Análisis comparativo del rendimiento de los algoritmos Quick Sort y Heap Sort

Comparative performance analysis of Quick Sort and Heap Sort algorithms

Dennis J.L.C. Cristopher D.E.C.M. Faridd R.

Resumen (ES): Este estudio presenta un análisis empírico del rendimiento de los algoritmos Quick Sort y Heap Sort, evaluando su comportamiento en distintos escenarios de entrada: aleatorios, ordenados, parcialmente ordenados y con duplicados. Las implementaciones se realizaron en C++ sobre un entorno controlado, registrando métricas como tiempo de ejecución, número de comparaciones e intercambios. Los resultados muestran que Quick Sort supera en velocidad promedio a Heap Sort en datos aleatorios, pero su rendimiento se degrada significativamente en arreglos ordenados debido a la profundidad de recursión. Heap Sort, por su parte, mantiene una eficiencia estable en todos los casos, destacando por su predictibilidad y resistencia a escenarios adversos. Se concluye que Quick Sort es ideal para grandes volúmenes de datos aleatorios, mientras que Heap Sort es preferible en contextos donde se requiere estabilidad y consistencia.

Abstract (EN): This study presents an empirical performance analysis of Quick Sort and Heap Sort algorithms, evaluating their behavior across various input scenarios: random, sorted, partially sorted, and with duplicates. Implementations were developed in C++ under a controlled environment, measuring execution time, number of comparisons, and swaps. Results show that Quick Sort outperforms Heap Sort in average speed for random data, but its performance degrades significantly with sorted arrays due to deep recursion. Heap Sort, on the other hand, maintains stable efficiency across all cases, standing out for its predictability and resilience in adverse scenarios. It is concluded that Quick Sort is optimal for large random datasets, while Heap Sort is preferable in contexts requiring stability and consistency

Palabras clave: Ordenamiento, Quick Sort, Heap Sort

Keywords: Sorting, Quick Sort, Heap Sort

Introducción:

El ordenamiento de datos es una operación fundamental en ciencias de la computación, con distintas aplicaciones desde bases de datos, procesamiento de información hasta análisis de datos.
Entre ellos, Quick Sort y Heap Sort, dos algoritmos de ordenamiento eficientes que difieren significativamente en su enfoque y rendimiento práctico. Quick Sort, basado en el paradigma “divide y vencerás” usando recursividad, es ampliamente utilizado por su velocidad promedio, aunque presenta vulnerabilidades en casos adversos.
Por su parte, Heap Sort utiliza estructuras de heap binario, un árbol binario, para garantizar un rendimiento consistente, destacándose en escenarios con restricciones de memoria.
Trabajos previos han documentado las complejidades teóricas de ambos algoritmos (Cormen y otros autores, 2009), pero aun existen las brechas en estudios comparativos empíricos que consideran implementaciones modernas y patrones de datos reales.
Este artículo tiene como objetivo principal comparar el rendimiento de Quick Sort y Heap Sort en términos de tiempo de ejecución, uso de memoria y estabilidad, evaluando su comportamiento en conjuntos de datos aleatorios, parcialmente ordenados, ordenados de forma ascendente y descendente, y con elementos repetidos. La hipótesis descriptiva postula que Quick Sort superará a Heap Sort en velocidad promedio, mientras que Heap Sort exhibirá mayor predictibilidad y eficiencia, especialmente en entornos con limitaciones de recursos.
La comparación se centra en implementaciones optimizadas con pivote indexada al primer elemento para Quick Sort, analizando métricas cuantitativas obtenidas mediante experimentación controlada, sin profundizar en aspectos teóricos ya establecidos en la literatura.

Método:

- **Algoritmo Quick Sort**: Se implementó Quick Sort como un algoritmo de ordenamiento basado en el paradigma divide y “vencerás”, usando la recursividad y a su vez, una estrategia de partición in-place optimizada. La variante empleada en esta implementación, definida en la función “Reduce”, realiza una partición bidireccional dinámica que ajusta simultáneamente los punteros izquierdo (izq) y derecho (der) alrededor de un pivote implícito, que al inicio, siempre se tomará que primer elemento en cada vector aun no ordenado, partiendo y seleccionando elementos de manera iterativa para reordenarlos. A diferencia de las versiones clásicas que eligen un pivote fijo como el último elemento, esta implementación no utiliza un pivote explícito predefinido, sino que ajusta dinámicamente la posición (pos) mediante comparaciones y swaps hasta estabilizar la partición. Este enfoque reduce la

recursión excesiva al priorizar la partición de subarreglos más pequeños primero, utilizando una lógica de balanceo basada en las longitudes de los subarreglos izquierdo (leftLen) y derecho (rightLen). La función quickSort sirve como interfaz principal, invocando Reduce con el rango completo del vector. Se incluye un conteo de comparaciones_qs y intercambios_qs como variables externas para monitorear el rendimiento, lo que permite evaluar la eficiencia y estabilidad del algoritmo. Esta implementación se desarrolló en C++ desde cero, evitando dependencias de la STL(Standard Template Library) más allá de <vector> y <algorithm> para usar funciones como swap(para el intercambio), asegurando un control detallado sobre el proceso y facilitando su adaptación a análisis empíricos específicos.

Complejidad:

- **Caso Mejor**: O(n log n)
  - Ocurre cuando la partición divide el arreglo en partes casi iguales en cada paso. La función Reduce realiza O(n) comparaciones e intercambios por nivel, y la profundidad de la recursión es O(log n) debido al balanceo dinámico de subarreglos (leftLen y rightLen). El costo total es O(n log n).
- **Caso Promedio**: O(n log n)
  - Asumiendo una distribución aleatoria de los elementos, la partición tiende a dividir el arreglo de manera razonablemente equilibrada. El bucle interno de Reduce sigue siendo O(n) por nivel, y la profundidad promedio de recursión sigue siendo O(log n) gracias al manejo de subarreglos más pequeños primero. Esto resulta en coste total de O(n log n).
- **Caso Peor**: $O(n^2)$
  - Ocurre cuando la partición genera subarreglos altamente desbalanceados como 1 y n-1 elementos, también en arreglos ya ordenados o con todos los elementos iguales. La profundidad de la recursión alcanza O(n), y cada nivel requiere O(n) operaciones, llevando a $O(n^2)$ comparaciones e intercambios.

- **Algoritmo Heap Sort:** Heap Sort es un algoritmo de ordenamiento basado en una estructura de datos llamada montículo (heap), que es un árbol binario casi completo donde cada nodo cumple una propiedad:

  - En un heap máximo, cada nodo es mayor o igual que sus hijos.
  - En un heap mínimo, cada nodo es menor o igual que sus hijos.

  Donde el algoritmo busca construir un heap máximo (Heapify)a partir de los elementos del arreglo. Así, el elemento más grande estará en la raíz (posición

0). Intercambia el primer elemento (el máximo) con el último del arreglo, de esta manera el elemento mayor queda al final, luego se reduce el tamaño del heap (ignorando el último elemento ya ordenado). Restaurar la propiedad del heap aplicando la función "**heapify"** sobre la raíz iterativamente hasta que se cree el heap máximo en nuestro caso.

- **Heapify:** se encarga de mantener la estructura de heap máximo en una porción del arreglo (vector)**.** Primero se compara el valor del nodo padre(empezando desde el primer elemento) con sus hijos. Si alguno de los hijos es mayor, se intercambia con el padre. Luego se aplica recursivamente al subárbol afectado, de esta forma se crea el heap máximo que posteriormente se ordenará en la función principal "Heap Sort".

  Complejidad:

  - **Caso Mejor: O(n log n)**
    - Ocurre cuando el arreglo ya se encuentra parcialmente ordenado o en una disposición que facilita la construcción del heap. Durante la fase de construcción, la función "heapify" realiza un número limitado de comparaciones, ya que la mayoría de los nodos cumplen con la propiedad del heap desde el inicio. Sin embargo, debido a la naturaleza del algoritmo, Heap Sort siempre debe recorrer todos los niveles del heap, aplicando la función "heapify" y realizando comparaciones en cada extracción del máximo aunque en muchas ocasiones termina siendo innecesario si el arreglo ya estaba parcialmente ordenado. Por ello, aunque haya menos intercambios, el costo total sigue siendo proporcional a O(n log n), ya que:
      - Se realizan O(n) operaciones en la construcción inicial del heap.
      - Se efectúan O(log n) operaciones por cada extracción.
      - La combinación de ambos procesos mantiene el tiempo total en O(n log n).
  - **Caso Promedio:** O(n log n)
    - Asumiendo que los elementos del arreglo se encuentran distribuidos de manera aleatoria, la función "`heapify`" tenderá a recorrer una parte significativa de los niveles del heap en cada iteración.
      En promedio, cada elemento será comparado varias veces a lo largo de las fases de reordenamiento, pero el número de operaciones por nivel se mantiene dentro del rango logarítmico. De esta forma:
      - En cada paso, `heapify` requiere O(log n) operaciones.

- Como se repite para n elementos, el trabajo total asciende a O(n log n).

Por tanto, el rendimiento promedio de Heap Sort es constante y estable, sin grandes variaciones frente a la distribución de los datos. Este equilibrio hace que el algoritmo sea especialmente útil para contextos donde se requiere un desempeño predecible.

- **Caso Peor:** O(n log n)
  - El peor caso se presenta cuando el arreglo está completamente desordenado o en orden inverso, obligando a `heapify` a recorrer la máxima altura posible del heap en casi cada iteración.
    Aun así, debido a la estructura jerárquica del heap, la profundidad nunca excede log n, y cada elemento solo puede desplazarse hacia abajo en ese número de niveles.
    En consecuencia:
    - Cada llamada completa de `heapify` tiene un costo de O(log n).
    - Como este proceso se repite para n elementos, el costo total asciende a O(n log n).
      A diferencia de otros algoritmos de ordenamiento (como QuickSort), Heap Sort mantiene su eficiencia incluso en los casos más desfavorables, garantizando un comportamiento estable sin degradarse a $O(n^2)$.

- **Entorno: Entorno de ejecución**
  - Hardware:
    Equipo portátil LAPTOP-INSU7CJT con procesador AMD Ryzen 3 5300U with Radeon Graphics (4 núcleos, 8 hilos, frecuencia base de 2.60 GHz), 8 GB de memoria RAM (7.33 GB utilizable), arquitectura x64,y almacenamiento SSD. Pantalla sin soporte para entrada táctil o manuscrita.
  - Sistema operativo:
    Windows 11 Home de 64 bits, versión 23H2 (Id. del producto: 00342-43253-44520-AAOEM).
  - Compilador y entorno de desarrollo:
    MinGW-w64 (GCC 13.2.0) utilizado desde el entorno Code::Blocks / Visual Studio Code, con configuración estándar de compilación para C++.
  - Flags de compilación:

- **Generación de datos:**

- Los datos se generaron mediante el generador de números aleatorios Mersenne Twister (std::mt19937) con una semilla derivada del reloj del sistema (chrono::steady_clock).

  Considerando los siguientes patrones de entrada:

  - "Aleatorio uniforme (valores entre 0 y 10 000)."
  - "Ascendente (ordenados de menor a mayor)."
  - "Descendente (de mayor a menor)."
  - "Casi ordenado (95 % de elementos ordenados)."
  - "Con duplicados (10-20 valores repetidos).

  Cada vector se generó nuevamente para cada repetición usando una semilla distinta, registrada en los resultados para reproducibilidad."

- **Tamaños y patrones**
  - Los tamaños de entrada evaluados fueron: n = 1 000, 10 000 y 100 000 elementos, ajustados según la capacidad de tiempo y memoria del equipo.
  - Para cada tamaño se probaron los cinco patrones de entrada mencionados (aleatorio, ascendente, descendente, casi ordenado y con duplicados).

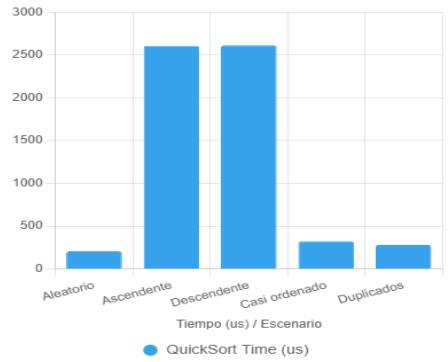
- **Métricas registradas:**
  - Para cada ejecución se midieron las siguientes métricas:
    - Tiempo de ejecución (ms, us, ns): obtenido con chrono::duration<double, milli>.
    - Número de comparaciones: contador incrementado cada vez que se compararon elementos del arreglo.
    - Número de intercambios: contador incrementado en cada intercambio lógico.

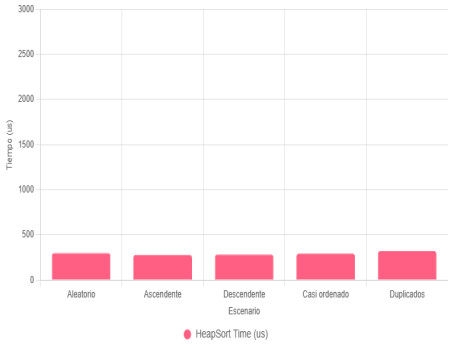
- **Repeticiones por condición:**

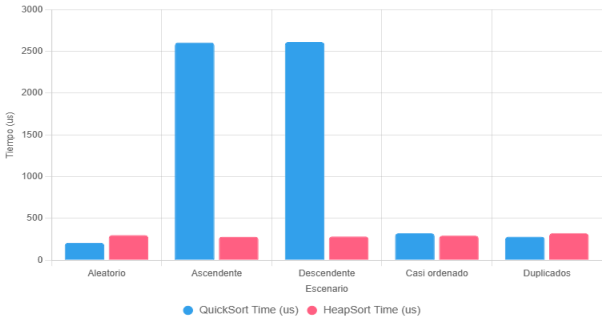
  Se realizó 5 intentos(repeticiones) para 1000, 10000 y 100000 elementos, donde luego con una gráfica, comparar el tiempo promedio que tardan para los distintos casos, arreglo aleatorio, ascendente, descendente, casi ordenado y con elementos repetidos (duplicados)

Resultados

Para 1000 elementos:

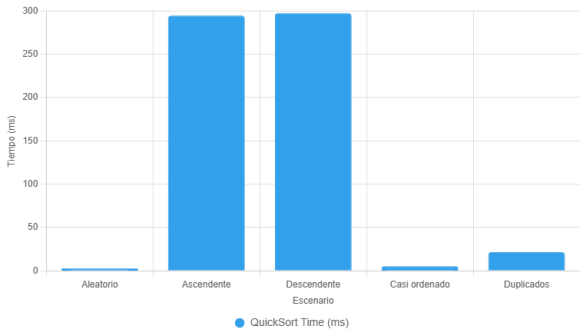


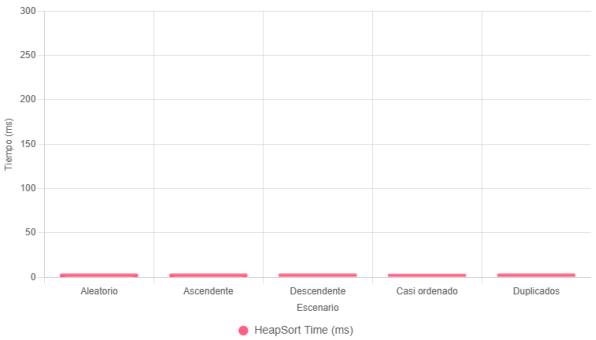


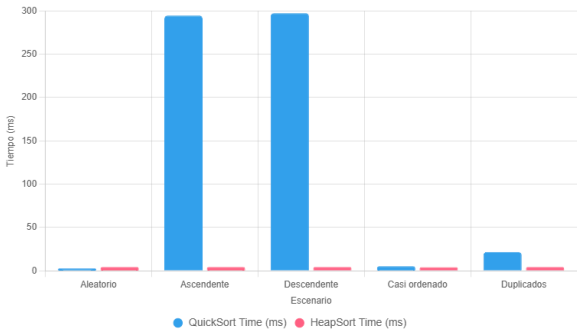


Escenario	Algoritmo	Promedio (us)	Mínimo (us)	Máximo (us)
Aleatorio	QuickSort	206.6	187	241
Ascendente	QuickSort	2770.6	2556	3409
Descendente	QuickSort	2615.2	2575	2717
Casi ordenado	QuickSort	321.2	296	355
Duplicados	QuickSort	281.6	232	376
Aleatorio	HeapSort	301.8	299	311
Ascendente	HeapSort	280.6	276	286
Descendente	HeapSort	283.2	256	335
Casi ordenado	HeapSort	313.6	291	393
Duplicados	HeapSort	321.6	278	370

Para 10000 elementos

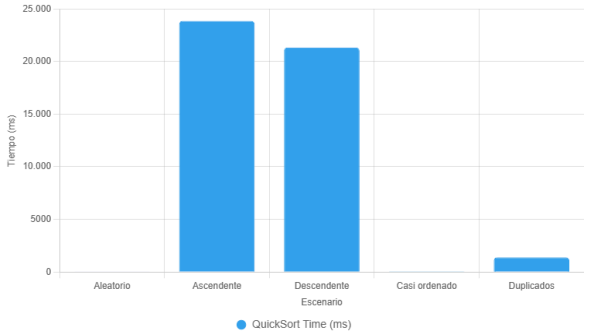


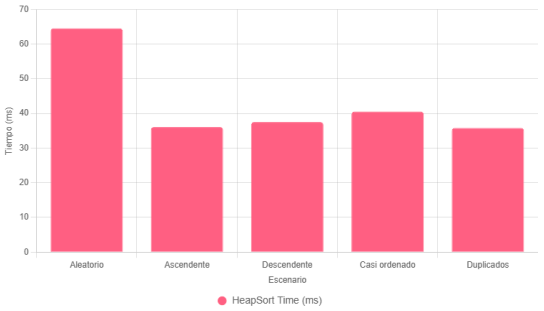


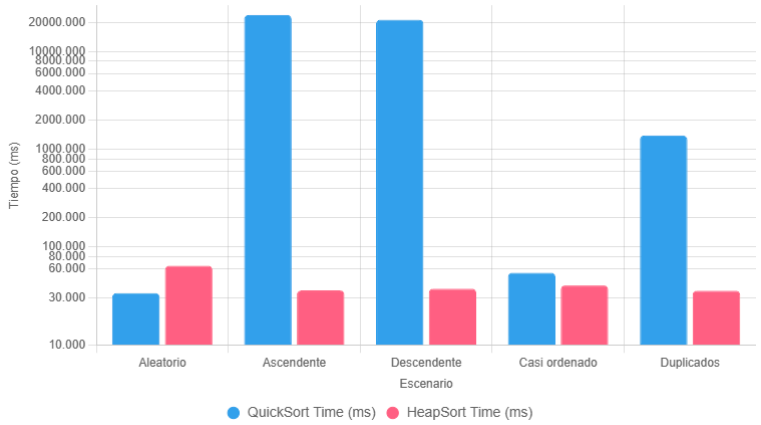


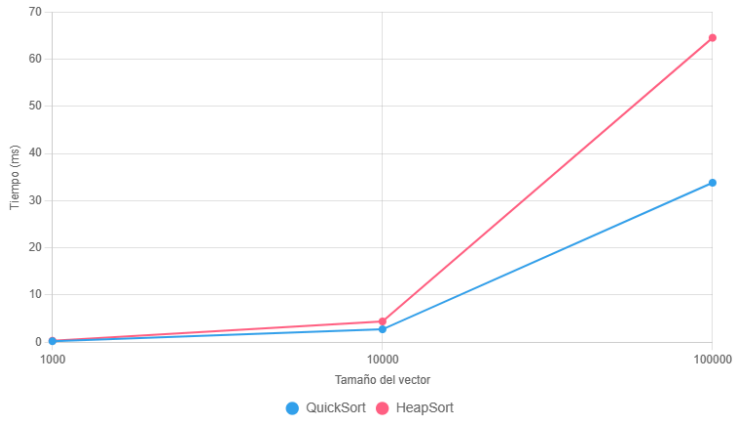
Escenario	Algoritmo	Promedio (us)	Mínimo (us)	Máximo (us)
Aleatorio	QuickSort	2724.8	2558	3015
Ascendente	QuickSort	294617.2	283232	303689
Descendente	QuickSort	297341.2	293122	314805
Casi ordenado	QuickSort	5303.0	4122	7017
Duplicados	QuickSort	21608.2	17439	29490
Aleatorio	HeapSort	4385.4	4027	4988
Ascendente	HeapSort	4349.4	3873	5248
Descendente	HeapSort	4428.2	3635	5711
Casi ordenado	HeapSort	4208.8	3937	4520
Duplicados	HeapSort	4444.2	3898	4825

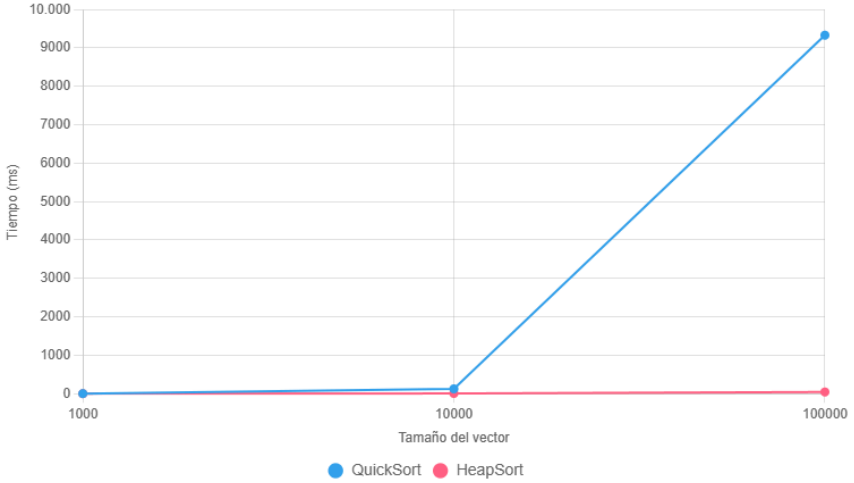
Para 100000 elementos











Escenario	Algoritmo	Promedio (us)	Mínimo (us)	Máximo (us)
Aleatorio	QuickSort	33796	32689	34609
Ascendente	QuickSort	23858508	23098708	24517966
Descendente	QuickSort	21329159	20104607	22189684
Casi ordenado	QuickSort	54728	48790	68180
Duplicados	QuickSort	1389137	1204025	1645712
Aleatorio	HeapSort	64529	61218	67172
Ascendente	HeapSort	36061	32958	39044
Descendente	HeapSort	37508	31733	50202
Casi ordenado	HeapSort	40525	37917	48223
Duplicados	HeapSort	35796	30622	39624

Discusiones

Los resultados obtenidos confirman la hipótesis inicial: Quick Sort ofrece un rendimiento superior en promedio para datos aleatorios, gracias a su estrategia de partición dinámica y optimización de subarreglos pequeños. Sin embargo, su vulnerabilidad ante entradas ordenadas o con elementos repetidos lo convierte en una opción menos robusta en escenarios adversos, donde la recursión profunda impacta negativamente en el tiempo de ejecución.

Heap Sort, aunque menos veloz en promedio, demuestra una notable estabilidad en todos los patrones de entrada. Su estructura jerárquica basada en montículos permite mantener la eficiencia incluso en el peor de los casos, lo que lo convierte en una herramienta confiable para aplicaciones donde la predictibilidad es crucial.

Este contraste entre velocidad y estabilidad plantea una elección estratégica: Quick Sort para entornos donde el rendimiento promedio es prioritario y los datos son aleatorios; Heap Sort para sistemas críticos donde se requiere consistencia y resistencia a variaciones en los datos.

Conlusiones

- Quick Sort es más rápido en promedio, especialmente con datos aleatorios, pero su rendimiento se degrada en casos ordenados o con duplicados.
- Heap Sort mantiene una eficiencia constante en todos los escenarios, siendo más estable y predecible.
- La elección entre ambos algoritmos debe considerar el tipo de datos y las restricciones del entorno: velocidad vs. estabilidad.
- Este análisis empírico refuerza la importancia de adaptar el algoritmo de ordenamiento al contexto específico de aplicación, más allá de su complejidad teórica.

Agradecimientos:

Agradezco a nuestro docente por su orientación y acompañamiento durante el desarrollo de este proyecto, que me permitió comprender de manera práctica el funcionamiento y análisis de los algoritmos de ordenamiento QuickSort y HeapSort. Asimismo, extiendo mi agradecimiento a mis compañeros de clase por su apoyo y colaboración en las pruebas y comparación de resultados, lo que contribuyó a enriquecer este trabajo.
Finalmente, valoró la oportunidad de aplicar los conocimientos adquiridos en análisis de complejidad y estructuras de datos, fortaleciendo mi comprensión de la eficiencia algorítmica en distintos escenarios.

Referencia Bibliográfica:

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

2. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley.

3. Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C++ (4th ed.). Pearson.

4. Knuth, D. E. (1998). The Art of Computer Programming, Volume 3: Sorting and Searching (2nd ed.). Addison-Wesley.

5. Williams, J. W. J. (1964). Algorithm 232: Heapsort. Communications of the ACM, 7(6), 347–348. https://doi.org/10.1145/364520.364540