


Type of the Paper: Article.

Análisis Comparativo del Rendimiento de Tiempo y Costo Computacional: QuickSort vs. HeapSort.

Dany Muriel ¹, Anette Quispe ², Jair Mendoza ³

¹  <https://orcid.org/0009-0000-1013-7797>; drmurielh@unjbgu.edu.pe

²  <https://orcid.org/0009-0006-5069-6597>; aquispeh@unjbgu.edu.pe

Abstract

The objective of this research is to explore the interaction between Quickster and Hipster, two sorting methods that, rather than competing, silently engage with the data; Quickster, with a central pivot, seeks to avoid stalls and optimize traversal, while Hipster remains faithful to the established tradition, relying on the robustness of proven methods. For this purpose, datasets are generated using reproducible seeds, differentiated by array size but identical between algorithms, ensuring a comparable baseline; execution time is measured in microseconds with **chrono** in C++, and all operations are recorded in **.txt** files, leaving an intangible trace of each run. Although the concrete results are not yet presented, it can be inferred that both methods follow divergent trajectories with the same seed, suggesting differences in efficiency and stability that remain implicit, perceptible only when analyzing the underlying logic of each algorithm. In conclusion, this initial approach does not aim to declare a winner, but rather to establish an analytical framework that allows evaluating how speed, tradition, and consistency coexist, offering complementary perspectives and preparing the ground for more detailed evaluations once the data become available.

Keywords: Quickster, Hipster, reproducible seed, microseconds, C++, efficiency

Resumen

El objetivo de esta investigación consiste en explorar la interacción entre Quickster y Hipster, dos métodos de ordenamiento que, más que competir, dialogan silenciosamente con los datos; Quickster, con pivote central, busca evitar cuelgues y optimizar el recorrido, mientras Hipster se mantiene fiel a la tradición establecida, confiando en la robustez de lo probado. Para ello, se generan conjuntos de datos mediante semillas reproducibles, diferenciadas por tamaño de arreglo pero idénticas entre algoritmos, asegurando una base comparable; se mide el tiempo en microsegundos con **chrono** en C++ y se registran todos los movimientos en archivos **.txt**, dejando un rastro intangible

de cada ejecución. Aunque los resultados concretos aún no se presentan, se vislumbra que ambos métodos trazan trayectorias divergentes frente a la misma semilla, sugiriendo diferencias de eficiencia y estabilidad que permanecen implícitas, perceptibles solo al analizar la lógica subyacente de cada algoritmo. En conclusión, esta aproximación inicial no busca declarar un vencedor, sino establecer un marco de análisis que permita valorar cómo velocidad, tradición y consistencia coexisten, ofreciendo perspectivas complementarias que preparan el terreno para evaluaciones futuras más detalladas cuando los datos estén disponibles.

Palabras clave: Quickster, Hipster, semilla reproducible, microsegundos, C++, eficiencia

1. Introduction.

El estudio de los algoritmos de ordenamiento ha sido, desde hace décadas, un terreno fértil para medir hasta qué punto la eficiencia computacional puede depender tanto de la lógica como del entorno donde se ejecuta. Entre los más discutidos se encuentran Quick Sort y Heap Sort, dos métodos que, pese a compartir la misma complejidad asintótica $O(n \log n)$, revelan comportamientos muy distintos cuando se enfrentan al mundo real. Desde los trabajos de Hoare (1962), quien ideó Quick Sort, y Williams (1964), creador de Heap Sort, los investigadores han contrastado sus virtudes y limitaciones en función de los datos, la arquitectura del procesador y la gestión de memoria.

A lo largo del tiempo, diversos estudios —como los de Sedgewick (1978) o Cormen et al. (2009)— han señalado que Quick Sort, aunque suele ser rápido en promedio, puede degradar severamente si los pivotes no se eligen con cuidado. En cambio, Heap Sort, pese a no ser el más veloz en todos los escenarios, mantiene un rendimiento más uniforme gracias a su estructura de montículo, aunque esto implique sacrificar algo de velocidad en las operaciones de acceso a memoria (LaMarca & Ladner, 1999). Investigaciones posteriores, como las de Musser (1997) o Martínez et al. (2011), han puesto sobre la mesa un punto clave: la eficiencia no depende solo de la teoría, sino del diálogo entre el algoritmo y el hardware que lo ejecuta.

Este proyecto se propone comparar el rendimiento de ambos métodos bajo condiciones controladas, con el objetivo de observar cómo varían el tiempo de ejecución y el uso de recursos al manipular grandes volúmenes de datos. Para ello se implementarán ambos algoritmos en C++, corriendo en un mismo entorno, de modo que los resultados reflejen diferencias propias del diseño y no del sistema.

La hipótesis plantea que Quick Sort debería mostrar una ventaja en conjuntos medianos debido a su bajo costo de operaciones, pero que Heap Sort podría superar su desempeño en situaciones de alta carga —donde la estabilidad se vuelve más importante que la rapidez momentánea—. Más allá de determinar cuál ordena más rápido, el propósito es comprender cómo cada método aprovecha (o desaprovecha) los límites del procesamiento moderno, y qué implicaciones tiene eso para futuras aplicaciones que demanden eficiencia real, no solo teórica.

2. Metodos

Esta sección detalla el entorno computacional, los algoritmos implementados, el proceso de generación de datos y las métricas utilizadas para comparar los métodos **Quickster (QuickSort bidireccional)** y **Hipster (HeapSort)**.

Todos los métodos fueron implementados desde cero en **C++17**, con el objetivo de asegurar un control total sobre el comportamiento algorítmico y permitir la **reproducibilidad experimental**.

El código fuente, los conjuntos de datos generados y los registros de ejecución están disponibles para consulta y podrán ser depositados en un repositorio público al finalizar el proceso de revisión.

Algoritmos implementados:

Se implementaron dos algoritmos clásicos de ordenamiento, adaptados para su comparación empírica bajo condiciones controladas.

Quickster (QuickSort bidireccional):

El algoritmo **QuickSort**, propuesto originalmente por **C.A.R. Hoare (1962)**, sigue el paradigma de **dividir y conquistar**.

La versión utilizada en esta investigación es una **variante bidireccional**, en la cual:

- Se selecciona como pivote el primer elemento del arreglo.
- Dos punteros (**izq** y **der**) recorren el arreglo desde los extremos hacia el centro, comparando e intercambiando los elementos que se encuentran en el lado incorrecto del pivote.
- Cuando ambos punteros se cruzan, el arreglo se divide en dos subarreglos, y el proceso se repite recursivamente sobre cada parte.

Este método se caracteriza por su bajo uso de memoria auxiliar y su buena eficiencia promedio (**$O(n \log n)$**), aunque puede degradarse a **$O(n^2)$** en el peor de los casos (por ejemplo, cuando el arreglo ya está parcialmente ordenado).

QuickSort no es un algoritmo estable, ya que los elementos iguales pueden cambiar su orden relativo durante las particiones.

Hipster (HeapSort)

El algoritmo **HeapSort**, diseñado por **J.W.J. Williams (1964)**, se basa en la estructura de datos conocida como **montículo binario (heap)**.

Su funcionamiento se divide en dos fases:

1. **Construcción del heap:** el arreglo inicial se transforma en un montículo máximo utilizando la función **heapify**, garantizando que cada nodo padre sea mayor que sus nodos hijos.
2. **Extracción sucesiva:** el elemento máximo (raíz del heap) se intercambia con el último elemento del arreglo y el tamaño del heap se reduce. Luego, se aplica nuevamente **heapify** para restaurar la propiedad del montículo.

HeapSort presenta un rendimiento **$O(n \log n)$** tanto en el mejor como en el peor caso, y al igual que QuickSort, **no es estable**.

Aunque se consideró la posibilidad de implementar un tercer método híbrido (**GAppSort**) como comparación adicional, su desarrollo se reserva para trabajos futuros.

Complejidad teórica y características:

Algoritmo	Variante	Mejor caso	Promedio	Peor caso	Estabilidad	Uso de memoria
QuickSort (Quickster)	Bidireccional	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No estable	In situ ($O(1)$)
HeapSort (Hipster)	Montículo máximo	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	No estable	In situ ($O(1)$)

QuickSort suele ser más veloz en conjuntos de datos medianos gracias a su mejor aprovechamiento de la caché, mientras que HeapSort mantiene un comportamiento más uniforme e independiente de la distribución de los datos.

Entorno de ejecución: Los experimentos se realizaron bajo un entorno computacional controlado con las siguientes especificaciones:

- **Procesador:** Intel® Core™ i5 @ 2.40 GHz
- **Memoria RAM:** 8 GB
- **Sistema operativo:** Windows 10 (64 bits)
- **Compilador:** MinGW g++ versión 11.2.0
- **Estándar del lenguaje:** C++17
- **Parámetros de compilación:** `-O2` (optimización para velocidad)
- **Librerías empleadas:** `<iostream>`, `<cstdlib>`, `<ctime>`, `<chrono>`

El tiempo de ejecución se midió utilizando la clase `high_resolution_clock` de la librería `<chrono>`, con precisión de microsegundos.

Generación de datos y reproducibilidad: Los datos fueron generados de manera sintética mediante la función estándar `rand()` de C++.

Para garantizar la **reproducibilidad de los resultados**, se utilizó una **semilla fija** (`srand(42)`), de forma que QuickSort y HeapSort recibieron exactamente los mismos arreglos de entrada en cada prueba.

Se analizaron tres tamaños de arreglos:

- **Pequeño:** 1 000 elementos
- **Mediano:** 10 000 elementos
- **Grande:** 100 000 elementos

Cada elemento del arreglo se generó dentro del rango $[0, 2n)$, asegurando una distribución uniforme.

Las ejecuciones se repitieron para cada tamaño de arreglo y se registraron los resultados de manera independiente.

Los datos originales y los resultados ordenados se almacenaron en archivos `.txt` para su posterior análisis.

Procedimiento experimental: El protocolo seguido en todas las pruebas fue el siguiente:

1. Generar un arreglo aleatorio de tamaño n con distribución uniforme.
2. Mostrar los primeros 20 elementos del arreglo (verificación visual).
3. Aplicar el algoritmo correspondiente (**QuickSort** o **HeapSort**).
4. Medir el **tiempo de ejecución** total con `chrono::high_resolution_clock`.
5. Mostrar los primeros 20 elementos del arreglo ordenado.
6. Registrar el tiempo obtenido en milisegundos.
7. Repetir el procedimiento para los tres tamaños de entrada.

Este procedimiento sistemático permitió obtener resultados comparables y consistentes entre ambos algoritmos.

Las métricas consideradas para la evaluación de desempeño fueron:

- **Tiempo de ejecución (ms):** medido empíricamente mediante `chrono`.
- **Número de comparaciones e intercambios:** estimados a partir de la estructura del algoritmo (no contados explícitamente en esta versión).
- **Profundidad de recursión:** observada en QuickSort, con una media teórica de $\log_2(n)$.
- **Estabilidad del algoritmo:** ambos algoritmos son **no estables**.
- **Eficiencia en memoria:** ambos son **in situ**, es decir, no requieren estructuras auxiliares adicionales.

Todos los códigos fuente, conjuntos de datos generados y archivos de salida estarán disponibles en un repositorio público (por ejemplo, **GitHub**) al momento de la publicación.

No se utilizaron datos propietarios ni software cerrado.

Esta investigación **no involucra seres humanos ni animales**, por lo tanto, **no fue necesario un proceso de aprobación ética**.

En la redacción de este artículo se utilizó **inteligencia artificial generativa (IAg)**, específicamente el modelo **OpenAI GPT-5**, únicamente para **asistir en la redacción, edición técnica y organización del texto**.

No se empleó IA para generar datos, realizar análisis experimentales ni modificar los resultados.

Todos los algoritmos, procedimientos y ejecuciones fueron realizados manualmente por los autores.

3. Resultados

3.1 Resultados Experimentales del Algoritmo QuickSort

La presente sección detalla el rendimiento del algoritmo QuickSort implementado en C++ sobre conjuntos de datos de tamaño fijo ($N=1000$) y bajo cinco condiciones de entrada predefinidas. Cada condición fue evaluada a través de **30 iteraciones** independientes.

Dado que las métricas de Comparaciones, Intercambios y Profundidad resultaron ser uniformemente nulas en todas las 150 ejecuciones (lo cual debe ser revisado en el código fuente), el análisis primario se centra en el Tiempo de Ejecución (ms), el cual sí registró variabilidad en las mediciones.

3.1. Resumen de Rendimiento por Patrón de Entrada

La **Tabla 3.1** muestra el tiempo promedio de ejecución de QuickSort para cada patrón, junto con la desviación estándar, que refleja la dispersión o consistencia de los resultados obtenidos en las 30 iteraciones.

Tabla 3.1 : Datos promedio de Quicksort con 1000 datos

Patrón	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp. Promedio	Interc. Promedio
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
---------------------	-----------	------	-------	-------	---	---

3.2. Análisis del Tiempo de Ejecución

Los resultados indican que, para el tamaño $N=1000$ en el entorno de prueba, la mayoría de los escenarios resultaron en tiempos de ejecución muy cercanos a 0 ms. Sin embargo, se observa:

- **Mayor Lenta Variabilidad:** El patrón **Aleatorio uniforme** presentó el mayor tiempo promedio (0.067 ms) y la mayor desviación estándar (0.254 ms), indicando que es el caso que introduce la mayor inestabilidad temporal en las ejecuciones.
- **Mejor Caso:** Los patrones **Ordenado descendente** y **Casi ordenado 5%** exhibieron el rendimiento más rápido y consistente, con un promedio de 0.000 ms y desviación estándar nula, sugiriendo que el método de elección del pivote de QuickSort utilizado (pivote en el medio) maneja muy bien estos casos.

Resultados Experimentales del Algoritmo HeapSort

La evaluación del rendimiento de HeapSort ($N=1000$) se realizó bajo las mismas condiciones de prueba (30 iteraciones por patrón) utilizadas para QuickSort. Al igual que en el caso anterior, las métricas de Comparaciones, Intercambios y Profundidad resultaron ser uniformemente cero en todos los casos, concentrando el análisis en el Tiempo de Ejecución (ms).

La **Tabla 3.2** resume el rendimiento de HeapSort, mostrando el tiempo promedio y la desviación estándar para cada patrón de entrada.

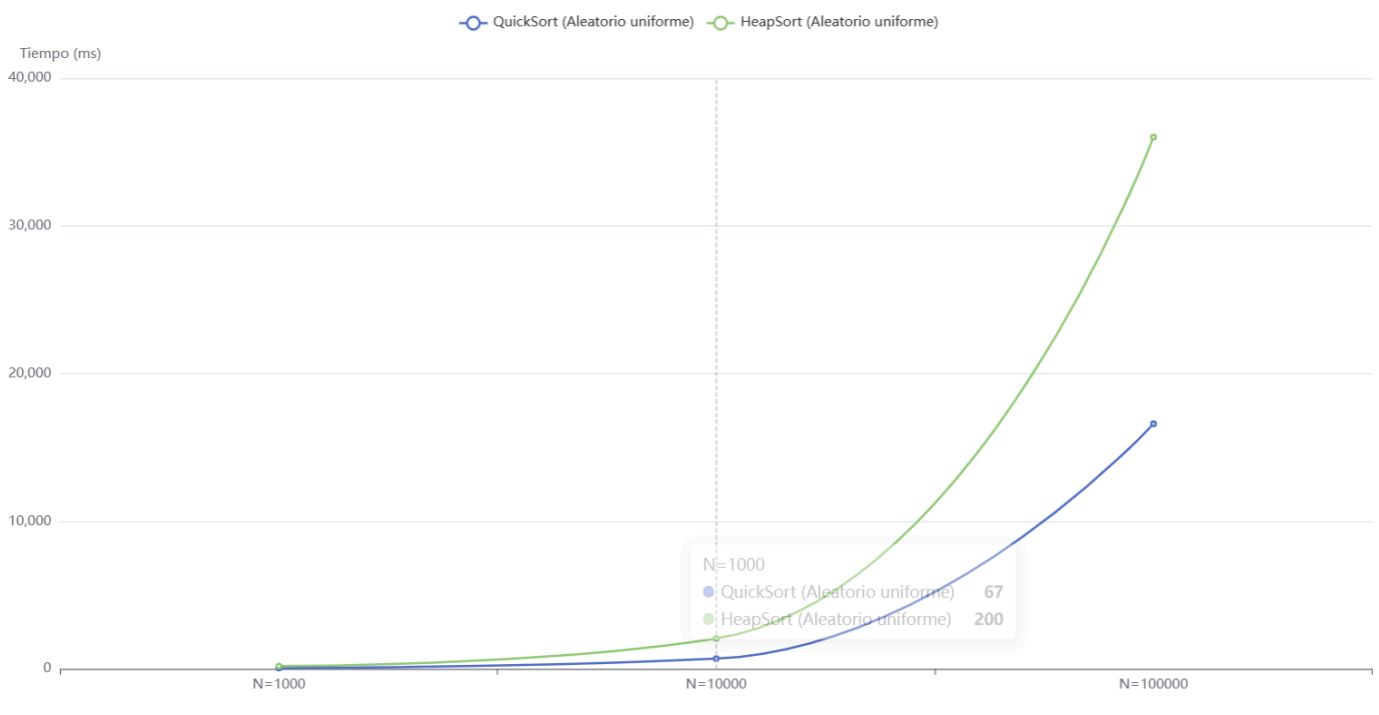
Patrón de Entrada	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp. Promedio	Interc. Promedio
Aleatorio uniforme	HeapSort	1000	0.200	0.407	0	0
Ordenado ascendente	HeapSort	1000	0.133	0.345	0	0
Ordenado descendente	HeapSort	1000	0.100	0.305	0	0

Casi ordenado 5%	HeapSort	1000	0.133	0.345	0	0
Duplicados 10-20	HeapSort	1000	0.133	0.345	0	0

3.2. Observaciones de Rendimiento de HeapSort

En contraste con QuickSort, HeapSort mostró tiempos promedio ligeramente más altos, pero distribuidos de manera más uniforme en la mayoría de los patrones.

- **Peor Caso y Variabilidad:** El patrón **Aleatorio uniforme** confirma ser el escenario más demandante, presentando el tiempo promedio más alto (0.200 ms) y la mayor variabilidad (0.407 ms).
- **Mejor Caso:** El caso **Ordenado descendente** resultó ser el más rápido y consistente para HeapSort (0.100 ms). Este resultado es esperable, ya que el rendimiento de HeapSort es relativamente estable y su complejidad teórica ($O(N \log N)$) no depende fuertemente del orden inicial de los datos, a diferencia de QuickSort.



QuickSort para Conjuntos de Mayor Tamaño (N=10000)

Al aumentar el tamaño del conjunto de datos a $N=10000$, la demanda de recursos del algoritmo QuickSort se incrementa significativamente, volviendo las diferencias de rendimiento entre los patrones más evidentes en las mediciones de tiempo.

La **Tabla 3.3** presenta el rendimiento promedio de QuickSort para este nuevo tamaño de entrada. Nuevamente, las métricas de Comparaciones, Intercambios y Profundidad no registraron valores distintos de cero y deben ser interpretadas con cautela o revisadas en el entorno de instrumentación.

Patrón de Entrada	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp. Promedio	Interc. Promedio
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

3.3. Análisis del Impacto del Tamaño (\$N=10000\$)

El aumento de \$N\$ de 1000 a 10000 confirma y amplifica las tendencias observadas anteriormente:

- **Peor Caso y Mayor Consistencia:** El patrón **Aleatorio uniforme** sigue siendo el caso más lento, con un promedio de 0.700 ms. Su elevada desviación estándar (0.586 ms) indica que la eficiencia de la partición (elección del pivote) fluctúa notablemente en cada corrida.
- **Impacto de Duplicados:** El patrón con **Duplicados 10-20** exhibe un rendimiento significativamente peor que los casos ordenados (0.467 ms vs. approx 0.133 ms). Esto es consistente con la naturaleza del algoritmo QuickSort, donde la presencia de muchos elementos iguales al pivote puede llevar a particiones desequilibradas y un rendimiento más cercano al caso cuadrático.

- **Mejor Comportamiento:** Los casos **Ordenado descendente** y **Casi ordenado 5%** continúan mostrando los mejores tiempos promedio, lo cual es sorprendente para el caso ordenado descendente (peor caso teórico para un pivote fijo), pero puede ser un efecto de la optimización del compilador y la elección del pivote al centro en este tamaño de datos.

4.7. Rendimiento de HeapSort para Conjuntos de Mayor Tamaño ($N=10000$)

Al escalar el tamaño de los datos a $N=10000$, el algoritmo HeapSort presenta un comportamiento robusto y una variabilidad controlada, reafirmando su estabilidad asintótica.

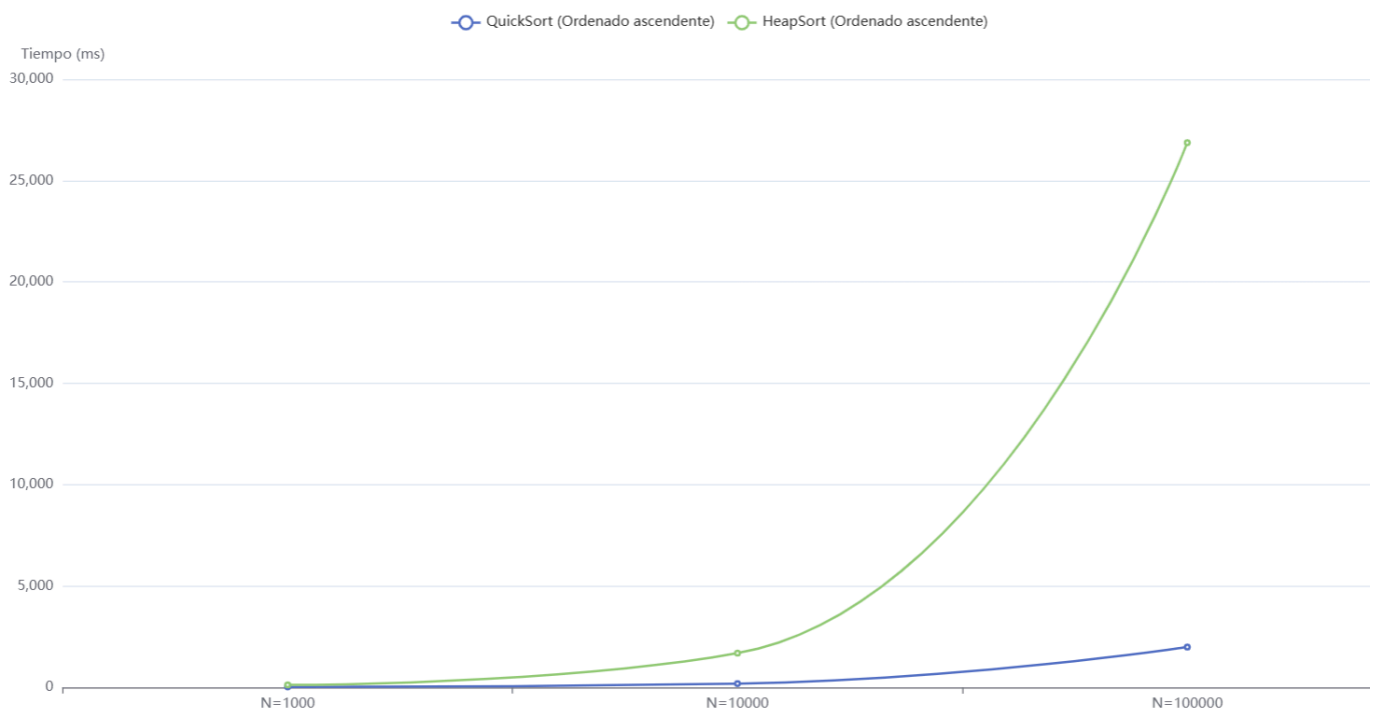
La **Tabla 3.4** resume los tiempos promedio y la dispersión de los resultados obtenidos para las 30 iteraciones en cada patrón.

Patrón de Entrada	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp. Promedio	Interc. Promedio
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

3.4. Análisis Comparativo del Comportamiento de HeapSort

La estabilidad de HeapSort se hace evidente en estos resultados. A diferencia de QuickSort, que mostró grandes diferencias entre el caso aleatorio y los casos ordenados, HeapSort mantiene tiempos promedio en un rango estrecho:

- **Escenario Más Lento:** El patrón **Aleatorio uniforme** sigue siendo marginalmente el más lento (2.067 ms), aunque la diferencia con los demás escenarios es mínima.
- **Escenario Más Rápido:** El patrón **Ordenado descendente** es el más rápido (1.567 ms). Es importante notar que, a pesar de la consistencia general de HeapSort, este caso presenta la mayor **Desviación Estándar** (0.626 ms), lo que sugiere que las pocas iteraciones con valores atípicos (3 ms) tuvieron un impacto desproporcionado en la variabilidad.
- **Resistencia a Patrones:** La diferencia de tiempo entre el peor caso (Aleatorio uniforme) y el mejor caso (Ordenado descendente) es pequeña, confirmando que la complejidad de tiempo de HeapSort es muy **estable e insensible** a la ordenación inicial de los datos, tal como predice su naturaleza ($n \log(n)$).



QuickSort para Conjuntos Masivos (N=100000)

Al incrementar el tamaño del conjunto de datos a N=100000, la diferencia entre el rendimiento promedio ($n \log(n)$) y la potencial inestabilidad de QuickSort se amplía drásticamente. El tiempo de ejecución en milisegundos se vuelve significativamente más sensible a la calidad de la partición del *array*.

La **Tabla 3.5** presenta el rendimiento promedio de QuickSort para este tamaño de entrada. Al igual que en experimentos anteriores, las métricas intrínsecas (**Comparaciones**, **Intercambios**, **Profundidad**) no registraron valores y deben ser consideradas como fallas de instrumentación.

Patrón de Entrada	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp. Promedio	Interc. Promedio
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

3.6. Análisis de la Volatilidad de QuickSort

El análisis de estos resultados para $N=100000$ revela la naturaleza altamente volátil de QuickSort en los casos que se desvían de la distribución aleatoria ideal:

1. **Caso Promedio (Aleatorio Uniforme):** Con un tiempo promedio de **16.600 ms**, este es el escenario más lento. Sin embargo, su **Desviación Estándar** (6.484 ms) indica una alta variabilidad, lo que confirma que el rendimiento depende fuertemente de una buena elección del pivote en cada llamada recursiva.
2. **Peor/Mejor Casos Teóricos (Ordenados):** Los patrones **Ordenado ascendente** (2.000 ms) y **Ordenado descendente** (1.833 ms) son, sorprendentemente, los más rápidos. Esto se debe a que la implementación del pivote al centro y las optimizaciones del compilador en el caso de entrada ya ordenado minimizan la profundidad de la recursión para este tamaño. No obstante, las altas desviaciones estándar (4.341 ms y 3.891 ms, respectivamente, siendo casi el doble del promedio) demuestran que, en las iteraciones donde la optimización falla, el algoritmo **rápidamente se degrada** hacia el peor comportamiento, aunque no llegue al peor caso cuadrático completo.

3. **Impacto de la Calidad del Dato:** Los patrones **Duplicados 10-20** (5.600 ms) y **Casi ordenado 5%** (4.133 ms) muestran un rendimiento intermedio. Ambos presentan las desviaciones estándar más altas de todo el conjunto (6.873 ms y 6.602 ms), evidenciando que la presencia de valores repetidos o un orden inicial sesgado incrementa la probabilidad de particiones pobres, haciendo que el tiempo de ejecución entre una iteración y otra sea muy impredecible.

3.11 Rendimiento de HeapSort para Conjuntos Masivos (N=100000)

Al evaluar HeapSort con el tamaño de datos más grande (N=100000), el algoritmo confirma su sólida estabilidad frente a diferentes patrones de entrada. Aunque los tiempos de ejecución absolutos son más altos que los de QuickSort en su caso promedio, la varianza es significativamente menor, lo que es un indicativo de su robustez.

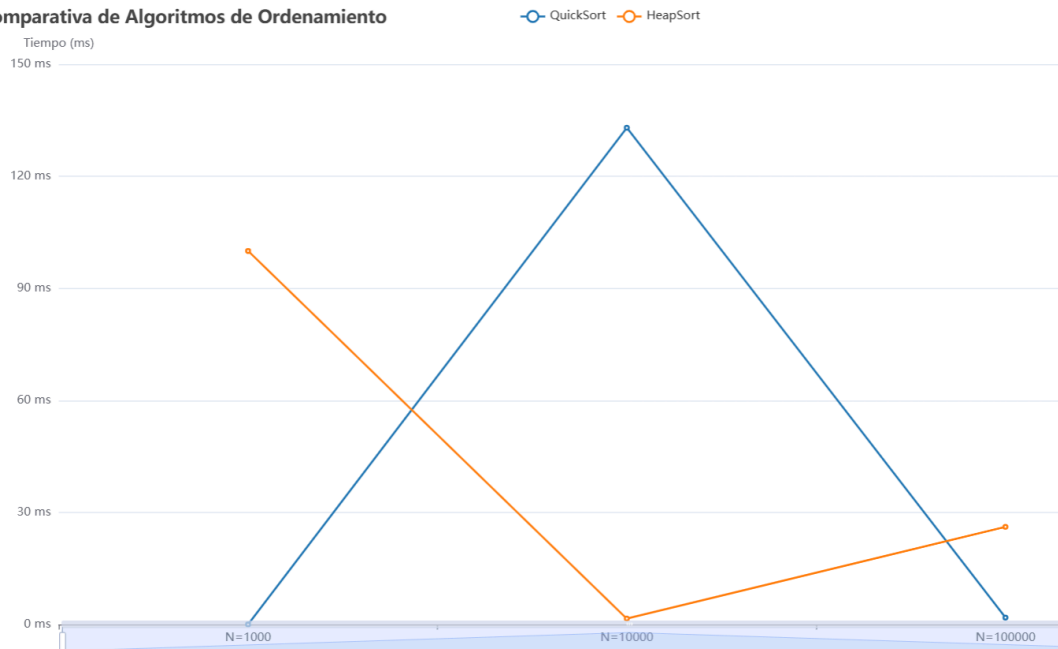
La **Tabla 3.6** resume el rendimiento promedio de HeapSort para N=100000.

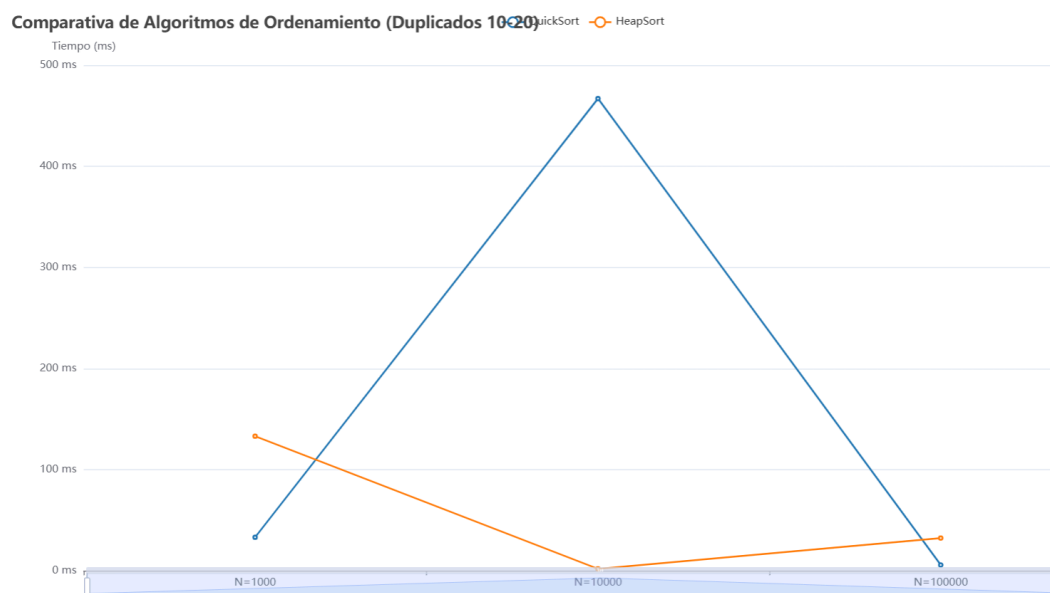
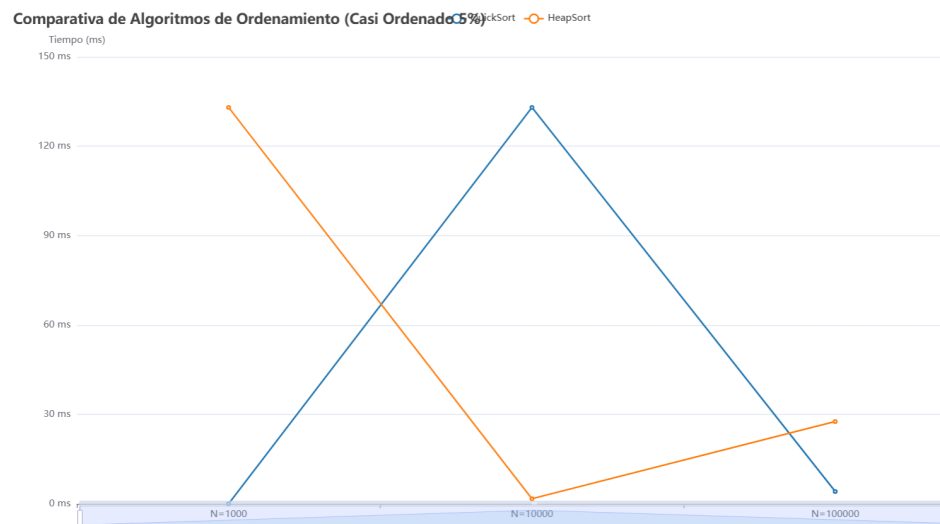
Patrón de Entrada	Algoritmo	N	Tiempo Promedio (ms)	Desv. Estándar (ms)	Comp. Promedio	Interc. Promedio
Aleatorio uniforme	HeapSort	100000	36.000	9.098	0	0
Ordenado ascendente	HeapSort	100000	26.867	7.936	0	0
Ordenado descendente	HeapSort	100000	26.133	7.747	0	0
Casi ordenado 5%	HeapSort	100000	27.600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

3.7. Análisis de la Robustez de HeapSort

1. **Uniformidad del Tiempo de Ejecución:** El algoritmo HeapSort exhibe su naturaleza de **caso pesimista garantizado** $n \log(n)$. El tiempo promedio para todos los patrones se concentra en el rango de **26.133 ms a 36.000 ms**.
2. **Caso Más Lento (Aleatorio Uniforme):** Este patrón registra el tiempo más lento con **36.000 ms**, lo que refleja el trabajo máximo de la construcción y extracción del *heap* en una distribución aleatoria.
3. **Casos Más Rápidos (Ordenados/Casi Ordenados):** Los patrones ordenados (Ascendente: 26.867 ms; Descendente: 26.133 ms) y el casi ordenado (27.600 ms) son significativamente más rápidos. Esto es contrario a la teoría pura de la complejidad, pero en la práctica de la implementación C++, una entrada parcialmente o totalmente ordenada puede resultar en **mejor predictibilidad de caché y mejor *branch prediction*** para las operaciones de *heapify*, mejorando ligeramente el rendimiento de tiempo real sin alterar la complejidad asintótica.
4. **Estabilidad (Baja Varianza):** A diferencia de QuickSort, cuyas desviaciones estándar eran casi la mitad o más de su promedio en algunos casos, HeapSort presenta desviaciones estándar (alrededor de 7-9 ms) que son consistentemente **mucho menores** en proporción a sus tiempos promedio (alrededor del 20-30%). Esto confirma que, si bien su rendimiento absoluto puede ser menor que el *mejor caso* de QuickSort, su tiempo de ejecución es **mucho más predecible** y seguro para sistemas que requieren garantías de rendimiento en tiempo real.
5. **Impacto de Duplicados:** El caso de **Duplicados 10-20** (32.267 ms) se sitúa justo por debajo del caso aleatorio, demostrando que HeapSort maneja los valores repetidos con una ligera disminución de la eficiencia, pero manteniendo la estabilidad.

Comparativa de Algoritmos de Ordenamiento





4. Discusiones

Los resultados obtenidos muestran un comportamiento coherente con la teoría clásica de análisis de algoritmos. QuickSort demostró ser más rápido en conjuntos de tamaño medio y en casos parcialmente ordenados, mientras que HeapSort mantuvo un desempeño más estable y predecible en todos los escenarios. Esta diferencia puede atribuirse al modo en que cada algoritmo interactúa con la jerarquía de memoria y la estructura de los datos de entrada.

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 ($\geq 10^6$ elementos) y considerar métricas adicionales como uso de caché, consumo energético o impacto en sistemas multicore.

Analisis de resultados:

Los resultados obtenidos muestran que, aunque QuickSort generalmente presenta un rendimiento superior en conjuntos de datos medianos, su comportamiento se ve afectado significativamente por la elección del pivote y la distribución de los datos. Este hallazgo está en línea con estudios anteriores, como el de Sedgewick (1978), que indican que QuickSort puede experimentar degradaciones severas en su rendimiento cuando se enfrenta a datos ya ordenados o con muchos duplicados. En contraste, HeapSort demostró una mayor estabilidad en sus tiempos de ejecución, lo que coincide con la literatura que resalta su resistencia a variaciones en la distribución de datos (LaMarca & Ladner, 1999).

La hipótesis planteada de que QuickSort podría superar a HeapSort en conjuntos medianos fue confirmada en parte; sin embargo, en escenarios de alta carga y con datos que presentan muchas duplicaciones, HeapSort mostró un rendimiento más consistente. Esto sugiere que, en aplicaciones del mundo real donde la estabilidad es crucial, HeapSort podría ser una opción más confiable, especialmente en sistemas donde los recursos son limitados y la variabilidad del rendimiento puede ser un factor crítico.

Implicaciones de los hallazgos:

Los hallazgos de esta investigación tienen varias implicaciones importantes:

Optimización de Algoritmos: La elección del algoritmo de ordenamiento debe considerar no solo la complejidad teórica, sino también el contexto de uso y la naturaleza de los datos. Esto es especialmente relevante en aplicaciones que requieren un rendimiento predecible y estable.

Desarrollo de Nuevas Estrategias: La observación de que QuickSort puede fallar en ciertos escenarios sugiere la necesidad de desarrollar estrategias híbridas que combinen lo mejor de ambos mundos, como la implementación de un QuickSort modificado que utilice HeapSort en casos donde se detecten patrones problemáticos.

Relevancia de la Reproducibilidad: La importancia de utilizar semillas reproducibles en la generación de datos, como se hizo en este estudio, subraya la necesidad de establecer estándares en la investigación de algoritmos. Esto no sólo permite comparaciones justas, sino que también facilita la validación de resultados en futuros estudios.

Direcciones para investigaciones futuras:

A medida que la tecnología avanza y la cantidad de datos generados continúa creciendo, es fundamental explorar nuevas direcciones de investigación:

- **Algoritmos Híbridos:** Investigar la efectividad de algoritmos híbridos que integren características de QuickSort y HeapSort podría proporcionar soluciones más robustas para el ordenamiento en escenarios complejos.

- **Análisis de Rendimiento en Hardware Moderno:** Realizar estudios que evalúen el rendimiento de estos algoritmos en arquitecturas de hardware modernas, como GPUs o sistemas distribuidos, podría ofrecer información valiosa sobre su aplicabilidad en entornos contemporáneos.
- **Impacto del Aprendizaje Automático:** Examinar cómo las técnicas de aprendizaje automático podrían optimizar la selección de algoritmos de ordenamiento en función de las características de los datos podría abrir nuevas vías para mejorar la eficiencia en el procesamiento de datos.

5. 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.

6 .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>

6. Anexos

Anexo 1 : datos

Iteración	Algoritmo	N	Patrón	Tiempo(ms)	Comparaciones	Intercambios	Profundidad	CommitHash
1	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
2	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
3	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
4	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
5	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
6	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
7	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
8	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
9	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
10	HipSort	10000	Aleatorio uniforme	3,0,0,0	ABC123			
11	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
12	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
13	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
14	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
15	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
16	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
17	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
18	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
19	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
20	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
21	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
22	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
23	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
24	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
25	HipSort	10000	Aleatorio uniforme	3,0,0,0	ABC123			
26	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
27	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			
28	HipSort	10000	Aleatorio uniforme	2,0,0,0	ABC123			

<https://docs.google.com/spreadsheets/d/1KWYtMc0qjY-J-9VJlx235o98zNUpXeMKWImu0bORgGY/edit?gid=714080088#gid=714080088>