Desarrollo e implementación de un programa de trabajo para el Semillero de Programación

A. Echavarría Uribe
Ingeniería Matemática
Universidad EAFIT
Medellín, Colombia
Email: aechava3@eafit.edu.co

J. F. Cardona Mc'Cormick
Escuela de Ingeniería
Universidad EAFIT
Medellín, Colombia
Email: fcardona@eafit.edu.co

***Resumen*—El Semillero de Programación de la Universidad EAFIT busca preparar a los estudiantes para las maratones de programación organizadas por ACIS/REDIS y por la ACM-ICPC. En este trabajo se muestra cómo se desarrolló y se trabajó un programa de clases para este Semillero buscando preparar a estudiantes novatos para las maratones de programación.**

Palabras claves*—*Semillero de Programación, Maratones de programación, ACM-ICPC, ACIS/REDIS, Curso de programación competitiva.

I. Introducción

El Semillero de Programación es un grupo de la Universidad EAFIT en el que los estudiantes con interés en la programación, las matemáticas y los algoritmos tienen un espacio para aprender acerca de estos temas y prepararse para participar en las maratones de programación realizadas a nivel nacional por ACIS/REDIS [1] y a nivel internacional por la ACM-ICPC [2]. En este Semillero se enseñan los temas más útiles [3]–[6] para estas competencias: los algoritmos de grafos, strings y teoría de números, programación dinámica, la recursividad y las estructuras de datos.

Durante los últimos años el Semillero ha estado a cargo de estudiantes destacados en las maratones de programación bajo la supervisión de docentes del Departamento de Ingeniería de Sistemas y, a pesar de que se han trabajado y discutido algoritmos, temas y problemas, no se desarrolló un plan de trabajo para este grupo. Dos consecuencias de lo anterior son que los miembros del Semillero a veces no tenían la fundamentación necesaria para aprender algunos de los algoritmos más complejos y que cuando algún estudiante nuevo se encargaba del Semillero no sabía cuáles eran los conocimientos que los estudiantes tenían y qué temas nuevos por discutir se ajustaban a su nivel. Por lo anterior se decidió crear, desarrollar e implementar un plan de trabajo para el Semillero de modo que este tenga una estructura que permita la apropiación progresiva del conocimiento necesario para poder resolver los problemas de las maratones de programación y que proporcione a quienes se encargarán del Semillero en versiones futuras información acerca de los conocimientos que tienen los estudiantes.

Actualmente, los estudiantes que pertenecen al Semillero de Programación son en su mayoría de tercer semestre, lo que quiere decir que tienen conocimientos acerca de cómo programar mas no conocen las técnicas más utilizadas en la solución de problemas de maratones de programación como los son los algoritmos de grafos, strings, la programación dinámica y los conceptos y algoritmos básicos de teoría de números.

Se espera que los temas enseñados a los estudiantes durante este semestre y el próximo sirvan para que ellos tengan un buen desempeño en la Maratón Nacional de Programación ACIS/REDIS que se realiza en octubre y tengan la posibilidad de participar en la Maratón Regional Suramericana ACM-ICPC y en la Maratón Mundial ACM-ICPC bajo el nombre la Universidad com ha ocurrido en ocasiones anteriores.

II. Metodología

El plan de trabajo del Semillero está basado principalmente en la metodología utilizada por Steven Halim en el curso Competitive Programming de la Universidad Nacional de Singapur (NUS). Este curso, al igual que el Semillero, busca preparar a los estudiantes para las competencias de la ACM-ICPC y está incorporado como curso electivo para estudiantes de matemáticas, ciencias de la computación e ingeniería electrónica de tercer año con conocimientos previos de programación, algoritmos y estructuras de datos. En el curso se trabajan algoritmos de nivel medio y avanzado pero, a diferencia de un curso de algoritmos corriente, se hace énfasis en cómo implementarlos eficientemente y en sus aplicaciones a las competencias de programación, en lugar de enfocarse en las demostraciones de su corrección y el análisis formal de su complejidad [7].

El método de enseñanza del curso de la NUS consiste tener clases teóricas en las cuales se enseñen los algoritmos a trabajar y hacer las evaluaciones por medio de competencias en las que los problemas, tomados de archivos históricos de competencias anteriores de la ICPC, tengan relación con los temas enseñados. Las dos razones principales por las que decidieron trabajar con competencias en lugar de exámenes son preparar a los estudiantes para las competencias oficiales de la ACM-ICPC y motivarlos a ser cada vez mejores al ponerlos a medir sus habilidades contra los demás estudiantes. Este método ha resultado beneficioso para los estudiantes de la NUS y el Profesor Halim cree que es porque incita a los estudiantes a estudiar más para mejorar ya que buscar ser los mejores es es una característica natural de los humanos.

La principal ventaja que ha tenido este curso en la NUS ha sido darle a los estudiantes talentosos e interesados en la programación un espacio en el que puedan prepararse para las competencias y conocer otras personas interesadas en este tema con las que puedan discutir, competir y formar equipos que tengan buen desempeño en las competencias de la ACM-ICPC, llegando a participar en la Maratón Mundial ACM-ICPC [8]. El Semillero de Programación de EAFIT tiene el mismo

objetivo y es por esto que se decidió desarrollar un plan de trabajo basado en esta metodología y adaptarlo para el nivel del Semillero que va dirigido a estudiantes de tercer semestre y no de tercer año como lo es en la NUS.

II-A. Contenido y estructura de las sesiones

Para el desarrollo del plan de trabajo fue necesario escoger el contenido a trabajar en cada sesión de manera que se trataran de cubrir la mayoría de las técnicas básicas e intermedias de programación requeridas para las maratones. La elección de los temas si hizo basándose en el nivel de los estudiantes del Semillero, el material de los cursos "Competitive Programming" de la NUS [9] y "Escuela de verano para maratones de programación" de la Universidad Estatal de Campinas [6], los temas de varios libros acerca de algoritmos y de competencias de programación [3]–[5], [10], [11] y otros temas que se consideraron importantes de acuerdo a la experiencia obtenida en la participación en diferentes competencias de programación.
Cada sesión de Semillero, independiente del tema que se trabaje, consiste en tres partes principales:

1. Discusión y solución de los problemas propuestos como tarea en la sesión anterior.
   Esto se hace con el fin de que los estudiantes, luego de haber intentado resolver los problemas propuestos de manera individual, entiendan su solución y de esta manera aprendan de ella. Para los estudiantes que lograron resolver los problemas este es un espacio en el que pueden compartir su solución con sus compañeros y ver una implementación diferente del problema que resolvieron; esto último les permite conocer diferentes formas de desarrollar y pensar en un mismo algoritmo y posiblemente conocer funciones y métodos del lenguaje C++ que hacen las implementaciones más cortas y sencillas.
2. Exposición del nuevo tema a trabajar, mostrando los algoritmos, los elementos matemáticos relacionados y sus implementaciones en el lenguaje C++ [12].
   La forma de abordar los temas para llevar al estudiante al entendimiento del algoritmo está basada principalmente en las métodos de enseñanza manejados en los cursos Algorithms: Design and Analysis Part 1/2 [13], [14] e Introduction to Algorithms [15] y Competitive Programming [9]. Por otra parte, la implementación de los algoritmos se basó en las implementaciones mostradas en [3], [4], [11] con el fin de tener implementaciones eficientes y lo más sencillas posibles.
3. Presentación breve de los problemas propuestos como ejercicio para la siguiente sesión.
   Los problemas propuestos son seleccionados entre los problemas disponibles en los jueces de programación en línea UVa [16], Codeforces [17] y Spoj [18] buscando que se resuelvan utilizando los temas de la sesión y algunos temas de sesiones anteriores. Dado que el archivo de problemas en estos jueces es bastante extenso, la búsqueda de los problemas según los temas se hace con ayuda de los problemas que proponen Steven y Felix Halim en sus libros Competitive Programming 1/2 [3], [4], Ahmed Shamsul Arefin en su libro "Art of Programming Contests" [11] y con ayuda de dos buscadores de problemas en los cuales la búsqueda se hace de acuerdo al tema del cual trata el problema que son: el buscador de Codeforces y el un buscador de problemas para el sitio UVa desarrollado por Mark Greve [19]. Luego de seleccionarlos, los problemas deben resolverse ya que sus soluciones no están disponibles en ninguno de los libros o sitios web mencionados anteriormente.

Semana	Temas
1	Introducción a C++ y a los jueces de programación ¿En qué consiste una maratón de programación? Solución a un problema básico de maratón de programación
2	Arreglos en C++, Vectores de C++ y Grafos
3	Representación de grafos en C++ Entrada usando `getline` y `stringstream`
4	Pila y Cola Búsqueda en anchura (BFS) Búsqueda en profundidad (DFS)
5	Problemas de BFS y DFS
6	Map, Set, Heap Ordenamiento topológico Componentes fuertemente conexas (SCC)
7	Algoritmo de Dijkstra
8	Algoritmo de Bellman-Ford
9	Programación dinámica: Problemas clásicos
10	Algoritmo de Floyd-Warshall
11	Árbol de mínima expansión
12	Algoritmo de Knuth-Morris-Pratt
13	Algoritmo de máximo flujo
14	Solución de problemas de la IV Maratón de Programación UTP
15	Algoritmos de teoría de números

Tabla I. TEMAS DESARROLLADOS EN EL SEMILLERO

III. RESULTADOS

III-A. Programa clase a clase

El Semillero se desarrolló en 15 sesiones semanales de dos horas de duración cada una. Los contenidos trabajados en cada sesión se muestran en la tabla I. Acá se puede ver que la dificultad de los algoritmos va aumentando progresivamente y los elementos necesarios para entender e implementar un algoritmo específico se enseñan antes de dicho algoritmo. Ejemplos de esto serían la necesidad de aprender a utilizar los arreglos y los vectores antes de aprender las formas de representar un grafo, la importancia de conocer y entender el algoritmo de búsqueda en anchura y la estructura de datos del heap antes de implementar el algoritmo de Dijkstra, o explicar cómo funciona la programación dinámica antes de discutir el algoritmo de Floyd-Warshall.

III-B. Documentación

Con el fin de que facilitar la comprensión de los temas, de que los estudiantes tuvieran material con el cual pudieran repasar los temas de las secciones de manera independiente y de dejar un legado para las personas que vayan a estar a cargo del Semillero en el futuro, se decidieron crear diapositivas con los contenidos trabajados en cada sesión. El contenido de estas diapositivas se mantiene actualizado en el repositorio público https://github.com/anaechavarria/SemilleroProgramacion/ y además se comparte con los estudiantes al final de cada sesión. Durante el transcurso del semestre, se decidió dar a conocer el contenido de este repositorio con el director de las maratones de programación de la Universidad Tecnológica de Pereira

(UTP) y el de la Universidad Pontificia Bolivariana (UPB) quienes difundieron la información entre sus estudiantes. Esta información llegó también a manos de estudiantes de la Universidad Sergio Arboleda, quienes escribieron para preguntar si ellos también podían hacer uso del material para prepararse para las maratones de programación. Luego de compartir la información con estas universidades, 8 personas nuevas empezaron a seguir el contenido del repositorio y otras 3 lo marcaron como favorito.

Adicional al material de las diapositivas, se desarrolló un manual con los algoritmos vistos en el Semillero en el transcurso del semestre (ver Anexo). Este manual se realizó pensando en que los estudiantes tuvieran documentados los algoritmos aprendidos durante el semestre y pudieran estudiarlos más fácilmente. Se espera que los estudiantes lo complementen con los temas que aprendan en el Semillero el próximo semestre y con otros temas que consideren de importancia, lo utilicen como su propio manual de algoritmos en las maratones de programación y les sirva como herramienta para solucionar los problemas que se les presenten en las competencias.

III-C. Competencias

Como se mencionó en la metodología, las competencias son parte fundamental del programa del Semillero. Es por esto que para cada sesión se buscaron y seleccionaron entre 2 y 4 problemas que tuvieran relación con los temas vistos en dicha sesión y se crearon competencias con dichos problemas en los sitios Contests: Factor Común [20] y Virtual Online Contests [21], dos páginas web para crear competencias con problemas de los jueces de programación mostrados en el Semillero.
Se crearon un total de 11 competencias, los resultados de cada competencia se muestran en la tabla II. Allí se pueden observar que ocurrieron dos fenómenos importantes, el primero es que la cantidad de personas que participaron en el Semillero se redujo en el transcurso del semestre y el segundo es que, en las competencias 6 a 9, el número de problemas resueltos es muy bajo. Estos fenómenos se deben a que a medida que avanza el semestre, los compromisos académicos son mayores lo que hace que se necesite más tiempo para las actividades la Universidad. Como el Semillero es de carácter opcional, muchos estudiantes se ven forzados a dejar de asistir para poder atender los compromisos académicos obligatorios o no tienen suficiente tiempo para resolver los problemas.

Competencia	Problemas	Participantes	Problemas resueltos
1	4	agomezl	4
		svanegas	4
		estebanf01	4
		cmejia49	3
		yampyer	3
		SantiSP	2
		jlopera8	1
		srincon2	1
2	2	agomezl	2
		luisponce	2
		svanegas	1
		estebanf01	0
		cmejia49	0
		yampyer	0
		SantiSP	0
		jlopera8	0
3	4	zubieta	4
		svanegas	3
		luisponce	1
		jlopera8	1
		estebanf01	0
		cmejia49	0
		yampyer	0
4	3	zubieta	3
		svanegas	2
		estebanf01	2
		luisponce	2
5	3	svanegas	3
		luisponce	3
		estebanf01	3
		zubieta	1
6	4	svanegas	1
		spalac24	1
7	3	yampyer	0
		estebanf01	0
8	3	svanegas	1
		estebanf01	1
		yampyer	0
9	2	svanegas	0
		estebanf01	0
10	2	svanegas	2
		estebanf01	1
11	4	svanegas	Competencia actualmente
		estebanf01	en ejecución

Tabla II. RESULTADOS DE LAS COMPETENCIAS REALIZADAS

III-D. Problemas resueltos

Para el desarrollo del Semillero y de las competencias fue necesario buscar y solucionar problemas de los archivos de los jueces de programación que se ajustaran a los temas discutidos en casa sesión. Se resolvieron un total de 49 problemas; de estos 2 fueron resueltos en reuniones del Semillero, 34 fueron problemas propuestos para las competencias y los 13 restantes fueron los problemas de la IV Maratón de Programación UTP cuyas soluciones se discutieron en una sesión del Semillero.

III-E. Maratones de programación

Como parte del objetivo del Semillero, se invitó a los estudiantes a participar en el Circuito Colombiano de Maratones de Programación. Estas son maratones que se realizan a nivel nacional y son preparatorias para la Maratón Nacional de Programación ACIS/REDIS que se realizará en octubre de este año. En dos de las cuatro competencias que se han realizado este año, se tuvo la participación de 2 equipos de estudiantes del Semillero [22].
Por otro lado, el 4 de mayo se llevó a cabo la IV Maratón de Programación UTP que tenía una dificultad básica/intermedia y estaba pensada para el fortalecimiento de los competidores novatos [23]. En esta competencia, un equipo conformado por dos estudiantes del Semillero quedó en cuarto lugar, compitiendo contra equipos de otras universidades de Colombia.

IV. CONCLUSIÓN

El Semillero de Programación es un grupo importante de la Universidad en el que los estudiantes interesados en la programación y especialmente en competir en las maratones de programación tienen un espacio para aprender, discutir y socializar problemas. En los últimos años tres equipos de personas que han pertenecido al Semillero han clasificado a la Maratón Mundial de Programación ACM-ICPC pero actualmente los estudiantes del Semillero son novatos en las competencias y asisten a las reuniones buscando aprender y mejorar para poder tener buen desempeño en la Maratón Nacional de Programación ACIS/REDIS para poder clasificar

a la Maratón Regional Suramericana ACM-ICPC.

El desarrollo de un plan de trabajo para el Semillero de Programación ha permitido que estos estudiantes mejoren sus conocimientos en estructuras de datos, el lenguaje de programación C++, los problemas clásicos de programación dinámica y en los principales algoritmos de grafos, teoría de números y strings. Por otro lado, la metodología utilizada para el desarrollo del Semillero, en la cual se hacen constantemente competencias, fomenta el trabajo individual de los estudiantes y fortalece sus habilidades de programación y de resolución de problemas, además de permitirles aplicar los conocimientos adquiridos en cada sesión.

El plan de trabajo y el material utilizado en el Semillero (diapositivas, competencias, problemas y el manual de algoritmos) fueron desarrollados pensando en darle una estructura al Semillero de forma que los estudiantes se motiven a asistir por que los temas son adecuados para su nivel de conocimiento. Se busca que este contenido pueda ser utilizado por los estudiantes de la Universidad EAFIT y de otras universidades no solo para el estudio de manera independiente, sino también para que se siga desarrollando el Semillero y se implementen cursos similares en otras universidades.

Se pudo observar que el número de estudiantes que hacen parte del Semillero fue disminuyendo a medida que avanzaba el semestre. Esto se atribuye a que el curso no es de caracter obligatorio y los estudiantes, dado un aumento en sus compromisos académicos, se ven forzados a utilizar ese tiempo en las actividades obligatorias de la Universidad. Para evitar esto, se sugiere que el Semillero sea incorporado como un curso electivo del pénsum de Ingeniería de Sistemas e Ingeniería Matemática y de esta manera los estudiantes reciban créditos académicos por asistir a este curso y tengan un mayor compromiso con las actividades que allí se desarrollan. En universidades como la Universidad de los Andes y la Universidad Nacional de Singapur, se han implementado cursos electivos similares y han dado buenos resultados.

Durante el transcurso del semestre los estudiantes del Semillero han competido en maratones de programación a nivel nacional obteniendo el cuarto lugar en la IV Maratón de Programación UTP. Se espera que los estudiantes sigan mejorando y participando en este tipo de competencias para prepararse para la Maratón Nacional de Programación ACIS/REDIS que se realizará en octubre del presente año y poder clasificar a la Maratón Regional Suramericana de Programación ACM-ICPC y posiblemente a la Maratón Mundial de Programación ACM-ICPC.

V. Agradecimientos

- Al Departamento de Ingeniería de Sistemas de la Universidad EAFIT por su aporte financiero para la realización de este proyecto.
- A la Universidad EAFIT por proporcionar el espacio para las reuniones del Semillero.
- Al estudiante Santiago Palacio Gómez por su constante apoyo durante las sesiones del Semillero.
- Al profesor Francisco Correa por sus observaciones para la elaboración de los reportes.
- A los integrantes del Semillero de Programación por ser nuestra motivación para ser siempre mejores.

Referencias

[1] "Asociación Colombiana de Ingenieros de Sistemas (ACIS)," http://acis.org.co/.

[2] "International Collegiate Programming Contest (ICPC)," http://icpc.baylor.edu/.

[3] S. Halim and F. Halim, *Competitive Programming*. Lulu. com, 2010.

[4] ——, *Competitive Programming 2*. Lulu. com, 2011.

[5] S. S. Skiena and M. A. Revilla, *Programming challenges: The programming contest training manual*. Springer Heidelberg, 2003.

[6] A. Lopatin, F. Dias Moreira, and F. I. Schaposnik Massolo, "Material de Curso: Escola de Verão - Maratona de Programação 2012 - Instituto de Computação - UNICAMP," http://maratona.ic.unicamp.br/MaratonaVerao2012.

[7] S. Halim and F. Halim, "Competitive Programming in National University of Singapore," in *Ediciones Sello Editorial SL (Presented at Collaborative Learning Initiative Symposium (CLIS) ACM ICPC World Final 2010, Harbin, China*, 2010.

[8] Universidad Nacional de Singapur (NUS), "Algorithmics @ NUS Wiki." http://algorithmics.comp.nus.edu.sg/wiki/.

[9] S. Halim, "Material de clases del curso Competitive Programming," https://sites.google.com/site/stevenhalim/home/material.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[11] A. S. Arefin, *Art of Programming Contest*. Gyankosh Prokashoni, 2006.

[12] B. Stroustrup, *El lenguaje de programación C++*. Addison Wesley, 2002.

[13] T. Roughgarden, "Algorithms: Design and Analysis, Part 1 (Stanford University)," https://www.coursera.org/course/algo.

[14] ——, "Algorithms: Design and Analysis, Part 2 (Stanford University)," https://www.coursera.org/course/algo2.

[15] C. Leiserson and E. Demaine, "6.046J Introduction to Algorithms (SMA 5503)," Massachusetts Institute of Technology: MIT OpenCourseWare, http://ocw.mit.edu.

[16] M. A. Revilla, "UVa Online Judge," http://uva.onlinejudge.org.

[17] M. Mirzayanoc, "Codeforces," http://www.codeforces.com.

[18] Sphere Research Labs, "Sphere Online Judge," http://www.spoj.com.

[19] M. Greve, "UVa Toolkit," http://uvatoolkit.com/about.php.

[20] A. Mejía Posada, "Contests: Factor Común," http://contests.factorcomun.org/.

[21] A. Aly, "Virtual Online Contests," http://ahmed-aly.com/.

[22] "Circuito Colombiano de Maratones de Programación (CCMP)," http://acm.javeriana.edu.co/maratones/.

[23] U. T. de Pereira, "IV Maratón de Programació UTP," http://acm.javeriana.edu.co/maratones/2013/05/02/iv-maraton-interna-de-programacion-utp-2013/.

Anexo

ANEXO
Manual de algoritmos del semillero de programación EAFIT

Ana Echavarría

Índice

1. Plantilla

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

// Template para recorrer contenedores usando iteradores
```

```
#define foreach(x, v) for (typeof (v).begin() x=(v).begin(); \
                           x !=(v).end(); ++x)
// Template que imprime valores de variables para depurar
#define D(x) cout << #x " = " << (x) << endl

// Función para comparar dos dobles sin problemas de presición
// Retorna -1 si x < y, 0 si x = y, 1 si x > y
const double EPS = 1e-9;
int cmp (double x, double y, double tol = EPS){
    return (x <= y + tol) ? (x + tol < y) ? -1 : 0 : 1;
}


int main() {
    // Entrada y salida desde / hacia archivo
    // Eliminar si la entrada es estándar
    // Cambiar in.txt / out.txt por los archivos de entrada/salida
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);

    return 0;
}
```

.....

2. Grafos

2.1. BFS

Algoritmo de recorrido de grafos en anchura que empieza desde una fuente s y visita todos los nodos alcanzables desde s.
El BFS también halla la distancia más corta entre s y los demás nodos si las aristas tienen todas peso 1.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN]; // La lista de adyacencia
int d[MAXN];          // Distancia de la fuente a cada nodo

void bfs(int s, int n){ // s = fuente, n = número de nodos
   for (int i = 0; i <= n; ++i) d[i] = -1;
   queue <int> q;
   q.push(s);
   d[s] = 0;
   while (q.size() > 0){
      int cur = q.front();
      q.pop();
      for (int i = 0; i < g[cur].size(); ++i){
         int next = g[cur][i];
         if (d[next] == -1){
            d[next] = d[cur] + 1;
            q.push(next);
         }
      }
   }
}
```

.....

2.2. DFS

Algoritmo de recorrido de grafos en profundidad que empieza visita todos los nodos del grafo.
El algoritmo puede ser modificado para que retorne información de los nodos según la necesidad del problema.
El grafo tiene un ciclo $\leftrightarrow$ si en algún momento se llega a un nodo marcado como gris.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];      // La lista de adyacencia
int color[MAXN];           // El arreglo de visitados
enum {WHITE, GRAY, BLACK}; // WHITE = 1, GRAY = 2, BLACK = 3

// Visita el nodo u y todos sus vecinos empezando por los más profundos
void dfs(int u){
   color[u] = GRAY;   // Marcar el nodo como semi-visitado
   for (int i = 0; i < g[u].size(); ++i){
     int v = g[u][i];
     if (color[v] == WHITE) dfs(v);  // Visitar los vecinos
   }
   color[u] = BLACK;  // Marcar el nodo como visitado
}
```

```
// Llama la función dfs para los nodos 0 a n-1
void call_dfs(int n){
   for (int u = 0; u < n; ++u) color[u] = WHITE;
   for (int u = 0; u < n; ++u)
     if (color[u] == WHITE) dfs(u);
}
```

.....

2.3. Ordenamiento topológico

Dado un grafo no cíclico y dirigido (DAG), ordena los nodos linealmente de tal forma que si existe una arista entre los nodos u y v entonces u aparece antes que v en el ordenamiento.
Este ordenamiento se puede ver como una forma de poner todos los nodos en una línea recta y que las aristas vayan todas de izquierda a derecha.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];    // La lista de adyacencia
bool seen[MAXN];         // El arreglo de visitados para el dfs
vector <int> topo_sort;  // El vector del ordemamiento

void dfs(int u){
   seen[u] = true;
   for (int i = 0; i < g[u].size(); ++i){
      int v = g[u][i];
      if (!seen[v]) dfs(v);
   }
   topo_sort.push_back(u); // Agregar el nodo al ordenamiento
}
void topological(int n){   // n = número de nodos
   topo_sort.clear();
   for (int i = 0; i < n; ++i) seen[i] = false;
   for (int i = 0; i < n; ++i) if (!seen[i]) dfs(i);
   reverse(topo_sort.begin(), topo_sort.end());
}
```

.....

2.4. Componentes fuertemente conexas

Dado un grafo dirigido, calcula la componente fuertemente conexa (SCC) a la que pertenece cada nodo.
Para cada pareja de nodos u, v que pertenecen a una misma SCC se cumple que hay un camino de u a v y de v a u.
Si se comprime el grafo dejando como nodos cada una de las componentes se quedará con un DAG.
Complejidad: $O(n + m)$ donde n es el número de nodos y m es el número de aristas.

```
vector <int> g[MAXN];    // El grafo
vector <int> grev[MAXN]; // El grafo con las aristas reversadas
vector <int> topo_sort;  // El "ordenamiento topologico" del grafo
int scc[MAXN];   // La componente a la que pertenece cada nodo
bool seen[MAXN]; // El arreglo de visitado para el primer DFS

// DFS donde se halla el ordenamiento topológico
void dfs1(int u){
   seen[u] = true;
   for (int i = 0; i < g[u].size(); ++i){
      int v = g[u][i];
      if (!seen[v]) dfs1(v);
   }
   topo_sort.push_back(u);
}
// DFS donde se hallan las componentes
void dfs2(int u, int comp){
   scc[u] = comp;
   for (int i = 0; i < grev[u].size(); ++i){
      int v = grev[u][i];
      if (scc[v] == -1) dfs2(v, comp);
   }
}

// Halla las componentes fuertemente conexas del grafo usando
// el algoritmo de Kosaraju. Retorna la cantidad de componentes
int find_scc(int n){ // n = número de nodos
   // Crear el grafo reversado
   for (int u = 0; u < n; ++u){
      for (int i = 0; i < g[u].size(); ++i){
         int v = g[u][i];
         grev[v].push_back(u);
      }
   }
```

```
   // Llamar el primer dfs
   for (int i = 0; i < n; ++i){
      if (!seen[i]) dfs1(i);
   }
   reverse(topo_sort.begin(), topo_sort.end());

   // Llamar el segundo dfs
   int comp = 0;
   for (int i = 0; i < n; ++i){
      int u = topo_sort[i];
      if (scc[u] == -1) dfs2(u, comp++);
   }
   return comp;
}
```

2.5. Algoritmo de Dijkstra

Dado un grafo con pesos **no negativos** en las aristas, halla la mínima distancia entre una fuente s y los demás nodos.
Al heap se inserta primero la distancia y luego en nodo al que se llega. Si se quieren modificar los pesos por `long long` o por `double` se debe cambiar en los tipos de dato `dist_node` y `edge`.
Complejidad: $O((n+m)\log n)$ donde n es el número de nodos y m es el número de aristas.

```
const int MAXN = 100005;
const int INF = 1 << 30;      // Usar 1LL << 60 para long long
typedef pair <int, int> dist_node; // Datos del heap (dist, nodo)
typedef pair <int, int> edge; // Dato de las arista (nodo, peso)
vector <edge> g[MAXN];        // g[u] = (v = nodo, w = peso)
int d[MAXN];    // d[u] La distancia más corta de s a u
int p[MAXN];    // p[u] El predecesor de u en el camino más corto

// La función recibe la fuente s y el número total de nodos n
void dijkstra(int s, int n){
   for (int i = 0; i <= n; ++i){
      d[i] = INF;  p[i] = -1;
   }
   priority_queue < dist_node, vector <dist_node>,
                    greater<dist_node> > q;
   d[s] = 0;
   q.push(dist_node(0, s));
   while (!q.empty()){
      int dist = q.top().first;
      int cur = q.top().second;
      q.pop();
      if (dist > d[cur]) continue;
      for (int i = 0; i < g[cur].size(); ++i){
         int next = g[cur][i].first;
         int w_extra = g[cur][i].second;
         if (d[cur] + w_extra < d[next]){
            d[next] = d[cur] + w_extra;
            p[next] = cur;
            q.push(dist_node(d[next], next));
         }
      }
   }
}

// La función que retorna los nodos del camino más corto de s a t
// Primero hay que correr dijktra desde s.
// Eliminar si no se necesita hallar el camino.
vector <int> find_path (int t){
   vector <int> path;
   int cur = t;
   while(cur != -1){
      path.push_back(cur);
      cur = p[cur];
   }
   reverse(path.begin(), path.end());
   return path;
}
```

2.6. Algoritmo de Bellman-Ford

Dado un grafo con pesos cualquiera, halla la mínima distancia entre una fuente s y los demás nodos.
Si hay un ciclo de peso negativo en el grafo, el algoritmo lo indica.
Complejidad: $O(n \times m)$ donde n es el número de nodos y m es el número de aristas.

```
const int MAXN = 105;
```

```
const int INF = 1 << 30;  // Para long long INF = 1LL << 60
typedef pair <int, int> edge; // Modificar según el problema
vector <edge> g[MAXN]; // g[u] = (v = nodo, w = peso)
int d[MAXN];           // d[u] = distancia más corta de s a u

// Retorna verdadero si el grafo tiene un ciclo de peso negativo
// alcanzable desde s y falso si no es así.
// Al finalizar el algoritmo, si no hubo ciclo de peso negativo,
// la distancia más corta entre s y u está almacenada en d[u]
bool bellman_ford(int s, int n){ // s = fuente, n = número nodos
   for (int u = 0; u <= n; ++u) d[u] = INF;
   d[s] = 0;

   for (int i = 1; i <= n - 1; ++i){
      for (int u = 0; u < n; ++u){
         for (int k = 0; k < g[u].size(); ++k){
            int v = g[u][k].first;
            int w = g[u][k].second;
            d[v] = min(d[v], d[u] + w);
         }
      }
   }

   for (int u = 0; u < n; ++u){
      for (int k = 0; k < g[u].size(); ++k){
         int v = g[u][k].first;
         int w = g[u][k].second;
         if (d[v] > d[u] + w) return true;
      }
   }
   return false;
}
```

2.7. Algoritmo de Floyd-Warshall

Dado un grafo con pesos cualquiera, halla la mínima distancia entre cualquier para de nodos.
Si este algoritmo es muy lento para el problema ejecutar n veces el algoritmo de Dijkstra o de Bellman-Ford según el caso.
Complejidad: $O(n^3)$ donde n es el número de nodos.

Casos base: `d[i][j]` $= \begin{cases} 0 & \text{si } i = j \\ w_{i,j} & \text{si existe una arista entre } i \text{ y } j \\ +\infty & \text{en otro caso} \end{cases}$

Nota: Utilizar el tipo de dato apropiado (`int`, `long long`, `double`) para `d` y para $+\infty$ según el problema.

```
// Los nodos están numerados de 0 a n-1
for (int k = 0; k < n; ++k){
   for (int i = 0; i < n; ++i){
      for (int j = 0; j < n; ++j){
         d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
      }
   }
}
// Acá d[i][j] es la mínima distancia entre el nodo i y el j
```

2.7.1. Clausura transitiva

Dado un grafo cualquiera, hallar si existe un camino desde i hasta j para cualquier pareja de nodos i, j

Casos base: `d[i][j]` $= \begin{cases} \texttt{true} & \text{si } i = j \\ \texttt{true} & \text{si existe una arista entre } i \text{ y } j \\ \texttt{false} & \text{en otro caso} \end{cases}$

Caso recursivo: `d[i][j] = d[i][j] or (d[i][k] and d[k][j]);`

2.7.2. Minimax

Dado un grafo con pesos, hallar el camino de i hasta j donde la arista más grande del camino sea lo más pequeña posible.
Ejemplos: Que el peaje más caro sea lo más barato posible, que la autopista más larga sea lo más corta posible.

Casos base: `d[i][j]` $= \begin{cases} 0 & \text{si } i = j \\ w_{i,j} & \text{si existe una arista entre } i \text{ y } j \\ +\infty & \text{en otro caso} \end{cases}$

Caso recursivo: `d[i][j] = min( d[i][j], max (d[i][k], d[k][j]) );`

2.7.3. Maximin

Dado un grafo con pesos, hallar el camino de i hasta j donde la arista más pequeña del camino sea lo más grande posible.
Ejemplos: Que el trayecto menos seguro sea lo más seguro posible, que la autopista de menos carriles tenga la mayor cantidad de carriles.

Casos base: `d[i][j]` $= \begin{cases} +\infty & \text{si } i = j \\ w_{i,j} & \text{si existe una arista entre } i \text{ y } j \\ -\infty & \text{en otro caso} \end{cases}$

Caso recursivo: `d[i][j] = max( d[i][j] , min(d[i][k], d[k][j]) )`

2.8. Algoritmo de Prim

Dado un grafo no dirigido y conexo, retorna el costo del árbol de mínima expansión de ese grafo.
El costo del árbol de mínima expansión también se puede ver como el mínimo costo de las aristas de manera que haya un camino entre cualquier par de nodos.
Complejidad: $O(m \log n)$ donde n es el número de nodos y m es el número de aristas.

```
const int MAXN = 10005;
typedef pair <int, int> edge;        // Pareja (nodo, peso)
typedef pair <int, int> weight_node; // Pareja (peso, nodo)
vector <edge> g[MAXN];               // Lista de adyacencia
bool visited[MAXN];

// Retorna el costo total del MST
int prim(int n){ // n = número de nodos
    for (int i = 0; i <= n; ++i) visited[i] = false;
    int total = 0;

    priority_queue<weight_node, vector <weight_node>,
                   greater<weight_node> > q;

    // Empezar el MST desde 0 (cambiar si el nodo 0 no existe)
    q.push(weight_node(0, 0));
    while (!q.empty()){
        int u = q.top().second;
        int w = q.top().first;
        q.pop();
        if (visited[u]) continue;

        visited[u] = true;
        total += w;
        for (int i = 0; i < g[u].size(); ++i){
            int v = g[u][i].first;
            int next_w = g[u][i].second;
            if (!visited[v]){
                q.push(weight_node(next_w, v));
            }
        }
    }
    return total;
}
```

.....

2.9. Algoritmo de Kruskal

2.9.1. Union-Find

Union-Find es una estructura de datos para almacenar una colección conjuntos disjuntos (no tienen elementos en común) que cambian dinámicamente. Identifica en cada conjunto un "padre" que es un elemento al azar de ese conjunto y hace que todos los elementos del conjunto "apunten" hacia ese padre.
Inicialmente se tiene una colección donde cada elemento es un conjunto unitario.
Complejidad aproximada: $O(m)$ donde m el número total de operaciones de `initialize`, `union` y `join` realizadas.

```
const int MAXN = 100005;
int p[MAXN];// El padre del conjunto al que pertenece cada nodo

// Inicializar cada conjunto como unitario
void initialize(int n){
   for (int i = 0; i <= n; ++i) p[i] = i;
}

// Encontrar el padre del conjunto al que pertenece u
int find(int u){
    if (p[u] == u) return u;
```

```
    return p[u] = find(p[u]);
}

// Unir los conjunto a los que pertenecen u y v
void join(int u, int v){
     int a = find(u);
     int b = find(v);
     if (a == b) return;
     p[a] = b;
}
```

.....

2.9.2. Algoritmo de Kruskal

Dado un grafo no dirigido y conexo, retorna el costo del árbol de mínima expansión de ese grafo.
El costo del árbol de mínima expansión también se puede ver como el mínimo costo de las aristas de manera que haya un camino entre cualquier par de nodos.
Utiliza Union-Find para ver rápidamente qué aristas generan ciclos.
Complejidad: $O(m \log n)$ donde n es el número de nodos y m es el número de aristas.

```
struct edge{
   int start, end, weight;

   edge(int u, int v, int w){
      start = u; end = v; weight = w;
   }
   bool operator < (const edge &other) const{
      return weight < other.weight;
   }
};

const int MAXN = 100005;
vector <edge> edges; // Lista de aristas y no lista de adyacencia
int p[MAXN];         // El padre de cada conjunto (union-find)

// Incluir las operaciones de Union-Find (initialize, find, join)

int kruskal(int n){
    initialize(n);
    sort(edges.begin(), edges.end());

    int total = 0;
    for (int i = 0; i < edges.size(); ++i){
        int u = edges[i].start;
        int v = edges[i].end;
        int w = edges[i].weight;
        if (find(u) != find(v)){
            total += w;
            join(u, v);
        }
    }
    return total;
}
```

.....

2.10. Algoritmo de máximo flujo

Dado un grafo con capacidades enteras, halla el máximo flujo entre una fuente s y un sumidero t.
Como el máximo flujo es igual al mínimo corte, halla también el mínimo costo de cortar aristas de manera que s y t queden desconectados.
Si hay varias fuentes o varios sumideros poner una súper-fuente / súper-sumidero que se conecte a las fuentes / sumideros con capacidad infinita.
Si los nodos también tienen capacidad, dividir cada nodo en dos nodos: uno al que lleguen todas las aristas y otro del que salgan todas las aristas y conectarlos con una arista que tenga la capacidad del nodo.
Complejidad: $O(n \cdot m^2)$ donde n es el número de nodos y m es el número de aristas.

```
const int MAXN = 105;
// Lista de adyacencia de la red residual
vector <int> g [MAXN];
// Capacidad de aristas de la red de flujos
int c [MAXN][MAXN];
// El flujo de cada arista
int f [MAXN][MAXN];
//El predecesor de cada nodo en el camino de aumentación de s a t
int prev [MAXN];

void connect (int i, int j, int cap){
    // Agregar SIEMPRE las dos aristas a g (red residual) así el
```

```
    // grafo sea dirigido. Esto es porque g representa la red
    // residual que tiene aristas en los dos sentidos.
    g[i].push_back(j);
    g[j].push_back(i);
    c[i][j] += cap;
    // Omitir esta línea si el grafo es dirigido
    c[j][i] += cap;
}

// s = fuente, t = sumidero, n = número de nodos
int maxflow(int s, int t, int n){
    for (int i = 0; i <= n; i++){
        for (int j = 0; j <= n; j++){
            f[i][j] = 0;
        }
    }

    int flow = 0;
    while (true){
        for (int i = 0; i <= n; i++) prev[i] = -1;

        queue <int> q;
        q.push(s);
        prev[s] = -2;

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

        int extra = 1 << 30;
        int end = t;
        while (end != s){
            int start = prev[end];
            extra = min(extra, c[start][end] - f[start][end]);
            end = start;
        }

        end = t;
        while (end != s){
            int start = prev[end];
            f[start][end] += extra;
            f[end][start] = -f[start][end];
            end = start;
        }

        flow += extra;
    }

    return flow;
}
```

.....

3. Teoría de números

3.1. Divisores de un número

Imprime los divisores de un número (cuidado que no lo hace en orden). Complejidad: $O(\sqrt{n})$ donde n es el número.

```
void divisors(int n){
    int i;
    for (i = 1; i * i < n; ++i){
        if (n % i == 0) printf("%d\n%d\n", i, n/i);
    }
    // Si existe, imprimir su raiz cuadrada una sola vez
    if (i * i == n) printf("%d\n", i);
}
```

.....

3.2. Máximo común divisor y mínimo común múltiplo

Para hallar el máximo común divisor entre dos números `a` y `b` ejecutar el comando `__gcd(a, b)`.

Para hallar el mínimo común múltiplo: $\text{lcm}(a,b) = \dfrac{|a \cdot b|}{\gcd(a,b)}$

3.3. Criba de Eratóstenes

Encuentra los primos desde 1 hasta un límite n.
`sieve[i]` es falso sí y solo sí i es un número primo.
Complejidad: $O(n)$ donde n es el límite superior.

```
const int MAXN = 1000000;
bool sieve[MAXN + 5];
vector <int> primes;

void build_sieve(){
    memset(sieve, false, sizeof(sieve));
    sieve[0] = sieve[1] = true;

    for (int i = 2; i * i <= MAXN; i ++){
        if (!sieve[i]){
            for (int j = i * i; j <= MAXN; j += i){
                sieve[j] = true;
            }
        }
    }
    for (int i = 2; i <= MAXN; ++i){
        if (!sieve[i]) primes.push_back(i);
    }
}
```

.....

3.4. Factorización prima de un número

Halla la factorización prima de un número a positivo. Si a es negativo llamar el algoritmo con $|a|$ y agregarle -1 a la factorización.
Se asume que ya se ha ejecutado el algoritmo para generar los primos hasta al menos $\sqrt{a}$.
El algoritmo genera la lista de primos en orden de menor a mayor.
Utiliza el hecho de que en la factorización prima de a aparece máximo un primo mayor a $\sqrt{a}$.
Complejidad aproximada: $O(\sqrt{a})$

```
const int MAXN = 1000000; // MAXN > sqrt(a)
bool sieve[MAXN + 5];
vector <int> primes;

vector <long long> factorization(long long a){
    // Se asume que se tiene y se llamó la función build_sieve()
    vector <long long> ans;
    long long b = a;
    for (int i = 0; 1LL * primes[i] * primes[i] <= a; ++i){
        int p = primes[i];
        while (b % p == 0){
            ans.push_back(p);
            b /= p;
        }
    }
    if (b != 1) ans.push_back(b);
    return ans;
}
```

.....

3.5. Exponenciación logarítmica

3.5.1. Propiedades de la operación módulo

- $(a \bmod n) \bmod n = a \bmod n$
- $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
- $(a \cdot b) \bmod n = ((a \bmod n) \cdot (b \bmod n)) \bmod n$
- $\left(\frac{a}{b}\right) \bmod n \neq \left(\frac{a \bmod n}{b \bmod n}\right) \bmod n$

3.5.2. Big mod

Halla rápidamente el valor de $b^p \bmod m$ para $0 \leq b, p, m \leq 2147483647$
Si se cambian los valores por `long long` los límites se cambian por $0 \leq b, p \leq 9223372036854775807$ y $1 \leq m \leq 3037000499$.
Complejidad: $O(\log p)$

```
int bigmod(int b, int p, int m){
    if (p == 0) return 1;
    if (p % 2 == 0){
        int mid = bigmod(b, p/2, m);
        return (1LL * mid * mid) % m;
    }else{
        int mid = bigmod(b, p-1, m);
```

```
        return (1LL * mid * b) % m;
    }
}
```

3.6. Combinatoria

3.6.1. Coeficientes binomiales

Halla el valor de $\binom{n}{k}$ para $0 \leq k \leq n \leq 66$. Para $n > 66$ los valores comienzan a ser muy grandes y no caben en un `long long`.
Complejidad: $O(n^2)$

```
const int MAXN = 66;
unsigned long long choose[MAXN+5][MAXN+5];

void binomial(int N){
    for (int n = 0; n <= N; ++n) choose[n][0] = choose[n][n] = 1;

    for (int n = 1; n <= N; ++n){
        for (int k = 1; k < n; ++k){
            choose[n][k] = choose[n-1][k-1] + choose[n-1][k];
        }
    }
}
```

3.6.2. Propiedades de combinatoria

- El número de permutaciones de n elementos diferentes es $n!$
- El número de permutaciones de n elementos donde hay m_1 elementos repetidos de tipo 1, m_2 elementos repetidos de tipo 2, ..., m_k elementos repetidos de tipo k es
$$\frac{n!}{m_1!m_2!\cdots m_k!}$$
- El número de permutaciones de k elementos diferentes tomados de un conjunto de n elementos es
$$\frac{n!}{(n-k)!} = k!\binom{n}{k}$$

4. Programación dinámica

4.1. Problema de la mochila

Halla el valor máximo que se puede obtener al empacar un subconjunto de n objetos en una mochila de tamaño W cuando se conoce el valor y el tamaño de cada objeto.
Complejidad: $O(n \times W)$ donde n es el número de objetos y W es la capacidad de la mochila.

```
// Máximo número de objetos
const int MAXN = 2005;
// Máximo tamaño de la mochila
const int MAXW = 2005;
// w[i] = peso del objeto i (i comienza en 1)
int w[MAXN];
// v[i] = valor del objeto i (i comienza en 1)
int v[MAXN];
// dp[i][j] máxima ganancia si se toman un subconjunto de los
// objetos 1 .. i y se tiene una capacidad de j
int dp[MAXN][MAXW];

int knapsack(int n, int W){
    for (int j = 0; j <= W; ++j) dp[0][j] = 0;

    for (int i = 1; i <= n; ++i){
        for (int j = 0; j <= W; ++j){
            dp[i][j] = dp[i-1][j];
            if (j - w[i] >= 0){
                dp[i][j] = max(dp[i][j], dp[i-1][j-w[i]] + v[i]);
            }
        }
    }
    return dp[n][W];
}
```

5. Strings

5.1. Longest common subsequence

Halla la longitud de la máxima subsecuencia (no substring) de dos cadenas s y t.
Una subsecuencia de una secuencia s es una secuencia que se puede obtener de s al borrarle algunos de sus elementos (probablemente todos) sin cambiar el orden de los elementos restantes.
El algoritmo también se puede aplicar para vectores de elementos, no sólo para strings.
Complejidad: $O(n \times m)$ donde n es la longitud de s y m es la longitud de t.

```
// Máximo tamaño de los strings
const int MAXN = 1005;
// dp[i][j] lcs de los substrings s[0..i-1] y t[0..j-1]
int dp[MAXN][MAXN];

int lcs(const string &s, const string &t){
    int n = s.size(), m = t.size();

    for (int j = 0; j <= m; ++j) dp[0][j] = 0;
    for (int i = 0; i <= n; ++i) dp[i][0] = 0;

    for (int i = 1; i <= n; ++i){
        for (int j = 1; j <= m; ++j){
            dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            if (s[i-1] == t[j-1]){
                dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1);
            }
        }
    }
    return dp[n][m];
}
```

5.2. Longest increasing subsequence

Halla la longitud de la subsecuencia creciente más larga que hay en un string s (también se puede usar con vectores).
Complejidad: $O(n^2)$ donde n es la longitud de s.

```
// Máximo tamaño del string
const int MAXN = 1005;
// dp[i] = la longitud de la máxima subsecuencia creciente que
// termina en el caracter i (usándolo)
int dp[MAXN];

int lis(const string &s){
    int n = s.size();
    dp[0] = 1;
    for (int i = 1; i < n; i++){
        dp[i] = 1;
        for (int j = 0; j < i; j++){
            // Cambiar el < por <= si la secuencia tiene que ser
            // es estrictamente creciente
            if (s[j] <= s[i]) dp[i] = max(dp[i], dp[j] + 1);
        }
    }

    int best = 0;
    for (int i = 0; i < n; i++) best = max(best, dp[i]);
    return best;
}
```

5.3. Algoritmo de KMP

Encuentra si el string `needle` aparece en en string `haystack`.
Si no se retorna directamente `true` cuando se halla la primera ocurrencia, el algoritmo encuentra todas las ocurrencias de `needle` en `haystack`.
La primera parte del algoritmo llena el arreglo `border` donde `border[i]` es la longitud del borde del prefijo de `needle` que termina en la posición `i`. Un borde de una cadena s es la cadena más larga que es a la vez prefijo y sufijo de s pero que es diferente de s.
Complejidad: $O(n)$ donde n es el tamaño de `haystack`.

```
bool kmp(const string &needle, const string &haystack){
   int m = needle.size();
   vector<int> border(m);
   border[0] = 0;

   for (int i = 1; i < m; ++i) {
      border[i] = border[i - 1];
```

```
      while (border[i] > 0 and needle[i] != needle[border[i]])
         border[i] = border[border[i] - 1];
      if (needle[i] == needle[border[i]]) border[i]++;
   }

   int n = haystack.size();
   int seen = 0;
   for (int i = 0; i < n; ++i){
      while (seen > 0 and haystack[i] != needle[seen])
         seen = border[seen - 1];
      if (haystack[i] == needle[seen]) seen++;
      if (seen == m) return true; // Ocurre entre [i - m + 1, i]
   }
   return false;
}
```