ANÁLISIS DE LAS GRÁFICAS DE MEASURE.PY

Iván Sánchez Bonacasa
Christian Grosso

ÍNDICE

Análisis de las gráficas de tiempo de ejecución **3**
1. Gráfica de la función itr_bs (Iterative Binary Search) 3
2. Gráfica de la función rec_bs (Recursive Binary Search) 4
3. Gráfica de la función two_sum 5

Comparaciones **6**
Comparación entre itr_bs y rec_bs: 6
Comparación entre two_sum y búsqueda binaria: 6

Mejoras al algoritmo **7**
Primera versión de two_sum: 7
Segunda versión de two_sum: 8
Conclusión de las versiones de two_sum 8

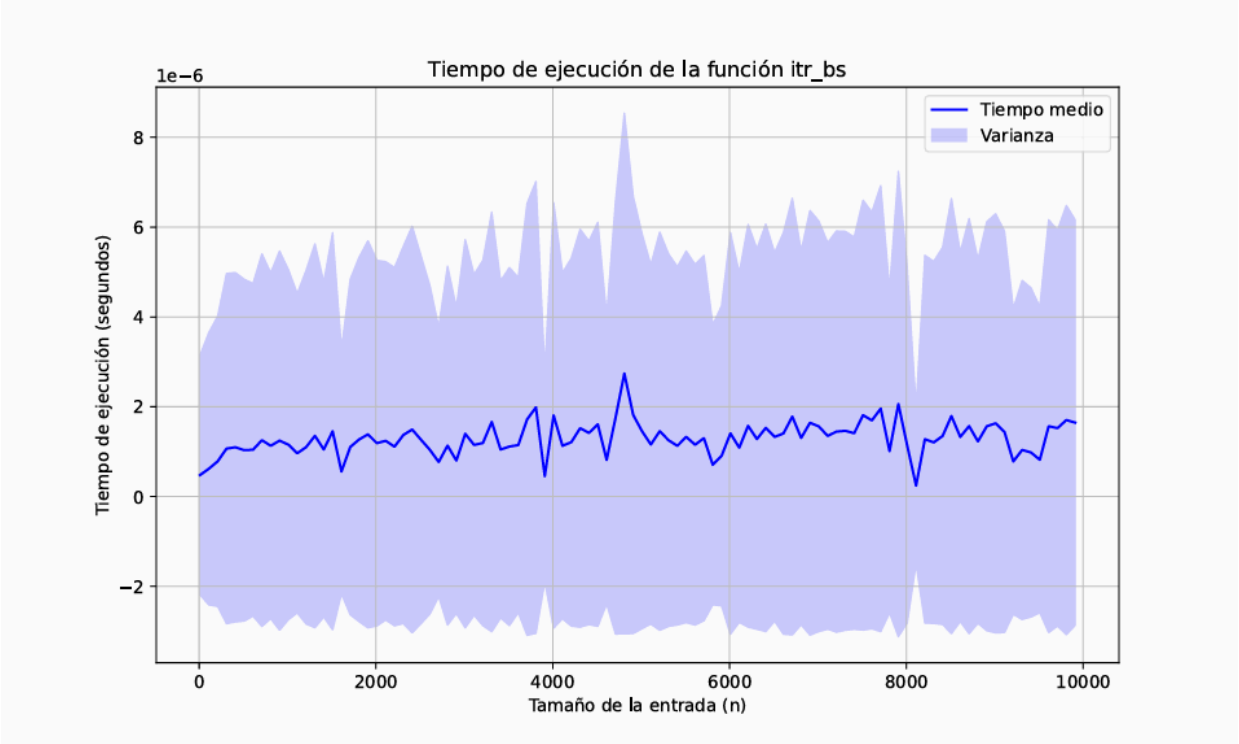
Análisis de las gráficas de tiempo de ejecución

Las tres gráficas muestran el tiempo de ejecución promedio y la varianza para tres funciones diferentes: **itr_bs**, **rec_bs** y **two_sum**. A continuación, analizamos cada gráfica en detalle:

1. Gráfica de la función itr_bs (Iterative Binary Search)

- **Comportamiento del tiempo medio**: El tiempo medio de ejecución se mantiene bastante constante, sin grandes variaciones a lo largo de los distintos tamaños de entrada. Esto sugiere que el algoritmo iterativo de búsqueda binaria escala de manera eficiente, con una complejidad aproximada de *O(log n)*, ya que el crecimiento en el tiempo de ejecución es muy lento a medida que el tamaño de la entrada aumenta.

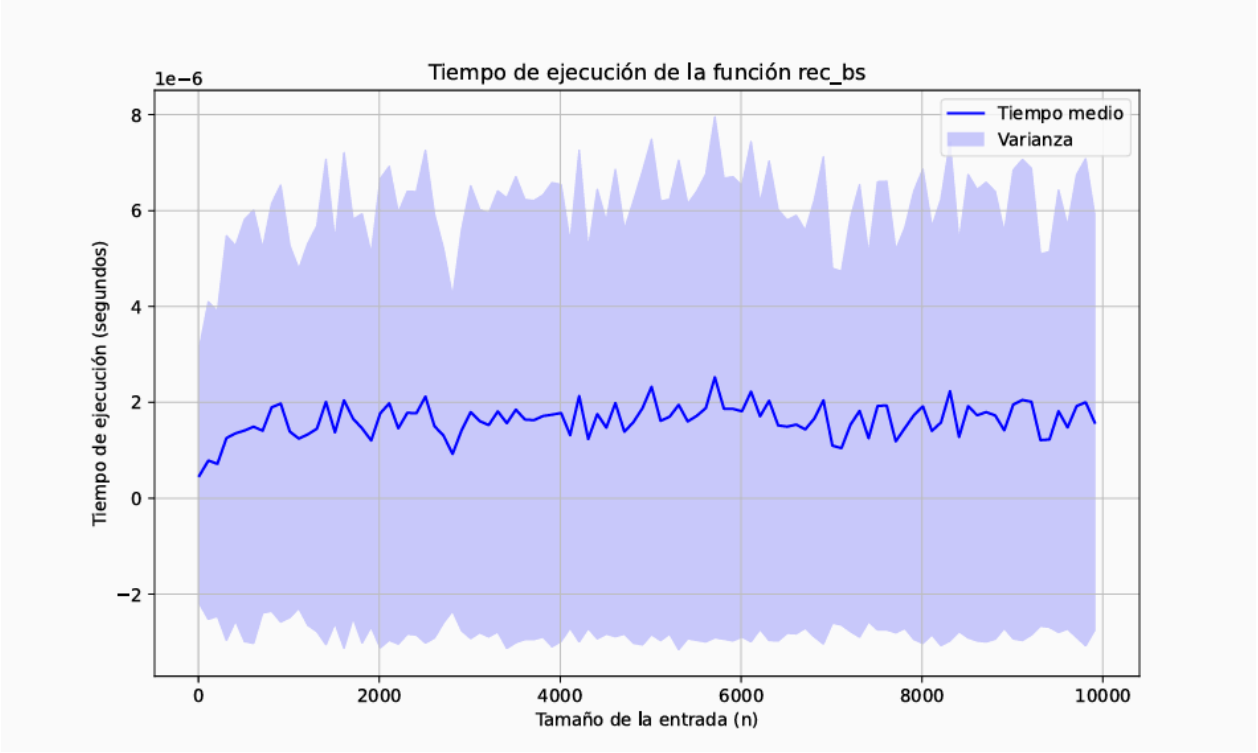
- **Varianza**: Se observa una fluctuación considerable en la varianza, aunque sin un patrón claro. Esto podría deberse a factores externos que afectan las mediciones, como la carga de la CPU en cada ejecución, o también al hecho de que, al tratarse de tiempos tan pequeños, las mediciones son más sensibles a estos factores.



2. Gráfica de la función rec_bs (Recursive Binary Search)

- **Comportamiento del tiempo medio**: Similar a la versión iterativa (itr_bs), la función recursiva de búsqueda binaria mantiene un tiempo de ejecución estable, que aumenta ligeramente con el tamaño de la entrada. Este comportamiento también es coherente con la complejidad *O(log n)*, propia de los algoritmos de búsqueda binaria.

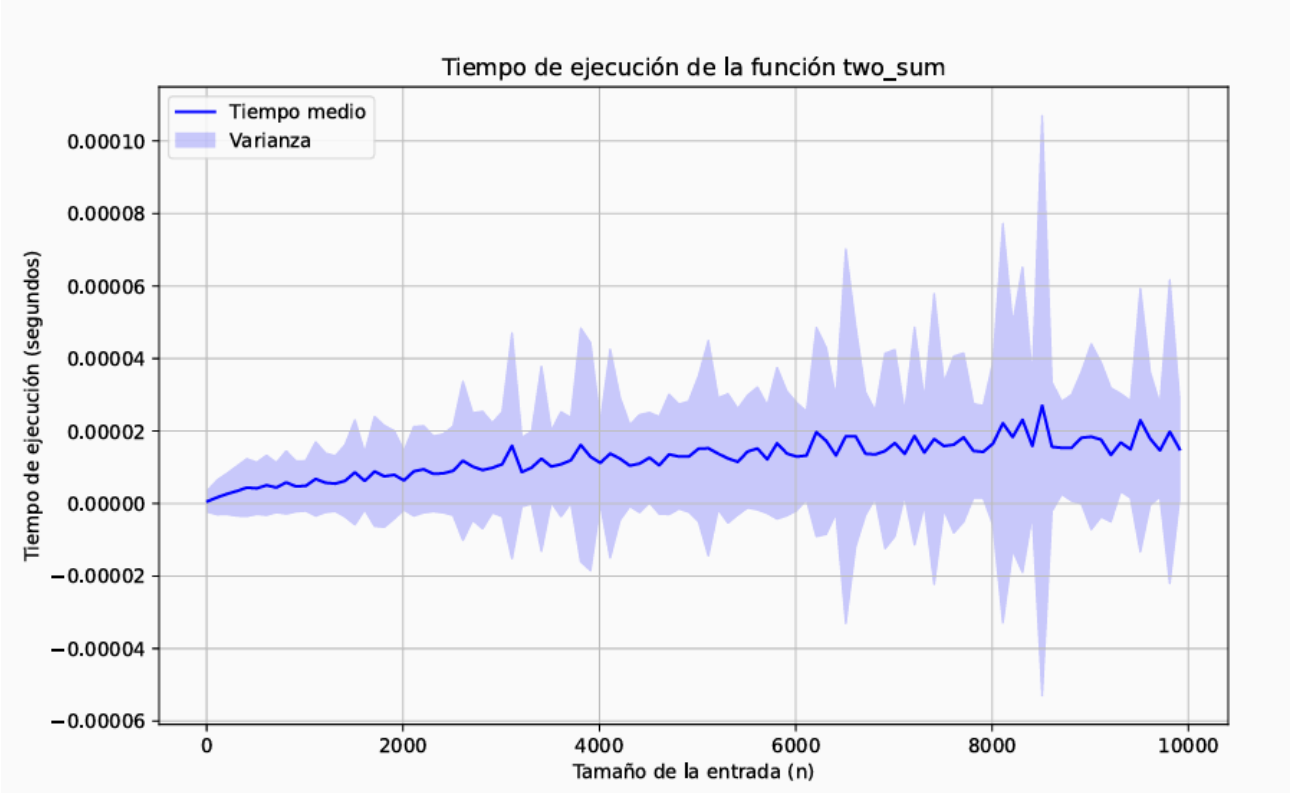
- **Varianza**: La varianza en este caso también es considerable, pero se muestra ligeramente más estable que en la gráfica de itr_bs. Al ser una versión recursiva, podría estar afectada por la sobrecarga del manejo de la pila de llamadas en cada recursión, lo que podría influir en las mediciones y en la varianza.



3. Gráfica de la función two_sum

- **Comportamiento del tiempo medio**: A diferencia de las funciones de búsqueda binaria, el tiempo medio de ejecución de two_sum aumenta de manera más notable a medida que el tamaño de la entrada crece, aunque de forma lineal. Esto es coherente con una complejidad temporal aproximada de *O(n)*, ya que el algoritmo recorre cada elemento de la lista y utiliza una estructura de datos para registrar los elementos vistos.

- **Varianza**: En este caso, la varianza es más notable y fluctúa con el tamaño de la entrada, presentando picos considerables en algunos tamaños. Esto puede deberse a la generación aleatoria de los datos de prueba, que afecta los tiempos de ejecución dependiendo de la distribución de los elementos en la lista y la facilidad con la que se encuentra una solución.



Comparaciones

Comparación entre itr_bs y rec_bs:

Las versiones iterativa y recursiva de la búsqueda binaria tienen un comportamiento similar en cuanto a la eficiencia temporal. Sin embargo, la recursiva presenta una varianza más estable, mientras que la iterativa presenta más picos, posiblemente debido a la carga del sistema en diferentes momentos. Ambos algoritmos muestran un crecimiento logarítmico en su tiempo de ejecución, lo cual es esperado para la búsqueda binaria.

Comparación entre two_sum y búsqueda binaria:

El tiempo medio de la función two_sum crece linealmente con el tamaño de la entrada, lo cual es menos eficiente en comparación con los algoritmos de búsqueda binaria (que crecen logarítmicamente). Sin embargo, two_sum utiliza un enfoque más simple de recorrido, y su varianza también es más significativa, lo que puede deberse a la aleatoriedad de los datos de entrada.

Es decir, los algoritmos de búsqueda binaria son significativamente más eficientes para entradas grandes, mientras que two_sum, aunque tiene un comportamiento lineal, sigue siendo aceptablemente rápido para tamaños moderados de entrada. Las fluctuaciones en la varianza son más evidentes en two_sum, creemos que esto es debido a la naturaleza del problema.

Mejoras al algoritmo

Primera versión de two_sum:

```
def two_sum(h, n):
    cont1 = 0
    cont2 = 1
    long = len(h)

    while cont1 != long and cont2 != long:
        v1 = h[cont1]
        v2 = h[cont2]
        res = v1 + v2
        if (res != n) and (cont2 == len(h) - 1):
            cont1 += 1
            cont2 = cont1 + 1
        elif res == n:
            return True
        else:
            cont2 += 1
    return False
```

En este código empleamos dos contadores (cont1 y cont2) para iterar por **todos los pares de elementos** en la lista. Compara cada par de elementos y comprueba si su suma es igual al valor objetivo. Si no lo es, incrementa los índices de forma manual, asegurando que se prueben todos los pares posibles.

Este enfoque tiene una **complejidad temporal** de *O(n^2)*, ya que realiza una búsqueda exhaustiva de todos los pares en la lista, lo cual es mucho menos eficiente que el enfoque de la segunda versión.

- **Desventajas**:
    - **Ineficiencia**: El enfoque tiene una complejidad cuadrática *O(n^2)*, lo que hace que el rendimiento se degrade significativamente a medida que aumenta el tamaño de la lista.
    - **Complejidad del código**: Es más complicado y menos intuitivo de seguir que la segunda versión, ya que requiere manejar índices manualmente.

Segunda versión de two_sum:

```
def two_sum(lst, target):
    seen = set()
    for num in lst:
        if target - num in seen:
            return True
        seen.add(num)
    return False
```

Para optimizarlo hemos decidido utilizar un conjunto (set) para almacenar los números que se han visto hasta el momento. Por cada número en la lista, comprueba si la diferencia entre el objetivo y el número actual está en el conjunto. Si es así, devuelve True inmediatamente.

Respecto a su complejidad, este enfoque tiene una **complejidad temporal** de *O(n)*, ya que recorre la lista una sola vez, lo que implica las siguientes ventajas:

- **Ventajas**:
    - **Eficiente**: Este enfoque es más eficiente para resolver el problema de two_sum en *O(n)*, ya que evita hacer comparaciones innecesarias entre pares de elementos.
    - **Simplicidad**: El código es sencillo y fácil de seguir.

Conclusión de las versiones de two_sum

La segunda versión es mucho más eficiente, con una complejidad *O(n)*, mientras que la primera tiene una complejidad cuadrática *O(n^2)*. Además, también es más concisa y fácil de entender. El uso de un bucle con un conjunto para almacenar los elementos ya vistos simplifica el algoritmo sin necesidad de controlar manualmente los índices como en la versión inicial.

En resumen, la **segunda versión** es una implementación más **eficiente** y **limpia**, adecuada para listas de gran tamaño (ya que con la primera versión daba problemas de ejecución con un tiempo de respuesta desproporcionado), mientras que la **versión inicial** la hicimos con una implementación de “**fuerza bruta**” la cual es poco eficiente en situaciones prácticas como la propuesta en el I.A.3.