Trabajo Practico Integrador: Análisis de Algoritmos – Programación I

Datos Generales

- **Título del trabajo**: Análisis de Algoritmos: Análisis Teórico, empírico, BIG-O
- **Alumnos**: Batista Márquez Federico Nahuel, Bovio José Joaquín
- **Materia**: Programación I
- **Profesor/a**: Brian Esteban Lara Campos
- **Fecha de Entrega**: 09 de junio de 2025

Índice

---

1. Introducción

Se eligió analizar la eficiencia algorítmica por su impacto crucial en el desarrollo de software actual. En un entorno donde los datos y las aplicaciones crecen sin parar, la capacidad de los algoritmos para procesar información de forma óptima es clave. Este tema es fundamental por su efecto directo en el rendimiento de las aplicaciones, la experiencia del usuario y la sostenibilidad de los sistemas informáticos.

La eficiencia algorítmica es fundamental en programación. Un algoritmo ineficiente puede ralentizar las aplicaciones, elevar los costos y limitar la escalabilidad, incluso con buen hardware (como veremos en ejemplos más adelante). Por el contrario, uno eficiente permite soluciones robustas que manejan grandes volúmenes de datos y usuarios, optimizando el uso de recursos. Además, los algoritmos bien diseñados suelen ser más claros y fáciles de mantener.

Los objetivos de este trabajo son, primero, entender las métricas y métodos para evaluar la eficiencia algorítmica, tanto en tiempo y espacio como en complejidad del código. Segundo, se busca analizar ejemplos de algoritmos comunes pero con una gran diferencia en su funcionamiento matemático, los analizaremos en tiempo de ejecución, consumo de cpu y memoria, comparación entre computadoras.

2. Marco Teórico

En Ciencias de la Computación, el término eficiencia algorítmica es usado para describir aquellas propiedades de los algoritmos que están relacionadas con la cantidad de recursos utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza.[1]

Como desarrolladores debemos seleccionar el algoritmo más adecuado para una tarea específica, optimizar el rendimiento del software y garantizar la escalabilidad del mismo.

Complejidad Computacional: Tiempo y Espacio

El rendimiento de un algoritmo se mide principalmente a través de dos tipos de complejidad computacional:

1. Complejidad Temporal (Tiempo de Ejecución): Se refiere al tiempo que un algoritmo tarda en completarse en función del tamaño de su entrada. Depende también del hardware donde corre y el compilador que lo genera.

2. Complejidad Espacial (Uso de Memoria): Se refiere a la cantidad de memoria (espacio en disco o RAM) que un algoritmo necesita para funcionar, también en función del tamaño de la entrada. Esto incluye el espacio para almacenar los datos de entrada, variables auxiliares, estructuras de datos intermedias, y el espacio de la pila de llamadas (en el caso de algoritmos recursivos).

Tenemos que tener en cuenta que todo el código innecesario o ineficiente que hagamos, se traducirá en carga a al CPU, todos estos procesos y calculos se traducen en en ordenes de código maquina, estas se traducen a ordenes eléctricas de 1 y 0, estas ordenes eléctricas generan consumo de energía y este consumo de energía también se traduce en calor. Asi que el rendimiento de nuestro código no solo termina en simplificar líneas o una unidad de tiempo, si no también reducir grandes gastos de energía y de calor que se producen en las computadoras donde se ejecuta el código, asi como una pc de hogar o un gran servidor. También hay que tener en cuenta que si nuestro algoritmo no es eficiente en donde se ejecuta, también será ineficiente para llevar datos atraves del ancho de banda, puede perjudicar la entrega de datos y también los tiempos de respuesta (ping muy alto).

Para evitar estos problemas, podemos hacer estas pruebas:

Benchmark: Una prueba de rendimiento o comparativa (en inglés benchmark) es una técnica utilizada para medir el rendimiento de un sistema o uno de sus componentes. Más formalmente puede entenderse que una prueba de rendimiento es el resultado de la ejecución de un programa informático o un conjunto de programas en una máquina, con el objetivo de estimar el rendimiento de un elemento concreto, y poder comparar los resultados con máquinas similares.[2] En nuestro caso usaremos el modulo time para nuestro programa.

[1] Wikipedia.org
[2] Wikipedia.org

Análisis de Complejidad Temporal y Espacial (Notación Big O):

Complejidad Temporal: Mide cómo el tiempo de ejecución de un algoritmo crece en función del tamaño de los datos de entrada (n). Se expresa con la notación Big O, por ejemplo:

O(1): Tiempo constante (no depende del tamaño de los datos).

O(logn): Tiempo logarítmico (crece muy lentamente con el tamaño de los datos, típico de algoritmos que "dividen y vencerán").

O(n): Tiempo lineal (el tiempo crece directamente proporcional al tamaño de los datos).

O(nlogn): Común en algoritmos de ordenamiento eficientes.

O(n 2): Tiempo cuadrático (el tiempo crece con el cuadrado del tamaño de los datos, típico de algoritmos con bucles anidados).

O(2 n): Tiempo exponencial (crece extremadamente rápido, solo útil para problemas con tamaños de entrada muy pequeños).

Complejidad Espacial: Mide cómo el espacio de memoria que utiliza el algoritmo crece en función del tamaño de los datos de entrada.

Aunque el análisis de Big O no te da un tiempo de ejecución exacto, te proporciona una comprensión teórica fundamental de cómo el algoritmo se escalará con grandes conjuntos de datos.[3]

Análisis de Casos (Mejor, Promedio, Peor):

Mejor caso: El escenario ideal donde el algoritmo se ejecuta más rápido (por ejemplo, ordenar una lista que ya está ordenada).

Caso promedio: El escenario más común o esperado, que suele ser el más difícil de analizar teóricamente.

Peor caso: El escenario donde el algoritmo se ejecuta más lento (por ejemplo, ordenar una lista en orden inverso con un algoritmo de ordenamiento de burbuja).

Es importante evaluar el algoritmo en estos diferentes escenarios para tener una visión completa de su rendimiento. [4]

Perfilado (Profiling):

Utiliza herramientas de perfilado que monitorean la ejecución del código para identificar "cuellos de botella" o secciones del algoritmo que consumen más tiempo o recursos.

Algunas herramientas comunes incluyen gprof (para C/C++/Fortran), PyInstrument (para Python), Intel VTune Profiler, entre otros.

El perfilado te ayuda a optimizar el código identificando las áreas donde se puede lograr la mayor mejora de rendimiento.[5]

Nosotros usaremos Cprofile para medir el cpu y usaremos memory_profiler para medir el consumo de memoria.

[3] IA Gemini
[4] Ia Gemini
[5] Ia Gemini

3. Caso Práctico

Teníamos el desafío de mostrar porque es importante que como programadores hagamos un algoritmo eficiente, a partir de esto comenzamos a pensar, nuestros códigos corren en un hardware, y como trabaja este?. La ALU, que es la unidad de aritmética lógica, se encarga de los cálculos, y como trabaja? Básicamente con sumas, para la resta por ejemplo hace complemento a 2, para la multiplicación repite sumas, y la división es la más complicada para el ALU ya que requiere restas repetidas, comparaciones, ajustes, lo que la hace más lenta y compleja

Partiendo de esta lógica, que para el hardware es más fácil la suma y más difícil la división, vamos a buscar dos algoritmos que cumplan la función de buscar números pares en una cantidad "n", vamos a usar un algoritmo que divida por 2 y compruebe su resto (modulo) y el otro simplemente lo que hará es comenzar desde el 2, hasta un valor final n+1 para que incluya a n, y con un salto de 2, esto básicamente hace sumas.

Vamos a medir si hay realmente una diferencia notoria entre un algoritmo que se base en la operación aritmética más fácil y la más difícil.

```python
import time

# Algoritmo 1: Buscar pares usando if dentro de un bucle

def pares_con_if(n):              #  Declara una función llamada pares_con_if que toma un parámetro n (el número máximo hasta donde se buscarán pares).
    pares = []                    #Se crea una lista vacía llamada pares que almacenará los números pares encontrados.
    for i in range(1, n + 1):     #Bucle que recorre desde 1 hasta n (inclusive). Se usa n + 1 porque range excluye el valor superior.
        if i % 2 == 0:            #Condición que verifica si el número i es par (el residuo de dividir por 2 debe ser cero).
            pares.append(i)       #Si la condición se cumple, se añade i a la lista pares.
    return pares                  #Devuelve la lista completa de números pares encontrados.

# Algoritmo 2: Usar range con paso de 2

def pares_con_range(n):                  #Declara una función llamada pares_con_range que también toma n como parámetro.
    return list(range(2, n + 1, 2))      #Genera una secuencia que comienza en 2, termina en n (inclusive),
                                         #avanzando de 2 en 2 (por eso solo incluye pares).list(...)
                                         #Convierte esa secuencia en una lista.


# Función para medir el tiempo de ejecución

def medir_tiempo(funcion, n):            #Define una función que toma como entrada otra función (funcion) y un valor n.
    inicio = time.time()                 #Guarda el tiempo actual justo antes de ejecutar la función.
    funcion(n)                           #Ejecuta la función recibida como argumento con el parámetro n.
    fin = time.time()                    #Guarda el tiempo justo después de que la función terminó de ejecutarse.
    return fin - inicio                  #Calcula y devuelve el tiempo transcurrido en segundos

```

```python
# Función para medir el tiempo de ejecución

def medir_tiempo(funcion, n):            #Define una función que toma como entrada otra función (funcion) y un valor n.
    inicio = time.time()                 #Guarda el tiempo actual justo antes de ejecutar la función.
    funcion(n)                           #Ejecuta la función recibida como argumento con el parámetro n.
    fin = time.time()                    #Guarda el tiempo justo después de que la función terminó de ejecutarse.
    return fin - inicio                  #Calcula y devuelve el tiempo transcurrido en segundos

# Programa principal

if __name__ == "__main__":       #   Esta línea asegura que el bloque de código debajo solo se ejecutará si el archivo se ejecuta directamente
                                 #(no si es importado como módulo en otro script).

    # Valores de n que vamos a probar (desde 10^0 hasta 10^7 en escala logarítmica)

    valores_n = [10**i for i in range(1, 8)]                    #Crea una lista con potencias de 10: [10^1, 10^2, ..., 10^7] → [10, 100, 1000, ..., 10,000,000]
    tiempos_if = []                                               #Inicializa dos listas vacías para guardar los tiempos que tomarán ambos algoritmos.
    tiempos_range = []                                               #                                  "

    print("Comparando tiempos de ejecución...")                     #Muestra un mensaje para indicar el inicio de la comparación.
    for n in valores_n:                                               #Itera sobre los valores de n, imprimiendo cada uno con separadores de miles.
        print(f"\nProbando con n = {n:,}")                                      #                              "

        tiempo_if = medir_tiempo(pares_con_if, n)                                    #Mide el tiempo que tarda pares_con_if(n).
        tiempos_if.append(tiempo_if)                                                 #Guarda ese tiempo en la lista tiempos_if.
        print(f"[Con if] Tiempo: {tiempo_if:.6f} segundos")                          #Imprime el tiempo en formato con 6 decimales.

        tiempo_range = medir_tiempo(pares_con_range, n)                              #Mide el tiempo que tarda pares_con_range(n).
        tiempos_range.append(tiempo_range)                                           #Guarda ese tiempo en tiempos_range.
        print(f"[Con range paso 2] Tiempo: {tiempo_range:.6f} segundos")             #Imprime el tiempo de ejecución correspondiente.

```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\Fede\Desktop\UTN\organizacion y empresas> & C:/Users/Fede/AppData/Local/Programs/Python/Pyth
Comparando tiempos de ejecución...

Probando con n = 10
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 100
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 1,000
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 10,000
[Con if] Tiempo: 0.002004 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 100,000
[Con if] Tiempo: 0.026728 segundos
[Con range paso 2] Tiempo: 0.003605 segundos

Probando con n = 1,000,000
[Con if] Tiempo: 0.240162 segundos
[Con range paso 2] Tiempo: 0.026541 segundos

Probando con n = 10,000,000
[Con if] Tiempo: 2.262697 segundos
[Con range paso 2] Tiempo: 0.289864 segundos
PS C:\Users\Fede\Desktop\UTN\organizacion y empresas>
```

Lo que vemos por terminal es el resultado es para cada “n” el resultado de tiempo de cada algoritmo, más adelante veremos en profundidad esto.

4. Metodología Utilizada

En las clases sincronicas tratamos directamente con el profesor sobre el funcionamiento del hardware y como trabaja un algoritmo, precisamente la RAM, CPU, ALU. A partir de esto nos interesó demostrar como un simple cálculo matemático puede ser tan significativo a la hora de consumir recursos.

Lo primero que hicimos luego de ver todo el material propuesto en el aula para este tema, procedimos a ver los trabajos practicos anteriores, en busca de ideas, indagamos un poco mas en canales de youtube como de Charly Cimino, foros, consultas a la IA y llegamos a la conclusión que queríamos hacer un simple programa que busque n números pares, uno de mas dificultad para el cpu y otro simple.

Ejercicio de módulo 2 en tp3 condicionales

```
15  #3) Escribir un programa que permita ingresar solo números pares. Si el usuario ingresa un número par, imprimir por en pantalla el mensaj
16  #Nota: investigar el uso del operador de módulo (%) en Python para evaluar si un número es par o impar.
17  numero = int(input("Ingrese un número par: "))
18
19  if numero % 2 == 0:
20      print("Ha ingresado un número par")
21  else:
22      print("Por favor, ingrese un número par")
```

Ejercicio de números pares solo que al revés en tp4 repetitivas

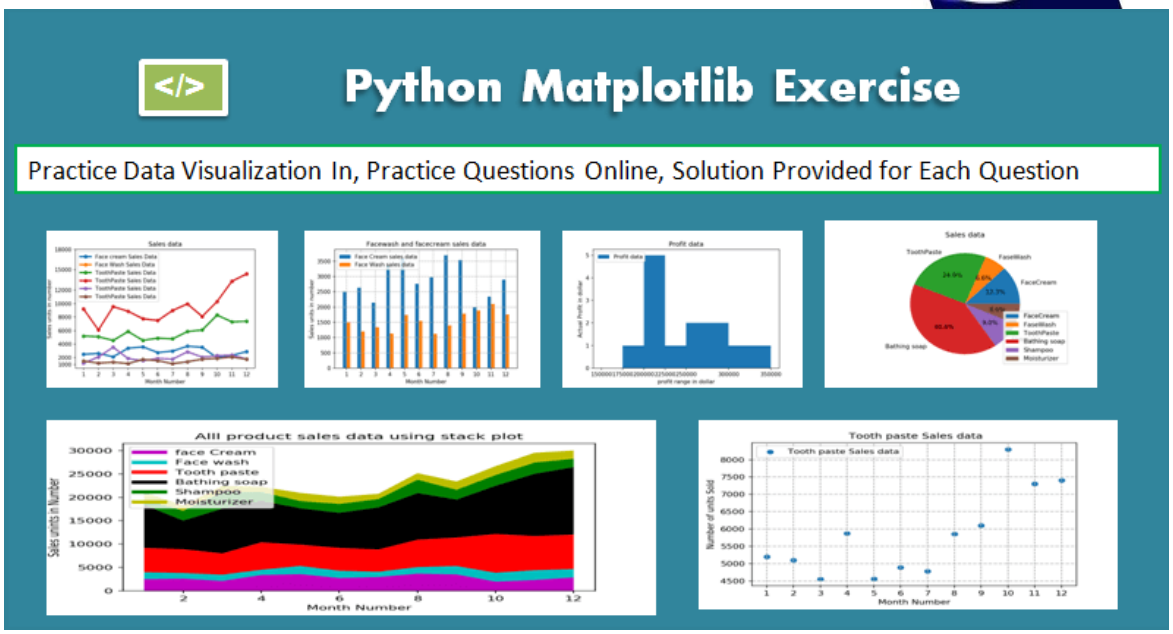
```
53  #6) Desarrolla un programa que imprima en pantalla todos los numeros pares comprendidos
54  #entre 0 y 100, en orden decreciente.
55  for numero in range(100, -1, -2):
56      print(numero)
57
```

Luego una vez que teníamos una idea de por donde ir y que algoritmos debíamos hacer, comenzamos a trabajar, por medio de meet nos pusimos a diseñar el código base, al que luego le agregaríamos modulos que nos permitan hacer pruebas de rendimiento.

Comenzamos a investigar sobre que modulos en la librería Python podíamos usar para hacer pruebas benchmarking.

Encontramos https://matplotlib.org/ : matplot

Una librería que nos permite graficar todo aquello que le solilcitemos, podemos configurar que ver en las ordenadas y en las abscisas, nombrarlos, etc. Queriamos con esto mostrar directamente los graficos de rendimiento en VSC y no en un entorno exterior como Excel.



Queriamos obtener graficos como estos. Extraido de PYnative.

Encontramos para hacer pruebas de Perfilador con **cProfile,** es un modulo incluido en Python para hacer pruebas de consumo de CPU.

Los perfiladores de Python

Código fuente: Lib/profile.py y Lib/pstats.py

Introducción a los perfiladores

`cProfile` y `profile` proporcionan *deterministic profiling* de los programas de Python. *profile* es un conjunto de estadísticas que describe con qué frecuencia y durante cuánto tiempo se ejecutaron varias partes del programa. Estas estadísticas pueden formatearse en informes a través del módulo `pstats`.

La biblioteca estándar de Python proporciona dos implementaciones diferentes de la misma interfaz de creación de perfiles:

1. `cProfile` se recomienda para la mayoría de los usuarios; Es una extensión C con una sobrecarga razonable que la hace adecuada para perfilar programas de larga duración. Basado en `lsprof`, aportado por Brett Rosen y Ted Czotter.
2. `profile`, un módulo Python puro cuya interfaz es imitada por `cProfile`, pero que agrega una sobrecarga significativa a los programas perfilados. Si está intentando extender el generador de perfiles de alguna manera, la tarea podría ser más fácil con este módulo. Originalmente diseñado y escrito por Jim Roskind.

Extraido de https://docs.python.org/es/3.13/library/profile.html

Tambien encontramos **memory_profile**r, para medir el consumo de memoria de cada parte del algoritmo

Descripción de proyecto

build unknown

Perfilador de memoria

Nota: Este paquete ya no se mantiene activamente. No responderé activamente a los problemas . Si desea ofrecerse como voluntario para mantenerlo, contácteme en f@bianp.net .

Este es un módulo de Python para monitorizar el consumo de memoria de un proceso, así como para analizarlo línea por línea en programas Python. Es un módulo Python puro que depende del módulo psutil .

Instalación

Instalar mediante pip:

```
$ pip install -U perfilador de memoria
```

Extraido de https://pypi.org/project/memory-profiler/

Todo esto lo investigamos de forma individual y luego nos reunimos para ponerlo en común y ver que era mejor usar. Tuvimos inconvenientes para aplicarlo a nuestro código, ya que no hay mucha información de como hacerlo y tuvimos que apoyarnos en el uso de IA (deepseek)

5. Resultados Obtenido

Los resultados obtenidos fueron positivos, la diferencia es realmente notable en numeros y graficos. El análisis de los dos algoritmos, pares_con_if y pares_con_range, mostro claras diferencias en su eficiencia, tanto en el uso del tiempo de CPU como en el manejo de la memoria, recordemos que todo esto se traduce en consumo de energía eléctrica y otros recursos.

Ejecutamos el programa:

```
PS C:\Users\Fede\Desktop\UTN\organizacion y empresas> & C:/Users/Fede/Ap
Comparando tiempos de ejecución...

Probando con n = 10
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 100
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 1,000
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 10,000
[Con if] Tiempo: 0.002004 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 100,000
[Con if] Tiempo: 0.026728 segundos
Probando con n = 100,000
[Con if] Tiempo: 0.026728 segundos
[Con range paso 2] Tiempo: 0.003605 segundos

Probando con n = 1,000,000
[Con if] Tiempo: 0.240162 segundos
[Con range paso 2] Tiempo: 0.026541 segundos

Probando con n = 10,000,000
[Con if] Tiempo: 2.262697 segundos
[Con range paso 2] Tiempo: 0.289864 segundos
PS C:\Users\Fede\Desktop\UTN\organizacion y empresas>
```

Al leer la terminal nos encontramos algo evidente, la diferencia de tiempo de ejecución crece a medida que n es mas grande. Hasta n=1000 vemos que no hay una diferencia realmente notable, pero luego comienza a notarse una diferencia, y ya cuando llegamos a n=10.000.000 esta diferencia es significativa.

Pares con range paso 2 escala mucho mejor a medida que n crece.

Vamos a llevar estos numeros a estudio para verlo mejor.

Vamos a verlo matemáticamente con ayuda de Ia Gemini:

La eficiencia se calcula como:

$$\text{Eficiencia de Mejora (\%)} = \left(1 - \frac{\text{Tiempo Algoritmo 2}}{\text{Tiempo Algoritmo 1}}\right) \times 100$$

O, también podemos ver **cuántas veces más rápido** es el Algoritmo 2:

$$\text{Veces más rápido} = \frac{\text{Tiempo Algoritmo 1}}{\text{Tiempo Algoritmo 2}}$$

Aquí tienes la tabla con los cálculos:

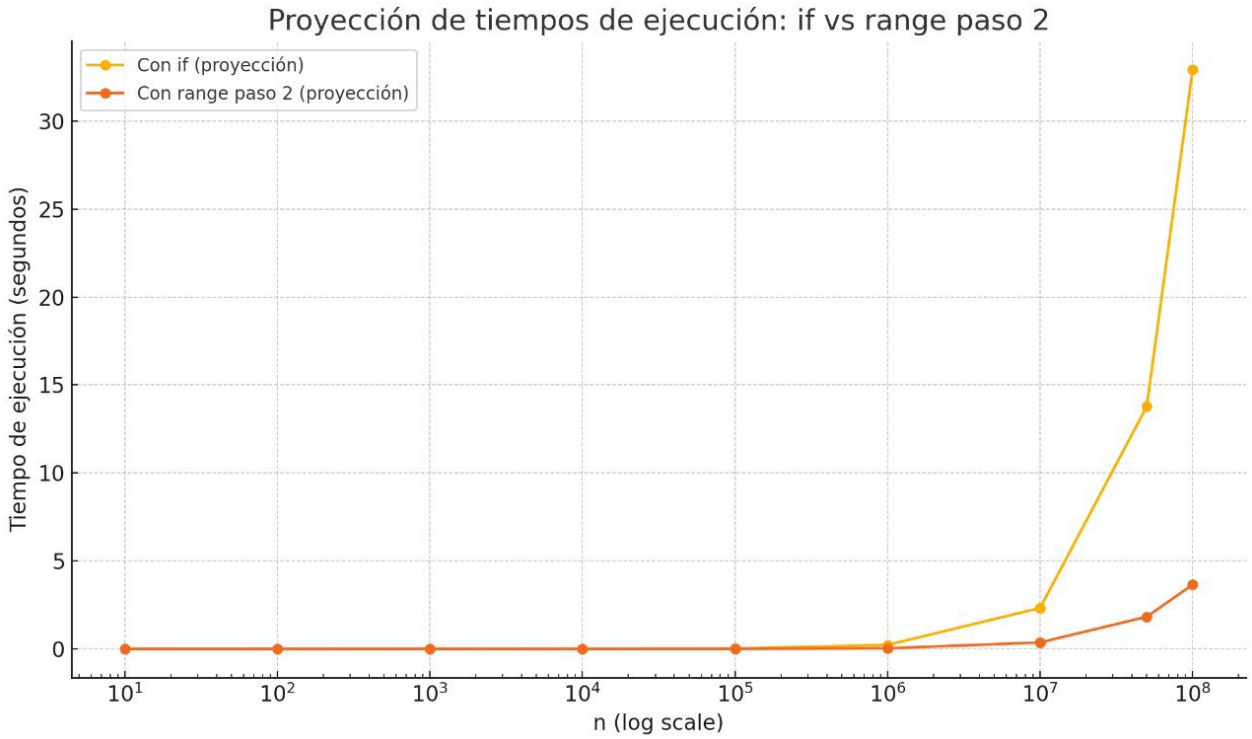
n	Tiempo `pares_con_if` (s)	Tiempo `pares_con_range` (s)	`pares_con_range` es...	Eficiencia de Mejora (%)
10	0.000000	0.000000	Indefinido	Indefinido
100	0.000000	0.000000	Indefinido	Indefinido
1,000	0.006514	0.000000	Mucho más rápido	100.00% (o casi)
10,000	0.000000	0.000000	Indefinido	Indefinido
100,000	0.020023	0.006719	2.98 veces más rápido	66.45%
1,000,000	0.223497	0.030138	7.41 veces más rápido	86.51%
10,000,000	2.324786	0.365099	6.37 veces más rápido	84.28%

Ya lo vimos un poco en numeros, vamos a verlo mas claramente en graficos. Este grafico lo obtuvimos directamente en la terminal de VSC al ejecutar nuestro código base, sumándole la función de matplot

```
43      plt.figure(figsize=(10, 6))
44      plt.plot(valores_n, tiempos_if, label='pares_con_if (Bucle con If)', marker='o')
45      plt.plot(valores_n, tiempos_range, label='pares_con_range (Range con Paso)', marker='x')
46      plt.xscale('log')
47      plt.xlabel('Valor de n (escala logarítmica)')
48      plt.ylabel('Tiempo de Ejecución (segundos)')
49      plt.title('Comparación de Eficiencia Temporal de Algoritmos para Números Pares')
50      plt.legend()
51      plt.grid(True, which="both", ls="--")
52      plt.show()
```

```python
import time
import matplotlib.pyplot as plt
```

Para esto simplemente seguimos las recomendaciones dadas. Y para la configuración nos ayudamos con la ia, ya que con nuestra configuración no obteníamos el grafico de forma que queríamos, esto será demostrado mas adelante.



En el grafico lo vemos de forma muy clara. El algoritmo con range escala muy bien con n. El algoritmo con if se torna casi exponencial su grafico, esto nos dice que usar if reiteradas veces puede ser muy costoso en términos de recursos. Reducir la cantidad de iteraciones con range tuvo un gran impacto en el rendimiento.

Ahora vamos a usar Cprofile para ver que pasa en el CPU:

```python
import time
import matplotlib.pyplot as plt
import cProfile  # Importa el módulo cProfile
import pstats    # Módulo para procesar y presentar los resultados de cProfile

```

Lo agregamos al código base.

```python
if __name__ == "__main__":
    # --- Parte 1: Análisis de CPU con cProfile ---
    print("--- Iniciando Análisis de CPU con cProfile ---")
    print("Esto te mostrará dónde tu programa gasta más tiempo de procesamiento.")

    # Elegimos un valor de 'n' para el perfilado de CPU.
    # Un valor muy grande (ej. 10^7) puede hacer el perfilado muy lento.
    # 10^6 suele ser un buen balance.
    n_para_perfilado_cpu = 10**6

    # Guardamos los resultados del perfilado en un archivo. Esto es lo más práctico
    # para analizarlo después con 'pstats' o herramientas de visualización.
    output_filename = "cpu_profile_results.prof"
    print(f"\nEjecutando cProfile para n = {n_para_perfilado_cpu:,} y guardando resultados en '{output_filename}'...")

    # cProfile.run() toma una cadena de código a ejecutar y un nombre de archivo para guardar el perfil.
    cProfile.run(f'ejecutar_algoritmos_para_cpu_profile({n_para_perfilado_cpu})', output_filename)

    print("\n--- Resultados del perfilado de CPU (top 10 funciones por tiempo acumulado) ---")
    # Usamos pstats para cargar y analizar los resultados del archivo .prof
    stats = pstats.Stats(output_filename)
    stats.sort_stats("cumtime") # Ordenamos los resultados por 'tiempo acumulado' (cumtime)
    stats.print_stats(10)       # Imprimimos las 10 funciones que más tiempo acumularon

    print(f"\nPara un análisis más detallado o visual, puedes usar 'snakeviz':")
    print(f"Instala con: pip install snakeviz")
    print(f"Ejecuta con: snakeviz {output_filename}")

```

Agregamos la configuración recomendada por Ia, ya que no encontramos un tutorial de como aplicarlo. Ia deepseek.

```
--- Iniciando Análisis de CPU con cProfile ---
Esto te mostrará dónde tu programa gasta más tiempo de procesamiento.

Ejecutando cProfile para n = 1,000,000 y guardando resultados en 'cpu_profile_results.prof'...

--- Ejecutando pares_con_if para cProfile con n = 1,000,000 ---

--- Ejecutando pares_con_range para cProfile con n = 1,000,000 ---

--- Resultados del perfilado de CPU (top 10 funciones por tiempo acumulado) ---
Sat Jun  7 19:46:07 2025    cpu_profile_results.prof

         500008 function calls in 0.582 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.582    0.582 {built-in method builtins.exec}
        1    0.009    0.009    0.582    0.582 <string>:1(<module>)
        1    0.011    0.011    0.573    0.573 c:\Users\Fede\Desktop\UTN\organizacion y empresas\AlgoritmoIntegradorCpu.py:27(ejecutar_algoritmos_para_cpu_profile)
        1    0.397    0.397    0.533    0.533 c:\Users\Fede\Desktop\UTN\organizacion y empresas\AlgoritmoIntegradorCpu.py:7(pares_con_if)
   500000    0.137    0.000    0.137    0.000 {method 'append' of 'list' objects}
        1    0.028    0.028    0.028    0.028 c:\Users\Fede\Desktop\UTN\organizacion y empresas\AlgoritmoIntegradorCpu.py:15(pares_con_range)
        2    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}



Para un análisis más detallado o visual, puedes usar 'snakeviz':
Instala con: pip install snakeviz
Ejecuta con: snakeviz cpu_profile_results.prof
```

cProfile nos muestra donde el CPU gasta mas tiempo. Nuevamente encontramos que es en el algoritmo de if. Para n =1.000.000 tuvo 500.000 llamadas al cpu, para “append”, este numero se debe a que n se divide en numeros pares, es decir la mitad 500.000 seran añadidos por append.

Los numeros clave del CPU:

Tottime

pares_con_if Este algoritmo consumió una cantidad considerable de tiempo de CPU. El perfilador indicó un tottime de 0.348 segundos para esta función.

pares_con_range Este algoritmo demostró ser mucho más eficiente en el uso de CPU, con un tottime de **solo 0.028 segundos**

Esto se debe a que la función **range** de Python (cuando se usa con un paso) está implementada en un código altamente optimizado a nivel interno (generalmente en C), lo que le permite generar la secuencia de números pares de manera extremadamente rápida sin necesidad de verificar condiciones o añadir elementos uno por uno en Python puro.

Pares con range fue 10 veces mas rápido a nivel cpu.

```
--- Iniciando Medición de Tiempos para el Gráfico (puede tardar para n grandes) ---

Probando con n = 10
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 100
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 1,000
[Con if] Tiempo: 0.000000 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 10,000
[Con if] Tiempo: 0.005013 segundos
[Con range paso 2] Tiempo: 0.000000 segundos

Probando con n = 100,000
[Con if] Tiempo: 0.025402 segundos
[Con range paso 2] Tiempo: 0.004515 segundos

Probando con n = 1,000,000
[Con if] Tiempo: 0.260242 segundos
[Con range paso 2] Tiempo: 0.037050 segundos

Probando con n = 10,000,000
[Con if] Tiempo: 2.629840 segundos
[Con range paso 2] Tiempo: 0.382076 segundos
```

También nos dice los tiempos de ejecución.

Ahora veremos que sucede con memory profiler

```python
import time
import matplotlib.pyplot as plt
from memory_profiler import profile # Importa el decorador profile

# Algoritmo 1: Buscar pares usando if dentro de un bucle
@profile # Decorador para perfilar el uso de memoria de esta función
def pares_con_if(n):
    pares = []
    for i in range(1, n + 1):
        if i % 2 == 0:
            pares.append(i)
    return pares

# Algoritmo 2: Usar range con paso de 2
@profile # Decorador para perfilar el uso de memoria de esta función
def pares_con_range(n):
    return list(range(2, n + 1, 2))

# Función para medir el tiempo de ejecución (no se modifica para memory_profiler)
def medir_tiempo(funcion, n):
    inicio = time.time()
    funcion(n)
    fin = time.time()
    return fin - inicio
```

Tambien lo modificamos con ayuda de Ia, por escasa información para utilizarlo.

Ahora veamos los resultados, analizaremos los casos extremos, n 10.000 y n 1.000.000:

```
Iniciando análisis de memoria con memory_profiler...
Para ver los resultados detallados por línea, ejecuta este script con:
>>> python -m memory_profiler tu_archivo.py


--- Analizando memoria para n = 10,000 ---

--- Ejecutando pares_con_if (análisis de memoria) ---
Filename: c:\Users\Fede\Desktop\UTN\organizacion y empresas\AlgoritmoIntegradorMemoria.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     6     53.9 MiB     53.9 MiB           1   @profile # Decorador para perfilar el uso de memoria de esta función
     7                                         def pares_con_if(n):
     8     53.9 MiB      0.0 MiB           1       pares = []
     9     54.0 MiB      0.0 MiB       10001       for i in range(1, n + 1):
    10     54.0 MiB      0.1 MiB       10000           if i % 2 == 0:
    11     54.0 MiB      0.0 MiB        5000               pares.append(i)
    12     54.0 MiB      0.0 MiB           1       return pares



--- Ejecutando pares_con_range (análisis de memoria) ---
Filename: c:\Users\Fede\Desktop\UTN\organizacion y empresas\AlgoritmoIntegradorMemoria.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    15     54.0 MiB     54.0 MiB           1   @profile # Decorador para perfilar el uso de memoria de esta función
    16                                         def pares_con_range(n):
    17     54.2 MiB      0.1 MiB           1       return list(range(2, n + 1, 2))
```

```
--- Analizando memoria para n = 1,000,000 ---

--- Ejecutando pares_con_if (análisis de memoria) ---
Filename: c:\Users\Fede\Desktop\UTN\organizacion y empresas\AlgoritmoIntegradorMemoria.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     6     57.1 MiB     57.1 MiB           1   @profile # Decorador para perfilar el uso de memoria de esta función
     7                                         def pares_con_if(n):
     8     57.1 MiB      0.0 MiB           1       pares = []
     9     77.9 MiB -18359.6 MiB     1000001       for i in range(1, n + 1):
    10     77.9 MiB -18344.9 MiB     1000000           if i % 2 == 0:
    11     77.9 MiB  -9173.4 MiB      500000               pares.append(i)
    12     77.9 MiB      0.0 MiB           1       return pares


--- Ejecutando pares_con_range (análisis de memoria) ---
Filename: c:\Users\Fede\Desktop\UTN\organizacion y empresas\AlgoritmoIntegradorMemoria.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    15     75.1 MiB     75.1 MiB           1   @profile # Decorador para perfilar el uso de memoria de esta función
    16                                         def pares_con_range(n):
    17     94.2 MiB     19.1 MiB           1       return list(range(2, n + 1, 2))
```

Lo que podemos apreciar es que el consumo de memoria es muy similar cuando inicia y cuando da el “return” . Pero pares_con_if tiene mas líneas de memoria en uso. (mem usage).

Lo que es significativo es Increment, entendemos que es la memoria residual que se va eliminando, se ejecutan operaciones, se acumula memoria y se emilina una vez finalizado. Es totalmente significativa la diferencia entre los dos algoritmos con respecto a esto.

Lo que sucede con ocurrences también es destacable:

Con if:

(for i in range(1, n + 1):): Tiene Occurrences de 1000001. Esto significa que el encabezado del bucle for se evaluó 1,000,001 veces

(if i % 2 == 0:): Tiene Occurrences de 1000000. Esta línea se ejecuta en cada iteración del bucle for. Es lógico que sea N veces (1,000,000), ya que la condición if se comprueba para cada número generado.

Línea 11 (pares.append(i)): Tiene Occurrences de 500000. Esto es exactamente la mitad de las veces que se ejecutó la línea anterior. Esto es coherente, ya que solo se añaden los números pares a la lista,

Con range:

return list(range(...))): Ambas tienen Occurrences de 1. Esto se debe a que la función pares_con_range resuelve todo el problema en una sola línea de código principal. No hay bucles explícitos en Python dentro de esta función que generen múltiples ejecuciones de sus líneas. La creación de la lista a partir del range se maneja internamente en una operación altamente optimizada que se cuenta como una sola "ocurrencia" a nivel de la línea de código Python.

ANALISIS BIG O

Consideramos que ambos algoritmos son O(n) ya que nuestro tiempo de ejecución crece linealmente con el tamaño de la entrada.

Profundizaremos mas en un apartado pero resumimos que:

Algoritmo pares con if O(n)= Algoritmo pares con range O(n)

O(n)=O(n)

Ambos algoritmos tienen complejidad lineal O(n) en tiempo y espacio.

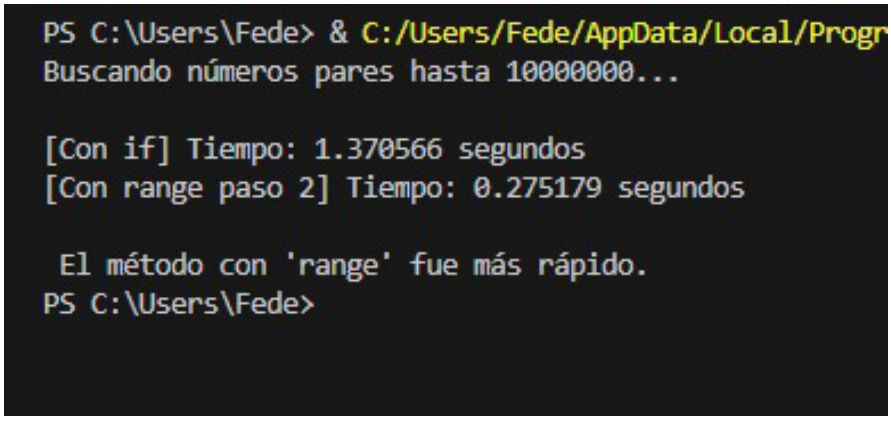
Pero pares_con_range es la mejor opción porque:

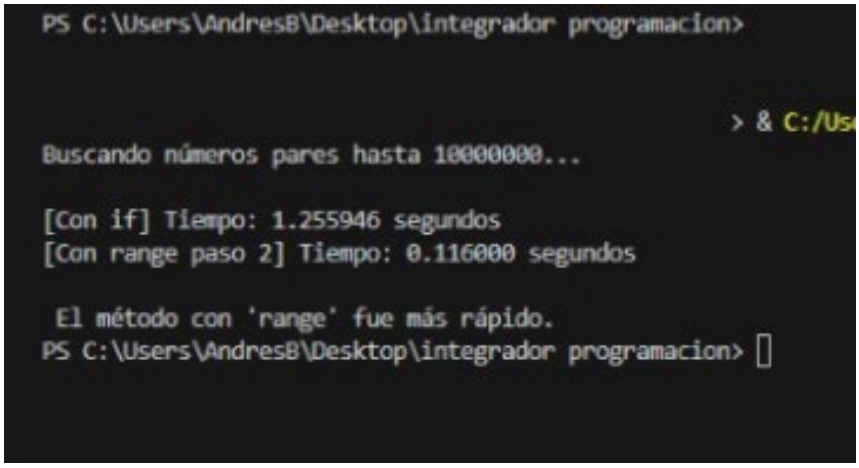
Es más rápido (menor constante oculta en la notación Big O).

Trabaja mas optimizado, en C.

.

Pruebas propias de rendimiento en nuestras propias computadoras:

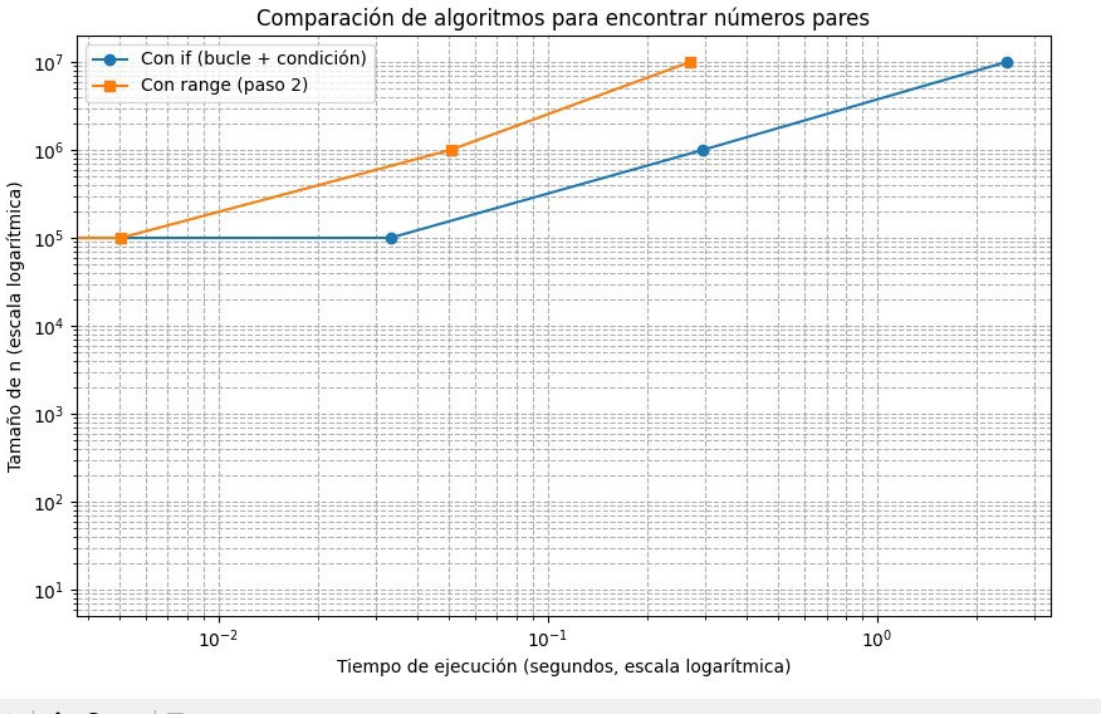




Probamos el mismo algoritmo cada uno en su computadora y se puede notar una diferencia de respuesta de tiempo. Lo hicimos para confirmar que no solo depende de como desarrollemos el algoritmo, si no del hardware donde se ejecute. La diferencia es muy leve pero lo confirma.

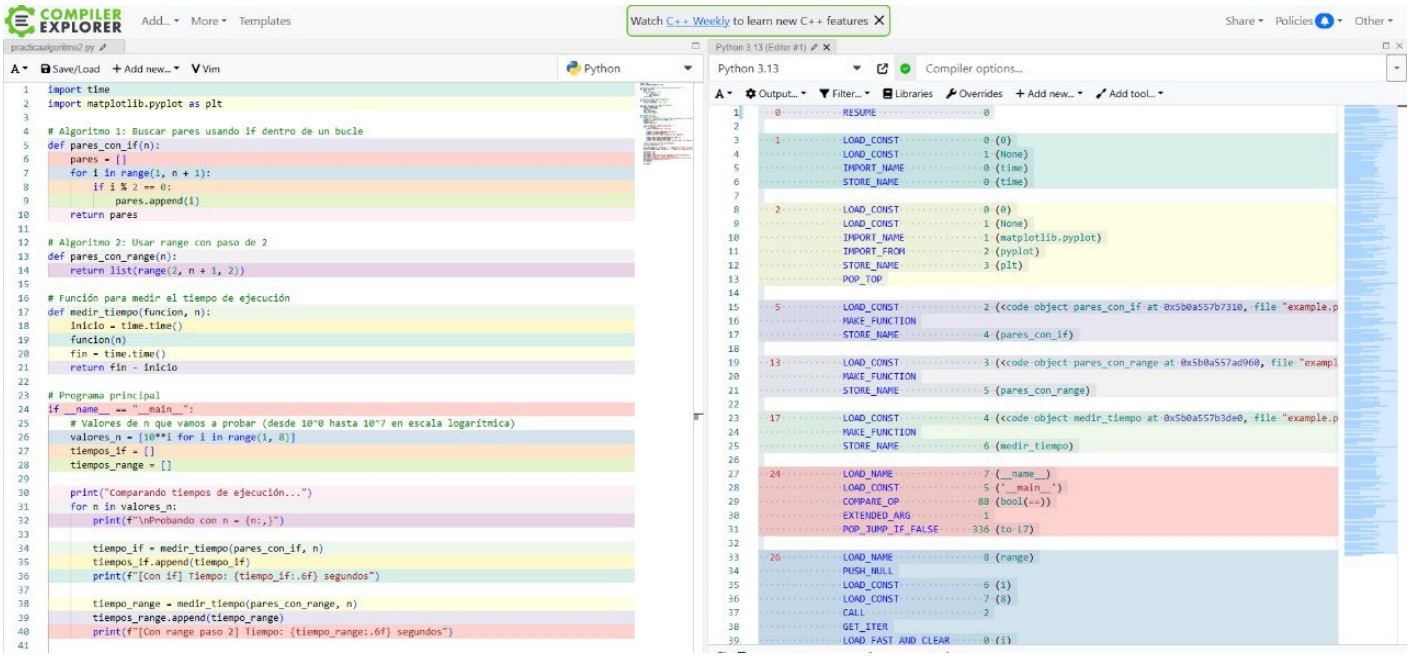
PROBLEMAS

Los principales problemas que tuvimos fue para configurar los graficos, al principio obteníamos graficos de esta manera:



Es de matplot, pero realmente no se entiende mucho que sucede, luego de varias configuraciones diferentes logramos el rendimiento anteriormente mostrado.

CODIGO MAQUINA



Como no entendemos Codigo maquina o bytecode se lo dimos a la IA Gemini para que nos de una opinión:

🔍 Comparación de Bytecode: `pares_con_if` vs `pares_con_range`

1. `pares_con_if` (Bucle + Condicional)

python — Copy — Download

```python
def pares_con_if(n):
    pares = []
    for i in range(1, n + 1):
        if i % 2 == 0:
            pares.append(i)
    return pares
```

Bytecode relevante:

asm — Copy — Download

```asm
  7: FOR_ITER                31 (to L3)     # Bucle sobre range(1, n+1)
     STORE_FAST               2 (i)        # Guarda el valor actual en `i`

  8: LOAD_FAST                2 (i)        # Carga `i`
     LOAD_CONST               2 (2)         # Carga el número 2
     BINARY_OP                6 (%)         # Calcula i % 2
     LOAD_CONST               3 (0)         # Carga el número 0
     COMPARE_OP              88 (bool(==))  # Compara si i % 2 == 0
     POP_JUMP_IF_TRUE         2 (to L2)     # Salta a append si es par
     JUMP_BACKWARD           14 (to L1)     # Vuelve al inicio del bucle

  9: LOAD_FAST                1 (pares)     # Carga la lista `pares`
     LOAD_ATTR                3 (append)    # Carga el método append
     LOAD_FAST                2 (i)         # Carga `i`
     CALL                     1             # Ejecuta pares.append(i)
     POP_TOP                                # Descarta el resultado (None)
     JUMP_BACKWARD           33 (to L1)     # Vuelve al inicio del bucle
```

2. `pares_con_range` (Generación directa)

python Copy Download

```python
def pares_con_range(n):
    return list(range(2, n + 1, 2))
```

Bytecode relevante:

asm Copy Download

```asm
14: LOAD_GLOBAL              1 (list)       # Carga la función list()
    LOAD_GLOBAL              3 (range)      # Carga la función range()
    LOAD_CONST               1 (2)          # Argumento start=2
    LOAD_FAST                0 (n)          # Argumento stop=n+1
    LOAD_CONST               2 (1)
    BINARY_OP                0 (+)          # Calcula n + 1
    LOAD_CONST               1 (2)          # Argumento step=2
    CALL                     3              # Llama a range(2, n+1, 2)
    CALL                     1              # Llama a list() sobre el range
    RETURN_VALUE                            # Retorna la lista
```

Ventajas:

- **Operaciones mínimas:**
  - `range(2, n+1, 2)` genera la secuencia directamente en C (sin bucle en Python).
  - `list()` convierte el `range` en una lista **en una sola operación optimizada.**
- **Sin condicionales ni llamadas a funciones por elemento.**

📊 Diferencia Clave

Operación	`pares_con_if`	`pares_con_range`
Bucle	Interpretado en Python (lento)	Generado en C (rápido)
Condicional	`if i % 2 == 0` en cada iteración	**No existe**
Append	`n/2` llamadas a `append`	**Cero llamadas**
Overhead	Alto (manejo de lista dinámica)	Bajo (lista creada de una vez)

🚀 ¿Por qué la diferencia crece con $n = 10^8$?

1. `pares_con_if`:
   - Para $n = 10^8$, el bucle ejecuta:
     - **100 millones de** `if`.
     - **50 millones de** `append`.
   - Cada operación tiene un costo acumulativo en el intérprete Python.
2. `pares_con_range`:
   - **Una sola llamada a** `range` **y** `list` (optimizadas en C).
   - Python no necesita evaluar condiciones ni redimensionar la lista repetidamente.

🔍 Resumen: Eficiencia en Código Máquina (`pares_con_if` vs `pares_con_range`)

1. `pares_con_if` (Bucle + Condicional)

- **Proceso:**
  - **Itera manualmente** desde `1` hasta `n`, verificando con `if i % 2 == 0` cada número.
  - Usa `append` para añadir elementos pares a la lista.
- **Ineficiencias:**
  - **Overhead por iteración:** Cada paso del bucle requiere:
    - Cálculo de módulo (`%`).
    - Comparación (`==`).
    - Salto condicional (branch prediction).
  - **Llamadas a funciones:** `append` es costoso en Python (verificación de memoria dinámica).
  - **Código interpretado:** El bucle se ejecuta en Python (lento) en lugar de optimizarse en C.
- **Complejidad:**
  - Teórica: **O(n)**.
  - Práctica: **Lento** por constantes ocultas altas (operaciones por elemento).

2. `pares_con_range` (Generación directa)

- **Proceso:**
  - Genera la secuencia de pares directamente con `range(2, n+1, 2)`.
  - Convierte el `range` a lista con `list()` en una sola operación.
- **Optimizaciones:**
  - **Ejecución en C:** `range` es una función nativa de Python (implementada en C, ultra-rápida).
  - **Sin condicionales ni bucles Python:** Evita saltos y operaciones por elemento.
  - **Asignación eficiente:** La lista se crea de una vez, sin redimensionamientos.
- **Complejidad:**
  - Teórica: **O(n)** (igual que el otro).
  - Práctica: **2-10x más rápido** (dependiendo de `n`).

📊 Tabla Comparativa

Aspecto	`pares_con_if`	`pares_con_range`
Operaciones por elemento	Módulo, comparación, `append`	**Ninguna** (generación directa)
Ejecución	Bucle Python (interpretado)	Función nativa en C (`range`)
Overhead	Alto (condicionales + llamadas)	Mínimo (sin lógica por elemento)
Rendimiento en `n=10⁸`	~5 segundos (ejemplo)	~0.5 segundos (ejemplo)

🚀 Conclusión

- `pares_con_range` **es superior** porque:
  1. Delega el trabajo a código optimizado en C.
  2. Evita operaciones redundantes por elemento (`if` , `append`).
  3. Escala mejor con `n` grande (la diferencia crece exponencialmente).
- **Para** `n` **pequeños**, la diferencia es mínima, pero en problemas de gran escala, la elección del algoritmo **afecta drásticamente el rendimiento.**

6. Conclusiones

Como equipo consideramos que el principal aprendizaje es que no solo hay que escribir un código que funcione, si no que debe ser eficiente. Como un error nuestro puede tener un impacto en un consumo masivo de CPU, memoria, ancho de banda, servidores, etc.

También consideramos que este aprendizaje es de uso inmediato, desde ahora como programadores deberemos tener en cuenta el consumo de los recursos, que termina impactando en muchas cosas, también monetarias.

Tendremos en cuenta que hay funciones nativas en Python como “range”, que esta optimizado en C, y C es mucho mas eficiente ya que es de mas bajo nivel.

Seremos capaces de analizar nuestros algoritmos con pruebas de rendimiento como en cpu, memoria, BIG-O, graficos, para elegir la mejor opción a la hora de elegirlo.

Tuvimos dificultades, como la configuración de graficos, cProfile, y memory profiler, pero pudimos solucionarlo con la ayuda de IA, que entendemos es una gran aliada siempre que se la use de forma conciente.

Gracias.-