Documentacion: Optimizacion de algoritmos

Introduccion

El objetivo de esta documentacion es explicar el proceso de optimizacion de algoritmos, mediante l uso de librerias como **numpy** y **matplotlib**

Descripcion del problema

El problema que se pretende resolver es la busqueda de numeros primos en un rango determinado, en este caso del 1 al 100000

El codigo original implementa un algoritmo simple para buscar numeros primos, sin optimizaciones basada en bucles iterativos

Optimizacion

A. reduccion de rango de busqueda

se limito el bucle principal hasta la raiz cuadrada del numero, ya que cualquier numero compuesto tiene al menos un factor primo menor o igual a su raiz cuadrada

B. Uso de numpy

se utilizo numpy para crear un array de booleanos que representa la primalidad de cada numero en el rango, las operaciones se aplican a bloques de memoria contiguos de forma simultanea, lo que permite aprovechar mejor el procesador y acelerar la ejecucion

C. Indexacion booleana

en lugar de usar if para marcar los numeros compuestos, se utiliza indexacion booleana para marcar directamente los indices correspondientes en el array. esto permitio marar todos los multiplos de un numero no primo en una sola operacion.

Tras la ejecución de ambas versiones en el mismo entorno de hardware, se obtuvieron los siguientes tiempos:

Versión del Código	Algoritmo	Tiempo de Ejecución (s)
Original	Fuerza Bruta (Iterativo)	**23.5390 s**
Optimizado	Criba de Eratóstenes (NumPy)	**0.0006 s**

5. Resultados

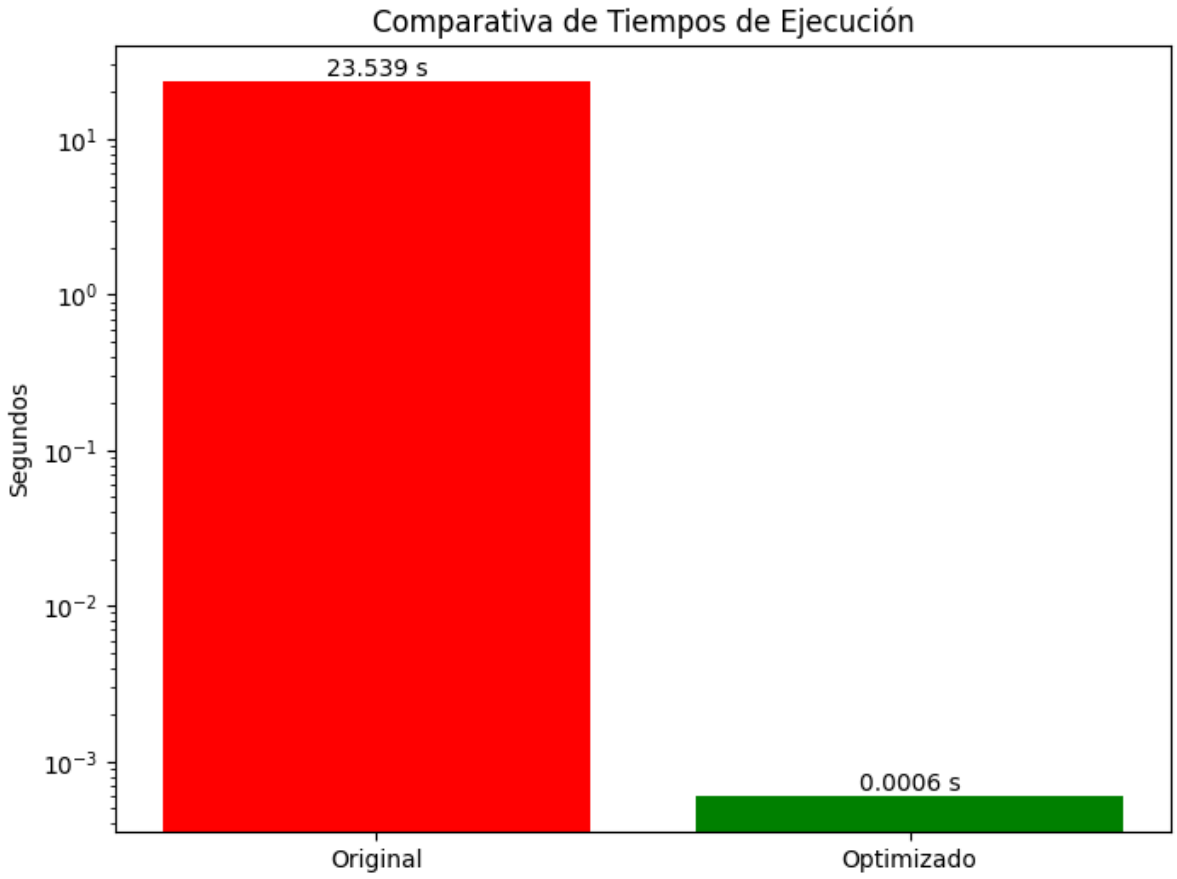
```
## A. comparativa de tiempos de ejecucion
las pruebas se realizaron en el mismo entorno de software
```

Métrica	Código Original (Lento)	Código Optimizado (Rápido)
Tiempo Total	23.5390 s	0.0006 s
Iteraciones	100,000	Vectorizado
Complejidad	$O(n^2)$	$O(n \log \log n)$

```
la optimizacion resulto ser **~39,231 veces más rápida**

## B. Analisis de Profiling

se utilizo la herramientoa 'cProfile' para analizar el codigo original y
optimizado
Nos mostro que el 99% de del tiempo de CPU se gasto en el bucle 'for' interno y en
operaciones aritmeticas repetidas millones de veces
```



Conlcusiones

hemos observado varios beneficios al aplicar optimizaciones al codigo 1. Se logro reducir el tiempo de ejecucion de 23 segundos a 0.0006 segundos, demostrando la importancia de elegir las estructuras correctas 2. Mientras que el código original se volvía inusable al aumentar el rango, la versión vectorizada con NumPy puede manejar millones de datos en tiempos razonables sin bloquear el procesador. 3. Al eliminar millones de

ciclos de CPU innecesarios, el código optimizado consume menos energía, lo cual es vital en entornos de producción en la nube o dispositivos móviles.

6. Recomendaciones

```
1. No se debe adivinar qué parte del código es lenta. usar herramientas como
`cProfile` es indispensable para identificar los verdaderos cuellos de botella
antes de reescribir código.
2. Para operaciones matemáticas intensivas en Python, se recomienda siempre el uso
de **NumPy** o **Pandas** en lugar de bucles nativos, aprovechando su ejecución en
lenguaje C.
3. Si el proyecto requiriera aún más velocidad o bucles personalizados que NumPy
no pueda vectorizar, el siguiente paso lógico sería utilizar librerías como
**Numba**, que compilan funciones de Python a código máquina en tiempo real.
```