_right) / 2;
int ans_left = initialize(2*node+1, node_left, half);
int ans_right = initialize(2*node+2, half+1, node_right);

if (arr[ans_left] <= arr[ans_right]){
    tree[node] = ans_left;
}else{
    tree[node] = ans_right;
}
return tree[node];
}

int SegmentTree::query(int node, int node_left, int node_right,
    int query_left, int query_right) const{
    if (node_right < query_left || query_right < node_left)
        return -1;
    if (query_left <= node_left && node_right <= query_right)
        return tree[node];

    int half = (node_left + node_right) / 2;
    int ans_left = query(2*node+1, node_left, half,
        query_left, query_right);
    int ans_right = query(2*node+2, half+1, node_right,
        query_left, query_right);

    if (ans_left == -1) return ans_right;
    if (ans_right == -1) return ans_left;

    return(arr[ans_left] <= arr[ans_right] ? ans_left : ans_right);
}

void SegmentTree::update(int node, int node_left, int node_right,
    int where, int what){
    if (where < node_left || node_right < where) return;
    if (node_left == where && where == node_right){
        arr[where] = what;
        tree[node] = where;
        return;
    }
    int half = (node_left + node_right) / 2;

```

```

    if (where <= half){
        update(2*node+1, node_left, half, where, what);
    }else{
        update(2*node+2, half+1, node_right, where, what);
    }
    if (arr[tree[2*node+1]] <= arr[tree[2*node+2]]){
        tree[node] = tree[2*node+1];
    }else{
        tree[node] = tree[2*node+2];
    }
}
}

```

8.3. Treap

```
#define null NULL
```

```

struct Node {
    int x, y, size;
    long long sum;
    Node *l, *r;
    Node(int k) : x(k), y(rand()), size(1),
        l(null), r(null), sum(0) { }
};

```

```

Node* relax(Node* p) {
    if (p) {
        p->size = 1;
        p->sum = p->x;
        if (p->l) {
            p->size += p->l->size;
            p->sum += p->l->sum;
        }
        if (p->r) {
            p->size += p->r->size;
            p->sum += p->r->sum;
        }
    }
    return p;
}

```

```
// Puts all elements <= x in l and all elements > x in r.
```

```

void split(Node* t, int x, Node* &l, Node* &r) {
    if (t == null) l = r = null; else {
        if (t->x <= x) {
            split(t->r, x, t->r, r);
            l = relax(t);
        } else {
            split(t->l, x, l, t->l);
            r = relax(t);
        }
    }
}

```

```

Node* merge(Node* l, Node *r) {
    if (l == null) return relax(r);
    if (r == null) return relax(l);
    if (l->y > r->y) {
        l->r = merge(l->r, r);
        return relax(l);
    } else {
        r->l = merge(l, r->l);
        return relax(r);
    }
}

```

```

Node* insert(Node* t, Node* m) {
    if (t == null || m->y > t->y) {
        split(t, m->x, m->l, m->r);
        return relax(m);
    }
    if (m->x < t->x) t->l = insert(t->l, m);
    else t->r = insert(t->r, m);
    return relax(t);
}

```

```

Node* erase(Node* t, int x) {
    if (t == null) return null;
    if (t->x == x) {
        Node *q = merge(t->l, t->r);
        delete t;
        return relax(q);
    } else {
        if (x < t->x) t->l = erase(t->l, x);
        else t->r = erase(t->r, x);
    }
}

```

```

        return relax(t);
    }
}

```

// Returns any node with the given x.

```

Node* find(Node* cur, int x) {
    while (cur != null and cur->x != x) {
        if (x < cur->x) cur = cur->l;
        else cur = cur->r;
    }
    return cur;
}

```

```

long long sum(Node* p, int x) { // find the sum of elements <= x
    if (p == null) return 0LL;
    if (p->x > x) return sum(p->l, x);
    long long ans = (p->l ? p->l->sum : 0) + p->x + sum(p->r, x);
    assert(ans >= 0);
    return ans;
}

```

.....

8.4. Rope

```
#define null NULL
```

```

struct Node {
    int y, size, payload;
    long long sum;
    Node *l, *r;
    bool pending_reversal;
    Node(int k) : payload(k), y(rand()), size(1),
                 l(null), r(null), sum(0),
                 pending_reversal(false) { }
};

```

```

Node* relax(Node* p) {
    if (p) {
        p->size = 1;
        p->sum = p->payload;
        if (p->l) {
            p->size += p->l->size;

```

```

        p->sum += p->l->sum;
    }
    if (p->r) {
        p->size += p->r->size;
        p->sum += p->r->sum;
    }
}
return p;
}

Node* propagate(Node * t) {
    assert(t);
    if (t->pending_reversal) {
        swap(t->l, t->r);
        t->pending_reversal = false;
        if (t->l) t->l->pending_reversal ^= 1;
        if (t->r) t->r->pending_reversal ^= 1;
        relax(t);
    }
    return t;
}

int leftCount(Node * t) {
    assert(t);
    return t->l ? t->l->size : 0;
}

// moves elements at positions [0..x] to l,
// and those at positions [x+1, n-1] to r.

void split(Node* t, int x, Node* &l, Node* &r) {
    if (t == null) l = r = null; else {
        // apply lazy propagation here to t
        propagate(t);
        if (x < leftCount(t)) {
            split(t->l, x, l, t->l);
            r = relax(t);
        } else {
            split(t->r, x - 1 - leftCount(t), t->r, r);
            l = relax(t);
        }
    }
}

```

```

}

Node* merge(Node* l, Node *r) {
    if (l == null) return relax(r);
    if (r == null) return relax(l);

    if (l->y > r->y) {
        // apply lazy propagation here to l
        propagate(l);
        l->r = merge(l->r, r);
        return relax(l);
    } else {
        // apply lazy propagation here to r
        propagate(r);
        r->l = merge(l, r->l);
        return relax(r);
    }
}

// Inserts node m at position x (0-based)
Node* insert(Node* t, Node* m, int x) {
    if (t == null || m->y > t->y) {
        split(t, x - 1, m->l, m->r);
        return relax(m);
    }
    // apply lazy propagation here to t
    propagate(t);
    if (x <= leftCount(t)) t->l = insert(t->l, m, x);
    else t->r = insert(t->r, m, x - 1 - leftCount(t));
    return relax(t);
}

Node* erase(Node* t, int x) {
    if (t == null) return null;
    // apply lazy propagation here to t
    propagate(t);
    if (leftCount(t) == x) {
        Node *q = merge(t->l, t->r);
        delete t;
        return relax(q);
    } else {
        if (x < leftCount(t)) t->l = erase(t->l, x);
        else t->r = erase(t->r, x - 1 - leftCount(t));
    }
}

```

```

        return relax(t);
    }
}

Node* find(Node* cur, int x) {
    while (cur != null) {
        propagate(cur);
        if (leftCount(cur) == x) break;
        else if (x < leftCount(cur)) cur = cur->l;
        else {
            x = x - 1 - leftCount(cur);
            cur = cur->r;
        }
    }
    return cur;
}

void traverse(Node * p) {
    if (p == null) return;
    propagate(p);
    traverse(p->l);
    printf("%d ", p->payload);
    traverse(p->r);
}

////////////////////////////////////
// Sample Usage:                                     //
////////////////////////////////////

// Given an array of n integers, support these two operations:
// 0 - Find the sum of all elements in positions l through
//     r, inclusive.
// 1 - Reverse the elements in positions l through r,
//     inclusive.

void usage() {
    // n = Number of elements in the array
    // m = Number of queries
    int n, m;
    scanf("%d %d", &n, &m);

    // Build the rope with n elements

```

```

    int x; scanf("%d", &x);
    Node * root = new Node(x);
    for (int i = 1; i < n; ++i) {
        scanf("%d", &x);
        root = merge(root, new Node(x));
    }

    for (int i = 0; i < m; ++i) {
        int type, l, r; scanf("%d %d %d", &type, &l, &r);
        // the range [l, r] was 1-based in the input,
        // so convert it to 0-based
        l--, r--;

        // Split the rope in three little ropes.
        // b is the segment we care about, since
        // it holds the subarray [l, r].
        Node * a, * b, * c;
        split(root, l - 1, a, b);
        split(b, r - l, b, c);

        if (type == 0) { // output the sum of the segment
            cout << b->sum << '\n';
        } else { // reverse the segment
            assert( !b->pending_reversal );
            b->pending_reversal = true;
        }
        root = merge(a, merge(b, c));
    }
}

.....

8.5. Range Minimum Query
8.5.1. Con tabla

// Usage:
// N - Set to size of the array.
// height - Fill with the data you want to run RMQ on.
//
// You might want to redefine the "better" function if
// you are using a different comparison operator (For
// example, to solve Range Maximum Query)

```

```

//
//
// RMQ::build(); // Preprocess
// RMQ::get(i, j); // Returns the index
//                  // of the smallest value
//                  // in range [i, j]
//

int height[MAXN];
int N;

namespace RMQ {
    const int L = 18; // 1 + log2(MAXN)
    int p[MAXN][L];

    int better(int a, int b) {
        assert(0 <= a and a < N and 0 <= b and b < N);
        return height[a] <= height[b] ? a : b;
    }

    void build() {
        for (int i = 0; i < N; ++i) {
            p[i][0] = i;
        }
        for (int j = 1; j < L; ++j) {
            for (int i = 0; i < N; ++i) {
                p[i][j] = p[i][j - 1];
                int length = 1 << (j - 1);
                if (i + length < N) {
                    p[i][j] = better(p[i][j - 1], p[i + length][j - 1]);
                }
            }
        }
    }
    // Returns the index of the best value in range [from, to]
    int get(int from, int to) {
        int length = to - from + 1;
        int j = 0;
        while ((1 << (j + 1)) < length) j++;
        assert(j < L);
        return better(p[from][j], p[to - (1 << j) + 1][j]);
    }
};

```

8.5.2. Con segment tree

```

.....

// Usage:
// N - Set to size of the array.
// height - Fill with the data you want to run RMQ on.
//
// You might want to redefine the "better" function if
// you are using a different comparison operator (For
// example, to solve Range Maximum Query)
//
// SegmentTree::build(0, 0, N - 1); // Preprocess
// SegmentTree::get(0, 0, N - 1, i, j); // Returns the index
//                                     // of the smallest value
//                                     // in range [i, j]
//

int height[MAXN];
int N;

namespace SegmentTree {
    int tree[1 << 18];

    int better(int a, int b) {
        assert(0 <= a and a < N and 0 <= b and b < N);
        return height[a] <= height[b] ? a : b;
    }

    int build(int u, int l, int r) {
        if (l == r) return tree[u] = l;
        int m = (l + r) / 2;
        int a = build(2*u+1, l, m);
        int b = build(2*u+2, m+1, r);
        return tree[u] = better(a, b);
    }

    // Returns the INDEX of the best element in range [p, q]
    int get(int u, int l, int r, int p, int q) {
        assert(l <= p and q <= r and p <= q);
        if (l == p and r == q) return tree[u];
        int m = (l + r) / 2;

```

```

    if (q <= m) return get(2*u+1, l, m, p, q);
    if (m + 1 <= p) return get(2*u+2, m+1, r, p, q);
    int a = get(2*u+1, l, m, p, m);
    int b = get(2*u+2, m+1, r, m+1, q);
    return better(a, b);
}
};

```

9. Misceláneo

9.1. Problema de Josephus

Hay n personas enumeradas de 0 a $n - 1$ paradas en un círculo. Un verdugo empieza a contar personas en sentido horario, empezando en la persona 0. Cuando la cuenta llega a k , el verdugo mata a la última persona contada y vuelve a empezar a contar a partir siguiente persona. ¿Quién es el sobreviviente?

Por ejemplo, si $n = 7$ y $k = 3$, el orden en que se eliminan las personas es 2, 5, 1, 6, 4, 0 y 3. El sobreviviente es 3.

```

// Recursivo:
// n <= 2500
// Es O(n), pero mejor usar la versión iterativa
// para evitar stack overflows.
int joseph(int n, int k) {
    assert(n >= 1);
    if (n == 1) return 0;
    return (joseph(n - 1, k) + k) % n;
}

```

```

// Iterativo:
// n <= 10^6, k <= 2^31 - 1
int joseph(int n, int k) {
    int ans = 0;
    for (int i = 2; i <= n; ++i) {
        ans = (ans + k) % i;
    }
    return ans;
}

```

// Más rápido:

```

// n <= 10^18, k <= 2500
// Posiblemente se puede incrementar k un poco más
// si se cambia la línea marcada con *** por la versión
// iterativa, para evitar stack overflows.
long long joseph(long long n, int k) {
    assert(n >= 1);
    if (n == 1) return 0LL;
    if (k == 1) return n - 1;
    if (n < k) return (joseph(n - 1, k) + k) % n; // ***
    long long w = joseph(n - n / k, k) - n % k;
    if (w < 0) return w + n;
    return w + w / (k - 1);
}

```

9.2. Distancia más corta para un caballo en un tablero de ajedrez infinito

```

// Returns the number of moves a knight must make
//to go from (x1, y1) to (x2, y2) in an INFINITE
// chess board.
long long knight_distance(long long x1, long long y1,
                           long long x2, long long y2) {
    long long x = abs(x2 - x1), y = abs(y2 - y1);
    if (x > y) swap(x, y);
    assert(x <= y);
    if (x == 0 and y == 1) return 3;
    if (x == 2 and y == 2) return 4;
    long long d = max((y + 1) / 2, (x + y + 2) / 3);
    if (d % 2 != (x + y) % 2) d++;
    return d;
}

```

9.3. Trucos con bits

9.3.1. Iterar sobre los subconjuntos de una máscara

```

for(int m = mask; m > 0; m = (m-1) & mask){
    // m tiene un subconjunto de mask
}

```

9.4. El *parser* más rápido del mundo

- Cada no-terminal: un método
- Cada lado derecho:
 - invocar los métodos de los no-terminales o
 - Cada terminal: invocar proceso *match*
- Alternativas en una producción: se hace un *if*

No funciona con gramáticas recursivas por izquierda ó en las que en algún momento haya varias posibles escogencias que empiezan por el mismo caracter (En ambos casos la gramática se puede factorizar).

Ejemplo: Para la gramática:

$$A \rightarrow (A)A$$

$$A \rightarrow \epsilon$$

```
//A -> (A)A | Epsilon
```

```
#include <iostream>
#include <string>
```

```
using namespace std;
```

```
bool ok;
char sgte;
int i;
string s;
```

```
bool match(char c){
    if (sgte != c){
        ok = false;
    }
    sgte = s[++i];
}
```

```
void A(){
    if (sgte == '('){
        match('(');
        A(); match(')'); A();
    }else if (sgte == '$' || sgte == ')'){
```

```
        //nada
    }else{
        ok = false;
    }
}

int main(){
    while(getline(cin, s) && s != ""){
        ok = true;
        s += '$';
        sgte = s[(i = 0)];
        A();
        if (i < s.length()-1) ok=false; //No consumí toda la cadena
        if (ok){
            cout << "Accepted\n";
        }else{
            cout << "Not accepted\n";
        }
    }
}
```

9.5. Checklist para corregir un Wrong Answer

Consideraciones que podrían ser causa de un Wrong Answer:

- Overflow.
- El programa termina anticipadamente por la condición en el ciclo de lectura. Por ejemplo, se tiene `while (cin >> n >> k && n && k)` y un caso válido de entrada es `n = 1` y `k = 0`.
- El grafo no es conexo.
- Puede haber varias aristas entre el mismo par de nodos.
- Las aristas pueden tener costos negativos.
- El grafo tiene un sólo nodo.
- La cadena puede ser vacía.
- Las líneas pueden tener espacios en blanco al principio o al final (Cuidado al usar `getline` o `fgets`).

- El arreglo no se limpia entre caso y caso.
- Estás imprimiendo una línea en blanco con un espacio (`printf(" \n")` en vez de `printf("\n")` ó `puts(" ")` en vez de `puts("")`).
- Hay pérdida de precisión al leer variables como `double` y convertirlas a enteros. Por ejemplo, en C++, `floor(0.29 * 100) == 28`.
- La rana se puede quedar quieta.
- El producto cruz está invertido. Realmente es

$$|\langle a_x, a_y \rangle \times \langle b_x, b_y \rangle| = a_x \mathbf{b}_y - a_y \mathbf{b}_x \neq a_x \mathbf{b}_x - a_y \mathbf{b}_y.$$

- Hay una resta módulo m pero no se revisa si el resultado es negativo. Se corrige haciendo

```
ans = (ans - b) % mod;
if (ans < 0) ans += mod;    // !
```

9.6. Redondeo de dobles

Para redondear un doble a k cifras, usar:

$$\frac{\lfloor 10^k \cdot x + 0,5 \rfloor}{10^k}$$

Ejemplo:

```
double d = 1.2345;
d = floor(1000 * d + 0.5) / 1000;
```

Al final, `d` es 1.235.

9.6.1. Convertir un doble al entero más cercano

| Código | Valores originales (d) | Nuevos valores (k) |
|---|--|------------------------|
| <pre>int k = floor(d + 0.5 + EPS); (con EPS = 1e-9)</pre> | 0.0 | 0 |
| | 0.1 | 0 |
| | 0.5 | 1 |
| | 0.4999999999999999 | 1 |
| | <code>cos(1e-7) * 0.5 =</code> 0.4999999999999975 | 1 |
| | 0.9 | 1 |
| | 1.0 | 1 |
| | 1.4 | 1 |
| | 1.5 | 2 |
| | 1.6 | 2 |
| | 1.9 | 2 |
| | 2.0 | 2 |
| | 2.1 | 2 |
| | -0.0 | 0 |
| | -0.1 | 0 |
| | -0.5 | 0 |
| | -0.4999999999999999 | 0 |
| | <code>-cos(1e-7) * 0.5 =</code> -0.4999999999999975 | 0 |
| | -0.9 | -1 |
| | -1.0 | -1 |
| | -1.4 | -1 |
| | -1.5 | -1 |
| | -1.6 | -2 |
| | -1.9 | -2 |
| | -2.0 | -2 |
| | -2.1 | -2 |

| Código | Valores originales (d) | Nuevos valores (k) |
|-------------------------|---|--------------------|
| int k = floor(d + 0.5); | 0.0 | 0 |
| | 0.1 | 0 |
| | 0.5 | 1 |
| | 0.499999999999999 | 0 |
| | cos(1e-7) * 0.5 = 0.4999999999999975 | 0 |
| | 0.9 | 1 |
| | 1.0 | 1 |
| | 1.4 | 1 |
| | 1.5 | 2 |
| | 1.6 | 2 |
| | 1.9 | 2 |
| | 2.0 | 2 |
| | 2.1 | 2 |
| | -0.0 | 0 |
| | -0.1 | 0 |
| | -0.5 | 0 |
| | -0.499999999999999 | 0 |
| | -cos(1e-7) * 0.5 = -0.4999999999999975 | 0 |
| | -0.9 | -1 |
| | -1.0 | -1 |
| | -1.4 | -1 |
| | -1.5 | -1 |
| | -1.6 | -2 |
| | -1.9 | -2 |
| | -2.0 | -2 |
| | -2.1 | -2 |

9.6.2. Redondear un doble a cierto número de cifras de precisión

10. Java

10.1. Entrada desde entrada estándar

Este primer método es muy fácil pero es mucho más ineficiente porque utiliza Scanner en vez de BufferedReader:

```
import java.io.*;
import java.util.*;
```

```
class Main{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()){
            String s= sc.nextLine();
            System.out.println("Leí: " + s);
        }
    }
}
```

.....

Este segundo es más rápido:

```
import java.util.*;
import java.io.*;
import java.math.*;
```

```
class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader reader =
            new BufferedReader(new InputStreamReader(System.in));
        String line = reader.readLine();
        StringTokenizer tokenizer = new StringTokenizer(line);
        int N = Integer.valueOf(tokenizer.nextToken());
        while (N-- > 0){
            String a, b;
            a = reader.readLine();
            b = reader.readLine();

            int A = a.length(), B = b.length();
            if (B > A){
```



```

System.out.println(m.get("Objeto inexistente"));

/*
 * Sets
 * La misma diferencia entre TreeSet y HashSet.
 */
System.out.println("\nSets");
/*
 * *OJO: El HashSet no está en orden, el TreeSet sí.
 */
//HashSet<Integer> s = new HashSet<Integer>();
TreeSet < Integer > s = new TreeSet < Integer > ();
s.add(3576);
s.add(new Integer("54"));
s.add(new Integer(1000000007));

if (s.contains(54)) {
    System.out.println("54 presente.");
}

if (s.isEmpty() == false) {
    System.out.println("s.size() = " + s.size());
    Iterator < Integer > i = s.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
        i.remove();
    }
    System.out.println("s.size() = " + s.size());
}
}
}
}

```

.....
La salida de este programa es:

```

Maps
m.size() = 1
465
null

Sets
54 presente.
s.size() = 3
54
3576
1000000007
s.size() = 0

```

Si quiere usarse una clase propia como llave del mapa o como elemento del set, la clase debe implementar algunos métodos especiales: Si va a usarse un TreeMap ó TreeSet hay que implementar los métodos `compareTo` y `equals` de la interfaz `Comparable` como en la sección 10.4. Si va a usarse un HashMap ó HashSet hay más complicaciones.

Sugerencia: Inventar una manera de codificar y decodificar la clase en una String o un Integer y meter esa representación en el mapa o set: esas clases ya tienen los métodos implementados.

10.4. Colas de prioridad

Hay que implementar unos métodos. Veamos un ejemplo:

```

import java.util.*;

class Item implements Comparable<Item>{
    int destino, peso;

    Item(int destino, int peso){
        this.peso = peso;
        this.destino = destino;
    }
    /*
     * Implementamos toda la javazofia.
     */
    public int compareTo(Item otro){
        // Return < 0 si this < otro
        // Return 0 si this == otro
    }
}

```

```

        // Return > 0 si this > otro
        /* Un nodo es menor que otro si tiene menos peso */
        return peso - otro.peso;
    }
    public boolean equals(Object otro){
        if (otro instanceof Item){
            Item ese = (Item)otro;
            return destino == ese.destino && peso == ese.peso;
        }
        return false;
    }
    public String toString(){
        return "peso = " + peso + ", destino = " + destino;
    }
}

class Ejemplo {
    public static void main(String[] args) {
        PriorityQueue<Item> q = new PriorityQueue<Item>();
        q.add(new Item(12, 0));
        q.add(new Item(4, 1876));
        q.add(new Item(13, 0));
        q.add(new Item(8, 0));
        q.add(new Item(7, 3));
        while (!q.isEmpty()){
            System.out.println(q.poll());
        }
    }
}

```

.....

La salida de este programa es:

| |
|--------------------------|
| peso = 0, destino = 12 |
| peso = 0, destino = 8 |
| peso = 0, destino = 13 |
| peso = 3, destino = 7 |
| peso = 1876, destino = 4 |

Vemos que la función de comparación que definimos no tiene en cuenta destino, por eso no desempata cuando dos Items tienen el mismo peso si no que escoge cualquiera de manera arbitraria.

11. C++

11.1. Entrada desde archivo

```

#include <iostream>
#include <fstream>

using namespace std;

int _main(){
    freopen("entrada.in", "r", stdin);
    freopen("entrada.out", "w", stdout);

    string s;
    while (cin >> s){
        cout << "Leí " << s << endl;
    }
    return 0;
}

int main(){
    ifstream fin("entrada.in");
    ofstream fout("entrada.out");

    string s;
    while (fin >> s){
        fout << "Leí " << s << endl;
    }
    return 0;
}

```

11.2. Strings con caracteres especiales

```

#include <iostream>
#include <cassert>
#include <stdio.h>
#include <assert.h>
#include <wchar.h>
#include <wctype.h>
#include <locale.h>

```

```
using namespace std;

int main(){
    assert(setlocale(LC_ALL, "en_US.UTF-8") != NULL);
    wchar_t c;

    wstring s;
    while (getline(wcin, s)){
        wcout << L"Leí : " << s << endl;
        for (int i=0; i<s.size(); ++i){
            c = s[i];
            wprintf(L"%lc %lc\n", tolower(s[i]), toupper(s[i]));
        }
    }

    return 0;
}
```

.....

Nota: Como alternativa a la función `getline`, se pueden utilizar las funciones `fgetws` y `fputws`, y más adelante `swscanf` y `wprintf`:

```
#include <iostream>
#include <cassert>
#include <stdio.h>
#include <assert.h>
#include <wchar.h>
#include <wctype.h>
#include <locale.h>
using namespace std;
int main(){
    assert(setlocale(LC_ALL, "en_US.UTF-8") != NULL);
    wchar_t in_buf[512], out_buf[512];
    swprintf(out_buf, 512,
        L"¿Podrías escribir un número?, Por ejemplo %d. "
        L"¡Gracias, pinguino español!\n", 3);
    fputws(out_buf, stdout);
    fgetws(in_buf, 512, stdin);
    int n;
    swscanf(in_buf, L"%d", &n);
    swprintf(out_buf, 512,
        L"Escribiste %d, yo escribo ¿0ïàÜÑ~\n", n);
```

```
fputws(out_buf, stdout);
return 0;
}
```

11.3. Imprimir un doble con `cout` con cierto número de cifras de precisión

Tener cuidado con números negativos, porque el comportamiento es diferente.

```
#include <iomanip>
```

```
cout << fixed << setprecision(3) << 1.1225 << endl;
```

.....

The centroid of a [semicircle](#) of radius R is given by

$$\bar{x} = \frac{2R}{\pi}.$$

The centroids of several common laminae bounded by the following curves along the nonsymmetrical axis are summarized in the following table.

| lamina | \bar{y} |
|------------------------------------|--|
| circular sector | $\frac{4R \sin\left(\frac{1}{2}\theta\right)}{3\theta}$ |
| circular segment | $\frac{4R \sin^3\left(\frac{1}{2}\theta\right)}{3(\theta - \sin\theta)}$ |
| isosceles triangle | $\frac{1}{3}h$ |
| parabolic segment | $\frac{2}{5}h$ |
| semicircle | $\frac{4R}{3\pi}$ |

In three dimensions, the mass of a solid with density function $\rho(x, y, z)$ is

$$M = \iiint \rho(x, y, z) dV,$$

and the coordinates of the center of mass are

| | | |
|-----------|---|--|
| \bar{x} | = | $\frac{\iiint x \rho(x, y, z) dV}{M}$ |
| \bar{y} | = | $\frac{\iiint y \rho(x, y, z) dV}{M}$ |
| \bar{z} | = | $\frac{\iiint z \rho(x, y, z) dV}{M},$ |

| | |
|---------------------------------|---|
| solid | \bar{z} |
| cone | $\frac{1}{4} h$ |
| conical frustum | $\frac{h (R_1^2 + 2 R_1 R_2 + 3 R_2^2)}{4 (R_1^2 + R_1 R_2 + R_2^2)}$ |
| half- ellipsoid | $\frac{3}{16} a, \frac{3}{16} b, \frac{3}{16} c$ |
| hemisphere | $\frac{3}{8} R$ |
| paraboloid | $\frac{2}{3} h$ |
| pyramid | $\frac{1}{4} h$ |
| spherical cap | $\frac{3 (2 R - h)^2}{4 (3 R - h)}$ |
| vault | $\frac{3}{8} r$ |

```
import java.awt.geom.Line2D;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Monster {

    ////////// Geometry Library //////////

    private static final double EPS = 1e-10;

    private static int cmp(double x, double y) {
        return (x <= y + EPS) ? (x + EPS < y) ? -1 : 0 : 1;
    }

    // Immutable Point Class.
    private static class Point implements Comparable<Point> {
        public double x;
        public double y;
        public Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
        public Point() {
            this.x = 0.0;
            this.y = 0.0;
        }
        public double dotProduct(Point o) {
            return this.x * o.x + this.y * o.y;
        }
        public double crossProduct(Point o) {
            return this.x * o.y - this.y * o.x;
        }
        public Point add(Point o) {
            return new Point(this.x + o.x, this.y + o.y);
        }
        public Point subtract(Point o) {
            return new Point(this.x - o.x, this.y - o.y);
        }
        public Point multiply(double m) {
            return new Point(this.x * m, this.y * m);
        }
        public Point divide(double m) {
            return new Point(this.x / m, this.y / m);
        }
        @Override
        public int compareTo(Point o) {
            if (this.x < o.x) return -1;
            if (this.x > o.x) return 1;
            if (this.y < o.y) return -1;
            if (this.y > o.y) return 1;
            return 0;
        }
        // Euclidean distance between two points;
        double distance(Point o) {
            double d1 = x - o.x, d2 = y - o.y;
            return Math.sqrt(d1 * d1 + d2 * d2);
        }
    }
}
```



```

}

// Calculates the angle between the two vectors defined by p - r and q - r.
// The formula comes from the definition of the dot and the cross product:
//
//  $A \cdot B = |A||B|\cos(c)$ 
//  $A \times B = |A||B|\sin(c)$ 
//
//  $\frac{\sin(c)}{\cos(c)} = \frac{A \times B}{A \cdot B} = \tan(c)$ 
private static double angle(Point p, Point q, Point r) {
    Point u = p.subtract(r), v = q.subtract(r);
    return Math.atan2(u.crossProduct(v), u.dotProduct(v));
}

// Calculates sign of the turn between the two vectors defined by <p-r> and
// <q-r>.
//
// Just to remember, the cross product is defined by  $(x1 * y2) - (x2 * y1)$  and
// is negative if it is a right turn and positive if it is a left turn. e.g.
//
//      .p3
//      ^
//      /
// .p2 /
// ^ /
// | /
// .p1
// The cross product between the vectors <p2-p1> and <p3-p1> is negative, that
// means it is a right turn.
private static int turn(Point p, Point q, Point r) {
    return cmp((p.subtract(r)).crossProduct(q.subtract(r)), 0.0);
}

// Decides if the point r is inside the segment defined by the points p and q.
// To do this, we have to check two conditions:
// 1. That the turn between the two vectors formed by p - q and r - q is zero
// (that means they are parallel).
// 2. That the dot product between the vector formed by p - r and q - r (that
// means the testing point as the initial point for both vectors) is less than
// or equal to zero (that means that the two vectors have opposite direction).
private static boolean between(Point p, Point q, Point r) {
    return turn(p, r, q) == 0 && cmp((p.subtract(r)).dotProduct(q.subtract(r)), 0.0) <= 0;
}

// Returns 0, -1 or 1 depending if p is in the exterior, the frontier or the
// interior of the given polygon respectively, the polygon must be in clockwise
// or counterclockwise order [MANDATORY!!].
// The idea is to iterate over each of the points in the polygon and consider
// the segment formed by two adjacent points, if the test points is inside that
// segment, the point is in the frontier, if not, we add the angles inside the
// vectors formed by the two points of the polygon and the test point. For a
// point outside the polygon this sum is zero because the angles cancel
// themselves.
private static int inPolygon(Point p, Point[] polygon, int polygonSize) {
    double a = 0; int N = polygonSize;
    for (int i = 0; i < N; ++i) {
        if (between(polygon[i], polygon[(i + 1) % N], p)) return -1;
        a += angle(polygon[i], polygon[(i + 1) % N], p);
    }

```

```

    }
    return (cmp(a, 0.0) == 0) ? 0 : 1;
}

private static Point GetIntersection(Line2D.Double l1, Line2D.Double l2) {
    double A1 = l1.y2 - l1.y1;
    double B1 = l1.x1 - l1.x2;
    double C1 = A1 * l1.x1 + B1 * l1.y1;
    double A2 = l2.y2 - l2.y1;
    double B2 = l2.x1 - l2.x2;
    double C2 = A2 * l2.x1 + B2 * l2.y1;
    double det = A1*B2 - A2*B1;
    if(det == 0){
        // Lines are parallel, check if they are on the same line.
        double m1 = A1 / B1;
        double m2 = A2 / B2;
        // Check whether their slopes are the same or not, or if they are vertical.
        if (cmp(m1, m2) == 0 || (B1 == 0 && B2 == 0)) {
            if ((l1.x1 == l2.x1 && l1.y1 == l2.y1) ||
                (l1.x1 == l2.x2 && l1.y1 == l2.y2)) return new Point(l1.x1, l1.y1);
            if ((l1.x2 == l2.x1 && l1.y2 == l2.y1) ||
                (l1.x2 == l2.x2 && l1.y2 == l2.y2)) return new Point(l1.x2, l1.y2);
        }
        return null;
    }
    double x = (B2*C1 - B1*C2) / det;
    double y = (A1*C2 - A2*C1) / det;
    return new Point(x, y);
}

```

```

////////////////////////////////////

```

```

private static Line2D.Double[] lines;
private static List<Integer>[] graph;
private static int[] marked;
private static int[] stack;
private static int[] cycle;
private static int stackLen;
private static int cycleLen;
private static boolean res;

private static void FindCycle(int node) {
    cycleLen = 0;
    cycle[cycleLen++] = node;
    int k = stackLen - 1;
    while (stack[k] != node) {
        cycle[cycleLen++] = stack[k];
        --k;
    }
    cycle[cycleLen++] = stack[k];
    Point[] points = new Point[cycleLen];
    for (int i = 0; i < cycleLen - 1; ++i) {
        points[i] = GetIntersection(lines[cycle[i]], lines[cycle[i + 1]]);
    }
    if (inPolygon(new Point(), points, cycleLen - 1) != 0) res = true;
}

private static void DoIt(int act, int last) {
    marked[act] = 1;
    stack[stackLen++] = act;
    for (Integer i : graph[act]) {

```

```
        if (marked[i] == 1 && i != last) {
            FindCycle(i);
        } else if (marked[i] == 0) {
            DoIt(i, act);
        }
    }
    --stackLen;
    marked[act] = 2;
}

@SuppressWarnings("unchecked")
public static void main(String[] args) throws IOException {
    System.setIn(new FileInputStream("monster.in"));
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    String[] parts;
    while (true) {
        int num = Integer.parseInt(reader.readLine());
        if (num == 0) break;
        lines = new Line2D.Double[num];
        for (int i = 0; i < num; ++i) {
            parts = reader.readLine().split("[ ]+");
            lines[i] = new Line2D.Double(Integer.parseInt(parts[0]),
                Integer.parseInt(parts[1]), Integer.parseInt(parts[2]),
                Integer.parseInt(parts[3]));
        }
        graph = (List<Integer>[]) new List[num];
        for (int i = 0; i < num; ++i) {
            graph[i] = new ArrayList<Integer>();
        }
        for (int i = 0; i < num; ++i) {
            for (int j = i + 1; j < num; ++j) {
                if (lines[i].intersectsLine(lines[j])) {
                    graph[i].add(j);
                    graph[j].add(i);
                }
            }
        }

        res = false;
        marked = new int[num];
        stack = new int[num];
        cycle = new int[num + 1];
        stackLen = 0;
        Arrays.fill(marked, 0);
        for (int i = 0; i < num; ++i) {
            if (marked[i] != 0) continue;
            DoIt(i, -1);
        }

        if (res) System.out.println("yes");
        else System.out.println("no");
    }
}
```

```

#include ...
using namespace std;

const int MAXN = 105;
int g[2 * MAXN][2 * MAXN]; // 0 source, n+m+1 sink
int flow[2 * MAXN][2 * MAXN];
int prev[2 * MAXN];
pair <double, double> gophers [MAXN];
pair <double, double> holes [MAXN];

bool canReach(int i, int j, double max_dist){
    double x1 = gophers[i].first; double y1 = gophers[i].second;
    double x2 = holes[j].first; double y2 = holes[j].second;
    double d = sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
    if (d <= max_dist) return true;
    return false;
}

int max_flow(int s, int t){
    int max_flow = 0;

    for (int i = s; i <= t; i++)
        for (int j = s; j <= t; j++)
            flow[i][j] = 0;

    while (true){
        // Find path s to t
        for (int i = s; i <= t; i++)
            prev[i] = -1;

        queue <int> q;
        q.push(s);
        prev[s] = -2;
        while (q.size() > 0){
            int u = q.front(); q.pop();
            if (u == t) break;
            for (int v = s; v <= t; v++){
                if (prev[v] == -1 and g[u][v] - flow[u][v] > 0){
                    q.push(v);
                    prev[v] = u;
                }
            }
        }
        if (prev[t] == -1) break;

        // Find bottleneck
        int curr = t;
        int bottleneck = 1 << 30;
        while (curr != s){
            bottleneck = min(bottleneck, g[prev[curr]][curr] - flow[prev[curr]][curr]);

```

```
        curr = prev[curr];
    }

    // Pump
    curr = t;
    while (curr != s){
        flow[prev[curr]][curr] += bottleneck;
        flow[curr][prev[curr]] -= bottleneck;
        curr = prev[curr];
    }

    // Add flow to answer
    max_flow += bottleneck;
}

return max_flow;
}
```

```
int main(){
    int m, n, sec, vel;
    while (cin >> n >> m >> sec >> vel){
        int s = 0;
        int t = n + m + 1;

        for (int i = 0; i <= t; i++)
            for (int j = 0; j <= t; j++)
                g[i][j] = 0;

        for (int i = 0; i < n; i++){
            double x, y;
            cin >> x >> y;
            gophers[i] = make_pair(x, y);
            g[s][i + 1] = 1;
        }
        for (int i = 0; i < m; i++){
            double x, y;
            cin >> x >> y;
            holes[i] = make_pair(x, y);
            g[i + 1 + n][t] = 1;
        }
        double max_dist = sec * vel;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < m; j++)
                if (canReach(i, j, max_dist)) {
                    g[i + 1][n + 1 + j] = 1;
                }
        cout << n - max_flow(s, t) << endl;
    }
}
```

```

// Encontrar puentes.
#include ...
using namespace std;

const int MAXN = 10005;

vector<int> g[MAXN];
int p[MAXN], d[MAXN], low[MAXN], tick;

int find(int x) {
    return p[x] == x ? x : p[x] = find(p[x]);
}

int link(int x, int y) {
    int a = find(x), b = find(y);
    if (a != b) {
        p[a] = b;
    }
}

// It's assumed that there is at most one edge
// between two nodes.
void dfs(int u, int parent = -1) {
    d[u] = low[u] = tick++;
    foreach(out, g[u]) {
        int v = *out;
        if (v == parent) continue;
        if (d[v] == -1) {
            dfs(v, u);
            low[u] = min(low[u], low[v]);
        } else {
            low[u] = min(low[u], d[v]);
        }

        if (low[v] > d[u]) {
            //printf("edge from %d to %d is a bridge\n", u + 1, v + 1);
            link(u, v);
        }
    }
}

int main(){
    int n, e, q;
    while (scanf("%d %d %d", &n, &e, &q) == 3) {
        if (n == 0 and e == 0 and q == 0) break;
        for (int i = 0; i < n; ++i) {
            g[i].clear();
            p[i] = i;
            d[i] = -1;
        }
        // read edges
        for (int i = 0; i < e; ++i) {
            int u, v; scanf("%d %d", &u, &v);
            u--, v--;
            g[u].push_back(v);
            g[v].push_back(u);
        }

        tick = 0;
        for (int i = 0; i < n; ++i) {

```

```
        if (d[i] == -1) dfs(i);
    }

    // read queries
    for (int i = 0; i < q; ++i) {
        int u, v; scanf("%d %d", &u, &v);
        u--, v--;

        if (find(u) == find(v)) {
            puts("Y");
        } else {
            puts("N");
        }
    }
    puts("-");
}
return 0;
}
```

```

// another fine solution by misof
#include <algorithm>
#include <numeric>

#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <queue>
#include <set>
#include <map>
#include <stack>

#include <cstdio>
#include <cstdlib>
#include <cctype>
#include <cassert>

#include <cmath>
#include <complex>
using namespace std;

#define SIZE(t) ((int)((t).size()))

// interval tree class
class interval_tree_vertex {
public:
    int color; // 0 empty, 1-7 rainbow
    int summary[8];
    void set_color(int c) { color=c; for (int i=0; i<8; ++i) summary[i]=0; }
    interval_tree_vertex() { set_color(0); }
};

class interval_tree {
    int L;
    void insert(int lo, int hi, int color, int kde, int left, int length);
    int count(int lo, int hi, int color, int kde, int left, int length);
public:
    vector<interval_tree_vertex> data;
    interval_tree(int N);
    void insert(int lo, int hi, int color);
    int count(int lo, int hi, int color);
};

interval_tree::interval_tree(int N) {
    for (int i=1; ; i*=2) if (i>N+2) { L=i; break; }
    data.resize(2*L);
}

void interval_tree::insert(int lo, int hi, int color) { insert(lo,hi,color,1,0,L); }
int interval_tree::count(int lo, int hi, int color) { return count(lo,hi,color,1,0,L); }

void interval_tree::insert(int lo, int hi, int color, int kde, int left, int length) {
    if (hi <= left || lo >= left+length) return; // mimo
    if (lo <= left && left+length <= hi) { // cely dnu
        data[kde].set_color(color);
        data[kde].summary[color] = length;
        return;
    }
    if (data[kde].color != 0) {

```



```

    int cc = data[kde].color;
    data[2*kde].set_color(cc);          data[2*kde+1].set_color(cc);
    data[2*kde].summary[cc] = length/2;  data[2*kde+1].summary[cc] = length/2;
}
data[kde].set_color(0);
insert(lo,hi,color,2*kde,left,length/2);
insert(lo,hi,color,2*kde+1,left+length/2,length/2);
for (int i=0; i<8; ++i) data[kde].summary[i] += data[2*kde].summary[i];
for (int i=0; i<8; ++i) data[kde].summary[i] += data[2*kde+1].summary[i];
}

int interval_tree::count(int lo, int hi, int color, int kde, int left, int length) {
    if (hi <= left || lo >= left+length) return 0; // mimo
    if (lo <= left && left+length <= hi) return data[kde].summary[color]; // cely dnu
    if (data[kde].color != 0) {
        int cc = data[kde].color;
        data[2*kde].set_color(cc);          data[2*kde+1].set_color(cc);
        data[2*kde].summary[cc] = length/2;  data[2*kde+1].summary[cc] = length/2;
    }
    return count(lo,hi,color,2*kde,left,length/2) + count(lo,hi,color,2*kde+1,left+length/2,length/2);
}

// the original tree
int N;
vector<vector<int> > G;

// rooted tree as parent/child edges
vector<vector<int> > children;
vector<int> parent;

// vertex processing times for the DFS
vector<int> time_in, time_out;

// heavy-light decomposition of the tree into paths
vector< vector<int> > paths;
vector<int> path_id, path_offset;

// an interval tree for each path
vector<interval_tree> trees;

void load() {
    cin >> N;
    G.clear(); G.resize(N);
    for (int i=0; i<N-1; ++i) {
        int x,y;
        cin >> x >> y;
        G[x].push_back(y);
        G[y].push_back(x);
    }
}

void dfs() {
    parent.clear(); parent.resize(N);
    children.clear(); children.resize(N);
    time_in.clear(); time_in.resize(N);
    time_out.clear(); time_out.resize(N);
    paths.clear();
    vector<bool> visited(N,false);
    vector<int> walk;
    vector<int> subtree_size(N,0);

```

```

int time = 0;

// run the DFS to compute lots of information
stack<int> vertex, edge;
visited[0]=true; time_in[0]=time; parent[0]=0;
vertex.push(0); edge.push(0);
while (!vertex.empty()) {
    ++time;
    int kde = vertex.top(); vertex.pop();
    int e = edge.top(); edge.pop();
    if (e == SIZE(G[kde])) {
        walk.push_back(kde);
        time_out[kde] = time;
        subtree_size[kde] = 1;
        for (int i=0; i<SIZE(children[kde]); ++i) subtree_size[kde] += subtree_size[children[kde][i]];
    } else {
        vertex.push(kde); edge.push(e+1);
        int kam = G[kde][e];
        if (!visited[kam]) {
            visited[kam]=true; time_in[kam]=time; parent[kam]=kde; children[kde].push_back(kam);
            vertex.push(kam); edge.push(0);
        }
    }
}

// compute the heavy-light decomposition
vector<bool> parent_edge_processed(N,false);
parent_edge_processed[0] = true;
for (int i=0; i<SIZE(walk); ++i) {
    int w = walk[i];
    if (parent_edge_processed[w]) continue;
    vector<int> this_path;
    this_path.push_back(w);
    while (1) {
        bool is_parent_edge_heavy = (2*subtree_size[w] >= subtree_size[parent[w]]);
        parent_edge_processed[w] = true;
        w = parent[w];
        this_path.push_back(w);
        if (!is_parent_edge_heavy) break;
        if (parent_edge_processed[w]) break;
    }
    paths.push_back(this_path);
}

path_id.clear(); path_id.resize(N); path_id[0]=-1;
path_offset.clear(); path_offset.resize(N);

for (int i=0; i<SIZE(paths); ++i)
    for (int j=0; j<SIZE(paths[i])-1; ++j) {
        path_id[ paths[i][j] ] = i;
        path_offset[ paths[i][j] ] = j;
    }

trees.clear();
for (int i=0; i<SIZE(paths); ++i) trees.push_back( interval_tree( SIZE(paths[i])-1 ) );
}

// return whether x is an ancestor of y
inline bool is_ancestor(int x, int y) {
    return (time_in[y] >= time_in[x] && time_out[y] <= time_out[x]);
}

```

```

}

// return the number of edges on the x-y path that do NOT have color c
// afterwards, color all edges on the x-y path using the color c
int query(int x, int y, int c) {
    if (x==y) return 0;
    if (is_ancestor(x,y)) return query(y,x,c);
    int p = path_id[x];
    int lo = path_offset[x], hi = SIZE(paths[p])-1;
    if (is_ancestor(paths[p][hi], y)) {
        while (hi-lo > 1) {
            int med = (hi+lo)/2;
            if (is_ancestor(paths[p][med], y)) hi=med; else lo=med;
        }
        lo = path_offset[x]; // keep hi at found value, restore lo
    }
    int result = hi-lo - trees[p].count(lo,hi,c);
    trees[p].insert(lo,hi,c);
    return result + query(paths[p][hi],y,c);
}

string color[7] = {"red","orange","yellow","green","blue","indigo","violet"};
map<string,int> C;

int main() {
    for (int i=0; i<7; ++i) C[color[i]]=i+1;
    int TC; cin >> TC;
    while (TC--) {
        load();
        dfs();
        int Q; cin >> Q;
        vector<long long> totals(8,0);
        while (Q--) {
            int x, y; string c; cin >> x >> y >> c;
            totals[C[c]] += query(x,y,C[c]);
        }
        for (int i=1; i<8; ++i) cout << color[i-1] << " " << totals[i] << endl;
    }
    return 0;
}

// vim: fdm=marker:commentstring=\ \"\ %s:nowrap:autoread

```

```
// Dijkstra con potenciales - 0.04s
```

```
int cost[50][50], cap[50][50], flow[50][50], s, t, V, inf = 1 << 29, par[50], dist[50], p[50];
```

```
inline int ecap(int a, int b){
    if(flow[b][a]) return flow[b][a];
    else return cap[a][b] - flow[a][b];
}
```

```
inline int ecost(int a, int b){
    if(flow[b][a]) return -cost[b][a] + p[a] - p[b];
    else return cost[a][b] + p[a] - p[b];
}
```

```
bool seen[50];
```

```
bool augment(){
```

```
    for(int i = 0; i < V; i++) par[i] = -1, dist[i] = inf;
    dist[s] = 0; par[s] = -2;
```

```
    memset(seen, 0, sizeof seen);
```

```
    int u = s;
```

```
    while(u != -1){
```

```
        seen[u] = true;
```

```
        for(int v = 0; v < V; v++){
```

```
            if(ecap(u, v) && dist[v] > ecost(u, v) + dist[u])
```

```
                dist[v] = dist[u] + ecost(u, v), par[v] = u;
```

```
        u = -1;
```

```
        for(int i = 0; i < V; i++) if(!seen[i] && dist[i] != inf && (u == -1 || dist[u] > dist[i])) u = i;
```

```
    }
```

```
    for(int v = 0; v < V; v++) if(dist[v] != inf) p[v] += dist[v];
```

```
    return dist[t] != inf;
```

```
}
```

```
int mcmf(){
```

```
    int res = 0;
```

```
    memset(p, 0, sizeof p);
```

```
    while(augment()){
```

```
        for(int v = t, u = par[v]; u != -2; u = par[v = u])
```

```
            if(flow[v][u]) flow[v][u]--, res -= cost[v][u];
```

```
            else flow[u][v]++, res += cost[u][v];
```

```
    }
```

```
    return res;
```

```
}
```

```
int main(){
```

```
    int y[16][2];
```

```
    int m;
```

```
    while(scanf("%d", &m) && m){
```

```
        for(int i = 0; i < m; i++) scanf("%d", &y[i][0]);
```

```
        for(int i = 0; i < m; i++) scanf("%d", &y[i][1]);
```

```
        s = m * 2; t = s + 1; V = t + 1;
```

```
        memset(cap, 0, sizeof cap);
```

```
        memset(cost, 0, sizeof cost);
```

```
        memset(flow, 0, sizeof flow);
```

```
        for(int i = 0; i < m; i++) for(int j = 0; j < m; j++)
```

```
            cap[i][j + m] = 1, cost[i][j + m] = abs(i - j) + abs(y[i][0] - y[j][1]);
```

```
        for(int i = 0; i < m; i++) cap[s][i] = cap[i + m][t] = 1;
```

```
        cout << mcmf() << endl;
```

```
    }
```

```
}
```

```

// Dijkstra sin potenciales - 0.03s
int cost[50][50], cap[50][50], flow[50][50], s, t, V, inf = 1 << 29, par[50], dist[50];

inline int ecap(int a, int b){
    if(flow[b][a]) return flow[b][a];
    else return cap[a][b] - flow[a][b];
}

inline int ecost(int a, int b){
    if(flow[b][a]) return -cost[b][a];
    else return cost[a][b];
}

bool seen[50];
bool augment(){
    for(int i = 0; i < V; i++) par[i] = -1, dist[i] = inf;
    dist[s] = 0; par[s] = -2;
    memset(seen, 0, sizeof seen);
    int u = s;
    while(u != -1){
        seen[u] = true;
        for(int v = 0; v < V; v++){
            if(ecap(u, v) && dist[v] > dist[u] + ecost(u, v)){
                dist[v] = dist[u] + ecost(u, v);
                par[v] = u;
                seen[v] = false;
            }
        }
        u = -1;
        for(int i = 0; i < V; i++) if(!seen[i] && dist[i] != inf && (u == -1 || dist[u] > dist[i])) u = i;
    }
    return dist[t] != inf;
}

int mcmf(){
    int res = 0;
    while(augment()){
        for(int v = t, u = par[v]; u != -2; u = par[v = u])
            if(flow[v][u]) flow[v][u]--, res -= cost[v][u];
            else flow[u][v]++, res += cost[u][v];
    }
    return res;
}

int main(){
    int y[16][2];
    int m;
    while(scanf("%d", &m) && m){
        for(int i = 0; i < m; i++) scanf("%d", &y[i][0]);
        for(int i = 0; i < m; i++) scanf("%d", &y[i][1]);
        s = m * 2; t = s + 1; V = t + 1;
        memset(cap, 0, sizeof cap);
        memset(cost, 0, sizeof cost);
        memset(flow, 0, sizeof flow);
        for(int i = 0; i < m; i++) for(int j = 0; j < m; j++)
            cap[i][j + m] = 1, cost[i][j + m] = abs(i - j) + abs(y[i][0] - y[j][1]);
        for(int i = 0; i < m; i++) cap[s][i] = cap[i + m][t] = 1;
        cout << mcmf() << endl;
    }
}

```

```

// Bellman-Ford - 0.04s
int cost[50][50], cap[50][50], flow[50][50], s, t, V, inf = 1 << 29, par[50], dist[50];

inline int ecap(int a, int b){
    if(flow[b][a]) return flow[b][a];
    else return cap[a][b] - flow[a][b];
}

inline int ecost(int a, int b){
    if(flow[b][a]) return -cost[b][a];
    else return cost[a][b];
}

bool augment(){
    for(int i = 0; i < V; i++) par[i] = -1, dist[i] = inf;
    dist[s] = 0; par[s] = -2;
    bool changed = true;
    while(changed){
        changed = false;
        for(int u = 0; u < V; u++) if(dist[u] != inf) for(int v = 0; v < V; v++){
            if(ecap(u, v) && dist[v] > dist[u] + ecost(u, v)){
                dist[v] = dist[u] + ecost(u, v);
                par[v] = u;
                changed = true;
            }
        }
    }
    return dist[t] != inf;
}

int mcmf(){
    int res = 0;
    while(augment()){
        for(int v = t, u = par[v]; u != -2; u = par[v = u])
            if(flow[v][u]) flow[v][u]--, res -= cost[v][u];
            else flow[u][v]++, res += cost[u][v];
    }
    return res;
}

int main(){
    int y[16][2];
    int m;
    while(scanf("%d", &m) && m){
        for(int i = 0; i < m; i++) scanf("%d", &y[i][0]);
        for(int i = 0; i < m; i++) scanf("%d", &y[i][1]);
        s = m * 2; t = s + 1; V = t + 1;
        memset(cap, 0, sizeof cap);
        memset(cost, 0, sizeof cost);
        memset(flow, 0, sizeof flow);
        for(int i = 0; i < m; i++) for(int j = 0; j < m; j++)
            cap[i][j + m] = 1, cost[i][j + m] = abs(i - j) + abs(y[i][0] - y[j][1]);
        for(int i = 0; i < m; i++) cap[s][i] = cap[i + m][t] = 1;
        cout << mcmf() << endl;
    }
}

```

```

// Star War de Filipe Martins
#include <bits/stdc++.h>
#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;

#define fr(a,b,c) for( int a = b ; a < c ; ++a )
#define rep(a,b) fr(a,0,b)
#define db(x) cout << #x " == " << x << endl
#define dbg db
#define _ << ", " <<

#define EPS 1e-7
int comp(double x, double y) {
    if( fabs(x-y) < EPS ) return 0;
    return x < y ? -1 : 1;
}

struct P{
    double x,y,z;
    P() {}
    P(double x, double y, double z): x(x), y(y), z(z) {}

    P operator+(P b) { return P(x+b.x, y+b.y, z+b.z); }
    P operator-(P b) { return P(x-b.x, y-b.y, z-b.z); }
    P operator-(P b) { return *this+-b; }
    double operator*(P b){ return x*b.x + y*b.y + z*b.z; }
    P operator*(double k){ return P(x*k, y*k, z*k); }
    P operator%(P b){ return P(y*b.z - z*b.y, z*b.x - x*b.z, x*b.y - y*b.x); } // cross product
    P operator/(P b){ return b*(*this*b/(b*b)); } // projection of this onto b
    double operator!() { return sqrt(*this**this); } // length
} p[4], q[4];

// Distance from point c to segment [a, b]
double distSP(P a, P b, P c) {
    P pp = a + (c-a)/(b-a);
    if( !comp(!(a-pp) + !(pp-b), !(a-b)) ) return !( c-pp );
    return min(!(a-c), !(b-c));
}

// Distance from segment [a, b] to segment [c, d]
double distSS(P a, P b, P c, P d) {
    P ba = b-a;
    P cd = c-d;
    P ca = c-a;
    P w = ba%cd;
    double dd = w*w;
    if( !comp(dd,0) ) { // both segments are parallel
        return min(min(distSP(a,b,c), distSP(a,b,d)), min(distSP(c,d,a), distSP(c,d,b)));
    }
    double x = ((ca%cd)*w)/dd;
    double y = ((ba%ca)*w)/dd;
    double z = ((ba%cd)*ca)/dd;
    if( x >= 0 && x <= 1 && y >= 0 && y <= 1 ) return !(w*z);
    return min(min(distSP(a,b,c), distSP(a,b,d)), min(distSP(c,d,a), distSP(c,d,b)));
}

// Distance from point d to triangle [a, b, c]
double distPP(P a, P b, P c, P d) {

```

```
P ba = b-a;
P ca = c-a;
P da = d-a;
P w = ba%ca;
P q = d-da/w;
double x = (b-a)%(q-a) * w, y = (c-b)%(q-b) * w, z = (a-c)%(q-c) * w;
if( x <= 0 && y <= 0 && z <= 0 || x >= 0 && y >= 0 && z >= 0 ) return !(da/w);
return min( min(distSP(a,b,d), distSP(b,c,d)), distSP(c,a,d));
}

int read() {
    fr(i,0,4) scanf("%lf%lf%lf", &p[i].x, &p[i].y, &p[i].z);
    fr(i,0,4) scanf("%lf%lf%lf", &q[i].x, &q[i].y, &q[i].z);

    double dist = 1./0. ;
    fr(i,0,4) fr(j,i+1,4) fr(k,j+1,4) fr(l,0,4) {
        double d = distPP(p[i], p[j], p[k], q[l]);
        if( d < dist ) dist = d;
        d = distPP(q[i], q[j], q[k], p[l]);
        if( d < dist ) dist = d;
    }
    fr(i,0,4) fr(j,i+1,4) fr(k,0,4) fr(l,k+1,4) {
        double d = distSS(p[i], p[j], q[k], q[l]);
        if( d < dist ) dist = d;
    }

    printf("%.2lf\n", dist);

    return 1;
}

void process() {
}

int main() {
    int t = 1;
    scanf("%d", &t);
    while( t-- && read() ) process();
    return 0;
}
```


ACM ICPC TEAM REFERENCE 2010 WORLD FINALS

Team Anuncie Aqui
Universidade Federal de Sergipe

1. CONFIGURATION FILES AND SCRIPTS

1.1. **.emacs**. Hash: b1040cede72bb06f9b3197eba2d833f5

```
(global-font-lock-mode t)
(setq transient-mark-mode t)

(global-set-key [f5] 'cxx-compile)
(defun cxx-compile()
  (interactive)
  (save-buffer)
```

```
(compile (concat "g++-g_-O2-o_" (file-name-sans-extension buffer-file-name)
  "_" buffer-file-name))
)

(add-hook 'c++-mode-hook (lambda () (c-set-style "stroustrup")
  (flymake-mode t)))
```

1.2. **Makefile**. Hash: 7381d22266f4ef5a9a601b80a76a956c

```
check-syntax:
```

```
g++ -Wall -fsyntax-only $(CHK_SOURCES)
```

1.3. **.vimrc**. Hash: da63747b3e94a58450094526d21a9e41

```
syn on
set nocp number ai si ts=4 sts=4 sw=4
```

```
ab #i #include
```

1.4. **Hash generator**. Hash: 0d22aecd779fc370b30a2c628aff517c

```
#!/bin/sh
```

```
sed ':a;N;$!ba;s/[_\n\t]//g' | md5sum | cut -d'_' -f1
```

1.5. Solution template. Hash: 220ea9d23d25447636bd67aeaf899fee

```
#include <algorithm>
#include <cassert>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <ctime>
#include <iostream>
#include <map>
#include <queue>
```

```
#include <set>
#include <sstream>
#include <string>
#include <utility>
#include <vector>

using namespace std;

int main() {
}
```

2. GRAPH ALGORITHMS

2.1. Tarjan's SCC algorithm. Hash: 3d598e293de54a302a6492c57304d745

```
int lowest[MAXV], num[MAXV], visited[MAXV], comp[MAXV];
int prev_edge[MAXE], last_edge[MAXV], adj[MAXE], nedges;
int cur_num, cur_comp;
stack<int> visiting;

void t_init() {
    memset(last_edge, -1, sizeof last_edge);
    nedges = 0;
}

void t_edge(int v, int w) {
    prev_edge[nedges] = last_edge[v];
    adj[nedges] = w;
    last_edge[v] = nedges++;
}

int tarjan_dfs(int v) {
    lowest[v] = num[v] = cur_num++;
    visiting.push(v);

    visited[v] = 1;
    for(int i = last_edge[v]; i != -1; i = prev_edge[i]) {
        int w = adj[i];
        if(visited[w] == 0) lowest[v] = min(lowest[v], tarjan_dfs(w));
        else if(visited[w] == 1) lowest[v] = min(lowest[v], num[w]);
    }
}
```

```
    }

    if(lowest[v] == num[v]) {
        int last = -1;
        while(last != v) {
            comp[last = visiting.top()] = cur_comp;
            visited[last] = 2;
            visiting.pop();
        }
        ++cur_comp;
    }

    return lowest[v];
}

void tarjan_scc(int num_v = MAXV) {
    visiting = stack<int>();
    memset(visited, 0, sizeof visited);
    cur_num = cur_comp = 0;

    for(int i = 0; i < num_v; ++i)
        if(!visited[i])
            tarjan_dfs(i);
}
```

2.2. Dinic's maximum flow algorithm. Hash: 3a41582d750893ec5cd598e8c9c0e2b2

```

int last_edge[MAXV], cur_edge[MAXV], dist[MAXV];
int prev_edge[MAXE], cap[MAXE], flow[MAXE], adj[MAXE];
int nedges;

void d_init() {
    nedges = 0;
    memset(last_edge, -1, sizeof last_edge);
}

void d_edge(int v, int w, int capacity, bool r = false) {
    prev_edge[nedges] = last_edge[v];
    cap[nedges] = capacity;
    adj[nedges] = w;
    flow[nedges] = 0;
    last_edge[v] = nedges++;
}

if(!r) d_edge(w, v, 0, true);
}

bool d_auxflow(int source, int sink) {
    queue<int> q;
    q.push(source);

    memset(dist, -1, sizeof dist);
    dist[source] = 0;
    memcpy(cur_edge, last_edge, sizeof last_edge);

    while(!q.empty()) {
        int v = q.front(); q.pop();
        for(int i = last_edge[v]; i != -1; i = prev_edge[i]) {
            if(cap[i] - flow[i] == 0) continue;

            if(dist[adj[i]] == -1) {
                dist[adj[i]] = dist[v] + 1;
                q.push(adj[i]);
            }
        }
    }
}

```

```

        if(adj[i] == sink) return true;
    }
}

return false;
}

int d_augmenting(int v, int sink, int c) {
    if(v == sink) return c;

    for(int& i = cur_edge[v]; i != -1; i = prev_edge[i]) {
        if(cap[i] - flow[i] == 0 || dist[adj[i]] != dist[v] + 1)
            continue;

        int val;
        if(val = d_augmenting(adj[i], sink, min(c, cap[i] - flow[i]))) {
            flow[i] += val;
            flow[i^1] -= val;
            return val;
        }
    }

    return 0;
}

int dinic(int source, int sink) {
    int ret = 0;
    while(d_auxflow(source, sink)) {
        int flow;
        while(flow = d_augmenting(source, sink, 0x3f3f3f3f))
            ret += flow;
    }

    return ret;
}

```

2.3. Successive shortest paths mincost maxflow algorithm. Hash: 1899233cb68a8d5f6e280654146e1747

```

int dist[MAXV], last_edge[MAXV], d_visited[MAXV], bg_prev[MAXV], pot[MAXV],
    capres[MAXV];
int prev_edge[MAXE], adj[MAXE], cap[MAXE], cost[MAXE], flow[MAXE];

```

```

int nedges;
priority_queue<pair<int, int> > d_q;

```

```

inline void bg_edge(int v, int w, int capacity, int cst, bool r = false) {
    prev_edge[nedges] = last_edge[v];
    adj[nedges] = w;
    cap[nedges] = capacity;
    flow[nedges] = 0;
    cost[nedges] = cst;
    last_edge[v] = nedges++;

    if(!r) bg_edge(w, v, 0, -cst, true);
}

inline int rev(int i) { return i ^ 1; }
inline int from(int i) { return adj[rev(i)]; }

inline void bg_init() {
    nedges = 0;
    memset(last_edge, -1, sizeof last_edge);
    memset(pot, 0, sizeof pot);
}

void bg_dijkstra(int s, int num_nodes = MAXV) {
    memset(dist, 0x3f, sizeof dist);
    memset(d_visited, 0, sizeof d_visited);
    d_q.push(make_pair(dist[s] = 0, s));
    capres[s] = 0x3f3f3f3f;

    while(!d_q.empty()) {
        int v = d_q.top().second; d_q.pop();
        if(d_visited[v]) continue; d_visited[v] = true;

        for(int i = last_edge[v]; i != -1; i = prev_edge[i]) {
            if(cap[i] - flow[i] == 0) continue;

```

```

            int w = adj[i], new_dist = dist[v] + cost[i] + pot[v] - pot[w];

            if(new_dist < dist[w]) {
                d_q.push(make_pair(-(dist[w] = new_dist), w));
                bg_prev[w] = rev(i);
                capres[w] = min(capres[v], cap[i] - flow[i]);
            }
        }
    }

pair<int, int> busacker_gowen(int src, int sink, int num_nodes = MAXV) {
    int ret_flow = 0, ret_cost = 0;

    bg_dijkstra(src, num_nodes);
    while(dist[sink] < 0x3f3f3f3f) {
        int cur = sink;
        while(cur != src) {
            flow[bg_prev[cur]] -= capres[sink];
            flow[rev(bg_prev[cur])] += capres[sink];
            ret_cost += cost[rev(bg_prev[cur])] * capres[sink];
            cur = adj[bg_prev[cur]];
        }
        ret_flow += capres[sink];

        for(int i = 0; i < MAXV; ++i)
            pot[i] = min(pot[i] + dist[i], 0x3f3f3f3f);

        bg_dijkstra(src, num_nodes);
    }
    return make_pair(ret_flow, ret_cost);
}

```

2.4. Kuhn-Munkres' weighted bipartite matching algorithm. Hash: a6cca19c70194378fb24a3d89b82eb53

```

int w[MAXV][MAXV], s[MAXV], rem[MAXV], remx[MAXV];
int mx[MAXV], my[MAXV], lx[MAXV], ly[MAXV];

void add(int x, int n) {
    s[x] = true;
    for(int y = 0; y < n; y++)
        if(rem[y] != -INF && rem[y] > lx[x] + ly[y] - w[x][y])
            rem[y] = lx[x] + ly[y] - w[x][y], remx[y] = x;
}

```

```

int kuhn_munkres(int n) {
    for(int i = 0; i < n; i++) mx[i] = my[i] = -1, lx[i] = ly[i] = 0;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            ly[j] = max(ly[j], w[i][j]);

    for(int i = 0; i < n; i++) {
        memset(s, 0, sizeof s); memset(rem, 0x3f, sizeof rem);

```

```

int st;
for(st = 0; st < n; st++) if(mx[st] == -1) { add(st, n); break; }
while(mx[st] == -1) {
    int miny = -1;
    for(int y = 0; y < n; y++)
        if(rem[y] != -INF && (miny == -1 || rem[miny] >= rem[y]))
            miny = y;

    if(rem[miny]) {
        for(int x = 0; x < n; x++) if(s[x] lx[x] == rem[miny]);
        for(int y = 0, d = rem[miny]; y < n; y++)
            if(rem[y] == -INF) ly[y] += d; else rem[y] -= d;
    }

    if(my[miny] == -1) {
        int cur = miny;

```

```

        while(remx[cur] != st) {
            int pmate = mx[remx[cur]];
            my[cur] = remx[cur], mx[my[cur]] = cur;
            my[pmate] = -1; cur = pmate;
        }
        my[cur] = remx[cur], mx[my[cur]] = cur;
    } else
        add(my[miny], n), rem[miny] = -INF;
}

int ret = 0;
for(int i = 0; i < n; i++)
    ret += w[i][mx[i]];
return ret;
}

```

2.5. Gomory-Hu tree. Hash: 61170139df2a1bd22618717b1d57ed1a

```

bool cut[MAXV];
int mincut(int s, int t) {
    memset(flow, 0, sizeof flow);
    memset(cut, 0, sizeof cut);
    int ret = dinic(s, t);

    queue<int> q;
    q.push(s); cut[s] = true;
    while(!q.empty()) {
        int v = q.front(); q.pop();
        for(int i = last_edge[v]; i != -1; i = prev_edge[i]) {
            int w = adj[i];
            if(cap[i] - flow[i] && !cut[w])
                cut[w] = true, q.push(w);
        }
    }
}

```

```

    }

    return ret;
}

int up[MAXV], val[MAXV];
void gomory_hu(int n) {
    memset(up, 0, sizeof up);
    for(int i = 1; i < n; i++) {
        val[i] = mincut(i, up[i]);
        for(int j = i+1; j < n; j++)
            if(cut[j] && up[j] == up[i])
                up[j] = i;
    }
}

```

2.6. Gabow's general matching algorithm. Hash: 85de6860f6b8472baad4c5b063815b18

```

int prev_edge[MAXE], v[MAXE], w[MAXE], last_edge[MAXV];
int type[MAXV], label[MAXV], first[MAXV], mate[MAXV], nedges;
bool g_flag[MAXV], g_souter[MAXV];

void g_init() {

```

```

    nedges = 0;
    memset(last_edge, -1, sizeof last_edge);
}

void g_edge(int a, int b, bool rev = false) {

```

```

    prev_edge[nedges] = last_edge[a];
    v[nedges] = a;
    w[nedges] = b;
    last_edge[a] = nedges++;

    if(!rev) return g_edge(b, a, true);
}

void g_label(int v, int join, int edge, queue<int>& outer) {
    if(v == join) return;
    if(label[v] == -1) outer.push(v);

    label[v] = edge;
    type[v] = 1;
    first[v] = join;

    g_label(first[label[mate[v]]], join, edge, outer);
}

void g_augment(int _v, int _w) {
    int t = mate[_v];
    mate[_v] = _w;

    if(mate[t] != _v) return;
    if(label[_v] == -1) return;

    if(type[_v] == 0) {
        mate[t] = label[_v];
        g_augment(label[_v], t);
    } else if(type[_v] == 1) {
        g_augment(v[label[_v]], w[label[_v]]);
        g_augment(w[label[_v]], v[label[_v]]);
    }
}

int gabow(int n) {
    memset(mate, -1, sizeof mate);
    memset(first, -1, sizeof first);

    int ret = 0;
    for(int z = 0; z < n; ++z) {
        if(mate[z] != -1) continue;

        memset(label, -1, sizeof label);
        memset(type, -1, sizeof type);
        memset(g_souter, 0, sizeof g_souter);

```

```

        label[z] = -1; type[z] = 0;

        queue<int> outer;
        outer.push(z);

        bool done = false;
        while(!outer.empty()) {
            int x = outer.front(); outer.pop();

            if(g_souter[x]) continue;
            g_souter[x] = true;

            for(int i = last_edge[x]; i != -1; i = prev_edge[i]) {
                if(mate[w[i]] == -1 && w[i] != z) {
                    mate[w[i]] = x;
                    g_augment(x, w[i]);
                    ++ret;

                    done = true;
                    break;
                }

                if(type[w[i]] == -1) {
                    int v = mate[w[i]];
                    if(type[v] == -1) {
                        type[v] = 0;
                        label[v] = x;
                        outer.push(v);

                        first[v] = w[i];
                    }
                    continue;
                }
            }

            int r = first[x], s = first[w[i]];
            if(r == s) continue;

            memset(g_flag, 0, sizeof g_flag);
            g_flag[r] = g_flag[s] = true;

            while(r != -1 || s != -1) {
                if(s != -1) swap(r, s);
                r = first[label[mate[r]]];
                if(r == -1) continue;
                if(g_flag[r]) break; g_flag[r] = true;
            }

```

```

    }

    g_label(first[x], r, i, outer);
    g_label(first[w[i]], r, i, outer);

    for(int c = 0; c < n; ++c)
        if(type[c] != -1 && first[c] != -1 && type[first[c]] != -1)

```

```

        first[c] = r;
    }
    if(done) break;
}
}
return ret;
}

```

2.7. Heavy-light decomposition. Hash: 097fd70cd19bf3b258d7648f9273bcc1

```

int last_edge[MAXV], prev_edge[MAXE], adj[MAXE], nedges;
int up[MAXV], subtree[MAXV], path[MAXV], offset[MAXV], depth[MAXV];
int nump, numv, psize[MAXV], pfirst[MAXV], walk[MAXV];

struct stree {
    vector<int> data;
    int sz;

    explicit stree(int tsz) : sz(1) {
        while(sz < tsz) sz *= 2;
        data.resize(2*sz);
    }

    int query(int a, int b, int root, int l, int r) {
        if(l == a && r == b) return data[root];
        int mid = (l+r)/2, ans = 0;
        if(a <= mid) ans = max(ans, query(a, min(b, mid), 2*root+1, l, mid));
        if(b > mid) ans = max(ans, query(max(a, mid+1), b, 2*root+2, mid+1, r));
        return ans;
    }

    int query(int a, int b) { return a<=b ? query(a, b, 0, 0, sz-1) : 0; }

    void update(int pos, int val, int root, int l, int r) {
        if(l == r) { data[root] = val; return; }
        int mid = (l+r)/2;
        if(pos <= mid) update(pos, val, 2*root+1, l, mid);
        else update(pos, val, 2*root+2, mid+1, r);
        data[root] = max(data[2*root+1], data[2*root+2]);
    }

    void update(int pos, int val) { update(pos, val, 0, 0, sz-1); }
};

void hl_init(int n) {
    memset(last_edge, -1, sizeof(int) * n);
    nedges = 0;

```

```

}

void hl_edge(int a, int b, bool rev = false) {
    prev_edge[nedges] = last_edge[a];
    adj[nedges] = b;
    last_edge[a] = nedges++;

    if(!rev) hl_edge(b, a, true);
}

vector<stree> segtree;
void heavy_light() {
    memset(up, -1, sizeof up);
    stack<int> s; if(last_edge[0] != -1) s.push(last_edge[0]);

    walk[0] = depth[0] = up[0] = 0; numv = subtree[0] = 1;
    while(!s.empty()) {
        int i = s.top(), v = adj[i^1], w = adj[i];
        if(up[w] == -1) {
            up[w] = v; depth[w] = depth[v]+1; subtree[w] = 1;
            walk[numv++] = w; s.push(last_edge[w]);
        } else {
            s.pop(); if(up[w] == v) subtree[v] += subtree[w];
            if(prev_edge[i] != -1) s.push(prev_edge[i]);
        }
    }

    pfirst[0] = path[0] = offset[0] = 0; nump = psize[0] = 1;
    for(int i = 1; i < numv; i++) {
        int v = walk[i], p = up[v];
        if(2*subtree[v] < subtree[p] || p == 0)
            offset[v] = 0, path[v] = nump, pfirst[nump] = v, psize[nump++] = 1;
        else
            offset[v] = offset[p]+1, path[v] = path[p], psize[path[v]]++;
    }
}

```

```

    segtree.clear(); segtree.reserve(nump);
    for(int i = 0; i < nump; i++) segtree.push_back(stree(psize[i]));
}

int lca(int v, int w) {
    int fpv = pfirst[path[v]], fpw = pfirst[path[w]];

```

2.8. Link-cut tree. Hash: eddb323eb229e5c7bb67a8326d84e371

```

class splay {
public:
    splay *sons[2], *up, *path_up;
    splay() : up(NULL), path_up(NULL) {
        sons[0] = sons[1] = NULL;
    }

    bool is_r(splay* n) {
        return n == sons[1];
    }
};

void rotate(splay* t, bool to_l) {
    splay* n = t->sons[to_l]; swap(t->path_up, n->path_up);
    t->sons[to_l] = n->sons[!to_l]; if(t->sons[to_l]) t->sons[to_l]->up = t;
    n->up = t->up; if(n->up) n->up->sons[n->up->is_r(t)] = n;
    n->sons[!to_l] = t; t->up = n;
}

void do_splay(splay* n) {
    for(splay* p; (p = n->up) != NULL; )
        if(p->up == NULL)
            rotate(p, p->is_r(n));
        else {
            bool dirp = p->is_r(n), dirg = p->up->is_r(p);
            if(dirp == dirg)
                rotate(p->up, dirg), rotate(p, dirp);
            else
                rotate(p, dirp), rotate(n->up, dirg);
        }
}

struct link_cut {
    splay* vtxs;
    link_cut(int numv) { vtxs = new splay[numv]; }
    ~link_cut() { delete[] vtxs; }

```

```

    while(v != 0 && w != 0 && fpv != fpw) {
        if(depth[up[fpv]] > depth[up[fpw]]) swap(v, w), swap(fpv, fpw);
        w = up[fpw]; fpw = pfirst[path[w]];
    }
    return depth[v] < depth[w] ? v : w;
}

```

```

void access(splay* ov) {
    for(splay *w = ov, *v = ov; w != NULL; v = w, w = w->path_up) {
        do_splay(w);
        if(w->sons[1]) w->sons[1]->path_up = w, w->sons[1]->up = NULL;
        if(w != v) w->sons[1] = v, v->up = w, v->path_up = NULL;
        else w->sons[1] = NULL;
    }
    do_splay(ov);
}

```

```

splay* find(int v) {
    splay* s = &vtxs[v];
    access(s); while(s->sons[0]) s = s->sons[0]; do_splay(s);
    return s;
}

```

```

void link(int parent, int son) {
    access(&vtxs[son]); access(&vtxs[parent]);
    assert(vtxs[son].sons[0] == NULL);
    vtxs[son].sons[0] = &vtxs[parent];
    vtxs[parent].up = &vtxs[son];
}

```

```

void cut(int v) {
    access(&vtxs[v]);
    if(vtxs[v].sons[0]) vtxs[v].sons[0]->up = NULL;
    vtxs[v].sons[0] = NULL;
}

```

```

int lca(int v, int w) {
    access(&vtxs[v]); access(&vtxs[w]); do_splay(&vtxs[v]);
    if(vtxs[v].path_up == NULL) return v;
    return vtxs[v].path_up - vtxs;
}
};

```


3. MATH

3.1. Fractions. Hash: 85739ae11b5b0351c0a9b5b6f813eaf8

```

struct frac {
    long long num, den;
    frac(long long num = 0, long long den = 1) { set_val(num, den); }

    void set_val(long long _num, long long _den) {
        num = _num/__gcd(_num, _den);
        den = _den/__gcd(_num, _den);
        if(den < 0) { num *= -1; den *= -1; }
    }

    void operator+=(frac f) { set_val(num * f.num, den * f.den); }
    void operator+=(frac f) { set_val(num * f.den + f.num * den, den * f.den); }
    void operator+=(frac f) { set_val(num * f.den - f.num * den, den * f.den); }
    void operator+=(frac f) { set_val(num * f.den, den * f.num); }
};

```

```

bool operator==(frac a, frac b) { return a.num * b.den == b.num * a.den; }
bool operator!=(frac a, frac b) { return !(a == b); }
bool operator<(frac a, frac b) { return a.num * b.den < b.num * a.den; }
bool operator<=(frac a, frac b) { return (a == b) || (a < b); }
bool operator>(frac a, frac b) { return !(a <= b); }
bool operator>=(frac a, frac b) { return !(a < b); }
frac operator/(frac a, frac b) { frac ret = a; ret /= b; return ret; }
frac operator*(frac a, frac b) { frac ret = a; ret *= b; return ret; }
frac operator+(frac a, frac b) { frac ret = a; ret += b; return ret; }
frac operator-(frac a, frac b) { frac ret = a; ret -= b; return ret; }
frac operator-(frac f) { return 0 - f; }

std::ostream& operator<<(std::ostream& o, const frac f) {
    o << f.num << "/" << f.den;
    return o;
}

```

3.2. Chinese remainder theorem. Hash: 06b5ebd5c44c204a4b11bbb76d09023d

```

struct t {
    long long a, b; int g;
    t(long long a, long long b, int g) : a(a), b(b), g(g) { }
    t swap() { return t(b, a, g); }
};

t egcd(int p, int q) {
    if(q == 0) return t(1, 0, p);

    t t2 = egcd(q, p % q);

```

```

    t2.a -= t2.b * (p/q);
    return t2.swap();
}

int crt(int a, int p, int b, int q) {
    t t2 = egcd(p, q); t2.a %= p*q; t2.b %= p*q;
    assert(t2.g == 1);
    int ret = ((b * t2.a)%(p*q) * p + (a * t2.b)%(p*q) * q) % (p*q);
    return ret >= 0 ? ret : ret + p*q;
}

```

3.3. Longest increasing subsequence. Hash: 8578a256b2926d8be6ace63e1ed4088c

```

vector<int> lis(vector<int>& seq) {
    int smallest_end[seq.size()+1], prev[seq.size()];
    smallest_end[1] = 0;

    int sz = 1;
    for(int i = 1; i < seq.size(); ++i) {
        int lo = 0, hi = sz;

```

```

        while(lo < hi) {
            int mid = (lo + hi + 1)/2;
            if(seq[smallest_end[mid]] <= seq[i])
                lo = mid;
            else
                hi = mid - 1;
        }

```

```

prev[i] = smallest_end[lo];
if(lo == sz)
    smallest_end[++sz] = i;
else if(seq[i] < seq[smallest_end[lo+1]])
    smallest_end[lo+1] = i;
}

```

```

vector<int> ret;
for(int cur = smallest_end[sz]; sz > 0; cur = prev[cur], --sz)
    ret.push_back(seq[cur]);
reverse(ret.begin(), ret.end());

return ret;
}

```

3.4. Simplex (Warsaw University). Hash: c687094970cf1953fd6f87a01adc6a95

```

const double EPS = 1e-9;
typedef long double T;
typedef vector<T> VT;
vector<VT> A;
VT b,c,res;
VI kt,N;
int m;
inline void pivot(int k,int l,int e){
    int x=kt[l]; T p=A[l][e];
    REP(i,k) A[l][i]/=p; b[l]/=p; N[e]=0;
    REP(i,m) if (i!=l) b[i]-=A[i][e]*b[l],A[i][x]=A[i][e]*A[l][x];
    REP(j,k) if (N[j]){
        c[j]-=c[e]*A[l][j];
        REP(i,m) if (i!=l) A[i][j]-=A[i][e]*A[l][j];
    }
    kt[l]=e; N[x]=1; c[x]=c[e]*A[l][x];
}

VT doit(int k){
    VT res; T best;
    while (1){
        int e=-1,l=-1; REP(i,k) if (N[i] && c[i]>EPS) {e=i; break;}
        if (e==-1) break;
        REP(i,m) if (A[i][e]>EPS && (l==-1 || best>b[i]/A[i][e]))
            best=b[i]/A[i][e];
    }
}

```

```

    if (l==-1) /*ilimitado*/ return VT();
    pivot(k,l,e);
}
res.resize(k,0); REP(i,m) res[kt[i]]=b[i];
return res;
}

VT simplex(vector<VT> &AA,VT &bb,VT &cc){
    int n=AA[0].size(),k;
    m=AA.size(); k=n+m+1; kt.resize(m); b=bb; c=cc; c.resize(n+m);
    A=AA; REP(i,m){ A[i].resize(k); A[i][n+i]=1; A[i][k-1]=-1; kt[i]=n+i;}
    N=VI(k,1); REP(i,m) N[kt[i]]=0;
    int pos=min_element(ALL(b))-b.begin();
    if (b[pos]<=-EPS){
        c=VT(k,0); c[k-1]=-1; pivot(k,pos,k-1); res=doit(k);
        if (res[k-1]>EPS) /*impossivel*/ return VT();
        REP(i,m) if (kt[i]==k-1)
            REP(j,k-1) if (N[j] && (A[i][j]<=-EPS || EPS<A[i][j])){
                pivot(k,i,j); break;
            }
        c=cc; c.resize(k,0); REP(i,m) REP(j,k) if (N[j]) c[j]-=c[kt[i]]*A[i][j];
    }
    res=doit(k-1); if (!res.empty()) res.resize(n);
    return res;
}

```

3.5. Romberg's method. Hash: a50b581a5c45b3266a7b2d1e76d8e453

```

long double romberg(long double a, long double b,
    long double(*func)(long double)) {
    long double approx[2][25];
    long double *cur=approx[1], *prev=approx[0];
}

```

```

prev[0] = 1/2.0 * (b-a) * (func(a) + func(b));
for(int it = 1; it < 25; ++it, swap(cur, prev)) {
    if(it > 1 && cmp(prev[it-1], prev[it-2]) == 0)

```

```

    return prev[it-1];

    cur[0] = 1/2.0 * prev[0];
    long double div = (b-a)/pow(2, it);
    for(long double sample = a + div; sample < b; sample += 2 * div)
        cur[0] += div * func(a + sample);

```

```

    for(int j = 1; j <= it; ++j)
        cur[j] = cur[j-1] + 1/(pow(4, it) - 1)*(cur[j-1] + prev[j-1]);
    }

    return prev[24];
}

```

3.6. Floyd's cycle detection algorithm. Hash: 97a42d1ac6750f912c5a06e04636c1db

```

pair<int, int> floyd(int x0) {
    int t = f(x0), h = f(f(x0)), start = 0, length = 1;
    while(t != h)
        t = f(t), h = f(f(h));

    h = t; t = x0;
    while(t != h)
        t = f(t), h = f(h), ++start;
}

```

```

h = f(t);
while(t != h)
    h = f(h), ++length;

return make_pair(start, length);
}

```

3.7. Pollard's rho algorithm. Hash: ad4ee1d4afc564b2c55f90d6269994c4

```

long long pollard_r, pollard_n;

inline long long f(long long val) { return (val*val + pollard_r) % pollard_n; }
inline long long myabs(long long a) { return a >= 0 ? a : -a; }

long long pollard(long long n) {
    srand(unsigned(time(0)));
    pollard_n = n;

    long long d = 1;
    do {

```

```

        d = 1;
        pollard_r = rand() % n;

        long long x = 2, y = 2;
        while(d == 1)
            x = f(x), y = f(f(y)), d = __gcd(myabs(x-y), n);
    } while(d == n);

    return d;
}

```

3.8. Miller-Rabin's algorithm. Hash: 5288cd2ac5d62a97ea1175eec20d0010

```

int fastpow(int base, int d, int n) {
    int ret = 1;
    for(long long pow = base; d > 0; d >>= 1, pow = (pow * pow) % n)
        if(d & 1)
            ret = (ret * pow) % n;
    return ret;
}

```

```

bool miller_rabin(int n, int base) {
    if(n <= 1) return false;
    if(n % 2 == 0) return n == 2;

    int s = 0, d = n - 1;
    while(d % 2 == 0) d /= 2, ++s;
}

```

```

int base_d = fastpow(base, d, n);
if(base_d == 1) return true;
int base_2r = base_d;

for(int i = 0; i < s; ++i) {
    if(base_2r == 1) return false;
    if(base_2r == n - 1) return true;
    base_2r = (long long)base_2r * base_2r % n;
}

```

3.9. Cooley-Tukey's algorithm. Hash: 2f8e032ae3f77a94dbe8ac547ef8c2b2

```

typedef complex<long double> pt;
pt tmp[1<<20];

void fft(pt *in, int sz, bool inv = false) {
    if(sz == 1) return;
    for(int i = 0; i < sz; i++)
        tmp[i] = in[i];

    sz /= 2;
    pt *even = in, *odd = in + sz;
    for(int i = 0; i < 2*sz; i++)
        if(i&1) odd[i/2] = tmp[i];
        else even[i/2] = tmp[i];
}

```

```

}

return false;
}

bool isprime(int n) {
    if(n == 2 || n == 7 || n == 61) return true;
    return miller_rabin(n, 2) && miller_rabin(n, 7) && miller_rabin(n, 61);
}

```

```

fft(even, sz, inv);
fft(odd, sz, inv);

long double p = (inv ? 1 : -1) * acosl(-1)/sz;
for(int i = 0; i < sz; i++) {
    pt conv = pt(cosl(i*p), sinl(i*p)) * odd[i];
    tmp[i] = even[i] + conv;
    tmp[i+sz] = even[i] - conv;
}

for(int i = 0; i < 2*sz; i++)
    in[i] = tmp[i];
}

```

3.10. Karatsuba's algorithm. Hash: baa2224f03b35ae422eed1c261dcf6b8

```

typedef vector<int> poly;

poly mult(const poly& p, const poly& q) {
    int sz = p.size(), half = sz/2;
    assert(sz == q.size() && !(sz&(sz-1)));

    if(sz <= 64) {
        poly ret(2*sz);
        for(int i = 0; i < sz; i++)
            for(int j = 0; j < sz; j++)
                ret[i+j] += p[i] * q[j];
        return ret;
    }

    poly p1(p.begin(), p.begin() + half), p2(p.begin() + half, p.end());
}

```

```

poly q1(q.begin(), q.begin() + half), q2(q.begin() + half, q.end());
poly p1p2(half), q1q2(half);
for(int i = 0; i < half; i++)
    p1p2[i] = p1[i] + p2[i], q1q2[i] = q1[i] + q2[i];

poly low = mult(p1, q1), high = mult(p2, q2), mid = mult(p1p2, q1q2);
for(int i = 0; i < sz; i++)
    mid[i] -= high[i] + low[i];

low.resize(2*sz);
for(int i = 0; i < sz; i++)
    low[i+half] += mid[i], low[i+sz] += high[i];

return low;
}

```

3.11. Optimized sieve of Erathostenes. Hash: f61dff82d061bab316148816c3b7ac01

```
const unsigned MAX = 1000000020/60, MAX_S = sqrt(MAX/60);

unsigned w[16] = {1, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49, 53, 59};
unsigned short composite[MAX];
vector<int> primes;

void sieve() {
    unsigned mod[16][16], di[16][16], num;
    for(int i = 0; i < 16; i++)
        for(int j = 0; j < 16; j++) {
            di[i][j] = (w[i]*w[j])/60;
            mod[i][j] = lower_bound(w, w + 16, (w[i]*w[j])%60) - w;
        }

    primes.push_back(2); primes.push_back(3); primes.push_back(5);
```

```
memset(composite, 0, sizeof composite);
for(unsigned i = 0; i < MAX; i++)
    for(int j = (i==0); j < 16; j++) {
        if(composite[i] & (1<<j)) continue;
        primes.push_back(num = 60*i + w[j]);

        if(i > MAX_S) continue;
        for(unsigned k = i, done = false; !done; k++)
            for(int l = (k==0); l < 16 && !done; l++) {
                unsigned mult = k*num + i*w[l] + di[j][l];
                if(mult >= MAX) done = true;
                else composite[mult] |= 1<<mod[j][l];
            }
    }
}
```

3.12. Polynomials (PUC-Rio). Hash: d69d1ad494e487327d2338e69eccfa2f

```
typedef complex<double> cdouble;
int cmp(cdouble x, cdouble y = 0) {
    return cmp(abs(x), abs(y));
}
const int TAM = 200;
struct poly {
    cdouble poly[TAM]; int n;
    poly(int n = 0): n(n) { memset(p, 0, sizeof(p)); }
    cdouble& operator [] (int i) { return p[i]; }
    poly operator ~() {
        poly r(n-1);
        for (int i = 1; i <= n; i++)
            r[i-1] = p[i] * cdouble(i);
        return r;
    }
    pair<poly, cdouble> ruffini(cdouble z) {
        if (n == 0) return make_pair(poly(), 0);
        poly r(n-1);
        for (int i = n; i > 0; i--) r[i-1] = r[i] * z + p[i];
        return make_pair(r, r[0] * z + p[0]);
    }
    cdouble operator () (cdouble z) { return ruffini(z).second; }
    cdouble find_one_root(cdouble x) {
```

```
poly p0 = *this, p1 = ~p0, p2 = ~p1;
int m = 1000;
while (m--) {
    cdouble y0 = p0(x);
    if (cmp(y0) == 0) break;
    cdouble G = p1(x) / y0;
    cdouble H = G * G - p2(x) - y0;
    cdouble R = sqrt(cdouble(n-1) * (H * cdouble(n) - G * G));
    cdouble D1 = G + R, D2 = G - R;
    cdouble a = cdouble(n) / (cmp(D1, D2) > 0 ? D1 : D2);
    x -= a;
    if (cmp(a) == 0) break;
}
return x;
}
vector<cdouble> roots() {
    poly q = *this;
    vector<cdouble> r;
    while (q.n > 1) {
        cdouble z(rand() / double(RAND_MAX), rand() / double(RAND_MAX));
        z = q.find_one_root(z); z = find_one_root(z);
        q = q.ruffini(z).first;
        r.push_back(z);
    }
```

```

    }
    return r;

```

```

    }
};

```

4. GEOMETRY

4.1. Point class. Hash: 4a0fc00fd27520d94e04b2fc6c05ed73

```

typedef double TYPE;
const TYPE EPS = 1e-9, INF = 1e9;

inline int sgn(TYPE a) { return a > EPS ? 1 : (a < -EPS ? -1 : 0); }
inline int cmp(TYPE a, TYPE b) { return sgn(a - b); }

struct pt {
    TYPE x, y;
    pt(TYPE x = 0, TYPE y = 0) : x(x), y(y) {}

    bool operator==(pt p) { return cmp(x, p.x) == 0 && cmp(y, p.y) == 0; }
    bool operator<(pt p) const {
        return cmp(x, p.x) ? cmp(x, p.x) < 0 : cmp(y, p.y) < 0;
    }
    bool operator<=(pt p) { return *this < p || *this == p; }
    TYPE operator||(pt p) { return x*p.x + y*p.y; }
    TYPE operator%(pt p) { return x*p.y - y*p.x; }
    pt operator~() { return pt(x, -y); }
    pt operator+(pt p) { return pt(x + p.x, y + p.y); }
    pt operator-(pt p) { return pt(x - p.x, y - p.y); }
    pt operator*(pt p) { return pt(x*p.x - y*p.y, x*p.y + y*p.x); }
    pt operator/(TYPE t) { return pt(x/t, y/t); }

```

```

    pt operator/(pt p) { return (*this * ~p)/(p||p); }
};

const pt I = pt(0,1);

struct circle {
    pt c; TYPE r;
    circle(pt c, TYPE r) : c(c), r(r) {}
};

TYPE norm(pt a) { return a||a; }
TYPE abs(pt a) { return sqrt(a||a); }
TYPE dist(pt a, pt b) { return abs(a - b); }
TYPE area(pt a, pt b, pt c) { return (a-c)%(b-c); }
int ccw(pt a, pt b, pt c) { return sgn(area(a, b, c)); }
pt unit(pt a) { return a/abs(a); }
double arg(pt a) { return atan2(a.y, a.x); }
pt f_polar(TYPE mod, double ang) { return pt(mod * cos(ang), mod * sin(ang)); }
inline int g_mod(int i, int n) { if(i == n) return 0; return i; }

ostream& operator<<(ostream& o, pt p) {
    return o << "(" << p.x << "," << p.y << ")";
}

```

4.2. Intersection primitives. Hash: ab780978106a5c062b8f7a129ebc9196

```

bool in_rect(pt a, pt b, pt c) {
    return sgn(c.x - min(a.x, b.x)) >= 0 && sgn(max(a.x, b.x) - c.x) >= 0 &&
        sgn(c.y - min(a.y, b.y)) >= 0 && sgn(max(a.y, b.y) - c.y) >= 0;
}
bool ps_isects(pt a, pt b, pt c) { return ccw(a,b,c) == 0 && in_rect(a,b,c); }

bool ss_isects(pt a, pt b, pt c, pt d) {
    if (ccw(a,b,c)*ccw(a,b,d) == -1 && ccw(c,d,a)*ccw(c,d,b) == -1) return true;
    return ps_isects(a, b, c) || ps_isects(a, b, d) ||
        ps_isects(c, d, a) || ps_isects(c, d, b);
}

```

```

pt parametric_isect(pt p, pt v, pt q, pt w) {
    double t = ((q-p)%w)/(v%w);
    return p + v*t;
}

pt ss_isect(pt p, pt q, pt r, pt s) {
    pt isect = parametric_isect(p, q-p, r, s-r);
    if(ps_isects(p, q, isect) && ps_isects(r, s, isect)) return isect;
    return pt(1/0.0, 1/0.0);
}

```

4.3. Polygon primitives. Hash: 621a339a657d07de8f651d55e13d988b

```
double p_signedarea(vector<pt>& pol) {
    double ret = 0;
    for(int i = 0; i < pol.size(); ++i)
        ret += pol[i] % pol[g_mod(i+1, pol.size())];
    return ret/2;
}

int point_polygon(pt p, vector<pt>& pol) {
    int n = pol.size(), count = 0;
```

```
    for(int i = 0; i < n; ++i) {
        int i1 = g_mod(i+1, n);
        if (ps_isects(pol[i], pol[i1], p)) return -1;
        else if(((sgn(pol[i].y - p.y) == 1) != (sgn(pol[i1].y - p.y) == 1)) &&
            ccw(pol[i], p, pol[i1]) == sgn(pol[i].y - pol[i1].y)) ++count;
    }
    return count % 2;
}
```

4.4. Miscellaneous primitives. Hash: be051245293a9db9c991d414c598e854

```
bool point_circle(pt p, circle c) {
    return cmp(abs(p - c.c), c.r) <= 0;
}

double ps_distance(pt p, pt a, pt b) {
    p = p - a; b = b - a;
    double coef = min(max((b||p)/(b||b), TYPE(0)), TYPE(1));
    return abs(p - b*coef);
}
```

```
pt circumcenter(pt a, pt b, pt c) {
    return parametric_isect((b+a)/2, (b-a)*I, (c+a)/2, (c-a)*I);
}

bool compy(pt a, pt b) {
    return cmp(a.y, b.y) ? cmp(a.y, b.y) < 0 : cmp(a.x, b.x) < 0;
}

bool compx(pt a, pt b) { return a < b; }
```

4.5. Smallest enclosing circle. Hash: 00dd4dbd6779989a64c1e935443a1d80

```
circle enclosing_circle(vector<pt>& pts) {
    srand(unsigned(time(0)));
    random_shuffle(pts.begin(), pts.end());

    circle c(pt(), -1);
    for(int i = 0; i < pts.size(); ++i) {
        if(point_circle(pts[i], c)) continue;
        c = circle(pts[i], 0);
        for(int j = 0; j < i; ++j) {
            if(point_circle(pts[j], c)) continue;
```

```
        c = circle((pts[i] + pts[j])/2, abs(pts[i] - pts[j])/2);
        for(int k = 0; k < j; ++k) {
            if(point_circle(pts[k], c)) continue;
            pt center = circumcenter(pts[i], pts[j], pts[k]);
            c = circle(center, abs(center - pts[i])/2);
        }
    }
    return c;
}
```

4.6. Convex hull. Hash: 2b14ae1a97e5ff686efb4d7e0e7ca78a

```
pt pivot;
```

```
bool hull_comp(pt a, pt b) {
    int turn = ccw(a, b, pivot);
```

```

    return turn == 1 || (turn == 0 && cmp(norm(a-pivot), norm(b-pivot)) < 0);
}

vector<pt> hull(vector<pt> pts) {
    if(pts.size() <= 1) return pts;
    vector<pt> ret;

    int mini = 0;
    for(int i = 1; i < pts.size(); ++i)
        if(pts[i] < pts[mini])
            mini = i;

    pivot = pts[mini];
    swap(pts[0], pts[mini]);

```

```

    sort(pts.begin() + 1, pts.end(), hull_comp);

    ret.push_back(pts[0]);
    ret.push_back(pts[1]);
    int sz = 2;

    for(int i = 2; i < pts.size(); ++i) {
        while(sz >= 2 && ccw(ret[sz-2], ret[sz-1], pts[i]) <= 0)
            ret.pop_back(), --sz;
        ret.push_back(pts[i]), ++sz;
    }

    return ret;
}

```

4.7. Closest pair of points. Hash: d704271ff258aac5dad13bb04cf0cfb6

```

pair<pt, pt> closest_points_rec(vector<pt>& px, vector<pt>& py) {
    pair<pt, pt> ret;
    double d;

    if(px.size() <= 3) {
        double best = 1e10;
        for(int i = 0; i < px.size(); ++i)
            for(int j = i + 1; j < px.size(); ++j)
                if(dist(px[i], px[j]) < best) {
                    ret = make_pair(px[i], px[j]);
                    best = dist(px[i], px[j]);
                }

        return ret;
    }

    pt split = px[(px.size() - 1)/2];
    vector<pt> qx, qy, rx, ry;
    for(int i = 0; i < px.size(); ++i)
        if(px[i] <= split) qx.push_back(px[i]);
        else rx.push_back(px[i]);

    for(int i = 0; i < py.size(); ++i)
        if(py[i] <= split) qy.push_back(py[i]);
        else ry.push_back(py[i]);

    ret = closest_points_rec(qx, qy);
    pair<pt, pt> rans = closest_points_rec(rx, ry);

```

```

    double delta = dist(ret.first, ret.second);

    if((d = dist(rans.first, rans.second)) < delta) {
        delta = d;
        ret = rans;
    }

    vector<pt> s;
    for(int i = 0; i < py.size(); ++i)
        if(cmp(abs(py[i].x - split.x), delta) <= 0)
            s.push_back(py[i]);

    for(int i = 0; i < s.size(); ++i)
        for(int j = i + 1; j < s.size(); ++j)
            if((d = dist(s[i], s[j])) < delta) {
                delta = d;
                ret = make_pair(s[i], s[j]);
            }

    return ret;
}

pair<pt, pt> closest_points(vector<pt> pts) {
    if(pts.size() == 1) return make_pair(pt(-INF, -INF), pt(INF, INF));

    sort(pts.begin(), pts.end());
    for(int i = 0; i + 1 < pts.size(); ++i)
        if(pts[i] == pts[i+1])

```



```

        return make_pair(pts[i], pts[i+1]);

vector<pt> py = pts;
sort(py.begin(), py.end(), compy);

```

4.8. Kd-tree. Hash: 181bc30d9b4f2bfc8c42ca71101934ba

```

int tree[4*MAXSZ], val[4*MAXSZ];
TYPE split[4*MAXSZ];
vector<pt> pts;

void kd_recurse(int root, int left, int right, bool x) {
    if(left == right) {
        tree[root] = left;
        val[root] = 1;
        return;
    }

    int mid = (right+left)/2;
    nth_element(pts.begin() + left, pts.begin() + mid,
        pts.begin() + right + 1, x ? compx : compy);
    split[root] = x ? pts[mid].x : pts[mid].y;

    kd_recurse(2*root+1, left, mid, !x);
    kd_recurse(2*root+2, mid+1, right, !x);

    val[root] = val[2*root+1] + val[2*root+2];
}

void kd_build() {
    memset(tree, -1, sizeof tree);
    kd_recurse(0, 0, pts.size() - 1, true);
}

int kd_query(int root, TYPE a, TYPE b, TYPE c, TYPE d, TYPE ca = -INF,
    TYPE cb = INF, TYPE cc = -INF, TYPE cd = INF, bool x = true) {
    if(a <= ca && cb <= b && c <= cc && cd <= d)
        return val[root];

    if(tree[root] != -1)
        return a <= pts[tree[root]].x && pts[tree[root]].x <= b &&

```

```

        return closest_points_rec(pts, py);
    }

```

```

        c <= pts[tree[root]].y && pts[tree[root]].y <= d ? val[root] : 0;

int ret = 0;
if(x) {
    if(a <= split[root])
        ret += kd_query(2*root+1, a, b, c, d, ca, split[root], cc, cd, !x);
    if(split[root] <= b)
        ret += kd_query(2*root+2, a, b, c, d, split[root], cb, cc, cd, !x);
} else {
    if(c <= split[root])
        ret += kd_query(2*root+1, a, b, c, d, ca, cb, cc, split[root], !x);
    if(split[root] <= d)
        ret += kd_query(2*root+2, a, b, c, d, ca, cb, split[root], cd, !x);
}
return ret;
}

pt kd_neighbor(int root, pt a, bool x) {
    if(tree[root] != -1)
        return a == pts[tree[root]] ? pt(INF, INF) : pts[tree[root]];

    TYPE num = x ? a.x : a.y;
    int term = num <= split[root] ? 1 : 2;
    pt ret;

    TYPE d = norm(a - (ret = kd_neighbor(2*root + term, a, !x)));
    if((split[root] - num)*(split[root] - num) < d) {
        pt ret2 = kd_neighbor(2*root + 3 - term, a, !x);
        if(norm(a - ret2) < d)
            ret = ret2;
    }

    return ret;
}

```

4.9. Range tree. Hash: c81f7107969ade9a64ad085c075d8310

```
vector<pt> pts, tree[MAXSZ];
vector<TYPE> xs;
vector<int> lnk[MAXSZ][2];

int rt_recurse(int root, int left, int right) {
    lnk[root][0].clear(); lnk[root][1].clear(); tree[root].clear();

    if(left == right) {
        vector<pt>::iterator it;
        it = lower_bound(pts.begin(), pts.end(), pt(xs[left], -INF));
        for(; it != pts.end() && cmp(it->x, xs[left]) == 0; ++it)
            tree[root].push_back(*it);
        return tree[root].size();
    }

    int mid = (left + right)/2, cl = 2*root + 1, cr = cl + 1;
    int sz1 = rt_recurse(cl, left, mid);
    int sz2 = rt_recurse(cr, mid + 1, right);

    lnk[root][0].reserve(sz1+sz2+1);
    lnk[root][1].reserve(sz1+sz2+1);
    tree[root].reserve(sz1+sz2);

    int l = 0, r = 0, llink = 0, rlink = 0; pt last;
    while(l < sz1 || r < sz2) {
        if(r == sz2 || (l < sz1 && compy(tree[cl][l], tree[cr][r])))
            tree[root].push_back(last = tree[cl][l++]);
        else tree[root].push_back(last = tree[cr][r++]);

        while(llink < sz1 && compy(tree[cl][llink], last))
            ++llink;
        while(rlink < sz2 && compy(tree[cr][rlink], last))
            ++rlink;

        lnk[root][0].push_back(llink);
        lnk[root][1].push_back(rlink);
    }
}
```

```
lnk[root][0].push_back(tree[cl].size());
lnk[root][1].push_back(tree[cr].size());

return tree[root].size();
}

void rt_build() {
    sort(pts.begin(), pts.end());
    xs.clear();
    for(int i = 0; i < pts.size(); ++i) xs.push_back(pts[i].x);
    xs.erase(unique(xs.begin(), xs.end()), xs.end());
    rt_recurse(0, 0, xs.size() - 1);
}

int rt_query(int root, int l, int r, TYPE a, TYPE b, TYPE c, TYPE d,
             int posl = -1, int posr = -1) {
    if(root == 0 && posl == -1) {
        posl = lower_bound(tree[0].begin(), tree[0].end(), pt(a, c), compy)
            - tree[0].begin();
        posr = upper_bound(tree[0].begin(), tree[0].end(), pt(b, d), compy)
            - tree[0].begin();
    }

    if(posl == posr) return 0;
    if(a <= xs[l] && xs[r] <= b)
        return posr - posl;

    int mid = (l+r)/2, ret = 0;
    if(cmp(a, xs[mid]) <= 0)
        ret += rt_query(2*root+1, l, mid, a, b, c, d,
            lnk[root][0][posl], lnk[root][0][posr]);
    if(cmp(xs[mid+1], b) <= 0)
        ret += rt_query(2*root+2, mid+1, r, a, b, c, d,
            lnk[root][1][posl], lnk[root][1][posr]);

    return ret;
}
```

5. DATA STRUCTURES

5.1. Treap. Hash: 2199b72803301716616a462d9d5e9a66

```

typedef int TYPE;

class treap {
public:
    treap *left, *right;
    int priority, sons;
    TYPE value;

    treap(TYPE value) : left(NULL), right(NULL), value(value), sons(0) {
        priority = rand();
    }

    ~treap() {
        if(left) delete left;
        if(right) delete right;
    }
};

treap* find(treap* t, TYPE val) {
    if(!t) return NULL;
    if(val == t->value) return t;

    if(val < t->value) return find(t->left, val);
    if(val > t->value) return find(t->right, val);
}

void rotate_to_right(treap* &t) {
    treap* n = t->left;
    t->left = n->right;
    n->right = t;
    t = n;
}

void rotate_to_left(treap* &t) {
    treap* n = t->right;
    t->right = n->left;
    n->left = t;
    t = n;
}

```

```

void fix_augment(treap* t) {
    if(!t) return;
    t->sons = (t->left ? t->left->sons + 1 : 0) +
        (t->right ? t->right->sons + 1 : 0);
}

void insert(treap* &t, TYPE val) {
    if(!t)
        t = new treap(val);
    else
        insert(val <= t->value ? t->left : t->right, val);

    if(t->left && t->left->priority > t->priority)
        rotate_to_right(t);
    else if(t->right && t->right->priority > t->priority)
        rotate_to_left(t);

    fix_augment(t->left); fix_augment(t->right); fix_augment(t);
}

inline int p(treap* t) {
    return t ? t->priority : -1;
}

void erase(treap* &t, TYPE val) {
    if(!t) return;

    if(t->value != val)
        erase(val < t->value ? t->left : t->right, val);
    else {
        if(!t->left && !t->right)
            delete t, t = NULL;
        else {
            p(t->left) < p(t->right) ? rotate_to_left(t) : rotate_to_right(t);
            erase(t, val);
        }
    }

    fix_augment(t->left); fix_augment(t->right); fix_augment(t);
}

```

5.2. Heap. Hash: e334218955a73d1286ad0fc19e84b642

```

struct heap {
    int heap[MAXV][2], v2n[MAXV];
    int size;

    void init(int sz) __attribute__((always_inline)) {
        memset(v2n, -1, sizeof(int) * sz);
        size = 0;
    }

    void swap(int& a, int& b) __attribute__((always_inline)) {
        int temp = a;
        a = b;
        b = temp;
    }

    void s(int a, int b) __attribute__((always_inline)) {
        swap(v2n[heap[a][1]], v2n[heap[b][1]]);
        swap(heap[a][0], heap[b][0]);
        swap(heap[a][1], heap[b][1]);
    }

    int extract_min() {
        int ret = heap[0][1];
        s(0, --size);

        int cur = 0, next = 2;
        while(next < size) {
            if(heap[next][0] > heap[next - 1][0])
                next--;
            if(heap[next][0] >= heap[cur][0])
                break;
        }
    }

```

```

        s(next, cur);
        cur = next;
        next = 2*cur + 2;
    }
    if(next == size && heap[next - 1][0] < heap[cur][0])
        s(next - 1, cur);

    return ret;
}

void decrease_key(int vertex, int new_value) __attribute__((always_inline))
{
    if(v2n[vertex] == -1) {
        v2n[vertex] = size;
        heap[size++][1] = vertex;
    }

    heap[v2n[vertex]][0] = new_value;

    int cur = v2n[vertex];
    while(cur >= 1) {
        int parent = (cur - 1)/2;
        if(new_value >= heap[parent][0])
            break;

        s(cur, parent);
        cur = parent;
    }
}
};

```

5.3. Big numbers (PUC-Rio). Hash: a7d74e7158634f9201c19235badd3364

```

const int DIG = 4;
const int BASE = 10000; // BASE**3 < 2**51
const int TAM = 2048;

struct bigint {
    int v[TAM], n;
    bigint(int x = 0): n(1) {
        memset(v, 0, sizeof(v));
    }

```

```

        v[n++] = x; fix();
    }
    bigint(char *s): n(1) {
        memset(v, 0, sizeof(v));
        int sign = 1;
        while (*s && !isdigit(*s)) if (*s++ == '-') sign *= -1;
        char *t = strdup(s), *p = t + strlen(t);
        while (p > t) {

```

```

        *p = 0; p = max(t, p - DIG);
        sscanf(p, "%d", &v[n]);
        v[n++] *= sign;
    }
    free(t); fix();
}

bigint& fix(int m = 0) {
    n = max(m, n);
    int sign = 0;
    for (int i = 1, e = 0; i <= n || e && (n = i); i++) {
        v[i] += e; e = v[i] / BASE; v[i] %= BASE;
        if (v[i]) sign = (v[i] > 0) ? 1 : -1;
    }

    for (int i = n - 1; i > 0; i--)
        if (v[i] * sign < 0) { v[i] += sign * BASE; v[i+1] -= sign; }
    while (n && !v[n]) n--;
    return *this;
}

int cmp(const bigint& x = 0) const {
    int i = max(n, x.n), t = 0;
    while (1) if ((t = ::cmp(v[i], x.v[i])) || i-- == 0) return t;
}

bool operator <(const bigint& x) const { return cmp(x) < 0; }
bool operator ==(const bigint& x) const { return cmp(x) == 0; }
bool operator !=(const bigint& x) const { return cmp(x) != 0; }

operator string() const {
    ostringstream s; s << v[n];
    for (int i = n - 1; i > 0; i--) {
        s.width(DIG); s.fill('0'); s << abs(v[i]);
    }
    return s.str();
}

friend ostream& operator <<(ostream& o, const bigint& x) {
    return o << (string) x;
}

bigint& operator +=(const bigint& x) {
    for (int i = 1; i <= x.n; i++) v[i] += x.v[i];
    return fix(x.n);
}

bigint operator +(const bigint& x) { return bigint(*this) += x; }
bigint& operator -=(const bigint& x) {
    for (int i = 1; i <= x.n; i++) v[i] -= x.v[i];

```

```

        return fix(x.n);
    }
    bigint operator -(const bigint& x) { return bigint(*this) -= x; }
    bigint operator -() { bigint r = 0; return r -= *this; }
    void ams(const bigint& x, int m, int b) { // *this += (x * m) << b;
        for (int i = 1, e = 0; (i <= x.n || e) && (n = i + b); i++) {
            v[i+b] += x.v[i] * m + e; e = v[i+b] / BASE; v[i+b] %= BASE;
        }
    }
    bigint operator *(const bigint& x) const {
        bigint r;
        for (int i = 1; i <= n; i++) r.ams(x, v[i], i-1);
        return r;
    }
    bigint& operator *=(const bigint& x) { return *this = *this * x; }
    // cmp(x / y) == cmp(x) * cmp(y); cmp(x % y) == cmp(x);
    bigint div(const bigint& x) {
        if (x == 0) return 0;
        bigint q; q.n = max(n - x.n + 1, 0);
        int d = x.v[x.n] * BASE + x.v[x.n-1];
        for (int i = q.n; i > 0; i--) {
            int j = x.n + i - 1;
            q.v[i] = int((v[j] * double(BASE) + v[j-1]) / d);
            ams(x, -q.v[i], i-1);
            if (i == 1 || j == 1) break;
            v[j-1] += BASE * v[j]; v[j] = 0;
        }
        fix(x.n); return q.fix();
    }
    bigint& operator /=(const bigint& x) { return *this = div(x); }
    bigint& operator %=(const bigint& x) { div(x); return *this; }
    bigint operator /(const bigint& x) { return bigint(*this).div(x); }
    bigint operator %(const bigint& x) { return bigint(*this) %= x; }
    bigint pow(int x) {
        if (x < 0) return (*this == 1 || *this == -1) ? pow(-x) : 0;
        bigint r = 1;
        for (int i = 0; i < x; i++) r *= *this;
        return r;
    }
    bigint root(int x) {
        if (cmp() == 0 || cmp() < 0 && x % 2 == 0) return 0;
        if (*this == 1 || x == 1) return *this;
        if (cmp() < 0) return -(*this).root(x);
        bigint a = 1, d = *this;
        while (d != 1) {

```

```

    bigint b = a + (d / 2);
    if (cmp(b.pow(x)) >= 0) { d += 1; a = b; }
}

```

```

    return a;
}
};

```

6. STRING ALGORITHMS

6.1. Morris-Pratt's algorithm. Hash: 0234dfb6e26b39d35704838d84f1e86e

```

int pi[MAXSZ], res[MAXSZ], nres;

void morris_pratt(string text, string pattern) {
    nres = 0;
    pi[0] = -1;
    for(int i = 1; i < pattern.size(); ++i) {
        pi[i] = pi[i-1];
        while(pi[i] >= 0 && pattern[pi[i] + 1] != pattern[i])
            pi[i] = pi[pi[i]];
        if(pattern[pi[i] + 1] == pattern[i]) ++pi[i];
    }
}

```

```

int k = -1; //k + 1 eh o tamanho do match atual
for(int i = 0; i < text.size(); ++i) {
    while(k >= 0 && pattern[k + 1] != text[i])
        k = pi[k];
    if(pattern[k + 1] == text[i]) ++k;
    if(k + 1 == pattern.size()) {
        res[nres++] = i - k;
        k = pi[k];
    }
}
}

```

6.2. Manacher's algorithm. Hash: 3dc4ba5a2519725da8ba9ae147d9f8d7

```

vector<int> manacher(const string& s) {
    string s2 = "#";
    for(unsigned int i = 0; i < s.size(); i++)
        s2 += s[i], s2 += '#';

    int c = 1, sz = (int)s2.size(), j;
    vector<int> ans(sz);
    while(c < sz) {
        while(c > ans[c] && c+ans[c]+1 < sz && s2[c-ans[c]-1] == s2[c+ans[c]+1])
            ans[c]++;
    }
}

```

```

j = 1;
while(c+j < sz && j < ans[c]-ans[c-j])
    ans[c+j] = ans[c-j], j++;
if(c+j < sz)
    ans[c+j] = ans[c]-j;
c += j;

return ans;
}

```

6.3. Kärkkäinen-Sanders' suffix array algorithm. Hash: f52d447fe031ca31834ce0b3c4c828f9

```

bool k_cmp(int a1, int b1, int a2, int b2, int a3 = 0, int b3 = 0) {
    return a1 != b1 ? a1 < b1 : (a2 != b2 ? a2 < b2 : a3 < b3);
}

int bucket[MAXSZ+1], tmp[MAXSZ];

```

```

template<class T> void k_radix(T keys, int *in, int *out,
    int off, int n, int k) {
    memset(bucket, 0, sizeof(int) * (k+1));

    for(int j = 0; j < n; j++)

```

```

        bucket[keys[in[j]+off]]++;
    for(int j = 0, sum = 0; j <= k; j++)
        sum += bucket[j], bucket[j] = sum - bucket[j];
    for(int j = 0; j < n; j++)
        out[bucket[keys[in[j]+off]]++] = in[j];
}

int m0[MAXSZ/3+1];
vector<int> k_rec(const vector<int>& v, int k) {
    int n = v.size()-3, sz = (n+2)/3, sz2 = sz + n/3;
    if(n < 2) return vector<int>(n);

    vector<int> sub(sz2+3);
    for(int i = 1, j = 0; j < sz2; i += i%3, j++)
        sub[j] = i;

    k_radix(v.begin(), &sub[0], tmp, 2, sz2, k);
    k_radix(v.begin(), tmp, &sub[0], 1, sz2, k);
    k_radix(v.begin(), &sub[0], tmp, 0, sz2, k);

    int last[3] = {-1, -1, -1}, unique = 0;
    for(int i = 0; i < sz2; i++) {
        bool diff = false;
        for(int j = 0; j < 3; last[j] = v[tmp[i]+j], j++)
            diff |= last[j] != v[tmp[i]+j];
        unique += diff;

        if(tmp[i]%3 == 1) sub[tmp[i]/3] = unique;
        else sub[tmp[i]/3 + sz] = unique;
    }

    vector<int> rec;
    if(unique < sz2) {
        rec = k_rec(sub, unique);
        rec.resize(sz2+sz);
        for(int i = 0; i < sz2; i++) sub[rec[i]] = i+1;
    } else {
        rec.resize(sz2+sz);
        for(int i = 0; i < sz2; i++) rec[sub[i]-1] = i;
    }

    for(int i = 0, j = 0; j < sz; i++)
        if(rec[i] < sz)
            tmp[j++] = 3*rec[i];
    k_radix(v.begin(), tmp, m0, 0, sz, k);

```

```

    for(int i = 0; i < sz2; i++)
        rec[i] = rec[i] < sz ? 3*rec[i] + 1 : 3*(rec[i] - sz) + 2;

    int prec = sz2-1, p0 = sz-1, pret = sz2+sz-1;
    while(prec >= 0 && p0 >= 0)
        if(rec[prec]%3 == 1 && k_cmp(v[m0[p0]], v[rec[prec]],
            sub[m0[p0]/3], sub[rec[prec]/3+sz]) ||
            rec[prec]%3 == 2 && k_cmp(v[m0[p0]], v[rec[prec]],
            v[m0[p0]+1], v[rec[prec]+1],
            sub[m0[p0]/3+sz], sub[rec[prec]/3+1]))
            rec[pret--] = rec[prec--];
        else
            rec[pret--] = m0[p0--];
    if(p0 >= 0) memcpy(&rec[0], m0, sizeof(int) * (p0+1));

    if(n%3==1) rec.erase(rec.begin());
    return rec;
}

vector<int> karkkainen(const string& s) {
    int n = s.size(), cnt = 1;
    vector<int> v(n + 3);

    for(int i = 0; i < n; i++) v[i] = i;
    k_radix(s.begin(), &v[0], tmp, 0, n, 256);
    for(int i = 0; i < n; cnt += (i+1 < n && s[tmp[i+1]] != s[tmp[i]]), i++)
        v[tmp[i]] = cnt;

    return k_rec(v, cnt);
}

vector<int> lcp(const string& s, const vector<int>& sa) {
    int n = sa.size();
    vector<int> prm(n), ans(n-1);
    for(int i = 0; i < n; i++) prm[sa[i]] = i;

    for(int h = 0, i = 0; i < n; i++)
        if(prm[i]) {
            int j = sa[prm[i]-1], ij = max(i, j);
            while(ij + h < n && s[i+h] == s[j+h]) h++;
            ans[prm[i]-1] = h;
            if(h) h--;
        }
    return ans;
}

```

6.4. Aho-Corasick's algorithm (Shinta). Hash: a4da9645039910c608ece0f3cf7ae9eb

```
int term[N]; map<char, int> next[N]; int T[N]; int cnt = 1;
void add(string s, int it){
    int node = 0;
    f(i, 0, s.size()){
        char c = s[i];
        if(!next[node].count(c)) term[cnt] = 0, next[node][c] = cnt, cnt++;
        node = next[node][c];
    }
    term[node] = 1 << it;
}
void aho(){
    queue<int> q;
    for(char c = 'a'; c <= 'z'; c++){
        if(next[0].count(c)) q.push(next[0][c]), T[next[0][c]] = 0;
        else next[0][c] = 0;
    }
}
```

```
while(!q.empty()){
    int u = q.front(); q.pop();
    for(char c = 'a'; c <= 'z'; c++) if(next[u].count(c)){
        int v = next[u][c];
        int x = T[u];
        while(!next[x].count(c)) x = T[x];
        x = next[x][c];
        T[v] = x;
        q.push(v);
        term[v] |= term[x];
    }
}
```

6.5. Aho-Corasick's algorithm (UFPE). Hash: 273f4391174d22898bfe3f2415f95915

```
struct No {
    int fail;
    vector< pair<int,int> > out; // num e tamanho do padrao
    //bool marc; // p/ decisao
    map<char, int> lista;
    int next; // aponta para o proximo sufixo que tenha out.size > 0
};
No arvore[1000003]; // quantida maxima de nos
//bool encontrado[1005]; // quantidade maxima de padroes, p/ decisao
int qtdNos, qtdPadroes;

// Funcao para inicializar
void inic() {
    arvore[0].fail = -1;
    arvore[0].lista.clear();
    arvore[0].out.clear();
    arvore[0].next = -1;
    qtdNos = 1;
    qtdPadroes = 0;
    //arvore[0].marc = false; // p/ decisao
    //memset(encontrado, false, sizeof(encontrado)); // p/ decisao
}

// Funcao para adicionar um padrao
```

```
void adicionar(char *padrao) {
    int no = 0, len = 0;
    for (int i = 0; padrao[i]; i++, len++) {
        if (arvore[no].lista.find(padrao[i]) == arvore[no].lista.end()) {
            arvore[qtdNos].lista.clear(); arvore[qtdNos].out.clear();
            //arvore[qtdNos].marc = false; // p/ decisao
            arvore[no].lista[padrao[i]] = qtdNos;
            no = qtdNos++;
        } else no = arvore[no].lista[padrao[i]];
    }
    arvore[no].out.push_back(pair<int,int>(qtdPadroes++, len));
}

// Ativar Aho-corasick, ajustando funcoes de falha
void ativar() {
    int no, v, f, w;
    queue<int> fila;
    for (map<char,int>::iterator it = arvore[0].lista.begin();
        it != arvore[0].lista.end(); it++) {
        arvore[no = it->second].fail = 0;
        arvore[no].next = arvore[0].out.size() ? 0 : -1;
        fila.push(no);
    }
    while (!fila.empty()) {
```



```

no = fila.front(); fila.pop();
for (map<char,int>::iterator it=arvore[no].lista.begin();
    it!=arvore[no].lista.end(); it++) {
    char c = it->first;
    v = it->second;
    fila.push(v);
    f = arvore[no].fail;
    while (arvore[f].lista.find(c) == arvore[f].lista.end()) {
        if (f == 0) { arvore[0].lista[c] = 0; break; }
        f = arvore[f].fail;
    }
    w = arvore[f].lista[c];
    arvore[v].fail = w;
    arvore[v].next = arvore[w].out.size() ? w : arvore[w].next;
}
}

// Buscar padroes no aho-corasik
void buscar(char *input) {
    int v, no = 0;

```

```

for (int i = 0 ; input[i] ; i++) {
    while (arvore[no].lista.find(input[i]) == arvore[no].lista.end()) {
        if (no == 0) { arvore[0].lista[input[i]] = 0; break; }
        no = arvore[no].fail;
    }
    v = no = arvore[no].lista[input[i]];
    // marcar os encontrados
    while (v != -1 /* && !arvore[v].marc */) { // p/ decisao
        //arvore[v].marc = true; // p/ decisao: nao continua a lista
        for (int k = 0 ; k < arvore[v].out.size() ; k++) {
            //encontrado[arvore[v].out[k].first] = true; // p/ decisao
            printf("Padrao_%d_na_posicao_%d\n", arvore[v].out[k].first,
                i-arvore[v].out[k].second+1);
        }
        v = arvore[v].next;
    }
}
// for (int i = 0 ; i < qtdPadroes ; i++)
//printf("%s\n", encontrado[i]?"y":"n"); // p/ decisao
}

```

7. USEFUL MATHEMATICAL FACTS

7.1. Prime counting function ($\pi(x)$). The prime counting function is asymptotic to $\frac{x}{\log x}$, by the prime number theorem.

| | | | | | | | | |
|----------|----|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| x | 10 | 10 ² | 10 ³ | 10 ⁴ | 10 ⁵ | 10 ⁶ | 10 ⁷ | 10 ⁸ |
| $\pi(x)$ | 4 | 25 | 168 | 1.229 | 9.592 | 78.498 | 664.579 | 5.761.455 |

7.2. Partition function. The partition function $p(x)$ counts show many ways there are to write the integer x as a sum of integers.

| | | | | | | | |
|------|--------|--------|--------|---------|---------|-------------|--------|
| x | 36 | 37 | 38 | 39 | 40 | 41 | 42 |
| p(x) | 17.977 | 21.637 | 26.015 | 31.185 | 37.338 | 44.583 | 53.174 |
| x | 43 | 44 | 45 | 46 | 47 | 100 | |
| p(x) | 63.261 | 75.175 | 89.134 | 105.558 | 125.754 | 190.569.292 | |

7.3. Catalan numbers. Catalan numbers are defined by the recurrence:

$$C_{n+1} = \sum_{i=0}^n C_i C_{n-i}$$

A closed formula for Catalan numbers is:

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1}$$

7.4. Stirling numbers of the first kind. These are the number of permutations of I_n with exactly k disjoint cycles. They obey the recurrence:

$$\begin{bmatrix} n \\ k \end{bmatrix} = (n-1) \begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$$

7.5. Stirling numbers of the second kind. These are the number of ways to partition I_n into exactly k sets. They obey the recurrence:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = k \begin{Bmatrix} n-1 \\ k \end{Bmatrix} + \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$$

A “closed” formula for it is:

$$\begin{Bmatrix} n \\ k \end{Bmatrix} = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

7.6. Bell numbers. These count the number of ways to partition I_n into subsets. They obey the recurrence:

$$\mathcal{B}_{n+1} = \sum_{k=0}^n \binom{n}{k} \mathcal{B}_k$$

| | | | | | | | | |
|-----------------|----|-----|-----|-------|--------|---------|---------|-----------|
| x | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| \mathcal{B}_x | 52 | 203 | 877 | 4.140 | 21.147 | 115.975 | 678.570 | 4.213.597 |

7.7. Turán’s theorem. No graph with n vertices that is K_{r+1} -free can have more edges than the Turán graph: A k -partite complete graph with sets of size as equal as possible.

7.8. Generating functions. A list of generating functions for useful sequences:

| | |
|--|-------------------------|
| $(1, 1, 1, 1, 1, \dots)$ | $\frac{1}{1-z}$ |
| $(1, -1, 1, -1, 1, \dots)$ | $\frac{1}{1+z}$ |
| $(1, 0, 1, 0, 1, 0, \dots)$ | $\frac{1}{1-z^2}$ |
| $(1, 0, \dots, 0, 1, 0, 1, 0, \dots, 0, 1, 0, \dots)$ | $\frac{1}{1-z^2}$ |
| $(1, 2, 3, 4, 5, 6, \dots)$ | $\frac{1}{(1-z)^2}$ |
| $(1, \binom{m+1}{m}, \binom{m+2}{m}, \binom{m+3}{m}, \dots)$ | $\frac{1}{(1-z)^{m+1}}$ |
| $(1, c, \binom{c+1}{2}, \binom{c+2}{3}, \dots)$ | $\frac{1}{(1-z)^c}$ |
| $(1, c, c^2, c^3, \dots)$ | $\frac{1}{1-cz}$ |
| $(0, 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots)$ | $\ln \frac{1}{1-z}$ |

A neat manipulation trick is:

$$\frac{1}{1-z} G(z) = \sum_n \sum_{k \leq n} g_k z^n$$

7.9. Polyominoes. How many free (rotation, reflection), one-sided (rotation) and fixed n -ominoes are there?

| n | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|-----------|---|----|----|-----|-----|-------|-------|--------|
| free | 2 | 5 | 12 | 35 | 108 | 369 | 1.285 | 4.655 |
| one-sided | 2 | 7 | 18 | 60 | 196 | 704 | 2.500 | 9.189 |
| fixed | 6 | 19 | 63 | 216 | 760 | 2.725 | 9.910 | 36.446 |

7.10. The twelvefold way (from Stanley). How many functions $f: N \rightarrow X$ are there?

| N | X | Any f | Injective | Surjective |
|---------|---------|---|----------------|--|
| dist. | dist. | x^n | $(x)_n$ | $x! \left\{ \begin{smallmatrix} n \\ x \end{smallmatrix} \right\}$ |
| indist. | dist. | $\binom{x+n-1}{n}$ | $\binom{x}{n}$ | $\binom{n-1}{n-x}$ |
| dist. | indist. | $\left\{ \begin{smallmatrix} n \\ 1 \end{smallmatrix} \right\} + \dots + \left\{ \begin{smallmatrix} n \\ x \end{smallmatrix} \right\}$ | $[n \leq x]$ | $\left\{ \begin{smallmatrix} n \\ k \end{smallmatrix} \right\}$ |
| indist. | indist. | $p_1(n) + \dots p_x(n)$ | $[n \leq x]$ | $p_x(n)$ |

Where $\binom{a}{b} = \frac{1}{b!}(a)_b$ and $p_x(n)$ is the number of ways to partition the integer n using x summands.

7.11. Common integral substitutions. And finally, a list of common substitutions:

| | | |
|----------------------------|-------------------|---------------------------------------|
| $\int F(\sqrt{ax+b})dx$ | $u = \sqrt{ax+b}$ | $\frac{2}{a} \int uF(u)du$ |
| $\int F(\sqrt{a^2-x^2})dx$ | $x = a \sin u$ | $a \int F(a \cos u) \cos u du$ |
| $\int F(\sqrt{x^2+a^2})dx$ | $x = a \tan u$ | $a \int F(a \sec u) \sec^2 u du$ |
| $\int F(\sqrt{x^2-a^2})dx$ | $x = a \sec u$ | $a \int F(a \tan u) \sec u \tan u du$ |
| $\int F(e^{ax})dx$ | $u = e^{ax}$ | $\frac{1}{a} \int \frac{F(u)}{u} du$ |
| $\int F(\ln x)dx$ | $u = \ln x$ | $\int F(u)e^u du$ |

7.12. Table of non-trigonometric integrals. Some useful integrals are:

| | |
|-----------------------------------|--|
| $\int \frac{dx}{x^2+a^2}$ | $\frac{1}{a} \arctan \frac{x}{a}$ |
| $\int \frac{dx}{x^2-a^2}$ | $\frac{1}{2a} \ln \frac{x-a}{x+a}$ |
| $\int \frac{dx}{a^2-x^2}$ | $\frac{1}{2a} \ln \frac{a+x}{a-x}$ |
| $\int \frac{dx}{\sqrt{a^2-x^2}}$ | $\arcsin \frac{x}{a}$ |
| $\int \frac{dx}{\sqrt{x^2-a^2}}$ | $\ln(u + \sqrt{x^2-a^2})$ |
| $\int \frac{dx}{x\sqrt{x^2-a^2}}$ | $\frac{1}{a} \operatorname{arcsec} \left \frac{u}{a} \right $ |
| $\int \frac{dx}{x\sqrt{x^2+a^2}}$ | $-\frac{1}{a} \ln \left(\frac{a+\sqrt{x^2+a^2}}{x} \right)$ |
| $\int \frac{dx}{x\sqrt{a^2+x^2}}$ | $-\frac{1}{a} \ln \left(\frac{a+\sqrt{a^2+x^2}}{x} \right)$ |

7.13. Table of trigonometric integrals. A list of common and not-so-common trigonometric integrals:

| | |
|---------------------|---|
| $\int \tan x dx$ | $-\ln \cos x $ |
| $\int \cot x dx$ | $\ln \sin x $ |
| $\int \sec x dx$ | $\ln \sec x + \tan x $ |
| $\int \csc x dx$ | $\ln \csc x - \cot x $ |
| $\int \sec^2 x dx$ | $\tan x$ |
| $\int \csc^2 x dx$ | $\cot x$ |
| $\int \sin^n x dx$ | $\frac{-\sin^{n-1} x \cos x}{n} + \frac{n-1}{n} \int \sin^{n-2} x dx$ |
| $\int \cos^n x dx$ | $\frac{\cos^{n-1} x \sin x}{n} + \frac{n-1}{n} \int \cos^{n-2} x dx$ |
| $\int \arcsin x dx$ | $x \arcsin x + \sqrt{1-x^2}$ |
| $\int \arccos x dx$ | $x \arccos x - \sqrt{1-x^2}$ |
| $\int \arctan x dx$ | $x \arctan x - \frac{1}{2} \ln 1-x^2 $ |

7.14. Centroid of a polygon. The x coordinate of the centroid of a polygon is given by $\frac{1}{3A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$, where A is twice the signed area of the polygon.

| | | | |
|--|-----------|-------------------------------------|-----------|
| 1. Estructuras de datos | 2 | 6. Geometría | 20 |
| 1.1. Heap | 2 | 6.1. Área de un polígono | 20 |
| 1.2. Binary indexed tree | 2 | 6.2. Número de enrollamiento | 20 |
| 1.3. RMQ | 3 | 6.3. Cápsula convexa | 20 |
| 1.3.1. Binary Search Tree | 3 | 6.4. Par de puntos más cercanos | 20 |
| 1.3.2. Sparse Table | 4 | 6.5. Terna de puntos más cercanos | 21 |
| 1.4. Sliding window minimum | 4 | 6.6. Librería de geometría | 21 |
| 1.5. Suffix array | 4 | 6.7. Teoremas y propiedades | 22 |
| 2. Matemática | 6 | 7. Programación dinámica | 23 |
| 2.1. GCD | 6 | 7.1. Longest common subsequence | 23 |
| 2.2. Extended GCD e Inverso mod m | 6 | 7.2. Longest increasing subsequence | 23 |
| 2.3. ModExp | 6 | 8. Miscelánea | 24 |
| 2.4. Chinese Remainder Theorem | 6 | 8.1. Algoritmo húngaro | 24 |
| 2.5. Find root | 6 | 8.2. Enteros largos | 24 |
| 2.6. Máscaras de bits | 6 | 8.3. Fracciones | 26 |
| 3. Matrices | 7 | 9. Librerías | 27 |
| 3.1. Operaciones elementales | 7 | 9.1. cmath | 27 |
| 3.2. Gauss-Jordan (con pivoteo) | 7 | 9.2. __builtin_ | 27 |
| 4. Grafos | 9 | 9.3. iomanip | 27 |
| 4.1. Prim | 9 | | |
| 4.2. Kruskal | 9 | | |
| 4.3. Dijkstra | 9 | | |
| 4.4. Bellman-Ford | 10 | | |
| 4.5. Floyd - Warshall | 10 | | |
| 4.6. Edmonds-Karp: flujo máximo en $\mathcal{O}((N + E)F)$ ó $\mathcal{O}(NE^2)$ | 11 | | |
| 4.7. Relabel to front: flujo máximo en $\mathcal{O}(N^3)$ | 11 | | |
| 4.8. Dinic: Flujo maximo en $\mathcal{O}(E.V^2)$ | 12 | | |
| 4.9. Karger: min-cut randomizado | 13 | | |
| 4.10. Flujo de costo mínimo en $\mathcal{O}(BN^3)$ | 13 | | |
| 4.11. Maximum bipartite matching | 14 | | |
| 4.12. Puntos de articulación | 14 | | |
| 4.13. Circuito euleriano | 15 | | |
| 4.14. Componentes fuertemente conexas | 16 | | |
| 4.15. Lowest Common Ancestor | 16 | | |
| 4.16. Extras y adaptaciones | 17 | | |
| 5. Strings | 18 | | |
| 5.1. Knuth-Morris-Pratt | 18 | | |
| 5.2. Aho-Corasik | 18 | | |

1.1. Heap

Funciones para transformar un array en una *heap*. La cantidad N de datos en la *heap* se encuentra en el primer elemento del array, y los elementos en sí están en los índices $1, \dots, N$. La función $SWAPH(a, b)$ intercambia dos elementos de la *heap* (posiblemente intercambiando también referencias a esos elementos en otras estructuras). La función $compare$ compara dos elementos de la *heap*: si se usa $a < b$ se tiene una *min-heap*, y si se usa $a > b$ se tiene una *max-heap*. [REQUIERE: $SWAPH(a, b)$]

```

1 #define SWAPH(a,b) ( SWAP(A[(a)],A[(b)]) )
2
3 bool compare(int a, int b) {return a < b;}
4
5 #define PARENT(n) ( (n)/2 )
6 #define LEFT(n) ( (n)<<1 )
7 #define RIGHT(n) ( ((n)<<1)+1 )
8
9 void heapify(int A[], int i, bool cmp(int, int)) {
10     int best;
11     int l = LEFT(i);
12     int r = RIGHT(i);
13     if (l <= A[0] && cmp(A[l], A[i]) == true) best = l;
14     else best = i;
15     if (r <= A[0] && cmp(A[r], A[best]) == true) best = r;
16     if (best != i) {
17         SWAPH(i, best);
18         heapify(A, best, cmp);
19     }
20 }
21
22 void build_heap(int A[], bool cmp(int, int)) {
23     for (int i=A[0]/2; i>=1; i--) heapify(A, i, cmp);
24 }
25
26 void heapsort(int A[], bool cmp(int, int)) {
27     build_heap(A, cmp);
28     for (int i=A[0]; i>=2; i--) {
29         SWAPH(1, i);
30         A[0]--;
31         heapify(A, 1, cmp);
32     }
33 }
34
35 int top(int A[]) {return A[1];}
36
37 int pop(int A[], bool cmp(int, int)) {
38     int max = A[1];
39     SWAPH(1, A[0]);
40     A[0]--;
41     heapify(A, 1, cmp);
42     return max;

```

```

43 }
44
45 void update(int A[], int i, bool cmp(int, int)) {
46     while (i > 1 && cmp(A[PARENT(i)], A[i]) == false) {
47         SWAPH(i, PARENT(i));
48         i = PARENT(i);
49     }
50 }
51
52 void insert(int A[], bool cmp(int, int)) {
53     A[0]++;
54     update(A, A[0], cmp);
55 }

```

1.2. Binary indexed tree

Funciones para trabajar con un array como un *binary indexed tree*. get_cf devuelve la suma de las frecuencias en el rango $[0, idx]$, get_f devuelve la frecuencia de idx , y upd_f incrementa la frecuencia de idx en la cantidad f (donde f puede ser tanto positivo como negativo). Todas las funciones toman como argumento adicional el array en el que está el *bit*, y funcionan en tiempo $\mathcal{O}(\log N)$. Para inicializar el *bit* hay que poner todos los elementos del array que se va a usar en 0, y definir la variable $MAXN$ como el tamaño máximo del array.

```

1 int get_cf(int idx, int *bit) {
2     int cf = bit[0];
3     while (idx > 0) {
4         cf += bit[idx];
5         idx &= idx-1;
6     }
7     return cf;
8 }
9
10 void upd_f(int idx, int f, int *bit) {
11     if (idx == 0) bit[idx] += f;
12     else while (idx < MAXN) {
13         bit[idx] += f;
14         idx += idx&(-idx);
15     }
16 }
17
18 int get_f(int idx, int *bit) {
19     int f = bit[idx];
20     if (idx > 0) {
21         int p = idx&(idx-1); idx--;
22         while (p != idx) {
23             f -= bit[idx];
24             idx &= idx-1;
25         }
26     }

```

```

27 |     return f;
28 | }

```

Para trabajar en dos dimensiones, hay que agregar las funciones *bi_get_cf*, *bi_get_f* y *bi_upd_f*, y definir *MAXM* como el número máximo de columnas (*MAXN* se transforma en el número máximo de filas).

```

1 | int bi_get_cf(int idx, int idy, int **bit) {
2 |     int cf = get_cf(idy, bit[0]);
3 |     while (idx > 0) {
4 |         cf += get_cf(idy, bit[idx]);
5 |         idx &= idx-1;
6 |     }
7 |     return cf;
8 | }
9 |
10 | void bi_upd_f(int idx, int idy, int f, int **bit) {
11 |     if (idx == 0) upd_f(idy, f, bit[0]);
12 |     else while (idx < MAXM) {
13 |         upd_f(idy, f, bit[idx]);
14 |         idx += idx&(-idx);
15 |     }
16 | }
17 |
18 | int bi_get_f(int idx, int idy, int **bit) {
19 |     int f = get_f(idy, bit[idx]);
20 |     if (idx > 0) {
21 |         int p = idx&(idx-1); idx--;
22 |         while (p != idx) {
23 |             f -= get_f(idy, bit[idx]);
24 |             idx &= idx-1;
25 |         }
26 |     }
27 |     return f;
28 | }

```

1.3. RMQ

1.3.1. Binary Search Tree

Funciones para armar el *binary search tree* de un array. *rmq_init* inicializa el árbol binario la primera vez que va a usarse (nótese que debe darse $s < e$); *rmq_query* devuelve el mínimo en el rango $[a, b]$; *rmq_update* actualiza el árbol binario luego de que se haya modificado la posición p del array; *rmq_find* encuentra el primer elemento en el rango $[a, b]$ que es más chico que el valor v . La inicialización es en tiempo $\mathcal{O}(N)$, y todas las demás operaciones son en tiempo $\mathcal{O}(\log N)$. Todas las funciones deben ser llamadas con los argumentos $n = 1$, $[s, e) = [0, N)$, donde N es el tamaño del array, siendo además m y rmq el array y su árbol binario de búsqueda respectivamente. Para encontrar

el máximo en lugar del mínimo, hay que modificar la función de comparación *compare*. El array en el que se va a guardar el árbol binario debe tener tamaño $2 * MAXN$, donde $MAXN$ es el tamaño del array original.

```

1 | #define LEFT(n) ( 2*(n) )
2 | #define RIGHT(n) ( 2*(n)+1 )
3 |
4 | bool compare(int a, int b) {
5 |     return a < b;
6 | }
7 |
8 | void rmq_init(int n, int s, int e, int *m, int *rmq) {
9 |     if (s+1 == e) rmq[n] = s;
10 |    else {
11 |        rmq_init(LEFT(n), s, (s+e)/2, m, rmq);
12 |        rmq_init(RIGHT(n), (s+e)/2, e, m, rmq);
13 |        if (compare(m[rmq[LEFT(n)]], m[rmq[RIGHT(n)]]) == true) rmq[n]
14 |            = rmq[LEFT(n)];
15 |        else rmq[n] = rmq[RIGHT(n)];
16 |    }
17 | }
18 | int rmq_query(int n, int s, int e, int *m, int *rmq, int a, int b)
19 | {
20 |     if (a >= e || b <= s) return -1;
21 |     else if (s >= a && e <= b) return rmq[n];
22 |     else {
23 |         int l = rmq_query(LEFT(n), s, (s+e)/2, m, rmq, a, b);
24 |         int r = rmq_query(RIGHT(n), (s+e)/2, e, m, rmq, a, b);
25 |         if (l == -1) return r;
26 |         else if (r == -1) return l;
27 |         else if (compare(m[l], m[r]) == true) return l;
28 |         else return r;
29 |     }
30 | }
31 | void rmq_update(int n, int s, int e, int *m, int *rmq, int p) {
32 |     if (s+1 == e) rmq[n] = s;
33 |     else {
34 |         if (p < (s+e)/2) rmq_update(LEFT(n), s, (s+e)/2, m, rmq, p);
35 |         else rmq_update(RIGHT(n), (s+e)/2, e, m, rmq, p);
36 |         if (compare(m[rmq[LEFT(n)]], m[rmq[RIGHT(n)]]) == true ) rmq[
37 |             n] = rmq[LEFT(n)];
38 |         else rmq[n] = rmq[RIGHT(n)];
39 |     }
40 | }
41 | int rmq_find(int n, int s, int e, int *m, int *rmq, int a, int b,
42 |     int v) {
43 |     if (a >= e || b <= s) return -1;
44 |     else if (s >= a && e <= b) {
45 |         if (compare(m[rmq[n]], v) == false) return -1;
46 |         else if (s+1 == e) return rmq[n];

```

```

46 | }
47 |
48 | int l = rmq_find(LEFT(n), s, (s+e)/2, m, rmq, a, b, v);
49 | if (l == -1) return rmq_find(RIGHT(n), (s+e)/2, e, m, rmq, a, b,
50 |     v);
51 | else return l;

```

1.3.2. Sparse Table

Funciones para armar la *sparse table* de un array, que permite encontrar la posición del mínimo en un rango determinado en tiempo $\mathcal{O}(1)$. La función *st_init* inicializa la *sparse table*, mientras que *st_query* devuelve la posición del mínimo en el rango $[s, e)$. Las dos funciones toman como argumentos el array m , su tamaño N , y la *sparse table* $st[LOGMAXN][MAXN]$. Para obtener el máximo en lugar del mínimo hay que invertir las comparaciones de las líneas 7 y 15.

```

1 | void st_init(int *m, int N, int **st) {
2 |     int i, j;
3 |
4 |     for (i=0; i<N; i++) st[0][i] = i;
5 |     for (j=1; (1<<j)<=N; j++) {
6 |         for (i=0; i+(1<<j)<=N; i++) {
7 |             if (m[st[j-1][i]] < m[st[j-1][i+(1<<(j-1))]]) st[j][i] =
8 |                 st[j-1][i];
9 |             else st[j][i] = st[j-1][i+(1<<(j-1))];
10 |        }
11 |    }
12 |
13 | int st_query(int *m, int N, int **st, int s, int e) {
14 |     int k = 31 - __builtin_clz(e-s);
15 |     if (m[st[k][s]] < m[st[k][e-(1<<k)]]) return st[k][s];
16 |     else return st[k][e-(1<<k)];
17 | }

```

1.4. Sliding window minimum

swm(int n[], int m[], int N, int K) genera la lista de mínimos en $n[0, \dots, N-1]$ cuando se la recorre con una ventana de tamaño K . Los mínimos se guardan en $m[0, \dots, N-1]$, donde $m[i]$ contiene el mínimo de la ventana que termina con el elemento $n[i]$ incluido, de modo que el mínimo de la primera ventana completa, $\{n[0], \dots, n[K-1]\}$, está guardado en $m[K-1]$. El algoritmo requiere $\mathcal{O}(N)$ tiempo y $\mathcal{O}(K)$ memoria.

```

1 | #define MAXN 1048576
2 | #define MAXK 1024

```

```

3 |
4 | int s[MAXK];
5 |
6 | void swm(int n[], int m[], int N, int K) {
7 |     int i, S, E;
8 |
9 |     s[0] = 0; S = 0; E = 1; m[0] = n[0];
10 |    for (i=1; i<N; i++) {
11 |        if (s[S] == i-K) {
12 |            S++;
13 |            if (S == MAXK) S = 0;
14 |        }
15 |        while (S != E && n[s[E-1]] >= n[i]) {
16 |            E--;
17 |            if (E == -1) E = MAXK;
18 |        }
19 |        s[E++] = i;
20 |        if (E == MAXK) E = 0;
21 |        m[i] = n[s[S]];
22 |    }
23 | }

```

1.5. Suffix array

Construcción en $\mathcal{O}(N \log^2 N)$ del suffix array y de los longest common prefixes de una cadena. El tamaño máximo de la cadena es $MAXN$, y la cantidad máxima de caracteres del alfabeto es ALF . Debe definirse la variable global N como el tamaño de la cadena cuyo suffix array se desea construir, y debe transformarse la cadena original a un array de int's en el rango $[0, ALF)$. Los longest common prefixes calculados son entre cada elemento y el elemento anterior. [REQUIERE: *stdlib*, *cstring*,]

```

1 | #define MAXN 131072
2 | #define ALF 64
3 |
4 | int DIST, prm[MAXN], N;
5 |
6 | int compare(const void *a, const void *b) {
7 |     int da=*(int*)a + DIST, db=*(int*)b + DIST;
8 |     if (da<N && db<N) return prm[da] - prm[db];
9 |     else if (da<N) return 1;
10 |    else if (db<N) return -1;
11 |    else return db-da;
12 | }
13 |
14 | void build_sufarray(int a[], int m[]) {
15 |     int i, j, c[ALF], p[ALF], g[MAXN], gg[MAXN], s, e;
16 |
17 |     memset(c, 0, sizeof(c));
18 |     for (i=0; i<N; i++) c[a[i]]++;

```

```

19
20 p[0] = 0;
21 for (i=1; i<ALF; i++) p[i] = p[i-1]+c[i-1];
22
23 memcpy(c, p, sizeof(p));
24 for (i=0; i<N; i++) {
25     m[p[a[i]]] = i;
26     prm[i] = p[a[i]];
27     g[p[a[i]]] = c[a[i]];
28     p[a[i]]++;
29 }
30 g[N] = N;
31
32 for (i=1; i<N; i<=&=1) {
33     s = e = 0;
34     while (e <= N) {
35         if (g[s] == g[e]) e++;
36         else {
37             DIST = i;
38             qsort(m+s, e-s, sizeof(int), compare);
39             gg[s] = s;
40             for (j=s+1; j<e; j++) {
41                 if (m[j]+i < N && m[j-1]+i < N && g[prm[m[j]+i]] ==
42                     g[prm[m[j-1]+i]]) {
43                     gg[j] = gg[j-1];
44                 } else gg[j] = j;
45             }
46             s = e;
47         }
48     }
49     memcpy(g, gg, sizeof(gg));
50     for (j=0; j<N; j++) prm[m[j]] = j;
51 }
52 // for (j=0; j<N; j++) printf("%d\n", m[j]);
53 }
54 void build_lcp(int a[], int m[], int lcp[]) {
55     int i, j, h;
56
57     h = lcp[0] = 0;
58     for (i=0; i<N; i++) {
59         if (prm[i] > 0) {
60             j = m[prm[i]-1];
61             while (i+h < N && j+h < N && a[i+h] == a[j+h]) h++;
62             lcp[prm[i]] = h;
63             if (h > 0) h--;
64         }
65     }
66 // for (i=1; i<N; i++) printf("entre %d y %d hay %d\n", m[i-1], m[i], lcp[i]);
67 }

```


2.1. GCD

```
1 | int gcd(int a, int b) { return (b==0) ? a : gcd(b,a%b); }
```

2.2. Extended GCD e Inverso mod m

Dado un par a, b , calcula los coef n, m de modo que $ma + nb = \gcd(a, b)$; $\text{inv}(n, m)$ es el inverso de n modulo m ($(n, m) = 1$)

```
1 | typedef pair<tint, tint> ptt;
2 | ptt egcd(tint a, tint b) {
3 |     if (b == 0) return make_pair(1, 0);
4 |     else {
5 |         ptt RES = egcd(b, a%b);
6 |         return make_pair(RES.second, RES.first - RES.second*(a/b));
7 |     }
8 | }
9 | tint inv(tint n, tint m) {
10 |     ptt EGCD = egcd(n, m);
11 |     return ( (EGCD.first %m) + m) %m;
12 | }
```

2.3. ModExp

Cálculo de $a^b \pmod n$ en $\mathcal{O}(\log b)$.

```
1 | int modexp(int a, int b, int n) {
2 |     int r = 1;
3 |     while (b != 0) {
4 |         if (b&1 == 1) r = (r*a)%n;
5 |         b >>= 1;
6 |         a = (a*a)%n;
7 |     }
8 |     return r;
9 | }
```

2.4. Chinese Reminder Theorem

Dados n modulos coprimos 2 a 2, y correspondientes restos, da la unica solucion modulo MOD al sistema de congruencias $X \equiv r_i \pmod{\text{modulo}_i}$ ($MOD = \prod \text{modulo}_i$) Usa: tint, MAXN, forn, inv (con egcd)

```
1 | #define MAXN 20
2 | tint MOD, modulo[MAXN], resto[MAXN];
3 | void init_MOD(int n) { MOD = 1; forn(i,n) MOD*= modulo[i]; }
4 | tint crt(int n) {
5 |     init_MOD(n);
6 |     tint res = 0;
```

```
7 |     forn(i,n) {
8 |         tint coef = 1;
9 |         forn(j,n) if (i!=j) coef*= modulo[j];
10 |         coef*= inv(coef, modulo[i]);
11 |         res+= (coef*resto[i]) %MOD;
12 |     }
13 |     return res %MOD;
14 | }
```

2.5. Find root

Encuentra la raíz de la función $func$ en el intervalo (s, e) , haciendo búsqueda binaria. [REQUIERE: EPS]

```
1 | double findroot(double func(double x), double s, double e){
2 |     while (e-s > EPS) {
3 |         double mid = (s+e)/2.0;
4 |         if (func(mid)*func(s) > 0.0) s = mid;
5 |         else e = mid;
6 |     }
7 |     return (s+e)/2.0;
8 | }
```

2.6. Máscaras de bits

Para recorrer todos los subconjuntos del conjunto s hacemos

```
1 | for (int mask = s; mask != 0; mask = (mask-1) & s) {
2 |     /* CODIGO */
3 | }
```

Para recorrer todos los subconjuntos de $fixed$ que no tienen ningún elemento de pro se hace

```
1 | int perm = (((1<<N)-1) & ~pro);
2 | int mask = fixed;
3 | while( (mask | pro) != ((1<<N)-1)){
4 |     /* CODIGO */
5 |     mask = ((mask + pro + 1) & perm) | fixed;
6 | }
```

O bien

```
1 | int rest = (1<<N)-1 ^ pro ^ fixed;
2 | for (int mask2 = rest; mask2 != 0; mask2 = (mask2-1)&rest) {
3 |     int mask = mask2 | fixed;
4 |     /* CODIGO */
5 | }
```

Todas las funciones de esta sección toman como argumento una matriz en la forma de una lista de punteros a las filas de la matriz. Entonces, si tenemos una matriz `int A[N][N]`, para pasarla como argumento a una función debemos definir una `int **AA[N]` de modo tal que $AA[i] = A[i]$ para $i = 0, 1, \dots, N - 1$.

3.1. Operaciones elementales

Suma de matrices

Calcula $C = A + B$ en $\mathcal{O}(N^2)$.

```
1 void sum(int **A, int **B, int **C, int N) {
2     for (int i=0; i<N; i++) {
3         for (int j=0; j<N; j++) {
4             C[i][j] = A[i][j] + B[i][j];
5         }
6     }
7 }
```

Producto de matrices

Calcula $C = A \cdot B$ en $\mathcal{O}(N^3)$.

```
1 void dot(int **A, int **B, int **C, int N) {
2     for (int i=0; i<N; i++) {
3         for (int j=0; j<N; j++) {
4             C[i][j] = 0;
5             for (int k=0; k<N; k++) {
6                 C[i][j] += A[i][k]*B[k][j];
7             }
8         }
9     }
10 }
```

Potencia de matrices

Calcula $B = A^k$ en $\mathcal{O}(N^3 \log k)$. Nótese que se modifica la matriz original $A[N][N]$. [REQUIERE: dot(int **A, int **B, int **C, N)]

```
1 void pow(int **A, int **B, int k, int N) {
2     int i, j, C[N][N], *CC[N];
3
4     for (i=0; i<N; i++) {
5         CC[i] = C[i];
6         for (j=0; j<N; j++) B[i][j] = (i==j);
7     }
8
9     while (k > 0) {
10        if ((k&1) == 1) {
```

```
11        dot(A, B, CC, N);
12        for (i=0; i<N; i++) for (j=0; j<N; j++)
13            B[i][j] = C[i][j];
14    }
15    dot(A, A, CC, N);
16    for (i=0; i<N; i++) for (j=0; j<N; j++)
17        A[i][j] = C[i][j];
18    k >>= 1;
19 }
20 }
```

3.2. Gauss-Jordan (con pivoteo)

Solución del sistema lineal $C \cdot \vec{x} = \vec{b}$. La matriz A es de $M \times N$, y se define como $A = C|\vec{b}$. Si se quiere resolver más de una ecuación con una única matriz C y distintos términos independientes \vec{b}_i , se define $A = C|B$, donde B es la matriz cuyas columnas son los vectores \vec{b}_i . Para encontrar A^{-1} , se utiliza $A = C|I_{M \times M}$. La solución del sistema al terminar el algoritmo queda donde antes estaban los términos independientes. El algoritmo es $\mathcal{O}(M^3)$, y hay que definir el tipo `tint` adecuadamente (generalmente con `typedef double tint`). [REQUIERE: ABS(n), SWAPD(a,b,buf)]

```
1 void gaussjordan(tint **A, int M, int N) {
2     int i, j, k, maxi; tint tmp;
3
4     for (i=0; i<M; i++) {
5         maxi = i;
6         for (k=i+1; k<M; k++) {
7             if (ABS(A[k][i]) > ABS(A[maxi][i])) maxi = k;
8         }
9
10        for (j=0; j<N; j++) SWAPD(A[i][j], A[maxi][j], tmp);
11
12        for (j=0; j<N; j++) {
13            if (j != i) A[i][j] /= A[i][i];
14        }
15        A[i][i] = 1;
16
17        for (k=0; k<M; k++) {
18            if (k != i) {
19                for (j=i+1; j<N; j++)
20                    A[k][j] -= A[k][i]*A[i][j];
21                A[k][i] = 0;
22            }
23        }
24    }
25 }
26
27
```

```
28
29 bool invert(double **A, double **B, int N) {
30     int i, j, k, jmax;
31     double tmp;
32
33     for (i=1; i<=N; i++) {
34         //Encontrar el maximo elemento de A en la columna i con fila
35             >= i
36         jmax = i;
37         for (j=i+1; j<=N; j++) {
38             if (ABS(A[j][i]) > ABS(A[jmax][i])) jmax = j;
39         }
40         //Intercambiar las filas i y jmax
41         for (j=1; j<=N; j++) {
42             swap(A[i][j], A[jmax][j]);
43             swap(B[i][j], B[jmax][j]);
44         }
45
46         //Controlar que la matriz sea invertible
47         if (A[i][i] == 0.0) return false;
48
49         //Normalizar la fila i
50         tmp = A[i][i];
51         for (j=1; j<=N; j++) {
52             A[i][j] /= tmp;
53             B[i][j] /= tmp;
54         }
55
56         //Eliminar los valores no nulos de la columna i
57         for (j=1; j<=N; j++) {
58             if (i == j) continue;
59
60             tmp = A[j][i];
61             for (k=1; k<=N; k++) {
62                 A[j][k] -= A[i][k]*tmp;
63                 B[j][k] -= B[i][k]*tmp;
64             }
65         }
66     }
67     return true;
68 }
```

4.1. Prim

Cálculo del peso del *minimum spanning tree* de un grafo no dirigido en $\mathcal{O}(V^2 + E)$. Los N nodos están en el array global $V[N]$, y tienen los atributos v (que indica si el nodo ya está en el árbol), c (el vector de nodos adyacentes) y ind (la posición de la arista que debe ser considerada cuando vuelva a analizarse el nodo). El vector c de cada nodo debe ser inicializado con los nodos adyacentes a él en orden creciente del valor de las aristas. El valor de la arista en sí se encuentra en la matriz $A[N][N]$. Si el grafo no es conexo, el algoritmo calcula sólo el peso del *minimum spanning tree* de la componente conexa del nodo R . [REQUIERE: *vector*]

```

1 int prim(int R, int N) {
2     int i, j, BEST, MIN, MST;
3
4     for (i=0; i<N; i++) {
5         n[i].v = false;
6         n[i].ind = 0;
7     }
8     n[R].v = true;
9
10    MST = 0;
11    for (j=0; j<N-1; j++) {
12        BEST = -1;
13        for (i=0; i<N; i++) {
14            while (n[i].v==true && n[i].ind < n[i].c.size() && n[n[i].
15                c[n[i].ind]].v==true) {
16                n[i].ind++;
17            }
18            if (n[i].v==true && n[i].ind < n[i].c.size() && (BEST ==
19                -1 || A[i][n[i].c[n[i].ind]] < MIN)) {
20                BEST = n[i].c[n[i].ind];
21                MIN = A[i][n[i].c[n[i].ind]];
22            }
23        }
24        if (BEST != -1) {MST += MIN; n[BEST].v = true;}
25        else {break;}
26    }
27    return MST;
28 }
```

4.2. Kruskal

Cálculo del peso del *minimum spanning tree* de una grafo no dirigido en $\mathcal{O}(E \log V)$. Los N nodos están en el array global $n[N]$, y las E aristas están ordenadas por pesos crecientes en el array global $e[E]$. Usar **void join_rank(int n)** para la máxima optimización, y **void join(int n)** si no es necesario. Los nodos tienen los atributos f (el padre) y r (el rango, necesario si se usa *join_rank*),

mientras que las aristas tienen los atributos w (el peso) y a y b (los extremos). Si el grafo no es conexo se obtiene el peso del *minimum spanning forest*, es decir del *minimum spanning tree* de cada componente conexa.

```

1 int getfather(int n) {
2     if (V[n].f != n) V[n].f = getfather(V[n].f);
3     return V[n].f;
4 }
5 void join(int a, int b) {V[getfather(a)].f = getfather(b);}
6 void join_rank(int a, int b) {
7     int fa = getfather(a), fb = getfather(b);
8     if (V[fa].r > V[fb].r) {V[fb].f = fa;}
9     else if (V[fa].r < V[fb].r) {V[fa].f = fb;}
10    else {V[fa].f = fb; V[fb].r++;}
11 }
12 int kruskal(int N, int E) {
13     int i, MST;
14     for (i=0; i<N; i++) { V[i].f = i, V[i].r = 0;}
15     MST = 0;
16     for (i=0; i<E; i++) if (getfather(e[i].a) != getfather(e[i].b))
17         {
18             join_rank(e[i].a, e[i].b);
19             MST += e[i].w;
20         }
21     return MST;
22 }
```

4.3. Dijkstra

Cálculo de la mínima distancia desde s hasta todos los demás nodos en un grafo con aristas de peso no negativo. El array de vectores $adj[i]$ contiene pares cuya primera componente es un nodo adyacente al nodo i , y cuya segunda componente es el peso de la arista que va de i a ese nodo. Al terminar el algoritmo, $mindist[i]$ contiene la mínima distancia del nodo s al nodo i . [REQUIERE: *queue*, *vector*, *forn(i,n)*, *decl(v,c)*, *forall(i,c)*, *mp*]

```

1 typedef pair<int,int> pii;
2 typedef priority_queue<pii> heap;
3
4 const int INF = 1<<29;
5 const int MAXN = 10000;
6
7 heap q;
8 bool vis[MAXN];
9 int mindist[MAXN];
10
11 void init() {
12     while (!q.empty()) q.pop();
13     forn(u,MAXN) {
14         vis[u] = false;
15     }
```

```

15     mindist[u] = INF;
16 }
17 }
18
19 void addNode(int u, int d) {
20     if (mindist[u] <= d) return;
21     mindist[u] = d;
22     q.push(mp(-d,u));
23 }
24
25 void dijkstra(int s, vector<pii> adj[]) {
26     init();
27     addNode(s,0);
28     while (!q.empty()) {
29         pii t = q.top(); q.pop();
30         int u = t.second, d = -t.first;
31         if (vis[u]) continue;
32         vis[u] = true;
33         forall(it, adj[u]) {
34             int v = it->first, cost = it->second;
35             addNode(v, d + cost);
36         }
37     }
38 }

```

Igual al algoritmo de arriba, solo que usando *heap* para correr en $\mathcal{O}(V \log V + E)$. [REQUIERE: *heap*, *SWAP(a,b)*]

```

1 #define MAXN 100000
2 #define INF -1
3
4 struct edge {
5     int dest, w;
6 };
7
8 struct node {
9     int inh, d, prev;
10    vector<edge> c;
11 } n[MAXN];
12
13 #define SWAPH(a,b) ( SWAP(n[A[(a)]] . inh , n[A[(b)]] . inh) , SWAP(A[(a)
14    ], A[(b)]] )
15
16 bool compare(int a, int b) {return n[a].d < n[b].d;}
17
18 int AA[MAXN];
19
20 void dijkstra(int S, int D, int N) {
21     int i, cur, dest, it, *A;
22     A = AA;

```

```

23     for (i=0; i<N; i++) {
24         n[i].d = INF;
25         n[i].inh = -1;
26         n[i].prev = -1;
27     }
28     n[S].d = 0;
29
30     A[0] = 1; A[1] = S; n[S].inh = 1;
31     while (A[0] > 0) {
32         cur = pop(A, compare);
33         n[cur].inh = -1;
34         if (cur == D) break;
35         for (it=0; it<n[cur].c.size(); it++) {
36             dest = n[cur].c[it].dest;
37             if (n[dest].d==INF || n[dest].d>n[cur].d+n[cur].c[it].w){
38                 n[dest].d = n[cur].d + n[cur].c[it].w;
39                 n[dest].prev = cur;
40                 if (n[dest].inh == -1) {
41                     A[A[0]+1] = dest;
42                     n[dest].inh = A[0]+1;
43                     insert(A, compare);
44                 } else update(A, n[dest].inh, compare);
45             }
46         }
47     }
48 }

```

4.4. Bellman-Ford

```

1 void bellman_ford(int V, edge e[], int E) {
2     for (int i=0; i<V-1; i++) {
3         for (int j=0; j<E; j++) {
4             if (n[e[j].s].d != INF && n[e[j].e].d > n[e[j].s].d + e[j].w)
5                 {
6                     n[e[j].e].d = n[e[j].s].d + e[j].w;
7                 }
8         }
9     }

```

4.5. Floyd - Warshall

Camino minimo de todos a todos en $\mathcal{O}(n^3)$.

```

1| forn(k,n) forn(i,n) forn(j,n) A[i][j]<?=A[i][k]+A[k][j];

```

4.6. Edmonds-Karp: flujo máximo en $\mathcal{O}((N + E)F)$ ó $\mathcal{O}(NE^2)$

Calcula el flujo máximo entre el nodo *SOURCE* y el nodo *SINK* de un grafo no dirigido en el que la capacidad de la arista (i, j) se encuentra en $A[i][j]$. Los nodos deben estar en el array $n[N]$, y deben tener los parámetros **int** *best* y **int** *prev* además de un vector de adyacencias *vector* $< \mathbf{int} > c$. Hay que definir *INF* como un valor mayor o igual que el flujo máximo. [REQUIERE *vector*, MIN(a,b)]

```

1 int edmondskarp(int **A, int SOURCE, int SINK) {
2     int s[MAXN], S, E, F[MAXN][MAXN], FLOW, tmp, it;
3     bool v[MAXN];
4
5     memset(F, 0, sizeof(F));
6     FLOW = 0;
7     while (1) {
8         memset(v, false, sizeof(v));
9         n[SOURCE].best = INF;
10        n[SOURCE].prev = -1;
11        s[0] = SOURCE; S = 0; E = 1; v[SOURCE] = true;
12        while (S != E && s[S] != SINK) {
13            for (it=0; it<n[s[S]].c.size(); it++) {
14                tmp = n[s[S]].c[it];
15                if (v[tmp]==false && A[s[S]][tmp]-F[s[S]][tmp]>0) {
16                    v[tmp] = true;
17                    n[tmp].prev = s[S];
18                    n[tmp].best = MIN(n[s[S]].best, A[s[S]][tmp] - F[s[S]
19                        ][tmp]);
20                    s[E] = tmp;
21                    E++;
22                }
23            }
24            S++;
25        }
26        if (S==E) return FLOW;
27        else {
28            FLOW += n[SINK].best;
29            tmp = SINK;
30            while (n[tmp].prev != -1) {
31                F[n[tmp].prev][tmp] += n[SINK].best;
32                F[tmp][n[tmp].prev] = - F[n[tmp].prev][tmp];
33                tmp = n[tmp].prev;
34            }
35        }
36    }
}

```

4.7. Relabel to front: flujo máximo en $\mathcal{O}(N^3)$

```

1 usa: algorithm, list, for
2 #define MAXN 200
3 typedef list<int> lint;
4 typedef lint::iterator lintIt;
5 //usadas para el flujo
6 tint f[MAXN][MAXN]; //flujo
7 tint e[MAXN]; //exceso
8 tint h[MAXN]; //altura
9 lintIt cur[MAXN];
10 //esto representa el grafo que hay que armar
11 lint ady[MAXN]; //lista de adyacencias (para los dos lados)
12 tint c[MAXN][MAXN]; //capacidad (para los dos lados)
13 tint n; //cant de nodos
14 tint cf(tint i, tint j) { return c[i][j] - f[i][j]; }
15 void push(tint i, tint j) {
16     tint p = min(e[i], cf(i,j));
17     f[j][i] = -(f[i][j] += p);
18     e[i] -= p;
19     e[j] += p;
20 }
21 void lift(tint i) {
22     tint hMin = n*n;
23     for(lintIt it = ady[i].begin(); it != ady[i].end(); ++it) {
24         if (cf(i, *it) > 0) hMin = min(hMin, h[*it]);
25     }
26     h[i] = hMin + 1;
27 }
28 void iniF(tint desde)
29 {
30     for(i,n) {
31         h[i] = e[i] = 0;
32         for(j,n) f[i][j] = 0;
33         cur[i] = ady[i].begin();
34     }
35     h[desde] = n;
36     for(lintIt it = ady[desde].begin(); it != ady[desde].end(); ++it)
37     {
38         f[*it][desde] = -(f[desde][*it] = e[*it] = c[desde][*it]);
39     }
40 }
41 void disch(tint i) {
42     while(e[i] > 0) {
43         lintIt& it = cur[i];
44         if (it == ady[i].end()) { lift(i); it = ady[i].begin(); }
45         else if (cf(i,*it) > 0 && h[i] == h[*it] + 1) push(i,*it);
46         else ++it;
47     }
48 }
49 tint calcF(tint desde, tint hasta) {
50     iniF(desde);

```

```

51 lint l;
52 forn(i,n) {if (i != desde && i != hasta) l.push_back(i);}
53 for(lintIt it = l.begin() ; it != l.end() ; ++it) {
54   tint antH = h[*it];
55   disch(*it);
56   if (h[*it] > antH) { //move to front
57     l.push_front(*it);
58     l.erase(it);
59     it = l.begin();
60   }
61 } return e[hasta];
62 }
63 void addEje(tint a, tint b, tint ca) {
64   //requiere reiniciar las capacidades
65   if (c[a][b] == 0) {//soporta muchos ejes mismo par de nodos
66     ady[a].push_back(b);
67     ady[b].push_back(a);
68   }
69   c[b][a] = c[a][b] += ca;
70 }
71 void iniGrafo(tint nn) { //requiere n ya leído
72   n=nn;
73   forn(i,n) {
74     forn(j,n) c[i][j] = 0;
75   } //solo si se usa la version de addeje con soporte multieje
76   ady[i].clear();
77 }
78 }

```

4.8. Dinic: Flujo maximo en $\mathcal{O}(E.V^2)$

Generalmente es mas rapido que preflow y edmonds-karp; para redes con capacidades 1 corre en $\mathcal{O}(E\sqrt{V})$ Usa tint, MAXN, vi, forn, si, pb, queue, vector, algorithm

```

1 /* Dinic
2 Flujo maximo en  $\mathcal{O}(E.V^2)$ ; generalmente es mas rapido que preflow y
   edmonds-karp;
3 para redes con capacidades 1 corre en  $\mathcal{O}(E.\sqrt{V})$ 
4 Usa tint, MAXN, vi, forn, si, pb, queue, vector, algorithm
5 */
6
7 typedef tint tipo;
8 const int INF = 1<<29;
9 struct edge {
10   tipo c,f;
11   tipo r() { return c-f; }
12 };
13 int N,S,T;
14 edge red[MAXN][MAXN];
15 vi adjG[MAXN];
16
17 void reset() {

```

```

18   forn(u,N) forn(i, si(adjG[u])) {
19     int v = adjG[u][i];
20     red[u][v].f = 0;
21   }
22 }
23 void initGraph(int n, int s, int t) {
24   N = n; S = s; T = t;
25   forn(u,N) {
26     adjG[u].clear();
27     forn(v,N) red[u][v] = (edge){0,0};
28   }
29 }
30 void addEdge(int u, int v, int c) {
31   if (!red[u][v].c && !red[v][u].c) { adjG[u].pb(v); adjG[v].pb(u)
32     ; }
33   red[u][v].c += c;
34 }
35 int dist[MAXN];
36 bool dinic_bfs() {
37   forn(u,N) dist[u] = INF;
38   queue<int> q; q.push(S); dist[S] = 0;
39   while (!q.empty()) {
40     int u = q.front(); q.pop();
41     forn(i, si(adjG[u])) {
42       int v = adjG[u][i];
43       if (dist[v] < INF || red[u][v].r() == 0) continue;
44       dist[v] = dist[u] + 1;
45       q.push(v);
46     }
47   }
48   return dist[T] < INF;
49 }
50 tipo dinic_dfs(int u, tipo cap) {
51   if (u == T) return cap;
52   tipo res = 0;
53   forn(i, si(adjG[u])) {
54     int v = adjG[u][i];
55     if (red[u][v].r() && dist[v] == dist[u] + 1) {
56       tipo send = dinic_dfs(v, min(cap, red[u][v].r()));
57       red[u][v].f += send; red[v][u].f -= send;
58       res += send; cap -= send;
59       if (cap == 0) break;
60     }
61   }
62   if (res == 0) dist[u] = INF;
63   return res;
64 }
65 tipo dinic() {
66   tipo res = 0; while (dinic_bfs()) res += dinic_dfs(S,INF);
67   return res;
68 }

```

4.9. Karger: min-cut randomizado

Encuentra con alta probabilidad un min-cut del grafo con nodos $[0, \dots, N]$ y aristas $e[0, \dots, E]$. Debe definirse MAX como el máximo número de iteraciones del algoritmo básico, y entonces el algoritmo es $\mathcal{O}((N + E) * MAX)$.
[REQUIERE: *cstdlib*, *ctime*]

```

1 int f[MAXN];
2
3 struct edge {
4     int a, b, w;
5 } e[MAXE];
6
7 int getf(int n) {
8     if (f[n] != n) f[n] = getf(f[n]);
9     return f[n];
10 }
11
12 int mincut(int N, int E) {
13     int i, j, tmp, tmp1, tmp2, RES, EE, NN;
14     RES = INF;
15     for (i=0; i<MAX; i++) {
16         for (j=0; j<N; j++) f[j] = j;
17
18         EE = E; NN = N;
19         while (EE > 0 && NN > 2) {
20             tmp = rand() % EE;
21             tmp1 = getf(e[tmp].a);
22             tmp2 = getf(e[tmp].b);
23             if (tmp1 != tmp2) {
24                 f[tmp1] = tmp2;
25                 NN--;
26             }
27             swap(e[tmp], e[EE-1]);
28             EE--;
29         }
30
31         tmp = 0;
32         for (j=0; j<EE; j++) if (getf(e[j].a) != getf(e[j].b)) tmp +=
33             e[j].w;
34         if (tmp < RES) RES = tmp;
35     }
36     return RES;
37 }
```

4.10. Flujo de costo mínimo en $\mathcal{O}(BN^3)$

Cálculo del flujo de costo mínimo en un grafo. No es seguro que ande si están presentes las aristas (i, j) y (j, i) con $i \neq j$. Corre en $\mathcal{O}(BN^3)$ donde B es

una cota para las capacidades. NO FUNCIONA si hay ciclos de costo negativo.
[REQUIERE: *algorithm*, *form*]

```

1 #define forn(i,n) for (int i=0; i<(n); ++i)
2 #define MAXN 120
3 const int INF = 1<<30;
4 struct Eje {
5     int f,m,p,rp; //f flujo, m capacidad, p costo
6     int d() {return m-f;}
7 };
8
9 Eje red[MAXN][MAXN];
10 int adyc[MAXN], ady[MAXN][MAXN];
11 int N,F,D;
12 void iniG(int n, int f, int d) {
13     N=n; F=f; D=d;
14     fill(red[0], red[N], (Eje){0,0,0,0});
15     fill(ady, ady+N,0);
16 }
17 void aEje(int d, int h, int m, int p) {
18     red[h][d].p = -(red[d][h].p = p);
19     red[h][d].rp = -(red[d][h].rp = p); // reduced cost
20     red[d][h].m = m;
21     ady[d][adyc[d]++] = h; ady[h][adyc[h]++] = d;
22 }
23 int md[MAXN], vd[MAXN], used[MAXN];
24 void reduceCost() {
25     forn(i,N) forn(j,N) if (red[i][j].d()>0) {
26         red[i][j].rp += md[i]-md[j];
27         red[j][i].rp = 0;
28     }
29 }
30 int camAu(int &v) {
31     fill(vd,vd+N,-1);
32     fill(used,used+N,false);
33     vd[F]=F; md[F]=0;
34     int i = F, next = F, cant = 0;
35     while (cant<N) {
36         used[i]=true;
37         forn(jj,adyc[i]) {
38             int j=ady[i][jj], nd=md[i]+red[i][j].rp;
39             if (red[i][j].d()>0) if (vd[j]==-1 || md[j]>nd)
40                 md[j]=nd,vd[j]=i;
41         }
42         int mn=INF;
43         forn(k,N) if (!used[k] && vd[k]!= -1 && md[k] < mn)
44             next=k,mn=md[k];
45         cant++;
46         if (next==i) break;
47         i=next;
48     }
49     v=0;
50     if (vd[D]==-1) return 0;
51     reduceCost();
52 }
```



```

52     int f=red[vd[D]][D].d();
53     for (int n=D;n!=F;n=vd[n]) f <?= red[vd[n]][n].d();
54     for (int n=D;n!=F;n=vd[n]) {
55         red[n][vd[n]].f = -(red[vd[n]][n].f+f);
56         v += red[vd[n]][n].p * f;
57     }
58     return f;
59 }
60
61 int flujo(int &r) {
62     fill(vd,vd+N,-1);
63     vd[F]=F; md[F]=0;
64     bool cambios = true;
65     for (int rep=0; rep<N && cambios; rep++) {
66         cambios = false;
67         forn(i,N) if (vd[i]!=-1) forn(jj,adyc[i]) {
68             int j = ady[i][jj], nd = md[i]+red[i][j].rp;
69             if (red[i][j].d()>0) if (vd[j]==-1||md[j]>nd)
70                 md[j]=nd,vd[j]=i,cambios=true;
71         }
72     }
73     reduceCost();
74     r=0; int v,f=0, c; // r = minCost, f = maxFlow
75     while ((c=camAu(v)) r+=v, f+=c;
76     return f;
77 }

```

4.11. Maximum bipartite matching

[REQUIERE: *vector*, *for*(*i*,*n*), *decl*(*v*,*c*), *forall*(*i*,*c*), *pb*]

```

1 #define MAXN 102
2 typedef vector<int> vi;
3
4 vi next[MAXN];
5 int M,pre[MAXN];
6 bool vis[MAXN];
7
8 void initGraph(int M, int N) {
9     :M = M;
10    forn(i,M) next[i].clear();
11    forn(i,N) pre[i] = -1;
12 }
13
14 void addEdge(int u, int v) {
15     next[u].pb(v);
16 }
17
18 bool augment(int u) {
19     if (u == -1) return true;
20     if (vis[u]) return false;
21     vis[u] = true;

```

```

22     forall(it,next[u]) {
23         int v = *it;
24         if (augment(pre[v])) {
25             pre[v] = u;
26             return true;
27         }
28     }
29     return false;
30 }
31
32 int matching() {
33     int res = 0;
34     forn(u,M) {
35         forn(v,M) vis[v] = false;
36         res += augment(u);
37     }
38     return res;
39 }

```

4.12. Puntos de articulación

```

1 const int INF = 1<<29;
2 const int MAXN = 100000 + 100;
3 const int MAXE = MAXN;
4
5 int m,n;
6 vector<pii> edges;
7 vi adj[MAXN];
8
9 void addE(int u, int v) {
10     int ne = si(edges);
11     edges.pb(mp(u,v));
12     adj[u].pb(ne); adj[v].pb(ne);
13 }
14
15 stack<int> estack; //stack de edges
16 int d[MAXN], b[MAXN], t;
17 int bc[MAXE], nbc;
18
19 struct param {
20     int le, u, v, i, ne;
21     pii e;
22 };
23
24 #define save(x) prm.top().x = x
25 #define load(x) x = prm.top().x
26
27 void NR_dfs(int le, int u) {
28     stack<param> prm;
29     param init; init.le = le; init.u = u;
30     prm.push(init);

```

```

31
32 int i,v,ne; pii e;
33 for (;) {
34     restart;
35     le = prm.top().le; u = prm.top().u;
36
37     b[u] = d[u] = t++;
38     for (i = 0; i < si(adj[u]); i++) {
39         ne = adj[u][i];
40         if (ne == le) continue;
41         e = edges[ne];
42         v = e.first ^ e.second ^ u;
43         if (d[v] == -1) {
44             estack.push(ne);
45
46             save(le); save(u); save(i); save(v); save(e); save(ne);
47             init.le = ne; init.u = v; prm.push(init);
48             goto restart;
49             ret;
50             load(le); load(u); load(i); load(v); load(e); load(ne);
51
52             //if (b[v] > d[u]) ne es bridge
53             if (b[v] >= d[u]) { //u es punto de articulacion
54                 int last;
55                 do{ //saca edges de la componente biconexa
56                     last = estack.top();
57                     bc[last] = nbc;
58                     estack.pop();
59                 } while (last != ne);
60                 nbc++;
61             }
62             else b[u] = min(b[u], b[v]);
63         }
64         else if (d[v] < d[u]) {
65             estack.push(ne);
66             b[u] = min(b[u], d[v]);
67         }
68     }
69     prm.pop();
70     if (prm.empty()) return;
71     else goto ret;
72 }
73 }

```

4.13. Circuito euleriano

[REQUIERE: *algorithm*, *vector*, *list*, *string*, *forn*]

```

1 typedef string ejeVal;
2 #define MNT "" //MENORATODOS
3 typedef pair<ejeVal, tint> eje;
4 tint n; vector<eje> ady[MAXN]; tint g[MAXN];
5 //grafo (inG = in grado o grado si es no dir)

```

```

6 tint aux[MAXN];
7 tint pinta(tint f) {
8     if (aux[f]) return 0;
9     tint r = 1; aux[f] = 1;
10    forn(i, ady[f].size()) r += pinta(ady[f][i].second);
11    return r;
12 }
13
14 tint compCon(){
15     fill(aux, aux+n, 0);
16     tint r=0;
17     forn(i,n){
18         if (!aux[i]){ r++; pinta(r); }
19     }
20     return r;
21 }
22
23 bool isEuler(bool path, bool dir) {
24     if (compCon() > 1) return false;
25     tint c = (path ? 2 : 0);
26     forn(i,n){
27         if (!dir ? ady[i].size() % 2 : g[i] != 0) {
28             if (dir && abs(g[i]) > 1) return false;
29             c--;
30             if (c < 0) return false;
31         }
32     }
33     return true;
34 }
35
36 bool findCycle(tint f, tint t, list<tint>& r) {
37     if (aux[f] >= ady[f].size()) return false;
38     tint va = ady[f][aux[f]++].second;
39     r.push_back(va);
40     return (va != t ? findCycle(va, t, r) : true);
41 }
42
43 list<tint> findEuler(bool path){//dir., sin valores rep.
44     if (!isEuler(path, true)) return list<tint>();
45     bool agrego = false;
46     if (path) {
47         tint i = max_element(g, g + n) - g;
48         tint j = min_element(g, g + n) - g;
49         if (g[i] != 0) {ady[i].push_back(eje(MNT, j)); agrego=1;}
50     }
51     tint x = -1;
52     forn(i,n) {
53         sort(ady[i].begin(), ady[i].end());
54         if (x < 0 || ady[i][0] < ady[x][0]) x=i;
55     }
56     fill(aux, aux+n, 0);
57     list<tint> r;
58     findCycle(x,x,r);

```

```

59 if (!agrego) r.push_front(r.back());
60 list<tint> aux; bool find=false;
61 list<tint>::iterator it = r.end();
62 do{
63     if (!find) —it;
64     for (find=findCycle(*it,*it,aux);!aux.empty();aux.pop_front()){
65         it = r.insert(++it, aux.front());
66     }
67 } while (it != r.begin());
68 return r;
69 }

```

4.14. Componentes fuertemente conexas

Busca las CFC en un grafo dirigido. Usa: vector, cstring, forn, MAXN. N = cant de vertices, ady/adyt la lista de adyacencias/lista traspuesta. Devuelve la cant de CFC's, y cc[] termina con la CFC de cada uno.

```

1 int vis[MAXN], cc[MAXN], CC,N, o[MAXN];
2 vector<int> ady[MAXN], adyt[MAXN];
3 // N = numero de vertices, ady lista de adyacencias, adyt la
   traspuesta
4 void dfs1(int i) {
5     vis[i] = 2;
6     forn(j, si(ady[i])) if (vis[ady[i][j]] == 0)
7         dfs1(ady[i][j]);
8     o[CC++] = i;
9 }
10 void dfs2(int i) {
11     vis[i] = 2;
12     cc[i] = CC;
13     forn(j, si(adyt[i])) if (vis[adyt[i][j]] == 0)
14         dfs2(adyt[i][j]);
15 }
16 int cfc() {
17     CC = 0; memset(vis,0,sizeof(vis));
18     forn(i,N) if (vis[i] == 0) dfs1(i);
19     CC = 0; memset(vis,0,sizeof(vis));
20     for(int i = N-1; i>=0; i--) if (vis[o[i]] == 0) {dfs2(o[i]); CC
        ++;}
21     return CC;
22 }

```

4.15. Lowest Common Ancestor

Funciones para trabajar con los *lowest common ancestors* de los pares de nodos de un árbol. El preprocesamiento es en $\mathcal{O}(N \log N)$, y $lca(a, b)$ opera en

$\mathcal{O}(\log N)$. $dist(a, b)$ calcula la distancia entre los nodos a y b , y también corre en $\mathcal{O}(\log N)$.

```

1 #define MAXN 16384
2 #define LOGMAXN 14
3
4 int N, p[MAXN][LOGMAXN];
5
6 struct node {
7     int l;
8     vector<int> c;
9 } n[MAXN];
10
11 void dfs(int cur) {
12     for (int i=0; i<(int)n[cur].c.size(); i++) {
13         if (n[cur].c[i] != p[cur][0]) {
14             p[n[cur].c[i]][0] = cur;
15             n[n[cur].c[i]].l = n[cur].l+1;
16             dfs(n[cur].c[i]);
17         }
18     }
19 }
20
21 void init_lca() {
22     n[0].l = 0; p[0][0] = 0;
23     dfs(0);
24
25     for (int i=1; i<LOGMAXN; i++) {
26         for (int j=0; j<N; j++) {
27             p[j][i] = p[p[j][i-1]][i-1];
28         }
29     }
30 }
31
32 int lca(int a, int b) {
33     if (n[a].l < n[b].l) return lca(b, a);
34     else {
35         int i, dl = n[a].l-n[b].l;
36
37         for (i=0; i<LOGMAXN; i++) {
38             if (((dl>>i)&1) == 1) {
39                 a = p[a][i];
40             }
41         }
42
43         if (a == b) return a;
44         else {
45             for (i=LOGMAXN-1; i>=0; i--) {
46                 if (p[a][i] != p[b][i]) {
47                     a = p[a][i];
48                     b = p[b][i];
49                 }
50             }
51             return p[a][0];

```

```

52     }
53 }
54 }
55
56 int dist(int a, int b) {
57     return n[a].l + n[b].l - 2*n[lca(a, b)].l;
58 }

```

4.16. Extras y adaptaciones

■ Bellman Ford

Para detectar ciclos negativos se usa

```

1 bool neg_cycle(int V, edge e, int E;) {
2     bellman_ford(V, e, E);
3     for (int i=0; i<E; i++) {
4         if (n[e[j].s].d != INF && n[e[j].e].d > n[e[j].s].d + e[j]
5             ].w) return 1;
6     }
7     return 0;
8 }

```

Para encontrar la mínima distancia de todos los nodos de un grafo a un único destino, se usa el algoritmo de Bellman-Ford con las aristas invertidas.

■ Grafos bipartitos

* A graph is bipartite if and only if it does not contain an odd cycle. Therefore, a bipartite graph cannot contain a clique of size 3 or more.

* A graph is bipartite if and only if it is 2-colorable.

* The size of the minimum vertex cover is equal to the size of the maximum matching (König's theorem).

* The size of the maximum independent set plus the size of the maximum matching is equal to the number of vertices.

* For a connected bipartite graph the size of the minimum edge cover is equal to the size of the maximum independent set.

* For a connected bipartite graph the size of the minimum edge cover plus the size of the minimum vertex cover is equal to the number of vertices.

* The spectrum of a graph is symmetric if and only if it's a bipartite graph.

■ Aplicaciones de flujos

* Dado un *DAG*, se puede encontrar la mínima cantidad de caminos (disjuntos?) que cubran a todos los vértices (problemas de los taxis, cajas, mamushkas, CodeJam). Para ello, se construye el grafo bipartito (A, B) con todas las aristas del grafo original, donde A contiene a los nodos que tengan $out - deg > 0$, y B contiene a los nodos que tengan $in - deg > 0$. La solución es $n - m$, donde n es el número de nodos del grafo y m es el maximum matching del grafo bipartito construido.

* Given a directed graph $G = (V, E)$ and two vertices s and t , we are to find the maximum number of independent paths from s to t . Two paths are said to be independent if they do not have a vertex in common apart from s and t . We can construct a network $N = (V, E)$ from G with vertex capacities, where: s and t are the source and the sink of N respectively; $c(v) = 1$ for each v in V ; $c(e) = 8$ for each e in E (vale ponerle capacidad 1 a la arisa, se usa a lo sumo una vez). Then the value of the maximum flow is equal to the maximum number of independent paths from s to t .

* Given a directed graph $G = (V, E)$ and two vertices s and t , we are to find the maximum number of edge-disjoint paths from s to t . This problem can be transformed to a maximum flow problem by constructing a network $N = (V, E)$ from G with s and t being the source and the sink of N respectively and assign each edge with unit capacity.

■ Otros

* A matrix A is an adjacency matrix of a *DAG* if and only if the eigenvalues of the $(0, 1)$ matrix $A + I$ are positive, where I denotes the identity matrix.

* Every tournament (grafo proveniente de asignarle direcciones a cada una de las aristas de un grafo no dirigido completo, o sea un grafo dirigido tal que para todo u y v entonces o bien $(u, v) \in E$ o bien $(v, u) \in E$) has an odd number of Hamiltonian paths (se demuestra facil por inducción).

* Teorema de Euler: En todo dibujo de un grafo planar se satisface $v - e + f = 2$, donde v es el número de vértices, e es el número de aristas, y f es el número de regiones en que se divide el plano.

* En un grafo conexo y planar existe un nodo que tiene grado a lo sumo 5.

5.1. Knuth-Morris-Pratt

[REQUIERE: *vector*, *string*, *algorithm*, *sz(a)*, *pb*, *all(c)*, *tr(c,i)*, *present(c,x)*, *cpresent(c,x)*, *forn(i,n)*, *forsn(i,m,n)*]

```

1 #define MAXN 1000
2 typedef vector<int> vi;
3 typedef vector<vi> vvi;
4 typedef pair<int,int> ii;
5
6 int F[MAXN];
7
8 int preproc(string pattern){
9     F[0] = F[1] = 0;
10    forsn(i, 2, sz(pattern)+1){
11        int j = F[i-1];
12        while(pattern[j] != pattern[i-1] && j>0) j = F[j];
13        F[i] = j + ((pattern[j] == pattern[i-1])?1:0);
14    }
15    return 0;
16 }
17
18 int KMP(string pat, string str){
19     preproc(pat);
20     int m = sz(pat);
21     for(int i=0, j = 0; j<sz(str); j++){
22         cout << j << ' ' << i;
23         if(str[j] == pat[i]){ if(++i == m) return j-m+1; }
24         else{
25             if(i>0) j--;
26             i = F[i];
27         }
28     }
29     return -1;
30 }
31
32 int KMP2(string pat, string str){
33     preproc(pat);
34     int m = sz(pat);
35     for(int i=0, j = 0; j<sz(str); j++){
36         while(i>0 && pat[i]!=str[j]) i = F[i];
37         if(str[j] == pat[i]){ if(++i == m) return j-m+1; }
38     }
39     return -1;
40 }
41
42 int main(){
43     string pattern, eaea;
44     cout << "escriba patron : ";
45     cin >> pattern;
46     cout << "escriba cadena : ";
47     cin >> eaea;
48     cout << KMP2(pattern, eaea) << endl;;

```

```

49     forn(i, sz(pattern)+1) cout << F[i] << ' '; cout << endl;
50
51     return 0;
52 }

```

5.2. Aho-Corasik

Busca todas las apariciones de ciertos patrones como subcadenas de cierta otra cadena. Los patrones se guardan en *pat*, y *match* devuelve un *vvi* tal que el vector *i*-ésimo guarda una lista de los índices en donde el patrón *i* aparece dentro de la cadena mayor. Las cadenas en *pat* no pueden repetirse.

```

1 struct node {
2     int id; char c; bool isfinal;
3     node *parent, *pre, *pfinal;
4     map<char,node*> child;
5
6     node(node* parent = NULL, char c = ' ') {
7         this->c = c; this->parent = parent;
8         pre = pfinal = NULL;
9         id = -1; isfinal = false;
10    }
11    void insert(const string& s, int id = -1) {
12        int n = s.size();
13        node* act = this;
14        forn(i,n)
15            if (act->child.count(s[i])) act = act->child[s[i]];
16            else act = act->child[s[i]] = new node(act,s[i]);
17        act->isfinal = true; act->id = id;
18    }
19    void clear() {
20        forall(it,child) it->second->clear();
21        child.clear(); c = ' '; parent = pre = pfinal = NULL;
22    }
23 };
24
25 vs pat;
26 node *root;
27
28 void precompute() {
29     root = new node();
30     int np = sz(pat);
31     forn(i,np) root->insert(pat[i], i);
32
33     queue<node*> q; forall(it,root->child) q.push(it->second);
34     while (!q.empty()) {
35         node *u = q.front(); q.pop();
36         node *p = u->parent->pre; char c = u->c;
37         while (p && !p->child.count(c)) p = p->pre;
38         if (p == NULL) u->pre = root;
39         else u->pre = p->child[c];

```

```
40
41     if (u->pre->isfinal) u->pfinal = u->pre;
42     else u->pfinal = u->pre->pfinal;
43     forall(it,u->child) q.push(it->second);
44 }
45 }
46
47 vvi match(const string& s) {
48     int n = si(s), np = si(pat);
49     vvi res(np);
50     node* act = root;
51     forn(i,n) {
52         while (act && !act->child.count(s[i])) act = act->pre;
53         if (act == NULL) act = root;
54         else {
55             act = act->child[s[i]];
56             node *fin = act;
57             while (fin && fin->isfinal) {
58                 int id = fin->id;
59                 res[id].pb(i-si(pat[id])+1);
60                 fin = fin->pfinal;
61             }
62         }
63     }
64     return res;
65 }
66
67 //Para limpiar el Trie
68 if(root != NULL)(*root).clear();
```

6.1. Área de un polígono

Cálculo del área del polígono $p[1, \dots, N-1]$ en $\mathcal{O}(N)$. Los segmentos del polígono tienen extremos $p[i]$ y $p[i+1]$ para $i=0, \dots, N-2$, y el polígono se cierra con el segmento $p[N-1]$ a $p[0]$. [REQUIERE: ABS(n)]

```
1 double polygon_area(point p[], int N) {
2     double A = 0;
3     for (int i=1; i<N-1; i++) {
4         A += (p[i].x - p[0].x)*(p[i+1].y - p[0].y) - (p[i+1].x - p
5             [0].x)*(p[i].y - p[0].y);
6     }
7     return ABS(A/2);
}
```

6.2. Número de enrollamiento

Cálculo del número de enrollamiento de un punto $P0$ respecto del polígono $p[0, \dots, N]$, donde los segmentos $p[i]$ a $p[i+1]$ para $i=0, \dots, N-1$ forman el polígono, y $p[0] = p[N]$. Devuelve

```
1 int winding_number(point p[], int N, point P0) {
2     int i, WN;
3
4     for (i=0; i<N; i++) {
5         if ( (p[i].x - P0.x)*(p[i+1].y - P0.y) - (p[i].y - P0.y)*(p[i
6             +1].x - P0.x) == 0 ) {
7             if (P0.x >= MIN(p[i].x, p[i+1].x) && P0.x <= MAX(p[i].x, p[i
8                 +1].x) && P0.y >= MIN(p[i].y, p[i+1].y) && P0.y <= MAX(
9                     p[i].y, p[i+1].y)) {
10                 return 1; //EN EL BORDE
11             }
12         }
13     }
14
15     WN = 0;
16     for (i=0; i<N; i++) {
17         if (p[i].y <= P0.y) {
18             if (p[i+1].y > P0.y) {
19                 if ( (p[i].x - P0.x)*(p[i+1].y - P0.y) - (p[i].y - P0.y)*(p[i
20                     +1].x - P0.x) > 0 ) WN++;
21             }
22         } else {
23             if (p[i+1].y <= P0.y) {
24                 if ( (p[i].x - P0.x)*(p[i+1].y - P0.y) - (p[i].y - P0.y)*(p[i
25                     +1].x - P0.x) < 0 ) WN--;
26             }
27         }
28     }
29     return WN;
30 }
```

6.3. Cápsula convexa

Calculo de la cápsula convexa de un conjunto de puntos. [REQUIERE: *vector*, *algorithm*]

```
1 struct pto { tint x,y; } r;
2 tint dist2(pto a, pto b)
3     { return (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y); }
4 tint crossprod(tint x1, tint y1, tint x2, tint y2)
5     { return (x1*y2-x2*y1); }
6 tint area(pto a, pto b, pto c) //area con signo
7     { return crossprod(b.x-a.x, b.y-a.y, c.x-a.x, c.y-a.y); }
8 bool comp(const pto &a, const pto &b) //lowerleft
9     { return (a.y==b.y) ? (a.x<b.x) : (a.y<b.y); }
10 bool operator<(const pto &a, const pto &b) {
11     tint tmp = area(r, a, b);
12     if (tmp == 0) return (dist2(a, r) < dist2(b, r));
13     else return (tmp > 0);
14 } //counterclockwise (para clockwise poner tmp < 0)
15 vector<pto> convex_hull(vector<pto> puntos) {
16     if (puntos.size() < 3) return puntos;
17     r = * (min_element(puntos.begin(), puntos.end(), comp));
18     sort(puntos.begin(), puntos.end()); //usa operator <
19     int i = 0;
20     vector<pto> ch;
21     ch.push_back(puntos[i++]);
22     ch.push_back(puntos[i++]);
23     while (i < puntos.size()) {
24         if (ch.size() > 1 && area(ch[ch.size()-2], ch[ch.size()-1],
25             puntos[i]) < 0) ch.pop_back();
26         else ch.push_back(puntos[i++]);
27     }
28     return ch;
29 } //en 24 va <= si NO se quieren incluir puntos alineados
```

6.4. Par de puntos más cercanos

run_closest_pair(P) encuentra el par de puntos más cercanos de $p[0, \dots, P-1]$ en $\mathcal{O}(N \log N)$. Puede llegar a ser lento si hay muchos puntos superpuestos. [REQUIERE: *cmath*, *stdlib*, *algorithm*]

```
1 #define SQ(x) ( (x)*(x) )
2 #define MAXP 131072
3 #define INF 1E9
4
5 typedef double tint;
6
7 struct pt {
8     tint x, y;
9 } p[MAXP], s[MAXP];
10
```

```

11 int cmpx(const void *a, const void *b) {
12     pt pa=(pt*)a, pb=(pt*)b;
13     if (pa.x < pb.x) return -1;
14     else if (pa.x > pb.x) return 1;
15     else if (pa.y < pb.y) return -1;
16     else if (pa.y > pb.y) return 1;
17     else return 0;
18 }
19
20 double dist(const pt &a, const pt &b) {
21     return sqrt(SQ(a.x-b.x) + SQ(a.y-b.y));
22 }
23
24 double closest_pair(int S, int E) {
25     if (E-S == 1) return INF;
26     else {
27         int i, j, k, l, r, cur, mid = (S+E)/2;
28         double tmp = min(closest_pair(S, mid), closest_pair(mid, E));
29
30         i = S; j = mid; l = r = 0;
31         while (i < mid || j < E) {
32             if (i < mid && abs(p[i].x - p[mid].x) > tmp) i++;
33             else if (j < E && abs(p[j].x - p[mid].x) > tmp) j++;
34             else {
35                 if (i == mid || (j < E && p[i].y > p[j].y)) cur = j++;
36                 else cur = i++;
37
38                 while (l < r && p[cur].y - s[l].y > 2.0*tmp) l++;
39                 for (k=l; k<r; k++) tmp = min(tmp, dist(p[cur], s[k]));
40                 s[r++] = p[cur];
41             }
42         }
43
44         i = k = S; j = mid;
45         while (i < mid || j < E) {
46             if (i == mid || (j < E && p[i].y > p[j].y)) s[k++] = p[j]
47                 ++;
48             else s[k++] = p[i++];
49         }
50         for (i=S; i<E; i++) p[i] = s[i];
51         return tmp;
52     }
53 }
54
55 double run_closest_pair(int P) {
56     qsort(p, P, sizeof(pt), cmpx);
57     return closest_pair(0, P);
58 }

```

6.5. Terna de puntos más cercanos

Código para encontrar la terna de puntos que minimicen el perímetro del triángulo formado por ellos. Se puede modificar para encontrar el par de puntos más cercanos.

```

1 #define MAXN 3002
2 const double INF = 1e20;
3
4 int n;
5 vector<pdd> p;
6
7 inline pdd inv(pdd& p) { return mp(p.second, p.first); }
8
9 int main() {
10     while (cin >> n && n != -1) {
11         forn(i, n) {
12             double x, y; cin >> x >> y;
13             p.pb(mp(x, y));
14         }
15         sort(all(p));
16
17         double res = INF;
18         set<pdd> sact;
19         queue<pdd> qact;
20
21         forn(i, n) {
22             while (!qact.empty() && qact.front().first < p[i].first -
23                 res/2) {
24                 sact.erase(inv(qact.front()));
25                 qact.pop();
26             }
27
28             decl(low, sact.lower_bound(mp(p[i].second - res/2, -INF)));
29             decl(hi, sact.upper_bound(mp(p[i].second + res/2, INF)));
30
31             for (decl(it, low); it != hi; it++) {
32                 decl(it2, it); it2++;
33                 for (; it2 != hi; it2++) {
34                     res <?= dist3(*it, *it2, inv(p[i]));
35                 }
36                 sact.insert(inv(p[i]));
37             }
38             cout << res << endl;
39         }
40     }

```

6.6. Librería de geometría

Ver aparte.

6.7. Teoremas y propiedades

Teorema de Pick

Dado un polígono simple con vértices en los puntos de una grilla, el área A del polígono se relaciona con el número B de puntos en el borde y el número I de puntos en el interior por medio de la expresión

$$A = I + B/2 - 1 \quad (6.1)$$

Definiciones y propiedades de las elipses

Una elipse queda definida por la ecuación

$$AX^2 + BXY + CY^2 + DX + EY = 1 \quad (6.2)$$

donde $B^2 - 4AC > 0$. La expresión anterior puede llevarse a la forma

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \quad (6.3)$$

donde si (x_0, y_0) es el centro de la elipse y (x_a, y_a) es el versor del eje x , entonces $x = x_a(x - x_0) + y_a(y - y_0)$ y $y = -y_a(x - x_0) + x_a(y - y_0)$. El área de la elipse es entonces πab , y se define su excentricidad e como

$$e = \sqrt{\frac{a^2 - b^2}{a^2}} \quad (6.4)$$

La separación entre los focos es $ae = \sqrt{a^2 - b^2}$, y la distancia de un punto de la elipse a los focos es $L = 2a$. En forma paramétrica, la elipse toma la forma

$$\begin{aligned} x(t) &= x_0 + a \cos(t) \cos(\phi) - b \sin(t) \sin(\phi) \\ y(t) &= y_0 + a \cos(t) \sin(\phi) + b \sin(t) \cos(\phi) \end{aligned}$$

donde $t \in [0, 2\pi)$ y ϕ es el ángulo que forman el eje x y el semieje mayor de la elipse.

7.1. Longest common subsequence

Cálculo de la *longest common subsequence* de dos cadenas.

```
1 #define MAXN 110
2 #define MAXM 110
3 int lcs [MAXN] [MAXM];
4 int c [2] [110];
5
6 int LCS(int N, int M){
7     int i, j;
8
9     for (i=1; i<N; i++) for (j=1; j<M; j++) lcs [i] [j]=0;
10    for (i=0; i<N; i++) lcs [i] [0] = (c [0] [i]==c [1] [0]);
11    for (i=0; i<M; i++) lcs [0] [i] = (c [0] [0]==c [1] [i]);
12    for (i = 1; i<N; i++){
13        for (j = 1; j<M; j++){
14            lcs [i] [j] >?= lcs [i -1] [j];
15            lcs [i] [j] >?= lcs [i] [j -1];
16            lcs [i] [j] >?= lcs [i -1] [j -1]+(c [0] [i]==c [1] [j]);
17        }
18    }
19    return lcs [N-1] [M-1];
20 }
```

7.2. Longest increasing subsequence

Encuentra e imprime la *longest increasing subsequence* en el array $X[MSX]$. $M[i]$ es la posición k del menor valor $X[k]$ tal que $k < i$ y existe una *increasing subsequence* de longitud i terminando en $X[k]$. $P[k]$ es la posición del predecesor de $X[k]$ en la *longest increasing subsequence* que termina en $X[k]$.

```
1 #define MAX 100010
2 tint X[MAX];
3 int P[MAX], M[MAX], ind1, ind2, i, j, k, L;
4
5 L = 0; M[0] = 0;
6 for (i=1; i<=N; i++) {
7     ind1 = 0; ind2 = L;
8     k = 0;
9     while (ind1 < ind2) { //Notar que nunca mira M[ind1]
10        j = (ind1+ind2)/2;
11        if (j == ind1) {
12            if (X[M[ind2]] < X[i]) k = ind2;
13            else k = ind1;
14            break;
15        }
16        if (X[M[j]] < X[i]) ind1 = j;
17        else ind2 = j;
18        k = ind1;
19    }
```

```
20    P[i] = M[k];
21    if (k == L || X[i] < X[M[k+1]]) {
22        M[k+1] = i;
23        L += (k==L);
24    }
25 }
26 cout << L << endl; //Longitud de la L. I. S.
27
28 cout << X[M[L]]; //Imprime la L. I. S. al revés
29 i = P[M[L]];
30 while(i != 0){ cout << ' ' << X[i]; i = P[i]; }
31 cout << endl;
```

8.1. Algoritmo húngaro

Cálculo de la asignación máxima en un grafo bipartito de n nodos en cada parte. Se maximiza el beneficio total considerando los beneficios individuales dados en la matriz $cost[n][n]$. Para minimizar el costo, se puede hacer $cost[i][j] \Rightarrow -cost[i][j]$. El algoritmo es $\mathcal{O}(n^3)$. [REQUIERE: `cstring`, `MAX(a,b)`, `MIN(a,b)`; INICIALIZAR: n , $cost[n][n]$]

El algoritmo de arriba, sólo que sirve para matrices no cuadradas. El costo se encuentra en la matriz $mat[m][n]$. [REQUIERE: `map`, `cstring`]

```

1 #define maxn 128
2 #define INF 10000000;
3 int mat[maxn][maxn];
4 int match1[maxn], match2[maxn];
5
6 int munkres(int m, int n) {
7     int s[maxn], t[maxn], p, q, ret=0, i, j, k, tmp;
8     int l1[maxn], l2[maxn];
9
10    memset(match1, -1, sizeof(match1));
11    memset(match2, -1, sizeof(match2));
12    memset(l2, 0, sizeof(l2));
13    for (i=0; i<m; i++)
14        for (j=0; j<n; j++) l1[i] = max(l1[i], mat[i][j]);
15
16    for (i=0; i<m; i++) {
17        memset(t, -1, sizeof(t));
18        for (s[ p=q=0 ]=i; p<=q && match1[i]<0; p++) {
19            k = s[p];
20            for (j=0; j<n && match1[i]<0; j++) {
21                if (mat[k][j] == l1[k]+l2[j] && t[j]<0) {
22                    s[++q] = match2[j];
23                    t[j] = k;
24                    if (s[q] < 0) {
25                        for (p=j; p>=0; j=p) {
26                            match2[j] = k = t[j]; p = match1[k];
27                            match1[k]=j;
28                        }
29                    }
30                }
31            }
32        }
33        if (match1[i] < 0) {
34            i--; tmp = INF;
35            for (k=0; k<=q; k++) {
36                for (j=0; j<n; j++) {
37                    if (t[j]<0 && (l1[s[k]] + l2[j] - mat[s[k]][j] < tmp)
38                        ) {
39                        tmp = l1[s[k]] + l2[j] - mat[s[k]][j];
40                    }
41                }
42            }
43        }
44    }
45    }
46    }
47    }
48    }

```

```

42         for (j=0; j<n; j++) if (t[j] >= 0) l2[j] += tmp;
43         for (k=0; k<=q; k++) l1[s[k]] -= tmp;
44     }
45 }
46 for (i=0; i<m; i++) ret += mat[i][match1[i]];
47 return ret;
48 }

```

8.2. Enteros largos

Librería para trabajar con enteros largos. Se define $BASE = 10^{BEXP}$, y el número máximo de dígitos que se puede representar es $BEXP * MAXN$. Hay que asegurarse de que $2 * BASE < 2^{32}$ y $MAXN * BASE^2 < 2^{64}$.

```

1 typedef unsigned int uint;
2 typedef unsigned long long ulong;
3
4 #define MAXN 16
5 #define BASE 100000000
6 #define BEXP 8
7
8 struct lint {
9     uint d[MAXN];
10
11     lint() {
12         memset(d, 0, sizeof(d));
13     }
14     lint(int n) {
15         for (int i=0; i<MAXN; i++, n/=BASE) {
16             d[i] = n%BASE;
17         }
18     }
19 };
20
21 void print(lint n) {
22     bool flag = false;
23     for (int i=MAXN-1; i>=0; i--) {
24         if (flag) printf("%0*d", BEXP, n.d[i]);
25         else if (n.d[i] > 0 || i == 0) {printf("%d", n.d[i]); flag =
26             true;}
27     }
28 }
29
30 lint read(const char input[]) {
31     lint RES;
32     int i, j, k;
33
34     for (i=strlen(input)-1, j=1, k=0; i>=0; i--, j*=10) {
35         if (j == BASE) {j=1; k++;}
36         RES.d[k] += j*(input[i]-'0');
37     }
38 }

```

```

37     return RES;
38 }
39
40 lint sum(const lint &a, const lint &b) {
41     lint RES;
42     for (int i=0; i<MAXN; i++) RES.d[i] = a.d[i]+b.d[i];
43     for (int i=0; i<MAXN-1; i++) {RES.d[i+1] += RES.d[i]/BASE; RES.d
44         [i] = RES.d[i]%BASE;}
45     return RES;
46 }
47 lint operator+(const lint &a, const lint &b) {return sum(a, b);}
48
49 lint sum(const lint &a, const uint &b) {
50     lint RES=a;
51     RES.d[0] += b;
52     for (int i=0; i<MAXN-1; i++) {RES.d[i+1] += RES.d[i]/BASE; RES.d
53         [i] = RES.d[i]%BASE;}
54     return RES;
55 }
56 lint operator+(const lint &a, const uint &b) {return sum(a, b);}
57
58 lint sub(const lint &a, const lint &b) //CUIDADO CON a < b!
59     lint RES;
60     int tmp[MAXN];
61     memset(tmp, 0, sizeof(tmp));
62     for (int i=0; i<MAXN; i++) {
63         tmp[i] += a.d[i]-b.d[i];
64         if (i < MAXN-1 && tmp[i] < 0) {
65             tmp[i+1]--;
66             tmp[i] += BASE;
67         }
68         RES.d[i] = tmp[i];
69     }
70     return RES;
71 }
72 lint operator-(const lint &a, const lint &b) {return sub(a, b);}
73
74 lint sub(const lint &a, const uint &b) {
75     return a-lint(b);
76 }
77 lint operator-(const lint &a, const uint &b) {return sub(a, b);}
78
79 lint mul(const lint &a, const lint &b) {
80     lint RES;
81     int i, j;
82     ulong tmp[MAXN];
83     memset(tmp, 0ULL, sizeof(tmp));
84     for (i=0; i<MAXN; i++) {
85         for (j=0; j<MAXN-i; j++) {
86             tmp[j+i] += ((ulong)a.d[i])*((ulong)b.d[j]);
87         }

```

```

88     }
89     for (i=0; i<MAXN-1; i++) {tmp[i+1] += (tmp[i]/BASE); RES.d[i] +=
90         tmp[i]%BASE;}
91     return RES;
92 }
93 lint operator*(const lint &a, const lint &b) {return mul(a, b);}
94
95 lint div(const lint &a, const uint &b) {
96     lint RES;
97     ulong tmp=0ULL;
98     for (int i=MAXN-1; i>=0; i--) {
99         tmp = tmp*BASE + a.d[i];
100        RES.d[i] = tmp/b;
101        tmp %= b;
102    }
103    return RES;
104 }
105 lint operator/(const lint &a, const uint &b) {return div(a, b);}
106
107 uint mod(const lint &a, const uint &b) {
108     ulong tmp=0ULL;
109     for (int i=MAXN-1; i>=0; i--) tmp = (tmp*BASE + a.d[i])%b;
110     return tmp;
111 }
112 uint operator%(const lint &a, const uint &b) {return mod(a, b);}
113
114 bool operator<(const lint &a, const lint &b) {
115     for (int i=MAXN-1; i>0; i--) if (a.d[i] != b.d[i]) return a.d[i]
116         < b.d[i];
117     return a.d[0] < b.d[0];
118 }
119
120 bool operator>(const lint &a, const lint &b) {
121     for (int i=MAXN-1; i>0; i--) if (a.d[i] != b.d[i]) return a.d[i]
122         > b.d[i];
123     return a.d[0] > b.d[0];
124 }
125
126 bool operator<=(const lint &a, const lint &b) {return !(a>b);}
127 bool operator>=(const lint &a, const lint &b) {return !(a<b);}
128
129 bool operator==(const lint &a, const lint &b) {
130     for (int i=MAXN-1; i>0; i--) if (a.d[i] != b.d[i]) return false;
131     return a.d[0] == b.d[0];
132 }
133
134 bool operator!=(const lint &a, const lint &b) {return !(a==b);}
135
136 lint div(const lint &a, const lint &b) {
137     lint S(0), E=a+1, M;
138     while (E > S+1) {

```

```

138     M = (S+E)/2;
139     if (M*b > a) E = M;
140     else S = M;
141 }
142 return S;
143 }
144 lint operator/(const lint &a, const lint &b) {return div(a,b);}
145
146 lint mod(const lint &a, const lint &b) {
147     return a - (a/b)*b;
148 }
149 lint operator%(const lint &a, const lint &b) {
150     return mod(a, b);
151 }

```

8.3. Fracciones

```

1 int gcd(int a, int b) { return b != 0 ? gcd(b, a%b) : a; }
2 int lcm(int a, int b) { return a!=0 || b!=0 ? a / gcd(a,b) * b : 0;
3     }
4
5 class fraction {
6     void norm() { int g = gcd(n,d); n/=g; d/=g; if( d<0 ) { n=-n; d
7         =-d; } }
8     int n, d;
9
10 public:
11     fraction() : n(0), d(1) {}
12     fraction(int n_, int d_) : n(n_), d(d_) {
13         assert(d != 0);
14         norm();
15     }
16
17     fraction operator+(const fraction& f) const {
18         int m = lcm(d, f.d);
19         return fraction(m/d*n + m/f.d*f.n, m);
20     }
21
22     fraction operator-(const fraction& f) const { return *this + (-f); }
23
24     fraction operator*(const fraction& f) const { return fraction(n*
25         f.n, d*f.d); }
26
27     fraction operator/(const fraction& f) const { return fraction(n*
28         f.d, d*f.n); }
29
30     bool fraction::operator<(const fraction& f) const { return n*f.d
31         < f.n*d; }
32
33     friend std::ostream& operator<<(std::ostream&, const fraction&);
34 };

```

```

29 ostream& operator<<(ostream& os, const fraction& f) {
30     os << f.n;
31     if(f.d != 1 && f.n != 0) os << '/' << f.d;
32     return os;
33 }

```

9.1. cmath

Las siguientes funciones pueden tomar como argumento **double**, **float** o **long double**, y devuelven un resultado del mismo tipo:

| | | |
|------------------------------|------------------------|--|
| $\sin(x)$ | $x \in \mathbb{R}$ | $\sin(x) \in [-1, 1]$ |
| $\cos(x)$ | $x \in \mathbb{R}$ | $\cos(x) \in [-1, 1]$ |
| $\tan(x)$ | $x \in \mathbb{R}$ | $\tan(x) \in \mathbb{R}$ |
| $\operatorname{asin}(x)$ | $x \in [-1, 1]$ | $\operatorname{asin}(x) \in [-\pi/2, \pi/2]$ |
| $\operatorname{acos}(x)$ | $x \in [-1, 1]$ | $\operatorname{acos}(x) \in [0, \pi]$ |
| $\operatorname{atan}(x)$ | $x \in \mathbb{R}$ | $\operatorname{atan}(x) \in [-\pi/2, \pi/2]$ |
| $\operatorname{atan2}(y, x)$ | $y, x \in \mathbb{R}$ | $\operatorname{atan2}(y, x) \in [-\pi, \pi]$ |
| $\cosh(x)$ | $x \in \mathbb{R}$ | $\cosh(x) \in [1, +\infty)$ |
| $\sinh(x)$ | $x \in \mathbb{R}$ | $\sinh(x) \in \mathbb{R}$ |
| $\tanh(x)$ | $x \in \mathbb{R}$ | $\tanh(x) \in (-1, 1)$ |
| $\exp(x)$ | $x \in \mathbb{R}$ | $\exp(x) \in \mathbb{R}^+$ |
| $\log(x)$ | $x \in \mathbb{R}^+$ | $\log(x) \in \mathbb{R}$ |
| $\log_{10}(x)$ | $x \in \mathbb{R}^+$ | $\log_{10}(x) \in \mathbb{R}$ |
| $\operatorname{sqr}(x)$ | $x \in \mathbb{R}_0^+$ | $\operatorname{sqr}(x) \in \mathbb{R}_0^+$ |
| $\operatorname{ceil}(x)$ | $x \in \mathbb{R}$ | $\operatorname{ceil}(x) \in \mathbb{Z}$ |
| $\operatorname{floor}(x)$ | $x \in \mathbb{R}$ | $\operatorname{floor}(x) \in \mathbb{Z}$ |
| $\operatorname{fabs}(x)$ | $x \in \mathbb{R}$ | $\operatorname{fabs}(x) \in \mathbb{R}_0^+$ |

9.2. __builtin__

| | |
|---|---|
| <code>int __builtin_ffs(unsigned int x)</code> | Posición del primer 1 desde la derecha. |
| <code>int __builtin_clz(unsigned int x)</code> | Cantidad de ceros desde la izquierda. |
| <code>int __builtin_ctz(unsigned int x)</code> | Cantidad de ceros desde la derecha. |
| <code>int __builtin_popcount(unsigned int x)</code> | Cantidad de 1's en x. |
| <code>int __builtin_parity(unsigned int x)</code> | 1 si x es par, 0 si es impar. |
| <code>double __builtin_powi(double x, int n)</code> | x^n sin garantías. |

9.3. iomanip

```
1| cout << fixed << setprecision(3);
```

Pontificia Universidad Catolica del Peru

Great circle
Graph facts
Minimum cost arborescence
Determine if a polynomial has no repeated root
Division de polinomios y gcd
Stable marriage
Pape conradt
Geometry algorithms
Manacher
Number theoretic algorithms
Dinic
Dado
Josephus
Minimum spanning circle
Bridges
Articulations
Catalan
Teorema de lucas
Triangulation
Numerous de bell
Stirling del segundo tipo
Stirling del primer tipo
Suffix array
Min lex rotation
z-function
Non recursive BCC
Stoer Wagner
Cribal lineal
Roman to int
Int to troman
Hl decomposition
Centros de masas
Paperweight
Tetrahedron
Bcc de chen
Nkmars
Circle from 3 points
Circle line intersection
Pairwise sum reconstruction
Digitos mas significativos
Treap
Closest pair of points
Pick, Kirchhoff y LIS2

1. Great Circle Distance

```
double greatCircle( double laa, double loa, double lab, double lob )
{
    double PI = acos( -1.0 ), R = 6378.0;
    double u[3] = { cos( laa ) * sin( loa ), cos( laa ) * cos( loa ), sin( laa ) };
    double v[3] = { cos( lab ) * sin( lob ), cos( lab ) * cos( lob ), sin( lab ) };
    double dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
    bool flip = false;
    if( dot < 0.0 )
    {
        flip = true;
        for( int i = 0; i < 3; i++ ) v[i] = -v[i];
    }
    double cr[3] = { u[1]*v[2] - u[2]*v[1], u[2]*v[0] - u[0]*v[2], u[0]*v[1] -
u[1]*v[0] };
    double theta = asin( sqrt( cr[0]*cr[0] + cr[1]*cr[1] + cr[2]*cr[2] ) );
    double len = theta * R;
    if( flip ) len = PI * R - len;
    return len;
}
```

2. Graph facts:

A graph is Hamiltonian if and only if its closure is Hamiltonian.

As complete graphs are Hamiltonian, all graphs whose closure is complete are Hamiltonian, which is the content of the following earlier theorems by Dirac and Ore.

Dirac (1952)

A simple graph with n vertices ($n \geq 3$) is Hamiltonian if each vertex has degree $n/2$ or greater.^[1]

Ore (1960)

A graph with n vertices ($n \geq 3$) is Hamiltonian if, for each pair of non-adjacent vertices, the sum of their degrees is n or greater (see Ore's theorem).

The following theorems can be regarded as directed versions:

Ghouila-Houiri (1960)

A strongly connected simple directed graph with n vertices is Hamiltonian or some vertex has a full degree smaller than n .

Meyniel (1973)

A strongly connected simple directed graph with n vertices is Hamiltonian or the sum of full degrees of some two distinct non-adjacent vertices is smaller than $2n - 1$.

The number of vertices must be doubled because each undirected edge corresponds to two directed arcs and thus the degree of a vertex in the directed graph is twice the

Pontificia Universidad Catolica del Peru

degree in the undirected graph.

A graph G is 2-edge-connected if and only if it has an orientation that is strongly connected.

Teorema de Euler: para un grafo planar conexo $V - E + F = 2$

minimum vertex cover \leq maximum matching (la igualdad ocurre con grafos bipartitos)

minimum vertex cover \leq maximum independent set

teorema de Kirchhoff: el numero de spanning trees de G es igual al determinante de cualquier cofactor de la matriz laplaciana de G

numero de labeled trees de n vertices: n^{n-2}

The number of spanning trees in a complete graph K_n with degrees d_1, d_2, \dots, d_n is equal to the multinomial coefficient

$$\binom{n-2}{d_1-1, d_2-1, \dots, d_n-1} = \frac{(n-2)!}{(d_1-1)!(d_2-1)! \cdots (d_n-1)!}$$

Si G es planar y $v \geq 3$, entonces $e \leq 3v - 6$

If G is planar and if $v \geq 3$ and there are no cycles of length 3, then $e \leq 2v - 4$.

siempre probar tests con los siguientes grafos:

3. MINIMUM COST ARBORESCENCE

```
#define MAX_V 1000
typedef int edge_cost;
edge_cost INF = INT_MAX;

int V,root,prev[MAX_V];
bool adj[MAX_V][MAX_V];
edge_cost G[MAX_V][MAX_V],MCA;
bool visited[MAX_V],cycle[MAX_V];

void add_edge(int u, int v, edge_cost c){
    if(adj[u][v]) G[u][v] = min(G[u][v],c);
    else G[u][v] = c;
    adj[u][v] = true;
}

void dfs(int v){
    visited[v] = true;

    for(int i = 0;i<V;++i)
        if(!visited[i] && adj[v][i])
            dfs(i);
}

bool check(){
    memset(visited,false,sizeof(visited));
    dfs(root);

    for(int i = 0;i<V;++i)
        if(!visited[i])
            return false;

    return true;
}

int exist_cycle(){
    prev[root] = root;

    for(int i = 0;i<V;++i){
        if(!cycle[i] && i!=root){
            prev[i] = i; G[i][i] = INF;

```

```
            for(int j = 0;j<V;++j)
                if(!cycle[j] && adj[j][i] && G[j][i]<G[prev[i]][i])
                    prev[i] = j;
        }
    }
    for(int i = 0,j;i<V;++i){
        if(cycle[i]) continue;
        memset(visited,false,sizeof(visited));
        j = i;
        while(!visited[j]){
            visited[j] = true;
            j = prev[j];
        }
        if(j==root) continue;
        return j;
    }
    return -1;
}

void update(int v){
    MCA += G[prev[v]][v];
    for(int i = prev[v];i!=v;i = prev[i]){
        MCA += G[prev[i]][i];
        cycle[i] = true;
    }
    for(int i = 0;i<V;++i)
        if(!cycle[i] && adj[i][v])
            G[i][v] -= G[prev[v]][v];
    for(int j = prev[v];j!=v;j = prev[j]){
        for(int i = 0;i<V;++i){
            if(cycle[i]) continue;

            if(adj[i][j]){
                if(adj[i][v]) G[i][v] = min(G[i][v],G[i][j]-G[prev[j]][j]);
                else G[i][v] = G[i][j]-G[prev[j]][j];
                adj[i][v] = true;
            }
        }

        if(adj[j][i]){
            if(adj[v][i]) G[v][i] = min(G[v][i],G[j][i]);
            else G[v][i] = G[j][i];
            adj[v][i] = true;
        }
    }
}
```

```

    }
  }
}

bool min_cost_arborescence(int _root){
    root = _root;
    if(!check()) return false;

    memset(cycle,false,sizeof(cycle));
    MCA = 0;

    int v;

    while((v = exist_cycle())!=-1)
        update(v);

    for(int i = 0;i<V;++i)
        if(i!=root && !cycle[i])
            MCA += G[prev[i]][i];

    return true;
}

```

4. DETERMINE IF A GIVEN POLYNOMIAL HAS NO REPEATED ROOT

Sylvester matrix

Formally, let p and q be two nonzero polynomials, respectively of degree m and n .

Thus:

$$p(z) = p_0 + p_1z + p_2z^2 + \dots + p_mz^m, \quad q(z) = q_0 + q_1z + q_2z^2 + \dots + q_nz^n.$$

The Sylvester matrix associated to p and q is then the

$$(n + m) \times (n + m)$$

Thus, if $m=4$ and $n=3$, the matrix

$$S_{pq} = \begin{pmatrix} p_4 & p_3 & p_2 & p_1 & p_0 & 0 & 0 \\ 0 & p_4 & p_3 & p_2 & p_1 & p_0 & 0 \\ 0 & 0 & p_4 & p_3 & p_2 & p_1 & p_0 \\ q_3 & q_2 & q_1 & q_0 & 0 & 0 & 0 \\ 0 & q_3 & q_2 & q_1 & q_0 & 0 & 0 \\ 0 & 0 & q_3 & q_2 & q_1 & q_0 & 0 \\ 0 & 0 & 0 & q_3 & q_2 & q_1 & q_0 \end{pmatrix}.$$

is:

// $p := p \bmod q$, scaled to integer coefficients; return new degree of p

5. Division de polinomios y gcd

```
public static int mod(BigInteger[] p, int dp, BigInteger[] q, int dq) {
```

```

    while (dp >= dq) {
        // p := p * q[dq] - q * p[dp] * x^(dp-dq)
        BigInteger f = p[dp];
        for (int i=0 ; i<=dp ; i++)
            p[i] = p[i].multiply(q[dq]);
        for (int i=0 ; i<=dq ; i++)
            p[dp-dq+i] = p[dp-dq+i].subtract(q[i].multiply(f));
        // degree of p has been reduced by at least 1

```

```

    while (dp >= 0 && p[dp].signum() == 0)
        --dp;
    }
    // divide coefficients of p by their gcd to keep them small
    BigInteger d = BigInteger.ZERO;
    for (int i=0 ; i<=dp ; i++)
        d = d.gcd(p[i]);
    for (int i=0 ; i<=dp ; i++)
        p[i] = p[i].divide(d);
    return dp;
}

```

// are the polynomials p and q relatively prime?

```
public static boolean relprime(BigInteger[] p, int dp, BigInteger[] q, int dq) {
```

Pontificia Universidad Catolica del Peru

```
// dp,dq = degrees of p,q (-1 for zero polynomial)
while (dp >= 0 && dq >= 0)
    // replace the greater of p,q by its remainder when divided by the other
    if (dp >= dq)
        dp = mod(p, dp, q, dq);
    else
        dq = mod(q, dq, p, dp);
// if either of p,q is zero, the other is their gcd which must be constant
return dp <= 0 && dq <= 0;
}
```

```
public static void main(String[] arg) throws Exception {
    StreamTokenizer st = new StreamTokenizer(new BufferedReader(new
InputStreamReader(System.in)));
    st.nextToken();
    for (int t = (int) st.nval ; t > 0 ; t--) {
        st.nextToken();
        int n = (int) st.nval;
        // a[i] is coefficient of x^i
        BigInteger[] a = new BigInteger[n+1];
        for (int i=0 ; i<=n ; i++) {
            st.nextToken();
            a[n-i] = BigInteger.valueOf((int) st.nval);
        }
        // d is derivative of polynomial a
        BigInteger[] d = new BigInteger[n];
        for (int i=0 ; i<n ; i++)
            d[i] = a[i+1].multiply(BigInteger.valueOf(i+1));
        // check a and d for common factors
        System.out.println(relprime(a, n, d, n-1) ? "Yes!" : "No!");
    }
}
```

6. Stable Marriage

```
* MAIN FUNCTION: stableMarriage()
* - m:      number of men.
* - n:      number of women (must be at least as large as m).
* - L[i][]: the list of women in order of decreasing preference of man i.
```

```
* - R[j][i]: the attractiveness of i to j.
* OUTPUTS:
* - L2R[]:   the mate of man i (always between 0 and n-1)
* - R2L[]:   the mate of woman j (or -1 if single)
#define MAXM 1024
#define MAXW 1024

int m, n;
int L[MAXM][MAXW], R[MAXW][MAXM];
int L2R[MAXM], R2L[MAXW];

int p[MAXM];

void stableMarriage()
{
    static int p[128];
    memset( R2L, -1, sizeof( R2L ) );
    memset( p, 0, sizeof( p ) );

    // Each man proposes...
    for( int i = 0; i < m; i++ )
    {
        int man = i;
        while( man >= 0 )
        {
            // to the next woman on his list in order of decreasing preference,
            // until one of them accepts;
            int wom;
            while( 1 )
            {
                wom = L[man][p[man]++];
                if( R2L[wom] < 0 || R[wom][man] > R[wom][R2L[wom]] ) break;
            }

            // Remember the old husband of wom.
            int hubby = R2L[wom];

            // Marry man and wom.
            R2L[L2R[man] = wom] = man;

            // If a guy was dumped in the process, remarry him now.
            man = hubby;
        }
    }
```

```
}
}
```

7. PAPE CONRADT

```
int n, mate[100], inner[100], gf[100];
#define NONE -1
.
for(int i = 0; i < n; i++){
    mate[i] = NONE;
    gf[i] = NONE;
}
for(int u = 0; u < n; u++){
    if(mate[u] == NONE)
        for(int i = 0; i < adj[u].size(); i++){
            int v = adj[u][i];
            if(mate[v] == NONE){
                mate[u] = v; mate[v] = u;
                break;
            }
        }
}

int w;
for(int r = 0; r < n; r++){
    if(mate[r] == NONE){
        memset(inner, 0, sizeof inner);
        gf[r] = NONE;
        inner[r] = true;
        queue<int> q;
        q.push(r);
        bool ok = false;

        while(!q.empty() && !ok){
            int u = q.front(); q.pop();
            for(int i = 0; i < adj[u].size(); i++){
                int v = adj[u][i];
                if(!inner[v]){
                    if(mate[v] == NONE){
```

```
while(u != NONE){
    w =
    mate[u] =
    v = w;
    u = gf[u];
}
ok = true;
break;
}

for(w = u; w != NONE &&
w != v; w = gf[w]);

if(w == NONE){
    inner[v] = true;
    w = mate[v];
    gf[w] = u;
    q.push(w);
}
}
}
}
}
```

8. GEOMETRY ALGORITHMS

Formulas for regular tetrahedron

For a regular tetrahedron of edge length a :

$$V = \frac{abc}{6} \sqrt{1 + 2 \cos \alpha \cos \beta \cos \gamma - \cos^2 \alpha - \cos^2 \beta - \cos^2 \gamma}$$

where α, β, γ are the angles at the vertices a, b, and c. The angle α is at vertex a, β at vertex b, and γ at vertex c.

Given the distances between the vertices of a tetrahedron the volume can be computed using the Cayley-Menger

$$288 \cdot V^2 = \begin{vmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & d_{12}^2 & d_{13}^2 & d_{14}^2 \\ 1 & d_{12}^2 & 0 & d_{23}^2 & d_{24}^2 \\ 1 & d_{13}^2 & d_{23}^2 & 0 & d_{34}^2 \\ 1 & d_{14}^2 & d_{24}^2 & d_{34}^2 & 0 \end{vmatrix}$$

determinant:

where the subscripts

$i, j \in \{1, 2, 3, 4\}$ $\{a, b, c, d\}$ d_{ij} Tartaglia's formula, is essentially due to the painter Piero della Francesca in the 15th century, as a three dimensional analogue of the 1st century Heron's formula for the area of a triangle.[4] Distance between the edges

Then another volume formula is given

$$V = \frac{d|(\mathbf{a} \times (\mathbf{b} - \mathbf{c}))|}{6},$$

If OABC forms a generalized tetrahedron with a vertex O as the origin and vectors

$$\mathbf{a}, \mathbf{b}, \mathbf{c} \quad \frac{6V}{|\mathbf{b} \times \mathbf{c}| + |\mathbf{c} \times \mathbf{a}| + |\mathbf{a} \times \mathbf{b}| + |(\mathbf{b} \times \mathbf{c}) + (\mathbf{c} \times \mathbf{a}) + (\mathbf{a} \times \mathbf{b})|}$$

and the radius of the circumsphere is given

$$R = \frac{|\mathbf{a}^2(\mathbf{b} \times \mathbf{c}) + \mathbf{b}^2(\mathbf{c} \times \mathbf{a}) + \mathbf{c}^2(\mathbf{a} \times \mathbf{b})|}{12V}$$

by:

which gives the radius of the twelve-point

$$r_T = \frac{|\mathbf{a}^2(\mathbf{b} \times \mathbf{c}) + \mathbf{b}^2(\mathbf{c} \times \mathbf{a}) + \mathbf{c}^2(\mathbf{a} \times \mathbf{b})|}{36V}$$

sphere:

$$6V = |\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})|.$$

where:

Nota bene, in the formulas throughout this section, the scalar \mathbf{a}^2 represents the inner vector product $\mathbf{a} \cdot \mathbf{a}$, similarly \mathbf{b}^2 and \mathbf{c}^2 .

The vector position of various centers are given as follows:

$$\mathbf{G} = \frac{\mathbf{a} + \mathbf{b} + \mathbf{c}}{4}.$$

The centroid

$$\mathbf{O} = \frac{\mathbf{a}^2(\mathbf{b} \times \mathbf{c}) + \mathbf{b}^2(\mathbf{c} \times \mathbf{a}) - \mathbf{c}^2(\mathbf{a} \times \mathbf{b})}{2\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})},$$

The circumcenter

The Monge

$$\mathbf{M} = \frac{\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) + \mathbf{b} \cdot (\mathbf{c} \times \mathbf{a}) + \mathbf{c} \cdot (\mathbf{a} \times \mathbf{b})}{2\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})},$$

point

The Euler line relationships

$$\mathbf{G} = \mathbf{M} + \frac{1}{2}(\mathbf{O} - \mathbf{M})_{\mathbf{T} - \mathbf{M} + \frac{1}{3}(\mathbf{O} - \mathbf{M})}$$

are:

where \mathbf{T}

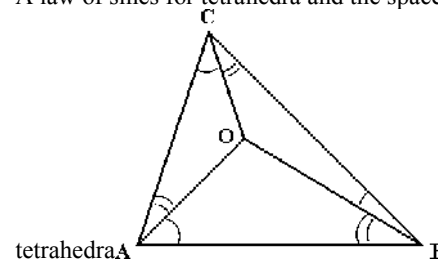
$$\mathbf{a} \cdot \mathbf{O} = \frac{\mathbf{a}^2}{2} \quad \mathbf{b} \cdot \mathbf{O} = \frac{\mathbf{b}^2}{2} \quad \mathbf{c} \cdot \mathbf{O} = \frac{\mathbf{c}^2}{2}$$

Also:

$$\mathbf{a} \cdot \mathbf{M} = \frac{\mathbf{a} \cdot (\mathbf{b} + \mathbf{c})}{2} \quad \mathbf{b} \cdot \mathbf{M} = \frac{\mathbf{b} \cdot (\mathbf{c} + \mathbf{a})}{2} \quad \mathbf{c} \cdot \mathbf{M} = \frac{\mathbf{c} \cdot (\mathbf{a} + \mathbf{b})}{2}.$$

and:

A law of sines for tetrahedra and the space of all shapes of



tetrahedra

A corollary of the usual law of sines is that in a tetrahedron with vertices O, A, B, C , we

have

$$\sin \angle OAB \cdot \sin \angle OBC \cdot \sin \angle OCA = \sin \angle OAC \cdot \sin \angle OCB \cdot \sin \angle OBA.$$

9. Manacher

vector<int> d1 (n);

int l=0, r=-1;

Pontificia Universidad Catolica del Peru

```
for (int i=0; i<n; ++i) {
    int k = (i>r ? 0 : min (d1[l+r-i], r-i)) + 1;
    while (i+k < n && i-k >= 0 && s[i+k] == s[i-k]) ++k;
    d1[i] = --k;
    if (i+k > r)
        l = i-k, r = i+k;
}

vector<int> d2 (n);
l=0, r=-1;
for (int i=0; i<n; ++i) {
    int k = (i>r ? 0 : min (d2[l+r-i+1], r-i+1)) + 1;
    while (i+k-1 < n && i-k >= 0 && s[i+k-1] == s[i-k]) ++k;
    d2[i] = --k;
    if (i+k-1 > r)
        l = i-k, r = i+k-1;
}
```

10. // Number Theoretic Algorithms //

```
/******
 * GCD *
*****
 * Euclidean algorithm. Works on non-negative integers.
**/
int gcd( int a, int b ) { return( b == 0 ? a : gcd( b, a % b ) ); }
long long gcd( long long a, long long b ) { return( b == 0 ? a : gcd( b, a % b ) ); }
template< Int > Int gcd( Int a, Int b ) { return( b == 0 ? a : gcd( b, a % b ); }

/******
 * A triple of integers *
*****
 * USED BY: egcd, msolve, inverse, ldioph
**/
template< class Int >
struct Triple
{
    Int d, x, y;
    Triple( Int q, Int w, Int e ) : d( q ), x( w ), y( e ) {}
};
```

```
};

/******
 * Extended GCD *
*****
 * Given nonnegative a and b, computes d = gcd( a, b )
 * along with integers x and y, such that d = ax + by
 * and returns the triple (d, x, y).
 * WARNING: needs a small modification to work on
 * negative integers (operator% fails).
 * REQUIRES: struct Triple
 * USED BY: msolve, inverse, ldioph
**/
template< class Int >
Triple< Int > egcd( Int a, Int b )
{
    if( !b ) return Triple< Int >( a, Int( 1 ), Int( 0 ) );
    Triple< Int > q = egcd( b, a % b );
    return Triple< Int >( q.d, q.y, q.x - a / b * q.y );
}

//TEOREMA CHINO
Chino(a1, a2, m1, m2){
<x, y> egcd(m1, m2)
return (m1 * x * a2 + m2 * y * a1) % m1 * m2

/******
 * Modular Linear Equation Solver *
*****
 * Given a, b and n, solves the equation ax = b (mod n)
 * for x. Returns the vector of solutions, all smaller
 * than n and sorted in increasing order. The vector is
 * empty if there are no solutions.
 * #include <vector>
 * REQUIRES: struct Triple, egcd
**/
template< class Int >
vector< Int > msolve( Int a, Int b, Int n )
{
    if( n < 0 ) n = -n;
    Triple< Int > t = egcd( a, n );
    vector< Int > r;
    if( b % t.d ) return r;
```

Pontificia Universidad Catolica del Peru

```
    Int x = ( b / t.d * t.x ) % n;
    if( x < Int( 0 ) ) x += n;
    for( Int i = 0; i < t.d; i++ )
        r.push_back( ( x + i * n / t.d ) % n );
    return r;
}
```

```
* Linear Diophantine Equation Solver
* Solves integer equations of the form ax + by = c
* for integers x and y. Returns a triple containing
* the answer (in .x and .y) and a flag (in .d).
* If the returned flag is zero, then there are no
* solutions. Otherwise, there is an infinite number
* of solutions of the form
*     x = t.x + k * b / t.d,
*     y = t.y - k * a / t.d;
* where t is the returned triple, and k is any
* integer.
* REQUIRES: struct Triple, egcd
**/
template< class Int >
Triple< Int > ldioph( Int a, Int b, Int c )
{
    Triple< Int > t = egcd( a, b );
    if( c % t.d ) return Triple< Int >( 0, 0, 0 );
    t.x *= c / t.d; t.y *= c / t.d;
    return t;
}

/*****
```

11. DINIC

```
const int maxnode=20000+5;
const int maxedge=1000000+5;
const int oo=1000000000;
```

```
int node,src,dest,nedge;
```

```
int head[maxnode],point[maxedge],
next[maxedge],flow[maxedge],capa[maxedge];
int dist[maxnode],Q[maxnode],work[maxnode];
```

```
//inicializa el network, con _node igual a numero de nodos, _src como fuente y
como _dest como sink
void init(int _node,int _src,int _dest)
{
    node=_node;
    src=_src-1;
    dest=_dest-1;
    for (int i=0;i<node;i++) head[i]=-1;
    nedge=0;
}
//anhade la arista de u a v con capacidad c1 y la arista de v a u con capacidad c2
void addedge(int u,int v,int c1,int c2)
{
    point[nedge]=v,capa[nedge]=c1,flow[nedge]=0,next[nedge]=head[u],head[u]=(nedge++);
    point[nedge]=u,capa[nedge]=c2,flow[nedge]=0,next[nedge]=head[v],head[v]=(nedge++);
}
//bfs de dinic
bool dinic_bfs()
{
    memset(dist,255,sizeof(dist));
    dist[src]=0;
    int sizeQ=0;
    Q[sizeQ++]=src;
    for (int cl=0;cl<sizeQ;cl++)
        for (int k=Q[cl],i=head[k];i>=0;i=next[i])
            if (flow[i]<capa[i] && dist[point[i]]<0)
            {
                dist[point[i]]=dist[k]+1;
                Q[sizeQ++]=point[i];
            }
    return dist[dest]>=0;
}
//dfs de dinic
int dinic_dfs(int x,int exp)
{
    if (x==dest) return exp;
    for (int &i=work[x];i>=0;i=next[i])
```

```

{
    int v=point[i],tmp;
    if (flow[i]<capa[i] && dist[v]==dist[x]+1 && (tmp=dinic_dfs(v,min(exp,capa[i]-
flow[i]))>0)
    {
        flow[i]+=tmp;
        flow[i^1]-=tmp;
        return tmp;
    }
}
return 0;
}
//flujo
int dinic_flow()
{
    int result=0;
    while (dinic_bfs())
    {
        for (int i=0;i<node;i++) work[i]=head[i];
        while (1)
        {
            int delta=dinic_dfs(src,oo);
            if (delta==0) break;
            result+=delta;
        }
    }
    return result;
}

int main(){

    init(node,src,dest);
    // addedge(u,v,c1,c2);
    int flow=dinic_flow();
    return flow==totalC && flow==totalD;
}

```

12. DADO

```

struct dice{
    //top, front, left, ri, ba, bo
    int is[6];
    void rollX(){
        roll(0, 1, 5, 4);
    }
    void rollY(){
        roll(0, 3, 5, 2);
    }
    void rollX(){
        roll(1, 3, 4, 2);
    }
    void roll(int a, int b, int c, int d) {
        int t = is[d];
        is[d] = is[c];
        is[c] = is[b];
        is[b] = is[a];
        is[a] = t;
    }

}

```

13. JOSEPHUS

```

#pragma comment(linker,"/STACK:256000000")

using namespace std;

long long joseph (long long n,long long k) {
    if (n==1LL) return 0LL;
    if (k==1LL) return n-1LL;
    if (k>n) return (joseph(n-1LL,k)+k)%n;
    long long cnt=n/k;
    long long res=joseph(n-cnt,k);
    res-=n%k;
    if (res<0LL) res+=n;
    else res+=res/(k-1LL);
}

```



```
    return res;
}
```

14. MINIMUM SPANNING CIRCLE

```
typedef pair<point, double> circle;
```

```
bool in_circle(circle C, point p){
    return cmp((p - C.first).abs(), C.second) <= 0;
}

point circumcenter(point p, point q, point r) {
    point a = p - r, b = q - r, c = point(a * (p + r) / 2, b * (q + r) / 2);
    return point(c % point(a.y, b.y), point(a.x, b.x) % c) / (a % b);
}

point T[100001];
circle spanning_circle(int n) {
    random_shuffle(T, T + n);
    circle C(point(), -INFINITY);
    for (int i = 0; i < n; i++) if (!in_circle(C, T[i])) {
        C = circle(T[i], 0);
        for (int j = 0; j < i; j++) if (!in_circle(C, T[j])) {
            C = circle((T[i] + T[j]) / 2, (T[i] - T[j]).abs() / 2);
            for (int k = 0; k < j; k++) if (!in_circle(C, T[k])) {
                point o = circumcenter(T[i], T[j], T[k]);
                C = circle(o, (o - T[k]).abs());
            }
        }
    }
    return C;
}
```

15. bridges

```
#define MAX 10000
vector<int> adj[MAX];
vector<pair<int,int> > bridges;
int times[MAX];
int timer;
```

```
int dfs(int u, int par){
    int low = times[u] = timer++;
    for(int i = 0; i < adj[u].size(); i++){
        int v = adj[u][i];
        if(times[v] == -1){
            int lowv = dfs(v, u);
            if(lowv > times[u])
                bridges.push_back(make_pair(u, v));
            else low = min(low, lowv);
        } else if(v != par) low = min(low, times[v]);
    }
    return low;
}
```

16. ARTICULATIONS

```
#define MAX 200
int times[MAX];
int timer;
vector<int> adj[MAX];
set<int> art;

int dfs(int u, int par){
    int low = times[u] = timer++;
    int child = 0;
    for(int i = 0; i < adj[u].size(); i++){
        int v = adj[u][i];
        if(times[v] == -1){
            child++;
            int lowv = dfs(v, u);
            if(lowv >= times[u]) art.insert(u);
            else low = min(low, lowv);
        } else if(v != par) low = min(low, times[v]);
    }
    if(u == 1 && child <= 1) art.erase(u);
    return low;
}
```

17. CATALAN

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \quad \text{for } n \geq 0.$$

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} \quad \text{for } n \geq 0,$$

1. C_n is the number of Dyck words of length $2n$. A Dyck word is a string consisting of n X's and n Y's such that no initial segment of the string has more Y's than X's
2. C_n counts the number of expressions containing n pairs of parentheses which are correctly matched
3. C_n is the number of different ways $n+1$ factors can be completely parenthesized (or the number of ways of associating n applications of a binary operator)
4. C_n is the number of full binary trees with $n+1$ leaves.
5. C_n is the number of stack-sortable permutations of $1; \dots; n$.
6. C_n is the number of permutations of $1; \dots; n$ that avoid the pattern 123 (or any of the other patterns of length 3); that is, the number of permutations with no three-term increasing subsequence

18. TEOREMA DE LUCAS

For non-negative integers m and n and a prime p , the following congruence relation holds:

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

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

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

are the base p expansions of m and n respectively.

//triangulation

From the Guide: Look at the edge $v_0 v_{n-1}$, in any triangulation it is contained in exactly one triangle.

Guess what will be the other point of that triangle. Let $m[a][b]$ ($a < b$) be the value of the best triangulation

of the (part of) polygon $v_a v_{a+1} \dots v_b v_a$.

This time, the solution of the longer segments depends on the shorter segments.

19. Best Triangulation

```
m[...][...] = 0;
for(len=3; len<n; len++)
for(a=0; a+len<n; a++){
b=a+len; m[a][b]=1e+10;
for(c=a+1; c<b; c++){
double t=m[a][c]+m[c][b];
if(c>a+1) t+=length(a to c);
if(c<b-1) t+=length(c to b);
m[a][b] <?= t;
}
}
```

20. triangulos 3D

```
bool insideT(punto P,punto A,punto B,punto C,bool tipo){
```

```
    punto u1 = B-A, u2 = C-A, N = u1%u2;
    if( abs( (P-A)*N )>eps ) return 0;
    double c1 = (P-A)*u1, c2 = (P-A)*u2;
    double u11 = u1*u1, u12 = u1*u2, u22 = u2*u2;
    double delta = u11*u22 - u12*u12;
    double s1 = (c1*u22 - c2*u12)/delta;
    double s2 = (c2*u11 - c1*u12)/delta;
    if( tipo==0 ){
        if( s1+s2>1+eps || s1<-eps || s2<-eps ) return 0;
        return 1;
    }else{
        if( s1>1+eps || s2>1+eps || s1<-eps || s2<-eps ) return 0;
        return 1;
    }
}
```

```

bool corta(punto P,punto Q,punto A,punto B,punto C){
    punto u1 = B-A, u2 = C-A, v = Q-P, N = u1%u2;
    if( abs(v*N)>eps ){
        double den = v*N;
        double t = (A-P)*N/den;
        double s1 = (P-A)*(u2%v)/den, s2 = (P-A)*(v%u1)/den;
        if( t>1+eps || t<-eps || s1<-eps || s2<-eps || s1+s2>1+eps ) return 0;

        return 1;
    } else{
        if( insideT(P-A,A,B,A-v,1) ) return 1;
        if( insideT(P-A,A,C,A-v,1) ) return 1;
        if( insideT(P-B,B,C,B-v,1) ) return 1;
        return 0;
    }
}

int T;
punto A,B,C,P,Q,R;
bool secruzan(){
    if( insideT(P,A,B,C,0) || insideT(Q,A,B,C,0) || insideT(R,A,B,C,0) ) return 1;
    if( insideT(A,P,Q,R,0) || insideT(B,P,Q,R,0) || insideT(C,P,Q,R,0) ) return 1;
    if( corta(P,Q,A,B,C) || corta(P,R,A,B,C) || corta(Q,R,A,B,C) ) return 1;
    if( corta(A,B,P,Q,R) || corta(A,C,P,Q,R) || corta(B,C,P,Q,R) ) return 1;
    return 0;
}

int main()
{
    cin >> T;
    punto A,B,C,a,b,c;
    f(cases,0,T){
        printf( "Case #%d: ", cases+1);
        A.read(); B.read(); C.read();
        a.read(); b.read(); c.read();
        solve(A,B,C,a,b,c).write();
    }
}

```

21. Numeros de Bell

1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, ...

The Bell numbers satisfy this recursion formula:

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

And they satisfy "Touchard's congruence": If p is any prime number then

$$B_{p+n} \equiv B_n + B_{n+1} \pmod{p},$$

$$B_{p^m+n} \equiv mB_n + B_{n+1} \pmod{p},$$

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

$$B_{r+m} = \sum_{k=0}^n \sum_{j=0}^m \left\{ \begin{matrix} m \\ j \end{matrix} \right\} \binom{n}{k} j^{n-k} B_k,$$

22. Stirling numbers of the second kind

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

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

for $k > 0$ with initial conditions

$$\left\{ \begin{matrix} 0 \\ 0 \end{matrix} \right\} = 1 \quad \text{and} \quad \left\{ \begin{matrix} n \\ 0 \end{matrix} \right\} = \left\{ \begin{matrix} 0 \\ n \end{matrix} \right\} = 0$$

22. Stirling numbers of the first kind

The unsigned Stirling numbers of the first kind can be calculated by the recurrence relation

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = n \begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix}$$

for $k > 0$, with the initial conditions

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1 \quad \text{and} \quad \begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0$$

23. SUFFIX ARRAY

```
int n,t; //n es el tamanho de la cadena
int p[MAXN],r[MAXN],h[MAXN];
string s;
```

```
bool comp(int i,int j){ return p[i+t] < p[j+t]; }
```

```
void suff_arr(){
    int bc[256];
        f(i,0,256) bc[i] = 0;
        f(i,0,n) ++bc[s[i]];
        f(i,1,256) bc[i] += bc[i-1];
        f(i,0,n) r[--bc[s[i]]] = i;
        f(i,0,n) p[i] = bc[s[i]];
        for(t=1; t<n; t*=2 ){
            for(int i = 0, j = 1; j<n; i = j++ ){
                while( j<n && p[r[j]] == p[r[i]] ) j++;
                if( j-i==1 ) continue;
                int *ini = r+i, *fin = r+j;
                sort(ini, fin, comp);
                int pri = p[*ini+t], num = i, pk;
                for( ; ini<fin; p[*ini++] = num ){
                    if(((pk=p[*ini+t]) == pri) || (i<=pri && pk<j)
                        else pri = pk, num = ini-r;
                }
            }
        }
    }
}

void lcp() {
    int tam = 0, i, j;
```

```
for(i = 0; i < n; i++) if (p[i] > 0) {
    j = r[p[i] - 1];
    while(s[i + tam] == s[j + tam]) ++tam;
    h[p[i] - 1] = tam;
    if (tam > 0) --tam;
}
h[n - 1] = 0;
}
```

24. MINIMUM LEX ROT

```
int n = s.size();
s = s + s;
int mini = 0, p = 1, l = 0;
while(p < n && mini + l + 1 < n)
    if(s[mini + l] == s[p + l])
        l++;
    else if(s[mini + l] < s[p + l]){
        p = p + l + 1;
        l = 0;
    }else if(s[mini + l] > s[p + l]){
        mini = max(mini + l + 1, p);
        p = mini + 1;
        l = 0;
    }
s = s.substr(mini, n);
```

25. Z-function

```
void zfunction(char s[], int z[], int sz){
    for(int i = 1, l = 0, r = 0; i < sz; ++i){
        z[i] = 0;
        if (i <= r)
            z[i] = min(z[i - l], r - i + 1);
        while (i + z[i] < sz && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r){
```

Pontificia Universidad Catolica del Peru

```
        l = i;
        r = i + z[i] - 1;
    }
}
z[0] = sz;
}
```

26. non recursive biconnected components

```
#define N 200001

const int size = 100005;
vector<set<int>> comps;
vector<int> E[N];
pii s[N]; int S = 0; int times[N], timer;

int indice[N], estado[N], par[N], low[N], child[N];
bool es[N];
int ss[N];
int q[N], qb;

void dfs2(int uu) {
    int szs = 0; ss[szs++] = uu; timer = 0;
    while(szs) {
        int u = ss[szs - 1];
        if(times[u] == -1) low[u] = times[u] = timer++;

        if(indice[u] == E[u].size()) szs--;
        else if(estado[u] == 0) {
            int v = E[u][indice[u]];
            if(times[v] != -1) {
                if(v != par[u] && times[v] <= times[u]) {
                    low[u] = min(low[u], times[v]); s[S++] = mp(u, v);
                }
            }
            indice[u]++;
        } else {
            child[u]++;
            par[v] = u;
            s[S++] = mp(u, v);
            estado[u]++;
            ss[szs++] = v;
        }
    }
}
```

```
    }
    } else {
        int v = E[u][indice[u]++];
        if(low[v] >= times[u]) {
            es[u] = true;
            int sz = 0;
            int tam = 0; pii x; pii e(u, v);
            set<int> CC;
            do {
                x = s[--S];
                CC.insert(x.fst); CC.insert(x.snd);
            } while(x != e);
            comps.pb(CC);
        } else low[u] = min(low[u], low[v]);
        estado[u] = 0;
    }
}
}
```

```
f(i, 0, n) times[i] = -1, es[i] = false, estado[i] = 0, indice[i] = 0, par[i] = -1, child[i] = 0;
```

```
    comps.clear();
    timer = 0;
    dfs2(0);
    if(child[0] < 2) es[0] = false;
```

27. STOER WAGNER

```
int g[100][100], v[100], w[100], na[100];
bool a[100];
```

```
int mincut(int n) {
    f(i, 0, n) v[i] = i;
    int best = inf;
    if(n <= 1) return 0;
    while(n > 1) {
        a[v[0]] = true;
        f(i, 1, n) {
```

```

        a[v[i]] = false;
        na[i - 1] = i;
        w[i] = g[v[0]][v[i]];
    }
    int prev = v[0];
    f(i, 1, n){
        int zj = -1;
        f(j, 1, n)
            if(!a[v[j]] && (zj < 0 || w[j] > w[zj]))
                zj = j;
        a[v[zj]] = true;
        if(i == n - 1){
            best = min(best, w[zj]);
            f(j, 0, n)
                g[v[j]][prev] = g[prev][v[j]] += g[v[zj]][v[j]];
            v[zj] = v[--n];
            break;
        }
        prev = v[zj];
        f(j, 1, n) if(!a[v[j]])
            w[j] += g[v[zj]][v[j]];
    }
}
if(best == inf) while(1);
return best;
}

```

28. CRIBA LINEAL

```

int A[MAX], P[MAX/10], pc = 0;

void criba(){
    f(i,2,MAX){
        if(!A[i]) P[++pc] = i, A[i] = pc;
        for(int j=1; j<=A[i] && i*P[j]<MAX; j++) A[i*P[j]] = j;
    }
}

```

29. ROMAN TO INT

```

string roman(string &s){
    int cont=0,pos=0;
    string x="";

    while(s[pos]=='M'){
        pos++;
        cont++;
    }

    x+='0'+cont;

    if(pos+1<s.size() && s[pos]=='C' && s[pos+1]=='M'){
        pos+=2;
        x+='9';
    }else if(pos<s.size() && s[pos]=='D'){
        pos++;
        cont=5;
        while(pos<s.size() && s[pos]=='C'){
            pos++;
            cont++;
        }
        x+='0'+cont;
    }else if(pos+1<s.size() && s[pos]=='C' && s[pos+1]=='D'){
        pos+=2;
        x+='4';
    }else{
        cont=0;
        while(pos<s.size() && s[pos]=='C'){
            pos++;
            cont++;
        }
        x+='0'+cont;
    }

    if(pos+1<s.size() && s[pos]=='X' && s[pos+1]=='C'){
        pos+=2;
        x+='9';
    }else if(pos<s.size() && s[pos]=='L'){

```

```
    pos++;
    cont=5;
    while(pos<s.size() && s[pos]=='X'){
        pos++;
        cont++;
    }
    x+='0'+cont;
} else if(pos+1<s.size() && s[pos]=='X' && s[pos+1]=='L'){
    pos+=2;
    x+='4';
} else{
    cont=0;
    while(pos<s.size() && s[pos]=='X'){
        pos++;
        cont++;
    }
    x+='0'+cont;
}

if(pos+1<s.size() && s[pos]=='I' && s[pos+1]=='X'){
    pos+=2;
    x+='9';
} else if(pos<s.size() && s[pos]=='V'){
    pos++;
    cont=5;
    while(pos<s.size() && s[pos]=='I'){
        pos++;
        cont++;
    }
    x+='0'+cont;
} else if(pos+1<s.size() && s[pos]=='I' && s[pos+1]=='V'){
    pos+=2;
    x+='4';
} else{
    cont=0;
    while(pos<s.size() && s[pos]=='I'){
        pos++;
        cont++;
    }
    x+='0'+cont;
}

while(x.size()>=1 && x[0]=='0') x.erase(0,1);
```

```
    return x;
}
```

30. INT TO ROMAN

```
string conv(int a){
    string romanos="";

    while(a>=1000) romanos+="M", a-=1000;
    if(a>=900) romanos+="CM", a-=900;
    if(a>=500) romanos+="D", a-=500;
    if(a>=400) romanos+="CD", a-=400;

    while(a>=100) romanos+="C", a-=100;
    if(a>=90) romanos+="XC", a-=90;
    if(a>=50) romanos+="L", a-=50;
    if(a>=40) romanos+="XL", a-=40;

    while(a>=10) romanos+="X", a-=10;
    if(a==9) romanos+="IX", a-=9;
    if(a>=5) romanos+="V", a-=5;
    if(a==4) romanos+="IV", a-=4;

    while(a>=1) romanos+="I", a--;

    return romanos;
}
```

31. HL Decomposition

```
#define N 100000
int n, par[N], q[N], h[N], head[N], tam[N], color[N];
vector<int> E[N];
struct segment{
    vector<int> has, indice;
    int t, m;

    void update(int pos, int ind, int val, int u, int le, int ri){
```

Pontificia Universidad Catolica del Peru


```
if(le == ri) has[u] = val, indice[u] = ind;
else{
    int mid = (le + ri) / 2;
    if(pos <= mid) update(pos, ind, val, u * 2 + 1, le, mid);
    else update(pos, ind, val, u * 2 + 2, mid + 1, ri);
    has[u] = has[u * 2 + 1] || has[u * 2 + 2];
}
}
void update(int alt, int ind, int val){
    update(alt - t, ind, val, 0, 0, m - 1);
}
int get(int to, int u, int le, int ri){
    if(!has[u] || le > to) return -1;

    if(le == ri) return indice[u];
    int mid = (le + ri) / 2;
    int aux = get(to, u * 2 + 1, le, mid);
    if(aux != -1) return aux;
    return get(to, u * 2 + 2, mid + 1, ri);
}
int get(int alt){
    return get(alt - t, 0, 0, m - 1);
}
void construct(int a, int b){
    m = h[b] - h[a] + 1;
    has.resize(4 * m);
    indice = has;
    t = h[a];
    for(int i = m - 1, u = b; i >= 0; i--, u = par[u])
        update(i, u + 1, 0, 0, 0, m - 1);
}
}S[N];

void make(){
    int qb = 0;
    q[qb++] = 0; h[0] = 0;
    f(i, 0, qb){
        int u = q[i];
        FOR(v, E[u]) if(h[*v] == -1)
            par[*v] = u, q[qb++] = *v, h[*v] = h[u] + 1;
    }
    for(int i = n - 1; i >= 0; i--){
        int u = q[i];
```

```
tam[u] = 1;
FOR(v, E[u]) if(par[*v] == u) tam[u] += tam[*v];
}
f(i, 0, n){
    int a = q[i];
    if(head[a] != -1) continue;
    int u = a;
    while(true){
        head[u] = a;
        int next = -1;
        FOR(v, E[u]) if(par[*v] == u && (next == -1 || tam[*v] > tam[next])) next = *v;
        if(next == -1) break;
        u = next;
    }
    S[a].construct(a, u);
}
}
```

32. Centros de masa

| | |
|--------------------|---|
| lamina |  |
| circular sector | $\frac{4R \sin\left(\frac{1}{2}\theta\right)}{3\theta}$ |
| circular segment | $\frac{4R \sin^3\left(\frac{1}{2}\theta\right)}{3(\theta - \sin\theta)}$ |
| isosceles triangle | $\frac{1}{3}h$ |
| parabolic segment | $\frac{2}{5}h$ |
| semicircle | $\frac{4R}{3\pi}$ |

33. PAPERWEIGHT

```
struct punto{
    double x,y,z;
```


Pontificia Universidad Catolica del Peru

```

    punto (){}
    punto (double a, double b, double c): x(a), y(b),
z(c){}
    punto operator+ (punto p){ return punto (x+p.x, y+p.y,
z+p.z); }
    punto operator- (punto p){ return punto (x-p.x, y-p.y,
z-p.z); }
    punto operator% (punto p){
        return punto (y*p.z - z*p.y, z*p.x - x*p.z,
x*p.y - y*p.x);
    }
    punto operator* (double t){ return punto (x*t, y*t,
z*t); }
    punto operator/ (double t){ return punto (x/t, y/t,
z/t); }
    double operator* (punto p){ return x*p.x + y*p.y +
z*p.z; }
    double norma (){ return sqrt (cua(x) + cua(y) +
cua(z)); }
    void read (){ scanf ("%lf%lf%lf", &x, &y, &z); }
};
struct lado{
    punto A,B;
    lado (){}
    lado (punto X, punto Y): A(X), B(Y) {}
};
struct cara{
    punto A,B,C;
    cara (){}
    cara (punto X, punto Y, punto Z): A(X), B(Y), C(Z) {}
};

double area (punto A, punto B, punto C){
    return ((B-A)%(C-A)).norma();
}

double d (punto P, cara face){
    punto A = face.A, B = face.B, C = face.C;
    punto N = (B-A) % (C-A);
    N = N/N.norma();
    return (P-A)*N;
}

```

```

punto H[10]; int sz;
bool operator< (punto A, punto B){
    if (A.x != B.x) return A.x < B.x;
    if (A.y != B.y) return A.y < B.y;
    return A.z < B.z;
}
void hull (vector<punto> v){
    punto N = (v[1]-v[0]) % (v[2]-v[0]);
    sz = 0;
    sort (all(v));

    f(i,0,v.size()) {
        while (sz>=2 && N*((H[sz-1] - v[i]) % (H[sz-2] -
v[i]))) > -eps) sz--;
        H[sz++] = v[i];
    }
    int t = sz+1;
    fd(i,v.size()-1,0){
        while (sz>=t && N*((H[sz-1] - v[i]) % (H[sz-2] -
v[i]))) > -eps) sz--;
        H[sz++] = v[i];
    }
    sz--;
}
bool dentro (punto P, punto A, punto B, punto C){
    double res = area (P,A,B) + area(P,B,C) + area(P,C,A) -
area (A,B,C);
    return abs(res) < eps;
}
bool dentro (punto P){
    double res = 0;
    f(i,1,sz-1){
        if (dentro (P, H[0], H[i], H[i+1])) return 1;
    }
    return 0;
}
double ud (punto P, punto A, punto B){
    return area (P,A,B) / (A-B).norma();
}

```

34. TETRAHEDRON

| | |
|--|--|
| Base plane area | $A_0 = \frac{\sqrt{3}}{4}a^2$ |
| Surface area ^[2] | $A = 4A_0 = \sqrt{3}a^2$ |
| Height ^[3] | $H = \frac{\sqrt{6}}{3}a$ |
| Volume ^[2] | $V = \frac{1}{3}A_0h = \frac{\sqrt{2}}{12}a^3$ |
| Angle between an edge and a face | $\arccos\left(\frac{1}{\sqrt{3}}\right) = \arctan(\sqrt{2})$ (approx. 54.7356°) |
| Angle between two faces ^[2] | $\arccos\left(\frac{1}{3}\right) = \arctan(2\sqrt{2})$ (approx. 70.5288°) |
| Angle between the segments joining the center and the vertices, ^[4] also known as the | $\arccos\left(\frac{-1}{3}\right) = 2\arctan(\sqrt{2})$ |

| | |
|---|---|
| "tetrahedral angle" | (approx. 109.4712°) |
| Solid angle at a vertex subtended by a face | $\arccos\left(\frac{23}{27}\right)$ (approx. 0.55129 steradians) |
| Radius of circumsphere ^[2] | $R = \sqrt{\frac{3}{8}}a$ |
| Radius of insphere that is tangent to faces ^[2] | $r = \frac{1}{3}R = \frac{a}{\sqrt{24}}$ |
| Radius of midsphere that is tangent to edges ^[2] | $r_M = \sqrt{rR} = \frac{a}{\sqrt{8}}$ |
| Radius of exspheres | $r_E = \frac{a}{\sqrt{6}}$ |
| Distance to exsphere center from a vertex | $\sqrt{\frac{3}{2}}a$ |

35. BCC de chen

```
int n,m,fin;
```

Pontificia Universidad Catolica del Peru

```

int orig[MAX], dest[MAX], pila[MAX], top = 0;
vint E[MAX];
int low[MAX], dfsn[MAX], part[MAX], timer;
int ponte[MAX], bicomp[MAX], nbicomp;

int dfsbcc (int u, int p = -1){
    low[u] = dfsn[u] = ++timer;
    int ch = 0;
    FOR(it, E[u]){
        int e = *it, v = VIZ (e, u);
        if (dfsn[v] == 0){
            pila[top++] = e;
            dfsbcc (v, u);
            low[u] = min (low[u], low[v]);
            ch++;
            if (low[v] >= dfsn[u]){
                part[u] = 1;
                nbicomp++;
                do{
                    fin = pila[--top];
                    bicomp[fin] = nbicomp;
                }while (fin != e);
            }
            if (low[v] == dfsn[v]) ponte[e] = 1;
        }else if (v!=p && dfsn[v] < dfsn[u]){
            pila[top++] = e;
            low[u] = min (low[u], dfsn[v]);
        }
    }
    return ch;
}

void bcc (){
    f(i,0,n) part[i] = dfsn[i] = 0;
    f(i,0,m) ponte[i] = 0;
    nbicomp = timer = 0;
    f(i,0,n) if (dfsn[i] == 0) part[i] = dfsbcc (i) >= 2;
}

```

36. NKMARS

```

struct node{
    int on, area;
    node (int x = 0, int y = 0) : on(x), area(y){}
}

```

```

};
int tam;
struct segmentTree{
    node seg[1<<17];
    int off;
    segmentTree(){
    };
    int area(){ return seg[0].area;}
    void add (int a, int b, int x = 0, int le = 0, int ri = tam){
        if (b<=le || ri<=a) return;
        if (a<=le && ri<=b){ seg[x].on++; seg[x].area = ri - le; return; }
        int me = (le + ri) / 2;
        add (a, b, L(x), le, me);
        add (a, b, R(x), me, ri);
        if(seg[x].on==0) seg[x].area = seg[L(x)].area + seg[R(x)].area;
        else seg[x].area = ri-le;
    }
    void take (int a, int b, int x = 0, int le = 0, int ri = tam){
        if (b<=le || ri<=a) return;
        if (a<=le && ri<=b){
            seg[x].on--;
            if (seg[x].on==0) seg[x].area = seg[L(x)].area + seg[R(x)].area;
            return;
        }
        int me = (le + ri) / 2;
        take (a, b, L(x), le, me);
        take (a, b, R(x), me, ri);
        if(seg[x].on==0) seg[x].area = seg[L(x)].area + seg[R(x)].area;
        else seg[x].area = ri-le;
    }
}
};
vector<pii> abre[MAX], cierra[MAX];
int main()
{
    int n;
    cin >> n;
    int a1,b1,a2,b2;
    f(i,0,n){
        scanf ("%d%d%d%d", &a1, &b1, &a2, &b2);
        abre[a1].pb (pii(b1,b2));
        cierra[a2].pb (pii(b1,b2));
    }
    segmentTree st;
}

```

Pontificia Universidad Catolica del Peru

```
tam = 1<<16;
int res = 0;
f(i,0,MAX){
    cout<<i;
    f(j,0,abre[i].size()) st.add (abre[i][j].fst, abre[i][j].snd);//cout <<1111<<"
"<<i<<endl;
    f(j,0,cierra[i].size()) st.take (cierra[i][j].fst, cierra[i][j].snd);// cout <<2222
<<" "<<i<<endl;
    res += st.area();
}
cout << res << endl;
```

37. CIRCLE FROM 3 POINTS

```
point center(double x1, double y1, double x2, double y2, double x3, double y3){
    point p;
    double A1 = x1-x2;
    double A2 = x2-x3;
    double B1 = y1-y2;
    double B2 = y2-y3;
    double C1 = (A1*(x1+x2) + B1*(y1+y2))/2;
    double C2 = (A2*(x2+x3) + B2*(y2+y3))/2;
    double d = A1*B2-A2*B1;
    if(fabs(d)<1e-7)return p;
    double y = (A1*C2-A2*C1)/d;
    double x = -(B1*C2-B2*C1)/d;
    p.x = x, p.y = y;
    return p;
}
```

38. CIRCLE LINE INTERSECTION

It is assumed that the circle is located at (0,0), r is the radius of the circle and a, b, c are the coefficients of the line. $ax + by + c = 0$

```
double r, a, b, c; // input

double x0 = -a*c/(a*a+b*b), y0 = -b*c/(a*a+b*b);
if (c*c > r*r*(a*a+b*b)+EPS)
    puts ("no points");
else if (abs (c*c - r*r*(a*a+b*b)) < EPS) {
    puts ("1 point");
    cout << x0 << " " << y0 << "\n";
```

```
}
else {
    double d = r*r - c*c/(a*a+b*b);
    double mult = sqrt (d / (a*a+b*b));
    double ax,ay,bx,by;
    ax = x0 + b * mult;
    bx = x0 - b * mult;
    ay = y0 - a * mult;
    by = y0 + a * mult;
    puts ("2 points");
    cout << ax << " " << ay << "\n" << bx << " " << by << "\n";
}
```

39. PAIRWISE SUM RECONSTRUCTION

```
bool pairsums( int *ans, multiset< int > &seq )
{
    int N = seq.size();
    if( N < 3 ) return false;
    __typeof( seq.end() ) it = seq.begin();
    int a = *it++, b = *it++, i = 2;

    for( ; i * ( i - 1 ) < 2 * N && it != seq.end(); i++, ++it )
    {
        // assume seq[i] = ans[1] + ans[2]
        ans[0] = a + b - *it;
        if( ans[0] & 1 ) continue;
        ans[0] >>= 1;

        // try ans[0] as a possible least element
        multiset< int > seq2 = seq;
        int j = 1;
        while( seq2.size() )
        {
            ans[j] = *seq2.begin() - ans[0];
            for( int k = 0; k < j; k++ )
            {
                __typeof( seq2.end() ) jt = seq2.find( ans[k] + ans[j] );
                if( jt == seq2.end() ) goto hell;
                seq2.erase( jt );
            }
            j++;
        }
```

```

    }
    hell;;
    if(j * (j - 1) < 2 * N) continue;
    return true;
}
return false;
}

```

40. Digos mas significativos

Como imprimir los N digitos mas significativos?... los 5 es asi: (notar q no es tan trivial como suena)

print n/d rounded to 5 sig digits

```

void PrintNumber(double x){
    double y;int p, left, moder;
    if(x>=1){
        left=0; while(x>=1){ x /= 10; left++; }
        // move x left 5 digits
        x *= 100000; p=int (x);
        y = x-p; if(y >= 0.5 - 0.000001) p++;
        //p has the 5 sig digits, so print
        if(left>=5){
            cout<<p; for(int i=5; i<left;i++) cout<<0;
        }
        else { // need a decimal
            moder=10000;
            for(int i=0;i<left;i++){
                cout<< p/moder; p %= moder; moder /=10;
            }
            cout<< '.';
            for(int i=left;i<5;i++){
                cout<<p/moder; p %= moder; moder /= 10;
            }
        } //else
    }
    else { // x<1
        cout<<"0.";
        // find how many 0's after decimal
        left=-1; while(x<1) {x *= 10; left++; }
        x *= 10000; p = int (x);
        y = x-p; if(y>=0.5) p++; //p now holds 5 sig digits
    }
}

```

```

// print 0's
for(int i=0;i<left;i++) cout<<0; cout<<p;
}

```

41. TREAP

```

ll mod = 10000000000;
struct node{
    node *L, *R;
    int y, x, rev, cnt, k;
    ll sum;
    node(int value, int key){
        L = R = NULL;
        sum = k = value;
    }
    x = key;
    y = rand();
    rev = 0; cnt = 1;
};

void refresh(node* T){
    if(!T) return;
    if(T->rev){
        if(T->L) T->L->rev ^= 1;
        if(T->R) T->R->rev ^= 1;
        swap(T->L, T->R);
        T->rev = 0;
    }
}

void update(node *T){
    if(!T) return;
    T->sum = T->k; T->cnt = 1;
    if(T->L) T->sum += T->L->sum,
        T->cnt += T->L->cnt;
    if(T->R) T->sum += T->R->sum,
        T->cnt += T->R->cnt;
}

void split(node *T, int x, node* &L, node* &R){
    if(!T) L = R = NULL;
    else if(T->x > x){
        split(T->L, x, L, T->L);
        R = T;
    }
}

```

```
        }else{
            split(T->R, x, T->R, R);
            L = T;
        }
        update(R); update(L);
    }
void split2(node *T, int cnt, node* &L, node* &R){
    refresh(T);
    if(!T) L = R = NULL;
    else if(cnt <= (T->L ? T->L->cnt : 0)){
        split2(T->L, cnt, L, T->L);
        R = T;
    }else{
        split2(T->R, cnt - (T->L ? T->L->cnt : 0)
- 1, T->R, R);
        L = T;
    }
    update(R); update(L);
}

node* merge(node*L, node*R){
    if(!L) return R;
    if(!R) return L;
    refresh(L); refresh(R);
    node *T;
    if(L->y > R->y)
        L->R = merge(L->R, R), T = L;
    else
        R->L = merge(L, R->L), T = R;
    update(T);
    return T;
}

node* add(node *T, node *N){
    if(!T || T->y < N->y){
        node* R; split(T, N->x, N->L, N->R);
        T = N;
    }else if(N->x < T->x)
        T->L = add(T->L, N);
    else
        T->R = add(T->R, N);
    update(T);
    return T;
}
```

```

}
int main(){
    freopen("reverse.in", "r", stdin);
    freopen("reverse.out", "w", stdout);
    ios::sync_with_stdio(false);
    int n, m;
    node *root = NULL;
    cin >> n >> m;
    f(i, 0, n){
        int x; cin >> x;
        root = add(root, new node(x, i));
    }
    while(m--){
        int q, l, r; cin >> q >> l >> r;
        node *A, *B, *C, *D;
        split2(root, r, A, B);
        split2(A, l - 1, C, D);
        if(q == 0)
            cout << D->sum << endl;
        else{
            D->rev ^= 1;
            refresh(D);
        }
        root = merge(merge(C, D), B);
    }
}
```

42. CLOSEST PAIR OF POINTS

```
#define px second
#define py first
typedef pair<long long, long long> pairll;
int n;
pairll pnts [100000];
set<pairll> box;
double best;
int compx(pairll a, pairll b) { return a.px<b.px; }
int main () {
    scanf("%d", &n);
    for (int i=0;i<n;++i) scanf("%lld %lld", &pnts[i].px, &pnts[i].py);
    sort(pnts, pnts+n, compx);
    best = 1500000000; // INF
```

```

box.insert(pnts[0]);
int left = 0;
for (int i=1;i<n;++i) {
while (left<i && pnts[i].px-pnts[left].px > best) box.erase(pnts[left++]);
for (typeof(box.begin()) it=box.lower_bound(make_pair(pnts[i].py-best, pnts[i].px-
best));
it!=box.end() && pnts[i].py+best>=it->py; it++)
best = min(best, sqrt(pow(pnts[i].py - it->py, 2.0)+pow(pnts[i].px - it->px, 2.0)));
box.insert(pnts[i]);
printf("%.2f\n", best);
}
return 0;
}

```

43. PICK

$$A = t + \frac{b}{2} - 1.$$

44. KIRCHHOFF para bipartitos

If G is the complete bipartite graph with vertices 1 to n_1 in one partition and vertices $n_1 + 1$ to n in the other partition, the number of labeled spanning trees of G is $n_1^{n_2-1} n_2^{n_1-1}$, where $n_2 = n - n_1$.

46. LIS2

```

set<par> A[MAX];
set<par>::iterator it,iit;
int n, x[MAX], y[MAX];
bool ok(int k,int i){
    it = A[k].lower_bound (par (x[i], -oo));
    if( it==A[k].begin() ) return 0;
    it--;
    return y[i] > it->second;
}

```

```

void update(int k, int i){
    it = A[k].lower_bound (par (x[i], y[i]));
    while( it!=A[k].end() && y[i] <= it->second ){
        iit = it;
        iit++;
        A[k].erase(it);
        it = iit;
    }
    if( (it==A[k].end() || x[i]!=it->first) && (it==A[k].begin() || y[i]!=(--it)->second)
)
        A[k].insert( par (x[i], y[i]));
}

```

```

int main()
{
    cin >> n;
    f(i,0,n) scanf("%d%d", x+i, y+i); // y[i] = -y[i];
    int tam = 0;
    f(i,0,n){
        int lo = 0, hi = tam;
        while( lo < hi ){
            int me = (lo+hi)/2;
            if( ok(me, i) ) lo = me+1;
            else hi = me;
        }
        if( lo==tam ) tam++;
        update (lo, i);
    }
    cout << tam << endl;
}

```



ANCHETA DE ALGORITMOS PARA MARATONES DE PROGRAMACIÓN

UNIVERSIDAD DE LOS ANDES

BOGOTÁ, COLOMBIA

Última versión: 2008/04/05 08:05 a.m.

AUTORES: ALEJANDRO SOTELO, FEDERICO ARBOLEDA E IVÁN REY.

CON LA GRAN COLABORACIÓN DE AUGUSTO TORRES.

AGRADECIMIENTOS ESPECIALES A RAFAEL GARCÍA POR SU DILIGENCIA
Y PACIENCIA PARA COLABORARNOS EN TODO MOMENTO.

POR PROBLEMAS DE ESPACIO NO SE SIGUIÓ NINGUNA REGLA ESTÁNDAR DE INDENTACIÓN.

ALERTA!: JAVA PUEDE IMPRIMIR “-0.0” EN VEZ DE “0.0”.

CLASES ÚTILES JDK 6.0

| | |
|----------------|--|
| java.awt | Point, Polygon, Shape |
| java.awt.geom | Area, GeneralPath, Line2D, Path2D, Rectangle2D |
| java.lang.math | Math, BigDecimal, BigInteger |
| java.util | ArrayList<E>, Arrays, Calendar, Collections, Date, LinkedList<E>, StringBuilder, TreeMap<K, V>, TreeSet<E> |
| java.text | DecimalFormat |

ARREGLOS (ORDENAMIENTO, SELECCIÓN, BÚSQUEDAS, PERMUTACIONES, ...)

CASOS DE EJEMPLO CON LONG

☒ Merge Sort – [John von Neumann (1945)] $\{O(n \cdot \log_2(n))\}$

```
void mergeSort(long[] arr, long[] arrTmp, int pi, int pf) {
    if (pf <= pi) return;
    int m = (pi + pf) / 2 + 1, i, j, k; mergeSort(arr, arrTmp, pi, m - 1); mergeSort(arr, arrTmp, m, pf);
    for (i = pi, j = m, k = pi; i <= m - 1 && j <= pf; k++) arrTmp[k] = arr[i] <= arr[j] ? arr[i++] : arr[j++];
    for (; i <= m - 1; k++, i++) arrTmp[k] = arr[i];
    for (k = pi; k < j; k++) arr[k] = arrTmp[k];
}
```

☒ Quick Sort – [C. A. R. Hoare (1960)] $\{O(n \cdot \log_2(n))\}$

```
void quickSort(long[] arr, int pi, int pf) {
    if (pi >= pf) return;
    long piv = arr[(pi + pf) / 2]; int i = pi, j = pf;
    for (; i <= j; i++, j--) {
        while (arr[i] < piv) i++;
        while (arr[j] > piv) j--;
        if (i > j) break; if (i != j) {long tmp = arr[i]; arr[i] = arr[j]; arr[j] = tmp;}
    }
    quickSort(arr, pi, j); quickSort(arr, i, pf);
}
```

☒ Insertion Sort $\{O(n^2)\}$

```
void insertionSort(long[] arr, int pi, int pf) {
    for (int i = pi + 1; i <= pf; i++)
        {int j = i; long v = arr[i]; for (; j > pi && arr[j - 1] > v; j--) arr[j] = arr[j - 1]; arr[j] = v;}
}
```

☒ Bubble Sort $\{O(n^2)\}$

```
void bubbleSort(long[] arr, int pi, int pf) {
    for (int i = pi + 1; i <= pf; i++) for (int j = pf; j >= i; j--) if (arr[j - 1] > arr[j])
        {long tmp = arr[j - 1]; arr[j - 1] = arr[j]; arr[j] = tmp;}
}
```

☒ Binary Search $\{\log_2(n)\}$

```
int busquedaBinaria(long[] a, int lI, int lS, long v) {
    while (lI <= lS)
        {int lM = (lI + lS) / 2; long p = a[lM]; if (p < v) lI = lM + 1; else if (p > v) lS = lM - 1; else return lM;}
    return -(lI + 1);
}
```

☒ Búsqueda en silla

☒ k-ésimo menor – Quickselect [C. A. R. Hoare (1960)] $\{\text{expected: } O(n); \text{worst: } O(n^2)\}$


```
<T> void swap(T[] arr, int i, int j) {T tmp=arr[i]; arr[i]=arr[j]; arr[j]=tmp;}
<T extends Comparable<T>> T select(T[] arr, int pI, int pF, int k) {
    while (true) {
        int s=pI; swap(arr,(pI+pF)/2,pF); T piv=arr[pF];
        for (int j=pI; j<pF; j++) if (arr[j].compareTo(piv)<0) swap(arr,s++,j);
        swap(arr,s,pF); if (k==s) return arr[k]; if (k<s) pF=s-1; else pI=s+1;
    }
}
```

☑ **Mínimo número de swaps entre elementos adyacentes para ordenar un arreglo sin repetidos (número de inversiones de una permutación) $\{O(n \log_2(n))\}$**

En mergeSort, cambiar "arrTmp[k]=arr[i]<=arr[j]?arr[i++]:arr[j++];" por
 "{if (arr[i]<=arr[j]) arrTmp[k]=arr[i++]; else {numSwaps+=j-k; arrTmp[k]=arr[j++];}}"
 declarando fuera del método una variable numSwaps que se inicialice cuando corresponda.

☑ **Mínimo número de swaps para ordenar un arreglo sin repetidos $\{O(n \log_2(n))\}$**

```
int minSwaps(long[] arr) {
    int n=arr.length,res=0; long m[][]=new long[n][n]; boolean used[]=new boolean[n];
    for (int i=0; i<n; i++) m[i]=new long[]{arr[i],i};
    Arrays.sort(m,new Comparator<long[]>() {
        public int compare(long[] a1, long[] a2)
        {return a1[0]!=a2[0]?(a1[0]<a2[0]?-1:1):(a1[1]!=a2[1]?(a1[1]<a2[1]?-1:1):0);}
    });
    for (int i=0,c,j; i<n; i++) if (!used[i])
    {for (j=i,c=0; !used[j]; used[j]=true,j=(int)(m[j][1]),c++); res+=c-1;}
    return res;
}
```

☑ **Rotar matriz a la derecha $\{O(n*m)\}$**

```
long[][] rotarMatrizDer(long[][] mat) {
    int n=mat.length,m=n==0?0:mat[0].length; long res[][]=new long[m][n];
    for (int i=0; i<n; i++) for (int j=0; j<m; j++) res[j][i]=mat[n-1-i][j];
    return res;
}
```

☑ **Rotar matriz a la izquierda $\{O(n*m)\}$**

```
long[][] rotarMatrizIzq(long[][] mat) {
    int n=mat.length,m=n==0?0:mat[0].length; long res[][]=new long[m][n];
    for (int i=0; i<n; i++) for (int j=0; j<m; j++) res[j][i]=mat[i][m-1-j];
    return res;
}
```

☑ **Permutaciones de una bolsa $\{O(n!/(n_1! * n_2! * \dots * n_k!))\}$**

```
void permutaciones(char[] arr, int[] frequencySymbols, char[] symbols, int k) {
    if (k==arr.length) {System.out.println(new String(arr)); return;}
    for (int i=0; i<symbols.length; i++) if (frequencySymbols[i]>0) {
        frequencySymbols[i]--; arr[k]=symbols[i];
        permutaciones(arr,frequencySymbols,symbols,k+1);
        frequencySymbols[i]++; arr[k]='\0';
    }
}
```

☑ **k-ésima permutación en orden lexicográfico $\{O(n^2)\}$**

```
char[] permutation(int k, char[] s) {
    int n=s.length,fact=1,i,j,tj; char ts; s=s.clone();
    for (j=2; j<n; j++) fact*=j;
    for (j=1; j<n; j++) {
        tj=(k/fact)%(n+1-j); ts=s[j+tj-1]; for (i=j+tj; i>j; i--) s[i-1]=s[i-2];
        s[j-1]=ts; fact/=n-j;
    }
    return s;
}
```

ÁLGEBRA LINEAL

☑ **Multipliación de matrices cuadradas $\{O(n^3)\}$**

```
void mmult(double[][] a, double[][] b, double[][] answer) { // answer=a*b
    for (int i=0,n=a.length; i<n; i++) for (int j=0; j<n; j++)
    {double r=0d; for (int k=0; k<n; k++) r+=a[i][k]*b[k][j]; answer[i][j]=r;}
}
```

☑ **Multiplicación de matrices cuadradas - Strassen [V. Strassen (1969)]** $\{O(n^{\log_2(7)})\}$

☑ **Solución sistemas lineales - Eliminación de Gauss-Jordan [Jiuzhang suanshu (179)]** $\{O(n^3)\}$

double gaussJordan(double[][] a) { // a puede ser una matriz extendida.

```
double det=1d,tmp[],z; int i,u,v,w,n=a.length,m=n==0?0:a[0].length;
for (i=0; i<n; i++) {
    for (w=i,u=i+1; u<n; u++) if (Math.abs(a[u][i])>Math.abs(a[w][i])) w=u;
    if (Math.abs(a[w][i])<1e-12) return 0d;
    if (w!=i) {tmp=a[w]; a[w]=a[i]; a[i]=tmp; det*=-1d;}
    for (v=i, det*=z=a[i][i]; v<m; v++) a[i][v]/=z;
    for (u=0; u<n; u++) if (u!=i&&Math.abs(z=a[u][i])>=1e-12)
        for (v=i; v<m; v++) a[u][v]-=z*a[i][v];
}
return det;
}
```

☑ **Solución sistemas lineales - Descomposición QR mediante reflexiones de Householder** $\{O(n^3)\}$

// <http://www.docjar.com/html/api/jmat/data/matrixDecompositions/QRDecomposition.java.html>

```
double[] solveQR(double[][] mat, double[] b) { // Least squares solution of mat*X=b
    int m=b.length,i; double nb[][]=new double[m][1],r[]=new double[m];
    for (i=0; i<m; i++) nb[i][0]=b[i]; nb=solveQR(mat,nb); for (i=0; i<m; i++) r[i]=nb[i][0];
    return r;
}
```

```
double[][] solveQR(double[][] mat, double[][] b) { // Least squares solution of mat*X=b
    int m=mat.length,n=m==0?0:mat[0].length,i,j,k; // m>=n, b.length==m
    double qr[][]=new double[m][n],rdiag[]=new double[n],nrm,s;
```

```
for (i=0; i<m; i++) qr[i]=mat[i].clone();
for (k=0; k<n; k++) {
    for (nrm=0,i=k; i<m; i++) nrm+=qr[i][k]*qr[i][k];
    if (Math.abs(nrm=Math.sqrt(nrm))>1e-15) {
        if (qr[k][k]<0) nrm*=-1;
        for (i=k; i<m; i++) qr[i][k]/=nrm;
        for (qr[k][k]+=1.0,j=k+1; j<n; j++) {
            for (s=0d,i=k; i<m; i++) s+=(qr[i][k]*qr[i][j]);
            for (s=-s/qr[k][k],i=k; i<m; i++) qr[i][j]+=(s*qr[i][k]);
        }
    }
    if (Math.abs(rdiag[k]==-nrm)<1e-15) return null;
}
```

```
int nx=m==0?0:b[0].length; double[][] x=new double[m][nx],res=new double[n][nx];
for (i=0; i<m; i++) x[i]=b[i].clone();
for (k=0; k<n; k++) for (j=0; j<nx; j++) {
    for (s=0d,i=k; i<m; i++) s+=(qr[i][k]*x[i][j]);
    for (s=-s/qr[k][k],i=k; i<m; i++) x[i][j]+=(s*qr[i][k]);
}
for (k=n-1; k>=0; k--) {
    for (j=0; j<nx; j++) x[k][j]/=rdiag[k];
    for (i=0; i<k; i++) for (j=0; j<nx; j++) x[i][j]-=(x[k][j]*qr[i][k]);
}
for (k=0; k<n; k++) for (j=0; j<nx; j++) res[k][j]=x[k][j];
return x;
}
```

☑ **Inversa de una matriz de Vandermonde - Algoritmo de Traub [J. Traub (1966)]** $\{O(n^2)\}$

```
double[][] vandermondeInverse(double[] e) { // vandermondeMatrix[i][j]=e[i]^(j-1)
    int n=e.length,k,i,j;
    double a[]=new double[n+1],b[]=new double[n+1],inv[][]=new double[n][n],p,q,r;
    for (a[0]=-e[0],a[1]=1,k=1; k<n; System.arraycopy(b,0,a,0,k+2),k++)
        for (i=0; i<=k+1; i++) b[i]=(i>0?a[i-1]:0)-e[k]*(i<=k?a[i]:0);
    for (j=0; j<n; j++) {
        for (p=e[j],q=inv[n-1][j]=1,r=a[1],k=1; k<n; r+=a[k+1]*(k+1)*p,k++,p*=e[j])
            q=inv[n-1-k][j]=e[j]*q+a[n-k];
        for (k=0; k<n; k++) inv[k][j]/=r;
    }
    return inv;
}
```

☑ **Polynomial interpolation - Algoritmo de Traub [J. Traub (1966)]** $\{O(n^2)\}$

```
double[] polinomialInterpolationTraub(double[] x, double[] y) {
    int n=x.length,i,j; double vi[][]=vandermondeInverse(x),a[]=new double[n];
    for (i=0; i<n; i++) for (j=0; j<n; j++) a[i]+=vi[i][j]*y[j];
    return a; // f(x)=a[0]+a[1]*x+...+a[i]*x^i+...+a[n]*x^n pasa por los puntos dados
}
☑ Polynomial interpolation - Algoritmo de Björck-Pereyra [A. Björck, V. Pereyra (1970)]  $\{O(n^2)\}$ 
double[] polinomialInterpolationBjorckPereyra(double[] x, double[] y) {
    int n=x.length,i,k; double[] a=y.clone();
    for (k=1; k<n; k++) for (i=n-1; i>=k; i--) a[i]=(a[i]-a[i-1])/(x[i]-x[i-k]);
    for (k=n-1; k>=1; k--) for (i=k-1; i<n-1; i++) a[i]-=a[i+1]*x[k-1];
    return a; // f(x)=a[0]+a[1]*x+...+a[i]*x^i+...+a[n]*x^n pasa por los puntos dados
}
☑ Regresión lineal - Mínimos cuadrados (lineal)  $\{O(n)\}$ 
double[] regresionLineal(double[][] pt) {
    int n=pt.length; double x,y,sX1=0d,sX2=0d,sY1=0d,sY2=0d,sXY=0d;
    for (double[] p:pt) {x=p[0]; y=p[1]; sX1+=x; sX2+=x*x; sY1+=y; sY2+=y*y; sXY+=x*y;}
    double xp=sX1/n,yp=sY1/n,ssxx=sX2-n*xp*xp,ssyy=sY2-n*yp*yp,ssxy=sXY-n*xp*yp;
    double b=ssxy/ssxx,a=yp-b*xp,s=Math.sqrt((ssyy-b*ssxy)/(n-2));
    double seA=s*Math.sqrt(1d/n*xp*xp/ssxx),seB=s/Math.sqrt(ssxx);
    return new double[]{a,b,seA,seB}; // y=a+b*x con errores estándar seA y seB
}
☑ Regresión polinomial - Mínimos cuadrados (polinomial)  $\{O((n*\text{grado}^2)\uparrow\text{grado}^3)\}$ 
double[] regresionPolinomial(double[][] pt, int grado) {
    int n=pt.length,m=grado+1,i,j,k; double mat[][]=new double[m][m],b[]=new double[m];
    for (i=0; i<m; i++) for (j=0; j<m; j++) for (k=0; k<n; k++) mat[i][j]+=Math.pow(pt[k][0],i+j);
    for (i=0; i<m; i++) for (k=0; k<n; k++) b[i]+=Math.pow(pt[k][0],i)*pt[k][1];
    double[] a=solveQR(mat,b);
    return a; // f(x)=a[0]+a[1]*x+...+a[i]*x^i+...+a[n]*x^n es el polinomio resultado
}

```

GRAFOS

**GRAFOS NO DIRIGIDOS TIENEN SIMÉTRICA SU MATRIZ DE ADYACENCIA
LA DIAGONAL DE LAS MATRICES DE ADYACENCIA DEBE CONTENER CEROS
EN LAS COMPLEJIDADES, V ES EL NÚMERO DE NODOS Y E ES EL NÚMERO DE ARCOS**

```
☑ Convertidor de matrices de adyacencia a listas de adyacencia  $\{O(V^2)\}$ 
int[][] to_lAdy(double[][] mAdy) {
    int n=mAdy.length,lAdy[][]=new int[n][],u,v;
    for (u=0; u<n; u++) {
        List<Integer> p=new ArrayList<Integer>();
        for (v=0; v<n; v++) if (u!=v&&mAdy[u][v]!=Double.POSITIVE_INFINITY) p.add(v);
        lAdy[u]=toArr(p);
    }
    return lAdy;
}
int[] toArr(List<Integer> p) {int r[]=new int[p.size()],i=0; for (int x:p) r[i++]=x; return r;}
☑ BFS (Breadth-First Search)  $\{O(V+E)\}$ 
int[] bfs(int[][] lAdy, int v) {
    int n=lAdy.length,res[]=new int[n],queue[]=new int[n],t=0,inf=Integer.MAX_VALUE;
    Arrays.fill(res,inf); res[queue[t++]=v]=0;
    for (int i=0; i<t; i++)
        {v=queue[i]; for (int w:lAdy[v]) if (res[w]==inf) res[queue[t++]=w]=res[v]+1;}
    return res;
}
☑ DFS (Depth-First Search)  $\{O(V+E)\}$ 
Véase: Sort topológico
☑ TSP (Traveling Salesman Problem)  $\{O(V^2*2^V)\}$ 
double tsp(double[][] mAdy, int v) {
    int n=mAdy.length,t=1<n; double mem[][]=new double[t][n];
    for (double[] arr:mem) Arrays.fill(arr,-1d); return tsp(mAdy,n,v,v,1<<v,mem);
}
double tsp(double[][] mAdy, int n, int v1, int v2, int visitados, double[][] mem) {
    if (mem[visitados][v1]>=0d) return mem[visitados][v1];
    if (visitados==(1<n)-1) return mem[visitados][v1]=mAdy[v1][v2];
}

```

```

double min=Double.POSITIVE_INFINITY,d;
for (int e=visitados,j=0; j<n; j++,e>>=1) if ((e&1)==0&&(d=mAdy[v1][j])<min)
    min=Math.min(min,d+tsp(mAdy,n,j,v2,visitados|(1<<j),mem));
return mem[visitados][v1]=min;
}

```

☑ **Rutas más cortas con caminos de longitud menor o igual que q $\{O(V^3 \cdot \log_2(q))\}$**

```

void mmult(double[][] a, double[][] b, double[][] tmp) { // a*=b
    int n=a.length;
    for (int i=0; i<n; i++) for (int j=0; j<n; j++) {
        double r=Double.POSITIVE_INFINITY;
        for (int k=0; k<n; k++) if (a[i][k]<r&&b[k][j]<r) r=Math.min(r,a[i][k]+b[k][j]);
        tmp[i][j]=r;
    }
    for (int i=0; i<n; i++) System.arraycopy(tmp[i],0,a[i],0,n);
}

```

```

double[][] costoCaminosQ(double[][] mAdy, int q) {
    int n=mAdy.length,y=q; double[][] x=new double[n][n],z=new double[n][n],u=new double[n][n];
    for (int i=0; i<n; i++)
        {Arrays.fill(z[i],Double.POSITIVE_INFINITY); System.arraycopy(mAdy[i],0,x[i],0,n); z[i][i]=0;}
    while (y!=0) {if ((y&1)==0) {y/=2; mmult(x,x,u);} else {y--; mmult(z,x,u);}}
    return z;
}

```

☑ **Rutas más cortas - Floyd-Warshall $\{O(V^3)\}$**

```

double[][] floydWarshall(double[][] mAdy) {
    int n=mAdy.length; double x[][]=new double[n][n];
    for (int i=0; i<n; i++) System.arraycopy(mAdy[i],0,x[i],0,n);
    for (int k=0; k<n; k++) for (int i=0; i<n; i++) for (int j=0; j<n; j++)
        x[i][j]=Math.min(x[i][j],x[i][k]+x[k][j]);
    return x;
}

```

☑ **Minimax - Floyd-Warshall $\{O(V^3)\}$**
Ponga $x[i][j]=\text{Math.min}(x[i][j], \text{Math.max}(x[i][k], x[k][j]))$ en el algoritmo de Floyd-Warshall.

☑ **Maximin - Floyd-Warshall $\{O(V^3)\}$**
Ponga $x[i][j]=\text{Math.max}(x[i][j], \text{Math.min}(x[i][k], x[k][j]))$ en el algoritmo de Floyd-Warshall.

☑ **Safest path - Floyd-Warshall $\{O(V^3)\}$**
Ponga $x[i][j]=\text{Math.max}(x[i][j], x[i][k]*x[k][j])$ en el algoritmo de Floyd-Warshall (los costos de los arcos representan probabilidades de supervivencia).

☑ **Clausura transitiva - Floyd-Warshall $\{O(V^3)\}$**

```

boolean[][] floydWarshall(boolean[][] mAdy) {
    int n=mAdy.length,i,j,k; boolean x[][]=new boolean[n][n];
    for (i=0; i<n; i++) for (j=0; j<n; j++) x[i][j]=i==j||mAdy[i][j]!=Double.POSITIVE_INFINITY;
    for (k=0; k<n; k++) for (i=0; i<n; i++) for (j=0; j<n; j++)
        x[i][j]=x[i][j]|| (x[i][k]&& x[k][j]);
    return x;
}

```

☑ **Rutas más cortas desde un nodo - Dijkstra [E. Dijkstra (1959)] $\{O(\log_2(V) \cdot (V+E))\}$**

```

double[] dijkstra(double[][] mAdy, int[][] lAdy, int v) {
    TreeMap<Double,List<Integer>> map=new TreeMap<Double,List<Integer>>();
    int n=mAdy.length; double mins[]=new double[n]; boolean[] vis=new boolean[n];
    Arrays.fill(mins,Double.POSITIVE_INFINITY);
    mins[v]=0d; map.put(0d,new ArrayList<Integer>(Arrays.asList(v)));
    while (map.size()>0) for (int f:map.pollFirstEntry().getValue()) if (!vis[f]) {
        vis[f]=true;
        for (int g:lAdy[f]) if (!vis[g]&&mins[f]+mAdy[f][g]<mins[g]-1e-11) {
            mins[g]=mins[f]+mAdy[f][g]; List<Integer> p=map.get(mins[g]);
            if (p==null) map.put(mins[g],p=new ArrayList<Integer>()); p.add(g);
        }
    }
    return mins;
}

```

☑ **Rutas más cortas desde un nodo - Dijkstra (Priority Queue) $\{O(\log_2(V) \cdot (V+E))\}$**

```

double[] dijkstraPriorityQueue(double[][] mAdy, int[][] lAdy, int v) {
    class Nodo implements Comparable<Nodo> {
        int i; double d; Nodo(int pi, double pd) {i=pi; d=pd;}
    }
}

```

```

    public int compareTo(Nodo e) {return d!=e.d?(d<e.d?-1:1):i-e.i;}
};
PriorityQueue<Nodo> pq=new PriorityQueue<Nodo>(lAdy.length);
int n=mAdy.length,f; double mins[]=new double[n]; boolean[] vis=new boolean[n];
Arrays.fill(mins,Double.POSITIVE_INFINITY);
mins[v]=0d; pq.add(new Nodo(v,0d));
while (!pq.isEmpty()) if (!vis[f=pq.poll().i]) {
    vis[f]=true;
    for (int g:lAdy[f]) if (!vis[g]&&mins[f]+mAdy[f][g]<mins[g]-1e-11)
        pq.add(new Nodo(g,mins[g]=mins[f]+mAdy[f][g]));
}
return mins;
}
}
☑ Rutas más cortas desde un nodo (costos negativos) - Bellman-Ford  $\{O(V^3E)\}$ 
double[] bellmanFord(double[][] mAdy, int[][] lAdy, int v) {
    int n=mAdy.length; double[] mins=new double[n];
    Arrays.fill(mins,Double.POSITIVE_INFINITY); mins[v]=0d;
    for (int k=1; k<n; k++) for (int i=0; i<n; i++) for (int j:lAdy[i])
        mins[j]=Math.min(mins[j],mins[i]+mAdy[i][j]);
    for (int i=0; i<n; i++) for (int j:lAdy[i])
        if (mins[j]>mins[i]+mAdy[i][j]+1e-10) return null; // Ciclo de peso negativo
    return mins;
}
}
☑ Rutas más cortas desde un nodo (costos enteros entre 1 y M) - Bucket Priority Queue  $\{O(V^2M+E)\}$ 
int[] dijkstraE(int[][] mAdy, int[][] lAdy, int v, int M) { // Apropiado para  $1 \leq M \leq 70$ 
    int n=mAdy.length,b[][]=new int[M+1][n],tamB[]=new int[M+1],res[]=new int[n],d,i,j,t,q,sum=1;
    boolean vis[]=new boolean[n]; Arrays.fill(res,Integer.MAX_VALUE);
    for (res[v]=0,b[0][tamB[0]++]=v,d=0; sum>0; d++) {
        for (i=0,t=tamB[0]; i<t; i++) if (!vis[v=b[0][i]]) {
            vis[v]=true;
            for (int w:lAdy[v]) if (d+(q=mAdy[v][w])<res[w]) {res[w]=d+q; b[q][tamB[q]++]=w;}
        }
        int arrTmp[]=b[0];
        for (j=1,sum=0; j<=M; j++) {b[j-1]=b[j]; sum+=(tamB[j-1]=tamB[j]);}
        b[M]=arrTmp; tamB[M]=0;
    }
    return res;
}
}
☑ Árbol de expansión minimal - Kruskal [J. Kruskal (1956)]  $\{O(E \cdot \log_2(V))\}$ 
boolean[] kruskal(final double[][] mAdy, int[][] lAdy) { // Para grafos conexos no dirigidos
    int n=mAdy.length,t=0; for (int[] a:lAdy) t+=a.length; int edges[][]=new int[t][2],k=0;
    for (int i=0; i<n; i++) for (int j:lAdy[i]) if (j>i) edges[k++]={i,j,k};
    Arrays.sort(edges,0,k,new Comparator<int[]>() {public int compare(int[] u, int[] v)
    {double r=mAdy[u[0]][u[1]]-mAdy[v[0]][v[1]]; return r!=0?(r<0?-1:1):u[2]-v[2];}});
    DisjointSet forest[]=new DisjointSet[n]; for (int i=0; i<n; i++) forest[i]=new DisjointSet();
    boolean[][] res=new boolean[n][n];
    for (int e[]:Arrays.copyOfRange(edges,0,k))
        if (DisjointSet.find(forest[e[0]])!=DisjointSet.find(forest[e[1]]))
            {res[e[0]][e[1]]=res[e[1]][e[0]]=true; DisjointSet.union(forest[e[0]],forest[e[1]]);}
    return res;
}
}
☑ Árbol de expansión minimal - Prim [V. Jarník (1930), R. C. Prim (1957)]  $\{O(E \cdot V)\}$ 
boolean[] prim(double[][] mAdy, int[][] lAdy) { // Para grafos conexos no dirigidos
    int n=mAdy.length,k,i,ie,je; boolean res[][]=new boolean[n][n],vis[]=new boolean[n];
    for (vis[0]=true,k=1; k<n; k++) {
        double me=Double.POSITIVE_INFINITY;
        for (i=0,ie=je=-1; i<n; i++) if (vis[i]) for (int j:lAdy[i]) if (!vis[j]&&mAdy[i][j]<me)
            {ie=i; je=j; me=mAdy[i][j];}
        res[ie][je]=res[je][ie]=vis[je]=true;
    }
    return res;
}
}
☑ Árbol de expansión minimal (grafos dirigidos) - Chu-Liu/Edmonds  $\{O(V^2)\}$ 
☑ Existencia de un ciclo/camino Euleriano (grafos dirigidos)  $\{O(V+E)\}$ 

```

Un grafo dirigido $G=\langle V,E \rangle$ tiene un ciclo Euleriano si es conexo y todo vértice v cumple que $\text{inDegree}(v)=\text{outDegree}(v)$. Un grafo dirigido $G=\langle V,E \rangle$ tiene un camino Euleriano si es conexo y todo vértice v cumple que $\text{inDegree}(v)=\text{outDegree}(v)$, excepto dos vértices v_1 y v_2 para los que se tiene que $\text{inDegree}(v_1)=\text{outDegree}(v_1)-1$ y $\text{inDegree}(v_2)=\text{outDegree}(v_2)+1$.

☑ **Ciclo/Camino Euleriano partiendo de un nodo (grafos dirigidos) $\{O(V+E)\}$**

```
int[] euler(int[][] lAdy, int v) { // Lo encuentra si existe
    List<Integer> r=new ArrayList<Integer>(); euler(lAdy,new int[lAdy.length],v,r);
    int t=r.size(),a[]=new int[t],i; for (i=0; i<t; i++) a[i]=r.get(t-1-i); return a;
}
```

```
void euler(int[][] lAdy, int[] tams, int v, List<Integer> r)
{while (tams[v]<lAdy[v].length) euler(lAdy,tams,lAdy[v][tams[v++]],r); r.add(v);}
```

☑ **Existencia de un ciclo/camino Euleriano (grafos no dirigidos) $\{O(V+E)\}$**

Un grafo no dirigido $G=\langle V,E \rangle$ tiene un ciclo Euleriano si es conexo y todo vértice v tiene grado par. Un grafo no dirigido $G=\langle V,E \rangle$ tiene un camino Euleriano si es conexo y todo vértice v tiene grado par, excepto dos vértices que tienen grado impar.

☑ **Ciclo/Camino Euleriano partiendo de un nodo (grafos no dirigidos) $\{O(V+E)\}$**

```
int[] eulerND(int[][] lAdy, int v) { // Lo encuentra si existe
    int n=lAdy.length; List<Integer> r=new ArrayList<Integer>();
    eulerND(lAdy,new int[n],new boolean[n][n],v,r);
    int t=r.size(),a[]=new int[t],i; for (i=0; i<t; i++) a[i]=r.get(t-1-i); return a;
}
```

```
void eulerND(int[][] lAdy, int[] tams, boolean[][] vis, int v, List<Integer> r) {
    for (int x; tams[v]<lAdy[v].length; ) if (!vis[v][x=lAdy[v][tams[v++]])
        {vis[v][x]=vis[x][v]=true; eulerND(lAdy,tams,vis,x,r);}
    r.add(v);
}
```

☑ **Ciclo Hamiltoniano $\{O(E^V)\}$**

```
int[] hamilton(int[][] lAdy) {
    int n=lAdy.length,cam[]=new int[n+1]; if (!hamilton(lAdy,new boolean[n],cam,0,0)) return null;
    return cam;
}
boolean hamilton(int[][] lAdy, boolean[] vis, int[] cam, int t, int v) {
    if (vis[v]&&t<cam.length-1) return false; cam[t++]=v; if (t==cam.length) return v==cam[0];
    vis[v]=true; for (int w:lAdy[v]) if (hamilton(lAdy,vis,cam,t,w)) return true;
    vis[v]=false; return false;
}
```

☑ **Sort topológico (Topological Sort) $\{O(V+E)\}$**

```
int[] topologicalSort(int[][] lAdy) { // Retorna null si no recibe un DAG
    int n=lAdy.length,state[]=new int[n]; List<Integer> r=new ArrayList<Integer>();
    for (int v=0; v<n; v++) if (state[v]==0&&!dfsTS(lAdy,v,state,r)) return null;
    int t=r.size(),a[]=new int[t],i; for (i=0; i<t; i++) a[i]=r.get(t-1-i); return a;
}
```

```
boolean dfsTS(int[][] lAdy, int v, int[] state, List<Integer> r) {
    state[v]=1;
    for (int w:lAdy[v]) if (state[w]==1||(state[w]==0&&!dfsTS(lAdy,w,state,r))) return false;
    state[v]=2; r.add(v); return true;
}
```

☑ **Decidir si un grafo es un DAG (Directed Acyclic Graph) $\{O(V+E)\}$**

```
boolean isDag(int[][] lAdy) {return topologicalSort(lAdy)!=null;}
```

☑ **Caminos más largos en un DAG $\{O(V+E)\}$**

```
int[] caminosMasLargosDAG(int[][] lAdy) {
    int n=lAdy.length,res[]=new int[n];
    for (int v:topologicalSort(lAdy)) for (int u:lAdy[v]) res[u]=Math.max(res[u],res[v]+1);
    return res;
}
```

☑ **Rutas más cortas desde un nodo en un DAG $\{O(V+E)\}$**

```
double[] rutasMasCortasDAG(double[][] mAdy, int[][] lAdy, int v) {
    int n=lAdy.length; double r[]=new double[n]; Arrays.fill(r,Double.POSITIVE_INFINITY); r[v]=0;
    // Para considerar varias fuentes  $v_1, \dots, v_k$  inicialice  $r[v_1], \dots, r[v_k]$  en 0.
    for (int w:topologicalSort(lAdy)) for (int u:lAdy[w]) r[u]=Math.min(r[u],r[w]+mAdy[w][u]);
    return r;
}
```

☑ **Minimum path cover en un DAG**

☑ **Strongly connected components en grafos dirigidos – Kosaraju [R. Kosaraju (1978)] $\{O(V+E)\}$**


```

int[][] stronglyConnectedComponents(int[][] lAdy) {
    int n=lAdy.length,lAdyT[][]=new int[n][],tams[]=new int[n]; boolean vis[]=new boolean[n];
    List<Integer> r=new ArrayList<Integer>(),s=new ArrayList<Integer>();
    for (int v=0; v<n; v++) if (!vis[v]) dfsSCC(lAdy,v,vis,r);
    for (int[] arr:lAdy) for (int j:arr) tams[j]++;
    for (int j=0; j<n; tams[j]=0,j++) lAdyT[j]=new int[tams[j]];
    for (int i=0; i<n; i++) for (int j:lAdy[i]) lAdyT[j][tams[j]++]=i;
    List<int[]> res=new ArrayList<int[]>(); Collections.reverse(r); Arrays.fill(vis,false);
    for (int v:r) if (!vis[v]) {
        s.clear(); dfsSCC(lAdyT,v,vis,s);
        int a[]=new int[s.size()],i=0; for (int x:s) a[i++]=x; res.add(a);
    }
    return res.toArray(new int[0][]);
}

void dfsSCC(int[][] lAdy, int v, boolean[] vis, List<Integer> r)
{vis[v]=true; for (int w:lAdy[v]) if (!vis[w]) dfsSCC(lAdy,w,vis,r); r.add(v);}

☑ Strongly connected components en grafos dirigidos - Tarjan [R. Tarjan (1972?)] {O(V+E)}
☑ Bridges en grafos dirigidos {O(V+E)}
// Un puente es un arco que al quitarse desconecta al grafo
int cnt,ord[],low[];
int[][] bridges(int[][] lAdy) { // Tomado de [Sed2004]
    int n=lAdy.length; ord=new int[n]; low=new int[n]; List<int[]> res=new ArrayList<int[]>();
    for (int v=0; v<n; v++) if (ord[v]==0) {cnt=1; bridges(v,v,lAdy,res);}
    return res.toArray(new int[0][]);
}

void bridges(int v, int w, int[][] lAdy, List<int[]> res) {
    low[w]=ord[w]=cnt++;
    for (int t:lAdy[w])
        if (ord[t]==0) {
            bridges(w,t,lAdy,res); low[w]=Math.min(low[w],low[t]);
            if (low[t]==ord[t]) res.add(new int[]{w,t});
        }
        else if (t!=v) low[w]=Math.min(low[w],ord[t]);
}

☑ Articulation Points en grafos dirigidos {O(V+E)}
// Un punto de articulación es un vértice que al quitarse desconecta al grafo
int cnt,ord[],low[];
boolean[] articulationPoints(int[][] lAdy) { // Basado en [Sed2004]
    int n=lAdy.length; ord=new int[n]; low=new int[n]; boolean[] res=new boolean[n];
    for (int v=0; v<n; v++) if (ord[v]==0) {cnt=1; articulationPoints(v,v,lAdy,res);}
    return res;
}

void articulationPoints(int v, int w, int[][] lAdy, boolean[] res) {
    low[w]=ord[w]=cnt++;
    for (int t:lAdy[w]) {
        if (ord[t]==0) {
            articulationPoints(w,t,lAdy,res); low[w]=Math.min(low[w],low[t]);
            if ((ord[w]==1&&ord[t]!=2)||((ord[w]!=1&&low[t]>=ord[w])) res[w]=true;
        }
        else if (t!=v) low[w]=Math.min(low[w],ord[t]);
    }
}

☑ Biconnected Components en grafos dirigidos {O(V+E)}
// Descomposición de un grafo en componentes maximales que no tienen puntos de articulación
// (hay mínimo dos caminos disyuntos entre todo par de vértices en cada componente biconnectada)
int cnt,ord[],low[],stack[][],tS;
int[][][] biconnectedComponents(int[][] lAdy) { // Basado en [Sed2004]
    int n=lAdy.length; ord=new int[n]; low=new int[n]; stack=new int[n][]; tS=0;
    List<int[][]> res=new ArrayList<int[][]>();
    for (int v=0; v<n; v++) if (ord[v]==0) {cnt=1; tS=0; biconnectedComponents(v,v,lAdy,res);}
    return res.toArray(new int[0][][]);
}

boolean cmp(int[] a, int[] b) {return (a[0]==b[0]&&a[1]==b[1])||(a[0]==b[1]&&a[1]==b[0]);}
void biconnectedComponents(int v, int w, int[][] lAdy, List<int[][]> res) {

```

```

low[w]=ord[w]=cnt++;
for (int t:lAdy[w]) if (t!=v) {
    if (ord[t]<ord[w]) stack[tS++]=new int[]{w,t};
    if (ord[t]==0) {
        biconnectedComponents(w,t,lAdy,res); low[w]=Math.min(low[w],low[t]);
        if (low[t]>=ord[w]) {
            int tvS=tS,e[]={w,t};
            while (tS>=1&&!cmp(stack[tS-1],e)) tS--;
            if (tS>=1) tS--; res.add(Arrays.copyOfRange(stack,tS,tvS));
        }
    }
    else low[w]=Math.min(low[w],ord[t]);
}
}
}

```

☑ **Determinar si un grafo es bipartito (bicoloreable) - BFS $\{O(V+E)\}$**

```

boolean grafoBipartitoBFS(int[][] lAdy) {
    int n=lAdy.length,res[]=new int[n],queue[]=new int[n],i,t,u,v,inf=Integer.MAX_VALUE;
    for (Arrays.fill(res,inf),u=0; u<n; u++) if (res[u]==inf) {
        t=0; res[queue[t++]]=u=0;
        for (i=0; i<t; i++) for (int w:lAdy[v=queue[i]])
            if (res[w]==inf) res[queue[t++]]=w=res[v]+1; else if ((res[w]+res[v])%2==0) return false;
    }
    return true;
}

```

☑ **Determinar si un grafo es bipartito (bicoloreable) - DFS $\{O(V+E)\}$**

```

boolean grafoBipartitoDFS(int[][] lAdy) {
    int n=lAdy.length,cs[]=new int[n];
    for (int v=0; v<n; v++) if (cs[v]==0&&!dfsGP(lAdy,v,cs,1)) return false;
    return true;
}
boolean dfsGP(int[][] lAdy, int v, int[] cs, int c) { // c: color
    cs[v]=c; c=3-c;
    for (int w:lAdy[v]) if ((cs[w]!=0&&cs[w]!=c)|| (cs[w]==0&&!dfsGP(lAdy,w,cs,c))) return false;
    return true;
}

```

☑ **Teorema de König para grafos bipartitos - [D. König (1914)]**

En cualquier grafo bipartito, el número de arcos en un maximum matching es igual al número de vértices en un minimum vertex cover y el número de vértices en un minimum vertex cover es igual al número total de vértices menos el número de vértices en un maximum independent set (un vertex cover en un grafo no dirigido $G=\langle V,E \rangle$ es un subconjunto de V tal que todo arco en E tiene un extremo en V ; un matching en un grafo $G=\langle V,E \rangle$ es un subconjunto de arcos que no comparten extremos dos a dos; un independent set en un grafo $G=\langle V,E \rangle$ es un conjunto de vértices que no son adyacentes).

☑ **Maximum matching en grafos bipartitos no dirigidos**

Divida el conjunto de vértices V del grafo bipartito en dos conjuntos disyuntos V_1 y V_2 tales que todo arco en E conecte un nodo en V_1 con un nodo en V_2 . Cree un nodo n_1 conectado a todos los vértices de V_1 y un nodo n_2 conectado a todos los vértices de V_2 . El maximum matching está dado por el flujo máximo entre los nodos n_1 y n_2 donde todos los arcos tienen capacidad 1.

☑ **Teorema de Dilworth para Posets (Partially Ordered Sets) - [R. P. Dilworth (1950)]**

En cualquier conjunto parcialmente ordenado, el número de elementos de la anticadena más grande es igual al tamaño de la partición más pequeña del conjunto en una familia de cadenas.

☑ **Maximum independent sets en árboles**

FLUJO EN REDES

☑ **Flujo máximo - Ford-Fulkerson/Edmonds-Karp [J. Edmonds, R. Karp (1972)] $\{O(V \cdot E^2)\}$**

```

double edmondsKarp(double[][] capacity, int v1, int v2) { // residualCapacity=capacity-flow
    int n=capacity.length,lAdy[][]=new int[n][n],ants[]=new int[n],queue[]=new int[n],v,u;
    double f=0,d,m,flow[][]=new double[n][n],minCap[]=new double[n]; List h[]=new List[n];
    for (u=0; u<n; u++) h[u]=new ArrayList<Integer>();
    for (u=0; u<n; u++) for (v=0; v<n; v++) if (capacity[u][v]>1e-10) {h[u].add(v); h[v].add(u);}
    for (u=0; u<n; u++) lAdy[u]=toArr(h[u]);
    for (; (m=bfsEK(capacity,flow,lAdy,ants,minCap,queue,v1,v2))>1e-10; f+=m)
        for (v=v2,u=ants[v]; v!=v1; v=u,u=ants[v]) {flow[u][v]+=m; flow[v][u]-=m;}
}

```



```

return f;
}
double bfsEK(double[][] capacity, double[][] flow, int[][] lAdy,
             int[] ants, double[] minCap, int[] queue, int v1, int v2) {
    int i,t=0,u; double z; Arrays.fill(ants,-1); ants[v1]=-2; minCap[v1]=Double.POSITIVE_INFINITY;
    for (queue[t++]=v1,i=0; i<t; i++)
        for (int v:lAdy[u=queue[i]]) if ((z=capacity[u][v]-flow[u][v])>1e-10&&ants[v]==-1)
            {ants[v]=u; minCap[v]=Math.min(minCap[u],z); if (v==v2) return minCap[v2]; queue[t++]=v;}
    return 0d;
}
int[] toArr(List<Integer> p) {int r[]=new int[p.size()],i=0; for (int x:p) r[i++]=x; return r;}

```

☑ **Teorema Max-flow/Min-cut** – [P.Elias/A.Feinstein/C.E.Shannon/L.R.Ford/D.R.Fulkerson (1956)]
 La máxima cantidad de flujo entre dos nodos v_1 y v_2 es igual al costo de un corte mínimo que separe al nodo v_1 del nodo v_2 (Un corte en un grafo $G=(V,E)$ es una partición de V en dos conjuntos A y B . Se dice que todo arco (a,b) en E tal que $a \in A$ y $b \in B$ atraviesa el corte. El costo de un corte es la suma de los costos de los arcos que lo atraviesan. En flujo en redes, el costo de un corte es la suma de los costos de los arcos que atraviesan el corte y que van de la parte de la fuente a la parte del destino.).

☑ **Flujo máximo de costo mínimo (minimum cost maximum flow (mincost maxflow))**
 Encuentre un flujo máximo (Edmonds-Karp). Mientras hayan ciclos de costo negativo (usar Bellman-Ford) en la red residual calibrada con los costos, aumente el flujo a lo largo del ciclo.

☑ **Máximo número de caminos disyuntos entre dos vértices**
 Corresponde al máximo flujo entre los dos vértices donde todo arco tiene capacidad 1.

PROGRAMACIÓN DINÁMICA

☑ **Sufijo común más largo (Longest common suffix) $\{O(n*m)\}$**
 Sufijo común más largo de dos cadenas $a[0..n-1], b[0..m-1]$:

$$\text{sufcm}(i,j) = \begin{cases} \text{sufcm}(i-1,j-1)+1 & \text{si } i>0 \wedge j>0 \wedge a[i-1]=b[j-1] \\ 0 & \text{de lo contrario} \end{cases}$$

☑ **Subcadena común más larga (Longest common substring) $\{O(n*m)\}$**
 Subcadena común más larga de dos cadenas $a[0..n-1], b[0..m-1]$:

$$\text{answer} = \max_{1 \leq i \leq n \wedge 1 \leq j \leq m} \text{sufcm}(i,j)$$

☑ **Supercadena común más corta entre k cadenas (Shortest common superstring)**

☑ **Subsecuencia común más larga (Longest common subsequence) $\{O(n*m)\}$**
 Subsecuencia común más larga de dos cadenas $a[0..n-1], b[0..m-1]$:

$$\text{subcm}(i,j) = \begin{cases} 0 & \text{si } i=0 \vee j=0 \\ \text{subcm}(i-1,j-1)+1 & \text{si } i>0 \wedge j>0 \wedge a[i-1]=b[j-1] \\ \max(\text{subcm}(i,j-1), \text{subcm}(i-1,j)) & \text{si } i>0 \wedge j>0 \wedge a[i-1] \neq b[j-1] \end{cases}$$

☑ **Supersecuencia común más corta (Shortest common supersequence) $\{O(n*m)\}$**
 Supersecuencia común más corta de dos cadenas $a[0..n-1], b[0..m-1]$:

$$\text{subcmc}(i,j) = \begin{cases} i+j & \text{si } i=0 \vee j=0 \\ \text{subcmc}(i-1,j-1)+1 & \text{si } i>0 \wedge j>0 \wedge a[i-1]=b[j-1] \\ 1 + (\text{subcmc}(i,j-1) \vee \text{subcmc}(i-1,j)) & \text{si } i>0 \wedge j>0 \wedge a[i-1] \neq b[j-1] \end{cases}$$

☑ **Distancia de Levenshtein $\{O(n*m)\}$**
 Distancia de edición entre dos cadenas $a[0..n-1], b[0..m-1]$ (mínimo número de modificaciones, inserciones y eliminaciones para transformar una cadena en otra):

$$\text{dl}(i,j) = \begin{cases} i+j & \text{si } i=0 \vee j=0 \\ \min(\text{dl}(i,j-1)+1, \text{dl}(i-1,j)+1, \text{dl}(i-1,j-1)) & \text{si } i>0 \wedge j>0 \wedge a[i-1]=b[j-1] \\ \min(\text{dl}(i,j-1)+1, \text{dl}(i-1,j)+1, \text{dl}(i-1,j-1)+1) & \text{si } i>0 \wedge j>0 \wedge a[i-1] \neq b[j-1] \end{cases}$$

☑ **Problema de la pavimentación $\{O(n*m)\}$**
 Número de maneras de pavimentar un camino de dimensiones $1 \times n$ con losas de dimensiones $1 \times [1..m]$:

$$\text{nmpc}(i) = \begin{cases} 1 & \text{si } i=0 \\ \sum_{1 \leq k \leq m \wedge i \geq k} \text{nmpc}(i-k) & \text{si } i>0 \end{cases}$$

☑ **Mínimo número de monedas para retornar un cambio (Coin change) $\{O(x*n)\}$**
 Mínimo número de monedas con denominaciones $d[0], d[1], \dots, d[n-1]$ para devolver una cantidad x :

$$\text{mnmc}(c) = \begin{cases} 0 & \text{si } c=0 \end{cases}$$

$= (\downarrow k | 0 \leq k < n \wedge d[k] \leq c : \text{nmrc}(c - d[k])) + 1$ si $c > 0$ donde la identidad de \downarrow es ∞

☑ **Número de maneras para retornar un cambio con monedas (Coin counting) $\{O(x*n)\}$**

Número de maneras para dar una cantidad x con monedas de denominaciones $d[0], d[1], \dots, d[n-1]$:

```

① answer=nmrc(x)
nmrc(c) = 1                                si c=0
         = ( $\sum k | 0 \leq k < n \wedge d[k] \leq c : \text{nmrc}(c - d[k])$ ) si c>0 donde la identidad de  $\sum$  es 0
② answer=nmrc(x,n)
nmrc(c,i) = 1                                si c=0
           = 0                                si c>0  $\wedge$  i=0
           = nmrc(c,i-1)                      si c>0  $\wedge$  i>0  $\wedge$  d[i-1]>c
           = nmrc(c,i-1)+nmrc(c-d[i-1],i)    si c>0  $\wedge$  i>0  $\wedge$  d[i-1]≤c

```

☑ **Decidir si algunos números de una lista de naturales suman un valor $\{O(x*n)\}$**

Decidir si algunos elementos de un arreglo de naturales $d[0..n-1]$ suman una cantidad natural x :

```

answer=aeon(x,n)
aeon(c,i) = true                                si c=0
           = false                             si c>0  $\wedge$  i=0
           = aeon(c,i-1)                       si c>0  $\wedge$  i>0  $\wedge$  d[i-1]>c
           = aeon(c,i-1)  $\vee$  aeon(c-d[i-1],i-1) si c>0  $\wedge$  i>0  $\wedge$  d[i-1]≤c

```

☑ **Problema del morral $\{O(m*n)\}$**

Se tienen $n > 0$ objetos con pesos $p[0], p[1], \dots, p[n-1]$ y utilidades $u[0], u[1], \dots, u[n-1]$ y un morral con capacidad m . Máxima utilidad conseguible:

```

answer=mr(m,n)
mr(c,i) = 0                                si i=0
         = mr(c,i-1)                      si i>0  $\wedge$  p[i-1]>c
         = mr(c,i-1)  $\uparrow$  (mr(c-p[i-1],i-1)+u[i-1]) si i>0  $\wedge$  p[i-1]≤c

```

☑ **Problema del CD $\{O(m*n)\}$**

Se tiene un CD con $n > 0$ pistas con duraciones $d[0], d[1], \dots, d[n-1]$ y un cassette con capacidad m . Máximo tiempo de grabación: Véase el problema del morral con $p=d$ y $u=d$.

☑ **Matrix chain multiplication $\{O(n^3)\}$**

Dada una lista de n matrices con dimensiones $d[0] \times d[1], d[1] \times d[2], \dots, d[n-1] \times d[n]$, dar el mínimo número de multiplicaciones necesarias para multiplicarlas en secuencia.

```

answer=mcm(0,n)
mcm(i,j) = 0                                si i≥j-1
         = ( $\downarrow k | i+1 \leq k \leq j-1 : \text{mcm}(i,k) + \text{mcm}(k,j) + d[i]*d[k]*d[j]$ ) si i<j-1

```

☑ **Mayor suma alcanzable por subarreglos - Algoritmo de Kadane $\{O(n)\}$**

```

long kadane(long[] arr) {
    long act=0, r=0;
    for (long v:arr) {act+=v; if (act>r) r=act; if (act<0) act=0;}
    return r;
}

```

☑ **Mayor suma alcanzable por submatrices - Algoritmo de Kadane $\{O(n^3)\}$**

```

long kadane2D(long[][] mat) {
    int m=mat.length, n=m==0?0:mat[0].length; long r=0, sumas[][]=new long[m][n], arr[]=new long[n];
    for (int i=0; i<m; i++) for (int j=0; j<n; j++) sumas[i][j]=mat[i][j]+(i>0?sumas[i-1][j]:0);
    for (int i1=0; i1<m; i1++) for (int i2=i1; i2<m; i2++) {
        for (int j=0; j<n; j++) arr[j]=sumas[i2][j]-(i1>0?sumas[i1-1][j]:0);
        r=Math.max(r, kadane(arr));
    }
    return r;
}

```

☑ **Subsecuencia creciente más larga (Longest increasing subsequence) $\{O(n^2)\}$**

```

int lis1(long[] arr) {
    int n=arr.length, maxs[]=new int[n], res=0, i, j;
    for (i=0; i<n; i++) {res=Math.max(res, maxs[i]);
        for (maxs[i]=1, j=0; j<i; j++) if (arr[i]>arr[j]) maxs[i]=Math.max(maxs[i], maxs[j]+1);
    }
    return res;
}

```

☑ **Subsecuencia creciente más larga (Longest increasing subsequence) $\{O(n*\log_2(n))\}$**

```

int lis2(long[] arr) {
    int n=arr.length, res=0; long val[]=new long[n+1]; val[0]=Long.MIN_VALUE; //inds[0]=0
    for (long v:arr) { // Sea i el índice de v en arr
        int j=Arrays.binarySearch(val, 0, res+1, v); j=(j<0?-j-1:j); //ants[i]=inds[j];
        if (j==res || v<val[j+1]) {val[j+1]=v; res=Math.max(res, j+1);} //inds[j+1]=i;
    }
}

```

```

    }
    return res;
}

```

GEOMETRÍA COMPUTACIONAL

LOS PUNTOS DE LOS POLÍGONOS SE ENUMERAN EN SENTIDO CONTRARIO AL DE LAS MANECILLAS DEL RELOJ
EL PRIMER PUNTO DE LOS POLÍGONOS NO SE REPITE AL FINAL

☑ **double[][] to Shape {0(n)}**

```

Path2D.Double getShape(double[][] pt) {
    Path2D.Double r=new Path2D.Double(Path2D.WIND_EVEN_ODD); r.moveTo(pt[0][0],pt[0][1]);
    for (int i=1; i<pt.length; i++) r.lineTo(pt[i][0],pt[i][1]);
    r.closePath(); return r;
}

```

☑ **Shape to List<double[][]> {0(n)}**

```

java.util.List<double[][]> getPolygons(Shape sh) {
    java.util.List<double[][]> r=new ArrayList<double[][]>();
    java.util.List<double[]> z=new ArrayList<double[]>();
    double[] c=new double[6];
    for (PathIterator ph=sh.getPathIterator(null); !ph.isDone(); ph.next()) {
        if (ph.currentSegment(c)!=PathIterator.SEG_CLOSE) z.add(new double[]{c[0],c[1]});
        else {r.add(z.toArray(new double[0][])); z.clear();}
    }
    return r;
}

```

☑ **Distancia al cuadrado entre dos puntos {0(1)}**

```

double ds(double[] a, double[] b) {return (b[0]-a[0])*(b[0]-a[0])+(b[1]-a[1])*(b[1]-a[1]);}

```

☑ **Distancia de punto a línea {0(1)}**

```

double distPL(double[] p1, double[] p2, double[] p) {
    return Math.abs((p2[0]-p1[0])*(p1[1]-p[1])-(p2[1]-p1[1])*(p1[0]-p[0]))/Math.sqrt(ds(p1,p2));
}

```

☑ **Distancia de punto a segmento {0(1)}**

```

double distPS(double[] p1, double[] p2, double[] p) {
    double dP=ds(p1,p),d1=ds(p1,p2),d2=ds(p2,p);
    return (d2+dP<d1||d1+dP<d2)?Math.sqrt(Math.min(d1,d2)):distPL(p1,p2,p);
}

```

☑ **Line Intersection 2D {0(1)}**

```

double[] intLineas(double x1, double y1, double x2, double y2,
    double x3, double y3, double x4, double y4) {
    double xa=x2-x1,xb=x4-x3,xc=x1-x3,ya=y2-y1,yb=y4-y3,yc=y1-y3,d=yb*xa-xb*ya,n=xb*yc-yb*xc;
    return Math.abs(d)<1e-11?null:new double[]{x1+xa*n/d,y1+ya*n/d};
}

```

☑ **Segment Intersection 2D {0(1)}**

```

double[] intSegmentos(double x1, double y1, double x2, double y2,
    double x3, double y3, double x4, double y4) { // No se sobrelapan
    double[] p1={x1,y1},p2={x2,y2},p3={x3,y3},p4={x4,y4},p12[]={p1,p2},p34[]={p3,p4};
    for (double[] p:p12) if (distPS(p3,p4,p)<1e-11) return p;
    for (double[] p:p34) if (distPS(p1,p2,p)<1e-11) return p;
    double[] p=intLineas(x1,y1,x2,y2,x3,y3,x4,y4);
    return p!=null&&distPS(p1,p2,p)<1e-11&&distPS(p3,p4,p)<1e-11?p:null;
}

```

☑ **Circle Intersection 2D {0(1)}**

```

double[][] intCirculos(double[] c1, double r1, double[] c2, double r2) {
    if (r2<r1) return intCirculos(c2,r2,c1,r1);
    double d=Math.sqrt(ds(c1,c2));
    if (d<1e-11||d<r2-r1-1e-11||d>r1+r2+1e-11) return null;
    double u=(d*d-r1*r1+r2*r2)/(d*2),v=Math.sqrt(Math.max(r2*r2-u*u,0));
    double dx=(c1[0]-c2[0])/d,dy=(c1[1]-c2[1])/d;
    return new double[][]{{c2[0]+dx*u-dy*v,c2[1]+dy*u+dx*v},{c2[0]+dx*u+dy*v,c2[1]+dy*u-dx*v}};
}

```

☑ **Sphere Intersection 3D**

☑ **Perímetro de un polígono {0(n)}**

```

double perimetro(double[][] pt) {
    double r=0d;

```

```

    for (int i=0,t=pt.length; i<t; i++) r+=Math.sqrt(ds(pt[i],pt[i+1==t?0:i+1]));
    return r;
}

```

☑ **Área de un polígono {O(n)}**

```

double area(double[][] pt) {
    double r=0d; int t=pt.length;
    for (int i=0,j=1; i<t; i++,j=j+1==t?0:j+1) r+=pt[i][0]*pt[j][1]-pt[i][1]*pt[j][0];
    return Math.abs(r)/2;
}

```

☑ **Área de un polígono con puntos coplanares en 3D {O(n)}**

Utilice la fórmula en 2D considerando las coordenadas en z de los puntos en el cálculo de la magnitud de los productos cruz.

☑ **Centroide (centro de masa) de un polígono {O(n)}**

```

double[] centroide(double[][] pt) {
    double p[]={0d,0d},d=area(pt)*6;
    for (int i=0,j=1,t=pt.length; i<t; i++,j=j+1==t?0:j+1)
        for (int k=0; k<2; k++) p[k]+=(pt[i][k]+pt[j][k])*(pt[i][0]*pt[j][1]-pt[j][0]*pt[i][1]);
    return new double[]{p[0]/d,p[1]/d};
}

```

☑ **Punto dentro de polígono? {O(n)}**

```

boolean dentroPoligono(double[][] pt, double[] p, boolean bd) { // bd: con borde?
    boolean b=false;
    for (int i=0,j=1,t=pt.length; i<t; i++,j=j+1==t?0:j+1) {
        if (distPS(pt[i],pt[j],p)<1e-11) return bd;
        if (((pt[j][1]<=p[1]&&pt[j][1]<pt[i][1]) || (pt[i][1]<=p[1]&&pt[i][1]<pt[j][1])) &&
            (p[0]-pt[j][0]<(p[1]-pt[j][1])*(pt[i][0]-pt[j][0])/(pt[i][1]-pt[j][1]))) b=!b;
    }
    return b;
}

```

☑ **Punto dentro de polígono? - Java {O(n)?}**

```

boolean dentroPoligono(double[][] pt, double[] p, boolean bd) { // bd: con borde?
    for (int i=0,j=1,t=pt.length; i<t; i++,j=j+1==t?0:j+1)
        if (Line2D.ptSegDist(pt[i][0],pt[i][1],pt[j][0],pt[j][1],p[0],p[1])<1e-11) return bd;
    return getShape(pt).contains(p[0],p[1]);
}

```

☑ **Círculo dados tres o dos puntos {O(1)}**

```

double[] circulo3P(double[] p1, double[] p2, double[] p3)
{return circulo3P(p1[0],p1[1],p2[0],p2[1],p3[0],p3[1]);}
double[] circulo3P(double x1, double y1, double x2, double y2, double x3, double y3) {
    double x4=(x1+x2)/2,y4=(y1+y2)/2,x5=(x3+x2)/2,y5=(y3+y2)/2;
    double c[]=intLineas(x4,y4,x4+y2-y1,y4+x1-x2,x5,y5,x5+y2-y3,y5+x3-x2),cx=c[0],cy=c[1];
    return new double[]{cx,cy,Math.sqrt((cx-x1)*(cx-x1)+(cy-y1)*(cy-y1))};
}
double[] circulo2P(double[] p1, double[] p2)
{return circulo2P(p1[0],p1[1],p2[0],p2[1]);}
double[] circulo2P(double x1, double y1, double x2, double y2) {
    return new double[]{(x1+x2)/2,(y1+y2)/2,Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1))/2};
}

```

☑ **Closest Pair {O(n*log₂(n))}**

```

Comparator<double[]> c=new Comparator<double[]>() {
    public int compare(double[] p, double[] q)
        {return Math.abs(p[0]-q[0])>=9.9e-12?(p[0]<q[0]?-1:1):(p[1]!=q[1]?(p[1]<q[1]?-1:1):0);}
};
int f(int n) {return n>=0?n:-n-1;}
double closestPair(double[][] pt) {
    Arrays.sort(pt,c); int k=pt.length-1,z[]=new int[k+1];
    for (int i=k; i>=0; i--) z[i]=i==k||Math.abs(pt[i][0]-pt[i+1][0])>=9.9e-12?i:z[i+1];
    for (int i=0; i<k; i++) if (c.compare(pt[i],pt[i+1])==0) return 0;
    return Math.sqrt(closestPair(pt,0,k,z));
}
double closestPair(double[][] pt, int lI, int lS, int[] z) {
    if (lI>=lS) return Double.POSITIVE_INFINITY;
    int lM=(lI+lS)/2; double inf=Double.NEGATIVE_INFINITY,x=pt[lM][0];
    double r=Math.min(closestPair(pt,lI,lM,z),closestPair(pt,lM+1,lS,z)),d=Math.sqrt(r);
}

```

```

for (int i=f(Arrays.binarySearch(pt,lI,lM+1,new double[] {x-d+1e-11,inf},c)); i<=lM; i++)
    for (int j1=lM+1,j2,j; j1<=lS&&pt[j1][0]<pt[i][0]+d; j1=j2+1) {
        j=f(Arrays.binarySearch(pt,j1,(j2=z[j1])+1,new double[] {pt[j1][0],pt[i][1]},c));
        for (int k=0; k<2; k++) if (j1<=j-k&&j-k<=j2) r=Math.min(r,ds(pt[i],pt[j-k]));
    }
return r;
}
}
☑ Convex Hull - Graham Scan [Ronald Graham (1972)]  $\{O(n \cdot \log_2(n))\}$ 
int sgn(double x) {return Math.abs(x)>9.9e-12?(x<0?-1:1):0;}
double cruz(double[] a, double[] b) {return a[0]*b[1]-a[1]*b[0];}
double cruz(double[] a, double[] b, double[] c) {return cruz(a,b)+cruz(b,c)+cruz(c,a);}
double[][] convexHullGS(double[][] pt, boolean bd) { // bd: con borde?
    int h=pt.length,i=h,t=0; double v[]=null,w[],r[][]=new double[h][];
    for (double[] p:pt) if (v==null||p[1]<v[1]||(p[1]==v[1]&&p[0]>v[0])) v=p; v=v.clone();
    for (double[] p:pt) {p[0]-=v[0]; p[1]-=v[1];}
    Arrays.sort(pt,new Comparator<double[]>() {public int compare(double[] a, double[] b)
    {double cz=cruz(b,a); return sgn(sgn(cz)!=0?cz:a[0]*a[0]+a[1]*a[1]-b[0]*b[0]-b[1]*b[1]);}});
    if (bd) while (i-1>=0&&cruz(pt[h-1],pt[i-1])==0) i--;
    if (bd) for (int k=i; k<(i+h)/2; k++) {w=pt[k]; pt[k]=pt[h-1-k+i]; pt[h-1-k+i]=w;}
    for (double[] p:pt) {while (t>=2&&sgn(cruz(r[t-1],p,r[t-2]))<(bd?0:1)) t--; r[t++]=p;}
    for (double[] p:pt) {p[0]+=v[0]; p[1]+=v[1];}
    return Arrays.copyOfRange(r,0,t);
}
}
☑ Minimal (~Smallest) Enclosing Circle (~Disc) - {expected:  $O(n)$ ; worst:  $O(n^3)$ }
double[] mecR(double[][] pt) { // Randomized algorithm
    int t=pt.length,i,j,k; double[] mc; if (t==1) return circulo2P(pt[0],pt[0]);
    for (mc=circulo2P(pt[0],pt[1]),i=2; i<t; i++) if (Math.sqrt(ds(pt[i],mc))>mc[2]+1e-11)
        for (mc=circulo2P(pt[0],pt[i]),j=1; j<i; j++) if (Math.sqrt(ds(pt[j],mc))>mc[2]+1e-11)
            for (mc=circulo2P(pt[i],pt[j]),k=0; k<j; k++) if (Math.sqrt(ds(pt[k],mc))>mc[2]+1e-11)
                mc=circulo3P(pt[i],pt[j],pt[k]);
    return mc;
}
}
☑ Minimal (~Smallest) Enclosing Circle (~Disc) - [Pr. Chrystal (1885)]  $\{O(n^2)\}$ 
double[] mec(double[][] pt) {
    pt=convexHullGS(pt,false); int t=pt.length; boolean g1=false,g2=false;
    if (t==1) return new double[] {pt[0][0],pt[0][1],0};
    for (int u=0,v=1,w=-1; true; v=g1?w:v,u=g2?w:u,w=-1) {
        double max=-2,aw=0,bw=0,cw=0;
        for (int i=0; i<t; i++) if (i!=u&&i!=v) {
            double a=ds(pt[i],pt[u]),b=ds(pt[i],pt[v]),c=ds(pt[u],pt[v]),d=(a+b-c)/Math.sqrt(a*b);
            if (d>max) {max=d; w=i; aw=a; bw=b; cw=c;}
        }
        if (max<=0) return circulo2P(pt[u],pt[v]);
        g1=bw+cw<aw; g2=cw+aw<bw; if (!g1&&!g2) return circulo3P(pt[u],pt[v],pt[w]);
    }
}
}
☑ Anchura mínima de un conjunto de puntos  $\{O(n \cdot \log_2(n))\}$ 
double anchura(double[][] pt) {
    pt=convexHullGS(pt,false); int t=pt.length,k=2; double r=Double.POSITIVE_INFINITY;
    if (t>2) for (int i=0,j=1; i<t; i++,j=j+1==t?0:j+1) {
        while (distPL(pt[i],pt[j],pt[k+1==t?0:k+1])>distPL(pt[i],pt[j],pt[k])) k=k+1==t?0:k+1;
        r=Math.min(r,distPL(pt[i],pt[j],pt[k]));
    }
    return t<=2?0:r;
}
}
☑ Diámetro de un conjunto de puntos  $\{O(n \cdot \log_2(n))\}$ 
double diametro(double[][] pt) {
    pt=convexHullGS(pt,false); int t=pt.length,k=1; double r=0;
    if (t>1) for (int i=0; i<t; i++) {
        while (ds(pt[i],pt[k+1==t?0:k+1])>ds(pt[i],pt[k])) k=k+1==t?0:k+1;
        r=Math.max(r,ds(pt[i],pt[k]));
    }
    return Math.sqrt(r);
}
}

```

☑ Line Segment Intersection - Plane Sweep Algorithm [Bentley-Ottmann (1979)] $\{O((n+k)*\log_2(n))\}$

```
// INESTABLE CON LA PRECISIÓN
int sgn(double v) {return Math.abs(v)>1e-11?(v<0?-1:1):0;}
double getM(double[] a, double[] b) {return (b[1]-a[1])/(b[0]-a[0]);}
double getY(double[] a, double[] b, double x) {return a[1]+getM(a,b)*(x-a[0]);}
double[] rot(double[] a, int f) {
    double ang=Math.atan2(a[1],a[0])+Math.E*f,d=Math.sqrt(a[0]*a[0]+a[1]*a[1]);
    return new double[]{d*Math.cos(ang),d*Math.sin(ang)};
}
TreeMap<Double,Set<Integer>> events=new TreeMap<Double,Set<Integer>>(new Comparator<Double>()
{public int compare(Double x1, Double x2) {return sgn(x1-x2);}});
void add(double x, int i)
{Set<Integer> s=events.get(x); if (s==null) events.put(x,s=new TreeSet<Integer>()); s.add(i);}
void notify(double[][] sgs, double x, int i, int j) {
    if (i<0||j<0) return;
    double[] p1=sgs[i][0],p2=sgs[i][1],p3=sgs[j][0],p4=sgs[j][1];
    double[] p=intSegmentos(p1[0],p1[1],p2[0],p2[1],p3[0],p3[1],p4[0],p4[1]);
    if (p!=null&&sgn(p[0]-x)>=0) {add(p[0],i); add(p[0],j);}
}
double[][] bentleyOttmann(double[][][] segments) { // No debe haber dos segmentos superpuestos
    class ComparatorSg implements Comparator<double[][]> {
        double x=0d;
        public int compare(double[][] sg1, double[][] sg2) {
            int cY=sgn(getY(sg1[0],sg1[1],x)-getY(sg2[0],sg2[1],x));
            int cM=sgn(getM(sg1[0],sg1[1])-getM(sg2[0],sg2[1]));
            return cY!=0?cY:(cM!=0?cM:sgn(sg1[0][0]-sg2[0][0]));
        }
    }; ComparatorSg c=new ComparatorSg();
    TreeMap<double[][],Integer> sweepLine=new TreeMap<double[][],Integer>(c);
    int n=segments.length; double sgs[][][]=new double[n][][],ult[]=null; events.clear();
    for (int i=0; i<n; i++) {
        double[][] sgA={rot(segments[i][0],1),rot(segments[i][1],1)},sgB={sgA[1],sgA[0]};
        for (double[] p:(sgs[i]=sgA[0][0]>sgA[1][0]?sgB:sgA)) add(p[0],i);
    }
    java.util.List<double[]> res=new ArrayList<double[]>();
    while (!events.isEmpty()) {
        Map.Entry<Double,Set<Integer>> e=events.pollFirstEntry(); Set<Integer> s=e.getValue();
        double sgI[][]=sgs[s.iterator().next()],x=e.getKey(),y=getY(sgI[0],sgI[1],x),p[]={x,y};
        double q[]=rot(p,-1); for (int id:s) sweepLine.remove(sgs[id]); c.x=x;
        if (s.size()>1&&(ult==null||sgn(Math.sqrt(ds(p,ult)))!=0)) {res.add(q); ult=p;}
        Map.Entry<double[][],Integer> eS=sweepLine.ceilingEntry(sgI),eI=sweepLine.lowerEntry(sgI);
        int j=eS==null?-1:eS.getValue(),i=eI==null?-1:eI.getValue();
        for (int k:s) if (sgn(sgs[k][1][0]-x)>0) {
            notify(sgs,x,j,k); notify(sgs,x,i,k); sweepLine.put(sgs[k],k); p=null;
        }
        if (p!=null) notify(sgs,x,j,i);
    }
    return res.toArray(new double[0][]);
}
```

☑ Determinar si un rectángulo cabe dentro de otro - Carver [W. Carver] $\{O(1)\}$

```
boolean cabe(long p, long q, long a, long b) { // El rectángulo pXq cabe en el rectángulo aXb?
    long x,y,z,w; if (p<q) {x=p; p=q; q=x;} if (a<b) {x=a; a=b; b=x;}
    if (p<=a&&q<=b) return true;
    if (p==q) return b>=q;
    x=2*p*q*a; y=p*p-q*q; z=p*p+q*q; w=z-a*a;
    return p>a&&1d*b*z>=x+y*Math.sqrt(w)-1e-10;
}
```

☑ Máximo número de segmentos disyuntos dos a dos en un conjunto (1D) $\{O(n*\log_2(n))\}$

```
int sgn(double x) {return x!=0?(x<0?-1:1):0;}
int maxSegmentosDisyuntos(double[][] sgs) {
    Arrays.sort(sgs,new Comparator<double[]>() {
        public int compare(double[] a, double[] b)
        {return -sgn(sgn(a[0]-b[0])!=0?a[0]-b[0]:a[1]-b[1]);}
    });
}
```



```
double ult=Double.POSITIVE_INFINITY; int res=0;
for (double[] sg:sgs) if (sg[1]<ult-1e-15) {ult=sg[0]; res++;}
return res;
}
```

- ☒ Voronoi Diagram
- ☒ Delaunay Triangulation
- ☒ Minimum Width Annuli
- ☒ Largest Empty Circle
- ☒ Euclidean Minimum Spanning Tree
- ☒ Árbol de Steiner
- ☒ Shortest length triangulation
- ☒ Teorema de Pick – [Georg Alexander Pick (1899)]

Dado un polígono simple cuyos puntos están sobre coordenadas enteras, se tiene que $A=i+b/2-1$ donde A es el área del polígono, i es el número de puntos con coordenadas enteras dentro del polígono y b es el número de puntos con coordenadas enteras en el borde del polígono.

- ☒ Fórmulas varias

$\theta = \arccos(u \cdot v / (|u| \cdot |v|))$

[Lados: a, b, c] [Ángulos: α, β, γ] [Alturas: x, y, z]...

$p=a+b+c$ $s=p/2$ $A=\sqrt{s(s-a)(s-b)(s-c)}$ $ax/2$ $a/\sin(\alpha)=b/\sin(\beta)=c/\sin(\gamma)$ $a^2=b^2+c^2-2bc\cos(\alpha)$

$a=b\cos(\gamma)+c\cos(\beta)$ $A=bc\sin(\alpha)/2$ $r=A/s$ $R=abc/(4A)$

$\sin(\alpha/2)=\sqrt{(s-b)(s-c)/(bc)}$ $\cos(\alpha/2)=\sqrt{s(s-a)/(bc)}$ $\tan(\alpha/2)=\sqrt{(s-b)(s-c)/(s(s-a))}$

$A=x^2y^2z^2/\sqrt{(yz+xz+xy)(-yz+xz+xy)(yz-xz+xy)(yz+xz-xy)}$

Coordenadas rectangulares a polares: $x=r\cos(\varphi)\cos(\theta)$ $y=r\cos(\varphi)\sin(\theta)$ $z=r\sin(\varphi)$

Coordenadas polares a rectangulares: $\tan(\theta)=y/x$ $\tan(\varphi)=z/\sqrt{x^2+y^2}$ $r=\sqrt{x^2+y^2+z^2}$

NÚMEROS (MCD, BINOMIAL, PRIMOS, FÓRMULAS, ...)

$\pi(n)=\text{PrimeCountingFunction}(n)=\#\text{primes}([2..n])$

$\xi(n)=e_1+e_2+\dots+e_k$ donde $n=p_1^{e_1}p_2^{e_2}\dots p_k^{e_k}$ con p_1, p_2, \dots, p_k primos

$\zeta(n)=k$ donde $n=p_1^{e_1}p_2^{e_2}\dots p_k^{e_k}$ con p_1, p_2, \dots, p_k primos

- ☒ Máximo común divisor – Algoritmo de Euclides [Euclides (300AC)] $\{O(\log_2(a \uparrow b))\}$

```
long gcd(long a, long b) {long t; while (b!=0) {t=b; b=a%b; a=t;} return a;}
```

- ☒ Máximo común divisor precalculado $\{O(M^2)\}$

```
int[][] gcd(int M) { // [0..M][0..M]
```

```
int r[][]=new int[M+1][M+1], i, j;
```

```
for (i=0; i<=M; i++) for (j=0; j<=M; j++) r[i][j]=i==0?j:(j==0?i:(i>j?r[i%j][j]:r[j%i][i]));
```

```
return r;
```

```
}
```

- ☒ Máximo común divisor – Algoritmo extendido de Euclides [Euclides (300AC)] $\{O(\log_2(a \uparrow b))\}$

```
long[] gcdExtendido(long a, long b) { // answer[0]=gcd(a,b), answer[1]*a+answer[2]*b=gcd(a,b)
```

```
boolean bs=a<b; long xAnt=1, yAnt=0, x=0, y=1;
```

```
if (bs) {long tmp=a; a=b; b=tmp;}
```

```
while (b!=0) {
```

```
long q=a/b, r=a%b, xTmp=xAnt-q*x, yTmp=yAnt-q*y; a=b; b=r; xAnt=x; yAnt=y; x=xTmp; y=yTmp;
```

```
}
```

```
return new long[]{a, bs?yAnt:xAnt, bs?xAnt:yAnt};
```

```
}
```

- ☒ Mínimo común múltiplo $\{O(\log_2(a \uparrow b))\}$

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

- ☒ Primos relativos (coprimos) $\{O(\log_2(a \uparrow b))\}$

```
boolean primosRelativos(long a, long b) {return gcd(a,b)==1L;}
```

- ☒ Coeficiente binomial $\{O(r \downarrow (n-r))\}$

```
BigInteger binomial(int n, int r) { // [n][r], 0<=r<=n
```

```
BigInteger res=BigInteger.valueOf(1);
```

```
for (int i=1, t=r>n-r?n-r:r; i<=t; i++)
```

```
res=res.multiply(BigInteger.valueOf(n-i+1)).divide(BigInteger.valueOf(i));
```

```
return res;
```

```
}
```

- ☒ Coeficiente binomial precalculado (fila) $\{O(n)\}$

```
BigInteger[] binomialF(int n) { // [n][0..n]
```

```
BigInteger arr[]=new BigInteger[n+1]; arr[0]=arr[n]=BigInteger.ONE;
```

```
for (int i=1, t=n>>1; i<=t; i++)
```

```
arr[i]=arr[n-i]=arr[i-1].multiply(BigInteger.valueOf(n-i+1)).divide(BigInteger.valueOf(i));
```

```
return arr;
```

```

}
☑ Coeficiente binomial precalculado (matriz)  $\{O(N^2)\}$ 
BigInteger[][] binomialM(int N) { // [0..N][0..N]
    BigInteger mat[][]=new BigInteger[N+1][N+1];
    for (int n=0; n<=N; n++) {
        mat[n][0]=mat[n][n]=BigInteger.ONE;
        for (int i=1,t=n>>1; i<=t; i++) mat[n][i]=mat[n][n-i]=mat[n-1][i-1].add(mat[n-1][i]);
    }
    return mat;
}
☑ Primos - Criba de Eratóstenes [Eratóstenes]  $\{O(M*\log_2(M)*\log_2(\log_2(M)))\}$ 
long[] primos(int M) { // [0..M-1], M>=2
    boolean b[]=new boolean[M]; int i,j,k,c=2;
    for (i=2; (k=i*i)<M; i++) if (!b[i]) for (j=k; j<M; j+=i) {b[j]=true; c++;}
    long r[]=new long[M-c]; for (i=2,j=0; i<M; i++) if (!b[i]) r[j++]=i;
    return r;
}
☑ Primos - Criba segmentada de Eratóstenes
long[] primos(long x1, long x2, long[] primos) { // [x1..x2], 2<=x1<=x2<M^2, M=max(primos)
    int T=(int)(x2-x1+1),c=0,i,n=primos.length; long R=(long)(Math.sqrt(x2)+1+1e-5),p,q;
    boolean b[]=new boolean[T];
    for (i=0; i<n&&(p=primos[i])<=R; i++) for (q=Math.max((x1+p-1)/p,2)*p; q<=x2; q+=p)
        if (!b[(int)(q-x1)]) {b[(int)(q-x1)]=true; c++;}
    for (p=x1; p<=R; p++) if (!b[(int)(p-x1)]) for (q=p*p; q<=x2; q+=p)
        if (!b[(int)(q-x1)]) {b[(int)(q-x1)]=true; c++;}
    long r[]=new long[T-c]; for (p=x1,i=0; p<=x2; p++) if (!b[(int)(p-x1)]) r[i++]=p;
    return r;
}
☑ Test de primalidad  $\{O(\pi(\sqrt{x}))\}$ 
boolean esPrimo(long x, long[] primos) { // 2<=x<M^2, M=max(primos)
    if ((x&1L)==0L) return x==2L;
    int i,n=primos.length; double R=(long)(Math.sqrt(x)+1+1e-5); long p;
    if (x<=primos[n-1]) return Arrays.binarySearch(primos,x)>=0;
    for (i=0; i<n&&(p=primos[i])<=R; i++) if (x%p==0) return false;
    return true;
}
☑ Descomponer un número en factores primos  $\{O(\pi(\sqrt{x})+\log_2(x))\}$ 
long[][] factoresPrimos(long x, long[] primos) { // 2<=x<M^2, M=max(primos). INMUTABLES.
    List<long[]> res=new ArrayList<long[]>(); double R=(long)(Math.sqrt(x)+1+1e-5); long p,c;
    for (int i=0,n=primos.length; i<n&&(p=primos[i])<=R; i++) {
        for (c=0; x%p==0; x/=p,c++); if (c>0) res.add(new long[]{p,c});
    }
    if (x>1) res.add(new long[]{x,1});
    return res.toArray(new long[0][]);
}
☑ Multiplicar y dividir números en descomposición prima  $\{O(\zeta(x)+\zeta(y))\}$ 
long[][] mulPowFP(long[][] fP1, long[][] fP2, long exp) { // answer=num(fP1)*num(fP2)^(exp)
    if (exp==0) return fP1;
    int i=0,j=0,n=fP1.length,m=fP2.length; List<long[]> res=new ArrayList<long[]>(n+m);
    while (true) {
        while ((i<n||j<m)&&!(i<n&&j<m&&fP1[i][0]==fP2[j][0])) {
            boolean b=i<n&&(j==m||fP1[i][0]<fP2[j][0]); long[] f=b?fP1[i++]:fP2[j++];
            res.add(b|exp==1?f:new long[]{f[0],f[1]*exp});
        }
        if (i==n&&j==m) break;
        long p=fP1[i][0],e=fP1[i++][1]+exp*fP2[j++][1]; if (e!=0) res.add(new long[]{p,e});
    }
    return res.toArray(new long[0][]);
}
☑ Cantidad de divisores de un número en descomposición prima  $\{O(\zeta(x))\}$ 
long divisoresFE(long[][] fp) { // fp sin factores primos con exponentes negativos
    long r=1L; for (long[] f:fp) r*=(f[1]+1); return r;
}
☑ Cantidad de factores primos en la descomposición prima  $\{O(M*\log_2(M)*\log_2(\log_2(M)))\}$ 

```



```
int[] factoresPrimos(int M) { // [0..M-1], M>=2
    int b[]=new int[M]; int i,j,k;
    for (i=2; (k=i*i)<M; i++) if (b[i]==0) for (j=k; j<M; j+=i) if (b[j]==0) b[j]=i;
    for (i=2; i<M; i++) b[i]=b[i]==0?1:b[i/b[i]]+1;
    return b;
}
```

☑ Factorizaciones de un número

```
List<String> factorizaciones(int n) {return iterarFc(n,new int[100],0,new ArrayList<String>());}
List<String> iterarFc(int n, int[] divs, int k, List<String> res) {
    for (int d=(k>0?divs[k-1]:2),dT=(int)(Math.sqrt(n)+1e-5); d<=dT; d++) if (n%d==0)
        iterarFc(n/(divs[k]=d),divs,k+1,res);
    if (k>=0) {String s=""; for (int i=0; i<k; i++) s+=divs[i]+"X"; s+=n; res.add(s);}
    return res;
}
```

☑ Aproximación de Stirling - [Abraham de Moivre, James Stirling]

$n! \approx \text{raiz}(2\pi n)(n/e)^n$

☑ Teorema pequeño de Fermat

$a^{p-1} \equiv 1 \pmod{p}$ donde p es primo y a no es múltiplo de p

☑ Euler's totient function

Sea p primo, p_1, \dots, p_m primos distintos y $n = p_1^{k_1} \dots p_m^{k_m}$.

$\varphi(1)=1$ $\varphi(p^k)=(p-1)*p^{k-1}$ $\varphi(n)=(p_1-1)*p_1^{k_1-1} \dots (p_m-1)*p_m^{k_m-1} = n*(1-1/p_1)*\dots*(1-1/p_m)$

$\varphi(a*b)=\varphi(a)*\varphi(b)$ donde $\text{gcd}(a,b)=1$

☑ Teorema de Euler

$a^{\varphi(n)} \equiv 1 \pmod{n}$ donde n es un entero positivo y $\text{gcd}(a,n)=1$

$a^b \equiv a^{b \bmod \varphi(n)} \pmod{n}$ donde n es un entero positivo y $\text{gcd}(a,n)=1$

☑ Teorema chino del residuo

Sean n_1, \dots, n_k enteros primos relativos de a pares. Entonces para todos a_1, \dots, a_k enteros, existe un x entero que soluciona el sistema de congruencias simultáneas $x \equiv a_1 \pmod{n_1}, \dots, x \equiv a_k \pmod{n_k}$.

```
long solTeoremaChinoResiduo(long a[], long[] n) { // n tiene coprimos de a pares
    long N=1,x=0; for (long y:n) N*=y;
    for (int i=0; i<a.length; i++) x=(x+a[i]*gcdExtendido(n[i],N/n[i])[2]*N/n[i])%N;
    return (x+N)%N;
}
```

☑ Regla de Simpson

$$\int_a^b f(x) dx \approx \frac{(b-a)}{6} * \left[f(a) + 4 * f\left(\frac{a+b}{2}\right) + f(b) \right]$$

☒ Método de Newton

☒ Método de la bisección

☑ Combinaciones, Permutaciones

$\text{binomialCoefficient}(n,r)=n!/(r!*(n-r)!)$

$r_permutations(n,r)=n!/(n-r)!)$

☑ Números de Catalán (Catalan Numbers)

$\text{catalan}(n)=\text{binomialCoefficient}(n*2,n)/(n+1)=(n*2)!/((n+1)!*n!)$

Usos:

* Cantidad de árboles de búsqueda binaria que pueden construirse con un conjunto de n números de tal manera que a cada elemento del conjunto se le asocie exactamente un nodo del árbol.

* Número de cadenas de longitud $n*2$ construidas con n ceros y n unos tales que ningún prefijo tenga más ceros que unos.

* Número de expresiones que contienen n pares de paréntesis bien anidados.

* Número de maneras de asociar $n+1$ factores en la aplicación de un operador binario asociativo.

* Número de árboles binarios de $n+1$ hojas donde cada nodo tiene cero o dos hijos.

* Número de caminos monótonos en una malla $n \times n$ entre esquinas opuestas, sin cruzar la diagonal.

* Número de maneras de dividir un polígono convexo de $n+2$ lados en triángulos con líneas de vértice a vértice que no se crucen.

☑ Factorial Cuádruple (Quadruple Factorial)

Corresponde a variantes etiquetadas de los problemas asociados a los números de Catalán.

$\text{quadrupleFact}(n)=\text{catalan}(n)*(n+1)!=(n*2)!/n!$

Usos:

* Número de árboles binarios que pueden construirse usando n elementos: $\text{quadrupleFact}(n)/(n+1)$

BIGINTEGER

☑ Raíz cuadrada BigInteger {0(log₂(n))}

```

BigInteger integerSqrt(BigInteger n) { //  $n^{(1/2)}$ 
    for (BigInteger x=n,y; ; x=y)
        {y=x.add(n.divide(x)).shiftRight(1); if (y.compareTo(x)>=0) return x;}
}
☒ Raíz p-ésima BigInteger { $O(\log_2(n))$ }
BigInteger integerRoot(BigInteger n, int p) { //  $n^{(1/p)}$ 
    BigInteger p0=BigInteger.valueOf(p),p1=BigInteger.valueOf(p-1);
    for (BigInteger x=n,y; ; x=y) {
        y=x.multiply(p1).add(n.divide(x.pow(p-1))).divide(p0);
        if (y.compareTo(x)>=0) return x;
    }
}
☒ Multiplicación de números grandes – Karatsuba [A. A. Karatsuba (1960)] { $O(n^{\log_2(3)})$ }
BigInteger karatsuba(BigInteger x, BigInteger y) { // Código de Robert Sedgewick y Kevin Wayne
    int n=Math.max(x.bitLength(),y.bitLength()); if (n<=2000) return x.multiply(y);
    n=(n/2)+(n%2);
    BigInteger b=x.shiftRight(n),a=x.subtract(b.shiftLeft(n));
    BigInteger d=y.shiftRight(n),c=y.subtract(d.shiftLeft(n));
    BigInteger ac=karatsuba(a,c),bd=karatsuba(b,d),abcd=karatsuba(a.add(b),c.add(d));
    return ac.add(abcd.subtract(ac).subtract(bd).shiftLeft(n)).add(bd.shiftLeft(n*2));
}
☒ Fibonacci { $O(\log_2(n))$ }
BigInteger fib(int n) {
    BigInteger i=BigInteger.ONE,h=i,j=BigInteger.ZERO,k=j,t;
    for (; n>0; n/=2) {
        if (n%2==1)
            {j=i.multiply(h).add(j.multiply(k)).add(t=j.multiply(h)); i=i.multiply(k).add(t);}
        t=h.pow(2); h=k.multiply(h).multiply(BigInteger.valueOf(2)).add(t); k=k.pow(2).add(t);
    }
    return j;
}
☒ Suma, multiplicación, división y potenciación de números grandes
Véase la clase java.math.BigInteger.

```

STRINGOLOGY

```

☒ Bordes de cadenas – Knuth-Morris-Pratt [D. Knuth, V. Pratt, J. H. Morris (1977)] { $O(|W|)$ }
int[] getBorderArray(char[] W) {
    int[] T=new int[W.length+1]; T[0]=-1; T[1]=0;
    for (int i=2,j=0; i<=W.length; )
        {if (W[i-1]==W[j]) T[i++]=++j; else if (j>0) j=T[j]; else T[i++]=0;}
    return T;
}
☒ String Search – Knuth-Morris-Pratt [D. Knuth, V. Pratt, J. H. Morris (1977)] { $O(|W|)$ }
int indexOf(char[] S, char[] W, int[] T) { // Índice donde ocurre S en W. T=getBorderArray(W).
    if (S.length==0) return 0;
    for (int m=0,i=0; m+i<W.length; )
        {if (S[i]==W[m+i]) {if (++i==S.length) return m;} else {m+=i-T[i]; if (i>0) i=T[i];}}
    return -1;
}
☒ Período de una cadena { $O(|W|)$ }
int periodo(char[] W) {int t=W.length,k=t-getBorderArray(W)[t]; return t%k==0?t/k:1;}
☒ Búsqueda de patrones en una cadena – Aho-Corasick [A. Aho, M. Corasick (1975)] { $O(n+m+z)$ }
// Busca todas las ocurrencias de los patrones en el objetivo
// La respuesta es una lista de parejas de la forma <pos,ind> donde pos es una posición
// donde ocurre el ind-ésimo patrón.
// k=|patterns|, n=|patterns[0]|+|patterns[1]|+...+|patterns[k-1]|, m=|target|, z=|answer|
List<int[]> search(char[][] patterns, char[] target) {
    List<int[]> res=new ArrayList<int[]>(); Trie root=new Trie(),q=root,p;
    for (int k=0; k<patterns.length; k++) add(root,patterns[k],k);
    prepare(root);
    for (int j=0; j<target.length; j++) {
        while ((p=q.gotoF(root,target[j]))==null) q=q.failure;
        q=p; if (q.out!=null) for (int id:q.out) res.add(new int[]{j-patterns[id].length+1,id});
    }
}

```

```

    }
    return res;
}
void prepare(Trie root) {
    Queue<Trie> queue=new LinkedList<Trie>(); Trie r,v,p;
    if (root.nodes!=null) for (Trie t:root.nodes) {t.failure=root; queue.add(t);}
    while (!queue.isEmpty()) if ((r=queue.poll()).nodes!=null) for (Trie u:r.nodes) {
        queue.add(u);
        for (v=r.failure; (p=v.gotoF(root,u.ch))==null; ) v=v.failure;
        u.failure=p; if (p.out!=null) u.addOutputs(p.out);
    }
}
void add(Trie root, char[] a, int idPattern) {
    Trie u=root,t;
    for (char c:a)
        if ((t=u.get(c))==null) {
            u.nodes=u.nodes==null?new Trie[1]:Arrays.copyOf(u.nodes,u.nodes.length+1);
            (u=u.nodes[u.nodes.length-1]=new Trie()).ch=c;
        }
        else u=t;
    u.addOutputs(idPattern);
}
static class Trie { // Keyword tree
    char ch=0; Trie[] nodes=null; int[] out=null; Trie failure=null; Trie() {}
    void addOutputs(int...arr) {
        if (out==null) {out=arr; return;}
        int g=out.length,f=arr.length; System.arraycopy(arr,0,out=Arrays.copyOf(out,g+f),g,f);
    }
    Trie get(char c) {if (nodes!=null) for (Trie t:nodes) if (t.ch==c) return t; return null;}
    Trie gotoF(Trie root, char c) {Trie t=get(c); return t!=null?t:(this==root?root:null);}
}

```

PARSING

☑ **Parser de expresiones aritméticas que incluyen operadores +,-,*,/, paréntesis y constantes flotantes, que concuerda con el parser de Java para evaluar expresiones de este tipo {0(|s|)}**

```

double parse(String s) {return new SimpleParser().parse(new StringTokenizer(s,"+/*()",true));}
static class SimpleParser { // El menos unario - se detecta y se convierte en ~
    Stack<String> p0=new Stack<String>(); Stack<Double> pV=new Stack<Double>(); String opA=null;
    double parse(StringTokenizer st) {
        for (shift(st); opA!=null; )
            if (p0.isEmpty()||(f(p0.peek(),"+-")&&f(opA,"/*"))||f(opA,"~")) shift(st); else reduce();
        while (!p0.isEmpty()) reduce();
        return pV.pop();
    }
    void shift(StringTokenizer st) {
        if (opA!=null) p0.push(opA);
        int i=0; for (opA=null; st.hasMoreTokens(); i++) {
            String s=st.nextToken();
            if (f(s,"+/*")) opA=i>0?s:"~";
            else if (f(s,"(")) pV.push(new SimpleParser().parse(st));
            else if (!f(s,")")) pV.push(Double.parseDouble(s));
            if (f(s,"+/*")) break;
        }
    }
    void reduce() {
        char c=p0.pop().charAt(0);
        if (c=='~') {pV.push(-pV.pop()); return;}
        double b=pV.pop(),a=pV.pop();
        pV.push(c=='+'?a+b:(c=='-'?a-b:(c=='*'?a*b:a/b)));
    }
    boolean f(String s, String f) {return f.indexOf(s)>=0;}
}

```

☑ **Gramáticas libres de contexto - Algoritmo CYK [Cocke, Younger, Kasami] {0(n³)}**

TEORIA DE JUEGOS

☒ Juegos de suma cero

☒ Nim

<http://en.wikipedia.org/wiki/Nim>

From Wikipedia, the free encyclopedia

Nim is a two-player mathematical game of strategy in which players take turns removing objects from distinct heaps. On each turn, a player must remove at least one object, and may remove any number of objects provided they all come from the same heap.

Variants of Nim have been played since ancient times. The game is said to have originated in China (it closely resembles the Chinese game of "Jianshizi", or "picking stones"), but the origin is uncertain; the earliest European references to Nim are from the beginning of the 16th century. Its current name was coined by Charles L. Bouton of Harvard University, who also developed the complete theory of the game in 1901, but the origins of the name were never fully explained. The name is probably derived from German *nimm!* meaning "take!", or the obsolete English verb *nim* of the same meaning. It should also be noted that turning the word NIM upside-down and backwards results in WIN (see Ambigram).

Nim is usually played as a *misère* game, in which the player to take the last object loses. Nim can also be played as a normal play game, which means that the person who makes the last move (i.e., who takes the last object) wins. This is called normal play because most games follow this convention, even though Nim usually does not.

Normal play Nim (or more precisely the system of numbers) is fundamental to the Sprague-Grundy theorem, which essentially says that in normal play every impartial game is equivalent to a Nim heap that yields the same outcome when played in parallel with other normal play impartial games (see disjunctive sum).

A version of Nim is played – and has symbolic importance– in the French New Wave film *Last Year at Marienbad* (1961).

Illustration

A normal play game may start with heaps of 3, 4 and 5 objects:

In order to win always leave an even number of 1's, 2's, and 4's.

Sizes of heaps Moves

```
A B C
3 4 5  I take 2 from A
1 4 5  You take 3 from C
1 4 2  I take 1 from B
1 3 2  You take 1 from B
1 2 2  I take entire A heap leaving two 2's.
0 2 2  You take 1 from B
0 1 2  I take 1 from C leaving two 1's. (In misère play I would take 2 from C leaving (0,1,0).)
0 1 1  You take 1 from B
0 0 1  I take entire C heap and win.
```

Mathematical theory

Nim has been mathematically solved for any number of initial heaps and objects; that is, there is an easily-calculated way to determine which player will win and what winning moves are open to that player. In a game that starts with heaps of 3, 4, and 5, the first player will win with optimal play, whether the *misère* or normal play convention is followed.

The key to the theory of the game is the binary digital sum of the heap sizes, that is, the sum (in binary) neglecting all carries from one digit to another. This operation is also known as exclusive or (xor) or vector addition over GF(2). Within combinatorial game theory it is usually called the *nim-sum*, as will be done here. The nim-sum of x and y is written $x \oplus y$ to distinguish it from the ordinary sum, $x + y$. An example of the calculation with heaps of size 3, 4, and 5 is as follows:

Binary Decimal

```
0112   310   Heap A
1002   410   Heap B
1012   510   Heap C
---
```

```
0102   210   The nim-sum of heaps A, B, and C,  $3 \oplus 4 \oplus 5 = 2$ 
```

An equivalent procedure, which is often easier to perform mentally, is to express the heap sizes as sums of distinct powers of 2, cancel pairs of equal powers, and then add what's left:

```
3 = 0 + 2 + 1 =      2   1   Heap A
4 = 4 + 0 + 0 = 4      Heap B
5 = 4 + 0 + 1 = 4      1   Heap C
```

2 = 2 What's left after cancelling 1s and 4s

In normal play, the winning strategy is to finish every move with a Nim-sum of 0, which is always possible if the Nim-sum is not zero before the move. If the Nim-sum is zero, then the next player will lose if the other player does not make a mistake. To find out which move to make, let X be the Nim-sum of all the heap sizes. Take the Nim-sum of each of the heap sizes with X , and find a heap whose size decreases. The winning strategy is to play in such a heap, reducing that heap to the Nim-sum of its original size with X . In the example above, taking the Nim-sum of the sizes is $X = 3 \oplus 4 \oplus 5 = 2$. The Nim-sums of the heap sizes $A=3$, $B=4$, and $C=5$ with $X=2$ are

$$A \oplus X = 3 \oplus 2 = 1 \quad B \oplus X = 4 \oplus 2 = 6 \quad C \oplus X = 5 \oplus 2 = 7$$

The only heap that is reduced is heap A, so the winning move is to reduce the size of heap A to 1 (by removing two objects).

As a particular simple case, if there are only two heaps left, the strategy is to reduce the number of objects in the bigger heap to make the heaps equal. After that, no matter what move your opponent makes, you can make the same move on the other heap, guaranteeing that you take the last object.

When played as a *misère* game, Nim strategy is different only when the normal play move would leave no heap of size 2 or larger. In that case, the correct move is to leave an odd number of heaps of size 1 (in normal play, the correct move would be to leave an even number of such heaps).

In a *misère* game with heaps of sizes 3, 4 and 5, the strategy would be applied like this:

| A | B | C | Nim-sum | |
|---|---|---|----------------|--|
| 3 | 4 | 5 | $010_2=2_{10}$ | I take 2 from A, leaving a sum of 000, so I will win. |
| 1 | 4 | 5 | $000_2=0_{10}$ | You take 2 from C |
| 1 | 4 | 3 | $110_2=6_{10}$ | I take 2 from B |
| 1 | 2 | 3 | $000_2=0_{10}$ | You take 1 from C |
| 1 | 2 | 2 | $001_2=1_{10}$ | I take 1 from A |
| 0 | 2 | 2 | $000_2=0_{10}$ | You take 1 from C |
| 0 | 2 | 1 | $011_2=3_{10}$ | The normal play strategy would be to take 1 from B, leaving an even number (2) heaps of size 1. For <i>misère</i> play, I take the entire B heap, to leave an odd number (1) of heaps of size 1. |
| 0 | 0 | 1 | $001_2=1_{10}$ | You take 1 from C, and lose. |

Proof of the winning formula

The soundness of the optimal strategy described above was demonstrated by C. Bouton.

Theorem. In a normal Nim game, the first player has a winning strategy if and only if the nim-sum of the sizes of the heaps is nonzero. Otherwise, the second player has a winning strategy.

Proof: Notice that the nim-sum (\oplus) obeys the usual associative and commutative laws of addition ($+$), and also satisfies an additional property, $x \oplus x = 0$ (technically speaking, the nonnegative integers under \oplus form an Abelian group of exponent 2).

Let x_1, \dots, x_n be the sizes of the heaps before a move, and y_1, \dots, y_n the corresponding sizes after a move. Let $s = x_1 \oplus \dots \oplus x_n$ and $t = y_1 \oplus \dots \oplus y_n$. If the move was in heap k , we have $x_i = y_i$ for all $i \neq k$, and $x_k > y_k$. By the properties of \oplus mentioned above, we have

$$\begin{aligned} t &= 0 \oplus t = s \oplus s \oplus t = s \oplus (x_1 \oplus \dots \oplus x_n) \oplus (y_1 \oplus \dots \oplus y_n) \\ &= s \oplus (x_1 \oplus y_1) \oplus \dots \oplus (x_n \oplus y_n) = s \oplus 0 \oplus \dots \oplus 0 \oplus (x_k \oplus y_k) \oplus 0 \oplus \dots \oplus 0 \\ &= s \oplus x_k \oplus y_k \end{aligned}$$

$$(*) \quad t = s \oplus x_k \oplus y_k.$$

The theorem follows by induction on the length of the game from these two lemmata.

Lemma 1. If $s = 0$, then $t \neq 0$ no matter what move is made.

Proof: If there is no possible move, then the lemma is vacuously true (and the first player loses the normal play game by definition). Otherwise, any move in heap k will produce $t = x_k \oplus y_k$ from (*). This number is nonzero, since $x_k \neq y_k$.

Lemma 2. If $s \neq 0$, it is possible to make a move so that $t = 0$.

Proof: Let d be the position of the leftmost (most significant) nonzero bit in the binary representation of s , and choose k such that the d th bit of x_k is also nonzero. (Such a k must exist, since otherwise the d th bit of s would be 0.) Then letting $y_k = s \oplus x_k$, we claim that $y_k < x_k$: all bits to the left of d are the same in x_k and y_k , bit d decreases from 1 to 0 (decreasing the value by 2^d), and any change in the remaining bits will amount to at most $2^k - 1$. The first player can thus make a move by taking $x_k - y_k$ objects from heap k , then

$$t = s \oplus x_k \oplus y_k \quad (\text{by } (*)) = s \oplus x_k \oplus (s \oplus x_k) = 0.$$

The modification for *misère* play is demonstrated by noting that the modification first arises in a position that has only one heap of size 2 or more. The normal play strategy is for the player to reduce this to size 0 or 1, leaving an even number of heaps with size 1, and the *misère* strategy is to do the opposite. From that point on, all moves are forced.

Other variations of the game

In another game which is commonly known as Nim (but is better called the subtraction game $S(1,2,\dots,k)$), an upper bound is imposed on the number of stones that can be removed in a turn. Instead of removing arbitrarily many stones, a player can only remove 1 or 2 or ... or k at a time. This game is commonly played in practice with only one heap (for instance with $k = 3$ in the game Thai 21 on Survivor: Thailand, where it appeared as an Immunity Challenge).

Bouton's analysis carries over easily to the general multiple-heap version of this game. The only difference is that as a first step, before computing the Nim-sums, we must reduce the sizes of the heaps modulo $k + 1$. If this makes all the heaps of size zero (in misère play), the winning move is to take k objects from one of the heaps. In particular, in a play from a single heap of n stones, the second player can win iff

$n \equiv 0 \pmod{k+1}$ (in normal play), or $n \equiv 1 \pmod{k+1}$ (in misère play).

VARIOS

Series

$$\begin{aligned} (\sum_{i|1 \leq i \leq n: i} i) &= n(n+1)/2 & (\sum_{i|1 \leq i \leq n: i^2} i^2) &= n(n+1)(2n+1)/6 \\ (\sum_{i|1 \leq i \leq n: i^3} i^3) &= n^2(n+1)^2/4 & (\sum_{i|1 \leq i \leq n: i^4} i^4) &= n(n+1)(2n+1)(3n^2+3n-1)/30 \\ (\sum_{i|1 \leq i \leq n: i^5} i^5) &= n^2(n+1)^2(2n^2+2n-1)/12 & (\sum_{i|1 \leq i \leq n: i^6} i^6) &= n(n+1)(2n+1)(3n^4+6n^3-3n+1)/42 \\ (\sum_{i|1 \leq i \leq n: i^7} i^7) &= n^2(n+1)^2(3n^4+6n^3-n^2-4n+2)/24 & (\sum_{i|1 \leq i \leq n: i^8} i^8) &= n(n+1)(2n+1)(5n^6+15n^5+5n^4-15n^3-n^2+9n-3)/90 \\ (\sum_{i|0 \leq i \leq n: A^i} A^i) &= (A^{n+1}-1)/(A-1) & (\sum_{i|1 \leq i \leq n: iA^i} iA^i) &= (n(A^{n+2}-A^{n+1})-A^{n+1}+A)/(A-1)^2 \\ (\sum_{i|0 \leq i \leq \infty: i \cdot A^i} i \cdot A^i) &= A/(1-A)^2 \\ (\sum_{i|0 \leq i \leq n: \text{binomial}(n,r) x^i y^{n-i}} x^i y^{n-i}) &= (x+y)^n & (\sum_{i|0 \leq i \leq n: \text{binomial}(n,r)} 1) &= 2^n \end{aligned}$$

Decimal a Romano

```
String[] r1={"I","X","C","M"},r2={"V","L","D"};
String intToRomano(int n) {
    if (n<=0 || n>=4000) return null; String s="";
    for (int i=0; n>0; i++, n/=10) {
        int d=(n%10),u=d%5;
        if (u==4) s=r1[i]+(d==4?r2[i]:r1[i+1])+s;
        else {for (int k=0; k<u; k++) s=r1[i]+s; if (d>4) s=r2[i]+s;}
    }
    return s;
}
```

Romano a Decimal

```
Map<String,Integer> mapRomanos=new TreeMap<String,Integer>();
int romanoToInt(String s) {
    if (mapRomanos.isEmpty()) for (int i=1; i<4000; i++) mapRomanos.put(intToRomano(i),i);
    return mapRomanos.containsKey(s)?mapRomanos.get(s):-1;
}
```

Postorden dado preorden e inorden

```
String postOr(String preOr, int i1, int t1, String inOr, int i2, int t2) {
    if (t1==0) return "";
    char r=preOr.charAt(i1); int k=inOr.indexOf(r,i2)-i2;
    return postOr(preOr,i1+1,k,inOr,i2,k)+postOr(preOr,i1+k+1,t1-k-1,inOr,i2+k+1,t2-k-1)+r;
}
```

Disjoint-set data structure {union,find: $O(\alpha(n))=O(\text{ackermann}^{-1}(n,n))$ }

```
static class DisjointSet {
    DisjointSet parent=this; int rank=0;
    DisjointSet() {}
    static void union(DisjointSet x, DisjointSet y) {
        DisjointSet xr=find(x),yr=find(y);
        if (xr.rank>yr.rank) yr.parent=xr;
        else if (xr.rank<yr.rank) xr.parent=yr;
        else if (xr!=yr) {yr.parent=xr; xr.rank++;}
    }
    static DisjointSet find(DisjointSet x) {return x.parent==x?(x.parent=find(x.parent));}
}
```

Sat-2 (2-satisfiability)

```
//-----
// Determinar si una expresión booleana en forma normal conjuntiva es satisfacible.
// Sean p[1],...,p[n] variables booleanas.
// Cada pareja <i,j> en 'clausulas' (1<=|i|<=n,1<=|j|<=n) representa la cláusula:
// p[|i|]v p[|j|] si i>0,j>0 -p[|i|]v p[|j|] si i<0,j>0
```

```
//      p[|i|]v~p[|j|] si i>0,j<0      ~p[|i|]v~p[|j|] si i<0,j<0
// Si la fórmula es insatisfacible, retorna null. Si la fórmula es satisfacible, retorna un
// estado que la satisface, donde estado[k-1] es el valor de la variable p[k] (k=1..n).
boolean[] sat2(int[][] clausulas, int n) {
    int t=clausulas.length,estado[]=new int[n]; boolean res[]=new boolean[n];
    if (!sat2(clausulas,estado,0)) return null;
    for (int i=0; i<n; i++) res[i]=estado[i]==2; return res;
}
boolean or(int a, int b) {return a==2||b==2;} // 0: sin valor, 1: false, 2: true
boolean sat2(int[][] m, int[] estado, int k) {
    if (k==m.length) return true;
    int i1=m[k][0],i2=m[k][1],a1=Math.abs(i1)-1,a2=Math.abs(i2)-1,e1=estado[a1],e2=estado[a2];
    for (int v1=1; v1<=2; v1++) for (int v2=1; v2<=2; v2++)
        if ((e1==0||(e1==v1))&&(e2==0||(e2==v2))&&or(i1<0?3-v1:v1,i2<0?3-v2:v2)&&(a1!=a2||v1==v2))
            {estado[a1]=v1; estado[a2]=v2; if (sat2(m,estado,k+1)) return true;}
    estado[a1]=e1; estado[a2]=e2; return false;
}
```

☑ Teorema maestro

(Tomado de la referencia [Cor2001])

Sean $a \geq 1$ y $b > 1$ constantes, $f(n)$ una función y $T(n)=aT(n/b)+f(n)$ una ecuación de recurrencia definida en los enteros no negativos. $T(n)$ puede ser acotado asintóticamente así:

1. Si $f(n)=O(n^{\log_b(a)-\epsilon})$ para alguna constante $\epsilon > 0$, entonces $T(n)=\Theta(n^{\log_b(a)})$.
2. Si $f(n)=\Theta(n^{\log_b(a)})$, entonces $T(n)=\Theta(n^{\log_b(a)} \log(n))$.
3. Si $f(n)=\Omega(n^{\log_b(a)+\epsilon})$ para alguna constante $\epsilon > 0$ y $a*f(n/b) \leq c*f(n)$ para alguna constante $c < 1$ y n suficientemente grande, entonces $T(n)=\Theta(f(n))$.

☑ Código gray

```
long gray(long n) {return n^(n>>1);}
long inverseGray(long n) {
    for (long ish=1,ans=n,ivid; true; ish<=&1)
        {ans^=(ivid=ans>>ish); if (ivid<=1||ish==32) return ans;}
}
```

☑ Polynominoes

Free: 1,1,1,2,5,12,35,108,369,1285,4655,17073,63600,238591,901971,3426576,13079255,50107909,192622052,742624232,2870671950,11123060678,43191857688,168047007728,654999700403,2557227044764,9999088822075,39153010938487,153511100594603

One-Sided: 1,1,2,7,18,60,196,704,2500,9189,33896,126759,476270,1802312,6849777,26152418,100203194,385221143,1485200848,5741256764,22245940545,86383382827,336093325058,1309998125640

Fixed: 1,2,6,19,63,216,760,2725,9910,36446,135268,505861,1903890,7204874,27394666,104592937,400795844,1540820542,5940738676,22964779660,88983512783,345532572678,1344372335524,5239988770268,20457802016011,79992676367108,313224032098244,1228088671826973

With holes: 0,0,0,0,0,0,1,6,37,195,979,4663,21474,96496,425449,1849252,7946380,33840946,143060339,601165888,2513617990,10466220315,43425174374

BIBLOGRAFÍA, RECURSOS

Textos

- [Ber2008] "Computational Geometry: Algorithms and Applications" de Mark de Berg.
- [Cor2001] "Introduction to Algorithms" de Thomas Cormen et. al.
- [Car1993] "ALGUNOS ALGORITMOS LOGARÍTMICOS" de Rodrigo Cardoso.
- [Gro2004] "Handbook of Graph Theory" de Jonathan L. Gross y Jay Yellen.
- [Sed2004] "Algorithms in Java. Part 5: Graph Algorithms" de Robert Sedgewick.
- [Ski1998] "The Algorithm Design Manual" de Steven S. Skiena.
- [Ski2003] "Programming Challenges" de Steven S. Skiena y Miguel A. Revilla.
- [Vil1996] "Diseño y Manejo de Estructuras de Datos en C" de Jorge A. Villalobos C.

Enlaces

A Tutorial on Dynamic Programming

<http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html>

ACM Solver

<http://www.acmsolver.org/>

ACMBEGINNER : ACM Programming Tutorial Site for Valladolid Online Judge

<http://www.acmbeginner.tk/>

Algorithm Tutorials

http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=alg_index

Algorithmist.com

<http://www.algorithmist.com/>
Computing intersections in a set of line segments: the Bentley-Ottmann algorithm
<http://www.scs.carleton.ca/~michiel/lecturenotes/ALGGEOM/bentley-ottmann.pdf>
Elementary Graph Algorithms
<http://www.cse.ohio-state.edu/~lai/780/9.graph.pdf>
Fibonacci number(fast method)
<http://wiki.cs.cityu.edu.hk/acm/>
Ford-Fulkerson's Algorithm
<http://www-b2.is.tokushima-u.ac.jp/~ikedasuuri/maxflow/Maxflow.shtml.en>
Geometry Algorithms
<http://geometryalgorithms.com/>
Geometry Algorithm Archive
http://www.geometryalgorithms.com/algorithm_archive.htm
Implementations of Algorithms For Line-Segment Intersection
<http://www.dike.de/~skanthak/university/cpsc516.pdf>
<http://www.dike.de/~skanthak/university/lineseg.tar.gz>
Intersection point of two lines (2 dimensions)
<http://local.wasp.uwa.edu.au/~pbourke/geometry/lineline2d/>
Karatsuba.java
<http://www.cs.princeton.edu/introcs/78crypto/Karatsuba.java.html>
Least Squares Fitting--Polynomial
<http://mathworld.wolfram.com/LeastSquaresFittingPolynomial.html>
Mathworld
<http://mathworld.wolfram.com/Circle-CircleIntersection.html>
<http://mathworld.wolfram.com/LeastSquaresFitting.html>
<http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html>
<http://mathworld.wolfram.com/Triangle.html>
Minimal Enclosing Circle - Modern Solutions
<http://www.cs.mcgill.ca/~cs507/projects/1998/jacob/solutions.html>
Parker-Traub algorithm for inversion of Vandermonde and related matrices
<http://www.math.uconn.edu/~olshevsky/papers/m5.pdf>
PNPOLY - Point Inclusion in Polygon Test
http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html
QRDecomposition.java
<http://www.docjar.com/html/api/jmat/data/matrixDecompositions/QRDecomposition.java.html>
Set Matching and Aho-Corasick Algorithm
<http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf>
The Convex Hull of a 2D Point Set or Polygon
http://geometryalgorithms.com/Archive/algorithm_0109/algorithm_0109.htm
The Traveling Salesperson Problem
<http://mat.gsia.cmu.edu/classes/dynamic/node8.html>
Topcoder
<http://www.topcoder.com>
http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=alg_index
Topological sort: implementation
<http://www.brpreiss.com/books/opus5/html/page556.html>
Voronoi Diagram using Divide-and-Conquer Paradigm
<http://www.personal.kent.edu/~rmuhamma/Compgeomerty/MyCG/Voronoi/DivConqVor/divConqVor.htm>
Voronoi Diagrams and a Day at the Beach
<http://www.ams.org/featurecolumn/archive/voronoi.html>
Wikipedia (<http://en.wikipedia.org/wiki/>)

| | |
|---|--------------------------------|
| Aho-Corasick_algorithm | Breadth-first_search |
| Chinese_remainder_theorem | Disjoint-set_data_structure |
| Edmonds-Karp_algorithm | Eulerian_path |
| Euler%27s_totient_function | Extended_Euclidean_algorithm |
| Floyd-Warshall_algorithm | Gray_code |
| Hopcroft-Karp_algorithm | Knuth-Morris-Pratt_algorithm |
| Kruskal%27s_algorithm | List_of_algorithms |
| Longest_common_substring_problem | Longest_increasing_subsequence |
| http://es.wikipedia.org/wiki/N%C3%BAmeros_de_Catalan | |
| Permutation | Prim%27s_algorithm |
| Selection_algorithm | Simpson's_rule |
| Vandermonde_matrix | |