.UBAfiuba
FACULTAD DE INGENIERÍA

TRABAJO PRÁCTICO 0

TEORÍA DE ALGORITMOS

(75.29 / 95.06 / TB024)

Cuatrimestre: 1º Cuatrimestre 2025

Fecha de entrega: 22 / 03 / 2025

Curso: 03 - Echevarría

Brian Céspedes	**108219**

Índice

1. PRIMERA SOLUCIÓN PROPUESTA

1.1. Supuestos

- El algoritmo funciona para cualquier valor positivo de n, pero idealmente se debe usar un valor mayor o igual a 12 para obtener resultados útiles.
- Para optimizar la búsqueda de números primos, podemos aplicar algunas consideraciones, planteando 3 casos base:
  - Números menores a 2 (no son primos).
  - El número 2 (primo).
  - El número 3 (primo).
- Luego del análisis anterior, podemos seguir buscando números primos del 5 en adelante. Verificando en incrementos de 6 en 6, ya que después del 3, los números primos se encuentran como $6k \pm 1$ (por ejemplo: $23 = 6 \cdot 4 - 1$. Otro ejemplo: $97 = 6 \cdot 16 + 1$). En cada iteración, comprobaremos el valor $6k - 1$ y $6k + 1$.
- Basta con encontrar divisores hasta la raíz cuadrada del número, ya que, si dicho número tiene un divisor mayor a su raíz, podemos garantizar que también tendrá un divisor menor a ella.
- Para buscar cuartetos de números primos, las candidatas son las decenas que son múltiplos de 30 al restarle 10 (por ejemplo la decena 40, 70, 100, etc). Los cuartetos están, como mínimo, a una distancia de 30 unidades a partir de la decena 10. Esto ocurre debido a que los números primos encontrados no pueden ser múltiplos ni de 2, ni de 3 ni de 5, por su neta definición ($2 \cdot 3 \cdot 5 = 30$).
- Inicialmente, para el punto anterior se había planteado un incremento de a 10 (únicamente nos importan las decenas), pero, en la práctica, se observó que de una decena a otra había una distancia que siempre era múltiplo de 30, por ejemplo, el cuarto cuarteto (821, 823, 827, 829) y el tercer cuarteto (191, 193, 197, 199) se encuentran en las decenas 820 y 190 respectivamente. 820 - 190 = 630, y a su vez, 630 es divisible por 30. Este patrón se repite en la totalidad de los cuartetos obtenidos, por lo que el salto de a decenas se puede optimizar aún más para ser realizado de a 3 decenas.

1.2. Diseño de la Solución

- Pseudocódigo de la función **es_primo()**:

```
es_primo(número_a_analizar):

      SI número_a_analizar < 2 ENTONCES:
            RETORNAR Falso
      SI número_a_analizar ES 2 O 3 ENTONCES:
            RETORNAR Verdadero
      SI número_a_analizar % 2 == 0 O número_a_analizar % 3 == 0 ENTONCES:
            RETORNAR Falso

      límite_divisor = raíz_entera(numero_a_analizar) + 1

      DESDE i = 5 HASTA límite_divisor CON UN INCREMENTO DE 6:
            SI número_a_analizar % i == 0 O número_a_analizar % (i + 2) == 0 ENTONCES:
                  RETORNAR Falso

      RETORNAR Verdadero
```

- Pseudocódigo de la función principal:

```
encontrar_cuartetos_de_números_primos(límite):

      tiempo_inicial = time()
      cantidad_de_cuartetos = 0
      resultados = lista_de_strings[]

      DESDE i = 11 HASTA límite CON UN INCREMENTO DE 30:

            primer_candidato = i
            segundo_candidato = i + 2
            tercer_candidato = i + 6
            cuarto_candidato = i + 8

            SI es_primo(primer_candidato) Y es_primo(segundo_candidato)
            Y es_primo(tercer_candidato) Y es_primo(cuarto_candidato) ENTONCES:
                  AGREGAR A resultados("{primer_candidato} - {segundo_candidato}
                  - {tercer_candidato} - {cuarto_candidato}")
                  cantidad_de_cuartetos += 1

      tiempo_final = time()
      tiempo_de_ejecución = tiempo_final - tiempo_inicial
      IMPRIMIR resultados, cantidad_de_cuartetos

      RETORNAR tiempo_de_ejecución
```

- Implementación en código:

```
def es_primo(numero_a_analizar: int) -> bool:

	if numero_a_analizar < 2:
		return False
	if numero_a_analizar in (2, 3):
		return True
	if numero_a_analizar % 2 == 0 or numero_a_analizar % 3 == 0:
		return False

	limite_divisor: int = isqrt(numero_a_analizar) + 1

	for i in range(5, limite_divisor, 6):
		if numero_a_analizar % i == 0 or numero_a_analizar % (i + 2) == 0:
			return False

	return True


def encontrar_cuartetos_de_numeros_primos(limite: int) -> float:

	tiempo_inicial: float = time()
	cantidad_de_cuartetos: int = 0
	resultados: list = []

	for i in range(11, limite, 30):

		primer_candidato: int = i
		segundo_candidato: int = i + 2
		tercer_candidato: int = i + 6
		cuarto_candidato: int = i + 8

		if es_primo(primer_candidato) and es_primo(segundo_candidato)
		and es_primo(tercer_candidato) and es_primo(cuarto_candidato):
			resultados.append(f"{primer_candidato} - {segundo_candidato}
			- {tercer_candidato} - {cuarto_candidato}")
			cantidad_de_cuartetos += 1

	tiempo_final: float = time()
	tiempo_de_ejecucion: float = tiempo_final - tiempo_inicial
	utilidades.imprimir_resultados(resultados, cantidad_de_cuartetos)

	return tiempo_de_ejecucion
```

Se puede observar en la implementación que se utiliza una lista para imprimir los cuartetos obtenidos. Se hizo porque en la práctica se observó una leve mejora en los tiempos de ejecución al realizarlo de esta manera.

1.3. Análisis de Complejidad

Para hacer un análisis de la complejidad temporal, podemos enfocarnos en cada función por separado:

- En la función **es_primo()** hay un ciclo el cual itera desde 5 hasta la raíz cuadrada del número a analizar / 6. El resto de operaciones en esta función es constante, O(1). El número a analizar en cada iteración tiende a ser igual al número límite.
  - **Complejidad resultante:** $O\left(\frac{\sqrt{n}}{6}\right) = O\left(\sqrt{n}\right)$.
- En la **función principal**, nuevamente tenemos un ciclo, esta vez uno que se ejecuta desde 11 hasta límite (n) / 30. Dentro del ciclo, se llama a la función **es_primo()** que ya analizamos y tiene una complejidad $O\left(\sqrt{n}\right)$. La complejidad de imprimir los cuartetos de números primos es O(m) siendo m la cantidad de elementos en la lista de resultados (típicamente, m < n). El resto de operaciones en esta función es constante, O(1).
  - **Complejidad resultante (total):** $O\left(\frac{n}{30}\right) \cdot O\left(\sqrt{n}\right) = O\left(n \cdot \sqrt{n}\right)$.

1.4. Seguimiento

Haremos un seguimiento del programa de las primeras 2 decenas en las cuales se encuentra un cuarteto, el resto de decenas se evalúan de manera análoga:

- Inicialmente, se analiza la primera decena (del 11 al 19): el primer candidato vale 11, el segundo candidato 13, el tercer candidato 17 y el cuarto candidato 19. Los 4 candidatos resultan ser primos (luego de ser evaluados en la función **es_primo()** pasan todos los filtros), guardamos los números e incrementamos en 1 la cantidad.
- La segunda iteración se centra en la decena del 41 al 49 (se descarta el 40, pues los números pares no tienen sentido ser analizados). El primer candidato vale 41, el segundo 43, el tercero 47 y el cuarto 49. Los 3 primeros candidatos son números primos, pero, es inmediato notar que el cuarto candidato no es primo. En este caso, el número 49, al pasar por la función **es_primo()** no va a pasar el filtro ya que 49 % 7 == 0.
- La tercera iteración se centra en la decena del 71 al 79: el primer candidato es 71, el segundo 73, el tercero 77 y el cuarto 79. En este caso, todos los números son primos, exceptuando el tercer candidato, que nuevamente es divisible por 7.
- La cuarta iteración se centra en la decena del 101 al 109: los candidatos son 101, 103, 107 y 109. Análogamente a la primera iteración, los 4 candidatos resultan primos, por lo que se almacenan y nuevamente se incrementa la cantidad de cuartetos.

1.5. Tiempos de Ejecución

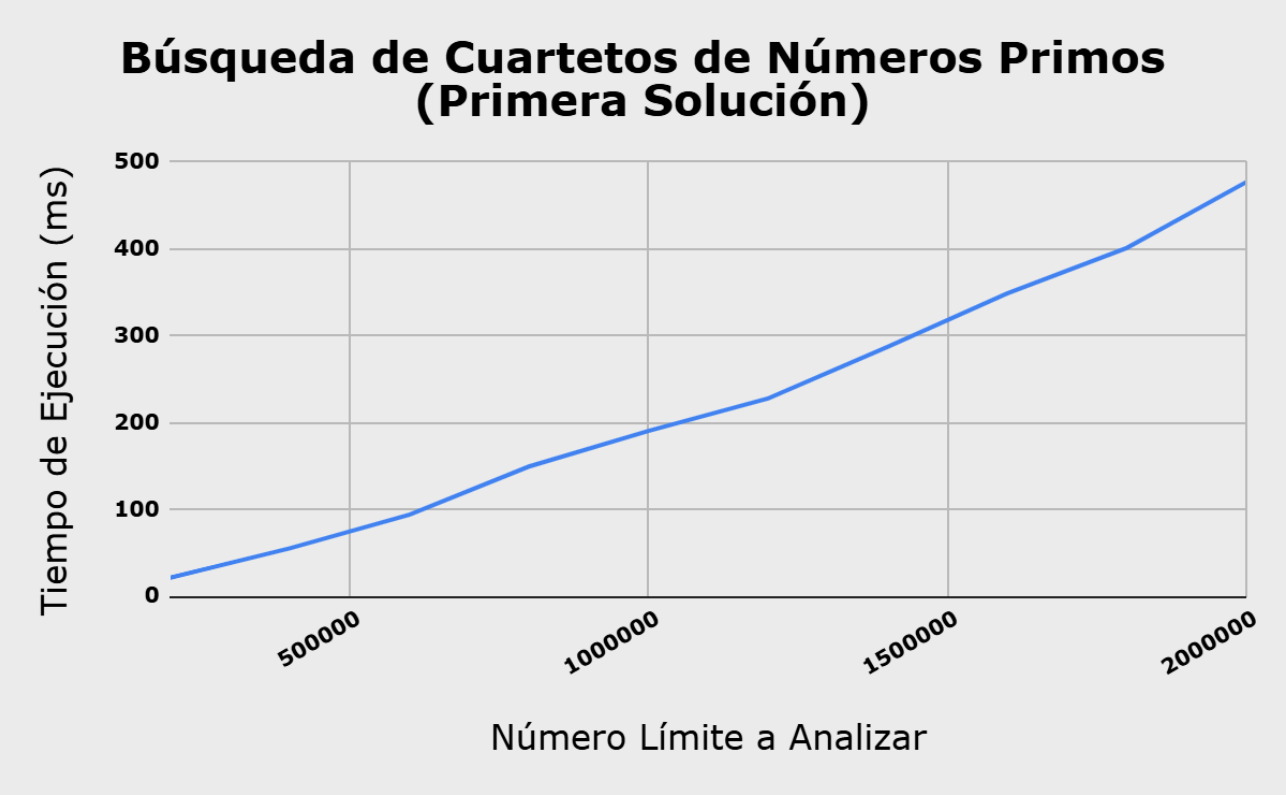

Gráfico que relaciona el valor de n con el tiempo de ejecución *

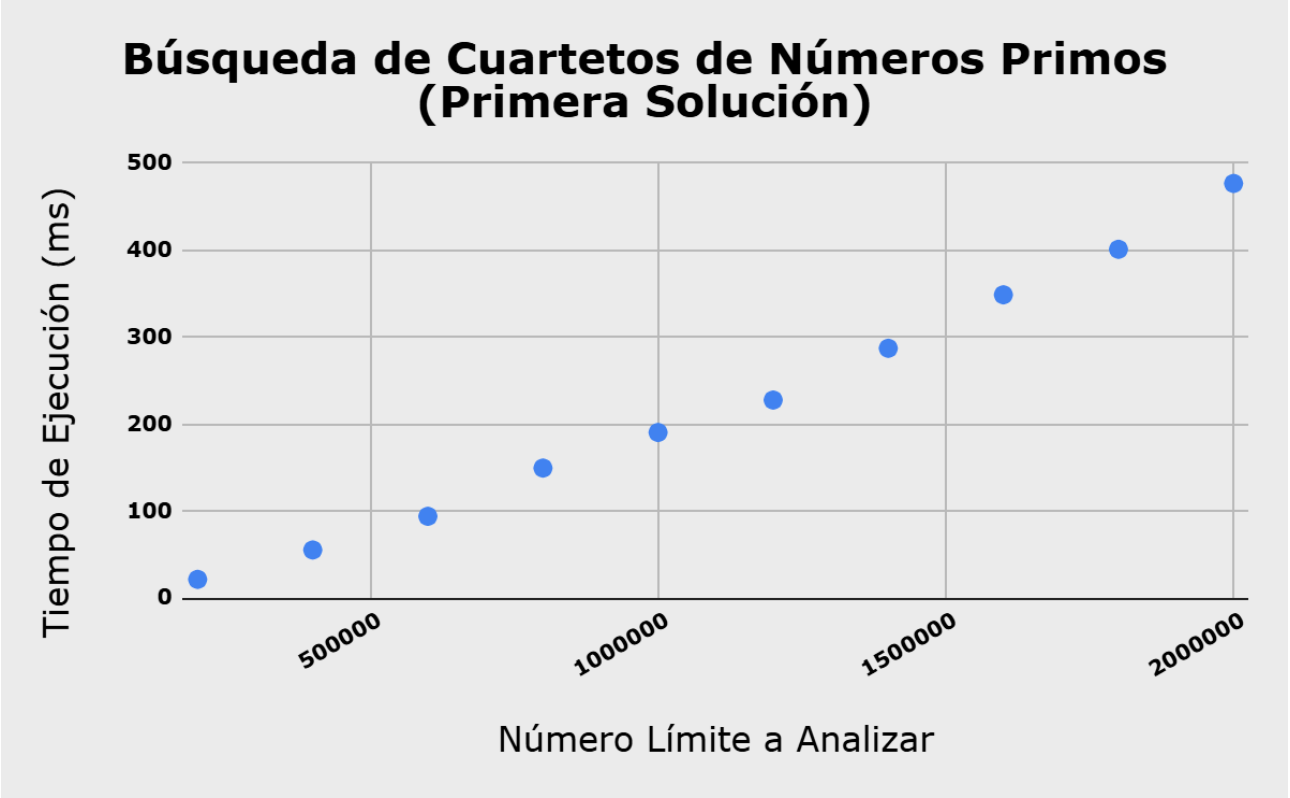

Gráfico de dispersión

** Datos obtenidos analizando el promedio, luego de 1000 ejecuciones en Windows 11 (ver apartado **Instrucciones de Ejecución**). El procesador de la máquina en la cual se realizó la prueba es un Intel i5 1235U, 1.3GHz, 16GB de RAM.*

Se observa en los gráficos anteriores que los mismos se condicen con la complejidad temporal analizada previamente. Si bien la complejidad teórica es $O\left(n \cdot \sqrt{n}\right)$, en la práctica, con las distintas optimizaciones aplicadas se reducen mucho los tiempos de ejecución, por eso podemos ver una tendencia menos marcada a escalar con valores de n mayores.

A continuación, se expone la tabla con los datos obtenidos:

Valor de n	Tiempo de Ejecución (ms)
200000	21,942
400000	55,734
600000	94,400
800000	149,828
1000000	190,735
1200000	227,984
1400000	287,445
1600000	348,905
1800000	401,164
2000000	476,836

2. SEGUNDA SOLUCIÓN PROPUESTA (ERATÓSTENES)

2.1. Supuestos

- El algoritmo funciona para cualquier valor positivo de n, pero idealmente se debe usar un valor mayor o igual a 20 para obtener resultados útiles.
- La criba de Eratóstenes tiene un proceso diferente a la anterior solución. Se basa en hallar primero todos los primos hasta cierto número n.
- Se asume que disponemos del espacio en memoria adecuado, ya que esta solución utiliza un vector de n posiciones. Esto quiere decir que la cantidad de memoria utilizada será equivalente al número límite hasta el cual se desean obtener los números primos.
- Basta con encontrar divisores hasta la raíz cuadrada del número, ya que, si dicho número tiene un divisor mayor a su raíz, podemos garantizar que también tendrá un divisor menor a ella.
- Para buscar cuartetos de números primos, las candidatas son las decenas que son múltiplos de 30 al restarle 10 (por ejemplo la decena 40, 70, 100, etc). Los cuartetos están, como mínimo, a una distancia de 30 unidades a partir de la decena 10. Esto ocurre debido a que los números primos encontrados no pueden ser múltiplos ni de 2, ni de 3 ni de 5, por su neta definición ($2 \cdot 3 \cdot 5 = 30$).
- Inicialmente, para el punto anterior se había planteado un incremento de a 10 (únicamente nos importan las decenas), pero, en la práctica, se observó que de una decena a otra había una distancia que siempre era múltiplo de 30, por ejemplo, el cuarto cuarteto (821, 823, 827, 829) y el tercer cuarteto (191, 193, 197, 199) se encuentran en las decenas 820 y 190 respectivamente. 820 - 190 = 630, y a su vez, 630 es divisible por 30. Este patrón se repite en la totalidad de los cuartetos obtenidos, por lo que el salto de a decenas se puede optimizar aún más para ser realizado de a 3 decenas.

2.2. Diseño de la solución

- Pseudocódigo de la función **criba_de_eratóstenes()**:

```
criba_de_eratóstenes(límite):

      límite_divisor = raíz_entera(límite) + 1
      límite += 1
      INICIAR primos COMO UN BYTE ARRAY DE TAMAÑO límite CON TODOS LOS VALORES EN Verdadero
      primos[0] = Falso
      primos[1] = Falso

      DESDE i = 2 HASTA límite_divisor:
            SI primos[i] ENTONCES:
                  cuadrado = i ** 2
                  DESDE j = cuadrado HASTA límite CON UN INCREMENTO DE i:
                        primos[j] = Falso

      RETORNAR primos
```

- Pseudocódigo de la función principal:

```
encontrar_cuartetos_de_números_primos(límite):

      tiempo_inicial = time()
      primos = criba_de_eratóstenes(límite)
      cantidad_de_cuartetos = 0
      resultados = lista_de_strings[]
      límite -= 8

      DESDE i = 11 HASTA límite CON UN INCREMENTO DE 30:

            primer_candidato = i
            segundo_candidato = i + 2
            tercer_candidato = i + 6
            cuarto_candidato = i + 8

            SI primos[primer_candidato] Y primos[segundo_candidato]
            Y primos[tercer_candidato] Y primos[cuarto_candidato] ENTONCES:
                  AGREGAR A resultados("{primer_candidato} - {segundo_candidato}
                  - {tercer_candidato} - {cuarto_candidato}")
                  cantidad_de_cuartetos += 1

      tiempo_final = time()
      tiempo_de_ejecución = tiempo_final - tiempo_inicial
      IMPRIMIR resultados, cantidad_de_cuartetos

      RETORNAR tiempo_de_ejecución
```

- Implementación en código:

```
def criba_de_eratostenes(limite: int) -> bytearray:

	limite_divisor: int = isqrt(limite) + 1
	limite += 1
	primos: bytearray = bytearray([True]) * limite
	primos[0] = False
	primos[1] = False

	for i in range(2, limite_divisor):
		if primos[i]:
			cuadrado = i ** 2
			primos[cuadrado:limite:i] = [False] * len(range(cuadrado, limite, i))

	return primos


def encontrar_cuartetos_de_numeros_primos(limite: int) -> float:

	tiempo_inicial: float = time()
	primos: bytearray = criba_de_eratostenes(limite)
	cantidad_de_cuartetos: int = 0
	resultados: list = []
	limite -= 8

	for i in range(11, limite, 30):

		primer_candidato: int = i
		segundo_candidato: int = i + 2
		tercer_candidato: int = i + 6
		cuarto_candidato: int = i + 8

		if primos[primer_candidato] and primos[segundo_candidato]
		and primos[tercer_candidato] and primos[cuarto_candidato]:
			resultados.append(f"{primer_candidato} - {segundo_candidato}
			- {tercer_candidato} - {cuarto_candidato}")
			cantidad_de_cuartetos += 1

	tiempo_final: float = time()
	tiempo_de_ejecucion: float = tiempo_final - tiempo_inicial
	utilidades.imprimir_resultados(resultados, cantidad_de_cuartetos)

	return tiempo_de_ejecucion
```

Se puede observar en la implementación que se utiliza un array de bytes para almacenar los booleanos al realizar la criba. Se hizo para mayor eficiencia, tanto temporal como espacial. Además, se utiliza una lista para imprimir los cuartetos obtenidos. En la práctica se observó una leve mejora en los tiempos de ejecución al realizarlo de esta manera.

2.3. Análisis de Complejidad

Para analizar la complejidad del algoritmo, podemos hacerlo por separado, analizando cada función con su complejidad asociada.

- La función **criba_de_eratostenes()** requiere de un análisis especial para determinar su complejidad, ya que se necesita de cierta rigurosidad matemática y aplicación de propiedades.
  - El primer ciclo, se ejecuta desde 2 hasta la raíz cuadrada del número límite que estamos analizando. El resto de operaciones dentro del ciclo se ejecutan en tiempo constante, O(1). La complejidad resultante es $O\left(\sqrt{n}\right)$.
  - El segundo ciclo requiere de un análisis mucho más exhaustivo. Al encontrar un número primo, luego se marcan todos sus múltiplos, por ejemplo, el primer número primo es 2, inmediatamente después marcamos a todos sus múltiplos (los números pares) desde 2^2 hasta el valor n.

    El número de múltiplos que tiene el 2 es $\frac{n}{2}$. Análogo a lo anterior, el número de múltiplos que tiene el 3 (segundo número primo) es $\frac{n}{3}$.

    Si continuamos desarrollando, veremos que la cantidad de operaciones totales para marcar a todos los múltiplos de números primos se puede hallar como $\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \ldots + \frac{n}{p}$, siendo p el último número primo menor o igual a n. Del cálculo anterior, factorizamos $n \cdot \left(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \ldots + \frac{1}{p}\right)$.

    La suma dentro del paréntesis es asintóticamente equivalente a $log(log(n))$ *(ver **referencias**)*. Finalmente, luego de nuestro análisis podemos concluir que la complejidad en este ciclo es de $O(n \cdot log(log(n)))$.

    Aunque el primer ciclo, según observamos, tiene una complejidad de $O(\sqrt{n})$, el segundo ciclo es el que realmente determina la complejidad total del algoritmo. La complejidad resultante de la criba de Eratóstenes es $O(n \cdot log(log(n)))$.

- En la **función principal**, inicialmente llamamos a la función **criba_de_eratostenes()** la cual su complejidad ya fue analizada. Luego, tenemos un ciclo que se ejecuta desde 11 hasta límite (n) / 30. La complejidad de imprimir los cuartetos de números primos es O(m) siendo m la cantidad de elementos en la lista de resultados (típicamente, m < n). El resto de operaciones en esta función es constante, O(1).
  - Complejidad **resultante:** $O(n \cdot log(log(n))) + O\left(\frac{n}{30}\right) = O(n \cdot log(log(n)))$.

2.4. Seguimiento

Haremos un seguimiento del programa de las primeras dos decenas (propongo un valor de límite = 110) en las cuales se encuentra un cuarteto, el resto de decenas se evalúan de manera análoga:

- Inicialmente, debemos realizar el proceso de criba para hallar todos los números primos hasta límite = 110. Se crea una lista de 111 elementos (incluyendo al cero), con todos los elementos en True. Se asigna False al valor 0 y al valor 1 (0 y 1 no son primos).
- Al entrar en el ciclo, el primer número con el que nos encontramos es el 2, calculamos su cuadrado (4). Como el 2 es primo (primos[2] = True), se entra en el segundo ciclo, en el cual marcamos todos los números múltiplos de 2 (números pares a partir del 4) como False (no son primos).
- En la segunda iteración, nos hallaremos con el 3, calculamos su cuadrado (9). Como el 3 es primo (primos[3] = True), se entra en el segundo ciclo, en el cual marcamos todos los múltiplos de 3 (a partir del 9, dando incrementos de a 3) como False (no son primos).
- En la tercera iteración, nos hallaremos con el 4 que no es un número primo (primos[4] = False, el cual fue marcado al analizar primos[2]). Pasamos a la próxima iteración.
- En la cuarta iteración, nos encontramos con el 5, calculamos su cuadrado (25). Como el 5 es primo (primos[5] = True), se entra en el segundo ciclo, en el cual marcamos todos los múltiplos de 5 (a partir del 25, dando incrementos de a 5) como False (no son primos).
- En la siguiente iteración que nos interesa (sexta), nos encontramos con el 7, calculamos su cuadrado (49). Como el 7 es primo (primos[7] = True), se entra en el segundo ciclo, en el cual marcamos todos los múltiplos de 7 (a partir del 49, dando incrementos de a 7) como False (no son primos).
- En la siguiente iteración que nos interesa (décima), nos encontramos con el 11, calculamos su cuadrado (121). 121 es un número mayor a límite (110), no se vuelve a ingresar más al segundo ciclo a partir de este punto.
- Si hacemos un análisis aún más exhaustivo de las iteraciones anteriores, veremos que la lista primos[] termina valiendo True en las posiciones: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107 y 109.
- Ya con la lista de primos, iteramos desde 11 hasta límite - 8 (102), con incrementos de a 30. En la primera iteración, el primer candidato vale 11, el segundo candidato vale 13, el tercer candidato vale 17 y el cuarto candidato vale 19. Los 4 números son primos y se agregan a la lista de resultados, además de aumentar la cantidad de cuartetos en 1.
- En la segunda iteración el primer candidato vale 41, el segundo vale 43, el tercero vale 47 y el cuarto vale 49. primos[41] = True, primos[43] = True, primos[47] = True, pero primos[49] = False, entonces resulta que no es un cuarteto.
- En la tercera iteración, el primer candidato vale 71, el segundo 73, el tercero 77 y el cuarto 79, primos[71] vale True, primos[73] vale True, primos[77] vale False y primos[79] vale True. Como primos[77] es False, entonces tampoco es un cuarteto.
- En la cuarta iteración, los candidatos son 101 103, 107 y 109. La lista primos[] en esas 4 posiciones vale True, entonces es un cuarteto y se agrega a la lista de resultados, además de incrementar en 1 la cantidad de cuartetos.

2.5. Tiempos de Ejecución

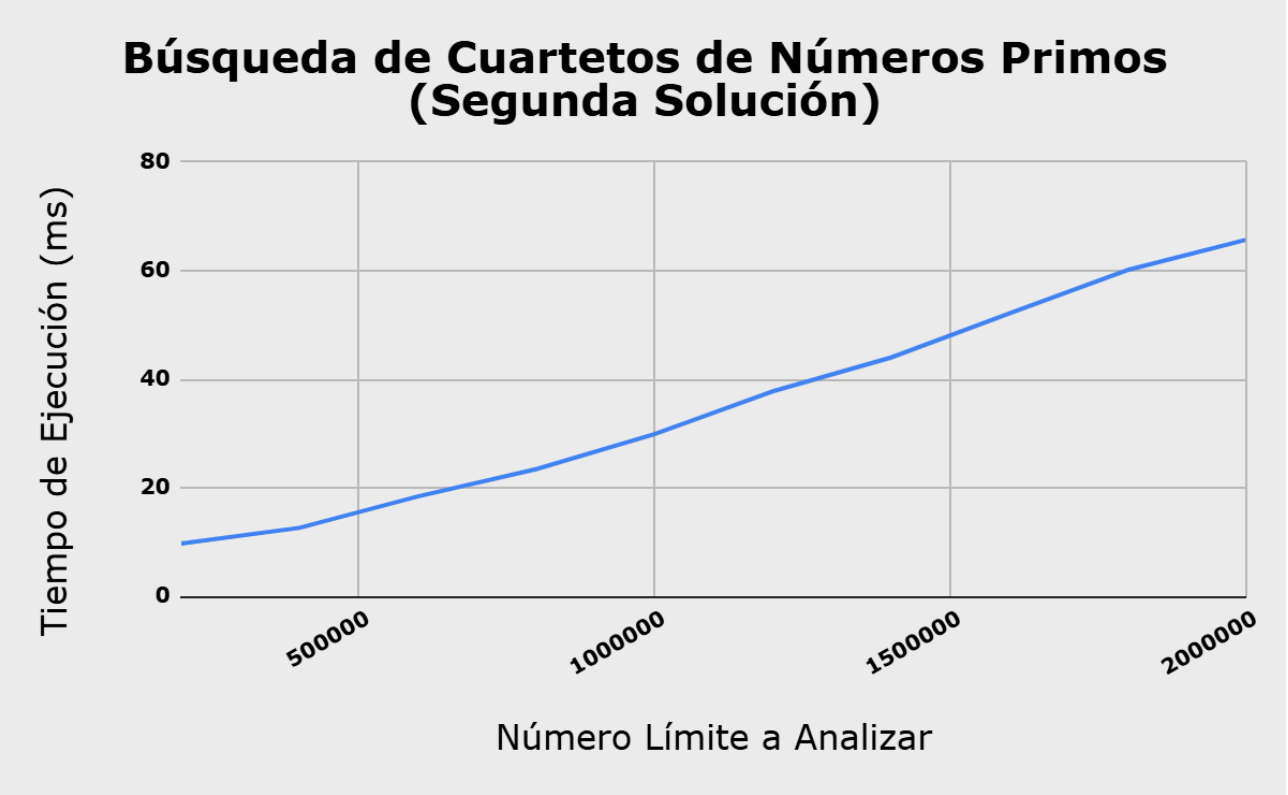

Gráfico que relaciona el valor de n con el tiempo de ejecución *

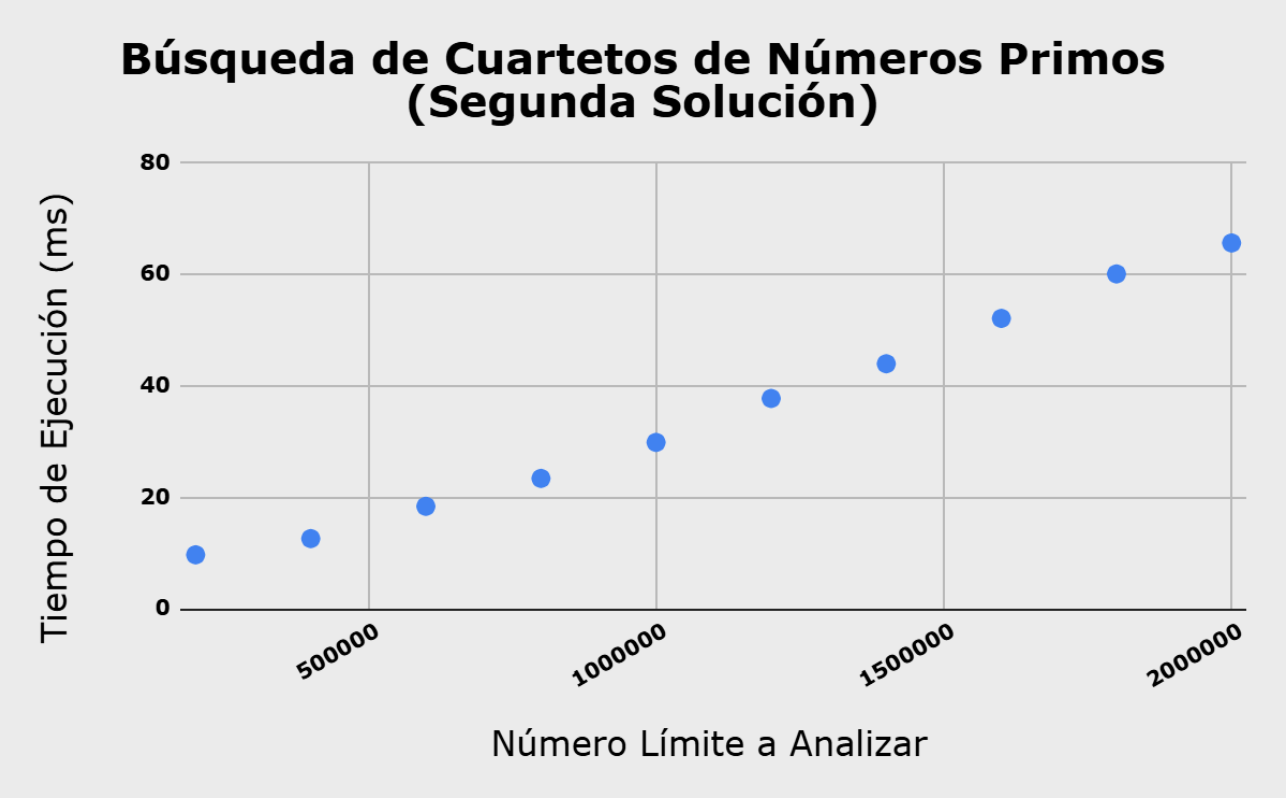

Gráfico de dispersión

** Datos obtenidos analizando el promedio, luego de 1000 ejecuciones en Windows 11 (ver apartado **Instrucciones de Ejecución**). El procesador de la máquina en la cual se realizó la prueba, es un Intel i5 1235U, 1.3GHz, 16GB de RAM.*

Se observa en los gráficos anteriores que los mismos se condicen con la complejidad temporal analizada previamente. Recordemos que la complejidad teórica obtenida es $O(n \cdot log(log(n)))$, que ya representa una complejidad (en general) muy buena.

A continuación, se expone la tabla con los datos obtenidos:

Valor de n	Tiempo de Ejecución (ms)
200000	9,763
400000	12,671
600000	18,458
800000	23,467
1000000	29,927
1200000	37,813
1400000	44,036
1600000	52,175
1800000	60,151
2000000	65,705

3. INFORME DE RESULTADOS

En este informe se presentan 2 soluciones, una es más fiel al código inicial provisto por la cátedra, y la otra lleva la optimización en términos de tiempos de ejecución al extremo. En base a las soluciones propuestas, podemos concluir que ambas son muy buenas en términos de eficiencia temporal, pese a que claramente hay una que se destaca más por sobre la otra. La complejidad teórica, los gráficos y los tiempos de ejecución son consistentes.

Podemos encontrar ventajas y desventajas para ambas soluciones. En principio, la complejidad espacial requerida para la segunda solución es mucho mayor, ya que requerimos de un vector en memoria para realizar la criba. En contraparte, la complejidad temporal de esta solución, como se mencionó en el párrafo anterior, es superior a la de la primera.

A valores de n pequeños, ambas soluciones arrojan tiempos de ejecución similares. A medida que ese valor n crece, la diferencia de tiempo se va acentuando más, por lo que podría ser un factor a tener en cuenta a la hora de elegir el algoritmo a utilizar.

3.1. Instrucciones de Ejecución

En la carpeta del TP0, además de este informe se encuentra un archivo de texto y 5 archivos .py. 2 de ellos se corresponden con las soluciones propuestas, y uno de utilidades varias (limpiar pantalla, validaciones, imprimir datos, etc).

Por otro lado, los últimos 2 archivos sirven para hacer pruebas:

- El archivo **tp0_cespedes_108219_test_ejecucion_estandar.py** ejecuta ambas soluciones para ciertos valores de n ya provistos, y luego genera un resumen con los tiempos de ejecución obtenidos.
- El archivo **tp0_cespedes_108219_test_ejecucion_intensivo.py** permite al usuario elegir:
  - La solución que quiere ejecutar.
  - El número de iteraciones que se desea realizar sobre dicha solución.
  - El valor límite hasta el cual se debe ejecutar dicha solución.
- Al finalizar la ejecución del anterior programa, se genera un resumen de los resultados obtenidos. Si se elige un número de iteraciones mayor a 1, se mostrará también el tiempo de ejecución mínimo y el promedio, resultado de todas las ejecuciones. Para correr el TP0 **se sugiere ejecutar este programa**.
- *Por si llegase a haber algún inconveniente con la ejecución del archivo anterior, se adiciona un .py en la carpeta **TP0_UNICO_ARCHIVO**, el archivo ejecutable en dicho directorio no requiere de ninguna dependencia entre archivos.*

REFERENCIAS

- Python Software Foundation. (s.f.). *Time complexity*. Python Wiki. https://wiki.python.org/moin/TimeComplexity
- Apostol, T. M. (1976). *Introduction to Analytic Number Theory*. Springer.
- Montgomery, H. L., & Vaughan, R. C. (2006). *Multiplicative Number Theory I: Classical Theory*. Cambridge University Press.
- Hardy, G. H., & Wright, E. M. (2008). *An Introduction to the Theory of Numbers*. Oxford University Press.
- GeeksforGeeks. (s.f.). *How is the time complexity of Sieve of Eratosthenes O(n log log n)?* GeeksforGeeks. https://www.geeksforgeeks.org/how-is-the-time-complexity-of-sieve-of-eratosthenes-is-nloglogn/