Trabajo Práctico Integrador

Análisis de algoritmos

Alumnos

Leandro Zenocratti y Santiago Almiron

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación 1

Docente Titular

Nicolás Quirós

Docente Tutor

Flor Gubiotti

Entrega

9 de Junio de 2025

Tabla de contenido

Sección	Página
1. Portada	1
2. Tabla de Contenido	2
3. Introducción	3-4
4. Marco Teórico	4
- 4.1 Concepto de Primalidad	4
- 4.2 Complejidad Computacional	4
- 4.3 Notación Asintótica (Big-O)	5
5. Análisis de Algoritmos	5-10
- 5.1 Algoritmo de Fuerza Bruta	5-7
- 5.2 Algoritmo Optimizado	8-9
- 5.3 Criba de Eratóstenes Adaptada	10
6. Comparación de Resultados	9-10
7. Conclusiones	11-12
- 7.1 Hallazgos Clave	11
- 7.2 Reflexión Final	12
8. Bibliografía	12

Trabajo Práctico integrador

Introducción

El presente trabajo práctico se centra en el análisis de algoritmos para determinar si un número es primo, un problema aparentemente sencillo que permite ilustrar cómo las decisiones de diseño iniciales pueden tener un impacto significativo en el rendimiento y la escalabilidad de una solución. Este estudio demuestra que una implementación correcta pero poco optimizada puede volverse rápidamente ineficiente, mientras que un análisis teórico previo ayuda a prevenir problemas costosos en etapas avanzadas del desarrollo.

Para llevar a cabo este análisis, se implementaron tres algoritmos con niveles crecientes de sofisticación:

Enfoque inicial: Una solución directa y literal, que verifica todos los divisores posibles de manera similar a un procedimiento manual.

Versión optimizada: Una mejora sustancial basada en principios matemáticos, que reduce drásticamente el espacio de búsqueda.

Criba de Eratóstenes: Un algoritmo clásico de la teoría de números, incluido a modo de referencia pero no analizado en profundidad, ya que en las primeras evaluaciones se evidenció su ineficiencia para el problema específico de verificación de primalidad de un único número.

A lo largo del trabajo, no solo se comparan los tiempos de ejecución empíricos, sino que también se realiza un análisis teórico detallado de cada implementación, identificando sus puntos críticos y fundamentando matemáticamente las optimizaciones aplicadas. Este proceso permite comprender cómo pequeños cambios en el diseño algorítmico pueden generar diferencias significativas en el rendimiento, especialmente al trabajar con entradas de gran tamaño.

Link al video en youtube

Link al repositorio de la actividad

Marco teórico

Concepto de Primalidad:

Un número primo es un número natural mayor que 1 que tiene exactamente dos divisores positivos distintos: 1 y de sí mismo. Aunque la definición es simple, su verificación computacional plantea desafíos significativos, especialmente al trabajar con números grandes.

Complejidad Computacional:

El análisis de algoritmos se basa en la notación asintótica (Big-O) para evaluar su eficiencia en tiempo y espacio. En este trabajo, se comparan tres enfoques con distintas complejidades, aunque en el material no se encuentra como notación √n, matemáticamente √n es equivalente a $n^{\frac{1}{2}}$ que si es una notación asintótica que aparece.

Algoritmo	Complejidad Temporal	Explicación
Fuerza Bruta	O(n)	Verifica divisibilidad desde 2 hasta *n-1*.
Optimizado (√n)	O(√n)	Reduce las comprobaciones hasta √n, basado en propiedades matemáticas.

Detalle de cada enfoque:

Fuerza Bruta (O(n)):

- Implementación directa de la definición.
- Inviable para números grandes (ej. n > 10^6), ya que el tiempo crece linealmente con *n*.

Optimizado (O(√n)):

- Fundamento matemático: Si *n* es compuesto, al menos uno de sus divisores es ≤ √n.
- Reduce el espacio de búsqueda drásticamente.

Notación Asintótica:

Origen y Relevancia:

La notación Big-O fue introducida por el matemático alemán Paul Bachmann en 1894 en su obra Analytische Zahlentheorie y popularizada por Edmund Landau en trabajos posteriores. Esta notación se ha convertido en el estándar para el análisis del peor caso de algoritmos en ciencias de la computación, permitiendo comparar la eficiencia de diferentes soluciones de manera abstracta e independiente de factores hardware o implementación específica.

Definición:

Para una función *f(n)*, se dice que *f(n)* es *O(g(n))* si existe una constante positiva *c* y un valor n_0 tal que:

$$f(n) \leq c \cdot g(n) \text{ para todo } n \geq n_0.$$

Este marco teórico sustenta el análisis empírico posterior, justificando las diferencias de rendimiento observadas.

Caso práctico

Análisis de los Algoritmos Implementados:

1. Algoritmo de Fuerza Bruta

Base teórica: Implementa directamente la definición matemática, verificando todos los posibles divisores.

Link al algoritmo.

Análisis teórico:
El peor caso posible es cuando se deben recorrer todos los números y el ciclo nunca encuentra un valor que permita determinar que el número no es primo.

```
def is_prime(n):
    if n <= 1:                  # T(n)= 1
        return 'No es primo'
    for i in range(2, n):       # T(n) = n - 2
        if n % i == 0:          # T(n)= 1
            return 'No es primo'
    return 'Es primo'           # T(n)= 1
```

Entonces: T(n)= 1+(n-2)*(1)+1 = n
La complejidad asintótica de is_prime(n) es O(n).

Limitaciones:

- Para n = 15,485,863 requiere ≈15 millones de operaciones.
- El tiempo de ejecución crece linealmente con el tamaño de entrada.
- Inviable para números mayores a 10^6 ya que a partir de este punto el proceso demora varios segundos para una simple comprobación, inaceptable si debe formar parte de procesos más grandes y complejos donde deba usarse varias veces.

Análisis práctico:

Escribimos un método para cada algoritmo que ejecuta una serie de pruebas del código usando otra función que creamos para medir el tiempo de ejecución del algoritmo con entradas cada vez más grandes.

Este método se puede ver en el link que dejamos más arriba.

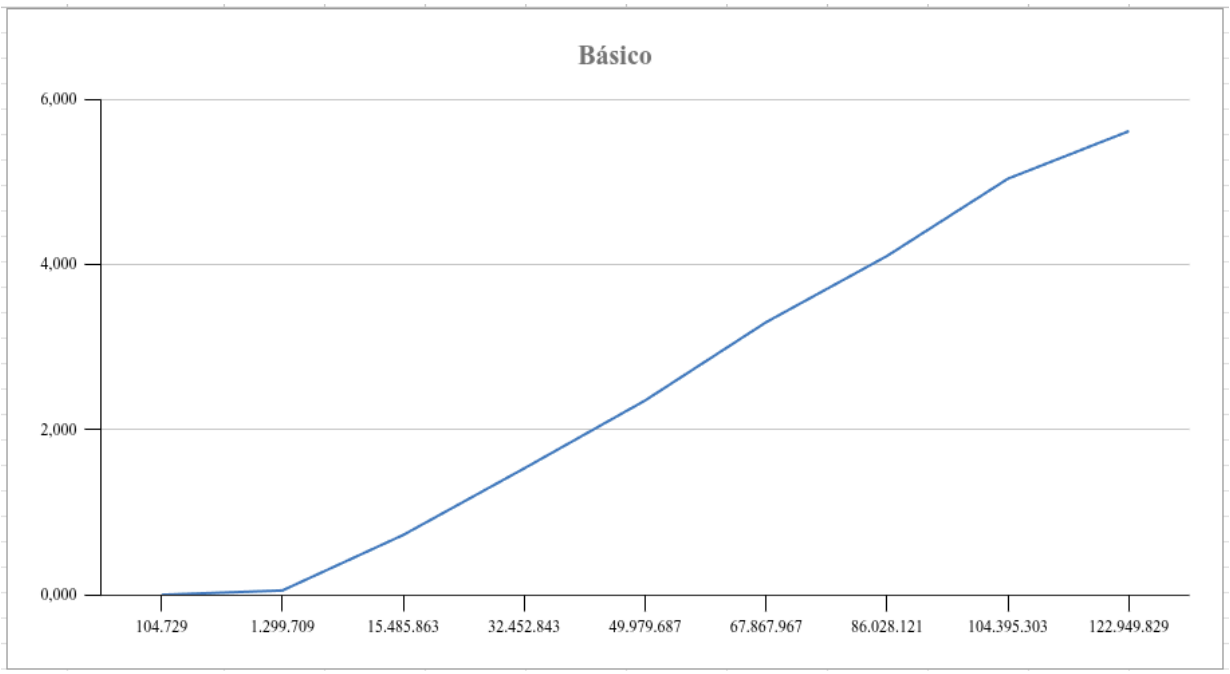
Al ejecutar la prueba con los siguientes números:

104.729, 1.299.709, 15.485.863, 32.452.843, 49.979.687, 67.867.967, 86.028.121, 104.395.303, 122.949.829, 141.650.939

podemos ver que nos devuelve los siguientes resultados.

```
Resultado: Es primo, Tiempo de ejecución: 0.004341 segundos
Resultado: Es primo, Tiempo de ejecución: 0.056779 segundos
Resultado: Es primo, Tiempo de ejecución: 0.728710 segundos
Resultado: Es primo, Tiempo de ejecución: 1.533631 segundos
Resultado: Es primo, Tiempo de ejecución: 2.355112 segundos
Resultado: Es primo, Tiempo de ejecución: 3.300970 segundos
Resultado: Es primo, Tiempo de ejecución: 4.102409 segundos
Resultado: Es primo, Tiempo de ejecución: 5.036804 segundos
Resultado: Es primo, Tiempo de ejecución: 5.611835 segundos
Resultado: Es primo, Tiempo de ejecución: 6.232944 segundos
```

Graficando:



Vemos que se cumple el pronóstico del análisis técnico de que la gráfica crece de manera lineal. En el eje de las X vemos los números que usamos y en el eje de las Y vemos el tiempo expresado en segundos.

2. Algoritmo Optimizado

Optimizaciones clave:

Límite superior en √n (basado en: si n = a×b, entonces a o b ≤ √n)

Salteo de pares después de verificar divisibilidad por 2

Link al algoritmo

Análisis teórico:

El peor camino posible es cuando no entra en ninguno de los if del principio y el número es primo

```
def is_prime(n):
  if n <= 1:                          # T(n)= 1
    return 'No es primo'
  if n == 2:                          # T(n)= 1
    return 'Es primo'
  if n % 2 == 0: # (pares descartados)  # T(n)= 1
    return 'No es primo'
  square_root = isqrt(n)              # T(n) = 2

  for i in range(3, square_root + 1, 2):  # T(n)= 1/2√n
    if n % i == 0:                    # T(n)= 1
      return 'No es primo'
  return 'Es primo'                   # T(n)= 1 (return)
```

Entonces: $T(n)=1+1+1+2+(\frac{1}{2}\sqrt{n}) * (1) + 1 = \frac{1}{2}\sqrt{n} + 6$

Mejoras observadas:

Reducción de O(n) a O(√n)

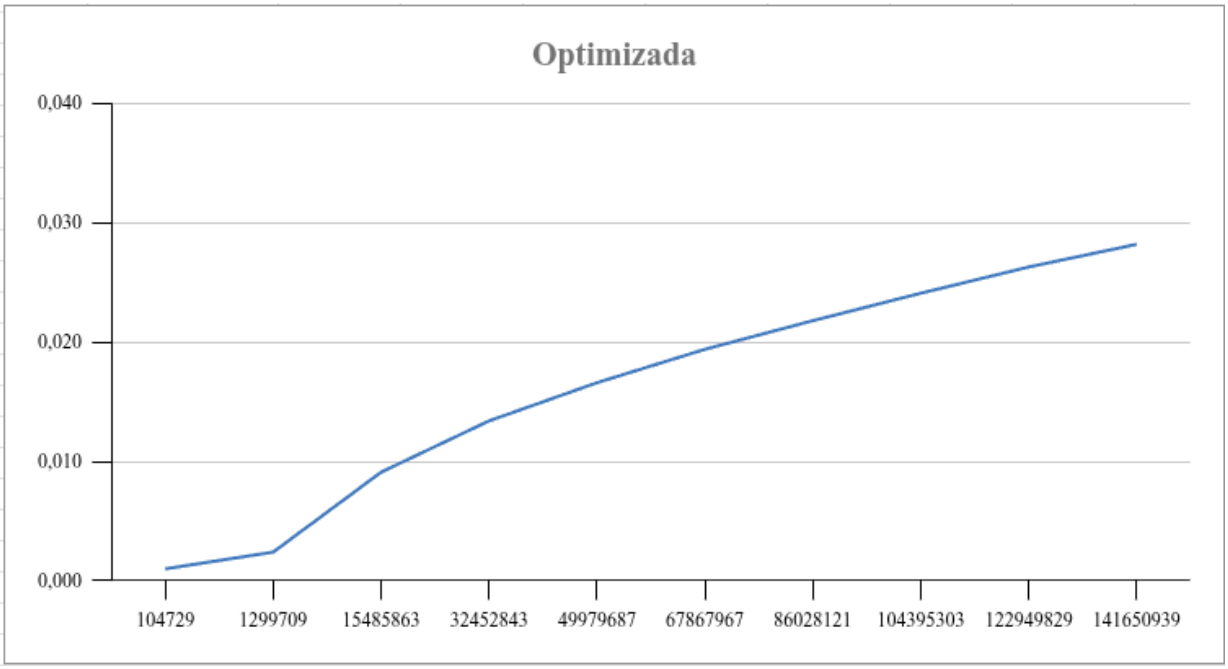
Para n = 15,485,863, el algoritmo original (O(n)) haría ≈15.5 millones de operaciones.

Con el algoritmo optimizado (O(√n)), se realizan aproximadamente 1,967 operaciones, lo que representa aproximadamente el 0.0127% del total original.

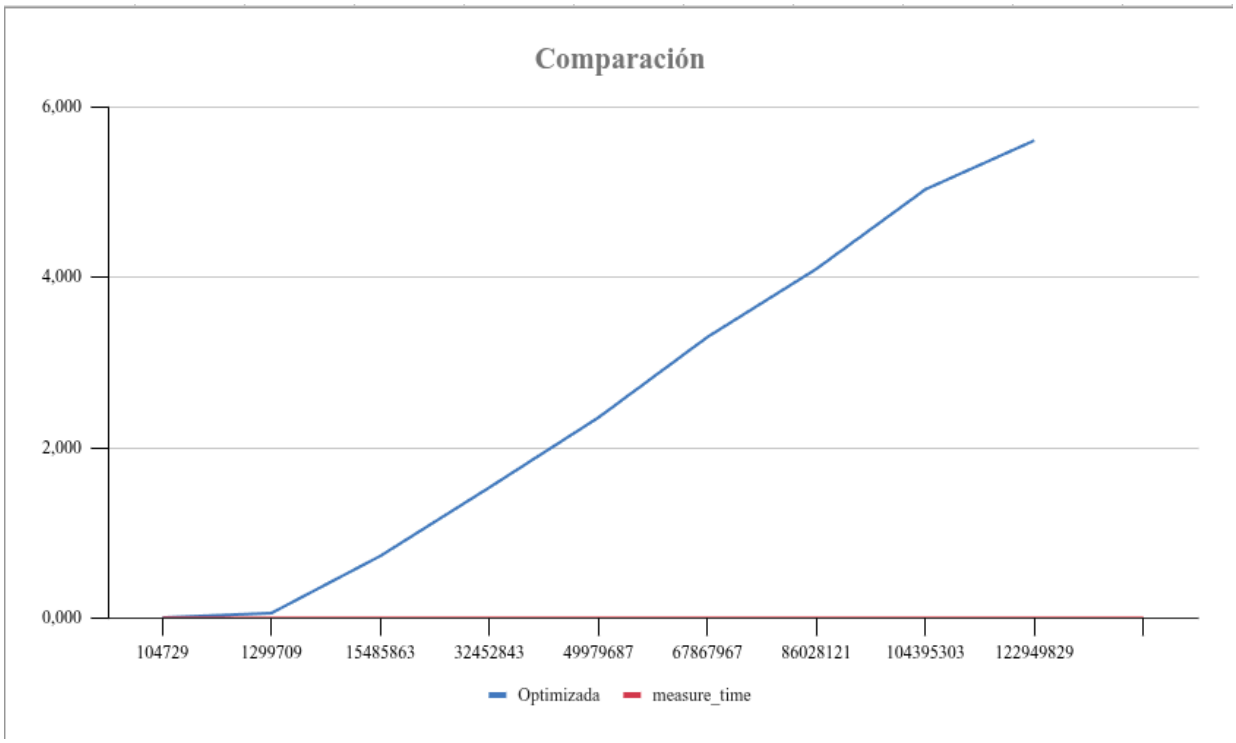
Al ejecutar la prueba con los mismos números vemos que podemos corroborar los resultados que obtuvimos con el análisis teórico, vemos que ninguno de los números demora más de 0.001 segundos.

```
Resultado: Es primo, Tiempo de ejecución: 0.000010 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000024 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000091 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000134 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000166 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000194 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000218 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000241 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000263 segundos
Resultado: Es primo, Tiempo de ejecución: 0.000282 segundos
```

Al graficar debemos ajustar la escala del eje Y, para que se aprecie la forma que describe la función que corroboramos se comporta como la función raíz de n.



Al ponerlas en una gráfica conjunta podemos notar que el algoritmo optimizado se comporta como una constante y el tiempo de ejecución es despreciable en comparación al algoritmo original.



Exploración adicional:

Criba de Eratóstenes Adaptada:

La criba de Eratóstenes es un algoritmo que en sí mismo está pensado para encontrar todos los números primos que existen entre 2 y un número *n* dado inicialmente. Esto ya nos planteó un problema en sí ya que antes de implementar una lógica que nos permita determinar si un número es efectivamente primo, debemos hacer cálculos para filtrar aquellos números que efectivamente son primos de aquellos que no, de una lista compuesta por *n* elementos. Este análisis previo fué suficiente para descartar la idea de entrada pero quisimos explorar un poco ayudándonos de algunos videos de youtube y de la IA para nombrarla como un elemento más que fue interesante en el desarrollo de este trabajo práctico.

El código implementado se puede ver en Github.

Resultados y Conclusiones: El trabajo práctico integró un análisis detallado de tres algoritmos para determinar la primalidad de un número, destacando cómo las decisiones de diseño impactan en el rendimiento. Los resultados obtenidos fueron contundentes y revelaron diferencias significativas entre los enfoques evaluados.

Hallazgos Clave: Optimización matemática: La implementación del algoritmo optimizado, basado en la reducción del espacio de búsqueda hasta la raíz cuadrada del número (√n), demostró ser una mejora drástica frente al método de fuerza bruta. Mientras que el algoritmo básico requería millones de operaciones para números grandes, la versión optimizada redujo este número a apenas unas miles, lo que se tradujo en tiempos de ejecución insignificantes (menos de 0.001 segundos en todos los casos probados). Esto confirmó que pequeñas optimizaciones basadas en propiedades matemáticas pueden generar ganancias de rendimiento exponenciales.

Contexto de uso: El estudio evidenció que la elección del algoritmo depende en gran medida del escenario. Por ejemplo, la criba de Eratóstenes, aunque eficiente para generar listas de números primos, resultó inadecuada para verificar la primalidad de un único número debido a su alto costo computacional en este contexto. Este hallazgo subraya la importancia de seleccionar herramientas adecuadas para cada problema específico.

Balance entre recursos: La versión optimizada logró un equilibrio ideal entre tiempo de ejecución y consumo de memoria, sin requerir estructuras de datos complejas ni almacenamiento adicional. Esto la convierte en una solución práctica y escalable, incluso para números muy grandes.

Reflexión Final: El trabajo dejó en claro que las expectativas iniciales no siempre coinciden con la realidad. Por ejemplo, la criba de Eratóstenes, que en teoría parecía prometedora, terminó siendo descartada tras un análisis preliminar. Este proceso ilustró la importancia de profundizar en el problema antes de implementar soluciones, así como de validar hipótesis mediante pruebas empíricas y teóricas.

En resumen, el estudio no sólo permitió comparar algoritmos, sino también entender cómo el diseño inicial puede marcar la diferencia entre una solución viable y una ineficiente. Las optimizaciones basadas en principios matemáticos simples, como limitar las comprobaciones a √n, demostraron ser suficientes para lograr un rendimiento sobresaliente, evitando la necesidad de soluciones más complejas. Finalmente, los resultados reforzaron la idea de que, en programación, a veces menos es más.

Bibliografía y recursos utilizados:

¿Qué es un número primo y cómo calcularlo? Video de youtube

Propuesta del algoritmo optimizado: link

Criba de Eratóstenes: Wikipedia

Análisis de algoritmos y Big O: material de la cátedra