***“****Año de la recuperación y consolidación de la economía peruana”*



UNIVERSIDAD NACIONAL DEL ALTIPLANO

**Facultad de Ingeniería Mecánica Eléctrica, Electrónica y Sistemas**

**ESCUELA DE INGENIERÍA DE SISTEMAS**

**Tarea: Practica N°2**

**Curso: SIS210 - Algoritmos y Estructuras de Datos**

**Docente**: Ing. Zanabria Galvez Aldo Hernan **Alumno:** Flores Macedo Anderson Leonardo Código: 236177

**Semestre**: IV Semestre

Puno-Perú 2025

## Resumen

El siguiente informe es una recopilación del trabajo encargado, con el objetivo de profundizar los conocimientos acerca de tiempo de complejidad, como tipos de este (temporal y espacial), se procederá a analizar ambos códigos, desde su funcionamiento, hasta realizando pruebas de distinto tipo

***Análisis Teorico***

## Tiempo de Complejidad

El tiempo de complejidad es el tiempo necesario (no necesariamente segundos) en ejecutarse un algoritmo, determinando que tan eficiente es el código a evaluarse, gracias al tiempo de complejidad, los programadores pueden definir si es eficiente o no su código al resolver su problema-

También se le conoce complejidad computacional, originado en 1965 por Juris Hartmanis y Richard E. Stearns, figuras acerca de este campo, recibió el Premio Turing en 1993 por crear los fundamentos del campo de la complejidad computacional, definiendo varios tipos de complejidad a investigar ahora.

Su notación se representa con la letra *O*(), donde el tiempo aproximado se definirá dentro de

*O*, o también llamada notación de cota superior, asintótica, o notación *O* grande (Big *O*)

Este tipo de complejidad permite, además, de crear una gráfica que permite visualizar como se comporta el algoritmo, estos gráficos pueden ser catalogados, dependiendo de su gráfica, se puede determinar incluso que tipo de algoritmo se ha utilizado para resolver el problema.

Normalmente se usa el tiempo de complejidad con distintos tipos de datos de entrada, prácticamente realizando control de calidad.

Ahora, tenemos distintos lenguajes de programación actualmente, con lo cual se puede realizar algunos trabajos, algunos mas fácilmente que otros, pero todos no pueden escapar del tiempo de complejidad, para ello, procederemos a hacer un análisis profundo al código de Python y C++, en este caso, contando el número de pares en una matriz.

#Importar Librerias

#random Permite generar números aleatorios import random

#time permite acceder al horario del sistema, perfecto para calcular el tiempo

import time

#Se define la función

def generar\_matriz(filas, columnas):

#Retorna una matríz aleatoria con valores del 0 al 100, usando bucles para las columnas y filas

return [[random.randint(0, 100) for \_ in range(columnas)] for \_ in range(filas)]

#Se define la función contar pares, contando los datos de la matriz def contar\_pares(matriz):

#Almacenará los datos del conteo conteo = 0

#Bucle para ir a cada fila de la matriz for fila in matriz:

#Bucle para cada valor de la fila seleccionada for valor in fila:

#Comparar si es par a través de modulo

if valor % 2 == 0: #Agregar valor de conteo

conteo += 1

#Retornar conteo

return ( conteo )

#Definir valores de filas y columnas filas, columnas = 100, 100

#Generar Matriz

matriz = generar\_matriz(filas, columnas) #Grabar un punto temporal

inicio = time.time() #Contar pares de la lista

resultado = contar\_pares(matriz) #Grabar punto temporal

fin = time.time() #Imprimir Pares

print(f"Números pares: {resultado}") #Imprimir Tiempo de ejecución redondeado

print(f"Tiempo de ejecución: {fin - inicio:.6f} segundos")

Código en C++

//Importar Librerias

//iostream para usar consola #include <iostream>

//vector para usar listas #include <vector>

//ctime para el uso de hora del sistema #include <ctime>

#include <cstdlib> using namespace std;+++

//funcion generarMatriz con plantillas

vector<vector<int>> generarMatriz(int filas, int columnas) {

//crear matriz

vector<vector<int>> matriz(filas, vector<int>(columnas));

//recorrer filas de matriz

for (int i = 0; i < filas; ++i)

//recorrer columnas de matriz

for (int j = 0; j < columnas; ++j)

//generar numeros del 1 al 100 matriz[i][j] = rand() % 101;

//retornar matriz return matriz;

}

//funcion contar pares

int contarPares(const vector<vector<int>>& matriz) {

//contador

int conteo = 0;

//Recorrer cada fila de la matriz for (const auto& fila : matriz)

//recorrer cada dato en la fila for (int val : fila)

//verificar si es par

if (val % 2 == 0) ++conteo;

//retornar valor

return conteo;

}

//funcion principal int main() {

//fijar generacion aleatoria por hora srand(time(0));

//definir filas y columnas

int filas = 100, columnas = 100;

//crear matriz

vector<vector<int>> matriz = generarMatriz(filas, columnas);

//inicio pruebas

clock\_t inicio = clock();

//calcular pares

int resultado = contarPares(matriz);

//finalizar prueba clock\_t fin = clock();

//imprimir cantidad pares

cout << "Números pares: " << resultado << endl;

//imprimir tiempo final

cout << "Tiempo de ejecución: " << double(fin - inicio) /

//tiempo final

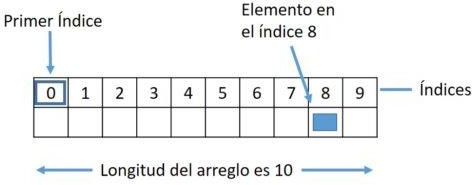
CLOCKS\_PER\_SEC << " segundos\n";

//finalizar funcion retornando 0 return 0;

}

## ¿Que estructura de datos se usa?

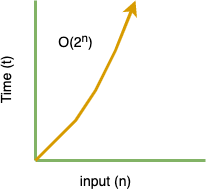
Para ambos ejercicios, se ha usado una estructuras de datos de tipo lineal, es este caso arreglos, o que tienen comportamiento de arreglos, ya que se han definido previamente las dimensiones, y en ningún momento se ha aumentado o disminuido la cantidad de elementos almacenados en ellas.



## Complejidad temporal y espacial

La complejidad temporal ya ha sido definida anteriormente, por ello.

Como se ha mencionado anteriormente, el tiempo de complejidad exponencial se basa en algoritmos que, su tiempo para resolver el problema crece exponencialmente según el tamaño de entrada, siendo en base 2, por ello, el tiempo crece exponencialmente según su entrada.



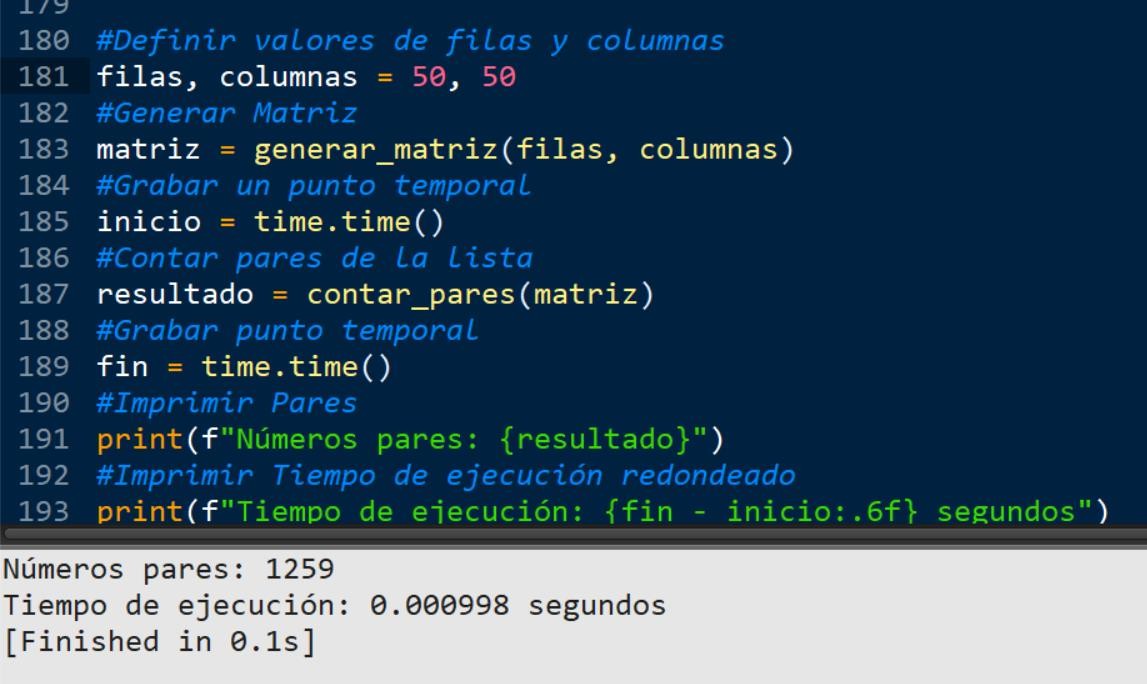
Su tipo de crecimiento es exponencial, denotado por la siguiente expresión:

𝑂(2ⁿ)

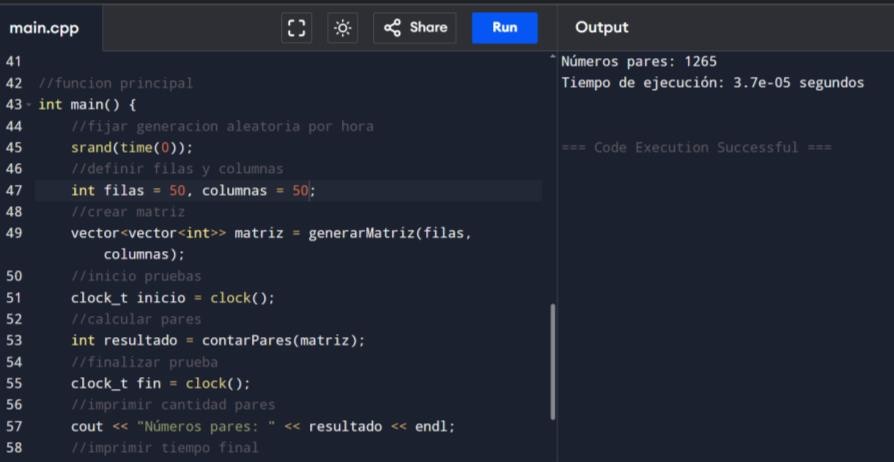
Por qué se define las matrices como tiempo exponencial, tenemos que, primero recorrer todos los datos de esa fila, luego, saltar a la siguiente fila, revisar cada dato en la fila, y así hasta acabar con la matriz.

1. Pruebas:
   1. 50x50

En Python



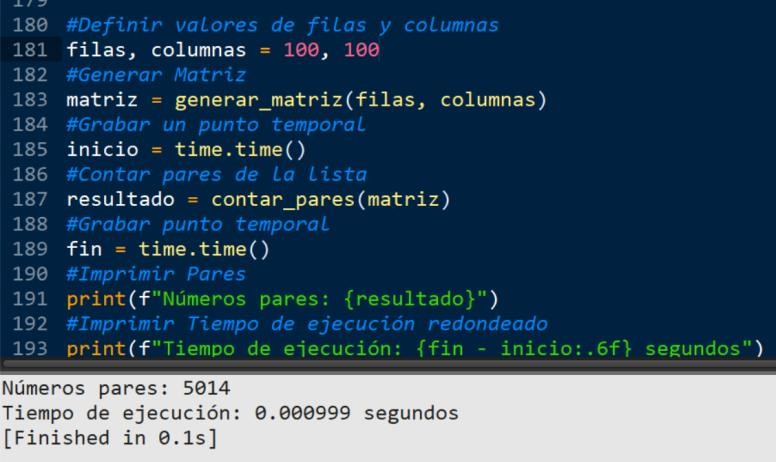
En C++



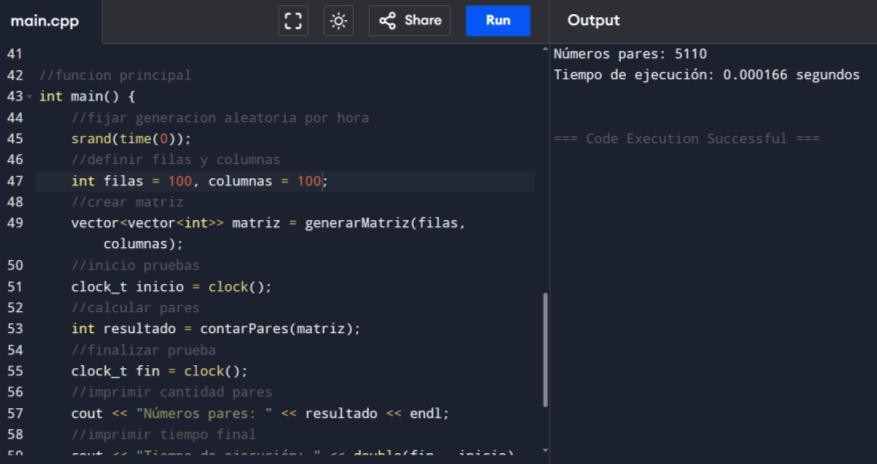
|  |  |  |
| --- | --- | --- |
| Prueba de 50x50 | PYTHON | C++ |
| TIEMPO | 0.000998seg | 3.7e-05seg |

* 1. 100x100

En Python



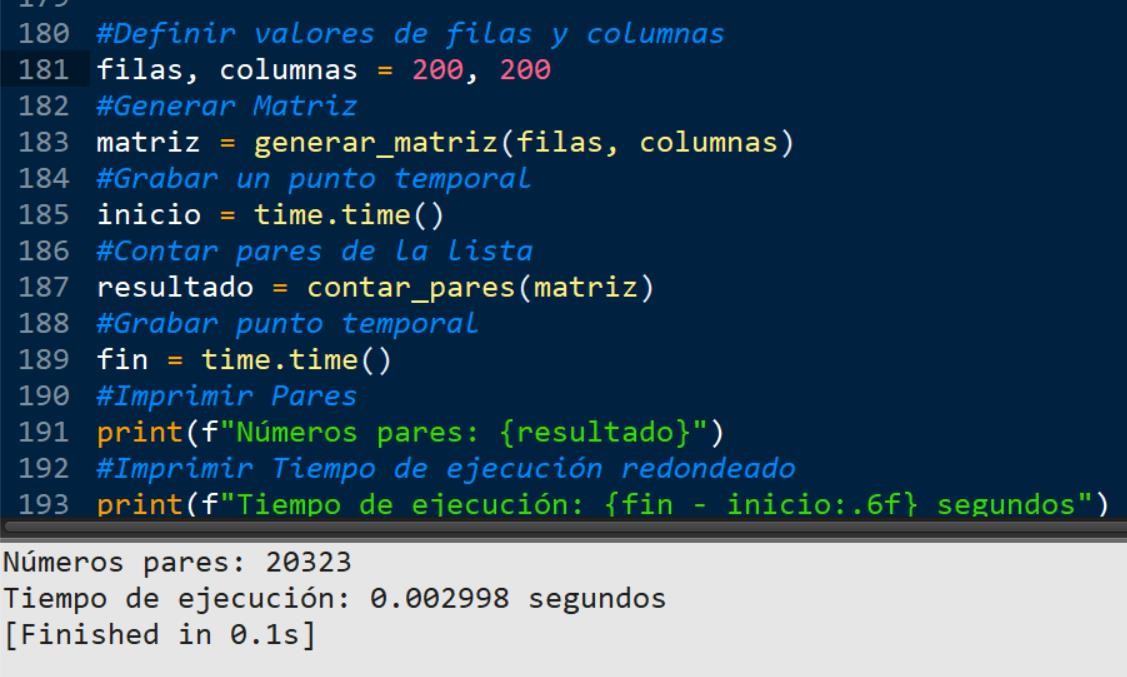
En C++



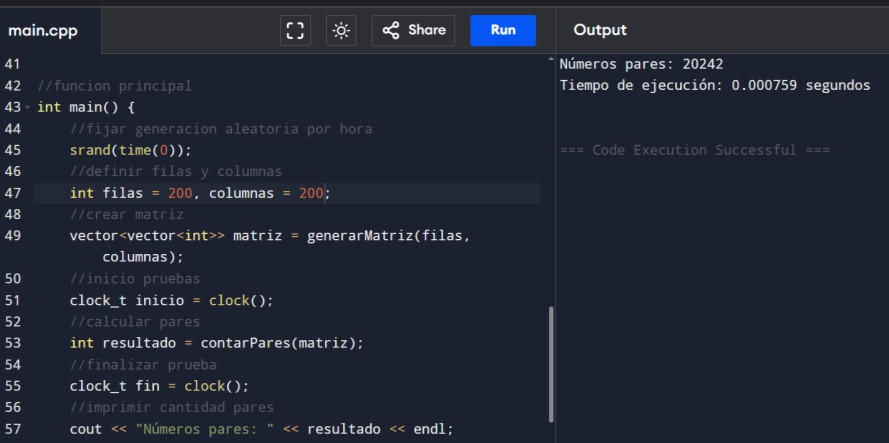
|  |  |  |
| --- | --- | --- |
| Prueba de 100x100 | PYTHON | C++ |
| TIEMPO | 0.000999seg | 0.000166seg |

* 1. 200x200

En Python



En C++



|  |  |  |
| --- | --- | --- |
| Prueba de 200x200 | PYTHON | C++ |
| TIEMPO | 0.002998seg | 0.000759seg |

1. Tiempo de ejecución y explicación

|  |  |  |
| --- | --- | --- |
| Prueba de 50x50 | PYTHON | C++ |
| TIEMPO | 0.000998seg | 3.7e-05seg |
| Prueba de 100x100 | PYTHON | C++ |
| TIEMPO | 0.000999seg | 0.000166seg |
| Prueba de 200x200 | PYTHON | C++ |
| TIEMPO | 0.002998seg | 0.000759seg |

0.0035

0.003

0.0025

0.002

0.0015

0.001

0.0005

0

50x50

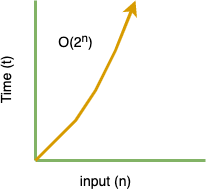
100x100

200x200

Python

C++

Podemos tener una referencia algebraica de como los tiempos crecen exponencialmente, tal y como se refiere el tiempo de complejidad.



A se debe esto, por que el nosotros al momento de crear un arreglo con dimensiones, solo estamos creando espacios multiplicando una dimensión por otra y así n veces, y cada vez que nosotros tengamos que hacer una revisión a cada valor en toda la lista, se tomará una cantina 2 a la n veces. Quedando con la definición la siguiente.

1. Función de números primos

## En Python

#Definición de función para verificar primo def primal\_check( e\_value ):

#Si siempre fue primo, no cambiará truth = True

#Iteración del primer número

for n in range ( 1, e\_value + 1 ):

#Si dividiendo el número, su residuo es 0, y no es ni 1 ni si mismo

if e\_value % n == 1 and n != 0 and n !=

e\_value:

#Nunca fue primo truth = False

#Finalizar bucle break;

#retornar valor return ( truth )

#Definir para contar numeros primos def count\_primal(matriz):

#Almacenará los datos del conteo conteo = 0

#Bucle para ir a cada fila de la matriz for fila in matriz:

#Bucle para cada valor de la fila seleccionada for valor in fila:

realprimal = primal\_check ( valor ) #Comparar si es par a través de modulo

If realprimal == True: #Agregar valor de conteo

conteo += 1

#Retornar conteo

return ( conteo )

## En C++

//Retorna valor booleano la función de verificacion bool primal\_check( int e\_num ){

//Es primo

bool truth = true;

//Verificar si es primo con un bucle

for ( int n = 1; n < e\_num + 1; n++ ){

//Si el número es divisible en 0, pero el número que lo divide no es 1 ni tampoco el mismo, entonces es falso

if ( e\_num % n == 0 && n != 1 && n != e\_num ){ truth = false;

}

}

return ( truth );

}

//contar primos

int contarprimos(const vector<vector<int>>& matriz) {

//contador

int conteo = 0;

//Booleano verificador bool truth = false;

//Recorrer cada fila de la matriz for (const auto& fila : matriz)

//recorrer cada dato en la fila for (int val : fila){

//verificar si es primo

truth = primal\_check ( val );

//agregar al contador if ( truth ) ++conteo;

}

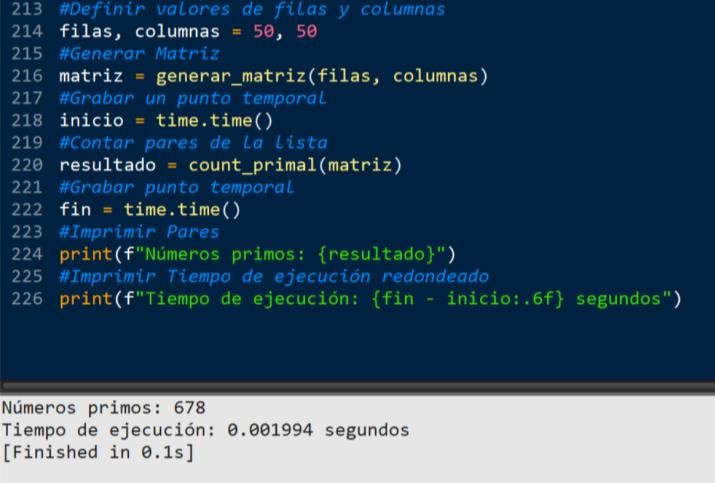
//retornar valor return conteo;

}

Repetimos las mismas pruebas

1. 50x50

En Python



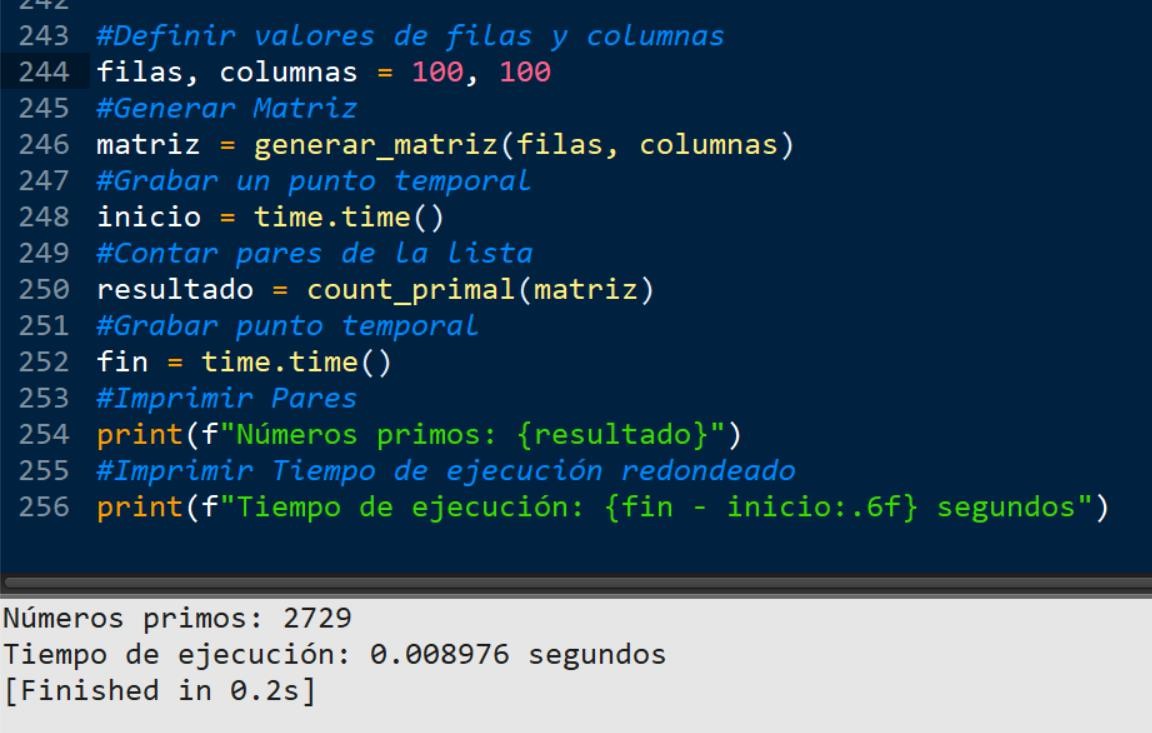
En C++



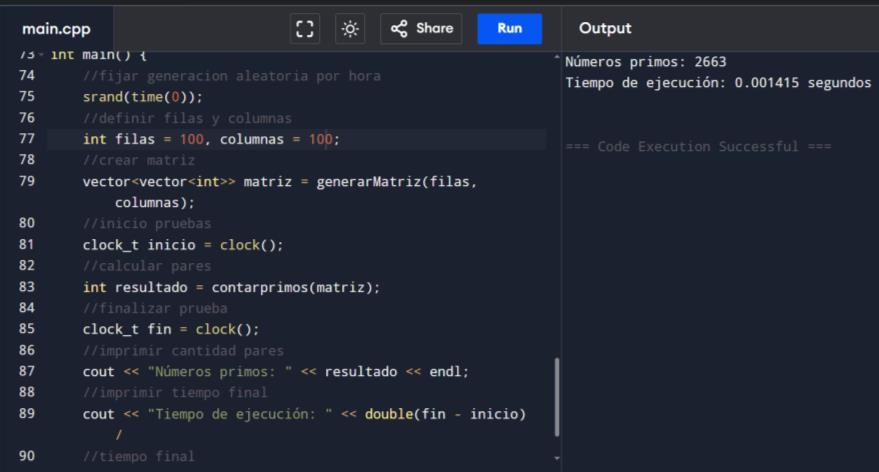
|  |  |  |
| --- | --- | --- |
| Prueba de 50x50 | PYTHON | C++ |
| TIEMPO | 0.001994seg | 0.000336seg |

1. 100x100

En Python



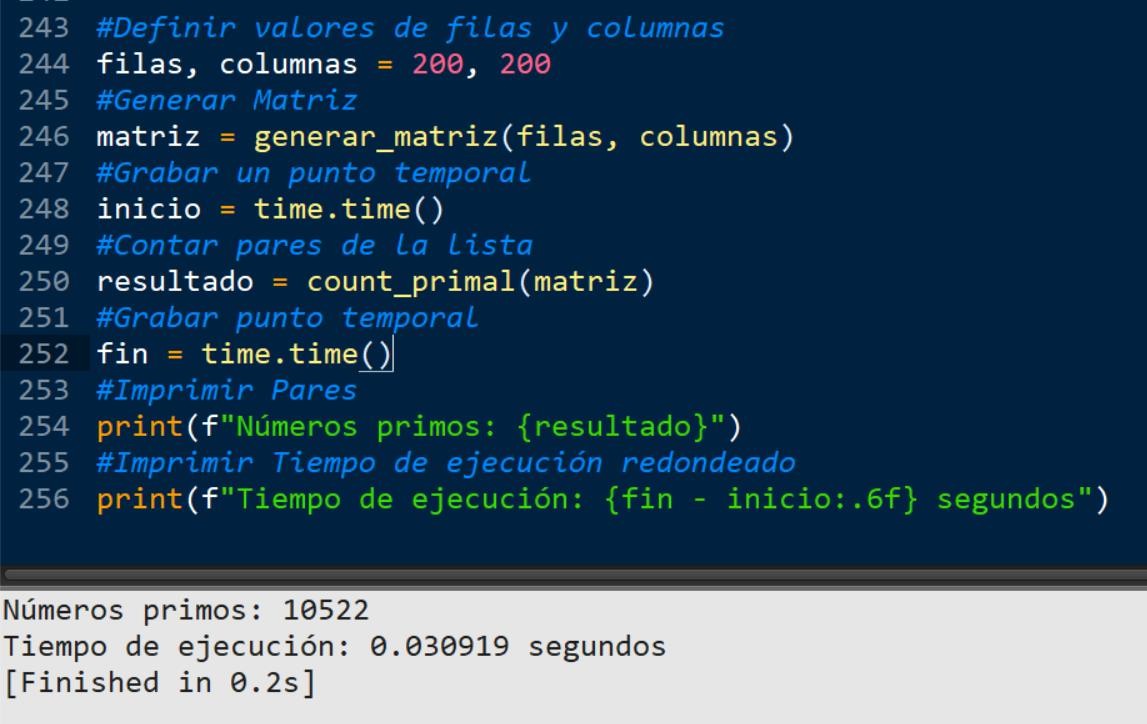
En C++



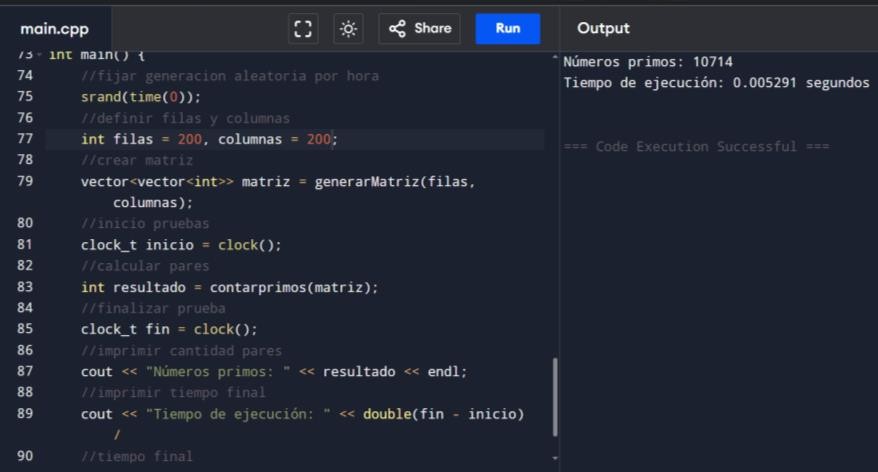
|  |  |  |
| --- | --- | --- |
| Prueba de  100x100 | PYTHON | C++ |
| TIEMPO | 0.008976seg | 0.001415seg |

1. 200x200

En Python



En C++



|  |  |  |
| --- | --- | --- |
| Prueba de  50x50 | PYTHON | C++ |
| TIEMPO | 0.030919seg | 0.005291seg |

0.035

0.03

0.025

0.02

0.015

0.01

0.005

0

50x50

100x100

200x200

Python

C++

1. ¿Cómo optimizarías la verificación de primalidad?

Tras verificar algunos algoritmos y observar que los números que son o no primos constan de divisibilidad, por ello, se puede aprovechar funciones matemáticas para entender que ocurre.

El número puede tener raíz, cuando un número tiene raíz, significa que hay un número que multiplicado por si mismo genera el supuesto número primo, descartándolo.

Tras ello, se debe verificar que no existan otros números, y como se ha comprobado que no tiene raíz, verificamos si existe un número impar que logre dividir el número, en este caso, se trata del algoritmo de Criba de Eratóstenes.

Código en Python

from math import sqrt def esprimo(val):

if val == 2:

return True elif val < 1:

return False

for i in range ( 3, sqrt ( val ), 2 ): if val % i == 0:

return False return True

Código en C++

#include <cmath>

bool esprimo ( int val ){ if ( val == 2 ){

return ( true )

}

else if < 1:

return ( false )

for ( int i = 3, i < sqrt ( val ), i++ ){ if ( val % i == 0 ){

return ( false )

}

}

return ( true )

}

# Enlaces:

C++ y Python: <https://github.com/themackenzie/Algoritmo-y-Estructuras-de-Datos.git>