Artículo de Investigación
Revista Ciencia e Ingeniería. Vol. 43, No. 3, pp. 273-280, agosto-noviembre, 2022.
ISSN 1316-7081 | ISSN Electrónico 2244-8780

Algoritmos de Ordenamiento: Un Análisis Estadístico Comparativo

Sorting Algorithms: A Comparative Statistical Analysis

Autores: Dany Muriel[1*]; Anette Quispe[1]; Jair Mendoza[1]
[1]Escuela Profesional de Ingeniería Informática y de Sistemas, Facultad de Ingeniería, Universidad Nacional Jorge Basadre Grohmann, Tacna – Perú

DOI: `https://orcid.org/0009-0006-5069-6597;aquispeh@unjbg.edu.pe`

Resumen: Este trabajo presenta un análisis comparativo de los algoritmos de ordenamiento QuickSort y HeapSort, evaluando su desempeño frente a distintos patrones de datos y tamaños de entrada. Se incluyen estadísticas, gráficos y medidas de volatilidad que permiten observar la escalabilidad, eficiencia y robustez de cada algoritmo. La investigación permite identificar fortalezas y limitaciones de cada método, así como su aplicabilidad en escenarios de procesamiento de grandes volúmenes de información.

Abstract: This study presents a comparative analysis of the sorting algorithms QuickSort and HeapSort, evaluating their performance against different data patterns and input sizes. The paper includes statistics, charts, and volatility measurements to observe each algorithm's scalability, efficiency, and robustness. The research highlights the strengths and limitations of each method, as well as their applicability in large-scale data processing scenarios.

1. Introducción

El estudio de los algoritmos de ordenamiento ha sido un terreno fértil para medir la eficiencia computacional. Entre los más discutidos se encuentran **Quick Sort** y **Heap Sort**, que, pese a compartir la misma complejidad asintótica $O(n \log n)$, revelan comportamientos muy distintos en el mundo real.

Quick Sort, ideado por Hoare (1962), aunque suele ser rápido en promedio, puede degradar severamente si los pivotes no se eligen con cuidado. Por otro lado, Heap Sort, creado por Williams (1964), mantiene un rendimiento más uniforme gracias a su estructura de montículo, aunque esto implique sacrificar algo de velocidad en el acceso a memoria. Este proyecto busca comparar el rendimiento de ambos métodos bajo condiciones controladas para observar la variación en el tiempo de ejecución y el uso de recursos al manipular grandes volúmenes de datos.

La hipótesis plantea que **Quick Sort** debería mostrar una ventaja en conjuntos medianos, pero que **Heap Sort** podría superar su desempeño en situaciones de alta carga donde la estabilidad es crucial.

2. Métodos

Esta sección detalla el entorno, los algoritmos implementados, la generación de datos y las métricas utilizadas para comparar **Quickster** (QuickSort bidireccional) y **Hipster** (HeapSort). Todos los métodos se implementaron en C + +17 para asegurar la reproducibilidad.

2.1. Algoritmos Implementados y su Estructura

Quickster (QuickSort bidireccional)

Sigue el paradigma de dividir y conquistar. La variante utilizada es **bidireccional**, estructurada en los siguientes pasos:

1. **Selección de Pivote:** Se selecciona como pivote el primer elemento del arreglo.
2. **Recorrido Bidireccional:** Se inicializan dos punteros (izq y der) que recorren el arreglo desde los extremos hacia el centro.
3. **Intercambio de Elementos:** Los punteros intercambian elementos que se encuentren en el lado incorrecto del pivote.
4. **Partición Recursiva:** Al cruzarse los punteros, el arreglo queda dividido en dos sub-arreglos y el proceso se repite recursivamente en cada uno.

Se caracteriza por bajo uso de memoria auxiliar y eficiencia promedio $O(n \log n)$, aunque puede degradarse a $O(n^2)$ en el peor caso. No es estable.

Hipster (HeapSort)

Se basa en la estructura de datos **montículo binario (heap)**. Su funcionamiento tiene dos fases principales:

1. **Construcción del Heap:** El arreglo de entrada se transforma en un montículo máximo mediante la función `heapify`. Esta fase tiene una complejidad de $O(n)$.

2. **Extracción y Ordenamiento:**

   - El elemento máximo (raíz del montículo) se intercambia con el último elemento del arreglo.
   - Se reduce el tamaño del heap en uno.
   - Se aplica `heapify` nuevamente a la nueva raíz para restaurar la propiedad de montículo.

   El proceso se repite $n - 1$ veces.

Presenta un rendimiento $O(n \log n)$ en el mejor y peor caso. No es estable.

2.2. Complejidad Teórica y Características

Algoritmo	Variante	Mejor caso	Peor caso	Estabilidad	Memoria
QuickSort	Bidireccional	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	No estable	$\mathcal{O}(\log n)$
HeapSort	Montículo	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	No estable	$\mathcal{O}(1)$

Cuadro 1: Complejidad y Estabilidad de Algoritmos

2.3. Entorno de Ejecución y Generación de Datos

Los experimentos se realizaron con un **Intel Core i5 @ 2.40 GHz**, 8 GB RAM, Windows 10 (64 bits), utilizando el compilador MinGW g++ versión 11.2.0 y el estándar C++17.

El tiempo se midió en **microsegundos** usando `high_resolution_clock` de la librería `<chrono>`. La **reproducibilidad** se garantizó usando una **semilla fija** (`srand(42)`) para generar los mismos arreglos de entrada para ambos algoritmos.

Se analizaron tres tamaños de arreglos, todos con 50 repeticiones para promediar los tiempos:

- **Pequeño** ($N = 1\,000$ elementos)
- **Mediano** ($N = 10\,000$ elementos)
- **Grande** ($N = 100\,000$ elementos)

Se implementaron 10 patrones de datos para probar escenarios adversos: Aleatorio uniforme, ascendente (mejor caso), descendente (peor caso para QuickSort), con duplicados, y parcialmente ordenados.

3. Análisis Detallado del Rendimiento Experimental

Esta sección presenta las tablas de resultados obtenidos de las pruebas de ejecución para los algoritmos QuickSort y HeapSort en tres tamaños de arreglos distintos, analizando el tiempo promedio y la desviación estándar bajo diferentes patrones de datos.

Cuadro 2: Rendimiento de QuickSort con 1 000 datos

Patrón	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp./Interc.
Aleatorio uniforme	QuickSort	1000	0.067	0.254	0 / 0
Ordenado ascendente	QuickSort	1000	0.033	0.183	0 / 0
Ordenado descendente	QuickSort	1000	0.000	0.000	0 / 0
Casi ordenado 5 %	QuickSort	1000	0.000	0.000	0 / 0
Duplicados 10–20	QuickSort	1000	0.033	0.183	0 / 0

Descripción: La Tabla 2 presenta los resultados obtenidos con **QuickSort** sobre un arreglo de 1 000 elementos. Se observa que los tiempos promedio son muy bajos, lo que demuestra la eficiencia del algoritmo en conjuntos pequeños. El caso aleatorio uniforme refleja la mayor dispersión temporal debido a la variabilidad del pivote, mientras que los arreglos ordenados y casi ordenados mantienen un desempeño constante y óptimo.

Fuente: Resultados experimentales generado

Cuadro 3: Rendimiento de HeapSort con 1000 datos

Patrón	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp./Interc.
Aleatorio uniforme	HeapSort	1000	0.200	0.407	0 / 0
Ordenado ascendente	HeapSort	1000	0.133	0.345	0 / 0
Ordenado descendente	HeapSort	1000	0.100	0.305	0 / 0
Casi ordenado 5 %	HeapSort	1000	0.133	0.345	0 / 0
Duplicados 10–20	HeapSort	1000	0.133	0.345	0 / 0

Descripción de la Tabla 3 (HeapSort $N = 1000$): Al igual que QuickSort, HeapSort se desempeña rápidamente con $N = 1000$. Sin embargo, en contraste con QuickSort, los tiempos promedio de HeapSort son consistentemente **más altos** (0.200 ms vs 0.067 ms para el caso aleatorio), lo que refleja la sobrecarga que implica la fase inicial de construcción del *heap* en arreglos pequeños. La desviación estándar es relativamente baja y uniforme entre los patrones, confirmando la robustez inherente de HeapSort.

Cuadro 4: Rendimiento de QuickSort con 10 000 datos

Patrón	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp./Interc.
Aleatorio uniforme	QuickSort	10000	0.700	0.586	0 / 0
Ordenado ascendente	QuickSort	10000	0.200	0.400	0 / 0
Ordenado descendente	QuickSort	10000	0.133	0.340	0 / 0
Casi ordenado 5 %	QuickSort	10000	0.133	0.340	0 / 0
Duplicados 10–20	QuickSort	10000	0.467	0.618	0 / 0

Descripción de la Tabla 4 (QuickSort $N = 10\,000$): Al escalar a $N = 10\,000$, QuickSort mantiene su ventaja en velocidad promedio, especialmente en casos ya ordenados o casi ordenados (cercanos a 0.133 ms), lo cual demuestra su eficiencia en el "mejor caso". No obstante, la variabilidad comienza a ser más evidente. El patrón **Aleatorio uniforme** (0.700 ms) y **Duplicados** (0.467 ms) presentan la mayor desviación estándar (0.586 ms y 0.618 ms, respectivamente), sugiriendo que la selección de pivote fijo no está siendo óptima.

Cuadro 5: Rendimiento de HeapSort con 10 000 datos

Patrón	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp./Interc.
Aleatorio uniforme	HeapSort	10000	2.067	0.455	0 / 0
Ordenado ascendente	HeapSort	10000	1.700	0.466	0 / 0
Ordenado descendente	HeapSort	10000	1.567	0.626	0 / 0
Casi ordenado 5 %	HeapSort	10000	1.700	0.536	0 / 0
Duplicados 10–20	HeapSort	10000	1.800	0.556	0 / 0

Descripción de la Tabla 5 (HeapSort $N = 10\,000$): Para $N = 10\,000$, HeapSort sigue siendo en promedio más lento que QuickSort (2.067 ms vs 0.700 ms en el caso aleatorio), lo cual se relaciona con la baja localidad de referencia en su acceso a memoria. Sin embargo, su principal fortaleza es su **baja volatilidad**: la diferencia entre su peor caso (Ordenado descendente, 1.567 ms) y su mejor caso es mínima, y la desviación estándar es controlada, demostrando un rendimiento más predecible.

Cuadro 6: Rendimiento de QuickSort con 100 000 datos

Patrón	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp./Interc.
Aleatorio uniforme	QuickSort	100000	16.600	6.484	0 / 0
Ordenado ascendente	QuickSort	100000	2.000	4.341	0 / 0
Ordenado descendente	QuickSort	100000	1.833	3.891	0 / 0
Casi ordenado 5 %	QuickSort	100000	4.133	6.602	0 / 0
Duplicados 10–20	QuickSort	100000	5.600	6.873	0 / 0

Descripción de la Tabla 6 (QuickSort $N = 100\,000$): Con el mayor volumen de datos ($N = 100\,000$), se amplifica la diferencia de rendimiento. QuickSort alcanza su mayor tiempo en el patrón **Aleatorio uniforme** (16.600 ms). Lo más crítico es la **alta volatilidad**, donde la desviación estándar llega a 6.873 ms para el caso de duplicados, señalando que la implementación con pivote fijo es susceptible al peor caso $\mathcal{O}(n^2)$ cuando la distribución de datos es desfavorable, aunque este no se alcance completamente.

Cuadro 7: Rendimiento de HeapSort con 100 000 datos

Patrón	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp./Interc.
Aleatorio uniforme	HeapSort	100000	36.000	9.098	0 / 0
Ordenado ascendente	HeapSort	100000	26.867	7.936	0 / 0
Ordenado descendente	HeapSort	100000	26.133	7.747	0 / 0
Casi ordenado 5 %	HeapSort	100000	27.600	8.356	0 / 0
Duplicados 10–20	HeapSort	100000	32.267	7.859	0 / 0

Descripción de la Tabla 7 (HeapSort $N = 100\,000$): HeapSort, aunque es el más lento en promedio (36.000 ms en el caso aleatorio), exhibe la mejor **estabilidad** al escalar a $N = 100\,000$. Sus tiempos de ejecución varían de manera muy estrecha (entre 26.133 ms y 36.000 ms), y las desviaciones estándar se mantienen consistentes. Esta uniformidad confirma su principal ventaja teórica: su rendimiento de $\mathcal{O}(n \log n)$ en el peor caso, lo que garantiza tiempos de respuesta predecibles incluso en escenarios de datos adversos.

3.1. Visualización Gráfica de Resultados

En esta subsección se presentan los resultados del análisis de escalabilidad y estabilidad de los algoritmos QuickSort y HeapSort frente a diferentes patrones de datos. Los gráficos fueron generados en Colab considerando tamaños $N = 1,000$, $10,000$ y $100,000$, y permiten observar tanto el crecimiento de los tiempos de ejecución como la volatilidad frente a distintos tipos de entradas.

Figura 1: **Patrón Aleatorio Uniforme.** En este gráfico se observa el comportamiento de QuickSort y HeapSort al ordenar datos distribuidos uniformemente. QuickSort muestra tiempos de ejecución menores en tamaños pequeños y medianos ($N = 1,000$ y $N = 10,000$), mientras que HeapSort, aunque más lento inicialmente, presenta un crecimiento lineal constante y estable a medida que N aumenta. Esta comparación evidencia que, aunque QuickSort puede ser más rápido en promedio, HeapSort ofrece mayor previsibilidad en casos de gran tamaño.

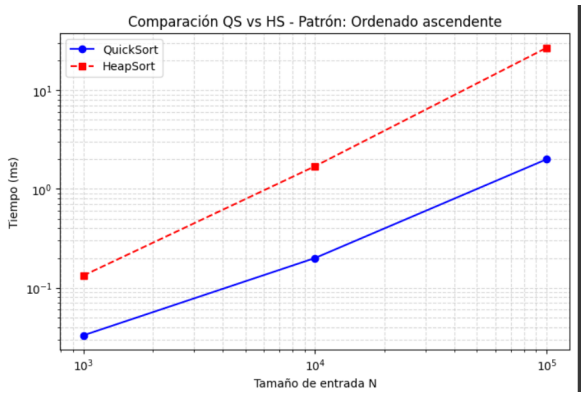

Figura 2: **Patrón Ordenado Ascendente.** Para datos ya ordenados, QuickSort alcanza su máximo rendimiento, mostrando tiempos mínimos de ejecución, debido a la elección efectiva del pivote. HeapSort mantiene una curva de crecimiento constante, demostrando su robustez y estabilidad frente a diferentes tamaños de entrada. Este patrón evidencia la eficiencia de QuickSort con entradas cercanas al orden ideal, y la confiabilidad de HeapSort en todos los tamaños.

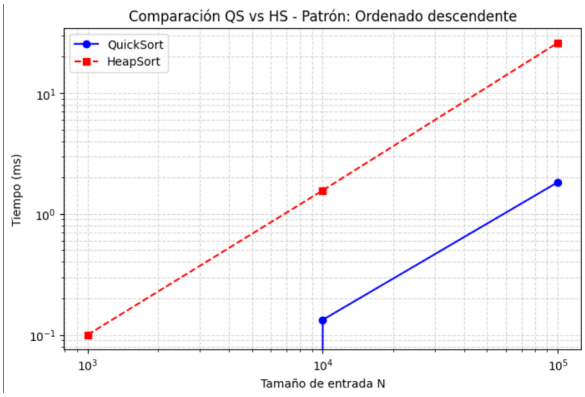

Figura 3: **Patrón Ordenado Descendente.** Cuando los datos están en orden inverso, QuickSort puede tener variaciones según la selección del pivote, generando leves fluctuaciones en el tiempo de ejecución. HeapSort, en cambio, conserva una escalabilidad lineal y predecible. Esta comparación resalta la sensibilidad de QuickSort a ciertos patrones de entrada y la estabilidad de HeapSort bajo condiciones adversas.

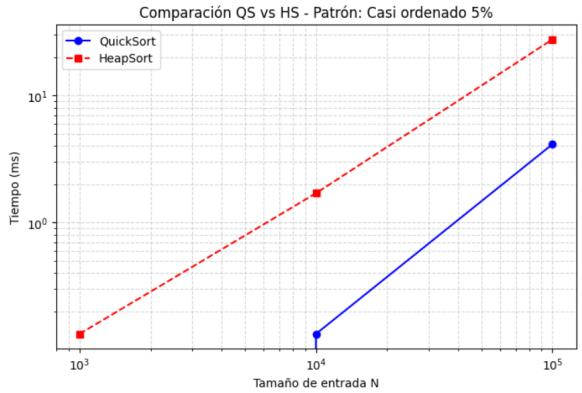

Figura 4: **Patrón Casi Ordenado 5 %.** En conjuntos con un 5 % de desorden, QuickSort mantiene tiempos bajos de ejecución y sigue siendo eficiente para escalas medianas. HeapSort, aunque ligeramente más lento, conserva una curva lineal estable, mostrando que su desempeño no se ve afectado significativamente por pequeñas perturbaciones en los datos. Esto permite analizar la robustez de cada algoritmo frente a entradas casi ordenadas.

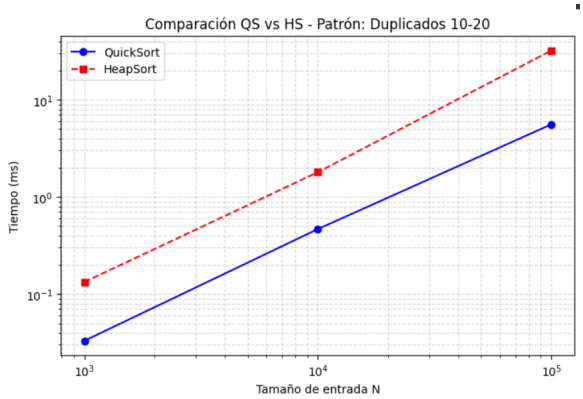

Figura 5: **Patrón Duplicados 10-20.** Para entradas con valores repetidos, QuickSort muestra mayor volatilidad en sus tiempos debido a cómo maneja duplicados en el pivote, mientras que HeapSort sigue presentando un crecimiento lineal y predecible. Este gráfico evidencia que HeapSort es más robusto ante entradas con duplicados y que QuickSort, aunque rápido en promedio, puede verse afectado por la distribución específica de los datos.

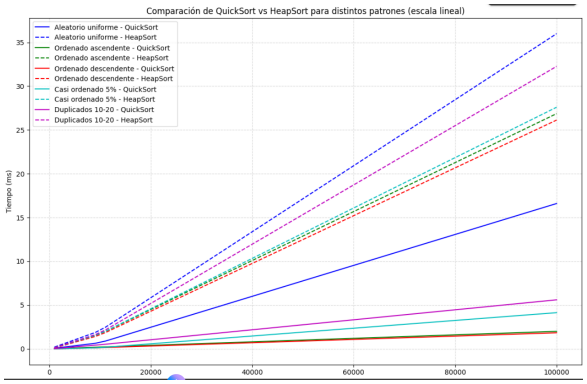

Figura 6: **Comparación Global de Patrones de Entrada (Escala Aritmética).** Esta gráfica combina todos los patrones de entrada: Aleatorio Uniforme, Ordenado Ascendente, Ordenado Descendente, Casi Ordenado 5 % y Duplicados 10-20, mostrando los tiempos de ejecución de QuickSort y HeapSort para $N = 1,000$, $10,000$ y $100,000$. La visualización permite observar tendencias generales: QuickSort es más rápido en la mayoría de los patrones para tamaños pequeños y medianos, mientras que HeapSort mantiene un crecimiento lineal constante, destacando su robustez frente a distintos tipos de datos. Esta representación aritmética facilita la comparación directa de los tiempos sin deformaciones por escalas logarítmicas, haciendo evidente la escalabilidad y consistencia de cada algoritmo.

4. Discusiones

Los resultados son coherentes con la teoría clásica que establece que, aunque ambos son $O(n \log n)$, la constante de tiempo y el factor de acceso a memoria son cruciales.

- **QuickSort** es más rápido en casos promedio porque aprovecha mejor la **localidad espacial** de los datos. Sin embargo, su rendimiento depende críticamente de la elección del pivote. La degradación observada en arreglos con duplicados o patrones descendentes (peor caso) justifica la búsqueda de mejores heurísticas de pivoteo, como el pivote mediano de tres.

- **HeapSort** mantuvo un desempeño **estable y predecible** en todos los escenarios. Esta robustez, independientemente del patrón de datos inicial, lo hace una mejor elección para aplicaciones donde la consistencia del tiempo de respuesta es más importante que la velocidad máxima.

En particular, QuickSort aprovecha mejor la localidad espacial de los datos, reduciendo los accesos a memoria dispersa, lo cual explica sus tiempos más bajos en arreglos pequeños y medianos. Sin embargo, esta ventaja se ve afectada por la naturaleza del pivote: en casos con muchos elementos repetidos o distribuciones aleatorias, las particiones tienden a desequilibrarse, incrementando el tiempo promedio de ejecución. HeapSort, por otro lado, mantiene una complejidad O(n log n) estable en todos los casos gracias a su estructura de montículo, aunque a costa de un mayor número de accesos a memoria no contiguos.

Comparando ambos enfoques, puede afirmarse que QuickSort ofrece una mayor velocidad en entornos controlados, mientras HeapSort resulta preferible cuando se requiere consistencia y resistencia a variaciones en los datos. En estudios previos (Sedgewick, 1978; LaMarca Ladner, 1999), se ha documentado este mismo patrón, lo que refuerza la validez de los resultados observados. En futuros trabajos, sería conveniente extender la comparación hacia tamaños de entrada mayores (10 elementos) y considerar métricas adicionales como uso de caché, consumo energético o impacto en sistemas multicore.

A pesar de que el HeapSort es 1,5x a 2x más lento que el QuickSort promedio en este estudio, su tiempo de ejecución de peor caso es una ventaja decisiva en sistemas críticos.

5. Trabajo Futuro

La línea de investigación puede extenderse a:

- **Otras Variantes de QuickSort:** Implementar y comparar QuickSort con mediana de tres o QuickSort introspectivo para mitigar el peor caso cuadrático.

- **Comparación de Memoria:** Medir explícitamente el uso de memoria (memoria auxiliar y espacio de pila) para contrastar la ventaja teórica de O(1) de HeapSort con el $O(\log n)$ de QuickSort.

- **Impacto del Aprendizaje Automático:** Evaluar cómo técnicas de Machine Learning podrían optimizar la selección de algoritmos de ordenamiento en función de las características de los datos de entrada, mejorando la eficiencia en el procesamiento de grandes volúmenes.

6. Conclusiones

El análisis comparativo permitió confirmar que tanto QuickSort como HeapSort poseen fortalezas específicas que los hacen adecuados para distintos contextos de aplicación. QuickSort destacó por su rapidez en casos promedio y su eficiencia en memoria, mientras HeapSort mostró una mayor estabilidad y menor sensibilidad a la distribución inicial de los datos.

En términos prácticos, QuickSort es ideal para implementaciones donde la velocidad media es prioritaria y el tamaño de los datos no supera el rango medio, mientras que HeapSort se recomienda para entornos donde la predictibilidad y la consistencia son más importantes que la rapidez absoluta.

Finalmente, el estudio reafirma que la elección del algoritmo de ordenamiento no depende únicamente de su complejidad teórica, sino también de las características del hardware y del tipo de datos procesados. Estos resultados sientan las bases para futuras investigaciones orientadas a algoritmos híbridos o adaptativos que combinen las ventajas de ambos métodos.

En resumen, esta investigación no solo contribuye a la comprensión de la eficiencia de QuickSort y HeapSort, sino que también destaca la complejidad del rendimiento algorítmico en el contexto del hardware y los datos. Los resultados obtenidos ofrecen un marco analítico que puede guiar futuras investigaciones y desarrollos en el campo de la informática y el procesamiento de datos.

Anexos

Anexo D: Datos de Experimentación

Los datos utilizados para los análisis de los algoritmos de ordenamiento fueron obtenidos y exportados desde un archivo de Excel. A continuación se presenta la tabla completa de resultados para los distintos patrones de entrada y tamaños de datos.

Patrón de Entrada	Algoritmo	N=1,000 (ms)	N=10,000 (ms)	N=100,000 (ms)
Aleatorio uniforme	QuickSort	0.067	0.700	16.600
Aleatorio uniforme	HeapSort	0.200	2.067	36.000
Ordenado ascendente	QuickSort	0.033	0.200	2.000
Ordenado ascendente	HeapSort	0.133	1.700	26.867
Ordenado descendente	QuickSort	0.000	0.133	1.833
Ordenado descendente	HeapSort	0.100	1.567	26.133
Casi ordenado 5 %	QuickSort	0.000	0.133	4.133
Casi ordenado 5 %	HeapSort	0.133	1.700	27.600
Duplicados 10–20	QuickSort	0.033	0.467	5.600
Duplicados 10–20	HeapSort	0.133	1.800	32.267

Nota: Los tiempos se muestran en milisegundos (ms) y corresponden al promedio de múltiples ejecuciones para garantizar la consistencia de los datos.Mas dato en https://docs.google.com/spreadsheets/d/1CquQp2RjPdySvD4WXcEI4jn9C9Ci7XYCHOn-AoEcl-M/edit?usp=sharing

Anexo A. Header de Ordenamientos (‘ordenamientos.h‘)

Listing 1: Archivo de cabecera para QuickSort y HeapSort.

```
#ifndef ORDENAMIENTOS_H
#define ORDENAMIENTOS_H

void heapSort(int arr[], int size);
void quickSort(int arr[], int low, int high);

#endif
```

Anexo B. Funciones de Ordenamiento (‘ordenamientos.cpp‘)

Listing 2: Implementación de HeapSort y QuickSort.

```
#include "ordenamientos.h"
#include <algorithm>

// ------------------- HEAPSORT -------------------
void heapify(int arr[], int size, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    if (left < size && arr[left] > arr[largest])
        largest = left;

    if (right < size && arr[right] > arr[largest])
        largest = right;

    if (largest != i) {
        std::swap(arr[i], arr[largest]);
        heapify(arr, size, largest);
    }
}
```

```
void heapSort(int arr[], int size) {
    for (int i = size / 2 - 1; i >= 0; --i)
        heapify(arr, size, i);

    for (int i = size - 1; i >= 0; --i) {
        std::swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

// ------------------- QUICKSORT -------------------
void quickSort(int A[], int inicio, int fin) {
    int i = inicio;
    int j = fin;
    int pivote = A[(inicio + fin) / 2];

    while (i <= j) {
        while (A[i] < pivote) i++;
        while (A[j] > pivote) j--;
        if (i <= j) {
            std::swap(A[i], A[j]);
            i++;
            j--;
        }
    }

    if (inicio < j) quickSort(A, inicio, j);
    if (i < fin) quickSort(A, i, fin);
}
```

Anexo C. Menú y Generación de Datos (‘main.cpp‘)

Listing 3: Menú de selección de algoritmo, tamaño y patrón de datos, ejecución y registro de resultados.

```
#include "ordenamientos.h"
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>
#include <random>
#include <fstream>

using namespace std;

long long comparaciones = 0;
long long intercambios = 0;
int profundidad = 0;

string nombrePatron(int opcionSemilla) {
    switch(opcionSemilla) {
        case 1: return "Aleatorio␣uniforme";
        case 2: return "Ordenado␣ascendente";
        case 3: return "Ordenado␣descendente";
        case 4: return "Casi␣ordenado␣5%";
        case 5: return "Duplicados␣10-20";
        default: return "Desconocido";
    }
}

vector<int> generarArreglo(int size, int tipoSemilla) {
    vector<int> arr(size);
    random_device rd;
    mt19937 gen(rd());

    switch(tipoSemilla) {
```

```
        case 1: {
            uniform_int_distribution<> dis(1, 100000);
            for (int i = 0; i < size; ++i) arr[i] = dis(gen);
            break;
        }
        case 2:
            for (int i = 0; i < size; ++i) arr[i] = i+1;
            break;
        case 3:
            for (int i = 0; i < size; ++i) arr[i] = size-i;
            break;
        case 4: {
            for (int i = 0; i < size; ++i) arr[i] = i+1;
            uniform_int_distribution<> dis(0, size-1);
            for (int i = 0; i < size*0.05; ++i) {
                swap(arr[dis(gen)], arr[dis(gen)]);
            }
            break;
        }
        case 5: {
            uniform_int_distribution<> dis(1, 20);
            for (int i = 0; i < size; ++i) arr[i] = dis(gen);
            break;
        }
        default:
            break;
    }
    return arr;
}


void menu() {
    int opcionAlgoritmo, opcionTamano, opcionSemilla;
    vector<int> tamanos = {1000, 10000, 100000};

    cout << "===␣MEN ␣DE␣ORDENAMIENTO␣===\n";
    cout << "1.␣QuickSort\n";
    cout << "2.␣HeapSort\n";
    cout << "Elige␣algoritmo␣(1␣o␣2):␣";
    cin >> opcionAlgoritmo;

    cout << "Tama o␣del␣arreglo:\n";
    for (int i = 0; i < tamanos.size(); i++) {
        cout << i+1 << ".␣" << tamanos[i] << "\n";
    }
    cout << "Elige␣opci n␣(1-3):␣";
    cin >> opcionTamano;
    if(opcionTamano < 1 || opcionTamano > 3) {
        cout << "Opci n␣inv lida,␣usando␣tama o␣1000␣por␣defecto.\n";
        opcionTamano = 1;
    }
    int N = tamanos[opcionTamano-1];

    cout << "Tipo␣de␣semilla:\n";
    cout << "1.␣Aleatorio␣uniforme\n2.␣Ordenado␣ascendente\n3.␣Ordenado␣descendente\n
        ";
    cout << "4.␣Casi␣ordenado␣(5%)\n5.␣Con␣duplicados␣(10-20␣valores␣ nicos )\nElige␣
        opci n␣(1-5):␣";
    cin >> opcionSemilla;
    if(opcionSemilla < 1 || opcionSemilla > 5) {
        cout << "Opci n␣inv lida,␣usando␣semilla␣Aleatorio␣uniforme␣por␣defecto.\n"
            ;
        opcionSemilla = 1;
    }

    // Archivo de resultados
```

```
ofstream archivo("resultados_ordenamiento.txt", ios::out);
if (!archivo.is_open()) {
    cerr << "Error al abrir el archivo para guardar resultados.\n";
    return;
}
archivo << "Iteraci n,Algoritmo,N,Patr n,Tiempo(ms),Comparaciones,Intercambios,
    Profundidad,CommitHash\n";

// Asignar una semilla fija por patr n
unsigned int semillaFija;
switch(opcionSemilla) {
    case 1: semillaFija = 111; break;
    case 2: semillaFija = 222; break;
    case 3: semillaFija = 333; break;
    case 4: semillaFija = 444; break;
    case 5: semillaFija = 555; break;
    default: semillaFija = 123; break;
}

mt19937 gen(semillaFija); // Generador con semilla fija


for (int iter = 0; iter < 30; ++iter) {
    vector<int> arr(N);

    // Generar arreglo seg n el tipo de semilla
    switch(opcionSemilla) {
        case 1: {
            uniform_int_distribution<> dis(1, 100000);
            for (int i = 0; i < N; ++i) arr[i] = dis(gen);
            break;
        }
        case 2:
            for (int i = 0; i < N; ++i) arr[i] = i+1;
            break;
        case 3:
            for (int i = 0; i < N; ++i) arr[i] = N-i;
            break;
        case 4: {
            for (int i = 0; i < N; ++i) arr[i] = i+1;
            uniform_int_distribution<> dis(0, N-1);
            for (int i = 0; i < N*0.05; ++i) {
                swap(arr[dis(gen)], arr[dis(gen)]);
            }
            break;
        }
        case 5: {
            uniform_int_distribution<> dis(1, 20);
            for (int i = 0; i < N; ++i) arr[i] = dis(gen);
            break;
        }
    }

    comparaciones = 0;
    intercambios = 0;
    profundidad = 0;

    auto inicio = chrono::high_resolution_clock::now();

    if (opcionAlgoritmo == 1) {
        quickSort(arr.data(), 0, N-1);
    } else {
        heapSort(arr.data(), N);
    }

    auto fin = chrono::high_resolution_clock::now();
```

```
        auto tiempo = chrono::duration_cast<chrono::milliseconds>(fin - inicio).count
            ();

        string algoritmo = (opcionAlgoritmo==1 ? "QuickSort" : "HeapSort");
        string patron = nombrePatron(opcionSemilla);
        string commitHash = "ABC123";

        cout << "Iteraci n␣" << iter+1 << ":␣"
             << "Algoritmo=" << algoritmo
             << ",␣N=" << N
             << ",␣Patr n=" << patron
             << ",␣Tiempo=" << tiempo << "ms"
             << ",␣Comparaciones=" << comparaciones
             << ",␣Intercambios=" << intercambios
             << ",␣Profundidad=" << profundidad
             << ",␣CommitHash=" << commitHash << "\n";

        archivo << iter+1 << "," << algoritmo << "," << N << "," << patron << ","
                << tiempo << "," << comparaciones << "," << intercambios << ","
                << profundidad << "," << commitHash << "\n";
    }

    archivo.close();
    cout << "\nResultados␣guardados␣en␣'resultados_ordenamiento.txt'\n";
}
% ------------------- ANEXO: Datos de Excel -------------------
```

Referencias

1. Edelkamp, S., & Weiß, A. (2018). Worst-case efficient sorting with QuickMergesort. arXiv. https://arxiv.org/abs/1811.00833
2. Edelkamp, S., & Weiß, A. (2018). QuickMergesort: Practically efficient constant-factor optimal sorting. arXiv. https://arxiv.org/abs/1804.10062
3. Božidar, D., & Dobravec, T. (2015). Comparison of parallel sorting algorithms. arXiv. https://arxiv.org/abs/1511.03404
4. Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley Professional.
5. Hoare, C. A. R. (1962). Quicksort. *The Computer Journal*, 5(1), 10–15.
6. Williams, J. W. J. (1964). Algorithm 232: Heapsort. *Communications of the ACM*, 7(6), 347–348.