## horizontal line



Práctica 4 : Cuadrado Latino

2019

**──**

Realizado por:

Miguel Ángel Campos Cubillas

Alejandro Pinel Martínez

Guillermo Palomino Sánchez

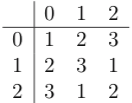
Pablo Lombardero Ros

Nikita Stetskiy

# Problema : Cuadrado Latino

Un cuadrado latino consiste en un tablero o matriz de tamaño n x n que hay que rellenar con los números de 1 a n sin que se repitan números en ninguna fila ni en ninguna columna.

Ejemplo (Cuadrado Latino de 3 x 3)



# Objetivo

1. Diseñar e implementar una algoritmo backtracking para conseguir cuadrados latinos de tamaño n x n ( para generar solo uno o todos los posibles).
2. Modificar el algoritmo para que los cuadrados latinos que genere cumplan un conjunto de restricciones (una asignación previa de valores en determinadas posiciones de la matriz).
3. Realizar un estudio empírico de su eficiencia.

# Algoritmo Backtracking

La clase que se usa para realizar el algoritmo, llamada *CuadradoLatino*, se compone de 3 atributos que son a su vez matrices. Una matriz de enteros llamada sol, que es donde irá almacenada la solución o las soluciones, una vez hayan sido encontradas y dos matrices de booleanos llamadas *Filas* y *Columnas*, que es donde irán almacenadas las posiciones ocupadas y cuya función es mantener un registro para sacar de forma más sencilla la solución.

El algoritmo de backtracking utilizado para resolver el problema planteado utiliza un método principal llamado *BuscarSolucion* que se apoya de la recursividad y otros métodos auxiliares, tales como el método *FactibleConstante* (que se encarga de ver la factibilidad de la solución), para resolver y formar una solución que sea un cuadrado latino.

|  |
| --- |
| #ifndef CUADRADOLATINO\_H  #define CUADRADOLATINO\_H  #include <vector>  using namespace std;  class CuadradoLatino {  private:  vector<vector<int> > sol;  vector<vector<bool> > Filas;  vector<vector<bool> > Columnas;  void Inicializar(int tam);  public:  CuadradoLatino(int tam);  CuadradoLatino(const CuadradoLatino & t);  int getTam() const;  bool setPos(int fil, int col, int n);  *bool FactibleConstante(vector<vector<int> > & cuadrado, int i, int j);*  void LimpiarPosiciones(vector<vector<int> > & cuadrado, int k, int l);  *bool BuscarSolucion(bool sol\_unica);*  *bool BuscarSolucion(vector<vector<int> > & cuadrado, bool sol\_unica, int k, int l);*  void imprimirSol();  void mostrarTablero(vector< vector <int>> tablero);  };  #endif |
| * Los métodos señalados en cursiva son los utilizados para desarrollar el algoritmo. * También se puede observar la estructura de datos utilizada para el cuadrado |

La forma en la que funciona este algoritmo es la siguiente, dado un cuadrado vacío, o con algunas de sus posiciones inicializadas, de n\*n tamaño se irá introduciendo un valor único, k, por columnas. Al introducir dicho valor en el cuadrado, se comprobará mediante el método *FactibleConstante* si el valor introducido es válido. Para ello se comprobará si el valor ya se ha introducido antes en esa fila y columna, en caso de ser un valor inválido se pasa a la siguiente posible solución, en caso contrario se continúa introduciendo valores de forma recursiva.

|  |
| --- |
| bool CuadradoLatino::BuscarSolucion(vector<vector<int> > & cuadrado, bool sol\_unica, int k, int l) {  bool resuelto = false;  int tam = getTam();  if (k == tam) {  sol = cuadrado;  resuelto = true;  imprimirSol();  }  else {  int nuevol = (l + 1)%tam;  int nuevok = k;  if (nuevol == 0)  nuevok++;  if (cuadrado[k][l] != -1) {  resuelto = BuscarSolucion(cuadrado, sol\_unica, nuevok, nuevol);  }  else {  bool terminar = resuelto && sol\_unica;  for (int i = 0; i < getTam() && !terminar; i++) {  cuadrado[k][l] = i; **//Introducción de un nuevo valor al cuadrado**  if (FactibleConstante(cuadrado, k, l)) { **//Comprobación Factibilidad**  resuelto = BuscarSolucion(cuadrado, sol\_unica, nuevok, nuevol);  LimpiarPosiciones(cuadrado, k, l);  terminar = resuelto && sol\_unica;  }  else  cuadrado[k][l] = -1;  }  }  }  return resuelto;  } |
| Este es el método principal del algoritmo implementado |

|  |
| --- |
| bool CuadradoLatino::FactibleConstante(vector<vector<int> > & cuadrado, int i, int j) {  int valor = cuadrado[i][j];  bool factible = true;  if (Filas[i][valor] || Columnas[j][valor])  factible = false;  else {  Filas[i][valor] = true;  Columnas[j][valor] = true;  }  return factible;  } |
| La comprobación de factibilidad se realiza comprobando si alguna posición de la fila o la columna contiene el valor *true* |

Una vez hallada una posible solución, se comprueba que, correctamente, se trata de una solución válida. Por último se imprime la solución por pantalla y se limpia el cuadrado para, en caso de haberlo indicado, buscar otra posible solución.

|  |
| --- |
| if (k == tam) {  sol = cuadrado;  resuelto = true;  imprimirSol();  } |
| Comprobación de solución correcta e impresión por pantalla |

|  |
| --- |
| if (FactibleConstante(cuadrado, k, l)) { **//Comprobación Factibilidad**  resuelto = BuscarSolucion(cuadrado, sol\_unica, nuevok, nuevol);  LimpiarPosiciones(cuadrado, k, l);  terminar = resuelto && sol\_unica;  } |
| Borrado del cuadrado para la siguiente posible solución |

Como se ha mencionado anteriormente, para el funcionamiento del algoritmo se necesita un cuadrado, una matriz n\*n, que puede estar vacía o tener algunas de sus posiciones inicializadas a algún valor y, a partir de ello, generar un cuadrado latino.

En cuanto a las restricciones del algoritmo:

Restricciones explícitas: Cada posición de la matriz sólo puede contener un valor.

Restricciones implícitas: El valor de cada posición debe variar entre 0 y n-1 y debe ser único en cada fila y cada columna.

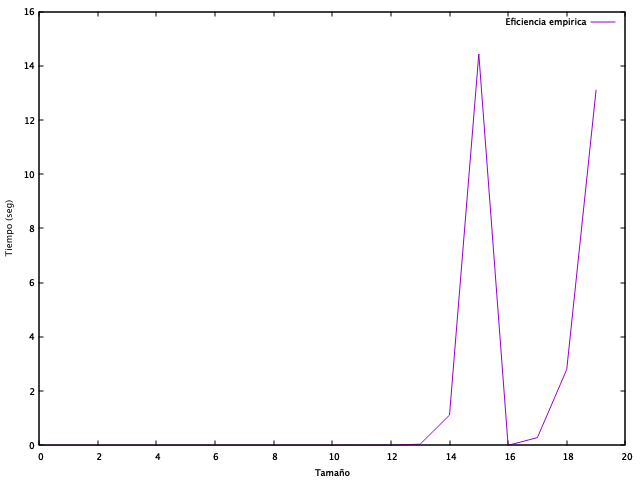
Por ejemplo, si tenemos un valor *k* en la posición (2,3), habrá que tener en cuenta que ya no podemos introducir k en la fila 2 ni en la columna 3.

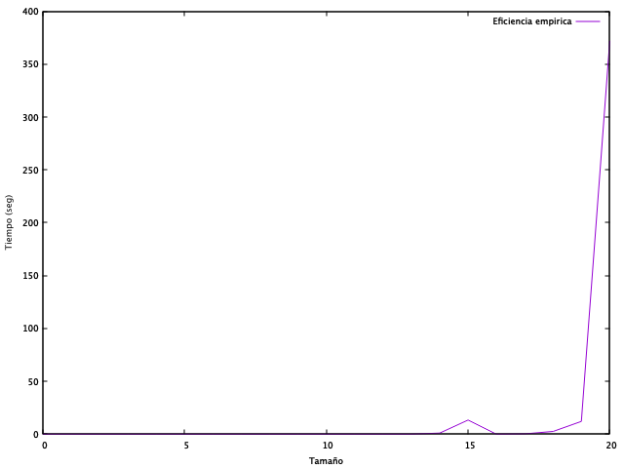
# Eficiencia Empirica

Como se ha mencionado en el anteriormente, este algoritmo funciona de manera recursiva y utiliza una matriz n\*n. Además de esto hemos implementado una serie de restricciones que provocan que haya que hacer más comprobaciones, por lo que el tiempo de ejecución será mayor cada vez que se aumente de tamaño la matriz.

|  |  |
| --- | --- |
| 0 | 6e-06 |
| 1 | 1.5e-05 |
| 2 | 1.7e-05 |
| 3 | 2.7e-05 |
| 4 | 1.8e-05 |
| 5 | 2.6e-05 |
| 6 | 3.5e-05 |
| 7 | 6.9e-05 |
| 8 | 5.1e-05 |
| 9 | 0.000169 |
| 10 | 0.000879 |
| 11 | 0.002558 |
| 12 | 0.006267 |
| 13 | 0.038836 |
| 14 | 1.11541 |
| 15 | 14.4352 |
| 16 | 0.000198 |
| 17 | 0.283231 |
| 18 | 2.80532 |
| 19 | 13.0959 |
| 20 | 372.565 |

La gráfica de la eficiencia de nuestro algoritmo backtracking es la siguiente:





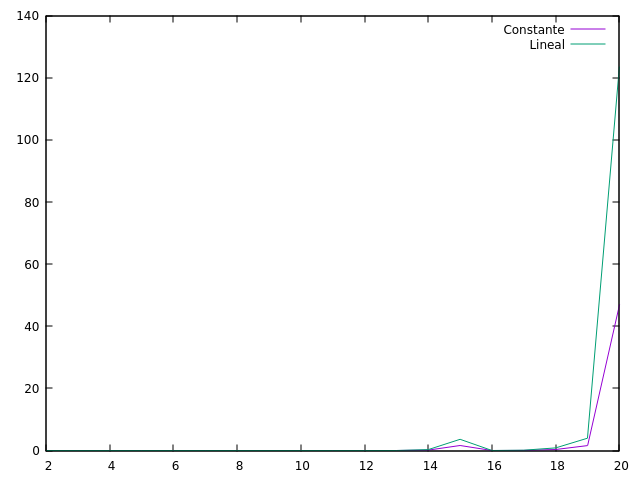
# Factibilidad Constante vs Lineal

La función de factibilidad utilizada es de orden constante, gracias a utilizar dos matrices de valores lógicos que indican en qué filas están ya situados ciertos valores. Sin embargo, el enfoque más directo sería comprobar directamente en la fila y columna deseada la disponibilidad de cierto valor.

Este es el algoritmo de factibilidad lineal:

|  |
| --- |
| bool CuadradoLatino::FactibleLineal(vector<vector<int> > & cuadrado, int i, int j) {  int valor = cuadrado[i][j];  bool factible = true;  for (int k = 0; k < j && factible; k++)  factible = valor != cuadrado[i][k];  for (int k = 0; k < i && factible; k++)  factible = valor != cuadrado[k][j];  return factible;  } |

Esta función más sencilla, sin embargo, convertiría a la función en un orden lineal y empeoraría el algoritmo.Como demostración, aquí está la comparativa de eficiencias entre utilizar la función lineal y la constante:

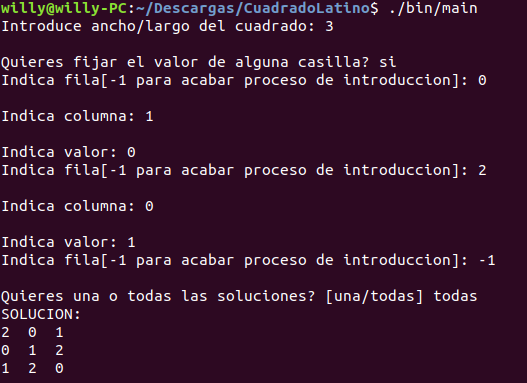


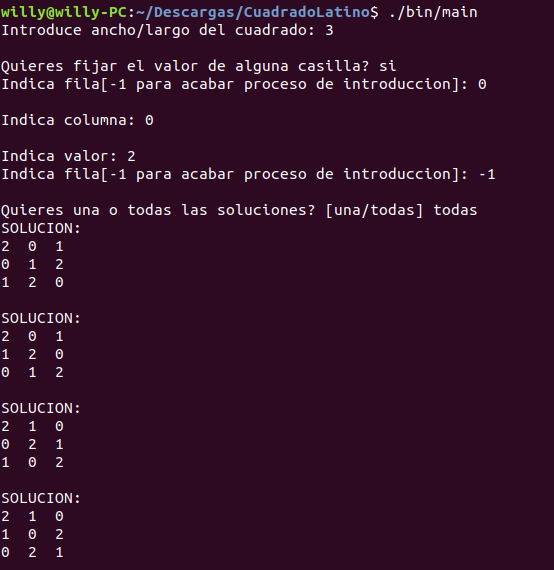
Como se puede comprobar, la función factibilidad de orden lineal es siempre menos eficiente en tiempo. Por ello, es preferible sacrificar cierto espacio de memoria, pero conseguir una función de orden constante.

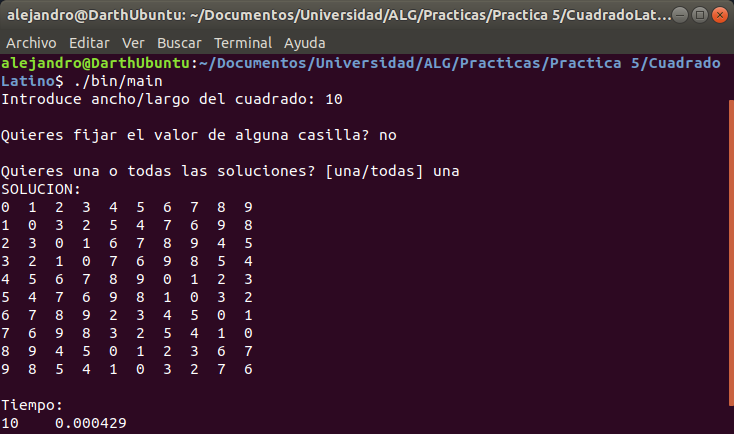
# 

# Ejemplos Prácticos

En este apartado se pueden observar algunos de los resultados obtenidos de la ejecución de este algoritmo:







Como se puede observar los resultados que da son correctos en todos los casos.