INSTITUTO POLITÉCNICO NACIONAL

Unidad Profesional Interdisciplinaria de Ingeniería
Campus Zacatecas.

MATERIA:

- Ánalisis y Diseño de Algoritmos

Reporte 01:

- QuickSort

DOCENTE:

- Erika Sánchez Femat

Alumno:

- Luis Eduardo Espino Gutierrez

Introduccion

El método Quicksort es uno de los algoritmos de ordenación más icónicos y eficientes en el campo de la informática y las ciencias de la computación. Su enfoque divide y conquista, junto con su capacidad para realizar clasificaciones rápidas y efectivas, lo convierte en una herramienta fundamental para organizar conjuntos de datos de manera eficiente.

En este algoritmo, se elige un elemento como "pivote" y se reorganizan los datos de manera que los elementos más pequeños que el pivote estén a la izquierda, y los elementos más grandes estén a la derecha. Luego, este proceso se repite de manera recursiva en las sublistas generadas a ambos lados del pivote hasta que todo el conjunto de datos esté ordenado.

Desarrollo

"QUICKSORT"

QUE ES...

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".

Eleccion del pivote sacando la media...

Suma de elementos y cálculo del pivote: Inicias calculando la suma de todos los elementos en el arreglo. Luego, divides esta suma por la cantidad de elementos en el arreglo para obtener un valor promedio. Este valor promedio se toma como el pivote.

Partición: A continuación, realizas la partición del arreglo de la misma manera que en el Quicksort estándar. Recorres el arreglo y mueves los elementos menores que el pivote a la izquierda y los elementos mayores a la derecha.

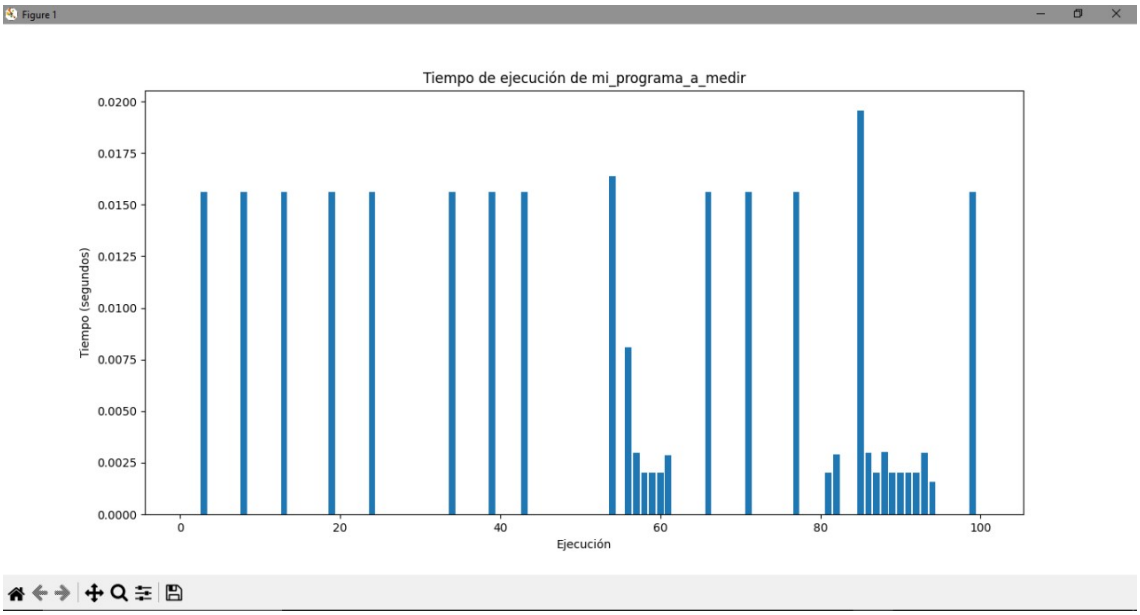
Llamadas recursivas: Después de la partición, aplicas el algoritmo Quicksort de forma recursiva a las dos sublistas resultantes: una con elementos menores que el pivote y otra con elementos mayores.

Repetición: Repites el proceso de partición y llamadas recursivas hasta que todas las sublistas estén ordenadas (tengan un solo elemento o estén vacías).

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()
```



En la gráfica, se puede apreciar que el tiempo necesario para completar la tarea varía significativamente según la cantidad de elementos en el arreglo. En particular, el arreglo con menos elementos requiere aproximadamente 0.0025 segundos, mientras que los arreglos con más elementos demandan alrededor de 0.0200 segundos. Estos resultados indican una clara dependencia entre el tiempo de ejecución y el tamaño de la entrada.

Primer elemnto como Pivote...

Se selecciona el primer elemento de la lista como el pivote. Este elemento "x" servirá como punto de referencia para dividir la lista en dos subconjuntos: aquellos menores o iguales a "x" y aquellos mayores a "x".

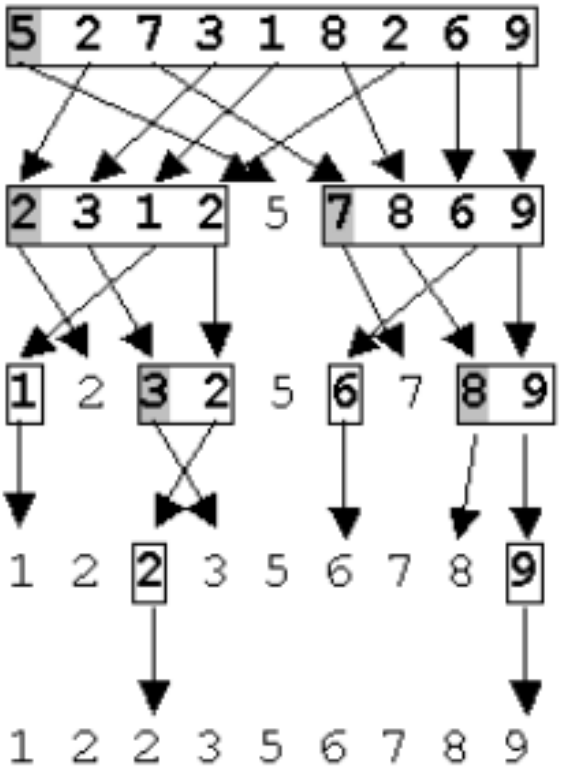
Se reorganiza la lista de tal manera que todos los elementos menores o iguales a "x" estén a la izquierda del pivote, y todos los elementos mayores estén a la derecha.

Se repiten los pasos anteriores, pero ahora para los subconjuntos de datos que se encuentran a la izquierda y a la derecha de la posición correcta del pivote "x" en el arreglo. Esto implica aplicar el mismo proceso de partición y reordenamiento de elementos de forma recursiva en ambos subconjuntos.

A medida que se realizan las llamadas recursivas y se continúa dividiendo la lista en subconjuntos más pequeños, el pivote "x" eventualmente ocupará su posición correcta en la lista ordenada, ya que estará rodeado por elementos menores o iguales a él en la parte izquierda y por elementos mayores en la parte derecha.

El proceso de repetir esta partición y reordenamiento de elementos de forma recursiva continúa hasta que todos los subconjuntos tengan un solo elemento o estén vacíos. En ese momento, todo

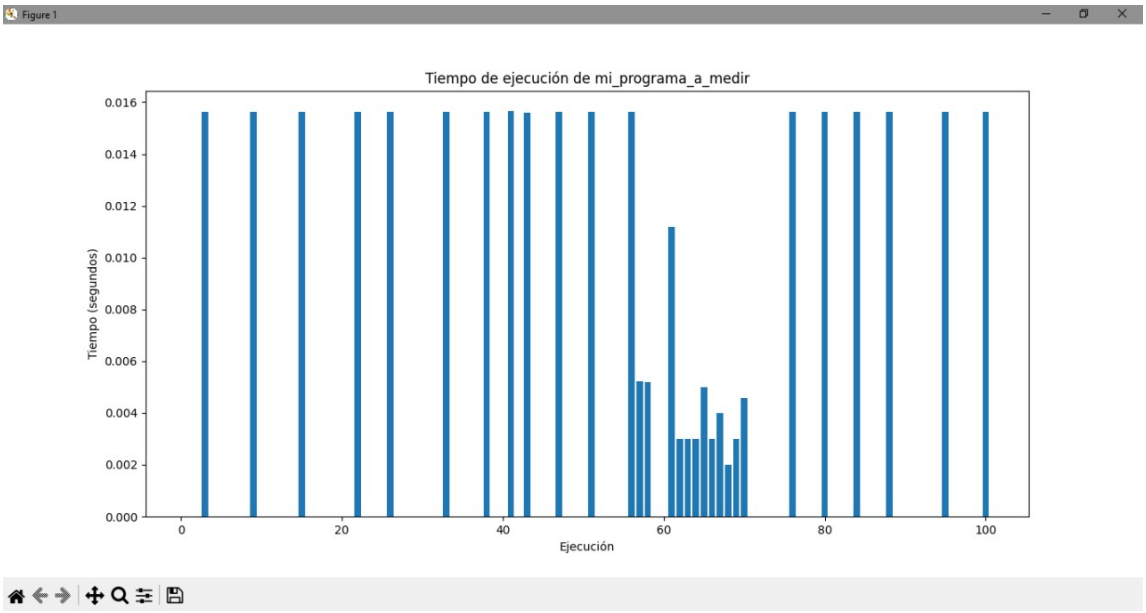
Ejemplo:



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()
```



En la gráfica, se puede apreciar que el tiempo necesario para completar la tarea varía significativamente según la cantidad de elementos en el arreglo. En particular, el arreglo con menos elementos requiere aproximadamente 0.002 segundos, mientras que los arreglos con más elementos demandan alrededor de 0.016 segundos. La diferencia principal esta en el método de selección del pivote. En el método donde se elige el primer pivote, se requiere menos tiempo en comparación con el que calcula la media para seleccionar el pivote, el cual tarda más tiempo.

En conclusión, el método Quicksort es un algoritmo de ordenación extremadamente eficiente que presenta varias ventajas y algunas desventajas notables:

Ventajas: Velocidad de Ejecución: Quicksort destaca por su rápido rendimiento promedio, lo que lo convierte en una elección excelente para ordenar conjuntos de datos grandes. Su tiempo de ejecución promedio es O(n log n), lo que lo hace muy eficiente en la práctica. Dividir y conquistar: El enfoque dividir y conquistar permite una implementación elegante y eficiente del algoritmo, lo que lo hace fácil de entender y programar.

Desventajas: Inestabilidad: Quicksort no garantiza la estabilidad en la clasificación. Esto significa que puede cambiar el orden relativo de elementos con claves iguales.

En resumen, Quicksort es un algoritmo de ordenación poderoso y eficiente que brinda un excelente rendimiento promedio y es especialmente útil para grandes conjuntos de datos. Sin embargo, su desempeño en el peor caso y su falta de estabilidad son aspectos a considerar al decidir si es la mejor opción para una tarea de ordenación específica. La elección de un pivote adecuado y una implementación cuidadosa son fundamentales para aprovechar al máximo las ventajas de Quicksort.