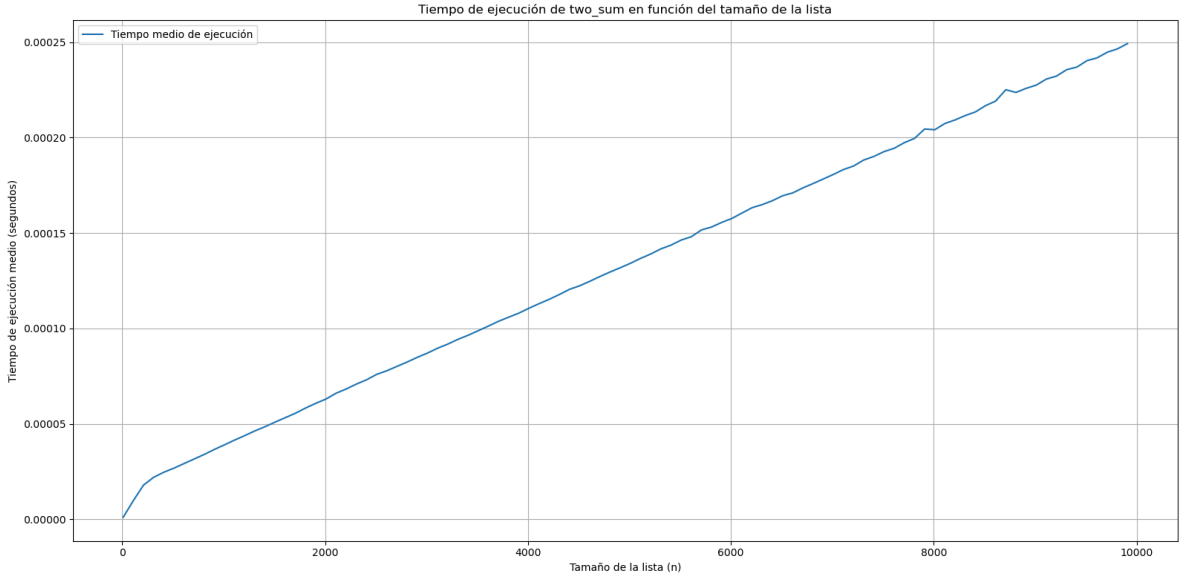
Algoritmia: Práctica 1.

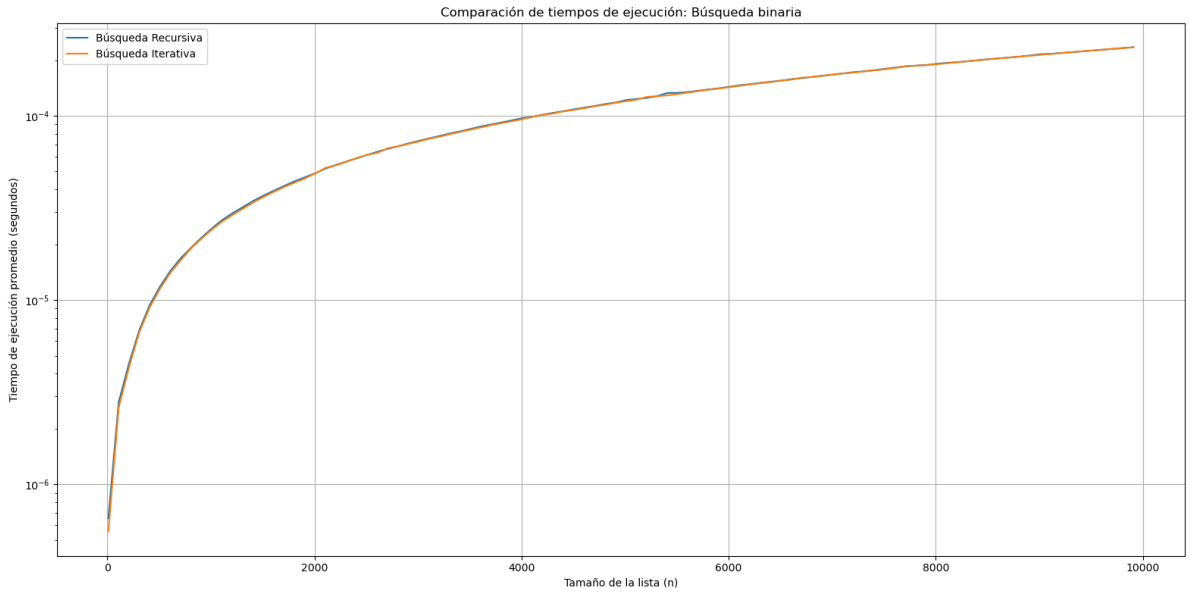
En el primer ejercicio nos piden analizar el tiempo de ejecución de una función *two_sum* cuya gráfica hemos analizado usando la librería de python *matplotlib.*

La siguiente es la gráfica de la ejecución:



Como podemos observar, comprobamos que tenemos una gráfica lineal tal como esperábamos, esto sugiere que la complejidad del algoritmo *two_sum* es de O(n)

En el segundo ejercicio hemos usado la misma librería para graficar las dos ejecuciones con el fin de contrastar tiempos de ejecución entre la búsqueda recursiva e iterativa.



Observamos que los tiempos de ejecución de los dos tipos de búsquedas binarias son prácticamente iguales, con una complejidad algorítmica de O(log n) por lo que la eficacia del algoritmo diseñado es la correcta.

Preguntas teóricas:

I.A.3:

- ¿Cuál es el comportamiento de la función?
    - La función two_sum() normalmente funciona con una complejidad algorítmica de $O(n^2)$ ya que por cada elemento de la lista dada a la función tiene que pasar por todos los elementos dando lugar a un doble for con complejidad $O(n^2)$
- ¿Es posible una implementación que tenga complejidad O(n)?
    - Si, utilizando un diccionario para la implementación de tal manera que solo recorramos la lista una vez, calculando el número exacto que necesita para realizar la suma y buscando el número en el diccionario, búsqueda cuyo coste es de O(1). Esta es nuestra implementación de two_sum con O(n):

```python
def two_sum(lst, n):
    #Hacemos un diccionario para almacenar los numeros
    number_is = {}

    for element in lst:
        number = n - element

        #Si el numero ya esta en el diccionario, hemos encontrado el complemento
        if number in number_is:
            return True

        #Añadimos el numero al diccionario si no estaba antes
        number_is[element] = True

    return False
```

I.B.3:

- ¿Cuál es el comportamiento de las funciones? ¿Parece logarítmico?
    - Las dos funciones, tanto itr_bs() como rec_bs() implementan una búsqueda binaria por lo que tienen las dos una complejidad de O(log n)
- ¿Cómo se puede crear una gráfica que enseñe claramente si el comportamiento es logarítmico?
    - Utilizamos matplotlib para mostrar que la gráfica es logarítmica
- Si no lo es, ¿qué comportamiento tiene?

    - Si no fuera logarítmico tendría que recorrer la lista 2 veces por iteración por lo que se comportaría como una función de complejidad $O(n^2)$
- ¿Qué conclusiones se pueden derivar sobre la indexación de las listas en Python?
    - Si el resultado no resulta logarítmico podemos concluir que la indexación de listas en python no es tan eficiente como debería.
- Dado que se ha elegido key para conseguir siempre el caso peor, la variancia de la medida es, en teoría, cero. ¿Es éste el caso?
    - En teoría si se usa siempre el peor caso la varianza debería ser próxima a cero, en el caso de esta práctica es cercana a cero pero puede ser que no todo lo que debería por factores como la carga del sistema en otras operaciones.

Observación: como por definición la función f debe recibir un parámetro, y two_sum necesita dos parámetros, lo que hemos realizado ha sido lo siguiente:
time_measure(lambda lst: two_sum(lst, 500), dataprep, range(10, 10001, 100))
Hemos usado lambda para crear una función de un solo parámetro (en este caso lst) para pasarlo a time_measure, que luego llamará a two_sum internamente. Usar lambda nos ha parecido una forma más concisa de hacerlo sin tener que definir una función con def.

Luis Núñez Fernández y Lucía Chacón Sanabria
Nº pareja: 02