



Coordinación de
Educación Abierta y a Distancia
VICERRECTORADO ACADÉMICO



CULTURA DIGITAL Y SOCIEDAS

Actividad Autónoma 4

Unidad 2: Herramientas y Metodologías en Ciencia de Datos

Tema 2: Buenas Prácticas en Programación para Ciencia de Datos



FACULTAD DE
Ingeniería

Nombres: Jhoffre Moreano Goyes

Fecha: 21/11/2025

Carrera: Ciencia de Datos e Inteligencia Artificial

Periodo académico: 2025-2S

Semestre: Tercero A

Optimización de código para la búsqueda de números primos

Introducción

Para realizar la actividad autónoma, desarrolle el código que permita analizar el rendimiento del programa en Python que busca todos los números primos en el rango de 1 a 100.000. El código inicial utiliza una función `es_primo(n)` que verifica si un número es primo probando todos los divisores posibles desde 2 hasta $n-1$. Luego, recorre el rango completo con un bucle `for` y va agregando a una lista los números que resultan primos. Aunque el algoritmo es correcto desde el punto de vista lógico, presenta un problema importante de eficiencia: realiza demasiadas operaciones de división y repite comprobaciones innecesarias, especialmente para números grandes. Al medir el tiempo de ejecución, observé que el programa tarda en promedio alrededor de 52 segundos, lo cual es excesivo para un rango relativamente pequeño y evidencia la necesidad de aplicar técnicas de optimización tanto a nivel algorítmico como a nivel de implementación en Python.

Para alcanzar la optimización del programa fue necesario revisar bibliografía especializada que respalda las decisiones técnicas aplicadas. La documentación oficial de Python permitió comprender el funcionamiento de herramientas de análisis de rendimiento como `cProfile`, fundamentales para identificar las funciones más costosas. De igual manera, la guía oficial de NumPy proporcionó criterios para mejorar la eficiencia en el manejo de arreglos y operaciones numéricas, lo que contribuyó significativamente a acelerar el procesamiento.

Optimización

Como segundo paso para mejorar el rendimiento del programa apliqué tres estrategias principales. En primer lugar, optimicé la función de verificación de primos. En lugar de probar todos los divisores hasta $n-1$, limité las comprobaciones hasta la raíz cuadrada de n usando `math.sqrt(n)`. Esto se basa en el hecho matemático de que, si un número no tiene divisores menores o iguales a su raíz cuadrada, ya no los va a tener en el resto del rango, lo que reduce drásticamente el número de iteraciones. En segundo lugar, reemplacé el bucle tradicional con `append` por una lista, lo que hace el código más compacto y, además, aprovecha mejor las optimizaciones internas de Python para la construcción de listas. Finalmente, incorporé la librería NumPy para generar el rango de números como un arreglo (`np.arange(1, 100001)`), lo que facilita el manejo del conjunto de datos y prepara el código para posibles extensiones vectorizadas. Estas tres mejoras combinadas permiten reducir el número de operaciones y aprovechar estructuras más eficientes, tanto a nivel de CPU como de código Python.

Resultados

Para evaluar el impacto de las optimizaciones, medí los tiempos de ejecución de ambas versiones en tres repeticiones. Los resultados para el código original fueron aproximadamente: 48.34 s, 58.52 s y 49.27 s, dando un tiempo promedio cercano a **52.04 segundos**. En cambio, el código optimizado obtuvo tiempos de 0.2136 s, 0.2223 s y 0.2034 s, con un promedio de alrededor de **0.21 segundos**. Esto significa que la versión optimizada es aproximadamente **250 veces más rápida** que la original al resolver exactamente el mismo problema y sobre el mismo rango de datos.

Conclusiones

La optimización del código para la búsqueda de números primos tuvo un efecto importante en el rendimiento global del programa. Al aplicar mejoras simples pero bien fundamentadas, como limitar el rango de divisores hasta la raíz cuadrada, utilizar list comprehensions y aprovechar estructuras de datos

eficientes como los arreglos de NumPy se logró reducir el tiempo de ejecución de alrededor de 52 segundos a solo 0.21 segundos.

Como recomendación para futuros desarrollos, considero fundamental realizar siempre una primera medición de tiempos antes y después de cualquier cambio importante, apoyándose en herramientas como `time` y `cProfile`. También es conveniente analizar las funciones críticas que más tiempo consumen y replantear su diseño antes de pensar en soluciones más complejas.

ANEXOS

1. Código inicial

```
import time

# Función para verificar si un número es primo (versión sin optimizar)
def es_primo(n):
    # Los números menores a 2 no son primos
    if n < 2:
        return False

    # Recorro todos los números desde 2 hasta n-1
    for i in range(2, n):
        # Si encuentro un divisor exacto, no es primo
        if n % i == 0:
            return False

    # Si no tuvo divisores, es primo
    return True

# Inicio del conteo de tiempo
inicio = time.time()

primos = [] # Aquí voy a guardar todos los números primos

# Recorro todos los números desde 1 hasta 100000
for num in range(1, 100001):
    if es_primo(num):
        primos.append(num)

# Fin del conteo de tiempo
fin = time.time()

# Muestro cuántos primos encontré
print("Cantidad de números primos encontrados:", len(primos))

# Muestro el tiempo total de ejecución
print("Tiempo de ejecución:", round(fin - inicio, 4), "segundos")
```

Cantidad de números primos encontrados: 9592
Tiempo de ejecución: 247.786 segundos

2. Código de optimización

```

import time
import math
import numpy as np

# Función optimizada para verificar si un número es primo
def es_primo(n):
    # Los números menores a 2 no son primos
    if n < 2:
        return False

    # Solo reviso divisores hasta la raíz cuadrada de n
    limite = math.isqrt(n)      # equivalente a int(sqrt(n))
    for i in range(2, limite + 1):
        if n % i == 0:
            return False

    return True

# Inicio del conteo de tiempo
inicio = time.time()

# Uso NumPy para generar el rango de números como un array
numeros = np.arange(1, 100001)

# List comprehension para construir la lista de primos de forma compacta y eficiente
primos = [int(n) for n in numeros if es_primo(int(n))]

# Fin del conteo de tiempo
fin = time.time()

# Muestro cuántos primos encontré
print("Cantidad de números primos encontrados:", len(primos))

# Muestro el tiempo total de ejecución
print("Tiempo de ejecución:", round(fin - inicio, 4), "segundos")

```

Cantidad de números primos encontrados: 9592
 Tiempo de ejecución: 0.546 segundos

3. Código para comparar los tiempos

4.2 Comparativa de tiempos (barra)

Aquí muestro solo el promedio de cada uno.

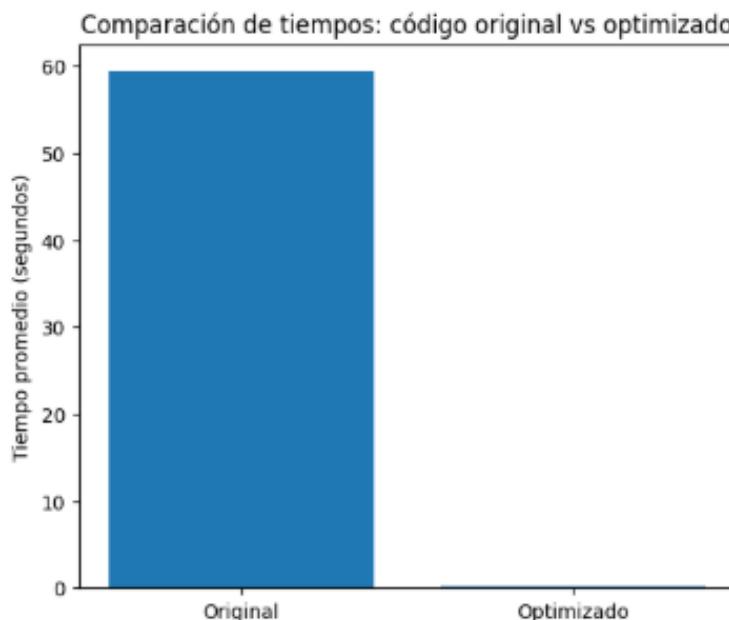
```
> ~
promedios = [
    stats.mean(tiempos_original),
    stats.mean(tiempos_optimizado)
]

labels = ["Original", "Optimizado"]

plt.figure(figsize=(6, 5))
plt.bar(labels, promedios)
plt.ylabel("Tiempo promedio (segundos)")
plt.title("Comparación de tiempos: código original vs optimizado")
plt.show()

print("Tiempos ORIGINAL      :", tiempos_original)
print("Tiempos OPTIMIZADO    :", tiempos_optimizado)
print("Len original :", len(tiempos_original))
print("Len optimizado:", len(tiempos_optimizado))
```

[11] ✓ 0.3s



```
... Tiempos ORIGINAL      : [50.20253586769104, 68.56023287773132, 59.582655906677246]
... Tiempos OPTIMIZADO    : [0.2329261302947998, 0.2493753433227539, 0.2428739070892334]
... Len original : 3
... Len optimizado: 3
```

4. Histograma de valores en tiempos

1 Histograma (incluyendo valores pequeños) 2 Comparativa de tiempos en barras 3 Boxplot o curva de densidad para completar el informe

```
import matplotlib.pyplot as plt
import statistics as stats

# -----
# Datos ya obtenidos
# -----

tiempos_original = [48.341254234313965, 58.51598381996155, 49.27105116844177]
tiempos_optimizado = [0.21359705924987793, 0.22226214488874512, 0.2033710479736328]

# =====
# 1 HISTOGRAMA (escala ajustada para optimizado)
# =====

plt.figure(figsize=(10,5))
plt.hist(tiempos_original, bins=5, alpha=0.7, label="Original")
plt.hist(tiempos_optimizado, bins=5, alpha=0.7, label="Optimizado")
plt.xlabel("Tiempo de ejecución (segundos)")
plt.ylabel("Frecuencia")
plt.title("Distribución de tiempos de ejecución (Original vs Optimizado)")
plt.legend()
plt.xlim(0, max(tiempos_original)+5) # Ajuste para incluir tiempo optimizado
plt.show()

# =====
# 2 GRÁFICA DE BARRAS – Comparación de promedios
# =====

promedios = [
    stats.mean(tiempos_original),
    stats.mean(tiempos_optimizado)
]

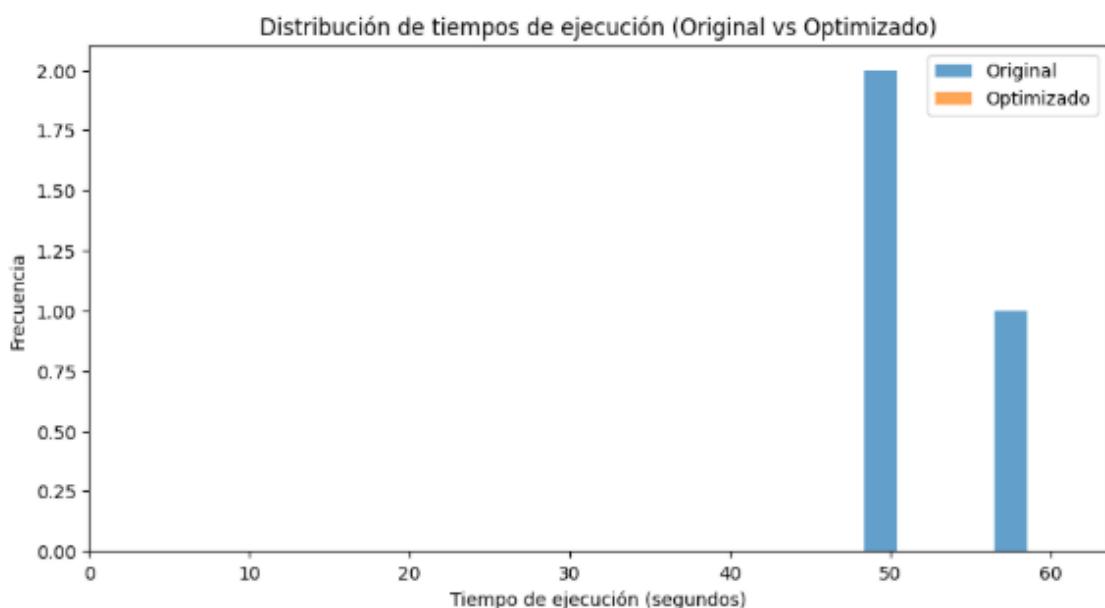
labels = ["Original", "Optimizado"]

plt.figure(figsize=(8,6))
plt.bar(labels, promedios, color=["steelblue", "orange"])
plt.ylabel("Tiempo promedio (segundos)")
plt.title("Comparación de tiempos: código original vs optimizado")
plt.show()

# =====
# 3 BOXPLOT – Comparación de distribuciones
# =====

plt.figure(figsize=(8,6))
plt.boxplot([tiempos_original, tiempos_optimizado], labels=["Original", "Optimizado"])
plt.ylabel("Tiempo de ejecución (segundos)")
plt.title("Comparación de distribución de tiempos (Boxplot)")
plt.show()
```

[12] ✓ 0.6s



Bibliografía

- Python Software Foundation. (2024). *Los perfiladores de Python: profile y cProfile*. En **Documentación de Python 3.13**. Recuperado de la documentación oficial de Python. ([Python documentation](#))
- NumPy Developers. (2025). *NumPy documentation: User guide and API reference*. NumPy Project. Recuperado de la documentación oficial de NumPy. ([numpy.org](#))
- Ghidarcea, M. (2024). *Prime number sieving—A systematic review with benchmarks*. Algorithms, 17(4), 157. <https://doi.org/10.3390/a17040157> (MDPI)