

Resumen de algoritmos para maratones de programación

Universidad Tecnológica de Pereira

17 de octubre de 2012

Índice

1. Plantilla	1		
2. Grafos	2		
2.1. Depth First Search	2		
2.2. Breadth First Search	2		
2.3. Shortest path problem	3		
2.3.1. Dijkstra	3		
2.3.2. Bellman-Ford	3		
2.4. All-pairs shortest paths	4		
2.4.1. Floyd-Warshall	4		
2.4.2. Johnson	5		
2.5. Minimum Spanning Tree	5		
2.5.1. Algoritmo de kruskal	5		
2.5.2. Algoritmo de Prim	6		
2.6. Strongly Connected Components	7		
2.7. Puntos de articulación	8		
2.8. 2-SAT	8		
2.9. Maximum bipartite: Kuhn's algorithm	9		
2.10. Flujo Máximo	10		
2.11. Lowest Common Ancestor	11		
2.12. Topological sort	13		
2.13. Shortest pair of edge disjoint paths	13		
2.14. Shortest pair of completely disjoint paths	13		
2.15. Eulerian path	14		
3. Programación Dinámica	14		
3.1. Edit Distance	14		
3.2. Integer Knapsack Problem	15		
3.3. Counting Boolean Parenthesizations	15		
3.4. Longest Increasing Subsequence	16		
		3.5. Josephus problem	16
		4. Matemáticas	17
		4.1. Fórmulas útiles	17
		4.2. Aritmética Modular	17
		4.3. Mayor exponente de un primo que divide a $n!$	18
		4.4. Potencia modular	18
		4.5. Criba de Eratóstenes	18
		4.6. Test de primalidad	19
		4.7. Combinatoria	20
		4.7.1. Sumas	20
		4.7.2. Cuadro Resumen	20
		5. Geometría	20
		5.1. Utilidades Geometría	20
		5.2. Transformaciones	21
		5.2.1. Rotación	21
		5.2.2. Desplazamiento dos dimensiones	21
		5.3. Distancia mínima: Punto-Segmento	21
		5.4. Distancia mínima: Punto-Recta	21
		5.5. Determinar si un polígono es convexo	22
		5.6. Determinar si un punto está dentro de un polígono convexo	22
		5.7. Determinar si un punto está dentro de un polígono cualquiera	22
		5.8. Intersección de dos rectas	23
		5.9. Intersección de dos segmentos	24
		5.10. Determinar si dos segmentos se intersectan o no	24
		5.11. Centro del círculo que pasa por tres puntos.	25
		5.12. Par de puntos más cercanos	25
		5.13. Par de puntos más alejados	26
		5.14. Área de un polígono	27
		5.15. Convexhull	27
		5.15.1. Graham Scan	27

5.16. Great circle distance	30
5.17. Picks theorem	31
6. Strings	31
6.1. Knuth-Morris-Pratt KMP	31
6.2. Aho-Corasick	32
6.3. Suffix Array	34
6.4. Minimum string rotation	35
7. Teoría de Juegos	35
8. Estructuras de Datos	35
8.1. RMQ	35
8.2. Union-find (disjoint-set)	36
8.3. Prefix Tree - Trie	36
8.4. Fenwick Tree	37
8.5. Interval Tree	38
9. Hashing	38
9.1. FNV Hash	38
9.2. JSW Hash	38
10. Miseláneo	38
10.1. Bitwise operations	38
10.2. Física	39
10.2.1. Movimiento parabólico	39
10.3. Stable marriage	39
10.4. Poker	40
10.5. Inversions	42
10.6. Nim	43
10.7. Sudoku	44
10.8. Gaussian elimination	48
10.9. Catalan numbers	49
10.10 Bell numbers	50
10.11 Polinomios	50

1. Plantilla

```
#include <cstdio>
#include <cmath>
#include <functional>
#include <algorithm>
```

```
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <list>
#include <map>

using namespace std;

#define all(x) x.begin(),x.end()
#define rep(i,a,b) for(int i=a;i<b;i++)
#define REP(i,n) rep(i,0,n)
#define foreach(x, v) for (typeof (v).begin() x = (v).begin(); \
x != (v).end(); ++x)
#define D(x) cout << #x " = " << x << endl;

typedef long long int lld;
typedef pair<int,int> pii;
typedef vector<int> vi;
typedef vector<pii> vpii;

int main(){

    return 0;
}

static class Scanner {

    BufferedReader br =
new BufferedReader(new InputStreamReader(System.in));

    StringTokenizer st = new StringTokenizer("");

    String nextLine() {
        try {
            String l = br.readLine();
            return l;
        } catch (Exception e) {
            throw new RuntimeException();
        }
    }
}
```

```

    }
}
public String next() {
    while (!st.hasMoreTokens()) {
        String linea = nextLine();
        if (linea == null) return null;
        st = new StringTokenizer(linea);
    }
    return st.nextToken();
}

public int nextInt() {
    return Integer.parseInt(next());
}

public double nextDouble() {
    return Double.parseDouble(next());
}

// Y asi para cualquier next (por ejemplo: nextBigInteger)

// Retorna null si es el fin de la entrada
public Integer nextInteger() {
    String l = next();
    if(l == null)
        return null;
    return Integer.parseInt(l);
}
}
.....

```

2. Grafos

2.1. Depth First Search

Es un algoritmo para recorrer o buscar en un grafo. Empieza visitando la raíz y luego explora a fondo cada rama antes de hacer el backtraing.

Complejidad $O(|V| + |E|)$

```

/*
recordar
#define all(x) x.begin(),x.end()
typedef vector<int> vi;
typedef vector<vi> vvi;
*/
int N; // Número de vertices.
vvi G; // Grafo.
vi visited; // Vector de visitados.

void dfs(int i) {
    if(!visited[i]) {
        visited[i] = true;
        for_each(all(G[i]), dfs);
    }
}

bool check_graph_connected_dfs() {
    int start_vertex = 0;
    visited = vi(N, false);
    dfs(start_vertex);
    return (find(all(V), 0) == V.end());
}

```

.....

2.2. Breadth First Search

Es un algoritmo de búsqueda que empieza en la raíz y explora todos los nodos vecinos. Luego para cada vecino repite el proceso hasta encontrar la meta.
Complejidad $O(|V| + |E|)$

```

/*
Se considera un Grafo NO dirigido.
*/

int N; // number of vertices
vvi G; // lists of adjacent vertices

void bfs(){
    int start_vertex = 0;

```

```

vi V(N, false);
queue<int> Q;
Q.push(start_vertex);
V[start_vertex] = true;
while(!Q.empty()) {
    int i = Q.front();
    Q.pop();
    foreach(G[i], it) {
        if(!V[*it]) {
            V[*it] = true;
            Q.push(*it);
        }
    }
}
}

```

2.3. Shortest path problem

El shortest path problem es el problema de encontrar el camino mas corto entre dos nodos de un grafo. Existen dos algoritmos para solucionarlo: Dijkstra y Bellman-Ford. Si bien Dijkstra tiene una complejidad mejor que Bellman-Ford, no funciona para grafos con aristas con peso negativo mientras que Bellman-Ford si lo hace.

2.3.1. Dijkstra

Calcula la ruta más corta a todos los nodos desde un origen. Todas las aristas deben ser NO negativas, en ese caso usar Bellman-ford. Complejidad $O(E \log V)$

```

/**
 * Recordar, typedef pair<int,int> pii;
 * typedef vector<pii> vii;
 * typedef vector<vii> vvii;
 */

vi D (N, 987654321);
vi P (N, -1);
// D[i] es la distancia desde el vértice inicial hasta i
// P[i] es el predecesor de i en la ruta más corta

```

```

priority_queue < pii, vector < pii >, greater < pii > >Q;
//Cola de prioridad con comparador Mayor,
//top() retorna la distancia más lejana.
//Inicializar el vértice inicial.
D[0] = 0;
Q.push (pii (0, 0));
while (!Q.empty ()) {
    pii top = Q.top ();
    Q.pop ();
    int v = top.second, d = top.first;
    if (d <= D[v]) {
        foreach(G[v], it) {
            int v2 = it->first, cost = it->second;
            if (D[v2] > D[v] + cost) {
                D[v2] = D[v] + cost;
                P[v2] = v;
                Q.push (pii (D[v2], v2));
            }
        }
    }
}
}

```

//Código basado en TopCoder Algorithms tutorials.

2.3.2. Bellman-Ford

Este algoritmo calcula la ruta más corta en un grafo dirigido, en el cual el peso de las aristas puede ser positivo o negativo. Para grafos con pesos NO-negativos, el algoritmo de Dijkstra resuelve más rápido este problema.

Si un grafo contiene un “Ciclo negativo”, por ejemplo, un ciclo cuya suma de aristas sea un valor negativo, entonces camina de forma arbitraria con los pesos que puede ser construida, por ejemplo, no puede haber camino más corto. El algoritmo puede detectar ciclos negativos y reportar su existencia, pero no puede producir una respuesta correcta si un ciclo negativo es alcanzable desde el nodo origen.

Para solucionar el longest path, simplemente se aplica bellman-ford con los pesos de los caminos negativos.

```

procedure BellmanFord(list vertices, list edges, vertex source)

```

```

// This implementation takes in a graph, represented
// as lists of vertices
// and edges, and modifies the vertices so that
// their distance and
// predecessor attributes store the shortest paths.
// Step 1: initialize graph
for each vertex v in vertices:
    if v is source then v.distance := 0
    else v.distance := infinity
    v.predecessor := null

// Step 2: relax edges repeatedly
for i from 1 to size(vertices)-1:
    for each edge uv in edges: // uv is the edge from u to v
        u := uv.source
        v := uv.destination
        if u.distance + uv.weight < v.distance:
            v.distance := u.distance + uv.weight
            v.predecessor := u

// Step 3: check for negative-weight cycles
for each edge uv in edges:
    u := uv.source
    v := uv.destination
    if u.distance + uv.weight < v.distance:
        error "Graph contains a negative-weight cycle"

```

2.4. All-pairs shortest paths

Es una versión del shortest path problem donde hay que hayar la menor distancia entre todos los nodos.

Existen dos algoritmos para solucionarlo, Floyd-Warshall y Jhonson. Floyd-Warshall lo hace en tiempo cúbico, mientras que Jhonson en VxE , por lo que solo es útil si E es asintóticamente menor que V^2 (es decir, el grafo no es muy denso).

2.4.1. Floyd-Warshall

In computer science, the Floyd–Warshall algorithm (sometimes known as the WFI Algorithm or Roy–Floyd algorithm) is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights).

A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices. The algorithm is an example of dynamic programming.

Normal

```

/*
    Assume a function edgeCost(i,j) which returns the cost
    of the edge from i to j
    (infinity if there is none).
    Also assume that n is the number of vertices
    and edgeCost(i,i) = 0

*/

int path[][];

/*
    A 2-dimensional matrix. At each step in the algorithm,
    path[i][j] is the shortest path
    from i to j using intermediate vertices $(1 .. k - 1)$.
    Each path[i][j] is initialized to
    edgeCost(i,j) or infinity if there is no edge
    between i and j.
*/

procedure FloydWarshall ()
    for k := 1 to n
        for i := 1 to n
            for j := 1 to n
                path[i][j] = min ( path[i][j], path[i][k]+path[k][j]);

Con reconstrucción de path

procedure FloydWarshallWithPathReconstruction ()
    for k := 1 to n
        for i := 1 to n
            for j := 1 to n
                if path[i][k] + path[k][j] < path[i][j] then
                    path[i][j] := path[i][k]+path[k][j];

```

```

        next[i][j] := k;

procedure GetPath (i,j)
    if path[i][j] equals infinity then
        return "no path";
    int intermediate := next[i][j];
    if intermediate equals 'null' then
        return " ";
    else
        return GetPath(i,intermediate) + intermediate
        + GetPath(intermediate,j);

.....

```

2.4.2. Johnson

Johnson's algorithm is a way to find the shortest paths between all pairs of vertices in a sparse directed graph. It allows some of the edge weights to be negative numbers, but no negative-weight cycles may exist. It works by using the Bellman–Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph.

Pseudocódigo:

Johnson's algorithm consists of the following steps:

First, a new node q is added to the graph, connected by zero-weight edges to each other node.

Second, the Bellman–Ford algorithm is used, starting from the new vertex q , to find for each vertex v the least weight $h(v)$ of a path from q to v . If this step detects a negative cycle, the algorithm is terminated.

Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from u to v , having length $w(u,v)$, is given the new length $w(u,v) + h(u) - h(v)$.

Finally, q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the reweighted graph.

In the reweighted graph, all paths between a pair s and t of nodes have the same quantity $h(s) - h(t)$ added to them, so a path that is shortest in the original graph remains shortest in the modified graph and vice versa. However,

due to the way the values $h(v)$ were computed, all modified edge lengths are non-negative, ensuring the optimality of the paths found by Dijkstra's algorithm. The distances in the original graph may be calculated from the distances calculated by Dijkstra's algorithm in the reweighted graph by reversing the reweighting transformation.

The time complexity of this algorithm, using Fibonacci heaps in the implementation of Dijkstra's algorithm, is $O(V^2 \log V + VE)$: the algorithm uses $O(VE)$ time for the Bellman–Ford stage of the algorithm, and $O(V \log V + E)$ for each of V instantiations of Dijkstra's algorithm. Thus, when the graph is sparse, the total time can be faster than the Floyd–Warshall algorithm, which solves the same problem in time $O(V^3)$.

2.5. Minimum Spanning Tree

Given a connected, undirected graph, a spanning tree of that graph is a sub-graph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a weight to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a minimum spanning forest, which is a union of minimum spanning trees for its connected components. One example would be a cable TV company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths, then there would be a graph representing which points are connected by those paths. Some of those paths might be more expensive, because they are longer, or require the cable to be buried deeper; these paths would be represented by edges with larger weights. A spanning tree for that graph would be a subset of those paths that has no cycles but still connects to every house. There might be several spanning trees possible. A minimum spanning tree would be one with the lowest total cost.

Hay dos algoritmos para solucionar MST, Kruskal y Prim. Cada uno tiene sus ventajas y desventajas.

2.5.1. Algoritmo de kruskal

Kruskal's algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight

of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component). Kruskal's algorithm is an example of a greedy algorithm.

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's algorithm can be shown to run in $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures. These running times are equivalent because:

E is at most V^2 and $\log V^2 = 2\log V$ is $O(\log V)$. If we ignore isolated vertices, which will each be their own component of the minimum spanning forest, $V \leq E+1$, so $\log V$ is $O(\log E)$. We can achieve this bound as follows: first sort the edges by weight using a comparison sort in $O(E \log E)$ time; this allows the step remove an edge with minimum weight from S to operate in constant time. Next, we use a disjoint-set data structure (Union and Find) to keep track of which vertices are in which components. We need to perform $O(E)$ operations, two find operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform $O(E)$ operations in $O(E \log V)$ time. Thus the total time is $O(E \log E) = O(E \log V)$. Provided that the edges are either already sorted or can be sorted in linear time (for example with counting sort or radix sort), the algorithm can use more sophisticated disjoint-set data structure to run in $O(E \alpha(V))$ time, where α is the extremely slowly-growing inverse of the single-valued Ackermann function.

```
function Kruskal(G = <N, A>: graph; length): set of edges
2   Define an elementary cluster  $C(v) = \{v\}$ .
3   Initialize a priority queue  $Q$  to contain all edges in  $G$ ,
    using the weights as keys.
4   Define a forest  $T$  (empty)    //T will ultimately
                                //contain the edges of the MST
5   // n is total number of vertices
6   while  $T$  has fewer than  $n-1$  edges do
7   // edge  $u,v$  is the minimum weighted route from  $u$  to  $v$ 
8    $(u,v) = Q.removeMin()$ 
9   // prevent cycles in  $T$ . add  $u,v$  only if  $T$  does not
//already contain a path between  $u$  and  $v$ .
10  // the vertices has been added to the tree.
11  Let  $C(v)$  be the cluster containing  $v$ , and let  $C(u)$  be
the cluster containing  $u$ .
13  if  $C(v) \neq C(u)$  then
14      Add edge  $(v,u)$  to  $T$ .
15      Merge  $C(v)$  and  $C(u)$  into 1 cluster ( $union(C(v), C(u))$ )
16  return tree  $T$ 
```

Este algoritmo se usa con la estructura de datos disjoint-set.

.....

2.5.2. Algoritmo de Prim

Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm.

El algoritmo de Prim tiene exactamente el mismo tiempo de ejecución asintótico que el de Dijkstra y usar una Fibonnaci Heap o una Binary Heap tienen los mismos beneficios.

Min-heap

Initialization

inputs: A graph, a function returning edge weights weight-function, and an initial vertex

Initial placement of all vertices in the 'not yet seen' set, set initial vertex to be added to the tree, and place all vertices in a min-heap to allow for removal of the min distance from the minimum graph for each vertex in graph

```
    set min_distance of vertex to 8
    set parent of vertex to null
    set minimum_adjacency_list of vertex to empty list
    set is_in_Q of vertex to true
set min_distance of initial vertex to zero
add to minimum-heap  $Q$  all vertices in graph, keyed by min_distance
```

/*

Algorithm

In the algorithm description above, nearest vertex is $Q[0]$, now latest addition fringe is v in Q where distance of $v < 8$ after nearest vertex is removed not seen is v in Q where distance of $v = 8$ after nearest vertex is removed The while loop will fail when remove minimum returns null. The adjacency list is set to allow a directional graph to be returned.

time complexity: V for loop, $\log(V)$ for the remove function

```

*/

while latest_addition = remove minimum in Q

set is_in_Q of latest_addition to false

add latest_addition to (minimum_adjacency_list of
(parent of latest_addition))

add (parent of latest_addition) to
(minimum_adjacency_list of latest_addition)

for each adjacent of latest_addition
{

if (is_in_Q of adjacent) and
(weight-function(latest_addition, adjacent) <
min_distance of adjacent)
{

set parent of adjacent to latest_addition

set min_distance of adjacent to
weight-function(latest_addition, adjacent)

update adjacent in Q, order by min_distance

}

}

.....

```

2.6. Strongly Connected Components

A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. In particular, this means paths in each direction; a path from a to b and also a path from b to a. The strongly connected components of a directed graph G are its maximal strongly connected subgraphs.

If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph. A directed graph is acyclic if and only

if it has no (nontrivial) strongly connected subgraphs (because a cycle is strongly connected, and every strongly connected graph contains at least one cycle).

```

public class SCCTarjan {
    static class Node implements Comparable < Node > {
        final int name;
        ArrayList < Node > adjacents = new ArrayList < Node > (100);
        boolean visited = false;
        int lowlink = -1;
        int index = -1;

        public Node(final int argName) {
            name = argName;
        }

        public int compareTo(final Node argNode) {
            return argNode == this ? 0 : -1;
        }
    }

    static int index = 0;
    static ArrayDeque < Node > stack = new ArrayDeque < Node > ();

    static ArrayList < ArrayList < Node > > SCC =
new ArrayList < ArrayList < Node > > ();

    public static ArrayList < ArrayList < Node > >
tarjanSCC(ArrayList < Node > nodes)
{
    index = 0;
    SCC.clear();
    stack.clear();
    for (Node n: nodes)
        if (n.index == -1) tarjan(n);
    return SCC;
}

    public static void tarjan(Node v) {
        v.index = index;
        v.lowlink = index;
        index++;
        stack.push(v);
    }
}

```



```

    for (Node n: v.adjacents) {
        if (n.index == -1) {
            tarjan(n);
            v.lowlink = Math.min(v.lowlink, n.lowlink);
        }
    }
else if (stack.contains(n))
v.lowlink = Math.min(v.lowlink, n.index);
}
if (v.lowlink == v.index) {
    Node n;
    ArrayList < Node > component =
new ArrayList < Node > ();
    do {
        n = stack.pop();
        component.add(n);
    }
    while (n != v);
    SCC.add(component);
}
}
}
}

```

2.7. 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; //contendra los puntos de articulacion
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;
}

```

2.8. 2-SAT

para construir el grafo dirigido, se crea un nodo para cada variable y su negación, luego para cada disjunción se crean dos aristas así: desde la negación de la primera variable a la segunda, y de la negación de la segunda variable a la primera, esto indica, que si no se puede cumplir una de las variables hay que cumplir la otra: $(x_0 \vee \neg x_3)$ equiv $(\neg x_0 \wedge \neg x_3)$ equiv $(x_3 \wedge x_0)$

In terms of the implication graph, two terms belong to the same strongly connected component whenever there exist chains of implications from one term to the other and vice versa. Therefore, the two terms must have the same value in any satisfying assignment to the given 2-satisfiability instance. In particular, if a variable and its negation both belong to the same strongly connected component, the instance cannot be satisfied, because it is impossible to assign both of these terms the same value.

this is a necessary and sufficient condition: a 2-CNF formula is satisfiable if and only if there is no variable that belongs to the same strongly connected component as its negation.

Their algorithm performs the following steps:

- Construct the implication graph of the instance, and find its strongly connected components using any of the known linear-time algorithms for strong connectivity analysis. (Tarjan)
- Check whether any strongly connected component contains both a variable and its negation. If so, report that the instance is not satisfiable and halt.
- Construct the condensation of the implication graph, a smaller graph that has one vertex for each strongly connected component, and an edge from component i to component j whenever the implication graph contains an edge uv such that u belongs to component i and v belongs to component j . The condensation is automatically a directed acyclic graph and, like the implication graph from which it was formed, it is skew-symmetric.
- Topologically order the vertices of the condensation; the order in which the components are generated by Kosaraju's algorithm is automatically a topological ordering.
- For each component in this order, if its variables do not already have truth assignments, set all the terms in the component to be false. This also causes all of the terms in the complementary component to be set to true.

2.9. Maximum bipartite: Kuhn's algorithm

There is a bipartite graph containing N vertices (n vertices in left part and k ($N-n$) vertices in right part of graph) and M edges. We are to find maximum bipartite matching, i.e. mark maximum number of edges, so that no one of them have adjacent vertices with each other.

```
#include < vector >
#include < utility >
using namespace std;

class KuhnImplementation {
public: int n, k;
vector < vector < int > > g;
vector < int > pairs_of_right, pairs_of_left;
vector < bool > used;

bool kuhn(int v) {
    if (used[v]) return false;
    used[v] = true;
```

```
    for (int i = 0; i < g[v].size(); ++i) {
        int to = g[v][i] - n;
        if (pairs_of_right[to] == -1 || kuhn(pairs_of_right[to])){
            pairs_of_right[to] = v;
            pairs_of_left[v] = to;
            return true;
        }
    }
    return false;
}

vector <pair<int,int>> find_max_matching ...
    ...(vector<vector<int>> & _g, int _n, int _k){
    g = _g;
    //g[i] is a list of adjacent vertices to vertex i, where
    //i is from left part and g[i][j] is from right part
    n = _n;
    //n is number of vertices in left part of graph
    k = _k;
    //k is number of vertices in right part of graph

    pairs_of_right = vector <int> (k, - 1);
    pairs_of_left = vector <int> (n, - 1);
    //pairs_of_right[i] is a neighbor of vertex i from right part,
    //on marked edge. pairs_of_left[i] is a neighbor of vertex
    //i from left part, on marked edge
    used = vector < bool > (n, false);

    bool path_found;
    do {
        fill(used.begin(), used.end(), false);
        path_found = false;
    } while (!path_found);
    //remember to start only from free vertices which are not visited yet
    for (int i = 0; i < n; ++i)
        if (pairs_of_left[i] < 0 && !used[i])
            path_found |= kuhn(i);
    } while (!path_found);

    vector < pair < int, int > > res;
```

```

for (int i = 0; i < k; i++)
    if (pairs_of_right[i] != -1)
        res.push_back(make_pair(pairs_of_right[i], i+n));

return res;
}
};

```

2.10. Flujo Máximo

En términos de teoría de grafos, nos dan una red - un grafo dirigido, en donde cada arista tiene cierta capacidad c asociada a él, un vértice inicial (la fuente) y un vértice final (el sumidero). Se nos pide asociar otro valor f que satisfaga $f \leq c$ para cada arista de tal forma de que para cada vértice, diferente a la fuente y el sumidero, la suma de los valores asociados a las aristas que entran a él sea igual a la suma de los valores que salen. Se llamará f al flujo a través de la arista. Además se pide maximizar la suma de los valores asociados a los arcos que salen de la fuente, el cual es el flujo total en la red.

La función find path se puede implementar con un breadth first search, el cual garantiza que toma como máximo $N \times M/2$ pasos, donde N es el número de vértices y M es el número de aristas de la red. También se puede implementar con un priority first search, para lo cual hay que definir una estructura nodo que tenga los atributos vertice, prioridad y vertice anterior en el camino. Se usa un arreglo bidimensional para guardar las capacidades de la red residual tras cada paso del algoritmo.

```

int max_flow()
result = 0
while (true)
    augmenting path found
    path_capacity = find_path()
    if (d = 0) exit while
    else result += path_capacity
end while
return result

int bfs()
queue Q
push source to Q
mark source as visited

```

keep an array from with the semnification: from[x] is the previous vertex visited in the shortest path from the source to x; initialize from with -1 (or any other sentinel value)

```

while Q is not empty
    where = pop from Q
    for each vertex next adjacent to where
        if next is not visited and capacity[where][next] > 0
            push next to Q
            mark next as visited
            from[next] = where
            if next = sink
                exit while loop
    end for
end while
where = sink, path_cap = infinity
while from[where] > -1
    prev = from[where] // the previous vertex
    path_cap = min(path_cap, capacity[prev][where])
    where = prev
end while
while
loop will not be entered
    where = sink
    while from[where] > -1
        prev = from[where]
        capacity[prev][where] -= path_capacity
        capacity[where][prev] += path_capacity
        where = prev
    end while
    if path_cap = infinity
        return 0
    else return path_cap

```

```

int pfs()
priority queue PQ
push node(source, infinity, -1) to PQ
keep the array from as in bfs()
// if no augmenting path is found, path_cap will remain 0
path_cap = 0
while PQ is not empty

```

```

node aux = pop from PQ
where = aux.vertex, cost = aux.priority
if we already visited where continue
from[where] = aux.from
if where = sink
    path_cap = cost
    exit while loop
mark where as visited
for each vertex next adjacent to where
    if capacity[where][next] > 0
        new_cost = min(cost, capacity[where][next])
        push node(next, new_cost, where) to PQ
    end for
end while
// update the residual network
where = sink
while from[where] > -1
    prev = from[where]
    capacity[prev][where] -= path_cap
    capacity[where][prev] += path_cap
    where = prev
end while
return path_cap

```

.....

2.11. Lowest Common Ancestor

The lowest common ancestor (LCA) is a concept in graph theory and computer science. Let T be a rooted tree with n nodes. The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).

The LCA of v and w in T is the shared ancestor of v and w that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from v to w can be computed as the distance from the root to v , plus the distance from the root to w , minus twice the distance from the root to their lowest common ancestor.

In a tree data structure where each node points to its parent, the lowest common ancestor can be easily determined by finding the first intersection of the paths from v and w to the root. In general, the computational time required for this algorithm is $O(h)$ where h is the height of the tree (length of longest path from a

leaf to the root). However, there exist several algorithms for processing trees so that lowest common ancestors may be found more quickly, in constant time per query after a linear time preprocessing stage.

In computer science, Tarjan's off-line least common ancestors algorithm (more precisely, least should actually be lowest) is an algorithm for computing lowest common ancestors for pairs of nodes in a tree, based on the union-find data structure. The lowest common ancestor of two nodes d and e in a rooted tree T is the node g that is an ancestor of both d and e and that has the greatest depth in T . It is named after Robert Tarjan, who discovered the technique in 1979. Tarjan's algorithm is offline; that is, unlike other lowest common ancestor algorithms, it requires that all pairs of nodes for which the lowest common ancestor is desired must be specified in advance. The simplest version of the algorithm uses the union find data structure, which unlike other lowest common ancestor data structures can take more than constant time per operation when the number of pairs of nodes is similar in magnitude to the number of nodes. A later refinement by Gabow and Tarjan (1983) speeds the algorithm up to linear time.

The pseudocode below determines the lowest common ancestor of each pair in P , given the root r of a tree in which the children of node n are in the set $n.children$. For this offline algorithm, the set P must be specified in advance. It uses the MakeSet, Find, and Union functions of a disjoint-set forest. MakeSet(u) removes u to a singleton set, Find(u) returns the standard representative of the set containing u , and Union(u,v) merges the set containing u with the set containing v . TarjanOLCA(r) is first called on the root r .

```

function TarjanOLCA(u)
    MakeSet(u);
    u.ancestor := u;
    for each v in u.children do
        TarjanOLCA(v);
        Union(u,v);
        Find(u).ancestor := u;
    u.colour := black;
    for each v such that {u,v} in P do
        if v.colour == black
            print
Tarjans Least Common Ancestor of
    u
    and
    v
    is
    Find(v).ancestor + ".";

```

```

struct Query;

struct Nodo
{
    vector <Nodo*> adjacentes;
    vector <Query*> queries;
    int numero;
    long long distanciaCero;
    Nodo *parent;
    Nodo *ancestor;
    int rank;
    bool encontrado;

    void clear(int i)
    {
        numero = i;
        distanciaCero = 0;
        adjacentes.clear();
        queries.clear();
        encontrado = false;
    }
};

struct Query
{
    Nodo *a, *b;
    long long respuesta;
};

Nodo nodos[100001];
Query queries[100001];

void makeSet(Nodo *x)
{
    x->parent = x;
    x->rank = 0;
}

Nodo* find(Nodo *x)
{

```

```

    if(x->parent == x)
        return x;
    else
    {
        x->parent = find(x->parent);
        return x->parent;
    }
}

int unir(Nodo *x, Nodo *y)
{
    Nodo *xRoot = find(x);
    Nodo *yRoot = find(y);
    if(xRoot->rank > yRoot->rank)
        yRoot->parent = xRoot;
    else if(xRoot->rank < yRoot->rank)
        xRoot->parent = yRoot;
    else if(xRoot != yRoot)
    {
        yRoot->parent = xRoot;
        xRoot->rank = xRoot->rank + 1;
    }
}

void tarjanOLCA(Nodo *u)
{
    makeSet(u);
    u->ancestor = u;
    for(int i = 0; i < u->adjacentes.size(); i++)
    {
        tarjanOLCA(u->adjacentes[i]);
        unir(u, u->adjacentes[i]);
        find(u)->ancestor = u;
    }
    u->encontrado = true;
    for(int i = 0; i < u->queries.size(); i++)
    {
        Query *actual = u->queries[i];
        Nodo *v;
        if(u == actual->a)
            v = actual->b;

```

```

else
    v = actual->a;
    if(v->encontrado)
    {
        Nodo *ancestro = find(v)->ancestor;
        actual->respuesta = (procesar query aqui)
    }
}
}
}

```

2.12. Topological sort

In graph theory, a topological sort or topological ordering of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts. More formally, define the reachability relation R over the nodes of the DAG such that xRy if and only if there is a directed path from x to y . Then, R is a partial order, and a topological sort is a linear extension of this partial order, that is, a total order compatible with the partial order.

The usual algorithms for topological sorting have running time linear in the number of nodes plus the number of edges ($O(|V|+|E|)$). One of these algorithms, first described by Kahn (1962), works by choosing vertices in the same order as the eventual topological sort. First, find a list of start nodes which have no incoming edges and insert them into a set S ; at least one such node must exist if graph is acyclic. Then:

```

L <- Empty list that will contain the sorted elements
S <- Set of all nodes with no incoming edges
while S is non-empty do
    remove a node n from S
    insert n into L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    output error message (graph has at least one cycle)

```

```

else
    output message (proposed topologically sorted order: L)
.....

```

2.13. Shortest pair of edge disjoint paths

Edge disjoint shortest pair algorithm is an algorithm in computer network routing. The algorithm is used for generating the shortest pair of edge disjoint paths between a given pair of vertices as follows:

- Run the shortest pair algorithm for the given pair of vertices
- Replace each edge of the shortest path (equivalent to two oppositely directed arcs) by a single arc directed towards the source vertex
- Make the length of each of the above arcs negative
- Run the shortest path algorithm (Note: the algorithm should accept negative costs)
- Erase the overlapping edges of the two paths found, and reverse the direction of the remaining arcs on the first shortest path such that each arc on it is directed towards the sink vertex now. The desired pair of paths results.

2.14. Shortest pair of completely disjoint paths

In theoretical computer science and network routing, Suurballe's algorithm is an algorithm for finding two disjoint paths in a nonnegatively-weighted directed graph, so that both paths connect the same pair of vertices and have minimum total length. The algorithm was conceived by J. W. Suurballe and published in 1974. The main idea of Suurballe's algorithm is to use Dijkstra's algorithm to find one path, to modify the weights of the graph edges, and then to run Dijkstra's algorithm a second time. The modification to the weights is similar to the weight modification in Johnson's algorithm, and preserves the non-negativity of the weights while allowing the second instance of Dijkstra's algorithm to find the correct second path.

Suurballe's algorithm performs the following steps:

Find the shortest path tree T rooted at node s by running Dijkstra's algorithm. This tree contains for every vertex u , a shortest path from s to u . Let P_1 be the shortest cost path from s to t . The edges in T are called tree edges and the remaining edges are called non tree edges.

Modify the cost of each edge in the graph by replacing the cost $w(u,v)$ of every edge (u,v) by $w'(u,v) = w(u,v) - d(s,v) + d(s,u)$. According to the resulting modified cost function, all tree edges have a cost of 0, and non tree edges have a non negative cost.

Create a residual graph G_t formed from G by removing the edges of G that are directed into s and by reversing the direction of the zero length edges along path P_1 .

Find the shortest path P_2 in the residual graph G_t by running Dijkstra's algorithm.

Discard the reversed edges of P_2 from both paths. The remaining edges of P_1 and P_2 form a subgraph with two outgoing edges at s , two incoming edges at t , and one incoming and one outgoing edge at each remaining vertex. Therefore, this subgraph consists of two edge-disjoint paths from s to t and possibly some additional (zero-length) cycles. Return the two disjoint paths from the subgraph.

2.15. Eulerian path

In graph theory, an Eulerian path is a path in a graph which visits each edge exactly once. Similarly, an Eulerian circuit is an Eulerian path which starts and ends on the same vertex. They were first discussed by Leonhard Euler while solving the famous Seven Bridges of Königsberg problem in 1736. Mathematically the problem can be stated like this: Given the graph on the right, is it possible to construct a path (or a cycle, i.e. a path starting and ending on the same vertex) which visits each edge exactly once? Euler proved that a necessary condition for the existence of Eulerian circuits is that all vertices in the graph have an even degree, and stated without proof that connected graphs with all vertices of even degree have an Eulerian circuit. The first complete proof of this latter claim was published in 1873 by Carl Hierholzer.[1] The term Eulerian graph has two common meanings in graph theory. One meaning is a graph with an Eulerian circuit, and the other is a graph with every vertex of even degree. These definitions coincide for connected graphs.[2] For the existence of Eulerian paths it is necessary that no more than two vertices have an odd degree; this means the Königsberg graph is not Eulerian. If there are no vertices of odd degree, all Eulerian paths are circuits. If there are exactly two vertices of odd degree, all Eulerian paths start at one of them and end at the other. Sometimes a graph that has an Eulerian path, but not an Eulerian circuit (in other words, it is an open path, and does not start and end at the same vertex) is called semi-Eulerian.

Properties

A connected undirected graph is Eulerian if and only if every graph vertex has an even degree. An undirected graph is Eulerian if it is connected and can be decomposed into edge-disjoint cycles. If an undirected graph G is Eulerian then its line graph $L(G)$ is Eulerian too. A directed graph is Eulerian if it is

strongly connected and every vertex has equal in degree and out degree. A directed graph is Eulerian if it is strongly connected and can be decomposed into edge-disjoint directed cycles. An Eulerian path exists in a directed graph if and only if the graph's underlying undirected graph is connected, at most one vertex has out degree-in degree=1, at most one vertex has in degree-out degree=1 and every other vertex has equal in degree and out degree. An undirected graph is traversable if it is connected and at most two vertices in the graph are of odd degree.

Constructing Eulerian paths and circuits

Consider a graph known to have all edges in the same component and at most two vertices of odd degree. We can construct an Eulerian path out of this graph by using Fleury's algorithm, which dates to 1883. We start with a vertex of odd degree—if the graph has none, then start with any vertex. At each step we move across an edge whose deletion would not disconnect the graph, unless we have no choice, then we delete that edge. At the end of the algorithm there are no edges left, and the sequence of edges we moved across forms an Eulerian cycle if the graph has no vertices of odd degree; or an Eulerian path if there are exactly two vertices of odd degree.

3. Programación Dinámica

3.1. Edit Distance

El edit distance entre dos cadenas está definido como el número mínimo de operaciones para convertir una cadena en otra con tres operaciones, inserción, eliminación y reemplazo.

Nótese que $d[i-1][j] + 1$ representa un costo de 1 para la inserción, $d[i][j-1] + 1$ costo 1 para eliminación, y $d[i-1][j-1] + (s1[i-1] \neq s2[j-1] ? 1 : 0)$ representa costo 1 para reemplazo (en caso de que no sean iguales). Con estas consideraciones es fácil adaptar este problema a otros similares.

```
unsigned int edit_distance(string s1, string s2) {
    const size_t len1 = s1.size(), len2 = s2.size();
    vector<vector<unsigned int>>> \
    d(len1 + 1, vector<unsigned int> (len2 + 1));

    d[0][0] = 0;
    for (unsigned int i = 1; i <= len1; ++i) d[i][0] = i;
    for (unsigned int i = 1; i <= len2; ++i) d[0][i] = i;

    for (unsigned int i = 1; i <= len1; ++i)
        for (unsigned int j = 1; j <= len2; ++j)
```

```

        d[i][j] = std::min(std::min(d[i - 1][j] + 1, d[i][j - 1] + 1),
        d[i - 1][j - 1] + (s1[i - 1] == s2[j - 1] ? 0 : 1));
    return d[len1][len2];
}

```

3.2. Integer Knapsack Problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

- * Generates an instance of the 0/1 knapsack problem with N items
- * and maximum weight W and solves it in time and space
- * proportional to $N * W$ using dynamic programming.

```

public class Knapsack {

    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]); // number of items
        int W = Integer.parseInt(args[1]); // maximum weight of knapsack

        int[] profit = new int[N + 1];
        int[] weight = new int[N + 1];

        // generate random instance, items 1..N
        for (int n = 1; n <= N; n++) {
            profit[n] = (int)(Math.random() * 1000);
            weight[n] = (int)(Math.random() * W);
        }

        // opt[n][w] = max profit of
        // packing items 1..n with weight limit w

        // sol[n][w] = does opt solution to pack
        // items 1..n with weight

        //limit w include item n?

        int[][] opt = new int[N + 1][W + 1];
    }
}

```

```

boolean[][] sol = new boolean[N + 1][W + 1];

for (int n = 1; n <= N; n++) {
    for (int w = 1; w <= W; w++) {

        // don't take item n
        int option1 = opt[n - 1][w];

        // take item n
        int option2 = Integer.MIN_VALUE;
        if (weight[n] <= w)
            option2 = profit[n] + opt[n - 1][w - weight[n]];

        // select better of two options
        opt[n][w] = Math.max(option1, option2);
        sol[n][w] = (option2 > option1);
    }
}

// determine which items to take
boolean[] take = new boolean[N + 1];
for (int n = N, w = W; n > 0; n--) {
    if (sol[n][w]) {
        take[n] = true;
        w = w - weight[n];
    } else {
        take[n] = false;
    }
}

// print results
System.out.println
    ("item" + "\t" + "profit" + "\t" + "weight" + "\t" + "take");
for (int n = 1; n <= N; n++) {
    System.out.println
        (n + "\t" + profit[n] + "\t" + weight[n] + "\t" + take[n]);
}
}
}

```


3.3. Counting Boolean Parenthesizations

Dada una expresión booleana (por ejemplo true or false and true) se debe determinar cuántas maneras existen de agrupar las variables (poner paréntesis) de tal forma que la expresión sea verdadera.

3.4. Longest Increasing Subsequence

In computer science, the longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible.

```
/**
 * Finds longest strictly increasing subsequence.
 * O(n log k).
 */
void find_lis(vector<int> &a, vector<int> &b){
    vector<int> p(a.size());
    int u, v;
    if (a.empty()) return;
    b.push_back(0);
    for (size_t i = 1; i < a.size(); i++){
        // If next element a[i] is greater than
        // last element of current
        // longest subsequence a[b.back()],
        // just push it at back of "b" and continue
        if (a[b.back()] < a[i]){
            p[i] = b.back();
            b.push_back(i);
            continue;
        }
        // Binary search to find the smallest
        // element referenced
        // by b which is just bigger than a[i]
        // Note : Binary search is performed on b
        // (and not a).
        // Size of b is always <=k and hence
        // contributes O(log k) to complexity.
        for (u = 0, v = b.size()-1; u < v;){
            int c = (u + v) / 2;
            if (a[b[c]] < a[i]) u=c+1; else v=c;
        }
    }
}
```

```
        // Update b if new value is smaller
        // then previously referenced value
        if (a[i] < a[b[u]]){
            if (u > 0) p[i] = b[u-1];
            b[u] = i;
        }
    }
    for (u = b.size(), v = b.back(); u--; v = p[v]) b[u] = v;
}
```

```
/* Example of usage: */
int main(){
    int a[] = { 1, 9, 3, 8, 11, 4, 5, 6, 4, 19, 7, 1, 7 };
    vector<int> seq(a, a+sizeof(a)/sizeof(a[0]));
    // seq : Input Vector
    vector<int> lis;
    // lis : Vector containing indexes
    // of longest subsequence
    find_lis(seq, lis);
    //Printing actual output
    for (size_t i = 0; i < lis.size(); i++){
        printf("%d ", seq[lis[i]]);
        printf("\n");
    }

    return 0;
}
```

3.5. Josephus problem

In computer science and mathematics, the Josephus problem (or Josephus permutation) is a theoretical problem related to a certain counting-out game. There are people standing in a circle waiting to be executed. After the first man is executed, certain number of people are skipped and one man is executed. Then again, people are skipped and a man is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last man remains, who is given freedom. The task is to choose the place in the initial circle so that you are the last one remaining and so survive.

```
public class Josephus
```

```

{
    /* todas las posiciones estan entre 0 y n - 1 */
    public static int josephus(int n, int k)
    {
        if(n == 1) return 0;
        return(josephus(n - 1, k) + k) % n;
    }
    /* inicialM es la primera persona que pierde
    (todas las posiciones estan entre 0 y n - 1) */
    public static int josephus(int n, int k, int inicialM)
    {
        int normal = josephus(n, k);
        k %= n;
        normal -= (k - 1);
        if(normal < 0) normal += n;
        normal += inicialM;
        return normal % n;
    }
}

```

4. Matemáticas

4.1. Fórmulas útiles

4.2. Aritmética Modular

Colección de códigos útiles para aritmética modular

```

int gcd (int a, int b) {
    int tmp;
    while (b) {
        a %= b;
        tmp = a;
        a = b;
        b = tmp;
    }
    return a;
}

```

```

// a % b (valor positivo)
int mod (int a, int b) {
    return ((a % b) + b) % b;
}

// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid (int a, int b, int &x, int &y) {
    int xx = y = 0;
    int yy = x = 1;
    while (b) {
        int q = a / b;
        int t = b;
        b = a % b;
        a = t;
        t = xx;
        xx = x - q * xx;
        x = t;
        t = yy;
        yy = y - q * yy;
        y = t;
    }
    return a;
}

```

```

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

```

```

// finds all solutions to ax = b (mod n)
vi modular_linear_equation_solver (int a, int b, int n) {
    int x, y;
    vi solutions;
    int d = extended_euclid (a, n, x, y);
    if (!(b % d)) {
        x = mod (x * (b / d), n);
        for (int i = 0; i < d; i++)
            solutions.push_back (mod (x + i * (n / d), n));
    }
    return solutions;
}

```

```

}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse (int a, int n) {
    int x, y;
    int d = extended_euclid (a, n, x, y);
    if (d > 1)
        return -1;
    return mod (x, n);
}

// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
// Return (z,M). On failure, M = -1.
pii chinese_remainder_theorem (int x, int a, int y, int b) {
    int s, t;
    int d = extended_euclid (x, y, s, t);
    if (a % d != b % d)
        return make_pair (0, -1);
    return make_pair \
        (mod (s * b * x + t * a * y, x * y) / d, x * y / d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i. Note that the solution is
// unique modulo M = lcm_i (x[i]). Return (z,M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
pii chinese_remainder_theorem (const vi & x, const vi & a) {
    pii ret = make_pair (a[0], x[0]);
    for (int i = 1; i < x.size (); i++) {
        ret = chinese_remainder_theorem \
            (ret.first, ret.second, x[i], a[i]);
        if (ret.second == -1) break;
    }
    return ret;
}

// computes x and y such that ax + by = c; on failure, x = y == -1
void linear_diophantine (int a, int b, int c, int &x, int &y) {
    int d = gcd (a, b);

```

```

    if (c % d) {
        x = y = -1;
    }
    else {
        x = c / d * mod_inverse (a / d, b / d);
        y = (c - a * x) / b;
    }
}

```

4.3. Mayor exponente de un primo que divide a n!

```

int pow_div_fact(int n, int p) {
    int sd = 0;
    for (int t = n; t > 0; t /= p)
        sd += t % p;
    return (n-sd)/(p-1);
}

```

4.4. Potencia modular

```

// Retorna (a^b)%c
long long mod_pow(long long a, long long b, long long c){
    long long x = 0, y = a % c;
    while(b > 0){
        if(b%2 == 1)
            x = (x+y)%c;
        y = (y*2)%c;
        b>>=1;
    }
    return x%c;
}

```

4.5. Criba de Eratóstenes

```

/**
 * Para usar, primero llamar la funcion que llena la criba.
 * Num::primeSieve();

```

```
* luego en Num::primes queda el vector con todos los primos
*/
```

```
typedef long long lli;
typedef vector<int> IV;
```

```
#define Sc(t,v) static_cast<t>(v)
#define cFor(t,v,c) for(t::const_iterator v=c.begin(); \
v != c.end(); v++)
```

```
//Cuidado con estas macros, son la parte mas fundamental
//del algoritmo
```

```
#define ISCOMP(n) (_c[(n)>>5]&(1<<((n)&31)))
#define SETCOMP(n) _c[(n)>>5]|=(1<<((n)&31))
```

```
namespace Num{
    const int MAX = 1000000; // sqrt(10^12)
    const int LMT = 1000; // sqrt(MAX)
    int _c[(MAX>>5)+1];
    IV primes;
    void primeSieve() {
        SETCOMP(0); SETCOMP(1);
        for (int i = 3; i <= LMT; i += 2)
            if (!ISCOMP(i))
                for (int j = i*i; j <= MAX; j+=i+i) SETCOMP(j);
        primes.push_back(2);
        for (int i=3; i <= MAX; i+=2)
            if (!ISCOMP(i)) primes.push_back(i);
    }
}
```

```
//Ejemplo
```

```
int main(){
    Num::primeSieve();
    using Num::primes;
    // se usa como primes[i] para el iesimo primo
    return 0;
}
```

4.6. Test de primalidad

```
/**
```

```
* Miller-Rabin. La probabilidad de error disminuye al aumentar
* el numero de iteraciones. Dicha probabilidad es 1/4^iter
*/
```

```
long long modpow(long long a,long long b,long long c){
    long long x = 0,y=a%c;
    while(b > 0){
        if(b&1) x = (x+y)%c;
        y = (y*2)%c;
        b>>=1;
    }
    return x%c;
}
```

```
long long modmul(long long a,long long b,long long m){
    return (((a*(b>>20)%m)<<20)+a*(b&((1<<20)-1)))%m;
}
```

```
bool Miller(long long p,int iteration){
    if(p<2)
        return false;
    if(p!=2 && p%2==0)
        return false;
    long long s = p-1;

    while(s%2==0)s>>=1;

    for(int i=0;i<iteration;i++){
        long long a=rand()%(p-1)+1,temp=s;
        long long mod=modpow(a,temp,p);
        while(temp!=p-1 && mod!=1 && mod!=p-1){
            mod=modmul(mod,mod,p);
            temp <<=1;
        }
        if(mod!=p-1 && temp%2==0){
            return false;
        }
    }
    return true;
}
```

4.7. Combinatoria

4.7.1. Sumas

$$\sum_{k=0}^n k = \frac{n(n+1)}{2}$$

$$\sum_{k=a}^b k = \frac{(a+b)(b-a+1)}{2}$$

$$\sum_{k=0}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^n k^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{k=0}^n k^4 = \frac{(6n^5 + 15n^4 + 10n^3 - n)}{30}$$

$$\sum_{k=0}^n k^5 = \frac{(2n^6 + 6n^5 + 5n^4 - n^2)}{12}$$

$$\sum_{k=0}^n x^k = \frac{x^{n+1}-1}{x-1}$$

$$ojo! \sum_{k=0}^n kx^k = \frac{x-n+1x^{n+1}+nx^{n+2}}{x-1+x+x^2+\dots} = \frac{1}{1-x}$$

4.7.2. Cuadro Resumen

Formulas para combinaciones y Permutaciones

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

5. Geometría

5.1. Utilidades Geometría

Código base para implementación de otros algoritmos.

```
struct point {
    double x;
    double y;

    point () {}
    point (double x_, double y_):x (x_), y (y_) {}

    bool operator < (const point & other) const {
        if (x == other.x)
            return y < other.y;
        return x < other.x;
    }

    double dist (const point & other) {
        double a = x - other.x;
        double b = y - other.y;
        return sqrt (a * a + b * b);
    }

    /**
     * Compara el punto C con el segmento AB.
     * Retorna 0 si son colineales.
     * Mayor que cero si está a la izquierda.
     * Menor que cero si está a la derecha.
     */
    int cross (const point & a, const point & b) {
        return (b.x - a.x) * (y - a.y) - (x - a.x) * (b.y - a.y);
    }
};
```

```

}

void multEsc (int e) {
    this->x *= e;
    this->y *= e;
}

};
.....

```

5.2. Transformaciones

5.2.1. Rotación

Para rotar un punto (x, y) un ángulo θ (counterclockwise) con respecto al origen se tiene:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix}$$

Para rotar un punto (x, y, z) un ángulo θ (counterclockwise) con respecto a un eje de rotación se tiene: (*Rodrigues' rotation formula*)

$$\mathbf{v}_{\text{rot}} = \mathbf{v} \cos \theta + (\mathbf{k} \times \mathbf{v}) \sin \theta + \mathbf{k}(\mathbf{k} \cdot \mathbf{v})(1 - \cos \theta).$$

5.2.2. Desplazamiento dos dimensiones

La traslación es el efecto que conseguiremos al mover un objeto de un lugar a otro. Para lograrlo vamos a suponer un punto en algún lugar, ese punto u objeto tiene unas coordenadas dadas en X y Y, muy bien ahora supongamos que queremos "trasladar" ese objeto a otro lugar a otra coordenada", entonces lo que haremos será :

Paso 1: Obtener el lugar donde se encuentra.

Paso 2: Dar a conocer cuantos lugares se moverá, este dato será en números enteros, es decir 1 espacio, 2 espacios, 3 espacios ó hasta n número de espacios.

Luego hacemos lo siguiente:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \Delta x & \Delta y & 1 \end{bmatrix} = \begin{bmatrix} x + 0 + \Delta x & y + 0 + \Delta y & z + 0 + 0 \end{bmatrix}$$

5.3. Distancia mínima: Punto-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);
}
.....

```

5.4. Distancia mínima: Punto-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);
}
.....

```

5.5. Determinar si un polígono es convexo

```
/*
Returns positive if a-b-c make a left turn.
Returns negative if a-b-c make 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;
}
```

5.6. 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;
}
```

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

```
//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));
    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 );
}

```

5.8. 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){

    #ifndef 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);
    }
}

```


5.9. Intersección de dos segmentos

```
.....

/*
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
        min(x0, x1) <= x && x <= max(x0, x1) &&
        min(y0, y1) <= y && y <= max(y0, y1);
}
/*
Finds the intersection between two segments (Not infinite
lines!)
Segment 1 goes from point (x0, y0) to (x1, y1).
Segment 2 goes from point (x2, y2) to (x3, y3).
(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(double x0, double y0,
double x1, double y1,
double x2, double y2,
double x3, double y3){
    #ifndef 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){
```

```
        //they lie on same line, but they may or may not intersect.
        return (point_in_box(x0, y0, x2, y2, x3, y3) ||
            point_in_box(x1, y1, x2, y2, x3, y3) ||
            point_in_box(x2, y2, x0, y0, x1, y1) ||
            point_in_box(x3, y3, x0, y0, x1, y1));
    }else{
        //just parallel, no intersection
        return false;
    }
}else{
    t0 /= det;
    t1 /= det;
    /*
    0 <= t0 <= 1 iff the intersection point lies in segment 1.
    0 <= t1 <= 1 iff the intersection point lies in segment 2.
    */
    if (0.0 <= t0 && t0 <= 1.0 && 0.0 <= t1 && t1 <= 1.0){
        double x = x0 + t0*(x1-x0);
        double y = y0 + t0*(y1-y0);
        //intersection is point (x, y)
        return true;
    }
    //the intersection points doesn't lie on both segments.
    return false;
}
}
```

5.10. Determinar si dos segmentos se intersectan o no

```
.....

/*Returns the cross product of the segment that goes from
(x1, y1) to (x3, y3) with the segment that goes from
(x1, y1) to (x2, y2)
*/
int direction(int x1, int y1, int x2, int y2, int x3, int y3) {
    return (x3 - x1) * (y2 - y1) - (y3 - y1) * (x2 - x1);
}
/*
Finds the intersection between two segments (Not infinite
lines!)
Segment 1 goes from point (x0, y0) to (x1, y1).
```

```

Segment 2 goes from point (x2, y2) to (x3, y3).
(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;
    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;
}

```

5.11. Centro del círculo que pasa por tres puntos.

```

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);
}

```

5.12. Par de puntos más cercanos

An $O(n \log n)$ algorithm for the closest pair problem, a divide and conquer approach. The main function `closestpair` receives the set of points ordered by x and y coordinates, in P_x and P_y , respectively. WARNING: You must be careful with the algorithm used for splitting the points into two equal groups, it can produce endless recursive calls(Runtime error). The algorithm used here doesn't work when there are a lot of points with the same x coordinate, for solving this you must be sure that your algorithm always splits the group of points into two smaller groups of approximately equal size.

```

double closestpair(ArrayList<Point> Px, ArrayList<Point> Py)
{
    if (Px.size() <= 12){
        double closest = Double.MAX_VALUE;
        for(int i=0; i<Px.size(); i++){
            for(int j=i+1; j<Px.size(); j++){
                closest = Math.min(distance(Px.get(i), Px.get(j)), closest);
            }
        }
        return closest;
    }
}

```

```

double x=Px.get(Px.size()/2).x;
ArrayList<Point> Lx=new ArrayList<Point>();
ArrayList<Point> Ly=new ArrayList<Point>();
ArrayList<Point> Rx=new ArrayList<Point>();
ArrayList<Point> Ry=new ArrayList<Point>();
for(Point p: Px){
    if (p.x<x)
        Lx.add(p);
    else
        Rx.add(p);
}
for(Point p: Py){
    if (p.x<x)
        Ly.add(p);
    else
        Ry.add(p);
}
double d1=closestpair(Lx,Ly);
double d2=closestpair(Rx,Ry);
double delta=Math.min(d1,d2);
double split=closestsplitpair(Px,Py,delta,x);
return Math.min(delta, split);
}

static double closestsplitpair(ArrayList<Point> Px,
ArrayList<Point> Py,double delta,double x){
    ArrayList<Point> Sy=new ArrayList<Point>();
    for(Point p: Py){
        if (x-delta<p.x && p.x<x+delta)
            Sy.add(p);
    }
    double min=delta;
    for(int i=0;i<Sy.size();i++){
        for(int j=1;j<=7 && i+j<Sy.size();j++){
            min=Math.min(min, distance(Sy.get(i),Sy.get(i+j)));
        }
    }
    return min;
}
}

```

5.13. Par de puntos más alejados

La función `farthest_point_pair_distance` encuentra la distancia mas grande que hay entre cualquier par de vértices de un polígono convexo (los puntos que se le pasan a la función deben estar ordenados clockwise) en $O(n)$ usando *rotatingcalipers*, por lo tanto si se quiere solucionar el problema de la distancia más grande entre cualquier par de puntos de una nube de puntos arbitraria se debe primero calcular el *convexhull* de la nube de puntos y ejecutar este algoritmo sobre ese polígono para un complejidad total de $O(n \log n)$ si se usa un algoritmo eficiente para el *convexhull*.

```

double farthest_point_pair_distance(ArrayList<Point> v){
    int index_min=0,index_max=0;
    double coor_min=Double.MAX_VALUE;
    double coor_max=Double.MIN_VALUE;
    for(int i=0;i<v.size();i++){
        Point a=v.get(i);
        if (a.y<coor_min){
            index_min=i;
            coor_min=a.y;
        }
        if (a.y>coor_max){
            index_max=i;
            coor_max=a.y;
        }
    }
    double d=v.get(index_min).sub(v.get(index_max)).norm();
    double rotated_angle=0;
    Point caliper_a=new Point(1.0,0.0);
    Point caliper_b=new Point(-1.0,0.0);
    while(rotated_angle<Math.PI){

        Point edge_a=
            v.get((index_max+1)%v.size()).sub(v.get(index_max));

        Point edge_b=
            v.get((index_min+1)%v.size()).sub(v.get(index_min));

        double angle_a=caliper_a.angle(edge_a);
        double angle_b=caliper_b.angle(edge_b);
    }
}

```

```

    caliper_a=caliper_a.rotate(-Math.min(angle_a, angle_b));
    caliper_b=caliper_b.rotate(-Math.min(angle_a, angle_b));
    if (angle_a>angle_b)
        index_miny=(index_miny+1)%v.size();
    else
        index_maxy=(index_maxy+1)%v.size();
    d=Math.max(v.get(index_miny).sub(v.get(index_maxy)).norm(),d);
    rotated_angle=rotated_angle + Math.min(angle_a, angle_b);
}
return d;
}

```

5.14. Área de un polígono

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

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

P es un polígono ordenado anticlockwise.

Si es clockwise, retorna el area negativa.

Si no esta ordenado retorna basura.

$P[0] \neq P[n-1]$

```

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;
}

```

5.15. Convexhull

In mathematics, the convex hull or convex envelope for a set of points X in a real vector space V is the minimal convex set containing X.

In computational geometry, a basic problem is finding the convex hull for a given finite nonempty set of points in the plane. It is common to use the term "convex hull" for the boundary of that set, which is a convex polygon, except in the degenerate case that points are collinear. The convex hull is then typically represented by a sequence of the vertices of the line segments forming the boundary of the polygon, ordered along that boundary.

5.15.1. Graham Scan

```

public class Convex_Hull
{
    static double cross(Point2D p1, Point2D p2, Point2D p3)
    {
        return (p2.getX() - p1.getX()) * (p3.getY() - p1.getY())
            - (p2.getY() - p1.getY()) * (p3.getX() - p1.getX());
    }

    static class Comp implements Comparator < Point2D >
    {
        public int compare(Point2D p1, Point2D p2)
        {
            if (p1.getY() < p2.getY()) return -1;
            if (p1.getY() > p2.getY()) return 1;
            if (p1.getX() < p2.getX()) return -1;
            return 1;
        }
    }

    static final double EPSILON = 1e-12;

    static void convexHull(List < Point2D > points,
        List < Point2D > result)
    {
        int n = points.size();
        Point2D[] p2 = new Point2D[points.size() + 1];
        Collections.sort(points, new Comp());
        int top = 0;
        p2[top++] = points.get(0);
        p2[top++] = points.get(1);
        for (int i = 2; i < n; i++)
        {
            while (top >= 2 && cross(p2[top - 2], p2[top - 1],

```

```

        points.get(i)) <= -EPSILON)
        --top;
        p2[top++] = points.get(i);
    }
    int r = top;
    for (int i = n - 2; i >= 0; i--)
    {
        while (top > r && cross(p2[top - 2], p2[top - 1],
            points.get(i)) <= -EPSILON)
            --top;
        if (i != 0)
            p2[top++] = points.get(i);
    }
    for (int i = 0; i < top; i++)
        result.add(p2[i]);
}

```

Otra implementacion

```

struct Point
{
    double x, y;
};

int n;

double dis(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x) * (p1.x - p2.x)
        + (p1.y - p2.y) * (p1.y - p2.y));
}

double cross(Point p1, Point p2, Point p3)
{
    return (p2.x - p1.x) * (p3.y - p1.y)
        - (p2.y - p1.y) * (p3.x - p1.x);
}

bool comp(const Point & p1, const Point & p2)

```

```

{
    if (p1.y < p2.y) return true;
    if (p1.y > p2.y) return false;
    if (p1.x < p2.x) return true;
}

void getconvexhull(Point p1[], Point p2[], int & top)
{
    sort(p1, p1 + n, comp);
    int i;
    top = 0;
    p2[top++] = p1[0];
    p2[top++] = p1[1];
    for (i = 2; i < n; i++)
    {
        while (top >= 2 && cross(p2[top - 2],
            p2[top - 1], p1[i]) <= 0)
            --top;
        p2[top++] = p1[i];
    }
    int r = top;
    for (i = n - 2; i >= 0; i--)
    {
        while (top > r && cross(p2[top - 2],
            p2[top - 1], p1[i]) <= 0)
            --top;
        if (i != 0)
            p2[top++] = p1[i];
    }
}

// Esta implementacion tiene en cuenta casos en los que todos
// los puntos son colineales y no hay en realidad un convex
// hull, en ese caso los retorna en orden.

public class Convex
{
    static int ccw(Point2D a, Point2D b, Point2D c) {
        double area2 = (b.x-a.x)*(c.y-a.y) -
            (b.y-a.y)*(c.x-a.x);
    }
}

```

```

        if (area2 < 0) return -1;
        else if (area2 > 0) return +1;
        else return 0;
    }

    static class Point2D implements Comparable<Point2D>
    {
        public final Comparator<Point2D> POLAR_ORDER =
new PolarOrder();

        private final double x;    // x coordinate
        private final double y;    // y coordinate

        public Point2D(double x, double y) {
            this.x = x;
            this.y = y;
        }

        public int compareTo(Point2D that) {
            if (this.y < that.y) return -1;
            if (this.y > that.y) return +1;
            if (this.x < that.x) return -1;
            if (this.x > that.x) return +1;
            return 0;
        }

        private class PolarOrder implements Comparator<Point2D> {
            public int compare(Point2D q1, Point2D q2) {
                double dx1 = q1.x - x;
                double dy1 = q1.y - y;
                double dx2 = q2.x - x;
                double dy2 = q2.y - y;

                if (dy1 >= 0 && dy2 < 0) return -1;
                else if (dy2 >= 0 && dy1 < 0) return +1;
                else if (dy1 == 0 && dy2 == 0) {
                    if (dx1 >= 0 && dx2 < 0) return -1;
                    else if (dx2 >= 0 && dx1 < 0) return +1;
                    else return 0;
                }
                else return -ccw(Point2D.this, q1, q2);
            }
        }
    }

```

```

    }
}

    public boolean equals(Object other) {
        if (other == this) return true;
        if (other == null) return false;
        if (other.getClass() != this.getClass())
return false;
        Point2D that = (Point2D) other;
        return this.x == that.x && this.y == that.y;
    }
}

    static class GrahamScan
    {
        private Stack<Point2D> hull = new Stack<Point2D>();

        public GrahamScan(ArrayList<Point2D> pts)
        {
            int N = pts.size();
            Point2D[] points = new Point2D[N];
            for (int i = 0; i < N; i++)
                points[i] = pts.get(i);
            Arrays.sort(points);
            Arrays.sort(points, 1, N, points[0].POLAR_ORDER);

            hull.push(points[0]);

            int k1;
            for (k1 = 1; k1 < N; k1++)
                if (!points[0].equals(points[k1]))
break;

            if (k1 == N) return;
            int k2;
            for (k2 = k1 + 1; k2 < N; k2++)
                if (ccw(points[0], points[k1], points[k2]) != 0)
break;

            hull.push(points[k2-1]);
            for (int i = k2; i < N; i++) {
                Point2D top = hull.pop();
                while (ccw(hull.peek(),

```

```

        top, points[i]) <= 0) {
            top = hull.pop();
        }
        hull.push(top);
        hull.push(points[i]);
    }
    assert isConvex();
}

public Iterable<Point2D> hull() {
    Stack<Point2D> s = new Stack<Point2D>();
    for (Point2D p : hull) s.push(p);
    return s;
}

private boolean isConvex()
{
    int N = hull.size();
    if (N <= 2) return true;

    Point2D[] points = new Point2D[N];
    int n = 0;
    for (Point2D p : hull()) {
        points[n++] = p;
    }

    for (int i = 0; i < N; i++) {
        if (ccw(points[i],
points[(i+1) % N],
points[(i+2) % N]) <= 0) {
            return false;
        }
    }
    return true;
}
}

```

5.16. Great circle distance

The great-circle distance or orthodromic distance is the shortest distance between any two points on the surface of a sphere measured along a path on the surface of the sphere (as opposed to going through the sphere's interior).

```

static double greatCircleDistance(double latitudeS,
double longitudeS, double latitudeF, double longitudeF,
double r)
{
    latitudeS = Math.toRadians(latitudeS);
    latitudeF = Math.toRadians(latitudeF);
    longitudeS = Math.toRadians(longitudeS);
    longitudeF = Math.toRadians(longitudeF);
    double deltaLongitude = longitudeF - longitudeS;
    double a = Math.cos(latitudeF) * Math.sin(deltaLongitude);
    double b = Math.cos(latitudeS) * Math.sin(latitudeF);

    b -= Math.sin(latitudeS) * Math.cos(latitudeF)
        * Math.cos(deltaLongitude);

    double c = Math.sin(latitudeS) * Math.sin(latitudeF);

    c += Math.cos(latitudeS) * Math.cos(latitudeF)
        * Math.cos(deltaLongitude);

    /*      En linea recta -> dist
    double ax = Math.cos(latitudeS) * Math.cos(longitudeS);
    double ay = Math.cos(latitudeS) * Math.sin(longitudeS);
    double az = Math.sin(latitudeS);
    double bx = Math.cos(latitudeF) * Math.cos(longitudeF);
    double by = Math.cos(latitudeF) * Math.sin(longitudeF);
    double bz = Math.sin(latitudeF);
    double dist = r*Math.sqrt(sqr(ax-bx)+
sqr(ay-by)+sqr(az-bz));*/

    return Math.atan(Math.sqrt(a * a + b * b) / c) * r;
}

```

5.17. Picks theorem

Often we have to deal with polygons whose vertices have integer coordinates. Such polygons are called lattice polygons. Now, suppose we do not know the exact position of the vertices and instead we are given two values:

B = number of lattice points on the boundary of the polygon

I = number of lattice points in the interior of the polygon

Amazingly, the area of this polygon is then given by:

$\text{Area} = B/2 + I - 1$

The above formula is called Pick's Theorem due to Georg Alexander Pick (1859 - 1943). In order to show that Pick's theorem holds for all lattice polygons we have to prove it in 4 separate parts. In the first part we show that the theorem holds for any lattice rectangle (with sides parallel to axis). Since a right-angled triangle is simply half of a rectangle it is not too difficult to show that the theorem also holds for any right-angled triangle (with sides parallel to axis). The next step is to consider a general triangle, which can be represented as a rectangle with some right-angled triangles cut out from its corners. Finally, we can show that if the theorem holds for any two lattice polygons sharing a common side then it will also hold for the lattice polygon, formed by removing the common side. Combining the previous result with the fact that every simple polygon is a union of triangles gives us the final version of Pick's Theorem. Pick's theorem is useful when we need to find the number of lattice points inside a large polygon.

Another formula worth remembering is Euler's Formula for polygonal nets. A polygonal net is a simple polygon divided into smaller polygons. The smaller polygons are called faces, the sides of the faces are called edges and the vertices of the faces are called vertices. Euler's Formula then states: $V - E + F = 2$, where

V = number of vertices E = number of edges F = number of faces For example, consider a square with both diagonals drawn. We have $V = 5$, $E = 8$ and $F = 5$ (the outside of the square is also a face) and so $V - E + F = 2$.

6. Strings

6.1. Knuth-Morris-Pratt KMP

The Knuth-Morris-Pratt string searching algorithm (or KMP algorithm) searches for occurrences of a "word" W within a main "text string" S by employing the observation that when a mismatch occurs, the word itself embodies sufficient information to determine where the next match could begin, thus bypassing re-examination of previously matched characters.

/*

```
* String Matching KMP
* preprocessing time: O(m) - pattern
* matching time: O(n) - text
*
* Key idea:
* prefix function : pi
* pi[q] = max { k : k < q and P_k is suffix of P_q }
* P_i are the i first characters of the pattern
*/
static int[] compute_prefix_function(char[] p) {
    int[] pi = new int[p.length];
    pi[0] = -1;
    int k = -1;
    for (int i = 1; i < p.length; i++) {
        while (k >= 0 && p[k + 1] != p[i]) k = pi[k];
        if (p[k + 1] == p[i]) k++;
        pi[i] = k;
    }
    return pi;
}

static void KMP_Matcher(String pattern, String text) {
    char[] p = pattern.toCharArray();
    char[] t = text.toCharArray();
    int[] pi = compute_prefix_function(p);
    int q = -1;
    for (int i = 0; i < text.length(); i++) {
        while (q >= 0 && p[q + 1] != t[i]) q = pi[q];
        if (p[q + 1] == t[i]) q++;
        if (q == p.length - 1) {
            // Pattern matched in position i-p.length+1
            q = pi[q];
        }
    }
    return;
}
```

.....

6.2. Aho-Corasick

The Aho-Corasick string matching algorithm is a string searching algorithm invented by Alfred V. Aho and Margaret J. Corasick. It is a kind of dictionary-matching algorithm that locates elements of a finite set of strings (the "dictionary") within an input text. It matches all patterns simultaneously. The complexity of the algorithm is linear in the length of the patterns plus the length of the searched text plus the number of output matches. Note that because all matches are found, there can be a quadratic number of matches if every substring matches (e.g. dictionary = a, aa, aaa, aaaa and input string is aaaa). Informally, the algorithm constructs a finite state machine that resembles a trie with additional links between the various internal nodes. These extra internal links allow fast transitions between failed pattern matches (e.g. a search for cat in a trie that does not contain cat, but contains cart, and thus would fail at the node prefixed by ca), to other branches of the trie that share a common prefix (e.g., in the previous case, a branch for attribute might be the best lateral transition). This allows the automaton to transition between pattern matches without the need for backtracking. When the pattern dictionary is known in advance (e.g. a computer virus database), the construction of the automaton can be performed once off-line and the compiled automaton stored for later use. In this case, its run time is linear in the length of the input plus the number of matched entries.

```
////////////////////////////////////
//      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
            out[g[state][c]] |= out[failure];
            q.push(g[state][c]);
        }
    }
}

return states;
}

// Finds the next state the machine will transition to.
//

```

```

// currentState - The current state of the machine. Must be
//                  between 0 and the number of states - 1,
//                  inclusive.
// nextInput - The next character that enters into the machine.
//              Should be between lowestChar and highestChar,
//              inclusive.
// lowestChar - Should be the same lowestChar that was passed
//              to "buildMatchingMachine".

// Returns the next state the machine will transition to.
// This is an integer between 0 and the number of states - 1,
// inclusive.
int findNextState(int currentState, char nextInput,
    char lowestChar = 'a') {
    int answer = currentState;
    int c = nextInput - lowestChar;
    while (g[answer][c] == -1) answer = f[answer];
    return g[answer][c];
}

// How to use this algorithm:
//
// 1. Modify the MAXS and MAXC constants as appropriate.
// 2. Call buildMatchingMachine with the set of keywords to
//    search for.
// 3. Start at state 0. Call findNextState to incrementally
//    transition between states.
// 4. Check the out function to see if a keyword has been
//    matched.
//
// Example:
//
// Assume keywords is a vector that contains
// {"he", "she", "hers", "his"} and text is a string that
// contains "ahishers".
//
// Consider this program:
//
// buildMatchingMachine(keywords, 'a', 'z');
// int currentState = 0;

```



```
};
```

6.4. Minimum string rotation

In computer science, the lexicographically minimal string rotation or lexicographically least circular substring is the problem of finding the rotation of a string possessing the lowest lexicographical order of all such rotations. For example, the lexicographically minimal rotation of "bbaaccaadd" would be ".accaaddbb". It is possible for a string to have multiple lexicographically minimal rotations, but for most applications this does not matter as the rotations must be equivalent. Finding the lexicographically minimal rotation is useful as a way of normalizing strings. If the strings represent potentially isomorphic structures such as graphs, normalizing in this way allows for simple equality checking. A common implementation trick when dealing with circular strings is to concatenate the string to itself instead of having to perform modular arithmetic on the string indices.

```
def LCS(S):
    n = len(S)
    S += S
    # Concatenate string to
    # self to avoid modular arithmetic
    f = [-1 for c in S]
    # Failure function
    k = 0
    # Least rotation of string found so far
    for j in range(1, 2*n):
        i = f[j-k-1]
        while i != -1 and S[j] != S[k+i+1]:
            if S[j] < S[k+i+1]:
                k = j-i-1
            i = f[i]
        if i == -1 and S[j] != S[k+i+1]:
            if S[j] < S[k+i+1]:
                k = j
            f[j-k] = -1
        else:
            f[j-k] = i+1
    return k
```

7. Teoría de Juegos

8. Estructuras de Datos

8.1. RMQ

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices. We will see later that the LCA problem can be reduced to a restricted version of an RMQ problem, in which consecutive array elements differ by exactly 1. However, RMQs are not only used with LCA. They have an important role in string preprocessing, where they are used with suffix arrays (a new data structure that supports string searches almost as fast as suffix trees, but uses less memory and less coding effort).

```
// Ojo: Utiliza N log N memoria
int[] buildRMQ(int[] vector, int n) {
    int logn = 0;
    for (int k = 1; k < n; k *= 2) ++logn;
    int[] b = new int[n * logn];
    System.arraycopy(vector, 0, b, 0, n);
    int delta = 0;
    for (int k = 1; k < n; k *= 2) {
        System.arraycopy(b, delta, b, n + delta, n);
        delta += n;

        for (int i = 0; i < n - k; i++)
            b[i + delta] = Math.min(b[i + delta],
                                    b[i + k + delta]);
    }
    return b;
}
```

```
// Responde queries en tiempo constante
// para mayor velocidad se pueden precalcular
// los k, e, z para todos los y - x posibles.
int minimum(int[] rmq, int x, int y) {
    int z = y - x, k = 0, e = 1, s;
    s = ((z & 0xffff0000) != 0) ? 1 : 0) << 4;
    z >>= s;
    e <<= s;
    k |= s;
    s = ((z & 0x0000ff00) != 0) ? 1 : 0) << 3;
```

```

z >>= s;
e <<= s;
k |= s;
s = (((z & 0x000000f0) != 0) ? 1 : 0) << 2;
z >>= s;
e <<= s;
k |= s;
s = (((z & 0x0000000c) != 0) ? 1 : 0) << 1;
z >>= s;
e <<= s;
k |= s;
s = (((z & 0x00000002) != 0) ? 1 : 0) << 0;
z >>= s;
e <<= s;
k |= s;
return Math.min(rmq[x + n * k], rmq[y + n * k - e + 1]);
}

```

.....

8.2. Union-find (disjoint-set)

Union Find is an algorithm which uses a disjoint-set data structure to solve the following problem: Say we have some number of items. We are allowed to merge any two items to consider them equal (where equality here obeys all of the properties of an Equivalence Relation). At any point, we are allowed to ask whether two items are considered equal or not. Definition

Basically a Union Find data structure implements two functions:

union(A, B) - merge A's set with B's set

find(A) - finds what set A belongs to

This is a common way to find connectivity between nodes, or to find connected components.

```

static class DisjointSet {
    int[] p, rank;
    public DisjointSet(int size) {
        rank = new int[size];
        p = new int[size];
        for (int i = 0; i < size; i++) {
            make_set(i);
        }
    }
}

```

```

public void make_set(int x) {
    p[x] = x;
    rank[x] = 0;
}

public void merge(int x, int y) {
    link(find_set(x), find_set(y));
}

public 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] += 1;
    }
}

public int find_set(int x) {
    if (x != p[x]) p[x] = find_set(p[x]);
    return p[x];
}
}

```

.....

8.3. Prefix Tree - Trie

The tries can insert and find strings in $O(L)$ time (where L represent the length of a single word). This is much faster than set , but is it a bit faster than a hash table.

```

struct Trie {
    struct Node {
        int ch[26];
        int n;
        Node() {
            n = 0;
            memset(ch,0,sizeof(ch));
        }
    };
    int sz;
    vector < Node > nodes;
    void init() {
        nodes.clear();
    }
}

```

```

    nodes.resize(1);
    sz = 1;
}
void insert(const char * s) {
    int idx, cur = 0;

    for (; * s; ++s) {
        idx = * s - 'A';
        if (!nodes[cur].ch[idx]) {
            nodes[cur].ch[idx] = sz++;
            nodes.resize(sz);
        }

        cur = nodes[cur].ch[idx];
    }
}
};
.....

```

8.4. Fenwick Tree

Fenwick tree (aka Binary indexed tree) is a data structure that maintains a sequence of elements, and is able to compute cumulative sum of any range of consecutive elements in $O(\log n)$ time. Changing value of any single element needs $O(\log n)$ time as well. The structure is space-efficient in the sense that it needs the same amount of storage as just a simple array of n elements.

```

//one dimension
// Creates a zero-initialized Fenwick tree for n elements.
vector < int > create(int n){
    return vector < int > (n, 0);
}
// Returns sum of elements with indexes a..b, inclusive
int query(const vector < int > & tree, int a, int b){
    if (a == 0) {
        int sum = 0;
        for (; b >= 0; b = (b & (b + 1)) - 1)
            sum += tree[b];
        return sum;
    }
    else {

```

```

        return query(tree, 0, b) - query(tree, 0, a - 1);
    }
}
// Increases value of k-th element by inc.
void increase(vector < int > & tree, int k, int inc){
    for (; k < (int) tree.size(); k |= k + 1)
        tree[k] += inc;
}
//Two dimension
typedef long long lint;
typedef unsigned long long uint;
const int MAX = 1050;
int tree[MAX][MAX], lastx, lasty;
int M[MAX][MAX];
int
readx(int x, int y){
    int sum = 0;
    while (x > 0) {
        sum += tree[x][y];
        x -= (x & -x);
    }
    return sum;
}
int getsum(int x, int y){
    int sum = 0;
    while (y > 0) {
        sum += readx(x, y);
        y -= (y & -y);
    }
    return sum;
}
void updatey(int x, int y, int val){
    while (y = lasty)
    {
        tree[x][y] += val;
        y += (y & -y);
    }
}
void update(int x, int y, int val){
    while (x = lastx) {
        updatey(x, y, val);

```

```

    x += (x & -x);
}
}
int get(int x1, int y1, int x2, int y2){
    int a = getsum(x1 - 1, y1 - 1);
    int b = getsum(x2, y2);
    int c = getsum(x2, y1 - 1);
    int d = getsum(x1 - 1, y2);
    return a + b - c - d;
}

```

8.5. Interval Tree

9. Hashing

9.1. FNV Hash

```

unsigned fnv_hash (void *key, int len ){
    unsigned char *p = key;
    unsigned h = 2166136261;
    for (int i = 0; i < len; i++ )
        h = ( h * 16777619 ) ^ p[i];
    return h;
}

```

9.2. JSW Hash

Este es el más recomendado en términos de distribución.

```

unsigned jswhash ( void *key, int len ){
    unsigned char *p = key;
    unsigned h = 16777551;
    for (int i = 0; i < len; i++ )
        h = ( h << 1 | h >> 31 ) ^ tab[p[i]];
    return h;
}

```

10. Miseláneo

10.1. Bitwise operations

Operaciones útiles con bits.

Use the following formula to turn off the rightmost 1-bit in a word, producing 0 if none (e.g., 01011000 becomes 01010000):

$$x \& (x - 1)$$

Use the following formula to isolate the rightmost 0-bit, producing 0 if none (e.g., 10100111 becomes 00001000):

$$\text{not}(x) \& (x + 1)$$

Use the following formula to right-propagate the rightmost 1-bit, producing all 1's if (e.g., 01011000 becomes 01011111):

$$x | (x - 1)$$

Use the following formula to turn off the rightmost contiguous string of 1-bits (e.g., 01011000 becomes 01000000):

$$((x | (x - 1)) + 1) \& x$$

```
typedef unsigned int uint;
```

```
//retorna el siguiente entero con la misma
//cantidad de 1's en la representación binaria
```

```

uint next_popcount(uint n){
    uint c = (n & -n);
    uint r = n+c;
    return (((r ^ n) >> 2) / c) | r;
}

```

```

//retorna el primer entero con n 1's en binario
uint init_popcount(int n){
    return (1 << n) - 1;
}

```

10.2. Física

10.2.1. Movimiento parabólico

estas son algunas ecuaciones del movimiento parabólico que pueden ser útiles: la velocidad en x es constante y la aceleración gravitacional apunta hacia la tierra.

velocidad en y: $vs_y = v_{0y} + (-g) * t$

posición en x: $x = v_{0x} * t$

posición en y: $y = y_0 + v_{0y} * t + 1/$

10.3. Stable marriage

In mathematics, the stable marriage problem (SMP) is the problem of finding a stable matching between two sets of elements given a set of preferences for each element. A matching is a mapping from the elements of one set to the elements of the other set. A matching is stable whenever it is not the case that both: some given element A of the first matched set prefers some given element B of the second matched set over the element to which A is already matched, and B also prefers A over the element to which B is already matched. In other words, a matching is stable when there does not exist any alternative pairing (A, B) in which both A and B are individually better off than they would be with the element to which they are currently matched. The problem is commonly stated as: Given n men and n women, where each person has ranked all members of the opposite sex with a unique number between 1 and n in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are "stable".

In 1962, David Gale and Lloyd Shapley proved that, for any equal number of men and women, it is always possible to solve the SMP and make all marriages stable. They presented an algorithm to do so.[1][2] The Gale-Shapley algorithm involves a number of rounds (or iterations) where each unengaged man "proposes" to the most-preferred woman to whom he has not yet proposed. Each woman then considers all her suitors and tells the one she most prefers "Maybe" and all the rest of them "No". She is then provisionally "engaged" to the suitor she most prefers so far, and that suitor is likewise provisionally engaged to her. In the first round, first a) each unengaged man proposes to the woman he prefers most, and then b) each woman replies "maybe" to her suitor she most prefers and "no" to all other suitors. In each subsequent round, first a) each unengaged man proposes to the most-preferred woman to whom he has not yet proposed (regardless of whether the woman is already engaged), and then b) each woman

replies "maybe" to her suitor she most prefers (whether her existing provisional partner or someone else) and rejects the rest (again, perhaps including her current provisional partner). The provisional nature of engagements preserves the right of an already-engaged woman to "trade up" (and, in the process, to "jilt" her until-then partner).

This algorithm guarantees that:

Everyone gets married

Once a woman becomes engaged, she is always engaged to someone. So, at the end, there cannot be a man and a woman both unengaged, as he must have proposed to her at some point (since a man will eventually propose to everyone, if necessary) and, being unengaged, she would have to have said yes.

The marriages are stable

Let Alice be a woman and Bob be a man who are both engaged, but not to each other. Upon completion of the algorithm, it is not possible for both Alice and Bob to prefer each other over their current partners. If Bob prefers Alice to his current partner, he must have proposed to Alice before he proposed to his current partner. If Alice accepted his proposal, yet is not married to him at the end, she must have dumped him for someone she likes more, and therefore doesn't like Bob more than her current partner. If Alice rejected his proposal, she was already with someone she liked more than Bob.

```
function stableMatching
{
    Initialize all m in M and w in W to free

    while(exists a free man m who still has
    a woman w to propose to)
    {
        w = m's highest ranked such woman who
        he has not proposed to yet

        if(w is free)
            (m, w) become engaged

        else
        {
            some pair (m_prime, w) already exists

            if(w prefers m to m_prime)
            {
                (m, w) become engaged
                m_prime becomes free
            }
        }
    }
}
```



```

int tipo = 0;
for(int i = 0; i < 5; i++)
    repetidas.add(1);
for(int i = 0; i < 5; i++)
{
    for(int j = 0; j < 5; j++)
    {
        if(j != i)
        {
            if (mano[i].compareTo(mano[j]) == 0)
            {
                repetidas.set(i, repetidas.get(i) + 1);
            }
        }
    }
}
ArrayList <Integer> manoN = new ArrayList <Integer> ();
if(repetidas.contains(4))
{
    tipo = 7;
    for(int i = 0; i < 4; i++)
        manoN.add(mano[repetidas.indexOf(4)].valor);
    manoN.add(mano[repetidas.indexOf(1)].valor);
}
else if(repetidas.contains(3) && repetidas.contains(2))
{
    tipo = 6;
    for(int i = 0; i < 3; i++)
        manoN.add(mano[repetidas.indexOf(3)].valor);
    for(int i = 0; i < 2; i++)
        manoN.add(mano[repetidas.indexOf(2)].valor);
}
else if(repetidas.contains(3))
{
    tipo = 3;
    for(int i = 0; i < 3; i++)
        manoN.add(mano[repetidas.indexOf(3)].valor);
    manoN.add(mano[repetidas.indexOf(1)].valor);
    manoN.add(mano[repetidas.lastIndexOf(1)].valor);
}
else if(repetidas.contains(2))

```

```

{
    if(repetidas.indexOf(2) + 1 != repetidas.lastIndexOf(2))
    {
        tipo = 2;
        for(int i = 0; i < 2; i++)
            manoN.add(mano[repetidas.indexOf(2)].valor);
        for(int i = 0; i < 2; i++)
            manoN.add(mano[repetidas.lastIndexOf(2)].valor);
        manoN.add(mano[repetidas.indexOf(1)].valor);
    }
    else
    {
        tipo = 1;
        for(int i = 0; i < 2; i++)
            manoN.add(mano[repetidas.indexOf(2)].valor);
        for(int i = 0; i < 5; i++)
        {
            if(repetidas.get(i) == 1)
            {
                manoN.add(mano[i].valor);
            }
        }
    }
}
else
{
    for(Carta c : mano)
    {
        manoN.add(c.valor);
    }
    boolean color = true;
    int pinta = mano[0].pinta;
    for(Carta c : mano)
    {
        if(c.pinta != pinta)
            color = false;
    }
    boolean escalera = true;
    // si la escalera A-5 es permitida
    // entonces se debe cambiar para
    // aceptar 2,3,4,5,14 como escalera

```

```

// aca.
int valor = mano[0].valor;
for(Carta c : mano)
{
    if(c.valor != valor)
        escalera = false;
    else
        valor--;
}
if(escalera && color)
    tipo = 8;
else if(color)
    tipo = 5;
else if(escalera)
    tipo = 4;
}
long valorMano = tipo * 10000000000L;
valorMano += manoN.get(0) * 100000000;
valorMano += manoN.get(1) * 1000000;
valorMano += manoN.get(2) * 10000;
valorMano += manoN.get(3) * 100;
valorMano += manoN.get(4);
return valorMano;
}
}

.....

```

10.5. Inversions

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Formally speaking, two elements $a[i]$ and $a[j]$ form an inversion if $a[i] > a[j]$ and $i < j$

Example:

The sequence 2, 4, 1, 3, 5 has three inversions (2, 1), (4, 1), (4, 3).

```

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

```

```

class Inversions

```

```

{
    public static int countInversions(int nums[])
    {
        int mid = nums.length/2, k;
        int countLeft, countRight, countMerge;
        if (nums.length <= 1)
            return 0;
        int left[] = new int[mid];
        int right[] = new int[nums.length - mid];
        for (k = 0; k < mid; k++)
            left[k] = nums[k];
        for (k = 0; k < nums.length - mid; k++)
            right[k] = nums[mid+k];
        countLeft = countInversions (left);
        countRight = countInversions (right);
        int result[] = new int[nums.length];
        countMerge = mergeAndCount (left, right, result);
        for (k = 0; k < nums.length; k++)
            nums[k] = result[k];
        return (countLeft + countRight + countMerge);
    }

    public static int mergeAndCount
    (int left[], int right[], int result[])
    {
        int a = 0, b = 0, count = 0, i, k=0;
        while ( ( a < left.length) && (b < right.length) )
        {
            if ( left[a] <= right[b] )
                result [k] = left[a++];
            else
            {
                result [k] = right[b++];
                count += left.length - a;
            }
            k++;
        }
        if ( a == left.length )
            for ( i = b; i < right.length; i++)
                result [k++] = right[i];
        else

```

```

        for ( i = a; i < left.length; i++)
            result [k++] = left[i];
    return count;
}
}

```

10.6. Nim

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 rotating the word NIM by 180 degrees results in WIN (see Ambigram). Nim is usually played as a *misère* game,[citation needed] 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 *nimbers*) 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). While all normal play impartial games can be assigned a nim value, that is not the case under the *misère* convention. Only tame games can be played using the same strategy as *misère* nim.

A normal play game may start with heaps of 3, 4 and 5 objects:

In order to win always leave an even total number of 1's, 2's, and 4's.

Sizes of heaps Moves

A B C

```

3 4 5 Player 1 takes 2 from A
1 4 5 Player 2 takes 3 from C
1 4 2 Player 1 takes 1 from B

```

```

1 3 2 Player 2 takes 1 from B
1 2 2 Player 1 takes entire A heap, leaving two 2s.
0 2 2 Player 2 takes 1 from B
0 1 2 Player 1 takes 1 from C leaving two 1s. (In misère play I would take 2 from
C leaving (0, 1, 0).)
0 1 1 Player 2 takes 1 from B
0 0 1 Player 1 takes entire C heap and wins.

```

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 $x * y$ 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, $3 * 4 * 5 = 2$

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 canceling 1s and 4s

In normal play, the winning strategy is to finish every move with a Nim-sum of 0. This 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 $X = 3 * 4 * 5 = 2$. The Nim-sums of the heap sizes $A=3$, $B=4$, and $C=5$ with $X=2$ are (* representa xor) $A * X = 3 * 2 = 1$ [Since $(011) * (010) = 001$]

$B * X = 4 * 2 = 6$

$C * X = 5 * 2 = 7$

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 0102=210 I take 2 from A, leaving a sum of 000, so I will win.

1 4 5 0002=010 You take 2 from C

1 4 3 1102=610 I take 2 from B

1 2 3 0002=010 You take 1 from C

1 2 2 0012=110 I take 1 from A

0 2 2 0002=010 You take 1 from C

0 2 1 0112=310 The normal play strategy would be to take 1 from B, leaving an even number (2)

heaps of size 1. For misère play, I take the entire B heap, to leave an odd number (1) of heaps of size 1.

0 0 1 0012=110 You take 1 from C, and lose.

10.7. Sudoku

```
//se llama inicialmente con i = 0 y j = 0 y en cells
//0 en los desconocidos y el valor en los conocidos.
//si retorna true al final la matriz queda llena
//con la solucion.
static boolean solve(int i, int j, int[][] cells) {
    if (i == 9) {
```

```
        i = 0;
        if (++j == 9)
            return true;
    }
    if (cells[i][j] != 0) // skip filled cells
        return solve(i+1,j,cells);

    for (int val = 1; val <= 9; ++val) {
        if (legal(i,j,val,cells)) {
            cells[i][j] = val;
            if (solve(i+1,j,cells))
                return true;
        }
    }
    cells[i][j] = 0; // reset on backtrack
    return false;
}

static boolean legal(int i, int j, int val, int[][] cells) {
    for (int k = 0; k < 9; ++k) // row
        if (val == cells[k][j])
            return false;

    for (int k = 0; k < 9; ++k) // col
        if (val == cells[i][k])
            return false;

    int boxRowOffset = (i / 3)*3;
    int boxColOffset = (j / 3)*3;
    for (int k = 0; k < 3; ++k) // box
        for (int m = 0; m < 3; ++m)
            if (val == cells[boxRowOffset+k][boxColOffset+m])
                return false;

    return true; // no violations, so it's legal
}

.....

public class Sudoku
{
    static final int tS = 4;
```

```

static final int tS2 = 16;
static int visitado = 1 << (tS2 + 1);

static void marcar(int i, int j, int n)
{
    int mascara = 1 << n;
    sudoku[i][j] = mascara + visitado;
    mascara = ~mascara;
    int iMenor = (i / tS) * tS;
    int iMayor = iMenor + tS;
    int jMenor = (j / tS) * tS;
    int jMayor = jMenor + tS;
    for(int a = 0; a < tS2; a++)
    {
        if(a != j)
            sudoku[i][a] &= mascara;
        if(a != i)
            sudoku[a][j] &= mascara;
    }
    for(int a = iMenor; a < iMayor; a++)
        for(int b = jMenor; b < jMayor; b++)
            if(a != i || b != j)
                sudoku[a][b] &= mascara;
}

static long tiempoDentro = 0;
static int idMascara = 0;
static int[][][] zonas = new int[tS2 * 3][tS2][2];
static int[] actuales = new int[tS2];

public static void limpiarZona(int[][] zona, int[] actuales,
                                int[] grupos, int[] tamGrupos,
                                int[][] sudoku, int[] visitados)
{
    while(true)
    {
        int iActual = 0;
        forExterno:
        for(int i = 0; i <= tS2; i++)
        {
            if(tamGrupos[i] == -1)

```

```

                break;
            for(int j = 0; j < tamGrupos[i]; j++)
            {
                int indice = grupos[iActual + j];
                int valor = sudoku[zona[indice][0]][zona[indice][1]];
                if(valor >= visitado || tamGrupos[i] == 1)
                {
                    tamGrupos[i]--;
                    for(int k = iActual + j; k < tS2 - 1; k++)
                        grupos[k] = grupos[k + 1];
                    if(tamGrupos[i] == 0)
                    {
                        for(int k = i; k < tS2; k++)
                            tamGrupos[k] = tamGrupos[k + 1];
                        i--;
                        continue forExterno;
                    }
                    else
                    {
                        j--;
                        continue;
                    }
                }
            }
            iActual += tamGrupos[i];
        }
        break;
    }
    idMascara++;
    while(true)
    {
        int iActual = 0;
        forExterno:
        for(int i = 0; i <= tS2; i++)
        {
            int tamActual = tamGrupos[i];
            if(tamActual == -1)
                break;
            int mascaraActual = 1 << tamActual;
            mascaraActual--;
            for(int mascara = 1; mascara < mascaraActual; mascara++)

```

```

{
    int temp = mascara;
    int m = 0;
    for(int j = 0; j < tamActual; j++)
    {
        if((temp & 1) == 1)
        {
            m |= sudoku[zona[grupos[iActual + j]][0]]
                [zona[grupos[iActual + j]][1]];
        }
        temp >>= 1;
    }
    int tamCuenta = Integer.bitCount(m);
    if(tamCuenta == tamActual)
        continue;
    if(visitados[m] == idMascara)
        continue;
    temp = ~m;
    int cuenta = 0;
    for(int j = 0; j < tamActual; j++)
    {
        int valor = sudoku[zona[grupos[iActual + j]][0]]
            [zona[grupos[iActual + j]][1]];
        if((valor & temp) == 0)
            cuenta++;
    }
    if(cuenta == tamCuenta)
    {
        int posA = 0;
        for(int j = 0; j < tamActual; j++)
        {
            int valor = sudoku[zona[grupos[iActual + j]][0]]
                [zona[grupos[iActual + j]][1]];
            if((valor & temp) == 0)
                actuales[posA++] = grupos[iActual + j];
        }
        for(int j = 0; j < tamActual; j++)
        {
            int valor = sudoku[zona[grupos[iActual + j]][0]]
                [zona[grupos[iActual + j]][1]];
            if((valor & temp) != 0)

```

```

{
    actuales[posA++] = grupos[iActual + j];
    sudoku[zona[grupos[iActual + j]][0]]
        [zona[grupos[iActual + j]][1]] &= temp;
}
}
for(int k = 0; k < tamActual; k++)
    grupos[iActual + k] = actuales[k];
tamGrupos[i] = cuenta;
for(int k = tS2; k > i; k--)
    tamGrupos[k] = tamGrupos[k - 1];
tamGrupos[i + 1] = tamActual - cuenta;
i--;
continue forExterno;
}
visitados[m] = idMascara;
}
iActual += tamActual;
}
break;
}
}

static int[] visitados = new int[1 << (tS2 + 1)];

public static void limpiar(int[] [] sudoku,
    int[] [] grupos,
    int[] [] tamGrupos)
{
    for(int i = 0; i < tS2 * 3; i++)
        limpiarZona(zonas[i], actuales, grupos[i],
            tamGrupos[i], sudoku, visitados);
}

static int[] [] sudoku = new int[tS2][tS2];
static int[] [] grupos = new int[tS2 * 3][tS2];
static int[] [] tamGrupos = new int[tS2 * 3][tS2 + 1];
static int[] [] [] backtrack = new int[tS2 * tS2][tS2][tS2];

static int[] [] [] backtrackGrupos = new int[tS2 * tS2][tS2 * 3]
    [tS2];

```

```

static int[][][] backtrackTamGrupos = new int[tS2 * tS2]
[tS2 * 3][tS2 + 1];

static boolean backtrack(int altura, int[][] sudoku,
int[][] grupos, int[][] tamGrupos)
{
    int sumAnterior = 0;
    int sumMejor = Integer.MAX_VALUE;
    boolean termino = true;
    int iMejor = 0;
    int jMejor = 0;
    while(sumMejor > 1 && sumAnterior != sumMejor)
    {
        sumAnterior = sumMejor;
        iMejor = 0;
        jMejor = 0;
        termino = true;
        sumMejor = Integer.MAX_VALUE;
        for(int i = 0; i < tS2; i++)
            for(int j = 0; j < tS2; j++)
            {
                int pos = Integer.bitCount(sudoku[i][j]);
                if(sudoku[i][j] == 0)
                    return false;
                if(sudoku[i][j] <= todos)
                    termino = false;
                if(sudoku[i][j] <= todos && pos <= sumMejor)
                {
                    sumMejor = pos;
                    iMejor = i;
                    jMejor = j;
                }
            }
        if(sumMejor != 1)
            limpiar(sudoku, grupos, tamGrupos);
    }
    if(termino)
    {
        for(int i = 0; i < tS2; i++)
        {

```

```

            for(int j = 0; j < tS2; j++)
            {
                int temp = sudoku[i][j];
                for(int k = 0; k < tS2; k++)
                {
                    if((temp & 1) == 1)
                        System.out.print((char) ('A' + k));
                    temp >>= 1;
                }
            }
            System.out.println();
        }
        return true;
    }
    if(sumMejor == 1)
    {
        int pos = 0;
        int temp = sudoku[iMejor][jMejor];
        for(int i = 0; i < tS2; i++)
        {
            if((temp & 1) == 1)
                pos = i;
            temp >>= 1;
        }
        marcar(iMejor, jMejor, pos);
        return backtrack(altura + 1, sudoku, grupos, tamGrupos);
    }
    else
    {
        for(int j = 0; j < tS2; j++)
            for(int k = 0; k < tS2; k++)
                backtrack[altura][j][k] = sudoku[j][k];
        for(int j = 0; j < tS2 * 3; j++)
            for(int k = 0; k < tS2; k++)
                backtrackGrupos[altura][j][k] = grupos[j][k];
        for(int j = 0; j < tS2 * 3; j++)
            for(int k = 0; k <= tS2; k++)
                backtrackTamGrupos[altura][j][k] = tamGrupos[j][k];
        int temp = sudoku[iMejor][jMejor];
        for(int i = 0; i < tS2; i++)
        {

```



```

if((temp & 1) == 1)
{
    int pos = i;
    for(int j = 0; j < tS2; j++)
        for(int k = 0; k < tS2; k++)
            sudoku[j][k] = backtrack[altura][j][k];
    for(int j = 0; j < tS2 * 3; j++)
        for(int k = 0; k < tS2; k++)
            grupos[j][k] = backtrackGrupos[altura][j][k];
    for(int j = 0; j < tS2 * 3; j++)
        for(int k = 0; k <= tS2; k++)
            tamGrupos[j][k] = backtrackTamGrupos[altura][j][k];
    marcar(iMejor, jMejor, pos);
    if(backtrack(altura + 1, sudoku, grupos, tamGrupos))
        return true;
}
temp >>= 1;
}
return false;
}
}

static void generarZonas()
{
    for(int i = 0; i < tS2; i++)
    {
        for(int j = 0; j < tS2; j++)
        {
            zonas[i][j][0] = i;
            zonas[i][j][1] = j;
        }
    }
    for(int i = 0; i < tS2; i++)
    {
        for(int j = 0; j < tS2; j++)
        {
            zonas[i + tS2][j][0] = j;
            zonas[i + tS2][j][1] = i;
        }
    }
}

```

```

int cuenta = 0;
for(int i = 0; i < tS; i++)
    for(int j = 0; j < tS; j++)
    {
        int ia = i * tS;
        int ib = ia + tS;
        int ja = j * tS;
        int jb = ja + tS;
        int actual = 0;
        for(int a = ia; a < ib; a++)
            for(int b = ja; b < jb; b++)
            {
                zonas[cuenta + tS2 * 2][actual][0] = a;
                zonas[cuenta + tS2 * 2][actual++][1] = b;
            }
        cuenta++;
    }
}

static int todos = (1 << tS2) - 1;
}

```

.....

10.8. Gaussian elimination

```

static class Matrix {
    Rational[][] data;
    int rows, cols;

    Matrix(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        data = new Rational[rows][cols];
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                data[i][j] = Rational.zero;
    }

    void clonar(Matrix a) {
        Matrix nueva = new Matrix(rows, cols);
        for (int i = 0; i < rows; i++)
            for (int j = 0; j < cols; j++)
                nueva.data[i][j] = data[i][j];
    }
}

```

```

}

void swapRow(int row1, int row2) {
    Rational[] tmp = data[row2];
    data[row2] = data[row1];
    data[row1] = tmp;
}

void multRow(int row, Rational coeff) {
    for (int j = 0; j < cols; j++)
        data[row][j] = data[row][j].times(coeff);
}

void addRows(int destRow, int srcRow, Rational factor) {
    for (int j = 0; j < cols; j++)
        data[destRow][j] = data[destRow][j].
        plus(data[srcRow][j].times(factor));
}

void printMat() {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++)
            System.out.print(data[i][j] + " ");
        System.out.println();
    }
}

};

static void gaussianElim(Matrix m){
    int rows = m.rows;
    for (int i = 0; i < rows; i++) {
        int maxrow = i;
        Rational maxval = m.data[i][i];
        for (int k = i + 1; k < rows; k++) {
            if (maxval.abs().compareTo(m.data[k][i].abs()) < 0) {
                maxval = m.data[k][i];
                maxrow = k;
            }
        }
        if (maxval.compareTo(Rational.zero) == 0)
            return;
    }
}

```

```

m.swapRow(maxrow, i);
m.multRow(i, maxval.reciprocal());
for (int k = 0; k < rows; k++)
    if (k != i)
        m.addRows(k, i, m.data[k][i].negate());
}
}

```

.....

10.9. Catalan numbers

In combinatorial mathematics, the Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively defined objects. They are named after the Belgian mathematician Eugène Charles Catalan (1814–1894).

The Nth catalan number is given by:

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} \quad \text{for } n \geq 0$$

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 (see also Dyck language). For example, the following are the Dyck words of length 6:

XXXYYY XYXXYY XYXYXY XYYXXY XYYXYY

Re-interpreting the symbol X as an open parenthesis and Y as a close parenthesis, C_n counts the number of expressions containing n pairs of parentheses which are correctly matched:

((())) ()(()) ()()() (())() ()()

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).

Successive applications of a binary operator can be represented in terms of a full binary tree. (A rooted binary tree is full if every vertex has either two children or no children.) It follows that C_n is the number of full binary trees with $n + 1$ leaves.

If the leaves are labelled, we have the quadruple factorial numbers.

C_n is the number of non-isomorphic ordered trees with $n+1$ vertices. (An ordered tree is a rooted tree in which the children of each vertex are given a fixed left-to-right order.)

C_n is the number of monotonic paths along the edges of a grid with $n \times n$ square cells, which do not pass above the diagonal. A monotonic path is one which starts in the lower left corner, finishes in the upper right corner, and consists entirely of edges pointing rightwards or upwards. Counting such paths is equivalent to counting Dyck words: X stands for move right and Y stands for move up.

C_n is the number of different ways a convex polygon with $n + 2$ sides can be cut into triangles by connecting vertices with straight lines.

C_n is the number of stack-sortable permutations of $1, \dots, n$. A permutation w is called stack-sortable if $S(w) = (1, \dots, n)$, where $S(w)$ is defined recursively as follows: write $w = unv$ where n is the largest element in w and u and v are shorter sequences, and set $S(w) = S(u)S(v)n$, with S being the identity for one-element sequences. These are the permutations that avoid the pattern 231.

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. For $n = 3$, these permutations are 132, 213, 231, 312 and 321. For $n = 4$, they are 1432, 2143, 2413, 2431, 3142, 3214, 3241, 3412, 3421, 4132, 4213, 4231, 4312 and 4321.

C_n is the number of noncrossing partitions of the set $1, \dots, n$. A fortiori, C_n never exceeds the n th Bell number. C_n is also the number of noncrossing partitions of the set $1, \dots, 2n$ in which every block is of size 2. The conjunction of these two facts may be used in a proof by mathematical induction that all of the free cumulants of degree more than 2 of the Wigner semicircle law are zero. This law is important in free probability theory and the theory of random matrices.

C_n is the number of ways to tile a staircase shape of height n with n rectangles.

C_n is the number of standard Young tableaux whose diagram is a 2-by- n rectangle. In other words, it is the number ways the numbers $1, 2, \dots, 2n$ can be arranged in a 2-by- n rectangle so that each row and each column is increasing. As such, the formula can be derived as a special case of the hook-length formula.

C_n is the number of ways that the vertices of a convex $2n$ -gon can be paired so that the line segments joining paired vertices do not intersect.

C_n is the number of semiorders on n unlabeled items.

10.10. Bell numbers

In combinatorics, the n th Bell number, named after Eric Temple Bell, is the number of partitions of a set with n members, or equivalently, the number of equivalence relations on it. Starting with $B_0 = B_1 = 1$, the first few Bell numbers are:

1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975

The Bell numbers satisfy this recursion formula:

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

10.11. Polinomios

```
#include <complex>
#include <iostream>
#include <cstdio>
#include <vector>
#include <algorithm>

using namespace std;

typedef complex <double> dcmplx;

#include <complex>
#include <cmath>

#define PREC 0.1
#define ERR 0.1

using namespace std;

class Polynom {
public:
    int grado, max;
    double *coef;

    Polynom(int grado, int max);
    ~Polynom();

    Polynom* operator+= (Polynom *b);
    Polynom* operator-= (Polynom *b);
    Polynom* operator*= (Polynom *b);
    Polynom* operator/= (Polynom *b);
    Polynom* operator%= (Polynom *b);

    complex<double> eval(complex<double> x);
    void derivate();
```

```

void clear();
void print();
void simplify();
void div(Polynom *b, Polynom *cos, Polynom *residuo);
void copyFrom(Polynom *p);
void remMultRoot();
void laguerre(complex<double>* roots);

bool isZero();
};

void mcd(Polynom *a, Polynom *b, Polynom *res);

Polynom::Polynom(int grado, int max) {
    if (max != 0)    this->coef = new double[max];
    this->max=max;
    clear();
    this->grado=grado;
}

Polynom::~Polynom() {
    delete [] coef;
}

Polynom* Polynom::operator +=(Polynom* b) {
    int i = (b->grado > grado) ? b->grado : grado;
    this->grado=i;
    for (; i>=0; i--) {
        coef[i] += b->coef[i];
    }
    simplify();
    return this;
}

Polynom* Polynom::operator -=(Polynom* b) {
    int i = (b->grado > grado) ? b->grado : grado;
    this->grado=i;
    for (; i>=0; i--) {
        coef[i] -= b->coef[i];
    }
}

```

```

    simplify();
    return this;
}

Polynom* Polynom::operator *=(Polynom* b) {
    double nc[max];
    for (int i=0; i<max; i++) nc[i]=0.0;
    for (int i=0; i<=grado; i++) {
        for (int j=0; j<=b->grado; j++) {
            nc[i+j] += coef[i]*b->coef[j];
        }
    }
    for (int i=0; i<max; i++) coef[i]=nc[i];
    grado+=b->grado;
    simplify();
    return this;
}

Polynom* Polynom::operator /=(Polynom* b) {
    this->div(b, this, NULL);
    return this;
}

Polynom* Polynom::operator %=(Polynom* b) {
    Polynom* cos = new Polynom(0,max);
    Polynom* res = new Polynom(0,max);
    div(b,cos,res);
    this->copyFrom(res);
    delete cos; delete res;
    return this;
}

complex<double> Polynom::eval(complex<double> x) {
    complex<double> fx;
    for (int i=grado; i>=0; i--) {
        fx *= x;
        fx += coef[i];
    }
    return fx;
}

```

```

void Polynom::derivate() {
    for (int i=1; i<=grado; i++) {
        coef[i-1] = coef[i]*i;
    }
    grado--;
}

void Polynom::clear() {
    for (int i=0; i<max; i++) {
        coef[i]=0.0;
    }
    this->grado=0;
}

void Polynom::print() {
    for (int i=0; i<=grado; i++) {
        printf("fx%i ",coef[i],i);
    }
    printf("\n");
}

void Polynom::simplify() {
    while (grado>0 && coef[grado] == 0) grado--;
}

void Polynom::copyFrom(Polynom* p) {
    this->grado = p->grado;
    for (int i=grado; i>=0; i--) {
        this->coef[i] = p->coef[i];
    }
}

void Polynom::remMultRoot() {
    Polynom *a = this;
    while(true)
    {
        Polynom b(0,max);
        Polynom gcd(0,max);
        b.copyFrom(this);
        b.derivate();
        mcd(a,&b,&gcd);
    }
}

```

```

        if (gcd.grado>0) *a /= &gcd;
        else break;
    }
}

// It doesn't change b or this.
// Just changes cos and return
// residuo if not null
void Polynom::div(Polynom* b, Polynom* cos, Polynom* residuo) {
    if (cos != this) cos->copyFrom(this);
    double *res = cos->coef;
    int n=grado, m=b->grado, ng=n-m;
    cos->coef = new double[max];
    clear();
    while (n>=m) {
        double val = res[n]/b->coef[m];
        cos->coef[n-m] = val;
        for (int i=m, j=n; i>=0; i--) {
            res[j--] -= val*b->coef[i];
        }
        n--;
    }
    cos->grado = ng;
    cos->simplify();
    if (residuo == NULL) delete [] res;
    else {
        delete [] residuo->coef;
        residuo->coef = res;
        residuo->max = cos->max;
        residuo->grado = b->grado;
        residuo->simplify();
    }
}

void Polynom::laguerre(complex<double>* roots) {
    Polynom poly(0,max); //the polynom
    Polynom polyp(0,max); //It's derivative
    Polynom polypp(0,max); //2nd order derivative
    poly.copyFrom(this);
    complex<double>* root = roots;
    complex<double> G,H,a,den,fx;
}

```

```

complex<double> n(grado,0), one(1,0);
while ( poly.grado>0 ) {
    polyp.copyFrom(&poly);
    polyp.derivate();
    polypp.copyFrom(&polyp);
    polypp.derivate();
    int i = 0;
    do {
        fx = poly.eval(*root);
        if (abs(fx) < ERR) break;
        G = polyp.eval(*root) / fx;
        H = G*G - polypp.eval(*root)/fx;
        den = sqrt( (n-one)*(H*n-G*G) );
        if (abs(G-den) > abs(G+den))
            den = G-den;
        else
            den = G+den;
        a = n / den;
        *root -= a;
    } while (i++ < 50);
    if (root->imag() < ERR) {
        polyp.grado=1;
        polyp.coef[0]=-1*root->real();
        polyp.coef[1]=1;
        poly /= &polyp;
        root++;
    } else {
        polyp.grado=2;
        polyp.coef[0]=norm(*root);
        polyp.coef[1]=-2*root->real();
        polyp.coef[2]=1;
        poly /= &polyp;
        root[1] = conj(*root);
        root+=2;
    }
}

}

bool Polynom::isZero() {
    return ( this->grado==0 && *(this->coef)==0.0 );
}

```

```

void mcd(Polynom* na, Polynom* nb, Polynom *res) {
    Polynom a(0,20); Polynom b(0,20);
    a.copyFrom(na); b.copyFrom(nb);
    Polynom *t = new Polynom(0,a.max);
    Polynom *residuo = new Polynom(0,a.max);
    while (! b.isZero() ) {
        a.div(&b,t,residuo);
        a.copyFrom(&b);
        b.copyFrom(residuo);
        t->clear();
    }
    delete t; delete residuo;
    res->copyFrom(&a);
}

dcmplx Polynom::evaluate(dcmplx x){
    dcmplx res = 0;
    for(int i = 0; i <= grado; i++)
        res += ((dcmplx) coef[i]) * pow(x, i);
    return res;
}

void Polynom::laGuerre(dcmplx *xk, int iteraciones) {
    dcmplx n = grado;
    Polynom *pp = new Polynom(max, max);
    pp->copyFrom(this);
    pp->derivate();
    Polynom *p2 = new Polynom(max, max);
    p2->copyFrom(pp);
    p2->derivate();
    for (int i = 0; i < iteraciones; i++) {
        dcmplx G = pp->evaluate(*xk) / evaluate(*xk);
        dcmplx H = G * G -
            p2->evaluate(*xk) / evaluate(*xk);

        dcmplx a1 = G +
            sqrt((n - (dcmplx) 1) * (n * H - G * G));

        dcmplx a2 = G -
            sqrt((n - (dcmplx) 1) * (n * H - G * G));
    }
}

```

