Material de referencia para la ICPC.

Índice

NOTA: En la mayoría de los casos, es necesario reiniciar las variables para poder reutilizar los algoritmos.

1. Estructuras de datos.

1.1. Policy Based Data Structures.

La STL de GNU C++ implementa algunas estructuras de datos adicionales. Probablemente la más interesante de todas es el árbol. Para utilizarlo debemos añadir antes las siguientes librerías:

```
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
```

Los contenedores basados en árboles tienen la siguiente declaración:

```
tree<Key, Mapped, Cmp_Fn = std::less<Key>, Tag = rb_tree_tag, node_update = null_node_update
    , Allocator = std::allocator<char> >
```

donde

- `Key` es el tipo de las llaves.
- `Mapped` es el tipo de los datos mapeados. Esto se asemeja bastante a un `map`. Si en cambio lo llenamos con `null_type`, obtenemos un contenedor similar a un `set`.
- `Cmp_Fn` es una función de comparación de llaves. Debe declararse en forma de `struct` con el operador `()` sobrecargado.
- `Tag` especifica la estructura de datos a utilizar. Debe ser alguno de `rb_tree_tag` (red-black tree), `splay_tree_tag` (splay tree) o `ov_tree_tag` (ordered-vector tree).
- `node_update` especifica como actualizar los invariantes de cada nodo. Por defecto, `null_node_update` indica que los nodos no guardan información adicional.

Split y join

Los contenedores basados en árboles soportan las mismas funciones que `set` y `map`, junto con dos funciones nuevas:

```
A.split(T key, Tree B);
A.join(Tree B);
```

La función `split` mueve todos los nodos con llaves mayores que `key` del árbol `A` al árbol `B`. La función `join`, por el contrario, mueve todos los nodos del árbol `B` al árbol `A`, siempre y cuando los rangos no se traslapen. En el caso de árboles rojo-negro, ambas funciones tienen complejidad poli-logarítmica.

Iteradores de nodo

Además de los iteradores convencionales de `set` y `map`, los contenedores basados en árboles implementan un tipo de iterador adicional, `node_iterator`, el cual nos permite recorrer el árbol. Así por ejemplo, las funciones

```
Tree::node_iterator root = A.node_begin();
Tree::node_iterator nil = A.node_end();
```

regresan un iterador de nodo correspondiente a la raíz y nodos nulos del árbol. Cada iterador de nodo incluye dos funciones miembro `get_l_child()` y `get_r_child()` que regresan los iteradores de nodos correspondientes a los hijos izquierdo y derecho.

Podemos hacer la conversión entre iteradores convencionales e iteradores de nodo de la siguiente manera:

```
it = *nd_it;
nd_it = it.m_p_nd;
```

La primera línea regresa el `iterator` correspondiente a un `node_iterator` mientras que la segunda realiza lo contrario.

Actualización de nodos

Recordemos que `node_update` especifica la información adicional que guardará cada nodo así como la forma en que se actualiza. Este debe ser declarado en forma de `struct`, el cual debe definir en su interior el tipo del dato adicional como `metadata_type`, y sobrecargar el operador `()` especificando cómo se actualizará cada nodo.

El operador `()` será llamado internamente cada vez que sea necesario, recibiendo como parámetros el nodo a actualizar y el nodo nulo. Las llamadas siempre se realizarán desde las hojas hasta la raíz. De esta manera, al actualizar la información de un nodo, la información de sus hijos ya está actualizada.

Cada iterador de nodo tiene una función miembro `get_metadata()` que regresa una referencia al dato adicional de ese nodo. Sin embargo, al ser una variable constante, debemos hacer antes un `const_cast<metadata_type &>` para modificarlo.

Por ejemplo, si queremos que cada nodo guarde el tamaño del sub-árbol correspondiente, podemos definir la etiqueta `size_node_update` de la siguiente manera:

```
template<typename node_const_iterator, typename node_iterator, typename Cmp_Fn, typename
    Allocator>
struct size_node_update {
    typedef int metadata_type;

    void operator() (node_iterator nd_it, node_const_iterator nil) {
        int lsize = 0, rsize = 0;
        if (nd_it.get_l_child() != nil)
            lsize = nd_it.get_l_child().get_metadata();
        if (nd_it.get_r_child() != nil)
            rsize = nd_it.get_r_child().get_metadata();
        const_cast<int &>(nd_it.get_metadata()) = lsize + rsize + 1;
    }
};
```

Árbol de Estadísticos de Orden

La STL incluye una etiqueta `tree_order_statistics_node_update`, que le indica a cada nodo que guarde el tamaño del sub-árbol correspondiente. Esta etiqueta incorpora dos funciones nuevas:

```
A.find_by_order(unsigned int k);
A.order_of_key(T key);
```

La función `find_by_order` regresa un iterador convencional que corresponde al k-ésimo elemento de `A` (indexado en 0). La función `order_of_key`, por su parte, regresa un entero que representa el número de elementos menores que `key`. Ambas funciones tienen complejidad logarítmica.

```
#include <iostream>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;

typedef tree<int, null_type, less<int>, rb_tree_tag, tree_order_statistics_node_update>
    ordered_set;
typedef ordered_set::node_iterator pnode;

//Imprime el arbol rotado 90 grados hacia la izquierda.
void print_tree(pnode it, pnode nil, int indent = 0) {
    if (it != nil) {
        print_tree(it.get_l_child(), nil, indent + 2);
        for (int i = 0; i < indent; ++i)
            cout << ' ';
        cout << **it << '\n';
        print_tree(it.get_r_child(), nil, indent + 2);
    }
}

int main() {
    //Datos de ejemplo.
    int n = 10;
    int arr[] = {20, 15, 50, 30, 25, 36, 10, 35, 40, 21};
    //Crea un arbol con los datos de arr y lo imprime.
    ordered_set v, w;
    for (int i = 0; i < n; ++i)
        v.insert(arr[i]);
    print_tree(v.node_begin(), v.node_end()); cout << '\n';
    //Separa el arbol en dos y los imprime.
    v.split(30, w);
    print_tree(v.node_begin(), v.node_end()); cout << '\n';
    print_tree(w.node_begin(), w.node_end()); cout << '\n';
    //Vuelve a unir ambos arboles y lo imprime.
    v.join(w);
    print_tree(v.node_begin(), v.node_end()); cout << '\n';
    //Imprime el indice de 35.
    cout << v.order_of_key(35) << '\n';
```

```
39      //Imprime el 7-esimo elemento.
40      cout << *v.find_by_order(7) << '\n';
41      return 0;
42  }
```

2. Grafos.

2.1. Caminos más cortos.

Algoritmo de Dijkstra. Complejidad: $O((E+V)\log V)$.

```
1   #include <iostream>
2   #include <algorithm>
3   #include <vector>
4   #include <queue>
5   #include <utility>
6   using namespace std;
7   #define maxv 100000 //Maximo numero de vertices.
8
9   typedef pair<int, int> edge;
10  #define length first
11  #define to     second
12
13  int V, E;                  //Numero de vertices y aristas.
14  vector<edge> graph[maxv]; //Aristas.
15
16  int s;                       //Vertice inicial.
17  int dist[maxv], pred[maxv]; //Distancia desde s y predecesor en el camino.
18  bool vis[maxv];              //Visitado.
19
20  //Encuentra el camino mas corto desde un vertice a todos los demas.
21  void Dijkstra() {
22      fill_n(dist, V, 1e9);
23      fill_n(pred, V, -1);
24      dist[s] = 0;
25      priority_queue<edge> pq;
26      pq.push(edge(dist[s], s));
27      while (!pq.empty()) {
28          int curr = pq.top().to;
29          pq.pop();
30          vis[curr] = true;
31          for (edge e : graph[curr])
32              if (!vis[e.to] && dist[curr] + e.length < dist[e.to]) {
33                  dist[e.to] = dist[curr] + e.length;
34                  pred[e.to] = curr;
35                  pq.push(edge(-dist[e.to], e.to));
36              }
37      }
38  }
39
40  int main() {
41      ios_base::sync_with_stdio(0); cin.tie();
42      cin >> V >> E >> s;
43      //Lee la informacion de las aristas.
44      for (int i = 0; i < E; ++i) {
45          int u, v, d;
46          cin >> u >> v >> d;
47          graph[u].push_back(edge(d, v));
48          graph[v].push_back(edge(d, u));
49      }
50      //Imprime la configuracion.
51      Dijkstra();
52      for (int i = 0; i < V; ++i)
53          cout << i << ": " << pred[i] << ' ' << dist[i] << '\n';
54      return 0;
55  }
```

Entrada	Salida
6 7 0 0 1 4 1 3 10 3 5 11 1 2 5 2 0 2 2 4 3 4 3 4	0: -1 0 1: 0 4 2: 0 2 3: 4 9 4: 2 5 5: 3 20

2.2. Árbol de expansión mínima.

Algoritmo de Kruskal. Complejidad: $O(E \log V)$.

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>
using namespace std;
#define maxv 100000 //Maximo numero de vertices y aristas.

typedef pair<int, pair<int, int>> edge;
#define weight first
#define from   second.first
#define to     second.second

int V, E;          //Numero de vertices y aristas.
edge graph[maxv]; //Aristas.

int parent[maxv], Rank[maxv]; //Union-Find por rango y compresion de camino.
vector<int> MST;              //Arbol de expansion minima.

int Find(int x) {
    if (parent[x] != x)
        parent[x] = Find(parent[x]);
    return parent[x];
}

void Union(int x, int y) {
    int a = Find(x), b = Find(y);
    if (Rank[a] < Rank[b])
        parent[a] = b;
    else {
        parent[b] = a;
        if (Rank[a] == Rank[b])
            Rank[a]++;
    }
}

//Encuentra el arbol de expansion minima.
int Kruskal() {
    int W = 0;
    for (int i = 0; i < V; ++i)
        parent[i] = i;
    sort(graph, graph + E);
    for (int i = 0; i < E; ++i)
        if (Find(graph[i].from) != Find(graph[i].to)) {
            Union(graph[i].from, graph[i].to);
            W += graph[i].weight;
            MST.push_back(i);
        }
    return W;
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> V >> E;
```

```
54      //Lee la informacion de las aristas.
55      for (int i = 0; i < E; ++i)
56          cin >> graph[i].from >> graph[i].to >> graph[i].weight;
57      //Imprime la configuracion del arbol de expansion minima.
58      cout << "Peso total: " << Kruskal() << '\n';
59      for (int i : MST)
60          cout << graph[i].from << ' ' << graph[i].to << ' ' << graph[i].weight << '\n';
61      return 0;
62  }
```

Entrada	Salida
6 8 0 1 2 0 3 1 3 1 9 4 1 10 3 4 3 2 0 11 2 5 20 3 5 1	Peso total: 18 0 3 1 3 5 1 0 1 2 3 4 3 2 0 11

2.3. Orden topológico.

Complejidad: $O(V + E)$.

```
1   #include <iostream>
2   #include <algorithm>
3   #include <vector>
4   using namespace std;
5   #define maxv 100000 //Maximo numero de vertices.
6
7   int V, E;                 //Numero de vertices y aristas.
8   vector<int> graph[maxv]; //Aristas.
9
10  bool cycle;           //Verifica si el grafo tiene ciclos.
11  vector<int> toposort; //Orden topologico.
12  int vis[maxv];        //Visitado.
13
14  void DFS(int u) {
15      if (vis[u] == 1)
16          cycle = true;
17      else if (!vis[u]) {
18          vis[u] = 1;
19          for (int v : graph[u])
20              DFS(v);
21          vis[u] = -1;
22          toposort.push_back(u);
23      }
24  }
25
26  //Encuentra el orden topologico.
27  void ToopologicalSort() {
28      for (int u = 0; u < V; ++u)
29          DFS(u);
30      reverse(toposort.begin(), toposort.end());
31  }
32
33  int main() {
34      ios_base::sync_with_stdio(0); cin.tie();
35      cin >> V >> E;
36      //Lee la informacion de las aristas.
37      for (int i = 0; i < E; ++i) {
38          int from, to;
39          cin >> from >> to;
40          graph[from].push_back(to);
41      }
42      //Imprime el orden topologico
43      ToopologicalSort();
44      if (cycle)
```

```
45          cout << "No es un DAG.";
46      else for (int u : toposort)
47          cout << u << ' ';
48      cout << '\n';
49      return 0;
50  }
```

Entrada	Salida
7 9 6 1 6 5 0 1 1 5 0 2 1 2 2 3 5 3 5 4	6 0 1 2 5 4 3

2.4. Componentes fuertemente conexas.

Algoritmo de Tarjan. Complejidad: $O(V + E)$.

```
1  #include <iostream>
2  #include <vector>
3  #include <stack>
4  using namespace std;
5  #define maxv 100000 //Maximo numero de vertices.
6
7  int V, E;                 //Numero de vertices y aristas.
8  vector<int> graph[maxv]; //Aristas.
9
10 vector<vector<int>> SCC;          //Componentes fuertemente conexas.
11 int idx[maxv], low[maxv], lst_id; //Indice de los vertices, menor vertice alcanzable.
12 stack<int> S;                     //Vertices pendientes.
13 bool onStack[maxv];               //Esta en la pila.
14
15 //Encuentra la componente fuertemente conexa de u.
16 void StrongConnect(int u) {
17     idx[u] = low[u] = ++lst_id;
18     S.push(u);
19     onStack[u] = true;
20     for (int v : graph[u]) {
21         if(!idx[v]) {
22             StrongConnect(v);
23             low[u] = min(low[u], low[v]);
24         }
25         else if (onStack[v])
26             low[u] = min(low[u], idx[v]);
27     }
28     if (low[u] == idx[u]) {
29         SCC.push_back(vector<int>());
30         while (S.top() != u) {
31             SCC.back().push_back(S.top());
32             S.pop();
33             onStack[S.top()] = false;
34         }
35         SCC.back().push_back(u);
36         S.pop();
37         onStack[u] = false;
38     }
39 }
40
41 //Algoritmo de Tarjan para encontrar las componentes fuertemente conexas.
42 void Tarjan() {
43     for (int u = 0; u < V; ++u)
44         if (!idx[u])
45             StrongConnect(u);
```

```
46  }
47
48  int main() {
49      ios_base::sync_with_stdio(0); cin.tie();
50      cin >> V >> E;
51      //Lee la informacion de las aristas.
52      for (int i = 0; i < E; ++i) {
53          int from, to;
54          cin >> from >> to;
55          graph[from].push_back(to);
56      }
57      //Imprime las componentes fuertemente conexas.
58      Tarjan();
59      for (int i = 0; i < SCC.size(); ++i) {
60          for (int u : SCC[i])
61              cout << u << ' ';
62          cout << '\n';
63      }
64      return 0;
65  }
```

Entrada	Salida
6 8 0 1 1 2 1 4 1 3 3 0 2 5 4 5 5 4	4 5 2 3 1 0

2.5. Puentes y puntos de articulación.

Complejidad: $O(V + E)$.

```
1   #include <iostream>
2   #include <vector>
3   #include <utility>
4   using namespace std;
5   #define maxv 100000 //Maximo numero de vertices.
6
7   int V, E;                  //Numero de vertices y aristas.
8   vector<int> graph[maxv]; //Aristas.
9
10  bool artpoint[maxv];               //Puntos de articulacion.
11  vector<pair<int, int>> bridge;     //Puentes.
12  int idx[maxv], low[maxv], lst_id; //Indice de los vertices, menor vertice alcanzable.
13
14  //Enumera los vertices con una DFS.
15  void DFS(int u, int pred) {
16      idx[u] = low[u] = ++lst_id;
17      int children = 0;
18      for (int v : graph[u]) {
19          if(!idx[v]) {
20              DFS(v, u);
21              low[u] = min(low[u], low[v]);
22              children++;
23              if ((pred == -1 && children > 1) || (pred != -1 && low[v] >= idx[u]))
24                  artpoint[u] = true;
25              if (low[v] > idx[u])
26                  bridge.push_back(make_pair(u, v));
27          }
28          else if (v != pred)
29              low[u] = min(low[u], idx[v]);
30      }
31  }
32
33  //Algoritmo de Tarjan.
```

```
34 void Tarjan() {
35     for (int u = 0; u < V; ++u)
36         if (!idx[u])
37             DFS(u, -1);
38 }
39
40 int main() {
41     ios_base::sync_with_stdio(0); cin.tie();
42     cin >> V >> E;
43     //Lee la informacion de las aristas.
44     for (int i = 0; i < E; ++i) {
45         int u, v;
46         cin >> u >> v;
47         graph[u].push_back(v);
48         graph[v].push_back(u);
49     }
50     //Imprime los puentes y puntos de articulacion.
51     Tarjan();
52     cout << "Puntos de articulacion:\n";
53     for (int i = 0; i < V; ++i)
54         if (artpoint[i])
55             cout << i << ' ';
56     cout << "\nPuentes:\n";
57     for (int i = 0; i < bridge.size(); ++i)
58         cout << bridge[i].first << ' ' << bridge[i].second << '\n';
59     return 0;
60 }
```

Entrada	Salida
7 8 0 2 0 1 1 2 1 6 2 3 3 4 4 5 3 5	Puntos de articulacion: 1 2 3 Puentes: 1 6 2 3

2.6. Flujo máximo.

Algoritmo de Dinic. Complejidad: $O(V^2E)$.

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  #include <queue>
5  using namespace std;
6  #define maxv 100000 //Maximo numero de vertices.
7
8  struct edge {
9      int to, rev;         //Destino, arista invertida.
10     int flow, capacity; //Flujo, capacidad.
11 };
12
13 int V, E;                  //Numero de vertices y aristas.
14 vector<edge> graph[maxv]; //Aristas.
15
16 int s, t;                   //Fuente y sumidero.
17 int level[maxv], ptr[maxv]; //Distancia desde s y numero de aristas visitadas.
18
19 //Verifica si se puede enviar flujo de s a t.
20 bool BFS() {
21     fill_n(level, V, -1);
22     level[s] = 0;
23     queue<int> Q;
24     Q.push(s);
25     while (!Q.empty() && level[t] == -1) {
26         int curr = Q.front();
```

```
27          Q.pop();
28          for (edge e : graph[curr])
29              if (level[e.to] == -1 && e.flow < e.capacity) {
30                  level[e.to] = level[curr] + 1;
31                  Q.push(e.to);
32              }
33      }
34      return level[t] != -1;
35  }
36
37  //Envia flujo de s a t.
38  int DFS(int curr, int flow) {
39      if (curr == t || !flow)
40          return flow;
41      for (int &i = ptr[curr]; i < graph[curr].size(); ++i) {
42          edge &e = graph[curr][i];
43          if (level[e.to] == level[curr] + 1)
44              if (int currflow = DFS(e.to, min(flow, e.capacity - e.flow))) {
45                  e.flow += currflow;
46                  graph[e.to][e.rev].flow -= currflow;
47                  return currflow;
48              }
49      }
50      return 0;
51  }
52
53  //Calcula el flujo maximo de s a t.
54  int Dinic() {
55      int flow = 0;
56      while (BFS()) {
57          fill_n(ptr, V, 0);
58          while (int currflow = DFS(s, 1e9))
59              flow += currflow;
60      }
61      return flow;
62  }
63
64  int main() {
65      ios_base::sync_with_stdio(0); cin.tie();
66      cin >> V >> E >> s >> t;
67      //Lee la informacion de las aristas.
68      for (int i = 0; i < E; ++i) {
69          int from, to, capacity;
70          cin >> from >> to >> capacity;
71          graph[from].push_back(edge{to, (int)graph[to].size(), 0, capacity});
72          graph[to].push_back(edge{from, (int)graph[from].size() - 1, 0, 0});
73      }
74      //Imprime la configuracion del flujo.
75      cout << "Flujo maximo: " << Dinic() << '\n';
76      for (int i = 0; i < V; ++i)
77          for (edge e : graph[i])
78              if (e.capacity)
79                  cout << i << ' ' << e.to << ": " << e.flow << '/' << e.capacity << '\n';
80      return 0;
81  }
```

Entrada	**Salida**
6 8 0 5 0 1 11 0 2 12 1 3 12 2 1 1 2 4 10 4 3 7 3 5 15 4 5 5	Flujo maximo: 20 0 1: 11/11 0 2: 9/12 1 3: 12/12 2 1: 1/1 2 4: 8/10 3 5: 15/15 4 3: 3/7 4 5: 5/5

2.7. Emparejamiento máximo.

Algoritmo de Hopcroft-Karp. Complejidad: $O(E\sqrt{V})$.

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;
#define maxv 100000 //Maximo numero de vertices.

int U, V, E;               //Numero de vertices en cada lado y numero de aristas.
vector<int> graph[maxv]; //Aristas que van de U a V.

int pairU[maxv], pairV[maxv], dist[maxv]; //Pares de vertices en el emparejamiento.

bool BFS() {
    queue<int> Q;
    for (int u = 1; u <= U; ++u) {
        if (!pairU[u]) {
            dist[u] = 0;
            Q.push(u);
        }
        else
            dist[u] = 1e9;
    }
    dist[0] = 1e9;
    while (!Q.empty()) {
        int u = Q.front();
        Q.pop();
        if (dist[u] < dist[0])
            for (int v : graph[u])
                if (dist[pairV[v]] == 1e9) {
                    dist[pairV[v]] = dist[u] + 1;
                    Q.push(pairV[v]);
                }
    }
    return dist[0] != 1e9;
}

bool DFS(int u) {
    if (!u)
        return true;
    for (int v : graph[u])
        if (dist[pairV[v]] == dist[u] + 1 && DFS(pairV[v])) {
            pairV[v] = u;
            pairU[u] = v;
            return true;
        }
    dist[u] = 1e9;
    return false;
}

//Busca un emparejamiento maximo.
int HopcroftKarp() {
    int size = 0;
    while (BFS())
        for (int u = 1; u <= U; ++u)
            if (!pairU[u] && DFS(u))
                size++;
    return size;
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    cin >> U >> V >> E;
    //Lee la informacion de las aristas. Los vertices estan indexados en 1.
    for (int i = 0; i < E; ++i) {
        int u, v;
        cin >> u >> v;
        graph[u].push_back(v);
    }
    //Imprime la configuracion del emparejamiento.
```

```
69      cout << "Emparejamiento: " <<  HopcroftKarp() << '\n';
70      for (int u = 1; u <= U; ++u)
71          if (pairU[u])
72              cout << u << " - " << pairU[u] << '\n';
73      return 0;
74  }
```

Entrada	Salida
5 4 8 1 1 2 1 2 3 3 2 3 3 3 4 4 3 5 3	Emparejamiento: 3 1 - 1 2 - 3 3 - 2

3. Matemáticas.

3.1. Fórmulas importantes.

Desarreglos

Un desarreglo es una permutación donde ningún elemento aparece en su posición original. El número de desarreglos está dado por

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

con $!0 = 1$ y $!1 = 0$.

Números de Catalán

Los números de Catalán cuentan: el número de expresiones con n pares de paréntesis correctamente balanceados; el número de caminos distintos sobre una cuadrícula de $n \times n$ que van de la esquina inferior izquierda a la esquina superior derecha, constan solamente de movimientos hacia arriba y hacia la derecha, y nunca cruzan la diagonal; el número de triangulaciones de un polígono convexo de $n+2$ lados; entre otras cosas. Están dados por

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-k-1} = \frac{1}{n+1}\binom{2n}{n}$$

con $C_0 = 1$.

Números de Stirling

Los números de Stirling de primer tipo cuentan el número de permutaciones de tamaño n con exactamente k ciclos disjuntos. Están dados por

$$S_1(n,k) = (n-1)S_1(n-1,k) + S_1(n-1,k-1).$$

Los números de Stirling de segundo tipo cuentan el número de particiones de un conjunto de tamaño n en k subconjuntos no vacíos. Están dados por

$$S_2(n,k) = k\,S_2(n-1,k) + S_2(n-1,k-1) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^i\binom{k}{i}(k-i)^n$$

con $S_1(n,0) = S_2(n,0) = 0$ y $S_1(n,n) = S_2(n,n) = 1$.

Números de Grundy

Un juego por turnos entre dos jugadores es *normal* si el jugador que no pueda mover pierde, y es *imparcial* si en todo momento ambos jugadores disponen del mismo conjunto de movimientos.

El juego de *Nim* es un juego normal e imparcial en donde cada jugador debe escoger una pila y eliminar al menos un objeto de esa pila. Sean $P_1, \ldots, P_n$ los tamaños de cada pila. El jugador en turno tiene estrategia ganadora si y sólo si P_1 xor $\ldots$ xor $P_n \neq 0$.

El **Teorema de Sprague-Grundy** afirma que todo juego normal e imparcial es equivalente a un juego de Nim.

3.2. Big Numbers.

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

typedef string BigInt;

//Regresa el i-esimo digito de derecha a izquierda de un numero.
unsigned int digit(const BigInt &num, unsigned int i) {
    if (i < num.size())
        return num[num.size() - 1 - i] - '0';
    return 0;
}

//Compara dos numeros y regresa: 1 si el primero es mayor; 0 si son iguales; -1 si el segundo es mayor.
int compareTo(const BigInt &a, const BigInt &b) {
    for (int i = max(a.size(), b.size()) - 1; i >= 0; --i) {
        if (digit(a, i) > digit(b, i))
            return 1;
        if (digit(b, i) > digit(a, i))
            return -1;
    }
    return 0;
}

//Regresa la suma de dos numeros.
BigInt sum(const BigInt &a, const BigInt &b) {
    BigInt ans;
    int carry = 0;
    for (int i = 0; i < max(a.size(), b.size()); ++i) {
        carry += digit(a, i) + digit(b, i);
        ans.push_back((carry % 10) + '0');
        carry /= 10;
    }
    if (carry)
        ans.push_back(carry + '0');
    reverse(ans.begin(), ans.end());
    return ans;
}

//Regresa la diferencia de dos numeros. El primer numero debe ser mayor o igual que el segundo.
BigInt substract(const BigInt &a, const BigInt &b) {
    BigInt ans;
    int carry = 0;
    for (int i = 0; i < a.size(); ++i) {
        carry += digit(a, i) - digit(b, i);
        if (carry >= 0) {
            ans.push_back(carry + '0');
            carry = 0;
        }
        else {
            ans.push_back(carry + 10 + '0');
            carry = -1;
        }
    }
    while (ans.size() > 1 && ans.back() == '0')
        ans.pop_back();
    reverse(ans.begin(), ans.end());
    return ans;
}

//Regresa el producto de dos numeros (BigInt x int).
BigInt multiply(const BigInt &a, unsigned int b) {
    if (b == 0)
        return "0";
    BigInt ans;
    int carry = 0;
    for (int i = 0; i < a.size(); ++i) {
```

```
68          carry += digit(a, i) * b;
69          ans.push_back((carry %10) + '0');
70          carry /= 10;
71      }
72      while (carry) {
73          ans.push_back((carry %10) + '0');
74          carry /= 10;
75      }
76      reverse(ans.begin(), ans.end());
77      return ans;
78  }
79
80  //Regresa el producto de dos numeros (BigInt x BigInt).
81  BigInt multiply(const BigInt &a, const BigInt &b) {
82      BigInt ans;
83      for (int i = 0; i < b.size(); ++i)
84          ans = sum(ans, multiply(a, digit(b, i)).append(i, '0'));
85      return ans;
86  }
87
88  //Regresa el cociente y el residuo de la division (BigInt / int).
89  pair<BigInt, unsigned int> divide(const BigInt &a, unsigned int b) {
90      pair<BigInt, int> ans;
91      for (int i = a.size() - 1; i >= 0; --i) {
92          ans.second = 10*ans.second + digit(a, i);
93          if (!ans.first.empty() || ans.second >= b || i == 0)
94              ans.first.push_back((ans.second / b) + '0');
95          ans.second %= b;
96      }
97      return ans;
98  }
99
100 //Regresa el cociente y el residuo de la division (BigInt / BigInt).
101 pair<BigInt, BigInt> divide(const BigInt &a, const BigInt &b) {
102     pair<BigInt, BigInt> ans;
103     BigInt table[10];
104     for (int i = 0; i < 10; ++i)
105         table[i] = multiply(b, i);
106     for (int i = a.size() - 1; i >= 0; --i) {
107         int q = 0;
108         ans.second.push_back(digit(a, i) + '0');
109         while (q < 9 && compareTo(ans.second, table[q + 1]) >= 0)
110             ++q;
111         if (!ans.first.empty() || q > 0 || i == 0)
112             ans.first.push_back(q + '0');
113         ans.second = substract(ans.second, table[q]);
114     }
115     return ans;
116 }
```

3.3. Test de Primalidad.

Algoritmo de Miller-Rabin.

```
1  #include <iostream>
2  #include <random>
3  using namespace std;
4
5  //Regresa base^expo %mod.
6  long long power(__int128 base, long long expo, long long mod) {
7      if (expo == 0)
8          return 1;
9      else if (expo %2)
10         return (base * power(base, expo - 1, mod)) %mod;
11     else {
12         __int128 p = power(base, expo / 2, mod);
13         return (p * p) %mod;
14     }
15 }
16
17 default_random_engine gen; //Generador de numeros aleatorios.
18
```

```cpp
//Regresa false si n es compuesto y true si es probablemente primo.
bool MillerTest(long long n, long long d) {
    uniform_int_distribution<long long> Rand(2, n - 2);
    __int128 x = power(Rand(gen), d, n);
    if (x == 1 || x == n - 1)
        return true;
    for (; d != n - 1; d *= 2) {
        x = (x * x) %n;
        if (x == 1)
            return false;
        if (x == n - 1)
            return true;
    }
    return false;
}

//Ejecuta el Test de Miller-Rabin varias veces.
bool isProbablePrime(long long n, int attemps) {
    if (n == 2 || n == 3)
        return true;
    if (n == 1 || n == 4)
        return false;
    long long d = n - 1;
    while (d %2 == 0)
        d /= 2;
    while (attemps--)
        if (!MillerTest(n, d))
            return false;
    return true;
}

int main() {
    ios_base::sync_with_stdio(0); cin.tie();
    long long n;
    while (cin >> n) {
        if (isProbablePrime(n, 10))
            cout << "Probablemente es primo.\n";
        else
            cout << "No es primo.\n";
    }
    return 0;
}
```

Entrada	**Salida**
1000000007	Probablemente es primo.
123456789	No es primo.
104729	Probablemente es primo.

3.4. Factorización en primos.

Complejidad: $O(\pi(\sqrt{n}))$ donde $\pi(x)$ es el número de primos menores o iguales que x.

```cpp
#include <iostream>
#include <vector>
using namespace std;
#define maxn 10000000 //Raiz cuadrada del mayor numero a factorizar.

long long n;                //Numero a factorizar.
vector<long long> primes;   //Lista de primos.
vector<long long> factors; //Lista de factores primos de n.

//Encuentra con la Criba de Eratostenes los primos menores que maxn.
void find_primes() {
    vector<bool> sieve(maxn);
    for (long long i = 2; i < maxn; ++i)
        if (!sieve[i]) {
            primes.push_back(i);
            for (long long j = i * i; j < maxn; j += i)
                sieve[j] = true;
```

```
        }
    }

    //Prueba por division.
    void prime_factor() {
        factors.clear();
        for (int i = 0; primes[i] * primes[i] <= n; ++i)
            while (n % primes[i] == 0) {
                factors.push_back(primes[i]);
                n /= primes[i];
            }
        if (n != 1)
            factors.push_back(n);
    }

    int main() {
        ios_base::sync_with_stdio(0); cin.tie();
        find_primes();
        while (cin >> n) {
            prime_factor();
            for (long long p : factors)
                cout << p << ' ';
            cout << '\n';
        }
        return 0;
    }
```

Entrada	Salida
180	2 2 3 3 5
3500	2 2 5 5 5 7
123456789	3 3 3607 3803
104729	104729

3.5. Sistemas de Ecuaciones Lineales.

Eliminación Gaussiana. Complejidad $O(n^3)$.

```
#include <iostream>
#include <cmath>
using namespace std;
#define maxn 100 //Maximo numero de ecuaciones-incognitas.

int n, m;                          //Dimensiones.
double AugMatrix[maxn][maxn]; //Matriz aumentada.

//Encuentra la forma escalonada reducida de la matriz aumentada [A | B]
//con A de n x n y B de n x m. Regresa el determinante de A.
double GaussianElimination() {
    double det = 1;
    for (int k = 0; k < n; ++k) {
        int r = k;
        for (int i = k + 1; i < n; ++i)
            if (fabs(AugMatrix[i][k]) > fabs(AugMatrix[r][k]))
                r = i;
        if (fabs(AugMatrix[r][k]) < 1e-9)
            return 0;
        if (r != k) {
            for (int j = k; j < n + m; ++j)
                swap(AugMatrix[k][j], AugMatrix[r][j]);
            det *= -1;
        }
        det *= AugMatrix[k][k];
        for (int j = n + m - 1; j >= k; --j) {
            AugMatrix[k][j] /= AugMatrix[k][k];
            for (int i = 0; i < n; ++i)
                if (i != k)
                    AugMatrix[i][j] -= AugMatrix[i][k] * AugMatrix[k][j];
        }
    }
```

```
33      return det;
34  }
35
36  int main() {
37      ios_base::sync_with_stdio(0); cin.tie();
38      cin >> n >> m;
39      for (int i = 0; i < n; ++i)
40          for (int j = 0; j < n + m; ++j)
41                  cin >> AugMatrix[i][j];
42      cout << "Determinante: " << GaussianElimination() << "\nSolucion:\n";
43      for (int i = 0; i < n; ++i) {
44          for (int j = 0; j < m; ++j)
45                  cout << AugMatrix[i][j + n] << ' ';
46          cout << '\n';
47      }
48      return 0;
49  }
```

Entrada	Salida
4 1 1 -2 2 -3 15 3 4 -1 1 -6 2 -3 2 -1 17 1 1 -3 -2 -7	Determinante: 142 Solucion: 2 -2 3 -1

3.6. Teorema Chino del Residuo.

```
1   #include <iostream>
2   using namespace std;
3   #define maxn 100000 //Maximo numero de ecuaciones.
4
5   int n;                                  //Numero de ecuaciones.
6   long long MOD, coef[maxn], mod[maxn]; //Datos de las ecuaciones.
7
8   //Algoritmo extendido de Euclides.
9   long long extendedEuclid(long long a, long long b, long long &x, long long &y) {
10      if (b == 0) {
11          x = 1;
12          y = 0;
13          return a;
14      }
15      else {
16          long long gcd = extendedEuclid(b, a %b, y, x);
17          y -= (a / b) * x;
18          return gcd;
19      }
20  }
21
22  //Teorema Chino del Residuo.
23  long long ChineseRemainder() {
24      MOD = 1;
25      for (int i = 0; i < n; ++i)
26          MOD *= mod[i];
27      long long x = 0;
28      for (int i = 0; i < n; ++i) {
29          long long N = MOD / mod[i], invN, invM;
30          extendedEuclid(N, mod[i], invN, invM);
31          x = (MOD + (x + coef[i] * N * invN) %MOD) %MOD;
32      }
33      return x;
34  }
35
36  int main() {
37      ios_base::sync_with_stdio(0); cin.tie();
38      cin >> n;
39      for (int i = 0; i < n; ++i)
```

```
40          cin >> coef[i] >> mod[i];
41      cout << "x = " << ChineseRemainder() << " (mod " << MOD << ")\n";
42      return 0;
43  }
```

Entrada	Salida
3 2 4 3 9 1 5	x = 66 (mod 180)

4. Strings

4.1. Búsqueda de patrones.

Arreglo Z. Complejidad: $O(|P| + |T|)$.

```
1  #include <iostream>
2  using namespace std;
3  #define maxn 100000 //Longitud maxima de los strings.
4
5  string text, pattern, str; //Texto, patron a buscar y string auxiliar.
6  int Z[maxn];               //Arreglo Z.
7
8  //Construye el arreglo Z de str.
9  void buildZ() {
10     int l = 0, r = 0;
11     for (int i = 1; i < str.size(); ++i) {
12         Z[i] = 0;
13         if (i <= r)
14             Z[i] = min(r - i + 1, Z[i - l]);
15         while (i + Z[i] < str.size() && str[Z[i]] == str[i + Z[i]])
16             Z[i]++;
17         if (i + Z[i] - 1 > r) {
18             l = i;
19             r = i + Z[i] - 1;
20         }
21     }
22 }
23
24 int main() {
25     ios_base::sync_with_stdio(0); cin.tie();
26     //Lee el texto y los patrones.
27     cin >> text >> pattern;
28     str = pattern + '$' + text;
29     //Imprime todas las ocurrencias.
30     buildZ();
31     for (int i = 0; i < text.size(); ++i)
32         if (Z[i + pattern.size() + 1] == pattern.size())
33             cout << "Patron encontrado en la posicion " << i << '\n';
34     return 0;
35 }
```

Entrada	Salida
AABAACAADAABAABA AABA	Patron encontrado en la posicion 0 Patron encontrado en la posicion 9 Patron encontrado en la posicion 12

Aho Corasick. Complejidad: $O(|T| + |P_1| + \ldots + |P_n| + \#Ocurrencias)$.

```
1  #include <iostream>
2  #include <queue>
3  using namespace std;
4  #define maxc 26     //Longitud del alfabeto.
5  #define maxn 100    //Maximo numero de patrones.
6  #define maxs 100000 //Maximo numero de nodos.
7
8  int n;                      //Numero de patrones.
9  string text, pattern[maxn]; //Texto y lista de patrones.
```

```
10
11 int nnodes;                  //Numero de nodos.
12 struct node {
13     node *nxt[maxc], *link; //Nodos adyacentes y mayor sufijo que es prefijo en el Trie.
14     bool isEnd[maxn];        //Es nodo terminal de algun patron.
15 } Trie[maxs];                //Nodos del Trie.
16
17 //Retorna el nodo siguiente.
18 node *nextNode(node *curr, char c) {
19     if (curr == NULL)
20         return Trie;
21     if (curr->nxt[c] == NULL)
22         return nextNode(curr->link, c);
23     return curr->nxt[c];
24 }
25
26 //Construye los links de cada nodo.
27 void buildLink() {
28     queue<node*> Q;
29     Q.push(Trie);
30     while (!Q.empty()) {
31         node *curr = Q.front();
32         Q.pop();
33         for (char c = 0; c < maxc; ++c)
34             if (curr->nxt[c] != NULL) {
35                 node *nxt = curr->nxt[c];
36                 nxt->link = nextNode(curr->link, c);
37                 for (int i = 0; i < n; ++i)
38                     if (nxt->link->isEnd[i])
39                         nxt->isEnd[i] = true;
40                 Q.push(nxt);
41             }
42     }
43 }
44
45 //Construye el Trie de patrones.
46 void buildTrie() {
47     for (int i = 0; i < n; ++i) {
48         node *curr = Trie;
49         for (int j = 0; j < pattern[i].size(); ++j) {
50             char c = pattern[i][j] - 'a';
51             if (curr->nxt[c] == NULL)
52                 curr->nxt[c] = Trie + (++nnodes);
53             curr = curr->nxt[c];
54         }
55         curr->isEnd[i] = true;
56     }
57     buildLink();
58 }
59
60 int main() {
61     ios_base::sync_with_stdio(0); cin.tie();
62     //Lee el texto y los patrones.
63     cin >> text >> n;
64     for (int i = 0; i < n; ++i)
65         cin >> pattern[i];
66     buildTrie();
67     //Imprime todas las ocurrencias.
68     node *curr = Trie;
69     for (int i = 0; i < text.size(); ++i) {
70         curr = nextNode(curr, text[i] - 'a');
71         for (int j = 0; j < n; ++j)
72             if (curr->isEnd[j])
73                 cout << pattern[j] << " aparece en la posicion " << i - pattern[j].size() +
                        1 << '\n';
74     }
75     return 0;
76 }
```

Entrada	Salida
abcdabccabbacefdabc	abc aparece en la posicion 0
4	bcd aparece en la posicion 1
abc	abcd aparece en la posicion 0
ca	abc aparece en la posicion 4
bcd	ca aparece en la posicion 7
abcd	abc aparece en la posicion 16

4.2. Arreglo de sufijos.

Complejidad: $O(|s| \log |s|)$.

```
#include <iostream>
#include <algorithm>
using namespace std;
#define maxn 100000 //Longitud maxima del string.

string word;                //String.
int n, SuffixArray[maxn]; //Arreglo de sufijos.

int rnk[maxn][2], bucket[maxn];     //Rango (SuffixArray) y Cubeta (RaxixSort).
int tempSA[maxn], tempRA[maxn][2]; //Arreglos temporales.

//Ordena de acuerdo a los rangos.
void RadixSort() {
    int M = max(n, 256);
    for (int k = 1; k >= 0; --k) {
        fill_n(bucket, M, 0);
        for (int i = 0; i < n; ++i)
            bucket[rnk[i][k]]++;
        for (int i = 1; i < M; ++i)
            bucket[i] += bucket[i - 1];
        for (int i = n - 1; i >= 0; --i) {
            int nxt_id = --bucket[rnk[i][k]];
            tempSA[nxt_id] = SuffixArray[i];
            tempRA[nxt_id][0] = rnk[i][0];
            tempRA[nxt_id][1] = rnk[i][1];
        }
        for (int i = 0; i < n; ++i) {
            SuffixArray[i] = tempSA[i];
            rnk[i][0] = tempRA[i][0];
            rnk[i][1] = tempRA[i][1];
        }
    }
}

//Construye el arreglo de sufijos.
void buildSA() {
    n = word.size();
    for (int i = 0; i < n; ++i) {
        SuffixArray[i] = i;
        rnk[i][0] = word[i];
    }
    RadixSort();
    for (int k = 1; k < n; k *= 2) {
        int curr = 0, prev = rnk[0][0];
        rnk[0][0] = curr;
        tempSA[SuffixArray[0]] = 0;
        for (int i = 1; i < n; ++i) {
            if (rnk[i][0] != prev || rnk[i][1] != rnk[i - 1][1])
                curr++;
            prev = rnk[i][0];
            rnk[i][0] = curr;
            tempSA[SuffixArray[i]] = i;
        }
        for (int i = 0; i < n; ++i) {
            int nxt_id = SuffixArray[i] + k;
            rnk[i][1] = (nxt_id < n) ? rnk[tempSA[nxt_id]][0] : 0;
        }
```

```
58          RadixSort();
59      }
60  }
61
62  int main() {
63      ios_base::sync_with_stdio(0); cin.tie();
64      //Lee la palabra.
65      cin >> word;
66      buildSA();
67      //Imprime los sufijos en orden lexicografico.
68      for (int i = 0; i < n; ++i) {
69          cout << SuffixArray[i] << ' ';
70          for (int j = SuffixArray[i]; j < n; ++j)
71              cout << word[j];
72          cout << '\n';
73      }
74      return 0;
75  }
```

Entrada	Salida
banana	5 a 3 ana 1 anana 0 banana 4 na 2 nana

5. Geometría

5.1. Geometría 2D

```
1  #include <cmath>
2  #include <vector>
3  using namespace std;
4  #define epsilon 1e-9 //Precision.
5
6  struct point {
7      double x, y;
8
9      point(double Px, double Py) : x(Px), y(Py) {}
10
11     //Regresa la suma de dos vectores.
12     point operator + (const point &P) const {
13         return point(x + P.x, y + P.y);
14     }
15
16     //Regresa la resta de dos vectores.
17     point operator - (const point &P) const {
18         return point(x - P.x, y - P.y);
19     }
20
21     //Regresa el producto por un escalar.
22     point operator * (double lambda) const {
23         return point(x * lambda, y * lambda);
24     }
25
26     //Regresa el cociente entre un escalar.
27     point operator / (double lambda) const {
28         return point(x / lambda, y / lambda);
29     }
30 };
31
32 //Regresa el producto punto de dos vectores.
33 double dotProduct(const point &P, const point &Q) {
34     return P.x * Q.x + P.y * Q.y;
35 }
36
37 //Regresa la componente en z del producto cruz de dos vectores.
```

```
38  double crossProduct(const point &P, const point &Q) {
39      return P.x * Q.y - P.y * Q.x;
40  }
41
42  //Regresa la norma de un vector.
43  double norm(const point &P) {
44      return sqrt(dotProduct(P, P));
45  }
46
47  //Regresa la distancia euclidiana entre dos puntos.
48  double dist(const point &P, const point &Q) {
49      return norm(P - Q);
50  }
51
52  //Regresa el angulo en radianes entre dos vectores. Para verificar si son ortogonales
53  //(o paralelos), es mas rapido comprobar si su producto punto (resp. cruz) es cero.
54  double angle(const point &P, const point &Q) {
55      return acos(dotProduct(P/norm(P), Q/norm(Q)));
56  }
57
58  //Regresa el vector rotado 90 grados en el sentido contrario de las manecillas del reloj.
59  point rotate90ccw(const point &P) {
60      return point(-P.y, P.x);
61  }
62
63  //Regresa el vector rotado theta radianes en el sentido contrario de las manecillas
64  //del reloj.
65  point rotateCCW(const point &P, double theta) {
66      return point(P.x * cos(theta) - P.y * sin(theta), P.x * sin(theta) + P.y * cos(theta));
67  }
68
69  //Regresa la proyeccion ortogonal del vector P sobre el vector Q.
70  point projection(const point &P, const point &Q) {
71      return Q * (dotProduct(P, Q) / dotProduct(Q, Q));
72  }
73
74  //Regresa la distancia de un punto P a la recta que pasa por A y B.
75  double distPointLine(const point &P, const point &A, const point &B) {
76      return dist(P, A + projection(P - A, B - A));
77  }
78
79  //Regresa true si el segmento CD corta a la recta AB, i.e., si los puntos estan en lados
80  //opuestos de la recta.
81  bool lineSegmentIntersect(const point &A, const point &B, const point &C, const point &D) {
82      return crossProduct(B - A, C - A) * crossProduct(B - A, D - A) < 0;
83  }
84
85  //Regresa el punto de interseccion de dos rectas no paralelas AB y CD.
86  point lineLineIntersection(const point &A, const point &B, const point &C, const point &D) {
87      point v = B - A, w = D - C;
88      return A + v * (crossProduct(C - A, w) / crossProduct(v, w));
89  }
90
91  //Regresa el centro de la circunferencia que pasa por A, B y C.
92  point circumcenter(const point &A, const point &B, const point &C) {
93      point MC = (A + B) / 2, MA = (B + C) / 2;
94      return lineLineIntersection(MC, MC + rotate90ccw(A - B), MA, MA + rotate90ccw(C - B));
95  }
96
97  //Regresa las intersecciones de la recta AB con la circunferencia con centro O y radio r.
98  vector<point> lineCircleIntersection(const point &A, const point &B, const point &O, double
        r) {
99      vector<point> ans;
100     point v = B - A;
101     double a = dotProduct(v, v), b = dotProduct(A, v), c = dotProduct(A, A) - r * r;
102     double d = b * b - a * c;
103     if (d >= -epsilon)
104         ans.push_back(A + v * ((-b + sqrt(d + eps)) / a));
105     if (d > epsilon)
106         ans.push_back(A + v * ((-b - sqrt(d + eps)) / a));
107     return ans;
```

```
108 }
109
110 //Regresa las intersecciones de las circunferencias con centros O1, O2 y radios r1, r2.
111 vector<point> circleCircleIntersection(const point &O1, double r1, const point &O2, double
        r2) {
112     vector<point> ans;
113     double d = dist(O1, O2);
114     if (r1 + r2 < d || d + min(r1, r2) < max(r1, r2))
115         return ans;
116     return ans;
117 }
118
119
120 //Regresa el area del triangulo con vertices A, B y C.
121 double areaTriangle(const point &A, const point &B, const point &C) {
122     return fabs(crossProduct(B - A, C - A)) / 2;
123 }
124
125 //Regresa el area del poligono con vertices P[0], P[1], ... , P[n-1].
126 double areaPolygon(int n, const point P[]) {
127     double area = P[n - 1].x * P[0].y - P[0].x * P[n - 1].y;
128     for (int i = 0; i < n - 1; ++i)
129         area += P[i].x * P[i + 1].y - P[i + 1].x * P[i].y;
130     return fabs(area) / 2;
131 }
132
133 //Regresa true si el punto P esta en el interior del triangulo con vertices A, B y C.
134 bool pointInTriangle(const point &P, const point &A, const point &B, const point &C) {
135     point G = (A + B + C) / 3;
136     return !lineSegmentIntersect(A, B, P, G) && !lineSegmentIntersect(B, C, P, G) &&
137            !lineSegmentIntersect(C, A, P, G);
138 }
139
140 //Regresa true si el poligono con vertices P (ordenados en el sentido de las manecillas del
141 //reloj) es convexo.
142 bool isConvexPolygon(int n, const point P) {
143     for (int i = 2; i < n; ++i) {
144         if (crossProduct(P[i] - P[i - 1], P[i] - P[i - 1]) < 0)
145             return false;
146     return true;
147 }
148
149 //Regresa true si el punto Q esta en el interior del poligono convexo con vertices P.
150 //Un punto Q estara en el interior de un poligono no convexo si y solo cualquier rayo
151 //con origen en Q intersecta al poligono (en aristas) un numero impar de veces.
152 bool pointInConvexPolygon(const point &Q, int n, const point P[]) {
153     point G = P[0];
154     for (int i = 1; i < n; ++i)
155         G = G + P[i];
156     G = G / n;
157     for (int i = 1; i < n; ++i)
158         if (lineSegmentIntersect(P[i - 1], P[i], Q, G))
159             return false;
160     return true;
161 }
162
163 int main() {
164         return 0;
165 }
```