Ejercicio 1 – Condición del algoritmo Quicksort

Durante todo el proceso del algoritmo Quicksort se mantiene una regla muy clara: todos los elementos que están ubicados a la izquierda del pivote son menores que el pivote, mientras que los que están a la derecha son mayores o iguales. Esta condición se cumple en cada partición que se realiza, y es lo que permite que el algoritmo funcione correctamente.

Ejercicio 2 – Quicksort paso a paso (con pivote especial)

En el documento no hay una resolución escrita de este punto. Solo figura una nota que dice "Insertar foto de cuaderno". Se espera que se simule paso a paso el algoritmo Quicksort aplicando una función que seleccione como pivote el mayor de los dos primeros elementos del conjunto. También se deben responder estas preguntas:

- Cuántas veces se llama al método quicksort (contando la primera llamada)
- Cuál es la máxima profundidad de recursión alcanzada
- Cómo se puede medir esa profundidad en Java

Una forma de medirlo sería agregar una variable global que registre el nivel actual de recursión y actualice el valor máximo si ese nivel es mayor al anterior.

Perfecto, vamos a resolver **el Ejercicio 2** de la **UT9 PD3** exactamente como lo pide la consigna, paso a paso.

EJERCICIO 2

Conjunto de datos a ordenar:

44 - 55 - 12 - 42 - 94 - 18 - 6 - 67

Reglas de la consigna:

- Usar **Quicksort**.
- La función encuentraPivote selecciona **el mayor de los dos primeros elementos del subarray actual** como pivote.
- Mostrar **paso a paso cómo se va ordenando**.
- Responder:

1. ¿Cuántas llamadas se hacen a quicksort (contando la inicial)?
2. ¿Cuál es el nivel máximo de recursión alcanzado?
3. ¿Cómo podrías medir ese nivel en una implementación en Java?

Paso a paso del algoritmo

Trabajamos con el array:
[44, 55, 12, 42, 94, 18, 6, 67]

- **Llamada 1 (nivel 1)**

Subarray: [44, 55, 12, 42, 94, 18, 6, 67]
Primeros dos elementos: 44 y 55 → pivote = **55**

Partición:

- Menores a 55: [44, 12, 42, 18, 6]
- Pivote: 55
- Mayores o iguales: [94, 67]

Queda:
[44, 12, 42, 18, 6] + [55] + [94, 67]

Llamadas recursivas a:

- [44, 12, 42, 18, 6] (izquierda)
- [94, 67] (derecha)

- **Llamada 2 (nivel 2)**

Subarray: [44, 12, 42, 18, 6]
Primeros dos: 44 y 12 → pivote = **44**

Partición:

- Menores a 44: [12, 42, 18, 6]
- Pivote: 44
- Mayores o iguales: []

Queda:
[12, 42, 18, 6] + [44]

Llamada recursiva a:

- [12, 42, 18, 6] (izquierda)

❖ Llamada 3 (nivel 3)

Subarray: [12, 42, 18, 6]
Primeros dos: 12 y 42 → pivote = **42**

Partición:

- Menores a 42: [12, 18, 6]
- Pivote: 42
- Mayores o iguales: []

Queda:
[12, 18, 6] + [42]

Llamada recursiva a:

- [12, 18, 6] (izquierda)

❖ Llamada 4 (nivel 4)

Subarray: [12, 18, 6]
Primeros dos: 12 y 18 → pivote = **18**

Partición:

- Menores a 18: [12, 6]
- Pivote: 18
- Mayores o iguales: []

Queda:
[12, 6] + [18]

Llamada recursiva a:

- [12, 6] (izquierda)

❖ Llamada 5 (nivel 5)

Subarray: [12, 6]
Primeros dos: 12 y 6 → pivote = **12**

Partición:

- Menores a 12: [6]
- Pivote: 12
- Mayores o iguales: []

Queda:
[6] + [12]

Llamada recursiva a:

- [6] (tamaño 1, ya ordenado)

❖ **Llamada 6 (nivel 2)**

Volvemos a rama derecha del paso 1: subarray [94, 67]
Primeros dos: 94 y 67 → pivote = **94**

Partición:

- Menores: [67]
- Pivote: 94
- Mayores o iguales: []

Queda:
[67] + [94]

Array ordenado final:

[6, 12, 18, 42, 44, 55, 67, 94]

Respuestas finales

1. ¿Cuántas llamadas a Quicksort se hicieron (contando la inicial)?

6 llamadas, una por cada nivel:

- Llamada 1: [44, 55, 12, 42, 94, 18, 6, 67]
- Llamada 2: [44, 12, 42, 18, 6]
- Llamada 3: [12, 42, 18, 6]
- Llamada 4: [12, 18, 6]

- Llamada 5: [12, 6]
- Llamada 6: [94, 67]

(El resto de subarrays tenían tamaño 1, así que no se vuelven a llamar)

---

2. ¿Cuál fue la máxima profundidad recursiva?

5 niveles de profundidad, en el camino:
[44, 55, ...] → [44, 12, ...] → [12, 42, ...] → [12, 18, 6] → [12, 6] → [6]

---

3. ¿Cómo podrías medir ese nivel en Java?

Una forma sencilla sería usar una variable estática que se actualice cada vez que se entra a quicksort. Por ejemplo:

```
static int profundidadActual = 0;

static int maxProfundidad = 0;


public void quicksort(int[] array, int inicio, int fin) {
   profundidadActual++;
   if (profundidadActual > maxProfundidad) {
      maxProfundidad = profundidadActual;
   }

   // lógica de partición y llamadas recursivas

   profundidadActual--; // al salir de la recursión
}
```

¿Querés que te lo deje en un formato para entregar o copiar en Word?

Ejercicio 3 – Nueva versión de Quicksort en Java (Dual-Pivot)

A partir de Java 7, el método Arrays.sort() utiliza una versión mejorada del algoritmo Quicksort, conocida como dual-pivot quicksort. Esta versión fue propuesta por Yaroslavskiy en 2009.

En lugar de un solo pivote, esta versión usa dos, dividiendo el array en tres partes. Eso mejora la eficiencia en varios casos, especialmente cuando hay muchos elementos repetidos.

Esta versión es más rápida en promedio que el quicksort clásico. El tiempo de ejecución promedio sigue siendo O(n log n), y el peor caso es $O(n^2)$, aunque este último es muy poco probable gracias a la estrategia que se usa para elegir los pivotes.

Además, en situaciones donde hay muchas claves repetidas, algunas variantes como la de tres vías (3-way) pueden llegar a un tiempo lineal O(n), y la implementación de Java hereda algunas ideas de esa técnica para optimizar aún más el dual-pivot.

Ejercicio 4 – Análisis de Quicksort y mejora para conjuntos pequeños

El algoritmo Quicksort divide el arreglo en dos partes alrededor de un pivote. A la izquierda quedan los valores menores, y a la derecha, los valores mayores o iguales. Luego aplica el mismo proceso de forma recursiva en cada mitad.

Este algoritmo sigue el enfoque divide y vencerás.

La función de partición que separa los elementos tiene un costo lineal O(n), porque recorre el arreglo una vez para ubicar los elementos en su lugar con respecto al pivote.

En cuanto a la complejidad del algoritmo:

- En el mejor de los casos, el pivote divide el arreglo en dos partes iguales, lo cual da un tiempo total de O(n log n)
- En el peor de los casos, el pivote siempre es el valor más chico o más grande, lo que lleva a una complejidad de $O(n^2)$
- En promedio, el tiempo sigue siendo O(n log n)

Para mejorar el rendimiento en arreglos pequeños, se puede combinar Quicksort con otro algoritmo más sencillo como Insertion Sort. Por ejemplo, si el subarray tiene menos de 10 elementos, en lugar de seguir con la recursión se puede aplicar

directamente Insertion Sort, que es más rápido para tamaños pequeños o casi ordenados.