Resumen de algoritmos para maratones de programación

Manuel Pineda

29 de marzo de 2013

Índice

1. Template

```
using namespace std;
#include<algorithm>
#include<iostream>
#include<sstream>
#include<string>
#include<vector>
#include<queue>
#include<stack>
#include<map>
#include<set>

#include<climits>
#include<cstring>
#include<cstdio>
#include<cmath>

#define For(i,a) for(int i=0;i<a;++i)
#define foreach(x,v) for(typeof (v).begin() x = (v).begin(); \
x!= (v).end(); x++)
#define all(x) x.begin(),x.end()
#define rall(x) x.rbegin(),x.rend()
#define D(x) cout<< #x " = "<<(x)<<endl
#define Dbg if(1)
#define MAXNODES 1000

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

const double pi=acos(-1);
typedef long long int lli;
typedef pair<int , int> pii;


int main(){

    return 0;
}
```

2. Grpahs

2.1. Shortest path

2.1.1. Dijkstra

Calcula la ruta de menor costo desde un nodo origen hasta el resto de nodos del grafo. **El grafo no puede contener ciclos negativos, se queda en un ciclo infinito.** Usar bellman-ford en ese caso.

Complejidad $O(E\ logV)$

```
struct edge{
  int to, weight;
  edge() {}
  edge(int t, int w) : to(t), weight(w) {}
  bool operator < (const edge &that) const {
    return weight > that.weight;
  }
};

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

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

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

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

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

2.1.2. 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.

```
//Complejidad: O(V*E)

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

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

  d[s] = 0;
  for (int i=0, changed = true; i<n-1 && changed; ++i){
    changed = false;
    for (int u=0; u<n; ++u){
      for (int k=0; k<g[u].size(); ++k){
        int v = g[u][k].v, w = g[u][k].w;
        if (d[u] + w < d[v]){
          d[v] = d[u] + w;
          p[v] = u;
          changed = true;
        }
      }
    }
  }

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

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

        return false;
      }
    }
  }
  return true;
}
```

2.2. All-pairs shortest paths

Es una versión del shortest path problem donde hay que hallar 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 VXV (es decir, el grafo no es muy

denso).

2.2.1. Floyd-Warshall

Este algoritmo funciona por programación dinámica.

```
//Complejidad: O(V^3)
//No funciona si hay ciclos de peso negativo
// g[i][j] = Distancia entre el nodo i y el j.
unsigned long long g[MAXNODES][MAXNODES];
void floyd(int n){
  //Llenar g antes
  for (int k=0; k<n; ++k){
    for (int i=0; i<n; ++i){
      for (int j=0; j<n; ++j){
        g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
      }
    }
  }
  //Acá se cumple que g[i][j] = Longitud de la ruta más corta
  //de i a j.
}
```

.....

2.2.2. 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.

Pseudocodigo:
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 lengthw(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(V2log V + VE): the algorithm uses O(VE) time for the Bellman–Ford stage of the algorithm, and $O(VlogV + E)$ 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 $O(V3)$.

2.3. Minimum Spanning Tree

El minimum spanning tree de un grafo no dirigido, es un subconjunto de aristas que contienen todos los vertices del grafo y además el peso total (sumatoria de las aristas) es minimizado.

2.3.1. Algoritmo de kruskal

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

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

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

.....

2.3.2. Algoritmo de Prim

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

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

.....

2.4. Strongly Connected Components

Un grafo dirigido es llamado strongly connected si existe un path desde cada vértice en el grafo a cualquier otro vértice. El strongly connected components de un grafo dirigido son los strongly connected subgrafos.

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

 *d[i] = Discovery time of node i. (Initialize to -1)
```

```
 *low[i] = Lowest discovery time reachable from node
 i. (Doesn't need to be initialized)
 *scc[i] = Strongly connected component of node i. (Doesn't
 need to be initialized)
 *s = Stack used by the algorithm (Initialize to an empty
 stack)
 *stacked[i] = True if i was pushed into s. (Initialize to
 false)
 *ticks = Clock used for discovery times (Initialize to 0)
 *current_scc = ID of the current_scc being discovered
 (Initialize to 0)
*/
vector<int> g[MAXN];
int d[MAXN], low[MAXN], scc[MAXN];
bool stacked[MAXN];
stack<int> s;
int ticks, current_scc;
void tarjan(int u){
  d[u] = low[u] = ticks++;
  s.push(u);
  stacked[u] = true;
  const vector<int> &out = g[u];
  for (int k=0, m=out.size(); k<m; ++k){
    const int &v = out[k];
    if (d[v] == -1){
      tarjan(v);
      low[u] = min(low[u], low[v]);
    }else if (stacked[v]){
      low[u] = min(low[u], low[v]);
    }
  }
  if (d[u] == low[u]){
    int v;
    do{
      v = s.top();
      s.pop();
      stacked[v] = false;
      scc[v] = current_scc;
    }while (u != v);
    current_scc++;
  }
}
```

.....

2.5. Bridges

Un bridge(también llamado cut-edge o cut arc) es una arista cuya eliminación incrementa el número de componentes conectados. De manera equivalente un edge es un bridge si y solo si este no es contenido en algún ciclo. Un grado es llamado bridgeless si este no contiene bridges.

```
#include <algorithm>
#include <vector>
#include <map>
#include <cstdio>
#include <cstring>
#define foreach(x,v) for(typeof ((v).begin()) x = (v).begin();\
x != (v).end() ; ++x)
using namespace std;

const int MP = 10001;

int visited[MP];
int prev[MP], low[MP], d[MP];
vector< vector<int> > g;
vector< pair<int,int> > bridges;
int n, ticks;

void dfs(int u){
  visited[u] = true;
  d[u] = low[u] = ticks++;
  for (int i=0; i<g[u].size(); ++i){
    int v = g[u][i];
    if (prev[u] != v){
      if(!visited[v]){
         prev[v] = u;
         dfs(v);
         if (d[u] < low[v]){
           bridges.push_back(make_pair(min(u,v),max(u,v)));
         }
         low[u] = min(low[u], low[v]);
      }else{
```

```
        low[u] = min(low[u], d[v]);
      }
    }
  }
}


/**Example of use **/

int main(){
  int ncases;scanf("%d",&ncases);
  for(int id = 1 ; id<=ncases; ++id){
    printf("Case %d:\n",id);
    scanf("%d",&n);
    memset(visited,false,sizeof(visited));
    memset(prev,-1,sizeof(prev));
    g.assign(n, vector<int>());
    bridges.clear();
    if (n == 0){ printf("0 critical links\n"); continue; }

    for (int i=0; i<n; ++i){
      int node, deg;
      scanf("%d (%d)", &node, &deg);
      g[node].resize(deg);
      for (int k=0, x; k<deg; ++k){
        scanf("%d", &x);
        g[node][k] = x;
      }
    }

    ticks = 0;
    for (int i=0; i<n; ++i){
      if (!visited[i]){
        dfs(i);
      }
    }

    sort(bridges.begin(), bridges.end());

    printf("%d critical links\n", bridges.size());
    foreach(p, bridges)
      printf("%d - %d\n", p->first, p->second);
  }
  return 0;
}
```

.....

2.6. Puntos de articulación

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

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

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

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

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

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

.....

2.7. 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 asi: desde la negación de la primera variable a la segunda, y de la negacion de la segunda variable a la primera, esto indica, que si no se puede cumplir una de las variables hay que cumplir la otra: (x0 — -x3) equiv (-x0 -¿-x3) equiv (x3 -¿x0)

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.

2.8. 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;
//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;
  }
};
```

.....

2.9. 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.

2.9.1. Dinnic

Adjacency list implementation of Dinic's blocking flow algorithm.
This is very fast in practice, and only loses to push-relabel flow.
Running time: $O(|V|^2|E|)$
INPUT:
- graph, constructed using AddEdge()
- source
- sink
OUTPUT:
- maximum flow value
- To obtain the actual flow values, look at all edges with tem $capacity > 0$(zero capacity edges are residual edges).

```
#include < cmath >
#include < vector >
#include < iostream >
#include < queue >

using namespace std;

const int INF = 2000000000;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index): from(from),
  to(to),
  cap(cap),
  flow(flow),
  index(index) {}
};

struct Dinic {
  int N;
  vector < vector < Edge > > G;
  vector < Edge * > dad;
```

```
vector < int > Q;
Dinic(int N): N(N),
G(N),
dad(N),
Q(N) {}
void AddEdge(int from, int to, int cap) {
  G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
  if (from == to) G[from].back().index++;
  G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
}
long long BlockingFlow(int s, int t) {
  fill(dad.begin(), dad.end(), (Edge * ) NULL);
  dad[s] = & G[0][0] - 1;
  int head = 0, tail = 0;
  Q[tail++] = s;
  while (head < tail) {
    int x = Q[head++];
    for (int i = 0; i < G[x].size(); i++) {
      Edge & e = G[x][i];
      if (!dad[e.to] && e.cap - e.flow >0) {
        dad[e.to] = & G[x][i];
        Q[tail++] = e.to;
      }
    }
  }
  if (!dad[t]) return 0;
  long long totflow = 0;
  for (int i = 0; i < G[t].size(); i++) {
    Edge * start = & G[G[t][i].to][G[t][i].index];
    int amt = INF;
    for (Edge * e = start; amt && e != dad[s]; e = dad[e->from]){
      if (!e) {
        amt = 0;
        break;
      }
      amt = min(amt, e->cap - e->flow);
    }
    if (amt == 0) continue;
    for (Edge * e = start; amt && e != dad[s]; e = dad[e->from]){
      e->flow += amt;
      G[e->to][e->index].flow -= amt;
      }
      totflow += amt;
    }
    return totflow;
  }
  long long GetMaxFlow(int s, int t) {
    long long totflow = 0;
    while (long long flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};
```

.....

2.9.2. Min cost max flow

Implementation of min cost max flow algorithm using adjacency matrix (Edmonds and Karp 1972). This implementation keeps track of forward and reverse edges separately (so you can set $cap[i][j] \neq cap[j][i]$). For a regular max flow, set all edge costs to 0.

Running time, $O(|V|^2)$ cost per augmentation
max flow: $O(|V|^3)$augmentations
min cost max flow: $O(|V|^4 * MAX_EDGE_COST)$ augmentations
INPUT:

- graph, constructed using AddEdge()
- source
- sink

OUTPUT:
- (maximum flow value, minimum cost value)
- To obtain the actual flow, look at positive values only.

```
#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector < int > VI;
typedef vector < VI > VVI;
```

```
typedef long long L;
typedef vector < L > VL;
typedef vector < VL > VVL;
typedef pair < int, int > PII;
typedef vector < PII > VPII;

const L INF = numeric_limits < L > ::max() / 4;

struct MinCostMaxFlow {
  int N;
  VVL cap, flow, cost;
  VI found;
  VL dist, pi, width;
  VPII dad;
  MinCostMaxFlow(int N): N(N),
  cap(N, VL(N)),
  flow(N, VL(N)),
  cost(N, VL(N)),
  found(N),
  dist(N),
  pi(N),
  width(N),
  dad(N) {}
  void AddEdge(int from, int to, L cap, L cost) {
    this - > cap[from][to] = cap;
    this - > cost[from][to] = cost;
  }
  void Relax(int s, int k, L cap, L cost, int dir) {
    L val = dist[s] + pi[s] - pi[k] + cost;
    if (cap && val < dist[k]) {
      dist[k] = val;
      dad[k] = make_pair(s, dir);
      width[k] = min(cap, width[s]);
    }
  }
  L Dijkstra(int s, int t) {
    fill(found.begin(), found.end(), false);
    fill(dist.begin(), dist.end(), INF);
    fill(width.begin(), width.end(), 0);
    dist[s] = 0;
    width[s] = INF;
    while (s != -1) {
      int best = -1;
      found[s] = true;
      for (int k = 0; k < N; k++) {
        if (found[k]) continue;
        Relax(s, k, cap[s][k] - flow[s][k], cost[s][k], 1);
        Relax(s, k, flow[k][s], -cost[k][s], -1);
        if (best == -1 || dist[k] < dist[best]) best = k;
      }
      s = best;
    }
    for (int k = 0; k < N; k++)
      pi[k] = min(pi[k] + dist[k], INF);
    return width[t];
  }
  pair < L,
  L > GetMaxFlow(int s, int t) {
    L totflow = 0, totcost = 0;
    while (L amt = Dijkstra(s, t)) {
      totflow += amt;
      for (int x = t; x != s; x = dad[x].first) {
        if (dad[x].second == 1) {
          flow[dad[x].first][x] += amt;
          totcost += amt * cost[dad[x].first][x];
        } else {
          flow[x][dad[x].first] -= amt;
          totcost -= amt * cost[x][dad[x].first];
        }
      }
    }
    return make_pair(totflow, totcost);
  }
};
```

2.9.3. Push Relabel

Adjacency list implementation of FIFO push relabel maximum flow with the gap relabeling heuristic. This implementation is significantly faster than straight Ford-Fulkerson. It solves random problems with 10000 vertices and 1000000 edges in a few seconds, though it is possible to construct test cases that achieve the

worst-case.
Running time:
$O(|V|^3)$
INPUT:
- graph, constructed using AddEdge()
- source
- sink
OUTPUT:
- maximum flow value
- To obtain the actual flow values, look at all edges with *capacity* > 0(zero capacity edges are residual edges).

```
#include <cmath>
#include <vector>
#include <iostream>
#include <queue>

using namespace std;

typedef long long LL;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index): from(from),
  to(to),
  cap(cap),
  flow(flow),
  index(index) {}
};
struct PushRelabel {
  int N;
  vector < vector < Edge > > G;
  vector < LL > excess;
  vector < int > dist, active, count;
  queue < int > Q;
  PushRelabel(int N): N(N),
  G(N),
  excess(N),
  dist(N),
  active(N),
  count(2 * N) {}
  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }
  void Enqueue(int v) {
    if (!active[v] && excess[v] > 0) {
      active[v] = true;
      Q.push(v);
    }
  }
  void Push(Edge & e) {
    int amt = int(min(excess[e.from], LL(e.cap - e.flow)));
    if (dist[e.from] <= dist[e.to] || amt == 0) return;
    e.flow += amt;
    G[e.to][e.index].flow -= amt;
    excess[e.to] += amt;
    excess[e.from] -= amt;
    Enqueue(e.to);
  }
  void Gap(int k) {
    for (int v = 0; v < N; v++) {
      if (dist[v] < k) continue;
      count[dist[v]]--;
      dist[v] = max(dist[v], N + 1);
      count[dist[v]]++;
      Enqueue(v);
    }
  }
  void Relabel(int v) {
    count[dist[v]]--;
    dist[v] = 2 * N;
    for (int i = 0; i < G[v].size(); i++)
      if (G[v][i].cap - G[v][i].flow > 0)
        dist[v] = min(dist[v], dist[G[v][i].to] + 1);
    count[dist[v]]++;
    Enqueue(v);
  }
  void Discharge(int v) {
    for (int i = 0; excess[v] > 0 && i < G[v].size(); i++) Push(G[v][i])
    if (excess[v] > 0) {
      if (count[dist[v]] == 1)
```

```
        Gap(dist[v]);
      else
        Relabel(v);
    }
  }
  LL GetMaxFlow(int s, int t) {
    count[0] = N - 1;
    count[N] = 1;
    dist[s] = N;
    active[s] = active[t] = true;
    for (int i = 0; i < G[s].size(); i++) {
      excess[s] += G[s][i].cap;
      Push(G[s][i]);
    }
    while (!Q.empty()) {
      int v = Q.front();
      Q.pop();
      active[v] = false;
      Discharge(v);
    }
    LL totflow = 0;
    for (int i = 0; i < G[s].size(); i++) totflow += G[s][i].flow;
    return totflow;
  }
};
```

.....

2.9.4. Min Cost matching

Min cost bipartite matching via shortest augmenting paths

This is an $O(n^3)$ implementation of a shortest augmenting path algorithm for finding min cost perfect matchings in dense graphs. In practice, it solves $1000x1000$ problems in around 1 second.

cost[i][j] = cost for pairing left node i with right node j

Lmate[i] = index of right node that left node i pairs with

Rmate[j] = index of left node that right node j pairs with

The values in cost[i][j] may be positive or negative. To perform maximization, simply negate the cost[][] matrix.

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

using namespace std;

typedef vector < double > VD;
typedef vector < VD > VVD;
typedef vector < int > VI;
double MinCostMatching(const VVD & cost, VI & Lmate, VI & Rmate) {
  int n = int(cost.size());
  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(v[j], cost[i][j] - u[i]);
  }
  // construct primal solution satisfying complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < 1e-10) {
        Lmate[i] = j;
        Rmate[j] = i;
        mated++;
        break;
      }
    }
  }
  VD dist(n);
  VI dad(n);
  VI seen(n);
  // repeat until primal solution is feasible
  while (mated < n) { // find an unmatched left node
```

```
int s = 0;
while (Lmate[s] != -1) s++;
// initialize Dijkstra
fill(dad.begin(), dad.end(), -1);
fill(seen.begin(), seen.end(), 0);
for (int k = 0; k < n; k++)
  dist[k] = cost[s][k] - u[s] - v[k];
int j = 0;
while (true) {
  // find closest
  j = -1;
  for (int k = 0; k < n; k++) {
    if (seen[k]) continue;
    if (j == -1 || dist[k] < dist[j]) j = k;
  }
  seen[j] = 1;
  // termination condition
  if (Rmate[j] == -1) break;
  // relax neighbors
  const int i = Rmate[j];
  for (int k = 0; k < n; k++) {
    if (seen[k]) continue;
    const double new_dist = dist[j] + cost[i][k] - u[i] - v[k];
    if (dist[k] > new_dist) {
      dist[k] = new_dist;
      dad[k] = j;
    }
  }
}
// update dual variables
for (int k = 0; k < n; k++) {
  if (k == j || !seen[k]) continue;
  const int i = Rmate[k];
  v[k] += dist[k] - dist[j];
  u[i] -= dist[k] - dist[j];
}
u[s] += dist[j];
// augment along path
while (dad[j] >= 0) {
  const int d = dad[j];
  Rmate[j] = Rmate[d];
  Lmate[Rmate[j]] = j;
  j = d;
}
Rmate[j] = s;
Lmate[s] = j;
mated++;
}
double value = 0;
for (int i = 0; i < n; i++)
  value += cost[i][Lmate[i]];
return value;
}
```

.....

2.9.5. Maximum bipartite matching

This code performs maximum bipartite matching.
Running time: $O(|E||V|)$ – often much faster in practice
INPUT: w[i][j] = edge between row node i and column node j OUTPUT: mr[i] = assignment for row node i, -1 if unassigned mc[j] = assignment for column node j, -1 if unassigned function returns number of matches made

```
#include <vector>

using namespace std;

typedef vector < int > VI;
typedef vector < VI > VVI;
bool FindMatch(int i, const VVI & w, VI & mr, VI & mc, VI & seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}
```

```
int BipartiteMatching(const VVI & w, VI & mr, VI & mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);
  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

.....

2.9.6. Min cut

Adjacency matrix implementation of Stoer-Wagner min cut algorithm.
Running time: $O(|V|^3)$
INPUT:
- graph, constructed using AddEdge()
OUTPUT: - (min cut value, nodes in half of min cut)

```
#include <cmath>
#include <vector>
#include <iostream>

using namespace std;

typedef vector < int > VI;
typedef vector < VI > VVI;
const int INF = 1000000000;

pair < int, VI > GetMinCut(VVI & weights) {
  int N = weights.size();
  VI used(N), cut, best_cut;
  int best_weight = -1;
  for (int phase = N - 1; phase >= 0; phase--) {
    VI w = weights[0];
    VI added = used;
    int prev, last = 0;
    for (int i = 0; i < phase; i++) {
      prev = last;
      last = -1;
      for (int j = 1; j < N; j++)
        if (!added[j] && (last == -1 || w[j] > w[last])) last = j;
      if (i == phase - 1) {
        for (int j = 0; j < N; j++)
          weights[prev][j] += weights[last][j];
        for (int j = 0; j < N; j++)
          weights[j][prev] = weights[prev][j];
        used[last] = true;
        cut.push_back(last);
        if (best_weight == -1 || w[last] < best_weight) {
          best_cut = cut;
          best_weight = w[last];
        }
      } else {
        for (int j = 0; j < N; j++)
          w[j] += weights[last][j];
        added[last] = true;
      }
    }
  }
  return make_pair(best_weight, best_cut);
}
```

.....

2.10. 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).
Code real example using TarjanOLCA.

```
struct edge{
    int ancestor, weight;
  int color;
  long long dist_root;
  vector<int> childrens;
  vector<int> q;
  edge(){}
```

```
  edge(int aa,int ww): ancestor(aa),weight(ww),color(0){}

  bool operator < (const edge &that) const {
    return weight < that.weight;
  }
};

struct consulta{
  int A,B;
  long long ans;
};

int p[MAXNODES], rank[MAXNODES];

void make_set(int x){ p[x] = x, rank[x] = 0; }
void link(int x, int y){
  if (rank[x] > rank[y]) p[y] = x;
  else{ p[x] = y; if (rank[x] == rank[y]) rank[y]++; }
}
int find_set(int x){
  return x != p[x] ? p[x] = find_set(p[x]) : p[x];
}
void merge(int x, int y){ link(find_set(x), find_set(y)); }


vector< pair<int,int> > LCA[MAXNODES];
edge G[MAXNODES];
consulta Q[MAXNODES];

void tarjanOLCA(int u){
  make_set(u);
  G[u].ancestor = u;
  foreach(v,G[u].childrens){
    tarjanOLCA(*v);
    merge(u,*v);
    G[find_set(*v)].ancestor = u;
  }
  G[u].color = black;
  int v;
  for(int i = 0;i < G[u].q.size(); ++i){
    if(u == Q[G[u].q[i]].A)v=Q[G[u].q[i]].B;
    else if(u == Q[G[u].q[i]].B)v=Q[G[u].q[i]].A;
    else  throw "Paila "+ toStr(u);
    if(G[v].color == black)
        Q[G[u].q[i]].ans = G[u].dist_root +
           G[v].dist_root - 2*G[G[find_set(v)].ancestor].dist_root;
  }
}


int main(){
  int nodes,queries;
  int to,we;
  while(1){
    cin>>nodes;
    if(!nodes)break;
    For(i,nodes+2){
      G[i].childrens = vector<int> ();
      G[i].q = vector<int> ();
    }
    G[0].dist_root=0;
    For(i,nodes-1){
      cin>>to>>we;
      G[i+1].color = 0;
      G[i+1].ancestor = to;
      G[to].childrens.push_back(i+1);
      G[i+1].dist_root = G[to].dist_root + we;
    }
    cin>>queries;
    For(i,queries){
      cin>>to>>we;
      Q[i].A = to;
      Q[i].B = we;
      G[to].q.push_back(i);
      G[we].q.push_back(i);
    }
    try {
      tarjanOLCA(0);
    } catch(string c){
      cout<<c<<endl;
    }
    cout<<Q[0].ans;
```

```
    for(int i=1;i<queries;++i){
      cout<<" "<<Q[i].ans;
    }
    cout<<endl;
  }

  return 0;
}
```

.....

2.11. Topological sort

```
queue<int> Q;

for(int i=0;i<v;++i)
  if(in[i]==0) Q.push(i);

while(!Q.empty()){
  int act = Q.front();Q.pop();
  for(int i=0;i<G[act].size();++i){
    in[G[act][i]]--;
    if(in[G[act][i]]==0) Q.push(G[act][i]);
  }
}
```

.....

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 $d[i-1][j-1]+(s1[i-1] == s2[j-1]?0:1))$ representa costo 1 para reemplazo (en caso de que no sean iguales). Con estas concideraciones 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]);
  }
 }
 }
```

.....

3.2.1. Repeat

If you can buy more of one article.

```
struct something{
    int price,cost;
something(){}
something(int P,int C):cost(C),price(P){}
};

vector <someting> todos;
//Llenar el vector con los datos correspondientes.

int dp[tengo+1];

dp[0] = 0;

for(int i = 0;i<=tengo;++i){
  int mm = 0;
  for(int j = 0; j<n ;++j)
    if(i - todos[j].p >= 0 )
  mm = max(mm, dp[i-todos[j].p] + todos[j].c);
  dp[i] = mm;
}

cout<<dp[tengo]<<endl;
```

.....

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. TSP

The travelling salesman problem (TSP) is an NP-hard problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find the shortest possible route that visits each city exactly once and returns to the origin city. The best know solution is $O(2^n)$.

```
static int[][] distancias;
static Integer[][] dpTsp;

// Se llama inicialmente
// tsp(0, (i << N) - 1)
// donde N es el numero de ciudades
static int tsp(int current, int mask)
{
    if(dpTsp[current][mask] != null)
    return dpTsp[current][mask];
if(Integer.bitCount(mask) == 1)
```

```
    {
        return dpTsp[current][mask] =
        distancia[0][Integer.numberOfTrailingZeros(mask)];
    }
    int maskT = mask;
    int j = 0;
    int best = Integer.MAX_VALUE;
    int nextMask = mask & (~(1 << current));
    while(maskT != 0)
    {
        if((maskT & 1) == 1 && j != current)
            best = Math.min(best,
                   distancias[current][j] + tsp(j, nextMask));
        j++;
        maskT >>= 1;
    }
    return dpTsp[current][mask] = best;
}
```

.....

3.6. 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;
  }
}
```

.....

3.7. Minimum biroute pass

The problem of the minimum biroute pass is two find a minimum weight (or length) route that starts from the west-most point or node, makes a pass to the east-most point or node visiting some of the nodes, then makes a second pass from the east-most point or node back to the first one visiting the remaining islands. In each pass the route moves steadily east (in the first pass) or west (in the second pass), but moves as far north or south as needed.

```
double T[MAXP][MAXP];
bool comp[MAXP][MAXP];
double point[MAXP][2];
int N;

double dist(int p1, int p2) {
    double ans = 0;
    for (int i=0; i<2; i++) {
        ans += SQ(point[p1][i]-point[p2][i]);
    }
    return sqrt(ans);
}


double minTour(int e1, int e2) {
    if (e1==N-1 || e2==N-1) return dist(e1,e2);
    if (comp[e1][e2]) return T[e1][e2];

    int n = max(e1,e2)+1;
```

```
    double d1 = minTour(n,e2) + dist(e1,n);
    double d2 = minTour(e1,n) + dist(e2,n);

    T[e1][e2] = min(d1,d2); T[e2][e1]=T[e1][e2];
    comp[e1][e2] = true; comp[e2][e1]=true;
    return T[e1][e2];
}

int main() {
    while ( scanf("%d",&N) != EOF ) {
        for (int i=0; i<N; i++) {
            scanf("%lf %lf", &point[i][0], &point[i][1]);
        }
        for (int i=0; i<=N; i++)
            for (int j=0; j<=N; j++)
                comp[i][j]=false;
        double ans = minTour(0,0);
        printf("%.2lf\n",ans);
    }
}
```

.....

4. Matemáticas

4.1. 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;
  }
}
```

.....

4.2. 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);
}
```

.....

4.3. 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;
}
```

.....

4.4. 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;
}
```

4.5. Test de primalidad

```
/**
* Miller-Rabin. La probabilidad de error disminuye al aumentar
* el numero de iteraciones. Dicha probalidad 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;
}
```

4.6. Combinatoria

4.6.1. Sumas

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

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

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

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

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

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

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

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

4.6.2. 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^2
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 v (x, y, z) un ángulo θ (counterclockwise) con respecto a k (vector unitario que describe el eje de rotación) se tiene:($Rodrigues' rotation formula$).

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

Si se desea representar la anterior rotación como una transformación se obtendría lo siguiente:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} =$$

$$\begin{bmatrix} \mathrm{ct} + (\mathrm{k}_x)^2 * mct & \mathrm{k}_x * k_y * mct - k_z * st & \mathrm{k}_y * st + k_x * k_z * mct \\ \mathrm{k}_z * st + k_x * k_y * mct & \mathrm{ct} + (\mathrm{k}_y)^2 * mct & \text{-}\mathrm{k}_x * st + k_y * k_z * mct \\ \text{-}\mathrm{k}_y * st + k_x * k_z * mct & \mathrm{k}_x * st + k_y * k_z * mct & \mathrm{ct} + (\mathrm{k}_z)^2 * mct \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Donde $ct = \cos\theta$, $st = \sin\theta$, $mct = (1 - \cos\theta)$ y k_x, k_y, k_z son las componentes de k.

5.2.2. Traslación

Para trasladar un punto (x, y, z) en $(\Delta x, \Delta y, \Delta z)$ se tiene(coordenadas homogeneas):

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

5.2.3. Escalamiento

Para escalar un punto (x, y, z) en (vx, vy, vz) se tiene(coordenadas homogeneas):

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} vx & 0 & 0 & 0 \\ 0 & vy & 0 & 0 \\ 0 & 0 & vy & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

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 colineal.
*/
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 colineal 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 %.2lf %.2lf\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(nlog 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 Px and Py, 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 $O(nlogn)$ si se usa un algoritmo eficiente para el *convexhull*.

```
double farthest_point_pair_distance(ArrayList<Point> v){
  int index_miny=0,index_maxy=0;
  double coor_miny=Double.MAX_VALUE;
  double coor_maxy=Double.MIN_VALUE;
  for(int i=0;i<v.size();i++){
    Point a=v.get(i);
    if (a.y<coor_miny){
      index_miny=i;
      coor_miny=a.y;
    }
    if (a.y>coor_maxy){
      index_maxy=i;
```

```
      coor_maxy=a.y;
    }
  }
  double d=v.get(index_miny).sub(v.get(index_maxy)).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_maxy+1)%v.size()).sub(v.get(index_maxy));

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

    double angle_a=caliper_a.angle(edge_a);
    double angle_b=caliper_b.angle(edge_b);
    double min=Math.min(angle_a, angle_b);
    caliper_a=caliper_a.rotate(-min);
    caliper_b=caliper_b.rotate(-min);
    if (Math.abs(angle_a-min)<eps) index_maxy=(index_maxy+1)%v.size();
    if (Math.abs(angle_b-min)<eps) index_miny=(index_miny+1)%v.size();
    d=Math.max(v.get(index_miny).sub(v.get(index_maxy)).norm(),d);
    rotated_angle=rotated_angle + min;
  }
  return d;
}
```

.....

5.14. Smallest Bounding Rectangle

El problema de encontrar el rectángulo de menor area o el rectángulo de menor perímetro que contenga un polígono convexo (el rectángulo no necesariamente tiene que tener sus lados paralelos a los ejes coordenados) se puede resolver en $O(n)$ usando *rotatingcalipers*, de tal manera que si el problema es hallar ese rectángulo para una nube de puntos en lugar de un polígono convexo se puede hallar el *convexhull* para esa nube de puntos y posteriormente aplicar el algoritmo que a continuación se describe. A este algoritmo se le pasa como argumento el conjunto de puntos de un polígono convexo en orden *counterclockwise*, si el parámetro `area` es `true` entonces el algoritmo retorna el área del rectángulo de menor área que envuelve el polígono, sino devuelve el perímetro del rectángulo de menor perímetro que envuelve el polígono. NOTA: la función `rotate2` recibe a $\cos\theta$ y a $\sin\theta$ como parámetros en lugar de θ (donde θ es el ángulo para la rotación de un vector), lo anterior para evitar usar funciones trigonométricas inversas las cuales conducen a errores de precisión en los resultados.

```
static double getminimunrectangle(point[] p,boolean area){
  int [] ind=new int[4];
  for(int i=0;i<4;i++)
   ind[i]=-1;
  double coor_miny=Double.MAX_VALUE;
  double coor_maxy=-1e100;
  double coor_minx=Double.MAX_VALUE;
  double coor_maxx=-1e100;
  for(int i=0;i<p.length;i++){
    point a=p[i];
    if (a.y<=coor_miny){
    if (ind[2]==-1 || a.y<coor_miny || p[ind[2]].x>a.x){
      ind[2]=i;
      coor_miny=a.y;
    }
    }
    if (a.y>=coor_maxy){
    if (ind[3]==-1 || a.y>coor_maxy || p[ind[3]].x<a.x){
      ind[3]=i;
      coor_maxy=a.y;
    }
    }
    if (a.x<=coor_minx){
    if (ind[0]==-1 || a.x<coor_minx || p[ind[0]].y<a.y){
      ind[0]=i;
      coor_minx=a.x;
    }
    }
    if (a.x>=coor_maxx){
    if (ind[1]==-1 || a.x>coor_maxx || p[ind[1]].y>a.y){
      ind[1]=i;
      coor_maxx=a.x;
    }
    }
  }
  double rotated_angle=0;
  point [] calipers=new point[4];
```

```
  calipers[2]=new point(16.0,0.0);
  calipers[3]=new point(-16.0,0.0);
  calipers[0]=new point(0.0,-16.0);
  calipers[1]=new point(0.0,16.0);
  point[] edges=new point[4];
  double min=Double.MAX_VALUE;
  point[] ncalipers=new point[4];
  while(rotated_angle<=3*Math.PI/4){
    for(int i=0;i<4;i++)
      edges[i]=p[(ind[i]+1)%p.length].sub(p[ind[i]]);
    int index=0;
    double max_cos=-2;
    for(int i=0;i<4;i++){
      double val=get_cos_with_caliper(calipers[i],edges[i]);
      if (val>max_cos){
        max_cos=val;
        index=i;
      }
    }
    for(int i=0;i<4;i++)
      ncalipers[i]=calipers[i].rotate2(max_cos,
      get_sin_with_caliper(calipers[index],edges[index]));
    for(int i=0;i<4;i++){
      double ttt=max_cos-get_cos_with_caliper(calipers[i],edges[i]);
      if (Math.abs(ttt)<eps)
        ind[i]=(ind[i]+1)%p.length;
    }
    for(int i=0;i<4;i++)
      calipers[i]=ncalipers[i];
    double value=compute(p[ind[1]],calipers[1],p[ind[0]],calipers[0],
    p[ind[3]],calipers[3],p[ind[2]],calipers[2],area);
    min=Math.min(min, value);
    rotated_angle=rotated_angle + Math.acos(max_cos);
  }
  return min;
}

double
compute(point a,point va,point b,point vb,point c,point vc,point d,
point vd,boolean area){
    point bd=intersectionbtwlines(d,d.add(vd),b,b.add(vb));
    point bc=intersectionbtwlines(b,b.add(vb),c,c.add(vc));
    point ad=intersectionbtwlines(a,a.add(va),d,d.add(vd));
    if (area)
      return ad.sub(bd).norm()* bc.sub(bd).norm();
    else
      return 2*ad.sub(bd).norm()+2*bc.sub(bd).norm();
  }

static double get_cos_with_caliper(point c,point v){
 return c.dot(v)/(16.0*v.norm());
}

static double get_sin_with_caliper(point c,point v){
 return Math.abs(c.cross(v))/(16.0*v.norm());
}
```

5.15. 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] != 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;
}
```

5.16. 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 çonvex 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.

5.16.1. Graham Scan

```
/** Monotone chain algoritm **/

#include <cstdio>
#include <cassert>
#include <vector>
#include <algorithm>
#include <cmath>

using namespace std;
#define REMOVE_REDUNDANT

typedef double T;

const T EPS = 1e-7;

struct PT {
  T x, y;
  PT() {}
  PT(T x, T y): x(x),
  y(y) {}
  bool operator < (const PT & rhs) const {
    return make_pair(y, x) < make_pair(rhs.y, rhs.x);
  }
  bool operator == (const PT & rhs) const {
    return make_pair(y, x) == make_pair(rhs.y, rhs.x);
  }
};
T cross(PT p, PT q) {
  return p.x * q.y - p.y * q.x;
}
T area2(PT a, PT b, PT c) {
  return cross(a, b) + cross(b, c) + cross(c, a);
}
#ifdef REMOVE_REDUNDANT

bool between(const PT & a, const PT & b, const PT & c) {
  return (fabs(area2(a, b, c)) < EPS &&
  (a.x - b.x) * (c.x - b.x) <= 0 && (a.y - b.y) * (c.y - b.y) <= 0);
}
#endif

void ConvexHull(vector < PT > & pts) {
  sort(pts.begin(), pts.end());
  pts.erase(unique(pts.begin(), pts.end()), pts.end());
  vector < PT > up, dn;
  for (int i = 0; i < pts.size(); i++) {
    while (up.size() > 1 && area2(up[up.size() - 2], up.back(),...
      ...pts[i]) >= 0) up.pop_back();
    while (dn.size() > 1 && area2(dn[dn.size() - 2], dn.back(),...
      ...pts[i]) <= 0) dn.pop_back();
    up.push_back(pts[i]);
    dn.push_back(pts[i]);
  }
  pts = dn;
  for (int i = (int) up.size() - 2; i >= 1; i--) pts.push_back(up[i]);
  #ifdef REMOVE_REDUNDANT
  if (pts.size() <= 2) return;
  dn.clear();
  dn.push_back(pts[0]);
  dn.push_back(pts[1]);
  for (int i = 2; i < pts.size(); i++) {
    if (between(dn[dn.size() - 2], dn[dn.size() - 1], pts[i]))
       dn.pop_back();
    dn.push_back(pts[i]);
  }
  if (dn.size() >= 3 && between(dn.back(), dn[0], dn[1])) {
    dn[0] = dn.back();
    dn.pop_back();
  }
  pts = dn;
```

```
  #endif
}

/***Graham scan Algorithm**/

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

point pivot;

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

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

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

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

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

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

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

  //Ahora sí!!!
  for (int i=3; i<p.size(); ++i){
    while (chull.size() >= 2 &&
           cross(chull[chull.size()-2],
                 chull[chull.size()-1],
                 p[i]) <=0){
      chull.erase(chull.end() - 1);
    }
    chull.push_back(p[i]);
  }
  //chull contiene los puntos del convex hull ordenados
  //anti-clockwise.  No contiene ningún punto repetido.  El
  //primer punto no es el mismo que el último, i.e, la última
  //arista va de chull[chull.size()-1] a chull[0]
  return chull;
}
```

.....

5.17. 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;
}
```

.....

5.18. Picks theorem

Área de un polígono en función de los lattice inside y sobre el borde.

$Area = B/2 + I - 1$

B: Lattice points in the boundary.

I: Lattice points inside.

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;
// for (int i = 0; i < text.size(); ++i) {
//    currentState = findNextState(currentState, text[i], 'a');
//
//    Nothing new, let's move on to the next character.
//    if (out[currentState] == 0) continue;
//
//    for (int j = 0; j < keywords.size(); ++j) {
//        if (out[currentState] & (1 << j)) {
//            //Matched keywords[j]
//            cout << "Keyword " << keywords[j]
//                 << " appears from "
//                 << i - keywords[j].size() + 1
//                 << " to " << i << endl;
//        }
//    }
// }
//
// The output of this program is:
//
// Keyword his appears from 1 to 3
// Keyword he appears from 4 to 5
// Keyword she appears from 3 to 5
// Keyword hers appears from 4 to 7

////////////////////////////////////////////////////////////
//               End of Aho-Corasick's algorithm          //
////////////////////////////////////////////////////////////
```

.....

6.3. Suffix Array

In computer science, a suffix array is an array of integers giving the starting positions of suffixes of a string in lexicographical order.

```
// Suffix array construction in O(L log^2 L) time. Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT: string s
//
// OUTPUT: array suffix[] such that suffix[i] = index (from 0 to L-1)
// of substring s[i...L-1] in the list of sorted suffixes.
// That is, if we take the inverse of the permutation suffix[],
// we get the actual suffix array.
#include <vector>
#include <iostream>
#include <string>

using namespace std;

struct SuffixArray {
  const int L;
  string s;
  vector < vector < int > > P;
  vector < pair < pair < int, int > , int > > M;
  SuffixArray(const string & s): L(s.length()),
  s(s),
  P(1, vector < int > (L, 0)),
  M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector < int > (L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level - 1][i], i + skip < L ? ...
             ... P[level - 1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first ...
        ... == M[i - 1].first) ? P[level][M[i - 1].second] : i;
    }
  }
  vector < int > GetSuffixArray() {
    return P.back();
  }
  // returns the length of the longest common prefix of
```

```
  // s[i...L-1] and s[j...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};
int main() {
  // bobocel is the 0'th suffix
  // obocel is the 5'th suffix
  // bocel is the 1'st suffix
  // ocel is the 6'th suffix
  // cel is the 2'nd suffix
  // el is the 3'rd suffix
  // l is the 4'th suffix
  SuffixArray suffix("bobocel");
  vector < int > v = suffix.GetSuffixArray();
  // Expected output: 0 5 1 6 2 3 4
  // 2
  for (int i = 0; i < v.size(); i++) cout << v[i] << " ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
```

.....

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 .ªaccaaddbb". 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)

```
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 - Triee

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 $O(logn)$ time. Changing value of any single element needs O(logn) 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.

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

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

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

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

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

};
```

.....

8.5. Segment Tree

```
using namespace std;
#include<iostream>
#include<vector>
#include<cstdio>
#include<cmath>
```

```
struct segment_tree{
  vector<int  M;
  vector<int> A;
  segment_tree(int n){
    M.resize((n<<2)+4);
    A.resize(n);
  }

  void initialize(int node, int b, int e){
    if (b == e)
      M[node] = b;
    else{
      initialize(node<<1, b, (b + e)>>1);
      initialize((node<<1) + 1, ((b + e)>>1) + 1, e);
      if (A[M[node<<1]] <= A[M[(node<<1) + 1]])
        M[node] = M[node<<1];
      else
        M[node] = M[(node<<1) + 1];
    }
  }

  int query(int node,int b,int e,int i,int j){
    if(i>e or j<b)
      return -1;
    if (b >= i && e <= j)
      return M[node];

    int p1 = query(node<<1, b, (b + e)>>1, i, j);
    int p2 = query((node<<1) + 1, ((b + e)>>1) + 1, e, i, j);

    if (p1 == -1)
      return p2;
    if (p2 == -1)
      return p1;
    if (A[p1] <= A[p2])
      return p1;
    return p2;
  }
};
```

```
/**Examble of use**/

int main(){
  int tc;cin>>tc;
  for(int id = 1;id<=tc;++id){
    int n,q;
    scanf("%d%d",&n,&q);
    segment_tree tree(n);
    for(int i=0;i<n;++i)
      scanf("%d",&tree.A[i]);

    tree.initialize(1,0,n-1);

    int i,j;
    printf("Case %d:\n",id);
    for(int k=0;k<q;++k){
      scanf("%d%d",&i,&j);
      printf("%d\n",tree.A[tree.query(1,0,n-1,i-1,j-1)]);
    }


  }

  return 0;
}
```

.....

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 jsw_hash ( 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):

$$not(x)\&(x+1)$$

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 mimsma
//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;
}
```

GCC definitions: Si se añade ll al final se puede usar con unsigned long long

- `__builtin_clz`(unsigned int x). Retorna la cantidad de leading zeros.
- `__builtin_ctz`(unsigned int x). Retorna la cantidad de trailing zeros.
- `__builtin_popcount`(unsigned int x). Retorna el número de bits en 1.
- `__builtin_parity`(unsigned int x). Retorna el número de bits en 1 módulo 2.

10.2. 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;
    }
```

```
}
```

.....

10.3. 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;
}
```

.....

10.4. 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());
  }
}
```

.....

10.5. 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:

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

Cn 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:

$$XXXYYY \quad XYXXYY \quad XYXYXY \quad XXYYXY \quad XXYXYY$$

.

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

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

Cn 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 Cn is the number of full binary trees with n + 1 leaves.

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

Cn 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.)

Cn is the number of monotonic paths along the edges of a grid with n x 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.

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

Cn is the number of stack-sortable permutations of 1, ..., n. A permutation w is called stack-sortable if S(w) = (1, ..., 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.

Cn is the number of permutations of 1, ..., 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.

Cn is the number of noncrossing partitions of the set 1, ..., n. A fortiori, Cn never exceeds the nth Bell number. Cn is also the number of noncrossing partitions of the set 1, ..., 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.

Cn is the number of ways to tile a stairstep shape of height n with n rectangles.

Cn 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, ..., 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.

Cn 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.

Cn is the number of semiorders on n unlabeled items.

10.6. Bell numbers

In combinatorics, the nth 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 B0 = B1 = 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:

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

%codigofuentesrc/misc/polynomials.cpp

10.7. Subconjuntos

El siguiente codigo genera todos los subconjuntos de una mascara de bits (dado por el entero i) en tiempo lineal en el numero de subconjuntos.

```
static class SubSets implements Iterator <Integer>, Iterable <Integer>
{
    int N;
    int X;
    boolean termino = false;

    SubSets(int n) {
        N = n;
        X = N;
    }


    public boolean hasNext() {
        if(!termino && X == 0) {
            termino = true;
            return true;
        }
        return !termino;
    }
```

```
    public Integer next() {
        int ant = X;
        X = (X - 1) & N;
        return ant;
    }

    public void remove() {}

    public Iterator<Integer> iterator() {
        return this;
    }
}

Se itera sobre los subsets de la siguiente manera

for(int subset : new SubSets(N))
{
    //trabajar con el subset aqui
}
```

.....

10.8. Combinations and permutations

```
public class CombinationGenerator {
    private int[] a;
    private int n;
    private int r;
    private BigInteger numLeft;
    private BigInteger total;

    public CombinationGenerator(int n, int r) {
        this.n = n;
        this.r = r;
        a = new int[r];
        BigInteger nFact = getFactorial(n);
        BigInteger rFact = getFactorial(r);
        BigInteger nminusrFact = getFactorial(n - r);
        total = nFact.divide(rFact.multiply(nminusrFact));
        reset();
    }

    public void reset() {
        for (int i = 0; i < a.length; i++)
            a[i] = i;
        numLeft = total;
    }

    public boolean hasMore() {
        return numLeft.compareTo(BigInteger.ZERO) == 1;
    }

    private static BigInteger getFactorial(int n) {
        BigInteger fact = BigInteger.ONE;
        for (int i = n; i > 1; i--)
            fact = fact.multiply(new BigInteger(Integer.toString(i)));
        return fact;
    }

    public int[] getNext() {
        if (numLeft.equals(total)) {
            numLeft = numLeft.subtract(BigInteger.ONE);
            return a;
        }
        int i = r - 1;
        while (a[i] == n - r + i)
            i--;
        a[i] = a[i] + 1;
        for (int j = i + 1; j < r; j++)
            a[j] = a[i] + j - i;
        numLeft = numLeft.subtract(BigInteger.ONE);
        return a;
    }
}

public class PermutationGenerator {
    private int[] a;
    private long numLeft;
    private long total;

    public PermutationGenerator(int n) {
        a = new int[n];
```

```
    total = getFactorial(n);
    reset();
}

public void reset() {
    for (int i = 0; i < a.length; i++)
        a[i] = i;
    numLeft = total;
}

public boolean hasMore() {
    return numLeft == 0;
}

private static long getFactorial(int n) {
    long fact = 1;
    for (int i = n; i > 1; i--)
        fact = fact * i;
    return fact;
}

public int[] getNext() {
    if (numLeft == total) {
        numLeft--;
        return a;
    }
    int temp;
    int j = a.length - 2;
    while (a[j] > a[j + 1])
        j--;
    int k = a.length - 1;
    while (a[j] > a[k])
        k--;
    temp = a[k];
    a[k] = a[j];
    a[j] = temp;
    int r = a.length - 1;
    int s = j + 1;
    while (r > s) {
        temp = a[s];
        a[s++] = a[r];
        a[r--] = temp;
    }
    numLeft--;
    return a;
  }
}
```

.....

10.9. Rationals

La siguiente clase implementa numeros racionales y evita overflow lo maximo posible. De ser necesario se puede cambiar el numerador y denominador por longs o BigIntegers facilmente.

```
public class Rational implements Comparable <Rational>
{
    static Rational zero = new Rational(0, 1);

    private int num;
    private int den;

    public Rational(int numerator, int denominator) {
        int g = gcd(numerator, denominator);
        num = numerator   / g;
        den = denominator / g;
        if (den < 0) { den = -den; num = -num; }
    }

    public int numerator()   { return num; }
    public int denominator() { return den; }

    public String toString() {
        if (den == 1) return num + "";
        else          return num + "/" + den;
    }

    public int compareTo(Rational b) {
        Rational a = this;
        int lhs = a.num * b.den;
        int rhs = a.den * b.num;
        if (lhs < rhs) return -1;
```

```
    if (lhs > rhs) return +1;
    return 0;
}

public boolean equals(Object y) {
    Rational b = (Rational) y;
    return compareTo(b) == 0;
}

public int hashCode() {
    return this.toString().hashCode();
}

private static int gcd(int m, int n) {
    if (m < 0) m = -m;
    if (n < 0) n = -n;
    if (0 == n) return m;
    else return gcd(n, m % n);
}

public static int lcm(int m, int n) {
    if (m < 0) m = -m;
    if (n < 0) n = -n;
    return m * (n / gcd(m, n));
}

public Rational times(Rational b) {
    Rational a = this;
    Rational c = new Rational(a.num, b.den);
    Rational d = new Rational(b.num, a.den);
    return new Rational(c.num * d.num, c.den * d.den);
}

public Rational plus(Rational b) {
    Rational a = this;
    if (a.compareTo(zero) == 0) return b;
    if (b.compareTo(zero) == 0) return a;
    int f = gcd(a.num, b.num);
    int g = gcd(a.den, b.den);

    Rational s =
        new Rational((a.num / f) * (b.den / g) + (b.num / f) * (a.den / g
        lcm(a.den, b.den));

        s.num *= f;
        return s;
    }

    public Rational negate() {
        return new Rational(-num, den);
    }

    public Rational minus(Rational b) {
        Rational a = this;
        return a.plus(b.negate());
    }

    public Rational reciprocal() { return new Rational(den, num);  }

    public Rational divides(Rational b) {
        Rational a = this;
        return a.times(b.reciprocal());
    }

    public Rational abs() {
        if(num < 0)
            return negate();
        else
            return this;
    }
}
```

.....

10.10. Fibonnacci numbers

In mathematics, the Fibonacci numbers or Fibonacci series or Fibonacci sequence are the numbers in the following integer sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation:

$F_n = F_{n-1} + F_{n-2},$

with seed values:

$F_0 = 0,\ F_1 = 1.$

The Fibonacci numbers can be found in different ways in the sequence of binary strings:

The number of binary strings of length n without consecutive 1s is the Fibonacci number Fn+2. For example, out of the 16 binary strings of length 4, there are F6 = 8 without consecutive 1s – they are 0000, 0100, 0010, 0001, 0101, 1000, 1010 and 1001. By symmetry, the number of strings of length n without consecutive 0s is also Fn+2.

The number of binary strings of length n without an odd number of consecutive 1s is the Fibonacci number Fn+1. For example, out of the 16 binary strings of length 4, there are F5 = 5 without an odd number of consecutive 1s – they are 0000, 0011, 0110, 1100, 1111.

The number of binary strings of length n without an even number of consecutive 0s or 1s is 2Fn. For example, out of the 16 binary strings of length 4, there are 2F4 = 6 without an even number of consecutive 0s or 1s – they are 0001, 1000, 1110, 0111, 0101, 1010.

Like every sequence defined by a linear recurrence with constant coefficients, the Fibonacci numbers have a closed-form solution:

$F_n = \frac{\varphi^n - \psi^n}{\varphi - \psi} = \frac{\varphi^n - \psi^n}{\sqrt{5}}$

where:

$\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}61803\,39887\cdots$

Therefore it can be found by rounding, or in terms of the floor function:

$F_n = \left\lfloor \frac{\varphi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor,\ n \geq 0.$

A 2-dimensional system of linear difference equations that describes the Fibonacci sequence is:

$\binom{F_{k+2}}{F_{k+1}} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \binom{F_{k+1}}{F_k}$

$\vec{F}_{k+1} = A\vec{F}_k$

Z is a Fibonacci number if and only if the closed interval:

$\left[\varphi z - \frac{1}{z}, \varphi z + \frac{1}{z}\right]$

contains a positive integer.

Other identities:

$F_n = F_{n-1} + F_{n-2},$

$\sum_{i=1}^{n} F_i = F_{n+2} - 1$

$F_n^2 - F_{n+r}F_{n-r} = (-1)^{n-r}F_r^2$

$F_n^2 - F_{n+1}F_{n-1} = (-1)^{n-1}$

$F_m F_{n+1} - F_{m+1}F_n = (-1)^n F_{m-n}$

$F_{3n} = 2F_n^3 + 3F_nF_{n+1}F_{n-1} = 5F_n^3 + 3(-1)^n F_n$

$F_{3n+1} = F_{n+1}^3 + 3F_{n+1}F_n^2 - F_n^3$

$F_{3n+2} = F_{n+1}^3 + 3F_{n+1}^2F_n + F_n^3$

$F_{4n} = 4F_nF_{n+1}(F_{n+1}^2 + 2F_n^2) - 3F_n^2(F_n^2 + 2F_{n+1}^2)$

$\gcd(F_m, F_n) = F_{\gcd(m,n)}$

The periodo of the fibonnacci numbers modulo n is less than or equal to 6n.

To calculate fib:

$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$