INSTITUTO POLITÉCNICO NACIONAL

Unidad Profesional Interdisciplinaria de Ingeniería
Campus Zacatecas.

MATERIA:

- Ánalisis y Diseño de Algoritmos

Reporte 02:

- QuickSort

DOCENTE:

- Erika Sánchez Femat

Alumno:

- Gael García Torres

Introduccion

En esta práctica, implementamos el algoritmo de Quicksort en Python de dos maneras diferentes. En la primera aproximación, calculamos el pivote tomando el valor medio del arreglo, dividiéndolo entre el número de elementos en el mismo. En la segunda estrategia, seleccionamos el primer elemento del arreglo como pivote y luego realizamos comparaciones con las demás entradas para reorganizar el arreglo.

Además, en cada uno de los métodos, generamos gráficas para medir el tiempo que tomaba el proceso en función del número de entradas en el arreglo. Es importante destacar que realizamos este proceso con 100 arreglos.

Estas implementaciones nos permitieron comparar y analizar el rendimiento de ambos métodos de Quicksort en diferentes situaciones, lo que contribuyó a una comprensión más profunda de sus ventajas y desventajas

Método de QuickSor:

QuickSort, método de ordenamiento rápido. El método de ordenamiento QuickSort es actualmente el más eficiente y veloz de los métodos de ordenación interna. Este método es una mejora sustancial del método de intercambio directo y recibe el nombre de QuickSort por la velocidad con que ordena los elementos del arreglo. Es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar "n" elementos en un tiempo proporcional a "n Log n". Fue desarrollado por el científico de la computación británico Tony Hoare en 1960 y se ha convertido en uno de los algoritmos de ordenación más eficientes en la práctica.

Como funciona:

Éste algoritmo se basa en el principio de divide y conquistarás. Resulta más fácil ordenar listas pequeñas que una grande, con lo cual irá descomponiendo la lista en dos partes y ordenando esas partes.

Para ésto utiliza la recursividad.

• Dada una lista, elegir uno de sus elementos, que llamamos pivote

• Dividir la lista en dos sublistas:

• una con los elementos "menores"

• otra con los elementos "mayores"

• Ordenar recursívamente ambas sublistas

• Armar la lista resultado como: menoresOrdenados + pivote + mayoresOrdenados

La mejora es que, como al final de cada iteración el elemento mayor queda situado en su posición, ya no es necesario volverlo a comparar con ningún otro número, reduciendo así el número de comparaciones por iteración.

Eleccion del pivote sacando la media:

El algoritmo Quicksort es un algoritmo de ordenamiento eficiente que se basa en la técnica de divide y conquista. Aunque la mayoría de las implementaciones de Quicksort utilizan un pivote (un elemento del arreglo) para dividir el arreglo en dos subarreglos, la idea de usar la suma del arreglo dividida por el número de elementos como pivote no es una técnica estándar en Quicksort.

Elección del pivote: En lugar de seleccionar un elemento del arreglo como pivote, tomarías la suma de todos los elementos en el arreglo y la dividirías entre el número de elementos en el arreglo. Esto te daría un valor que sería tu pivote.

División del arreglo: Ahora, necesitas dividir el arreglo original en dos subarreglos: uno con elementos menores que el pivote y otro con elementos mayores que el pivote. Para hacer esto, recorrerías el arreglo original y compararías cada elemento con el valor del pivote calculado en el paso 1. Los elementos menores que el pivote se colocarían en un subarreglo y los elementos mayores en otro.

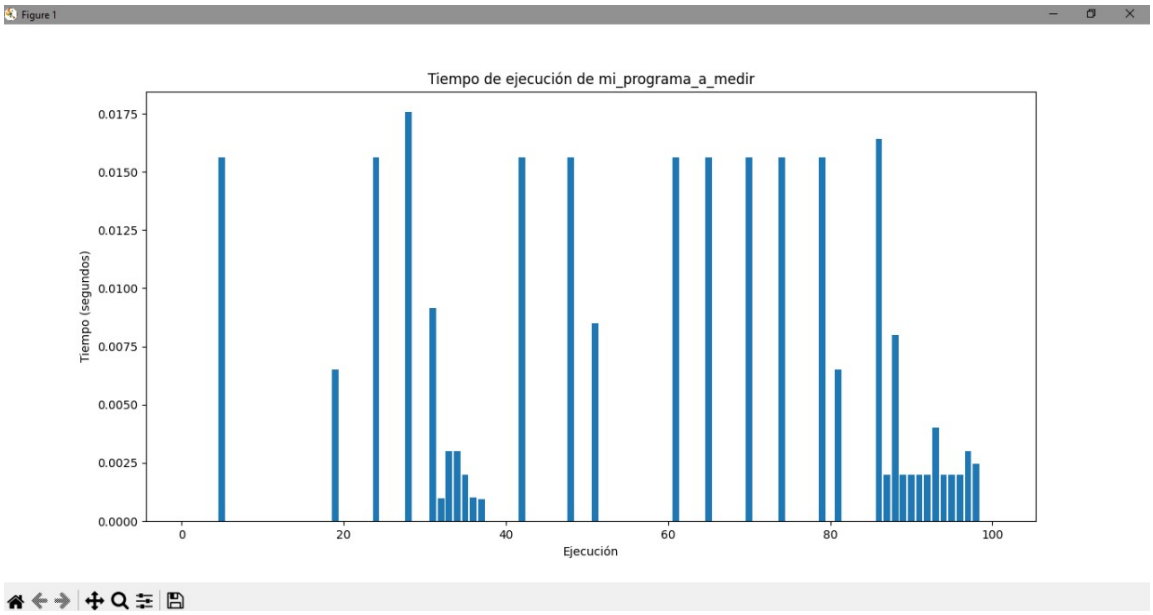
Recursión: Repetirías los pasos 1 y 2 para cada uno de los dos subarreglos resultantes. Continuarías dividiendo los subarreglos hasta que tengas subarreglos de un solo elemento. Estos subarreglos ya están ordenados por definición.

Combinación: Finalmente, combinarías todos los subarreglos ordenados en un solo arreglo ordenado.

Codigo:

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

def quicksort(arreglo):
    if len(arreglo) <= 1:
        return arreglo
    else:
    # calcular media
        pivote = sum(arreglo) / len(arreglo)
        izq = [x for x in arreglo if x < pivote]
        der = [x for x in arreglo if x > pivote]
        return quicksort(izq) + [pivote] *
        arreglo.count(pivote) + quicksort(izq)

def tiempo():

    execution_times = []  # arreglo

    # generar los 100 arreglos
    for i in range(100):
arreglo = [random.randint(1, 100) for _ in range(1000)]
        inicio = time.time()
        resultado = quicksort(arreglo)
        fin = time.time()
        tiempo_transcurrido = fin - inicio
        execution_times.append(tiempo_transcurrido)
```

```
            print(f"Arreglo",[i+1]," :",arreglo)
            print(f"Arreglo ordenado ",[i+1]," ;",resultado)
        print(f"Tiempo de ejecucion de ordenacion ",[i+1],"
        :",tiempo_transcurrido," segundos")

        # grafica
        plt.bar(range(1, 101), execution_times)
        plt.xlabel("Ejecucion")
        plt.ylabel("Tiempo (segundos)")
plt.title("Tiempo de ejecucion de mi_programa_a_medir")
        plt.show()

tiempo()
```



Al analizar la gráfica, podemos llegar a la conclusión de que a medida que disminuye el número de elementos en el arreglo, el tiempo de ejecución se reduce, mientras que con un aumento en la cantidad de elementos en el arreglo, el tiempo de ejecución se incrementa. El arreglo con menos elementos muestra un tiempo de ejecución menor a 0.0025, mientras que el arreglo con más elementos tiene un tiempo de ejecución de alrededor de 0.0175.

Eleccion del pivote usando el primer elemento:

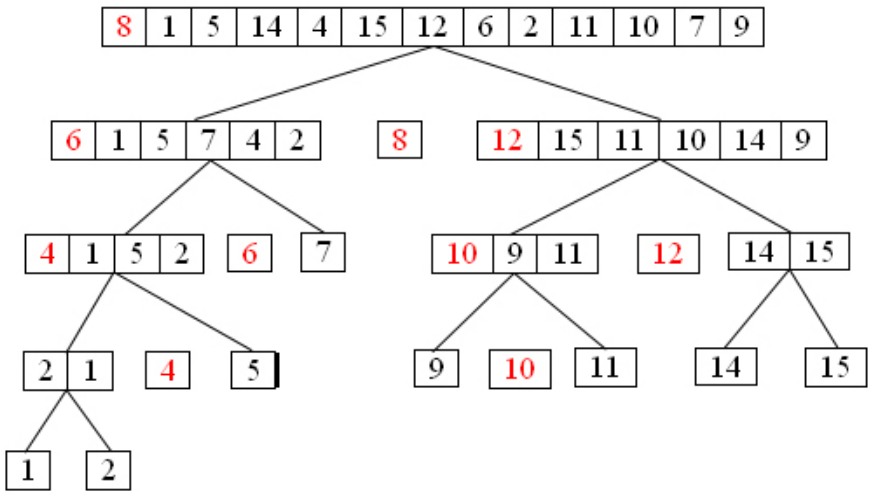
Elección del pivote: Se elige un elemento de la lista como pivote. En este caso, vamos a elegir el primer elemento de la lista.

Partición: Se recorre la lista desde el principio hasta el final, reorganizando los elementos de manera que los elementos menores o iguales al pivote estén en la parte izquierda de la lista, y los elementos mayores estén en la parte derecha. Esto se hace mediante dos índices, uno que se mueve de izquierda a derecha (i) y otro de derecha a izquierda (j). Comenzamos con i en el segundo elemento y j en el último elemento de la lista. Mientras que el elemento en la posición i sea menor o igual al pivote y i sea menor que j, incrementamos i. Mientras que el elemento en la posición j sea mayor que el pivote, decrementamos j. Si i es menor o igual a j, intercambiamos los elementos en las posiciones i y j. Repetimos este proceso hasta que i sea mayor que j.

Intercambio del pivote: Una vez que i es mayor que j, el pivote se encuentra en la posición j. Intercambiamos el pivote con el elemento en la posición j para asegurarnos de que el pivote esté en la posición correcta.

Llamadas recursivas: Dividimos la lista en dos subconjuntos: El subconjunto izquierdo que contiene elementos menores o iguales al pivote, y el subconjunto derecho que contiene elementos mayores. Luego, aplicamos el algoritmo Quicksort de forma recursiva a cada uno de estos subconjuntos.

Combinación: Cuando todas las llamadas recursivas hayan terminado, la lista estará ordenada.



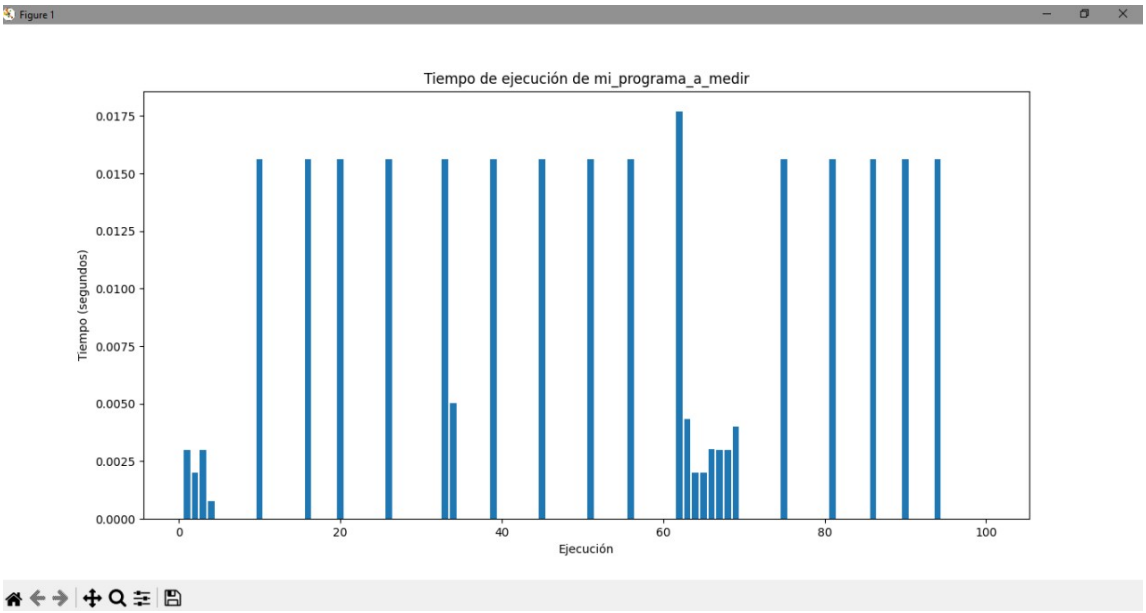
Juntando los elementos, el arreglo quedaría ordenado

1	2	4	5	6	7	8	9	10	11	12	14	15

Codigo:

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

def quicksort(arreglo):
    if len(arreglo) <= 1:
        return arreglo
    else:
        pivot = arreglo[0]
        less = [x for x in arreglo[1:] if x <= pivot]
        greater = [x for x in arreglo[1:] if x > pivot]
return quicksort(less) + [pivot] + quicksort(greater)

# Crear los 100 arreglos
arreglos = []
for _ in range(100):
arreglo = [random.randint(1, 100) for _ in range(1000)]
    arreglos.append(arreglo)

execution_times = []

for i, arreglo in enumerate(arreglos):
    start_time = time.time()
    arreglos_ordenados = quicksort(arreglo)
    end_time = time.time()
    tiempo_transcurrido = end_time - start_time
    execution_times.append(tiempo_transcurrido)
```

```
        print(f"Arreglo {i+1}: {arreglo}")
        print(f"Arreglo ordenado: {arreglos_ordenados}")
        print(f"Tiempo de ejecucion de ordenacion {i+1}:
        {tiempo_transcurrido} segundos")

# grafica
plt.bar(range(1, 101), execution_times)
plt.xlabel("Ejecucion")
plt.ylabel("Tiempo (segundos)")
plt.title("Tiempo de ejecucion de mi_programa_a_medir")
plt.show()
```



La gráfica revela datos similares, pero con una mejora notable en el tiempo de ejecución en este método en comparación con el anterior. El arreglo con menos elementos ahora requiere aproximadamente 0.0010 segundos, considerablemente más rápido que el valor previo de 0.0025 segundos. En contraste, el arreglo con más elementos sigue teniendo un tiempo de ejecución de 0.0175 seg.

Conclusion:

Quicksort es un eficiente algoritmo de ordenación utilizado en informática que opera dividiendo una lista de elementos en subgrupos más pequeños, ordenándolos y luego combinándolos.

En resumen, Quicksort es un algoritmo de ordenación eficiente que se basa en la estrategia de dividir y conquistar. Su complejidad promedio de O(n log n) lo hace una opción preferida para ordenar listas en la mayoría de las situaciones. Sin embargo, es importante considerar la elección del pivote y la implementación para lograr un rendimiento óptimo en la práctica.