GEOLETARIO 2.0

```
#include <iostream>
#include <complex>
#include <vector>
#include <stack>
#include <algorithm>
using namespace std;

const long double inf=1e9;
const long double eps=1e-9;
const long double pi=acos(-1.0);

typedef complex<long double> point;
typedef complex<long double> vect;

struct line {
public:
	long double a; //coeficiente x
	long double b; //coeficiente y
	long double c; //termino independiente

	line(long double a, long double b, long double c) {
		this->a=a;
		this->b=b;
		this->c=c;
	}
};

typedef pair<point, point> segment;

typedef struct {
	point center;
	long double radius;
} circle;

bool menorX(point p1, point p2) {
	if(p1.real()<p2.real()) return true;
	else if (p1.imag()<p2.imag()) return true;
	return false;
}

bool menorY(point p1, point p2) {
	if (p1.imag()<p2.imag()) return true;
	else if(p1.real()<p2.real()) return true;
	return false;
}

vect inline perp(vect v) {
	return vect(v.imag(),-v.real());
}

//calcula el doble del area CON SIGNO tomando los puntos en sentido antihorario
//los hermanos halim lo llaman cross(p,q,r)
//si c esta a la izda de ab, vale positivo, si esta a la derecha, negativo
//si estan alinados valdra (casi) 0
long double darea(point a, point b, point c) {
	/*
	| a.x a.y 1 |
	| b.x b.y 1 |
	| c.x c.y 1 |
	*/
	return
		a.real()*b.imag() +
		b.real()*c.imag() +
		c.real()*a.imag() -
		a.imag()*b.real() -
		b.imag()*c.real() -
		c.imag()*a.real();
}

//devuelve el angulo bac, es decir, centrado en a. Siempre es positivo
long double angle(point a, point b, point c) {
	point u = b-a;
	point v = c-a;
	return acos((u.real()*v.real() + u.imag()*v.imag()) /abs(u*v));
}

//para que funcione, no deben estar alineados
```

```
point circumcenter(point a, point b, point c) {
	point b2=b-a, c2=c-a;

	long double d=2*(b2.real()*c2.imag()-b2.imag()*c2.real());
	point u2((c2.imag()*norm(b2)-b2.imag()*norm(c2))/d, (b2.real()*norm(c2)-c2.real()*norm(b2))/d);
	return u2+a;
}

//para que funcione, no deben estar alineados
point incenter(point a, point b, point c) {
	long double la=abs(b-c), lb=abs(c-a), lc=abs(a-b);
	long double p=la+lb+lc;
	point result( (la*a.real() + lb*b.real() + lc*c.real()) /p,(la*a.imag()+lb*b.imag()+lc*c.imag())/p);
	return result;
}

//a!=b
line pointsToLine(point a, point b) {
	line result;

	result.a=b.imag()-a.imag();
	result.b=a.real()-b.real();
	result.c=a.imag()*b.real()-a.real()*b.imag();
	return result;
}

line pointSlopeToLine(point p, long double m)
{
	line l;
	l.a=-m;
	l.b=1;
	l.c= -l.a*p.real() -l.b*p.imag();
	return l;
}

line pointVectorToLine(point p, vect v) {
	return pointsToLine(p,p+v);
}

//devuelve un vector unitario
vect direction(line l) {
	vect result=vect(l.b,-l.a);
	result/=abs(result);

	return result;
}

//prec:no deben ser paralelas ni iguales.
point intersection(line r, line s) {
	long double den=r.b*s.a-r.a*s.b;
	point result( (r.c*s.b-r.b*s.c)/den , (r.a*s.c-r.c*s.a)/den );
	return result;
}

//estos dos metodos hacen paralelas y perpendiculares por un punto
line parallel(line l, point p) {
	line result;
	result.a=l.a;
	result.b=l.b;
	result.c= -l.a*p.real() - l.b*p.imag();
	return result;
}

line perpendicular(line l, point p) {
	line result;
	result.a=-l.b;
	result.b=l.a;
	result.c= l.b*p.real()-l.a*p.imag();
	return result;
}

//devuelve las paralelas a distancia d
pair<line,line> parallel(line l, long double d) {
	line l1, l2;

	l1.a=l2.a=l.a;
	l1.b=l2.b=l.b;
	l1.c=l.c-d*hypot(l.a,l.b);
	l2.c=l.c+d*hypot(l.a,l.b);
```

```
	return pair<line, line>(l1,l2);
}

//devuelve el coseno del MENOR angulo de ambas rectas.
long double cosAngle(line l1, line l2) {
	return abs(l1.a*l2.a+l1.b*l2.b)/(hypot(l1.a,l1.b)*hypot(l2.a,l2.b));
}

point closestPoint(line l, point p) {
	long double d=l.a*l.a+l.b*l.b;
	point result( (l.b*l.b*p.real()-l.a*l.c-l.a*l.b*p.imag())/d , (l.a*l.a*p.imag()-l.b*l.c-l.a*l.b*p.real())/d );
	return result;
}

long double dist(point p, line l) {
	return abs(l.a*p.real()+l.b*p.imag()+l.c)/hypot(l.a,l.b);
}

//prec: se cumple que se cortan y NO son tangentes
pair<point,point> intersection(circle c, line l) {
	point aux=closestPoint(l, c.center);
	long double d=sqrt(c.radius*c.radius-norm(aux-c.center));
	vect v=direction(l)*d;

	return pair<point, point>(aux+v,aux-v);
}

//prec: se cortan y NO son tangentes  Revisese
pair<point, point> intersection(circle c1, circle c2) {
	long double d= (2*c1.radius*c1.radius -c2.radius*c2.radius)/(2*c1.radius);
	long double h= sqrt(c1.radius*c1.radius-d*d);

	vect v=c2.center-c1.center;
	v/=abs(v);
	point p=c1.center+v*d;
	vect u=vect(v.imag(),-v.real());
	u*=h;

	return pair<point, point>(p+u,p-u);
}

//devuelve los puntos de tangencia
//prec: que haya puntos de tangencia
pair<point, point> tangente(point p, circle c) {
	circle aux;
	aux.center=(p+c.center)/2.0;
	aux.radius=abs(p-c.center)/2.0;

	return intersection(aux,c);
}

//en adelante, representare los poligonos como vectores de puntos.
//calcula el perimetro de un poligono
long double perimeter(vector<point> P) {
	long double result = 0.0;
	for (int i = 0; i < (int)P.size(); i++)
		result += abs(P[i]- P[(i + 1) % P.size()]);
	return result;
}

//area de un poligono
double area(vector<point> P) {
	long double result = 0.0, x1, y1, x2, y2;
	for (int i = 0; i < (int)P.size(); i++)
	{
		x1 = P[i].real(); x2 = P[(i + 1) % P.size()].real();
		y1 = P[i].imag(); y2 = P[(i + 1) % P.size()].imag();
		result += (x1 * y2 - x2 * y1);
	}
	return abs(result) / 2.0;
}

//devuelve true si p pertenece a P
bool inPolygon(point p, vector<point> P) {
	if ((int)P.size() == 0) return false;
	double sum = 0;
	for (int i = 0; i < (int)P.size() - 1; i++)
	{
```

```
                if (darea(p, P[i], P[(i + 1)%P.size()]) < 0) //si es negativo, a la derecha
                        sum -= angle(p, P[i], P[(i + 1)%P.size()]);
                else sum += angle(p, P[i], P[(i + 1)%P.size()]);
        }
        return (abs(sum - 2*pi) < eps || abs(sum + 2*pi) < eps);
}

//hace mas facil la funcion cutPolygon
point intersectSeg(point p, point q, point A, point B) {
        long double a = B.imag() - A.imag();
        long double b = A.real() - B.real();
        long double c = B.real() * A.imag() - A.real() * B.imag();
        long double u = abs(a * p.x + b * p.y + c);
        long double v = abs(a * q.x + b * q.y + c);
        return point((p.x * v + q.x * u) / (u + v),
                (p.y * v + q.y * u) / (u + v));
}

//corta el poligono de modo que devuelve el poligono de los puntos a la izda de ab.
//para obtener el otro lado, invertir ab
//Todavia no comprobado, no estoy del todo seguro que funcione
vector<point> cutPolygon(point a, point b, vector<point> Q) {
        vector<point> P;
        for (int i = 0; i < (int)Q.size(); i++)
        {
                long double left1 = darea(a, b, Q[i]);
                long double left2 = darea(a, b, Q[(i + 1)%Q.size()]);
                if (left1 > -eps) P.push_back(Q[i]);
                if (left1*left2 < -eps)
                        P.push_back(intersectSeg(Q[i], Q[(i + 1)%Q.size()],a,b));
        }
        if (P.empty()) return P;
        return P;
}


//envolvente convexa, algoritmo de Graham. Devuelve un vector de point ordenado.
//Casi copiado y pegado del libro de los hermanos halim
//Que va, esta completamente copiado
point pivot(0, 0);
bool angle_cmp(point a, point b) // angle-sorting function {
        if (abs(darea(pivot, a, b))<eps)
                return abs(pivot-a) < abs(pivot- b); // which one is closer?
        point d1 = a - pivot;
        point d2 = b - pivot;
        return (arg(d1) - arg(d2)) < 0;
}

vector<point> convexHull(vector<point> P) {
  int i, N = (int)P.size();
  if (N <= 3) return P; // special case, the CH is P itself

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < N; i++)
    if (P[i].imag()  < P[P0].imag() ||
       (P[i]imag() == P[P0].imag() && P[i].real() > P[P0].real()))
      P0 = i;
  // swap selected vertex with P[0]
  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;

  // second, sort points by angle w.r.t. P0, skipping P[0]
  pivot = P[0]; // use this global variable as reference
  sort(1+P.begin(), P.end(), angle_cmp);

  // third, the ccw tests
  point prev(0, 0), now(0, 0);
  stack<point> S; S.push(P[N - 1]); S.push(P[0]); // initial
  i = 1; // and start checking the rest
  while (i < N) { // note: N must be >= 3 for this method to work
    now = S.top();
    S.pop(); prev = S.top(); S.push(now); // get 2nd from top
    if (darea(prev, now, P[i])>0) S.push(P[i++]); // left turn, ACC
    else S.pop(); // otherwise, pop until we have a left turn
  }

  vector<point> ConvexHull; // from stack back to vector
  while (!S.empty()) { ConvexHull.push_back(S.top()); S.pop(); }
  return ConvexHull; } // return the result
```

NÚMEROS GRANDES

```
#include <cmath> //log10,pow,floor
#include <cstring> //memset
#include <string> //string class
#include <sstream> //stringstream class
#include <algorithm> //max and min
using namespace std;

const int NUM=25; //number of elementes in array

typedef long long int base_t;
const base_t BASE=pow(10,floor(log10(2)*4*sizeof(base_t)));
const int zeros=log10(BASE);

struct longint{
   longint(long long int i=0) :used(0),sign((i>=0)?1:-1){
            memset(inner,0,NUM*sizeof(base_t));
            i*=sign;
            for(;i>=BASE;i=i/BASE)
               inner[used++]=i%BASE;
            inner[used++]=i;
   }

   longint operator+(longint b) const{
            if(sign!=b.sign) return b.sign*=-1,((sign<0)?*this-b: b-*this);
            b.used=min(max(b.used,used)+1 , NUM); base_t carry=0;
            for(int i=0;i<b.used;i++){
               carry=(b.inner[i]+=inner[i]+carry)/BASE;
               b.inner[i]%=BASE;
            }
            while(b.used && !b.inner[b.used-1]) --b.used;
            return b;
   }

   longint operator-(longint b) const{
            longint const *m=this,*M=&b;
            if(used>b.used || used==b.used && inner[used-1]>b.inner[used-1])
               M=this,m=&b;
            if(b.sign!=sign) return b.sign*=-1,((*M)+(*m));
            longint res(*M); base_t carry=0;
            for(int i=0;i<=min(m->used,NUM-1);i++)
               carry=(res.inner[i]-=(m->inner[i]+carry))<0?res.inner[i]+=BASE,1:0;
            while(res.used && !res.inner[res.used-1]) --res.used;
            return res;
   }

   longint operator*(const longint& b) const{
            longint res;
            longint const *m, *M= (used>b.used)? (m=&b,this) : (m=this,&b);
            for(int i=0;i<m->used; i++)
               for(base_t j=0,carry=0 ; j<=(min(M->used,NUM-i-1)) ; j++){
                        carry=(res.inner[j+i]=(M->inner[j]*(m->inner[i]))+carry)/BASE;
                        res.inner[j+i]%=BASE;
               }
            res.used=min(M->used+m->used,NUM);
            while(res.used && !res.inner[res.used-1]) --res.used;
            res.sign=sign*b.sign;
            return res;
   }

   string str() const{
            stringstream ss;
            ss << sign*inner[used-1];
            for(int i=used-2;i>=0;i--){
               int cfr= inner[i] ? floor(log10(inner[i]))+1 : 1;
               for(int j=cfr;j<zeros;j++) ss << 0;
               ss << inner[i];
            }
            return ss.str();
   }

   char sign; //sign bit
   int used; //number of base_t in used
   base_t inner[NUM]; //base array
};

longint operator+(long long int i, const longint& l){ return l+i; }
longint operator*(long long int i, const longint& l){ return l*i; }
longint operator-(long long int i, const longint& l){ return longint(i)-l;}
```

GRAFOLETARIO

```
#include <queue>

#define MAXV 100;  //Máximo de vértices
#define MAXDEGREE 50; //Grado máximo de un veŕtice


/*Implementación de un grafo como matriz de adyacencia
* bool graph[MAXV][MAXV];
* donde graph[i][j]==TRUE <==> hay un arco conectando el nodo i con el nodo j
*/
//Implementación de un grafo como lista de adyacencias como matriz.
//También se pueden guardar las adyacencias en una lista,
//ocupando menos memoria.
struct Graph {
        int edges[MAXV+1][MAXDEGREE];  //Nodos adyacentes a cada nodo
        int degree[MAXV+1]; //Contiene el grado de cada nodo
        int nvertices; //Total de vertices
        int nedges; //Total de aristas

        Graph(){
                nvertices=0;
                nedges=0;
                for(int i=0; i<=MAXV; i++){
                        degree[i]=0;
        }

        insertEdge(int x, int y, bool directed){
                if (degree[x] > MAXDEGREE); //Error
                edges[x][degree[x]] = y;
                degree[x]++;
                if (!directed) insertEdge(x,y,TRUE);
                else nedges++;
        }

        //Recorrido en anchura. La función deberá modificarse para diversos
        //propósitos, por ejemplo añadiendo un elemento a buscar o usando
        //una cola de parejas <int, int> para guardar la profundidad de los
        //nodos. También puede ser útil usar una matriz parent[MAXV] en la
        //que se guarde para cada nodo i su padre en parent[i].
        void bfs(int start){  // bfs(int start,int end)
                queue <int> q; //queue <Pair<int,int> q;
                q.push(start);
                bool visited[MAXV];
                for (int i=0; i<nvertices; i++) visited[i]=FALSE;

                while(!q.empty()){
                        int v=q.front(); q.pop();
                        visited[v]=TRUE
                        //Procesar v. Por ejemplo: if(v==end) ...
                        //Guardar los sucesores si no se han visitado
                        for (int i=0; i<degree[v]; i++){
                                if (!visited[edges[v][i]]){
                                        q.push(edges[v][i]);
                                        //parent[edges[v][i]]=v;
                                }
                        }
                }
        }


        //Recorrido en profundidad. También debe modificarse segun el problema.
        //Implementación recursiva (se puede hacer no recursiva con una pila,
        // y ocuparía menos memoria)
        void dfs(int v){
                //if (finished) return
                bool visited[MAXV];
                for (int i=0; i<nvertices; i++) visited[i]=FALSE;
                visited[v]=TRUE;
                //Procesar vértice

                for(int i=0; i<degree[v]; i++){
                        int y= edges[v][i];
                        if (!visited[y]){
                                dfs(y);
                        }
                        //if (finished) return
                }
        }
```

```
// Encontrar las componentes conexas
void connected_components(){
	bool visited[MAXV];
	for (int i=0; i<nvertices; i++) visited[i]=FALSE;
	int c=0; //Número de componentes
	for(int i=0; i<nvertices; i++){
		if(!discovered[i]){
			c++;
			dfs(i);
			//dfs necesita visitados, se puede pasar como
			//argumento o declarar como variable global

		}
	}
}

//Ordenación topológica. Dado un grafo dirigido sin ciclos,
//consiste en dar un orden de forma que no haya ningún arco
//de un elemento a otro anterior en ese orden.
void topsort(int sorted[]){
	int indegree[MAXV] //Grado de entrada
	queue <int> zeroin //Vértices con grado de entrada 0

	for (int i=0; i<nvertices; i++) indegree[i]=0;
	for (int i=0; i<nvertices; i++)
		for (int j=0; j<degree[i];j++)
			in[edges[i][j]] ++;

	for (int i=0; i<nvertices; i++)
		if(indegree[i]==0) zeroin.push(i);

	int j=0;
	while (!zeroin.empty()){
		j++;
		int x=zeroin.front();
		zeroin.pop();
		sorted[j]=x;
		for(int i=0; i<degree[x]; i++){
			int y=edges[x][i];
			indegree[y] --;
			if (indegree[y]==0) zeroin.push(y);
		}
	}

	if (j!=nvertices); //No es un grafo dirigido acíclico
}


/*Algoritmo de Prim. Hay que modificar la declaración del struct:
	struct Edge{
		int v;
		int weight;
	};

	struct Graph{
		Edge edges[MAXV+1][MAXDEGREE];
		...
	}
También se modifican el resto de métodos consecuentemente.
*/

void prim(int start){
	bool intree[MAXV];  //Vertices en el árbol
	int distance[MAXV]; //Distancia mínima de cada vértice de fuera del
				//árbol a un vértice del árbol
	int parent[MAXV]; //Topología del árbol

	for(int i=0; i<nvertices; i++){
		intree[i] = FALSE;
		distance[i] = MAXINT; // (1>>32)
		parent[i] = -1;
	}

	distance[start]=0;
	int v=start;

	while(!intree[v]){
```

```
                        intree[v] = TRUE;

                        //Actualiza para cada nodo la distancia mínima
                        for (int i=0; i<degree[v]; i++){
                                int w = edges[v][i].v;
                                int weight = edges[v][i].weight;
                                if ((distance[w] > weight) && (!intree[w]){
                                        distance[w] = weight;
                                        parent[w] = v;
                                }
                        }

                        v=1;
                        int dist = MAXINT;

                        //Busca el nodo externo más cercano al árbol
                        for (int i=0; i<nvertices; i++){
                                if ((!intree[i])&&(dist>distance[i])){
                                        dist = distance[i];
                                        v = i;
                                }
                        }
                }
}

//Algoritmo de Dijkstra. Basta con copiar Prim y cambiar dos líneas
//En distance[i] se guarda la distancia al nodo i, y para recuperar
//el camino se usa parent
void dijkstra(int start){
        bool intree[MAXV];  //Vertices en el árbol
        int distance[MAXV]; //Distancia de cada vértice al origen
        int parent[MAXV];

        for(int i=0; i<nvertices; i++){
                intree[i] = FALSE;
                distance[i] = MAXINT; // (1>>32)
                parent[i] = -1;
        }

        distance[start]=0;
        int v=start;

        while(!intree[v]){
                intree[v] = TRUE;

                //Para cada vértice mira si es más corto pasar por v o no
                for (int i=0; i<degree[v]; i++){
                        int w = edges[v][i].v;
                        int weight = edges[v][i].weight;
/*Cambiar*/                       if ((distance[w] > (distance[v] + weight)){
/*Cambiar*/                               distance[w] = distance[v] + weight;
                                        parent[w] = v;
                        }
                }

                v=1;
                int dist = MAXINT;

                //Busca el nodo externo más cercano al árbol
                for (int i=0; i<nvertices; i++){
                        if ((!intree[i])&&(dist>distance[i])){
                                dist = distance[i];
                                v = i;
                        }
                }
        }
}

/*Algoritmo de Floyd. Se usa una representación de un grafo como
  matriz de adyacencia con peso de las aristas:
        struct AdjacencyMatrix{
                int weight[MAXV+1][MAXV+1];
                int nvertices;
        }
La matriz se inicializa con 0 en la diagonal, los valores correspondientes
a las aristas que nos dan como datos y MAXINT en el resto*/

void floyd(){
        for(int k=0; k<nvertices; k++){
```

```
                    for(int i=0; i<nvertices; i++){
                        for (int j=0; j<nvertices; j++){
                            int throughK = weight[i][k]+weight[k][j];
                            if (throughK < weight[i][j])
                                weight[i][j] = throughK;
                        }
                    }
                }
            }

}
```

MISC-LETARIO

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

typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;

//mascaras de bits
#define isOn(S, j) (S & (1 << j))
#define setBit(S, j) (S |= (1 << j))
#define clearBit(S, j) (S &= ~(1 << j))
#define toggleBit(S, j) (S ^= (1 << j))
#define lowBit(S) (S & (-S))
#define setAll(S, n) (S = (1 << n) - 1)

//Arbol de segmentos. Permite buscar el minimo elemento de un array LEERLO
// Segment Tree Library: The segment tree is stored like a heap array
void st_build(vi &st, const vi &A, int vertex, int L, int R) {
  if (L == R)                       // as L == R, either one is fine
    st[vertex] = L;                 // store the index
  else {   // recursively compute the values in the left and right subtrees
    int nL = 2 * vertex, nR = 2 * vertex + 1;
    st_build(st, A, nL, L             , (L + R) / 2);
    st_build(st, A, nR, (L + R) / 2 + 1, R          );
    int lContent = st[nL]     , rContent = st[nR];
    int lValue   = A[lContent], rValue   = A[rContent];
    st[vertex] = (lValue <= rValue) ? lContent : rContent;
} }

void st_create(vi &st, const vi &A) {       // if original array size is N,
     // the required segment tree array length is 2*2^(floor(log2(N)) + 1);
  int len = (int)(2*pow(2.0, floor((log((double)A.size())/log(2.0)) + 1)));
  st.assign(len, 0); // create vector of size `len' and fill it with zeroes
  st_build(st, A, 1, 0, (int)A.size() - 1);             // recursive build
}

int st_rmq(vi &st, const vi &A, int vertex, int L, int R, int i, int j) {
  if (i >  R || j <  L) return -1;   // current segment outside query range
  if (L >= i && R <= j) return st[vertex];           // inside query range

    // compute the min position in the left and right part of the interval
  int p1 = st_rmq(st, A, 2 * vertex    , L             , (L+R) / 2, i, j);
  int p2 = st_rmq(st, A, 2 * vertex + 1, (L+R) / 2 + 1, R          , i, j);

                    // return the position where the overall minimum is
  if (p1 == -1) return p2;     // if we try to access segment outside query
  if (p2 == -1) return p1;                          // same as above
  return (A[p1] <= A[p2]) ? p1 : p2; }

int st_rmq(vi &st, const vi& A, int i, int j) {     // function overloading
  return st_rmq(st, A, 1, 0, (int)A.size() - 1, i, j); }

int st_update_point(vi &st, vi &A, int node, int b, int e, int idx, int new_value) {
  // this update code is still preliminary, i == j
  // must be able to update range in the future!
  int i = idx, j = idx;

  // if the current interval does not intersect
  // the update interval, return this st node value!
  if (i > e || j < b)
    return st[node];
```

```
  // if the current interval is included in the update range,
  // update that st[node]
  if (b == i && e == j) {
    A[i] = new_value; // update the underlying array
    return st[node] = b; // this index
  }

  // compute the minimum position in the
  // left and right part of the interval
  int p1, p2;
  p1 = st_update_point(st, A, 2 * node    , b             , (b + e) / 2, idx, new_value);
  p2 = st_update_point(st, A, 2 * node + 1, (b + e) / 2 + 1, e          , idx, new_value);

  // return the position where the overall minimum is
  return st[node] = (A[p1] <= A[p2]) ? p1 : p2;
}

int st_update_point(vi &st, vi &A, int idx, int new_value) {
  return st_update_point(st, A, 1, 0, (int)A.size() - 1, idx, new_value); }

//Arbol de Fenwick. Permite resolver el problema de la suma en un rango
//con actualizaciones. (RSQ(a,b)=suma de los a[i], i entre a y b)
// initialization: n + 1 zeroes, ignoring index 0, just using index [1..n]
void ft_create(vi &ft, int n) { ft.assign(n + 1, 0); }

int ft_rsq(const vi &ft, int b) {                    // returns RSQ(1, b)
  int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
  return sum; }

int ft_rsq(const vi &ft, int a, int b) {             // returns RSQ(a, b)
  return ft_rsq(ft, b) - (a == 1 ? 0 : ft_rsq(ft, a - 1)); }

// adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
void ft_adjust(vi &ft, int k, int v) {          // note: n = ft.size() - 1
  for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }


//Algoritmo de Euclides extendidio
//ax+by=g=gcd(a,b)
void eea (int a, int b,
        int& gcd, int& x, int& y) {
        x=0, y=1;
        int u=1, v=0, m, n, q, r;
        gcd = b;
        while (a!=0) {
                q=gcd/a; r=gcd%a;
                m=x-u*q; n=y-v*q;
                gcd=a; a=r; x=u; y=v; u=m; v=n;
        }
}


//ecuacion diofantica
//con d=mcd(A,B)
x=x0 + L*(B/d)
y=y0 – L*(A/d)


//Criba de Eratostenes
#include <iostream>


int main(void)
{
    const int NMAX = 1000;

    //Armamos la "lista"
    bool tachado[NMAX];
    for (int i = 0; i < NMAX; ++i)
        tachado[i] = false; //empezamos con todos sin tachar
    tachado[0] = true; //0 no es primo, lo tacho
    tachado[1] = true; //1 no es primo, lo tacho

    //y ahora empezamos a tachar
    for (int p = 0; p < NMAX; ++p) {
        if (!tachado[p]) {
            //p no esta tachado, asi que es primo
            std::cout << p << ' ';
```

```
            //tachamos todos los múltiplos que estén en la "lista" o tabla
            for (int j = 2*p; j < NMAX; j += p)
                tachado[j] = true;
        }
    }
    std::cout << std::endl;

    return 0;
}
```

Funcion Backtracking (Etapai) devuelve: boolean

```
Inicio
        Éxito = falso;
        IniciarOpciones(i, GrupoOpciones o);
        Repetir
                SeleccionarnuevaOpcion(o, Opcion n);
                Si (Aceptable(n)) entonces
                        AnotarOpcion(i, n);
                        SiSolucionCompleta(i) entonces
                                Éxito = verdadero;
                        Sino
                                Éxito = Backtracking(i+1);
                                Si Éxito = false entonces
                                        cancelamosAnotacion(i, n);
                                finsi;
                        Finsi;
                Finsi;
        Hasta (éxito = verdadero) o (NoQuedanOpciones(o));
        Retorna Éxito;
Fin;
```