




NOVIEMBRE DE 2024

# ¿QUÉ ES BIG O?

NOTACIÓN Y EJEMPLOS DE ALGORITMOS DE ORDENAMIENTO

JAIME ERNESTO GIL DIAZ

ACTIVIDAD EXTRA  
SIC 2024



## INDICE

1. INTRODUCCIÓN.....	2
2. OBJETIVOS .....	3
2.1. OBJETIVO GENERAL:.....	3
2.2. OBJETIVOS ESPECIFICOS:.....	3
3. JUSTIFICACIÓN.....	4
4. MARCO TEORICO.....	5
4.1. Conceptos:.....	5
4.2. Descripción y comparación .....	8
4.3. Comparación entre Clasificación de Burbujas, Ordenación por Selección y Ordenación por Inserción .....	14
4.4. Ordenación rápida frente a ordenación combinada.....	16
5. ANEXOS .....	18
5.1. Código En Java.....	18
5.2. Código En C++.....	20
5.3. Código En Piton .....	23
5.4. Código En C#.....	25
5.5. Código En JavaScript.....	27
5.6. Resultados .....	30
5.7. Código de Búsqueda binaria.....	31
5.8. Código de Selection Sort .....	32
5.9. Código de Insertion Sort.....	33
5.10. Código de Merge Sort .....	34
5.11. Código de Quick Sort.....	36
5.12. Complejidad .....	37
6. BIBLIOGRAFIA.....	38

## ***1. INTRODUCCIÓN***

En el ámbito de la ciencia de la computación y el desarrollo de software, la eficiencia de los algoritmos es fundamental, especialmente cuando se trabaja con grandes volúmenes de datos o sistemas que requieren respuestas rápidas. Para analizar y comparar la eficiencia de diferentes algoritmos, se utiliza la notación Big O, una herramienta que permite describir la complejidad de un algoritmo en términos de tiempo y espacio en función del tamaño de su entrada. Esta notación es crucial para entender cómo se comporta un algoritmo a medida que crece la cantidad de datos a procesar, ayudando así a seleccionar o diseñar soluciones óptimas.

Entre los algoritmos clásicos que suelen ser objeto de estudio en la teoría de la complejidad se encuentran los algoritmos de búsqueda y ordenación. En este documento, abordaremos el análisis de la notación Big O de algunos de los algoritmos de búsqueda y ordenación más comunes: la Búsqueda Binaria, Bubble Sort, Selection Sort e Insertion Sort. Estos algoritmos representan distintos enfoques y grados de eficiencia en la organización y búsqueda de datos, y su análisis proporciona una base sólida para entender los principios de la optimización algorítmica.

La Búsqueda Binaria es conocida por su eficiencia en listas previamente ordenadas, ya que divide el espacio de búsqueda en cada paso, mientras que algoritmos de ordenación como Bubble Sort, Selection Sort e Insertion Sort, aunque menos eficientes en algunos casos, son fáciles de implementar y entender. Cada uno de estos algoritmos tiene diferentes comportamientos según el caso (mejor, promedio y peor), y la elección de uno sobre otro puede depender tanto de las características específicas de los datos como de los requisitos del sistema.

A lo largo de este documento, se analizarán estos algoritmos en términos de su complejidad temporal y espacial mediante la notación Big O, con el objetivo de proporcionar una visión clara y fundamentada de sus capacidades y limitaciones. Este análisis servirá de base para aquellos interesados en optimizar aplicaciones y diseñar sistemas de software eficientes y escalables.

## **2. OBJETIVOS**

### **2.1. OBJETIVO GENERAL:**

Analizar la complejidad y eficiencia de los algoritmos de Búsqueda Binaria, Bubble Sort, Selection Sort e Insertion Sort utilizando la notación Big O, para comprender su comportamiento en diferentes escenarios y optimizar la selección de algoritmos en aplicaciones de desarrollo de software.

### **2.2. OBJETIVOS ESPECIFICOS:**

- Explicar la notación Big O y su importancia en la evaluación de la eficiencia temporal y espacial de los algoritmos.
- Evaluar la complejidad en el mejor, promedio y peor caso de los algoritmos de Búsqueda Binaria, Bubble Sort, Selection Sort e Insertion Sort.
- Comparar las ventajas y desventajas de cada algoritmo en función de su complejidad y aplicabilidad a diferentes tipos de datos y requerimientos de software.

### ***3. JUSTIFICACIÓN***

La eficiencia de los algoritmos es un factor crucial en el rendimiento y la escalabilidad de los sistemas de software, especialmente en entornos donde se manejan grandes volúmenes de datos o se requieren tiempos de respuesta rápidos. Entender la complejidad computacional de algoritmos básicos como Búsqueda Binaria, Bubble Sort, Selection Sort e Insertion Sort mediante la notación Big O permite a los desarrolladores seleccionar las soluciones más adecuadas para cada situación. Este análisis no solo optimiza el uso de recursos del sistema, sino que también facilita el diseño de aplicaciones más rápidas y sostenibles. Además, la comprensión de estos conceptos es esencial para el aprendizaje de técnicas avanzadas en algoritmia y estructura de datos, beneficiando tanto el rendimiento de los programas como el desarrollo profesional en el campo de la computación.

## 4. MARCO TEORICO

### 4.1. Conceptos:

"Big O" es una notación matemática utilizada para describir el comportamiento de un algoritmo en términos de su eficiencia, especialmente en cuanto a la cantidad de tiempo o espacio que requiere a medida que aumenta el tamaño de la entrada ( $n$ ). Es una medida para clasificar y comparar la complejidad de diferentes algoritmos en función de su rendimiento en el peor de los casos, aunque también se puede aplicar a casos promedio o mejores.

Algunos ejemplos comunes de notaciones Big O:

1.  $O(1)$  - Constante: La operación siempre tarda el mismo tiempo, sin importar el tamaño de la entrada. Ejemplo: Acceder a un elemento específico de un array.
2.  $O(\log n)$  - Logarítmica: A medida que el tamaño de la entrada aumenta, el tiempo de ejecución crece lentamente. Ejemplo: Búsqueda binaria en un array ordenado.
3.  $O(n)$  - Lineal: El tiempo de ejecución crece proporcionalmente al tamaño de la entrada. Ejemplo: Recorrer todos los elementos de un array.
4.  $O(n \log n)$  - Logarítmica lineal: Común en algoritmos de ordenación eficientes como Merge Sort o Quick Sort en su caso promedio.
5.  $O(n^2)$  - Cuadrática: El tiempo de ejecución crece al cuadrado del tamaño de la entrada, común en algoritmos que requieren bucles anidados, como en una búsqueda de pares.
6.  $O(2^n)$  - Exponencial: El tiempo de ejecución crece exponencialmente, siendo ineficiente para entradas grandes. Ejemplo: Algoritmos de fuerza bruta como el de la subsecuencia de máxima longitud.
7.  $O(n!)$  - Factorial: Extremadamente ineficiente, el tiempo de ejecución crece factorialmente. Ejemplo: Algoritmos de permutaciones.

La notación Big O te ayuda a comprender qué tan bien o mal puede escalar un algoritmo y te permite elegir el más adecuado para un problema, especialmente cuando se trabaja con grandes cantidades de datos.

Claro, Big O no solo es una medida para describir el tiempo o el espacio en el peor de los casos; también ayuda a comprender y analizar el rendimiento de algoritmos en términos de eficiencia. Aquí algunos puntos adicionales que pueden darte un entendimiento más completo:

#### 1. Propósito de Big O

Big O permite entender la eficiencia relativa de los algoritmos sin entrar en detalles específicos de implementación, como el hardware o el lenguaje de programación. Esto es útil

porque las variaciones de hardware o las optimizaciones específicas no afectan la eficiencia general que describe la notación Big O.

## 2. Comparación entre algoritmos

Big O ayuda a decidir entre varios algoritmos para resolver un problema al enfocarse en el crecimiento de su complejidad conforme aumenta el tamaño de la entrada. Por ejemplo, en listas desordenadas, usar un algoritmo de búsqueda lineal ( $O(n)$ ) puede ser rápido en listas pequeñas, pero en listas grandes, una búsqueda binaria ( $O(\log n)$ ) en una lista ordenada será mucho más eficiente.

## 3. Tipos de complejidades y su impacto

Cada tipo de complejidad tiene un impacto diferente en la eficiencia del algoritmo. Aquí se desglosan algunos de los tipos más comunes:

- Complejidad Constante ( $O(1)$ ): Ideal, porque no importa el tamaño de la entrada, el tiempo de ejecución permanece constante.
- Complejidad Logarítmica ( $O(\log n)$ ): Muy eficiente en grandes entradas. Se encuentra en algoritmos de búsqueda binaria.
- Complejidad Lineal ( $O(n)$ ): Escalable y eficiente, ya que crece de forma proporcional al tamaño de la entrada. Es común en algoritmos de recorrido.
- Complejidad Cuadrática ( $O(n^2)$ ): Menos eficiente para entradas grandes. Suele aparecer en algoritmos de fuerza bruta con bucles anidados, como en algunos métodos de ordenación (por ejemplo, Bubble Sort).
- Complejidad Exponencial ( $O(2^n)$ ) y Factorial ( $O(n!)$ ): Usualmente es una "mala práctica" utilizar algoritmos con estas complejidades para entradas grandes, ya que el tiempo de ejecución crece demasiado rápido.

## 4. Notación Big Theta ( $\Theta$ ) y Big Omega ( $\Omega$ )

Aunque Big O se usa para el peor caso, existen otras notaciones relacionadas:

- Big Theta ( $\Theta(n)$ ): Describe la complejidad en el caso promedio, indicando que el tiempo de ejecución crecerá de manera similar a  $(n)$  en promedio.
- Big Omega ( $\Omega(n)$ ): Describe la complejidad en el mejor caso, representando el mínimo tiempo de ejecución que un algoritmo necesitará.

Estas notaciones son menos usadas en la práctica, pero son útiles para una comprensión completa del rendimiento de un algoritmo.

## 5. Optimización y Escalabilidad

Cuando los desarrolladores seleccionan o diseñan algoritmos, tener en cuenta la notación Big O es clave para crear software escalable. Un algoritmo que funciona bien con entradas pequeñas podría no ser adecuado cuando se enfrentan grandes volúmenes de datos, como ocurre en aplicaciones de Big Data o Machine Learning.

## 6. Reglas para simplificar la notación Big O

La notación Big O suele simplificarse para centrarse en la parte dominante, o la que más influye en el crecimiento del tiempo o espacio. Aquí algunas reglas:

- Ignorar constantes multiplicativas:  $O(2n)$  se simplifica a  $O(n)$ , ya que solo el crecimiento en función de  $(n)$  importa, no los multiplicadores constantes.
- Usar solo el término más grande: En  $O(n + n^2)$ , la notación se simplifica a  $O(n^2)$  porque  $(n^2)$  crece más rápido que  $(n)$  cuando  $(n)$  es grande.

## 7. Ejemplos prácticos de Big O en algoritmos comunes

- Búsqueda lineal:  $O(n)$
- Búsqueda binaria en un arreglo ordenado:  $O(\log n)$
- Bubble Sort:  $O(n^2)$
- Merge Sort:  $O(n \log n)$
- Fibonacci recursivo:  $O(2^n)$ , por su recursividad sin memorización

## 8. Big O en espacio (memoria)

Además del tiempo, Big O se aplica a la cantidad de memoria que un algoritmo necesita. Algunos algoritmos que son rápidos (como la búsqueda binaria) también pueden tener una baja complejidad de espacio, mientras que otros, como los que requieren mucha memoria adicional para funcionar, pueden tener una alta complejidad espacial.



En resumen, la notación Big O es esencial para elegir el algoritmo adecuado y hacer que los programas sean eficientes y escalables, especialmente cuando se trabaja con grandes volúmenes de datos.

#### **4.2. Descripción y comparación**

La notación Big O es una manera de describir la rapidez o complejidad de un algoritmo dado. Si tu proyecto actual requiere un algoritmo predefinido, es importante entender qué tan rápido o lento es comparado con otras opciones.

##### **¿Qué es la notación Big O y cómo funciona?**

En palabras simples, la notación Big O te dice el número de operaciones que hará un algoritmo. Toma su nombre de la "O grande" en frente del número estimado de operaciones.

Lo que la notación Big O no te dice es la rapidez del algoritmo en segundos. Hay demasiados factores que influyen en el tiempo que tarde en ejecutarse un algoritmo. En su lugar, usarás la notación Big O para comparar diferentes algoritmos por el número de operaciones que hacen.

##### **Big O establece un tiempo de ejecución en el peor de los casos**

Imagina que eres un maestro y tienes una estudiante de nombre Jane. Quieres encontrar sus registros, así que usas un algoritmo de búsqueda simple para recorrer las bases de datos de tu distrito escolar.

Sabes que a la búsqueda simple le toma  $O(n)$  veces ejecutarse. Esto significa que, en el peor de los casos, tendrás que buscar en cada uno de los registros (representados por  $n$ ) para encontrar el de Jane.

Pero cuando ejecutas la búsqueda simple, encuentras que los registros de Jane son la primera entrada en la base de datos. No tienes que mirar cada entrada – la encontraste en tu primer intento.

*¿Este algoritmo tardó  $O(n)$  tiempo? ¿O tardó  $O(1)$  porque encontraste los registros de Jane en el primer intento?*

En este caso,  $O(1)$  es el mejor de los casos: tuviste suerte de que los registros de Jane estuvieran al principio. Pero la notación Big O se enfoca en el peor de los casos, el cual es  $O(n)$  para la búsqueda simple. Es una garantía de que la búsqueda simple nunca será más lenta que el tiempo  $O(n)$ .

##### **Los tiempos de ejecución de los algoritmos crecen a ritmos diferentes**

Asume que toma 1 milisegundo revisar cada elemento de la base de datos del distrito escolar.

Con la búsqueda simple, si tienes que revisar 10 entradas, le tomará 10 ms en ejecutarse. Pero con el *algoritmo de búsqueda binaria*, solo tienes que revisar 3 elementos, lo que toma 3 ms en ejecutarse.

En la mayoría de los casos, la lista o base de datos en la que necesitas buscar tendrá cientos o miles de elementos.

Si hay mil millones de elementos, usar la búsqueda simple tardará hasta 1 mil millones de ms, u 11 días. En cambio, la búsqueda binaria tardará solamente 32 ms en el peor de los casos:

Número de elementos	Búsqueda simple	Búsqueda binaria
Tiempo de ejecución en notación Big O	$O(n)$	$O(\log n)$
10	10 ms	3 ms
100	100 ms	7 ms
10.000	10 seg	14 ms
1000.000.000	11 días	32 ms

Claramente, los tiempos de ejecución de la búsqueda simple y de la búsqueda binaria no crecen ni de cerca al mismo ritmo. Mientras más aumenta la lista de entradas, a la búsqueda binaria solo le toma un poco más de tiempo en ejecutarse. El tiempo de ejecución de la búsqueda simple crece exponencialmente mientras la lista de entradas aumente.

Por esto es muy importante saber cómo el tiempo de ejecución se incrementa en relación con el tamaño de una lista. Y aquí es exactamente donde la notación Big O es muy útil.

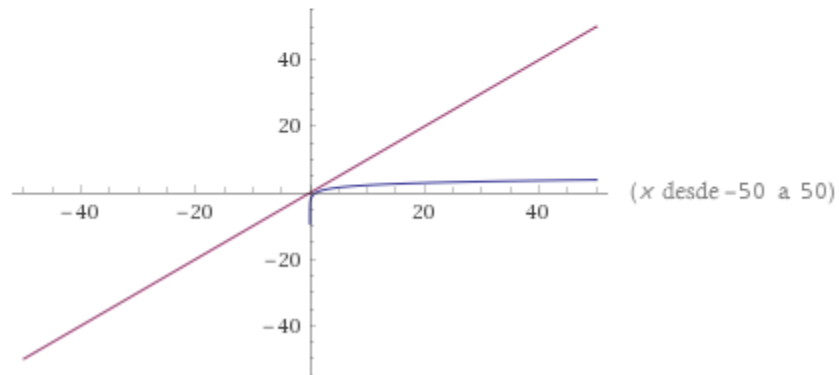
### La notación Big O muestra el número de operaciones

Como se mencionó antes, la notación Big O no muestra el *tiempo* que tardará en ejecutarse un algoritmo. En su lugar, muestra el número de operaciones que procesará. Te dice qué tan rápido crece un algoritmo y te permite compararlo con otros.

Interpretación de los datos de entrada:

puntos a trazar	$\log(x)$	$x = -50$ a $50$
	$x$	$y = -50$ a $50$

Gráfico:



He aquí algunos algoritmos comunes y sus tiempos de ejecución en notación Big O:

NOTACIÓN BIG O	ALGORITMO DE EJEMPLO
----------------	----------------------

$O(\log n)$	Búsqueda binaria
-------------	------------------

$O(n)$	Búsqueda simple
--------	-----------------

$O(n * \log n)$	Ordenación rápida (Quicksort)
-----------------	-------------------------------

$O(n^2)$	Ordenación por selección
----------	--------------------------

$O(n!)$	Vendedor viajero
---------	------------------

Ahora ya sabes lo suficiente para ser peligroso con la notación Big O. Sal allá y empieza a comparar algoritmos.

Veremos su complejidad en términos de tiempo, que suele ser el factor más crítico, en diferentes casos (mejor, promedio y peor), además de un análisis básico de su complejidad espacial.

## 1. Búsqueda Binaria

La Búsqueda Binaria es un algoritmo de búsqueda eficiente en listas ordenadas. Divide el espacio de búsqueda a la mitad en cada iteración, por lo que su complejidad crece logarítmicamente.

- Mejor caso:  $O(1)$  — Cuando el elemento a buscar está en el centro de la lista desde el inicio.
- Promedio y peor caso:  $O(\log n)$  — En cada paso, se descarta la mitad de los elementos, y el proceso se repite hasta encontrar el elemento o quedarse sin opciones.
- Complejidad espacial:  $O(1)$  en su implementación iterativa y  $O(\log n)$  en su implementación recursiva, ya que requiere almacenamiento en la pila para cada llamada recursiva.

Resumen: La búsqueda binaria es muy eficiente con  $O(\log n)$  en promedio y peor caso, pero requiere una lista previamente ordenada.

## 2. Bubble Sort

Bubble Sort es un algoritmo de ordenamiento simple pero ineficiente. Compara pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que no hay intercambios en una iteración completa.

- Mejor caso:  $O(n)$  — Si la lista ya está ordenada, solo se necesita una pasada para confirmar que no hay intercambios.
- Promedio y peor caso:  $O(n^2)$  — En listas desordenadas, el algoritmo realiza aproximadamente  $(n^2/2)$  comparaciones e intercambios.
- Complejidad espacial:  $O(1)$  — Solo requiere memoria adicional constante para realizar los intercambios.

Resumen: Bubble Sort es ineficiente para listas grandes debido a su complejidad cuadrática en promedio y peor caso.

## 3. Selection Sort

Selection Sort selecciona el elemento mínimo (o máximo) y lo coloca en la posición correcta en cada iteración. Repite este proceso para cada posición de la lista.

- Mejor, promedio y peor caso:  $O(n^2)$  — Siempre necesita  $(n^2/2)$  comparaciones, sin importar si la lista está ordenada o no.

- Complejidad espacial:  $O(1)$  — No requiere espacio adicional aparte de una variable temporal para intercambiar elementos.

Resumen: Aunque Selection Sort tiene una complejidad cuadrática en todos los casos, realiza menos intercambios que Bubble Sort, lo cual puede ser ventajoso cuando el costo de intercambio es significativo.

#### 4. Insertion Sort

Insertion Sort construye la lista ordenada de forma incremental, tomando cada elemento de la lista no ordenada y colocándolo en su posición correcta en la lista ordenada.

- Mejor caso:  $O(n)$  — Si la lista ya está ordenada, solo necesita una pasada para verificar que cada elemento está en su lugar.

- Promedio y peor caso:  $O(n^2)$  — En listas desordenadas, se necesitan aproximadamente  $(n^2/2)$  comparaciones e intercambios.

- Complejidad espacial:  $O(1)$  — Requiere solo un espacio constante adicional para intercambiar elementos.

Resumen: Insertion Sort es eficiente para listas pequeñas o parcialmente ordenadas, ya que su mejor caso es lineal y tiene una ventaja en el ordenamiento casi ordenado en comparación con otros algoritmos cuadráticos.

5. Merge Sort es un algoritmo de ordenación que divide la lista en mitades recursivamente hasta llegar a listas de un solo elemento (que se consideran ordenadas). Luego, combina estas sub-listas ordenadas en orden creciente hasta obtener la lista completamente ordenada.

- **Mejor, promedio y peor caso:**  $O(n \log n)$  — Debido a que el algoritmo siempre divide la lista en dos y luego las combina, su tiempo de ejecución en todos los casos es  $O(n \log n)$ , independientemente del orden de la lista inicial.
- **Complejidad espacial:**  $O(n)$  — Merge Sort requiere memoria adicional proporcional al tamaño de la lista para realizar las combinaciones de elementos.

Merge Sort es eficiente en todos los casos con una complejidad de  $O(n \log n)$ , pero consume más espacio en comparación con otros algoritmos debido a la memoria adicional para las combinaciones.

6. Quick Sort también utiliza el enfoque "divide y vencerás", pero en lugar de dividir la lista en mitades, selecciona un **pivote** y reorganiza los elementos alrededor de este: los elementos menores al pivote se colocan a su izquierda y los mayores, a su derecha. Luego, se aplica el mismo proceso de manera recursiva a las sub-listas.

- **Mejor y promedio caso:**  $O(n \log n)$  — En el mejor y promedio caso, Quick Sort divide la lista en sub-listas balanceadas (alrededor de la mitad), lo que le permite alcanzar una eficiencia logarítmica.
- **Peor caso:**  $O(n^2)$  — Si el pivote elegido es el menor o el mayor elemento de la lista repetidamente, el algoritmo se degrada a  $O(n^2)$  debido a que las divisiones no están balanceadas.
- **Complejidad espacial:**  $O(\log n)$  — Quick Sort suele ser más eficiente en memoria que Merge Sort, ya que no necesita espacio adicional para almacenar combinaciones, solo espacio adicional para las llamadas recursivas.

Quick Sort es eficiente y rápido en promedio con  $O(n \log n)$ , aunque su peor caso es  $O(n^2)$ . Sin embargo, con una buena elección de pivote (como pivote aleatorio o pivote mediano), este caso rara vez ocurre, y suele ser más rápido en la práctica que Merge Sort debido a su menor sobrecarga de memoria.

#### Comparación de Complejidades de Tiempo (Big O)

Algoritmo	Mejor caso	Caso promedio	Peor caso	Complejidad espacial
Búsqueda Binaria	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$ iterativa, $O(\log n)$ recursiva
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

- **Búsqueda Binaria** es una técnica de búsqueda eficiente para listas ordenadas, con un crecimiento logarítmico en el tiempo.
- **Bubble Sort**, **Selection Sort** e **Insertion Sort** tienen una complejidad de  $O(n^2)$  en el promedio y peor caso, lo que los hace poco eficientes para listas grandes. Sin embargo, Insertion Sort es lineal en el mejor caso cuando la lista está parcialmente ordenada.
- **Merge Sort** es consistente con una complejidad de  $O(n \log n)$  en todos los casos, aunque requiere mayor espacio adicional.
- **Quick Sort** suele ser más rápido que Merge Sort en promedio debido a su baja sobrecarga de memoria, aunque su peor caso  $O(n^2)$  es menos favorable.

### **4.3. Comparación entre Clasificación de Burbujas, Ordenación por Selección y Ordenación por Inserción**

La ordenación por burbujas, la ordenación por selección y la ordenación por inserción son algoritmos de ordenación sencillos que se utilizan habitualmente para ordenar pequeños conjuntos de datos o como bloques de creación para algoritmos de ordenación más complejos. Esta es una comparación de los tres algoritmos:

#### **Clasificación de burbujas:**

1. Complejidad del tiempo:  $O(n^2)$  en el peor y promedio de los casos,  $O(n)$  en el mejor de los casos (cuando la matriz de entrada ya está ordenada) Complejidad del espacio:  $O(1)$
2. Idea básica: Iterar a través de la matriz repetidamente, comparando pares de elementos adyacentes e intercambiándolos si están en el orden incorrecto. Repita hasta que la matriz esté completamente ordenada.

#### **Clasificación de selección:**

1. Complejidad del espacio:  $O(n^2)$  en todos los casos (peor, medio y mejor) Complejidad del espacio:  $O(1)$
2. Idea básica: Encuentre el elemento mínimo en la parte no ordenada de la matriz e intercambie con el primer elemento sin ordenar. Repita hasta que la matriz esté completamente ordenada.

#### **Ordenación de inserción:**

1. Complejidad del tiempo:  $O(n^2)$  en el peor y promedio de los casos,  $O(n)$  en el mejor de los casos (cuando la matriz de entrada ya está ordenada) Complejidad del espacio:  $O(1)$
2. Idea básica: Construya una submatriz ordenada de izquierda a derecha insertando cada nuevo elemento en su posición correcta en la submatriz. Repita hasta que la matriz esté completamente ordenada.

#### **Comparación:**

1. La ordenación de burbujas, la ordenación por selección y la ordenación por inserción tienen la misma complejidad de tiempo en el peor de los casos y en el caso medio de  $O(n^2)$ . Sin embargo, la ordenación por inserción suele tener un mejor rendimiento en la práctica, especialmente en datos casi ordenados, debido a que hay menos

intercambios y comparaciones, lo que la hace más eficiente en escenarios medios en comparación con la ordenación por burbujas y la ordenación por selección.

2. La ordenación por inserción tiene la complejidad temporal  $O(n)$  en el mejor de los casos cuando la matriz de entrada ya está ordenada, lo que no es posible para la ordenación por burbujas y la ordenación por selección.
3. La ordenación por selección y la ordenación por inserción tienen la misma complejidad espacial de  $O(1)$ , mientras que la ordenación por burbujas también tiene una complejidad espacial de  $O(1)$ .
4. La ordenación de burbujas y la ordenación por inserción son algoritmos de ordenación estables, lo que significa que conservan el orden relativo de los elementos iguales en la matriz ordenada, mientras que la ordenación por selección no es estable.
5. En términos de rendimiento, la ordenación por inserción tiende a funcionar mejor que la ordenación por burbujas y la ordenación por selección para conjuntos de datos pequeños, mientras que la ordenación por burbujas y la ordenación por selección pueden funcionar mejor que la ordenación por inserción para conjuntos de datos más grandes o conjuntos de datos que se ordenan parcialmente. En general, cada algoritmo tiene sus propias ventajas y desventajas, y la elección de qué algoritmo utilizar depende de los requisitos específicos del problema en cuestión.

### **Ventajas y desventajas de cada algoritmo**

#### **Clasificación de burbujas:**

Ventajas: Implementación simple, funciona bien para conjuntos de datos pequeños, requiere solo espacio constante, algoritmo de clasificación estable  
Desventajas: Ineficiente para conjuntos de datos grandes, complejidad temporal en el peor de los casos de  $O(n^2)$ , no óptimo para conjuntos de datos parcialmente ordenados

#### **Clasificación de selección:**

Ventajas: Implementación simple, funciona bien para conjuntos de datos pequeños, requiere solo espacio constante, algoritmo de clasificación en el lugar

Desventajas: Ineficiente para conjuntos de datos grandes, complejidad temporal en el peor de los casos de  $O(n^2)$ , no es óptimo para conjuntos de datos parcialmente ordenados, no es un algoritmo de clasificación estable

#### **Ordenación de inserción:**



Ventajas: Implementación simple, funciona bien para conjuntos de datos pequeños, requiere solo espacio constante, eficiente para conjuntos de datos parcialmente ordenados, algoritmo de clasificación estable Desventajas: Ineficiente para conjuntos de datos grandes, complejidad de tiempo en el peor de los casos de  $O(n^2)$ .

#### **4.4. Ordenación rápida frente a ordenación combinada**

La ordenación rápida primero particiona la matriz y, a continuación, realiza dos llamadas recursivas. La ordenación por combinación realiza primero llamadas recursivas para las dos mitades y, a continuación, combina las dos mitades ordenadas.

A continuación, se muestran las diferencias entre los dos algoritmos de ordenación.

- I. Partición de los elementos de la matriz: En el orden de fusión, la matriz se divide en solo 2 mitades (es decir,  $n/2$ ), mientras que, en el caso de la clasificación rápida, la matriz se divide en cualquier proporción. No hay compulsión de dividir la matriz de elementos en partes iguales en una clasificación rápida.
- II. Complejidad del peor de los casos: La complejidad del peor caso de la clasificación rápida es  $O(n^2)$ , ya que se necesitan muchas comparaciones en las peores condiciones, mientras que en el orden de fusión, el peor de los casos y el caso promedio tienen las mismas complejidades  $O(n \log n)$ .
- III. Requisito de espacio de almacenamiento adicional: la ordenación por combinación no está en su lugar porque requiere espacio de memoria adicional para almacenar las matrices auxiliares, mientras que la ordenación rápida está en su lugar, ya que no requiere ningún almacenamiento adicional.
- IV. Método de ordenación: La ordenación rápida es un método de ordenación interna en el que los datos se ordenan en la memoria principal, mientras que la ordenación por fusión es un método de ordenación externo en el que los datos que se van a ordenar no se pueden acomodar en la memoria y se necesita la memoria auxiliar para la ordenación.
- V. Estabilidad: La ordenación por fusión es estable ya que dos elementos con el mismo valor aparecen en el mismo orden en la salida ordenada que en la matriz sin ordenar de entrada, mientras que la ordenación rápida no es estable en este escenario. Pero se puede hacer estable usando algunos cambios en el código.
- VI. Preferido para: Se prefiere la ordenación rápida para las matrices, mientras que la ordenación por combinación es preferible para las listas enlazadas. La ordenación rápida funciona mejor en general, pero la ordenación combinada funciona mejor para la ordenación externa.
- VII. Localidad de referencia: Quicksort exhibe una buena localidad de caché y esto hace que quicksort sea más rápido que el orden combinado (en muchos casos como en el entorno de memoria virtual).

VIII.    Recursividad de la cola: La clasificación rápida es de naturaleza recursiva de la cola y, por lo tanto, se optimiza fácilmente mediante la eliminación de llamadas de cola. Mientras que Merge Sort no es recursivo de cola.

## 5. ANEXOS

### 5.1. Código En Java

```
import java.util.Random;
```

```
public class SortingAlgorithms {
```

```
    public static void bubbleSort(int[] arr) {
```

```
        int n = arr.length;
```

```
        for (int i = 0; i < n; i++) {
```

```
            for (int j = 0; j < n - i - 1; j++) {
```

```
                if (arr[j] > arr[j + 1]) {
```

```
                    int temp = arr[j];
```

```
                    arr[j] = arr[j + 1];
```

```
                    arr[j + 1] = temp;
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

```
    public static void selectionSort(int[] arr) {
```

```
        int n = arr.length;
```

```
        for (int i = 0; i < n; i++) {
```

```
            int minIndex = i;
```

```
            for (int j = i + 1; j < n; j++) {
```

```
                if (arr[j] < arr[minIndex]) {
```

```
                    minIndex = j;
```

```
                }
```

```
            }
```

```
            int temp = arr[i];
```

```

        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

```

```

public static void insertionSort(int[] arr) {
    int n = arr.length;
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

```

```

public static void main(String[] args) {
    int[] arr = new int[10000];
    Random rand = new Random();
    for (int i = 0; i < 10000; i++) {
        arr[i] = rand.nextInt(10000) + 1;
    }
}

```

```

long startTime = System.nanoTime();
bubbleSort(arr.clone());
long bubbleSortTime = System.nanoTime() - startTime;

```

```

        startTime = System.nanoTime();
        selectionSort(arr.clone());
        long selectionSortTime = System.nanoTime() - startTime;

        startTime = System.nanoTime();
        insertionSort(arr.clone());
        long insertionSortTime = System.nanoTime() - startTime;

        System.out.println("Bubble Sort time: " + bubbleSortTime);
        System.out.println("Selection Sort time: " + selectionSortTime);
        System.out.println("Insertion Sort time: " + insertionSortTime);
    }
}

```

## 5.2. Código En C++

```

#include <iostream>

#include <vector>

#include <ctime>

#include <cstdlib>

using namespace std;

// Function to perform Bubble Sort
void bubble_sort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                // Swapping elements
                int temp = arr[j];

```

```

        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
    }
}
}
}

```

```

// Function to perform Selection Sort
void selection_sort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n; ++i) {
        int min_index = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }
        // Swapping elements
        int temp = arr[i];
        arr[i] = arr[min_index];
        arr[min_index] = temp;
    }
}

```

```

// Function to perform Insertion Sort
void insertion_sort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; ++i) {
        int key = arr[i];

```

```

    int j = i - 1;
    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}
}

```

```

int main() {
    // Generate a vector of 10000 random integers
    vector<int> arr;
    srand(time(nullptr)); // Seed for random number generation
    for (int i = 0; i < 10000; ++i) {
        arr.push_back(rand() % 10000 + 1); // Generate random numbers between 1 and 10000
    }

    // Sort the vector using each algorithm and measure time
    clock_t start_time, end_time;

    start_time = clock();
    bubble_sort(arr);
    end_time = clock();
    double bubble_sort_time = double(end_time - start_time) / CLOCKS_PER_SEC;

    start_time = clock();
    selection_sort(arr);
    end_time = clock();
    double selection_sort_time = double(end_time - start_time) / CLOCKS_PER_SEC;
}

```

```

start_time = clock();
insertion_sort(arr);
end_time = clock();
double insertion_sort_time = double(end_time - start_time) / CLOCKS_PER_SEC;

// Print the time taken by each sorting algorithm
cout << "Bubble Sort time: " << bubble_sort_time << endl;
cout << "Selection Sort time: " << selection_sort_time << endl;
cout << "Insertion Sort time: " << insertion_sort_time << endl;

return 0;
}

```

### 5.3. Código En Piton

```

import random

def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:

```



```
        min_index = j
    arr[i], arr[min_index] = arr[min_index], arr[i]
```

```
def insertion_sort(arr):
```

```
    n = len(arr)
```

```
    for i in range(1, n):
```

```
        key = arr[i]
```

```
        j = i - 1
```

```
        while j >= 0 and arr[j] > key:
```

```
            arr[j + 1] = arr[j]
```

```
            j -= 1
```

```
        arr[j + 1] = key
```

```
# Generate a list of 10000 random integers
```

```
arr = [random.randint(1, 10000) for i in range(10000)]
```

```
# Sort the list using each algorithm and time it
```

```
import time
```

```
start_time = time.time()
```

```
bubble_sort(arr.copy())
```

```
bubble_sort_time = time.time() - start_time
```

```
start_time = time.time()
```

```
selection_sort(arr.copy())
```

```
selection_sort_time = time.time() - start_time
```

```
start_time = time.time()
```

```
insertion_sort(arr.copy())
```

```
insertion_sort_time = time.time() - start_time
```

```
print("Bubble Sort time:", bubble_sort_time)
```

```
print("Selection Sort time:", selection_sort_time)
```

```
print("Insertion Sort time:", insertion_sort_time)
```

#### **5.4. Código En C#**

```
using System;
```

```
using System.Diagnostics;
```

```
using System.Linq;
```

```
public class SortingAlgorithms {
```

```
    public static void BubbleSort(int[] arr) {
```

```
        int n = arr.Length;
```

```
        for (int i = 0; i < n; i++) {
```

```
            for (int j = 0; j < n - i - 1; j++) {
```

```
                if (arr[j] > arr[j + 1]) {
```

```
                    int temp = arr[j];
```

```
                    arr[j] = arr[j + 1];
```

```
                    arr[j + 1] = temp;
```

```
                }
```

```
            }
```

```
        }
```

```
    }
```

```
    public static void SelectionSort(int[] arr) {
```

```
        int n = arr.Length;
```

```
        for (int i = 0; i < n; i++) {
```

```
            int minIndex = i;
```

```
            for (int j = i + 1; j < n; j++) {
```

```

        if (arr[j] < arr[minIndex]) {
            minIndex = j;
        }
    }

    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
}
}

```

```

public static void InsertionSort(int[] arr) {
    int n = arr.Length;
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

```

```

public static void Main(string[] args) {
    int[] arr = new int[10000];
    Random rand = new Random();
    for (int i = 0; i < 10000; i++) {
        arr[i] = rand.Next(10000) + 1;
    }
}

```

```

Stopwatch sw = new Stopwatch();

sw.Start();
BubbleSort((int[])arr.Clone());
sw.Stop();
long bubbleSortTime = sw.ElapsedTicks;

sw.Restart();
SelectionSort((int[])arr.Clone());
sw.Stop();
long selectionSortTime = sw.ElapsedTicks;

sw.Restart();
InsertionSort((int[])arr.Clone());
sw.Stop();
long insertionSortTime = sw.ElapsedTicks;

Console.WriteLine("Bubble Sort time: " + bubbleSortTime);
Console.WriteLine("Selection Sort time: " + selectionSortTime);
Console.WriteLine("Insertion Sort time: " + insertionSortTime);
}
}

```

### 5.5. Código En JavaScript

```

// Function to generate a random integer between min (inclusive) and max (inclusive)
function getRandomInt(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

```

```
// Function to perform Bubble Sort
```

```
function bubbleSort(arr) {  
  const n = arr.length;  
  for (let i = 0; i < n; i++) {  
    for (let j = 0; j < n - i - 1; j++) {  
      if (arr[j] > arr[j + 1]) {  
        // Swap arr[j] and arr[j + 1]  
        const temp = arr[j];  
        arr[j] = arr[j + 1];  
        arr[j + 1] = temp;  
      }  
    }  
  }  
}
```

```
// Function to perform Selection Sort
```

```
function selectionSort(arr) {  
  const n = arr.length;  
  for (let i = 0; i < n; i++) {  
    let minIndex = i;  
    for (let j = i + 1; j < n; j++) {  
      if (arr[j] < arr[minIndex]) {  
        minIndex = j;  
      }  
    }  
    // Swap arr[i] and arr[minIndex]  
    const temp = arr[i];  
    arr[i] = arr[minIndex];  
    arr[minIndex] = temp;  
  }  
}
```

```
}  
}
```

```
// Function to perform Insertion Sort
```

```
function insertionSort(arr) {
```

```
    const n = arr.length;
```

```
    for (let i = 1; i < n; i++) {
```

```
        const key = arr[i];
```

```
        let j = i - 1;
```

```
        while (j >= 0 && arr[j] > key) {
```

```
            arr[j + 1] = arr[j];
```

```
            j--;
```

```
        }
```

```
        arr[j + 1] = key;
```

```
    }
```

```
}
```

```
// Main function
```

```
function main() {
```

```
    const arr = new Array(10000);
```

```
// Populate the array with random values
```

```
for (let i = 0; i < 10000; i++) {
```

```
    arr[i] = getRandomInt(1, 10000); // Adjust the range as needed
```

```
}
```

```
const startTimeBubble = performance.now();
```

```
bubbleSort(arr);
```

```
const bubbleSortTime = performance.now() - startTimeBubble;
```

```
const startTimeSelection = performance.now();
selectionSort([...arr]);
const selectionSortTime = performance.now() - startTimeSelection;

const startTimeInsertion = performance.now();
insertionSort([...arr]);
const insertionSortTime = performance.now() - startTimeInsertion;

console.log("Bubble Sort time: " + bubbleSortTime.toFixed(2) + " ms");
console.log("Selection Sort time: " + selectionSortTime.toFixed(2) + " ms");
console.log("Insertion Sort time: " + insertionSortTime.toFixed(2) + " ms");
}

// Call the main function
main();
```

### 5.6. Resultados

```
Burbuja Tiempo de ordenación: 16.710935354232788
Selección Tiempo de ordenación: 7.3090105056762695
Inserción Tiempo de ordenación: 0.003000974655151367
```

## 5.7. Código de Búsqueda binaria


```
1 def busqueda_binaria_iterativa(lista, objetivo):
2     inicio = 0
3     fin = len(lista) - 1
4
5     while inicio <= fin:
6         medio = (inicio + fin) // 2
7         if lista[medio] == objetivo:
8             return medio
9         elif lista[medio] < objetivo:
10            inicio = medio + 1
11        else:
12            fin = medio - 1
13
14    return -1
```

```
1 def busqueda_binaria_recursiva(lista, objetivo, inicio, fin):
2     if inicio > fin:
3         return -1 # Caso base
4
5     medio = (inicio + fin) // 2
6     if lista[medio] == objetivo:
7         return medio
8     elif lista[medio] < objetivo:
9         return busqueda_binaria_recursiva(lista, objetivo, medio + 1, fin)
10    else:
11        return busqueda_binaria_recursiva(lista, objetivo, inicio, medio - 1)
12
```

```
1 lista_ordenada = [1, 3, 5, 7, 9, 11, 13, 15]
2 objetivo = 7
3
4 # Usando la implementación iterativa
5 indice_iterativo = busqueda_binaria_iterativa(lista_ordenada, objetivo)
6 print(f"Resultado iterativo: {indice_iterativo}")
7
8 # Usando la implementación recursiva
9 indice_recursivo = busqueda_binaria_recursiva(lista_ordenada, objetivo, 0, len(lista_ordenada) - 1)
10 print(f"Resultado recursivo: {indice_recursivo}")
```




## 5.8. Código de Selection Sort



```
1  def selection_sort(lista):
2      n = len(lista)
3      for i in range(n):
4          min_index = i
5          for j in range(i + 1, n):
6              if lista[j] < lista[min_index]:
7                  min_index = j
8          lista[i], lista[min_index] = lista[min_index], lista[i]
9
10 # Datos de ejemplo
11 lista = [64, 25, 12, 22, 11]
12
13 # Mostramos el arreglo antes de ordenar
14 print("Arreglo original:", lista)
15
16 # Ordenamos el arreglo usando Selection Sort
17 selection_sort(lista)
18
19 # Mostramos el arreglo ordenado
20 print("Arreglo ordenado:", lista)
21
```

## 5.9. Código de Insertion Sort



```
1  def insertion_sort(lista):
2      for i in range(1, len(lista)):
3          actual = lista[i]
4          j = i - 1
5          while j >= 0 and lista[j] > actual:
6              lista[j + 1] = lista[j]
7              j -= 1
8          lista[j + 1] = actual
9
10     # Datos de ejemplo
11     lista = [12, 11, 13, 5, 6]
12
13     # Mostramos el arreglo antes de ordenar
14     print("Arreglo original:", lista)
15
16     # Ordenamos el arreglo usando Insertion Sort
17     insertion_sort(lista)
18
19     # Mostramos el arreglo ordenado
20     print("Arreglo ordenado:", lista)
21
```

## **5.10. Código de Merge Sort**



```
1  def merge_sort(lista):
2      if len(lista) <= 1:
3          return lista
4
5      medio = len(lista) // 2
6      izquierda = lista[:medio]
7      derecha = lista[medio:]
8
9      izquierda = merge_sort(izquierda)
10     derecha = merge_sort(derecha)
11
12     return merge(izquierda, derecha)
13
14 def merge(izquierda, derecha):
15     resultado = []
16     i = j = 0
17
18     while i < len(izquierda) and j < len(derecha):
19         if izquierda[i] < derecha[j]:
20             resultado.append(izquierda[i])
21             i += 1
22         else:
23             resultado.append(derecha[j])
24             j += 1
25
26     resultado.extend(izquierda[i:])
27     resultado.extend(derecha[j:])
28
29     return resultado
30
31 # Datos de ejemplo
32 lista = [38, 27, 43, 3, 9, 82, 10]
33
34 # Mostramos el arreglo antes de ordenar
35 print("Arreglo original:", lista)
36
37 # Ordenamos el arreglo usando Merge Sort
38 lista_ordenada = merge_sort(lista)
39
40 # Mostramos el arreglo ordenado
41 print("Arreglo ordenado:", lista_ordenada)
42
```

## 5.11. Código de Quick Sort

```
1 def quick_sort(lista):
2     if len(lista) <= 1:
3         return lista
4
5     pivote = lista[-1]
6     izquierda = []
7     derecha = []
8     iguales = []
9
10    for elemento in lista:
11        if elemento < pivote:
12            izquierda.append(elemento)
13        elif elemento == pivote:
14            iguales.append(elemento)
15        else:
16            derecha.append(elemento)
17
18    return quick_sort(izquierda) + iguales + quick_sort(derecha)
19
20 # Datos de ejemplo
21 lista = [10, 7, 8, 9, 1, 5]
22
23 # Mostramos el arreglo antes de ordenar
24 print("Arreglo original:", lista)
25
26 # Ordenamos el arreglo usando Quick Sort
27 lista_ordenada = quick_sort(lista)
28
29 # Mostramos el arreglo ordenado
30 print("Arreglo ordenado:", lista_ordenada)
31
```

## 5.12. Complejidad

### Complejidad temporal y espacial:

Algoritmo de clasificación	Complejidad del tiempo			Complejidad del espacio
	El mejor de los casos	Caso promedio	El peor de los casos	
Clasificación de burbujas	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$
Ordenación por selección	$O(N^2)$	$O(N^2)$	$O(N^2)$	$O(1)$
Ordenación de inserción	$O(N)$	$O(N^2)$	$O(N^2)$	$O(1)$

Algoritmo	Mejor caso	Caso promedio	Peor caso	Complejidad espacial
Búsqueda Binaria	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$ iterativa, $O(\log n)$ recursiva
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

## 6. BIBLIOGRAFIA

- GeeksforGeeks. (2024, 11 octubre). *Comparison among Bubble Sort, Selection Sort and Insertion Sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/comparison-among-bubble-sort-selection-sort-and-insertion-sort/>
- M, G. P. (2023, 21 abril). *Explicación de la notación Big O con Ejemplos*. freeCodeCamp.org. <https://www.freecodecamp.org/espanol/news/explicacion-de-la-notacion-big-o-con-ejemplo/>
- Latam, A. (2022, 10 junio). *Algoritmo MergeSort: cómo implementarlo en Python*. Alura. [https://www.aluracursos.com/blog/algoritmo-mergesort-como-implementarlo-en-python?utm\\_source=gnarus&utm\\_medium=timeline](https://www.aluracursos.com/blog/algoritmo-mergesort-como-implementarlo-en-python?utm_source=gnarus&utm_medium=timeline)
- Latam, A. (2022b, junio 13). *Algoritmo Quicksort: cómo implementar en Python*. Alura. [https://www.aluracursos.com/blog/algoritmo-quicksort-como-implementar-en-python?utm\\_source=gnarus&utm\\_medium=timeline](https://www.aluracursos.com/blog/algoritmo-quicksort-como-implementar-en-python?utm_source=gnarus&utm_medium=timeline)
- GeeksforGeeks. (2024b, octubre 19). *Quick Sort vs Merge Sort*. GeeksforGeeks. <https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/>