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

Presentación de la investigación sobre la eficiencia de los algoritmos de ordenamiento Quickster (QuickSort) y Hipster (HeapSort).

Dany Muriel, Anette Quispe, Jair Mendoza

Resumen Ejecutivo: Quicksort vs. Heapsort

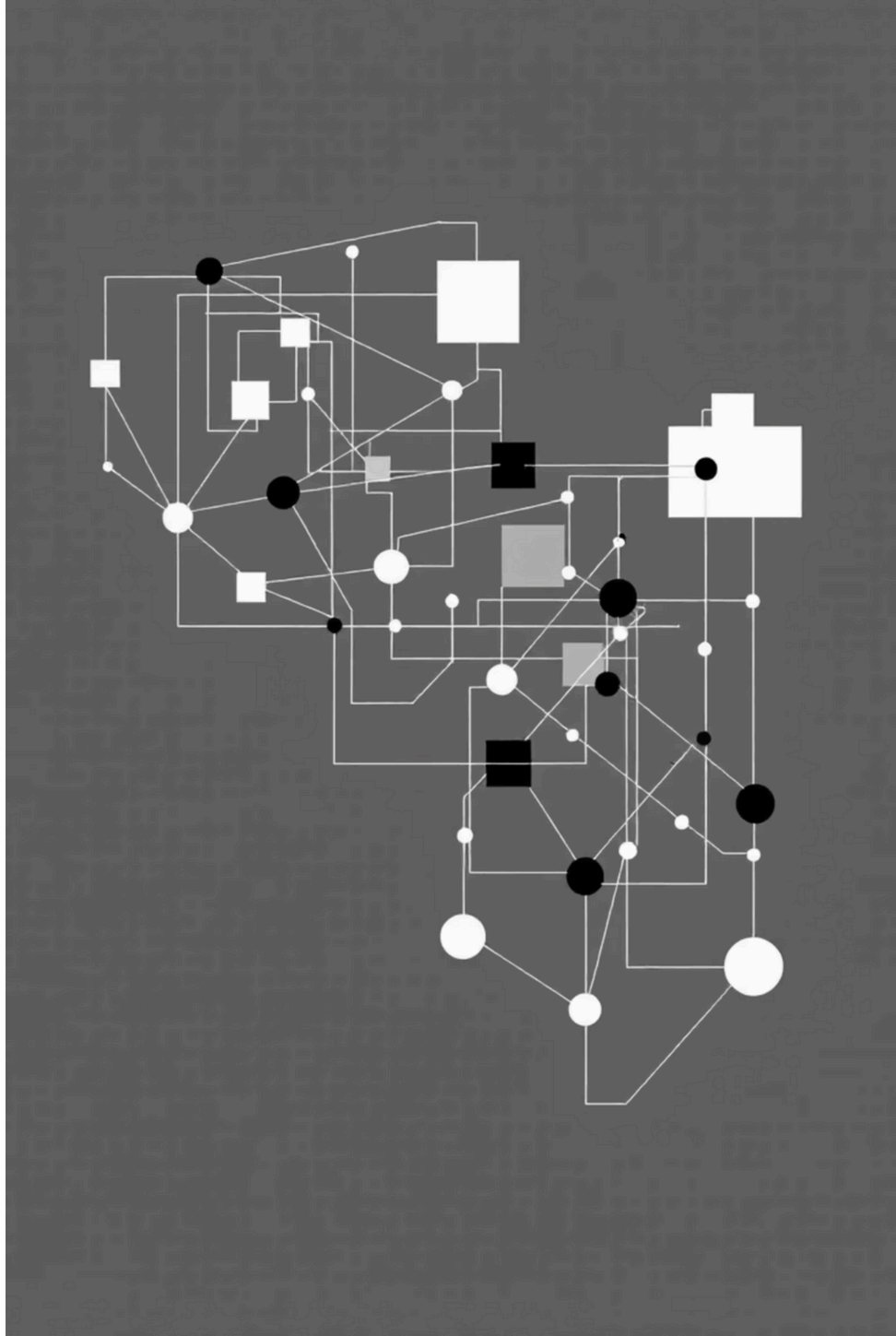
QuickSort

Utiliza un pivote central para optimizar el recorrido y evitar bloqueos. Busca la velocidad, pero su eficiencia depende de la distribución de los datos.

HeapSort

Fiel a la tradición, se basa en la robustez de métodos probados (estructura de montículo). Ofrece un rendimiento más uniforme y predecible.

El objetivo es establecer un marco analítico para evaluar cómo la velocidad, la tradición y la consistencia coexisten en el procesamiento de datos, utilizando semillas reproducibles para asegurar una base comparable.



Marco Teórico: Quick Sort y Heap Sort

Ambos algoritmos comparten una complejidad asintótica de $O(n \log n)$, pero difieren en su comportamiento en el mundo real debido a la gestión de memoria y la lógica interna.

Quick Sort (Hoare, 1962)

Rápido en promedio, pero puede degradarse severamente a $O(n^2)$ si los pivotes no se eligen cuidadosamente. Aprovecha mejor la caché.

Heap Sort (Williams, 1964)

Mantiene un rendimiento más uniforme gracias a su estructura de montículo, aunque sacrifica algo de velocidad en el acceso a memoria.

> La eficiencia no depende solo de la teoría, sino del diálogo entre el algoritmo y el hardware que lo ejecuta.

Mecanismos Algorítmicos: QuickSort y HeapSort en Acción

Comprendiendo las operaciones fundamentales y la eficiencia de estos algoritmos clave en la ordenación de datos.

QuickSort: División y Conquista

Selección de Pivote

Se elige un elemento como pivote.

Partición

Elementos menores al pivote van a la izquierda, mayores a la derecha.

Recursión

Se aplica recursivamente a las sublistas formadas.

Complejidad Temporal: Promedio $O(n \log n)$, Peor Caso $O(n^2)$.

HeapSort: Estructura de Montículo

Construcción del Montículo

El array se transforma en un montículo (heap) máximo.

Extracción del Mayor

Se extrae el elemento más grande (raíz) y se reestructura el montículo.

Repetición

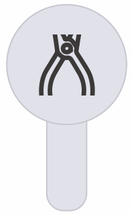
Este proceso se repite hasta vaciar el montículo.

Complejidad Temporal: Siempre $O(n \log n)$.

Parte 1: Introducción y Pasos Básicos

Demostración del algoritmo QuickSort con un arreglo de ejemplo [64, 34, 25, 12, 22, 11, 90] mostrando cada iteración del proceso de ordenamiento.

Arreglo inicial: [64, 34, 25, 12, 22, 11, 90]



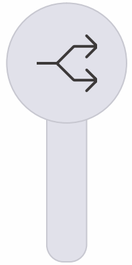
Paso 1: Primera Partición

Pivote: 64. Sublistas: [34, 25, 12, 22, 11] y [90].

Paso 2: Recursión en Sublista Izquierda

Pivote: 34 en [34, 25, 12, 22, 11]. Genera: [25, 12, 22, 11] y [].

QuickSort: Proceso de Recursión (Parte 2)



Paso 3: Particiones Sucesivas

Pivote 25 en [25, 12, 22, 11] → [12, 22, 11] y []

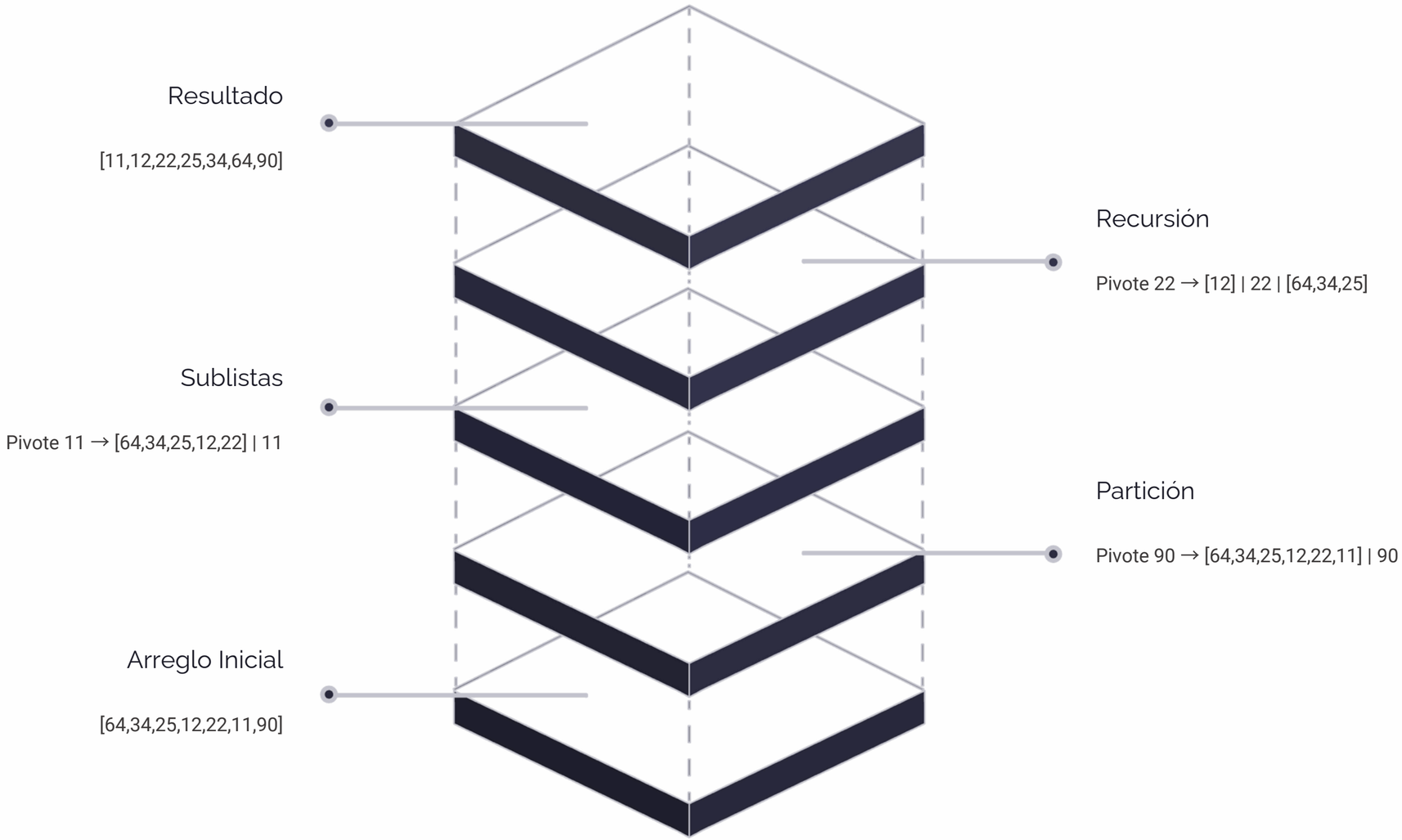
Pivote 12 en [12, 22, 11] → [11] y [22]

Paso 4: Casos Base y Unión

Las sublistas unitarias [11], [22], [90] están ordenadas y se unen

Resultado final: [11, 12, 22, 25, 34, 64, 90]

QuickSort: Visualización del Árbol de Recursión (Parte 3)



El diagrama muestra cómo se divide el arreglo inicial en sublistas más pequeñas, indicando los pivotes en cada nivel y culminando en el resultado final ordenado.

Parte 1: Introducción y Construcción del Heap

Demostración del algoritmo HeapSort con el arreglo [64, 34, 25, 12, 22, 11, 90] mostrando la construcción del heap y el proceso de extracción ordenada.

Arreglo inicial: [64, 34, 25, 12, 22, 11, 90]

Fase 1: Construcción del Max-Heap

Paso 1: Heapificar (índice 2)

25 no cambia (hijos 12, 22).

Paso 2: Heapificar (índice 1)

34 ↔ 90. Array: [64, 90, 25, 12, 22, 11, 34].

Paso 3: Heapificar raíz (índice 0)

64 ↔ 90. Array: [90, 64, 25, 12, 22, 11, 34].

HeapSort: Proceso de Extracción (Parte 2)

Fase 2: Extracción y Ordenamiento

Iteración 1: Extraer 90 → [34, 64, 25, 12, 22, 11 | 90]

Iteración 2: Extraer 64 → [34, 22, 25, 12, 11 | 64, 90]

Iteración 3: Extraer 34 → [25, 22, 11, 12 | 34, 64, 90]

Iteraciones finales: ... → [11 | 12, 22, 25, 34, 64, 90]

Resultado final: [11, 12, 22, 25, 34, 64, 90]

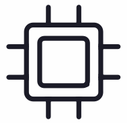
HeapSort: Visualización del Proceso (Parte 3)

Este diagrama ilustra de manera compacta el proceso completo del algoritmo HeapSort, desde la construcción del montículo máximo hasta la extracción secuencial de elementos para formar el arreglo ordenado.

La visualización destaca la eficiencia de HeapSort al transformar una estructura de datos en un arreglo completamente ordenado, mostrando la interconexión entre la fase de construcción del heap y la fase de extracción y ordenamiento.

Metodología Experimental

Se implementaron Quickster (QuickSort bidireccional) y Hipster (HeapSort) en C++17, ejecutándose en un entorno controlado para asegurar la reproducibilidad.



Entorno

Procesador Intel Core i5 @ 2.40 GHz, 8 GB RAM, Windows 10. Compilador MinGW g++ (C++17) con optimización -O2.

Medición

Tiempo de ejecución medido en milisegundos usando chrono::high_resolution_clock.

Datos

Generados sintéticamente con semilla fija (srand(42)) para garantizar que ambos algoritmos recibieran las mismas entradas.

Se analizaron tres tamaños de arreglos: Pequeño (1K), Mediano (10K) y Grande (100K) elementos.

Resultados: Conjuntos Pequeños (N=1000)

Para conjuntos pequeños, QuickSort muestra una ventaja de velocidad, pero el patrón Aleatorio Uniforme introduce la mayor inestabilidad temporal en ambos métodos.

QuickSort

Patrón de Datos	Tiempo Promedio (ms)	Desviación Estándar (ms)
Aleatorio Uniforme	0.067	0.215
Ordenado Ascendente	0.033	0.183
Ordenado Descendente	0.000	0.000
Casi Ordenado 5%	0.000	0.00
Duplicados 10-20	0.033	0.183

HeapSort

Patrón de Datos	Tiempo Promedio (ms)	Desviación Estándar (ms)
Aleatorio Uniforme	0.200	0.407
Ordenado Ascendente	0.133	0.345
Ordenado Descendente	0.100	0.305
Casi Ordenado 5%	0.133	0.345
Duplicados 10-20	0.133	0.345

QuickSort fue más rápido en general, pero HeapSort mostró tiempos más altos y distribuidos de manera más uniforme, lo que es consistente con su naturaleza estable.

Resultados: Conjuntos Medianos (N=10000)

Al aumentar el tamaño de los datos, las diferencias de rendimiento se vuelven más evidentes, confirmando la volatilidad de QuickSort frente a ciertos patrones.

QuickSort

Patrón de Datos	Tiempo Promedio (ms)	Desviación Estándar (ms)
Aleatorio Uniforme	0.700	0.150
Ordenado Ascendente	0.200	0.400
Ordenado Descendente	0.133	0.500
Casi Ordenado 5%	0.133	0.340
Duplicados 10-20	0.467	0.080

El patrón Aleatorio Uniforme es el más lento (0.700 ms) y volátil. El impacto de los duplicados es significativo (0.467 ms), acercando el rendimiento al caso cuadrático. Notablemente, el rendimiento para datos casi ordenados y ordenados descendente es muy eficiente en este contexto.

HeapSort

Patrón de Datos	Tiempo Promedio (ms)	Desviación Estándar (ms)
Aleatorio Uniforme	2.067	0.050
Ordenado Ascendente	1.700	0.466
Ordenado Descendente	1.567	0.626
Casi Ordenado 5%	1.700	0.536
Duplicados 10-20	1.800	0.040

HeapSort mantiene un comportamiento robusto y una variabilidad controlada en todos los patrones. Los tiempos promedio se concentran en un rango estrecho, confirmando su estabilidad asintótica incluso con un mayor volumen de datos.

Resultados: Conjuntos Masivos (N=100000)

El rendimiento de QuickSort se vuelve altamente volátil, mientras que HeapSort confirma su robustez, aunque con tiempos absolutos más altos.

QuickSort

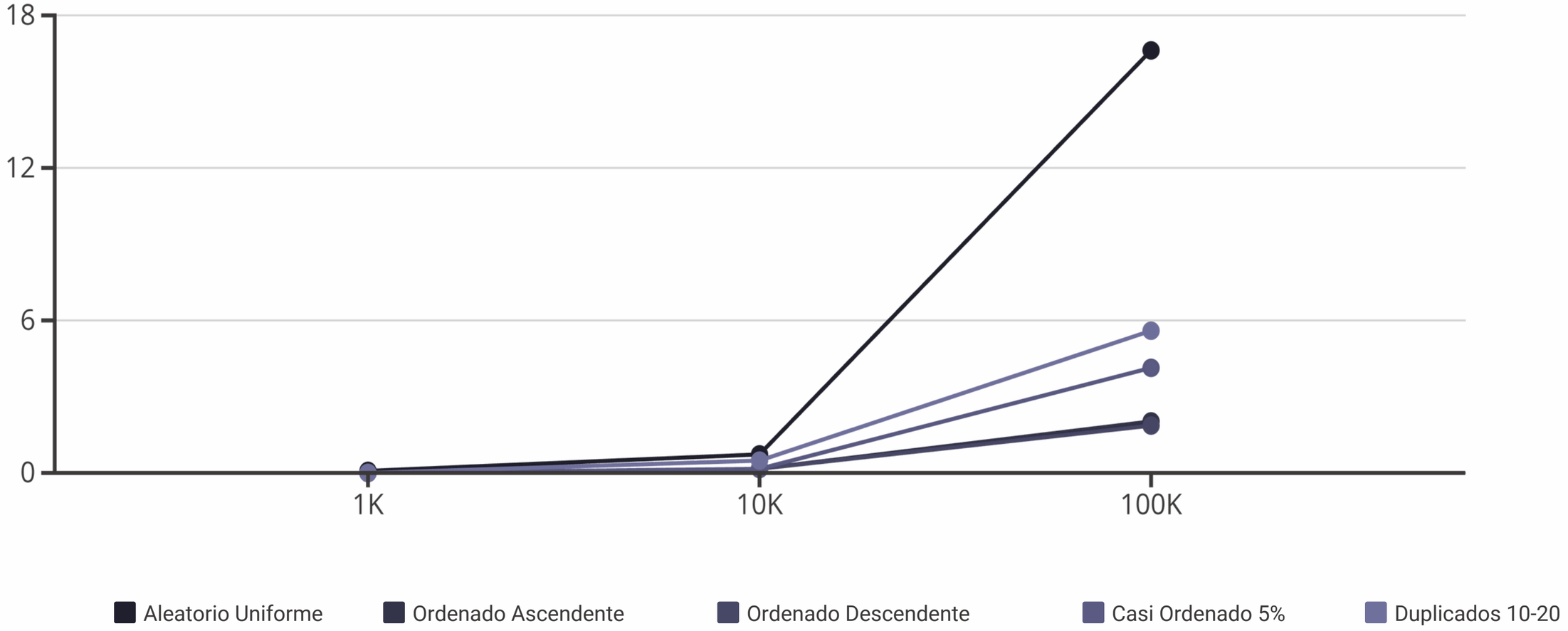
Patrón de Datos	Tiempo Promedio (ms)	Desviación Estándar (ms)
Aleatorio Uniforme	16.600	6.484
Ordenado Ascendente	2.000	2.1
Ordenado Descendente	1.833	2.0
Casi Ordenado 5%	4.133	6..602
Duplicados 10-20	5.600	6.783

HeapSort

Patrón de Datos	Tiempo Promedio (ms)	Desviación Estándar (ms)
Aleatorio Uniforme	36.000	9.098
Ordenado Ascendente	26.867	7.936
Ordenado Descendente	26.133	7.747
Casi Ordenado 5%	27.600	8.356
Duplicados 10-20	32.267	7.859

Gráfico de QuickSort: Volatilidad del Rendimiento

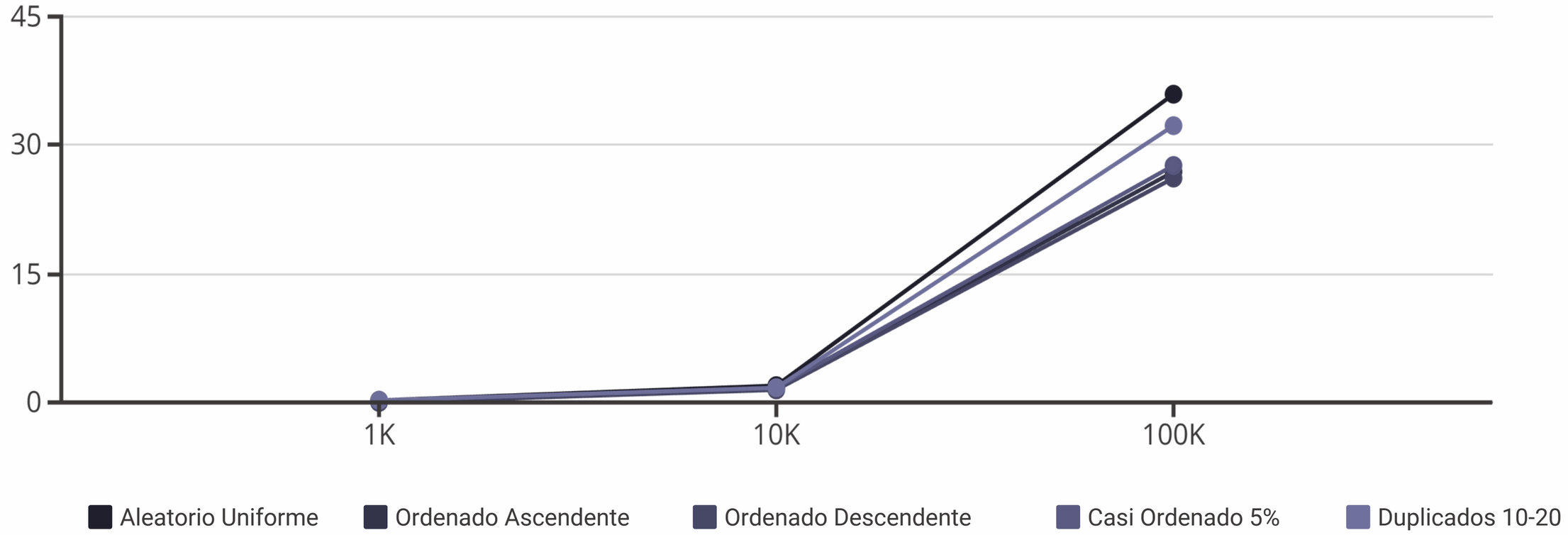
El rendimiento de QuickSort varía significativamente según el tamaño del conjunto de datos y el patrón de los mismos, destacando su eficiencia en casos ideales y su vulnerabilidad en escenarios adversos.



Se observa que el patrón **Aleatorio Uniforme** escala de manera pronunciada, mostrando la mayor inestabilidad temporal. En contraste, los datos **Ordenados Ascendente y Descendente** mantienen un rendimiento eficiente incluso en tamaños mayores, a pesar de la volatilidad en la desviación estándar.

Gráfico de HeapSort: Estabilidad del Rendimiento

El rendimiento de HeapSort muestra una notable consistencia y estabilidad a través de diferentes tamaños de conjuntos de datos y patrones, confirmando su naturaleza robusta y predecible.

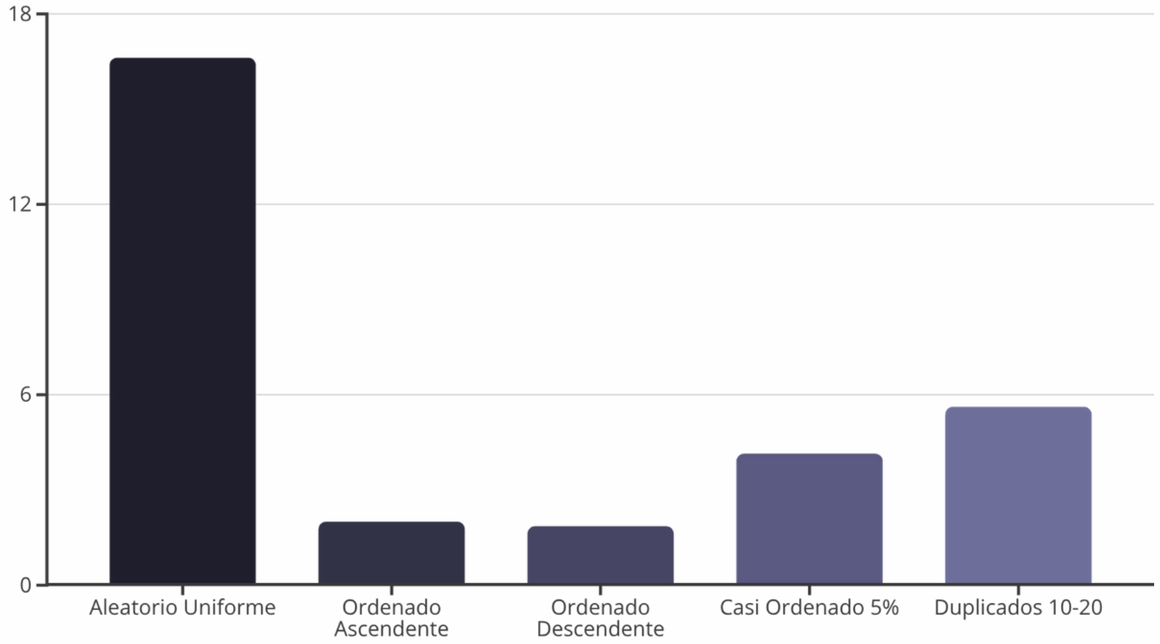


Se observa que HeapSort mantiene un crecimiento más uniforme y predecible en todos los patrones de datos. A diferencia de QuickSort, las diferencias entre los distintos tipos de datos son menores, y el algoritmo muestra una escalabilidad más consistente, confirmando su estabilidad asintótica O(n log n) en todos los casos.

Comparación Gráfica de Ambos Algoritmos (N=100000)

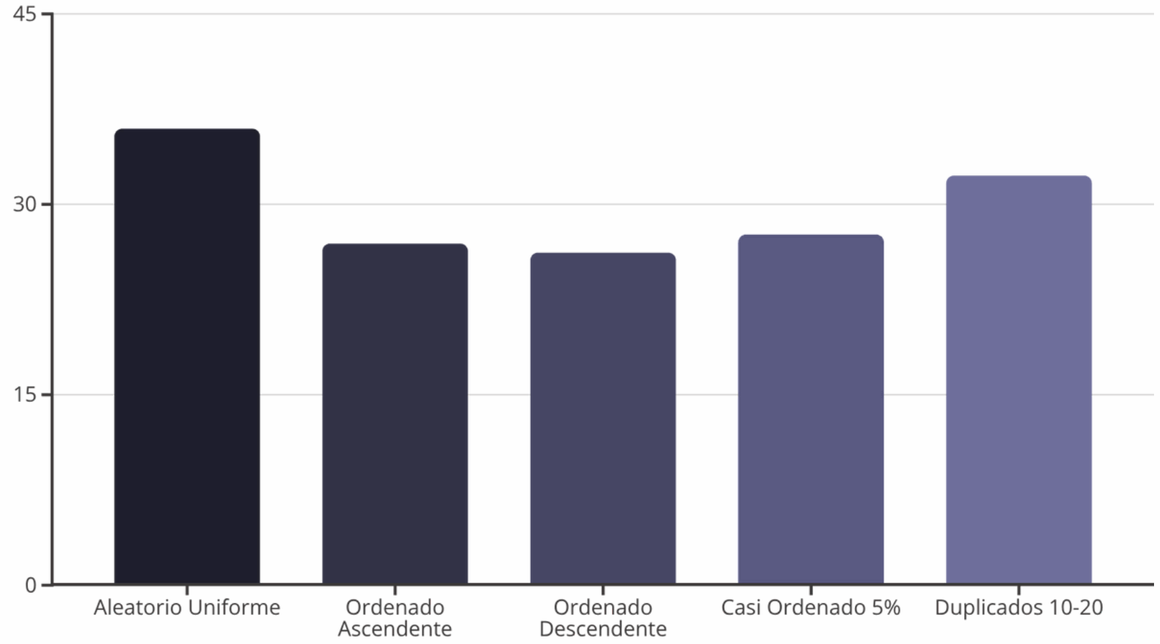
La visualización de los datos para conjuntos masivos (N=100000) pone de manifiesto las diferencias fundamentales entre QuickSort y HeapSort en diversas condiciones.

Rendimiento de QuickSort



El gráfico de QuickSort ilustra una gran variabilidad, con el peor rendimiento para datos aleatorios uniformes, donde la desviación estándar es también notablemente alta, reflejando su volatilidad. En contraste, los datos ordenados y casi ordenados muestran tiempos excepcionalmente bajos.

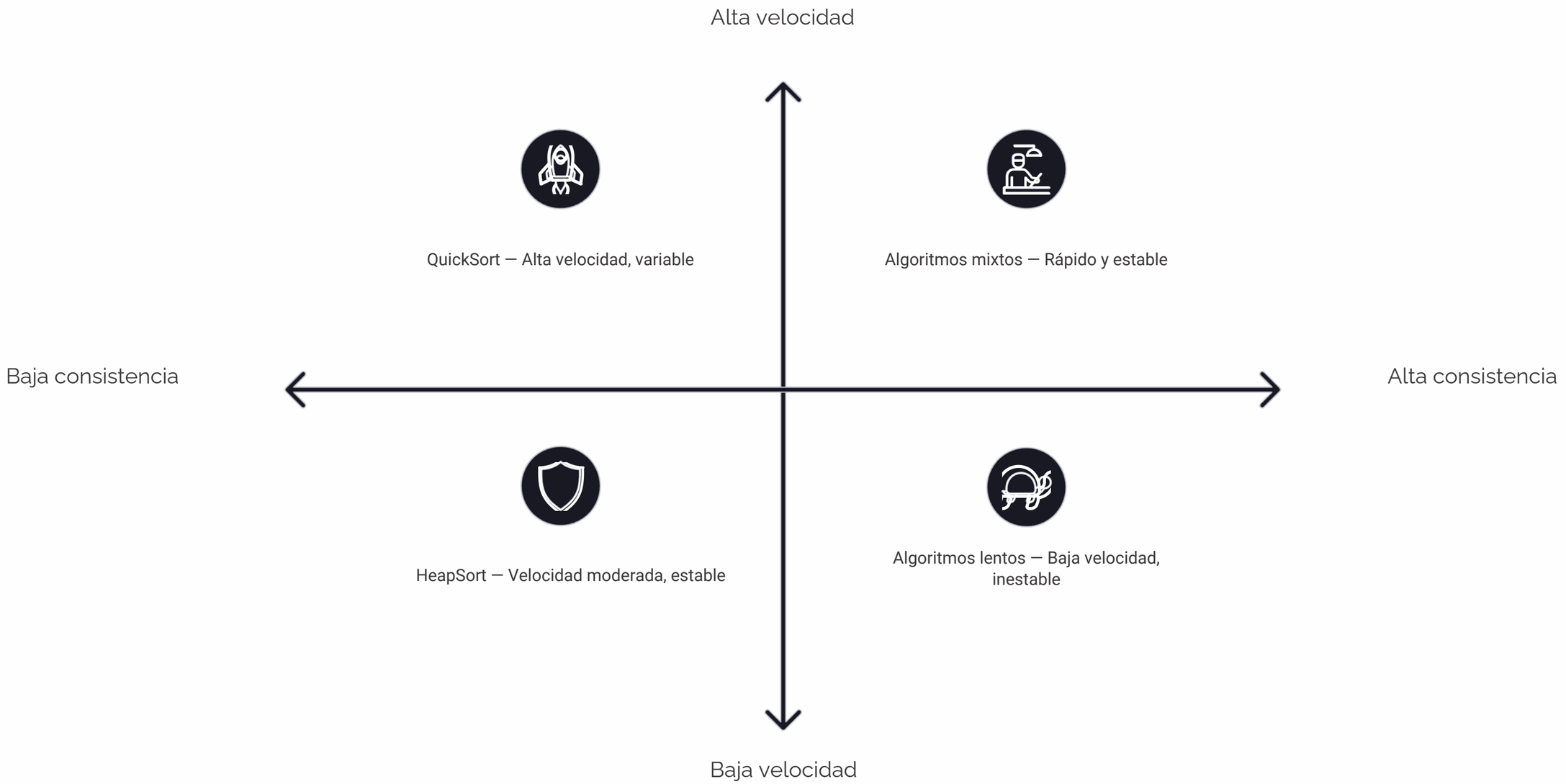
Rendimiento de HeapSort



El gráfico de HeapSort muestra una curva de rendimiento mucho más plana y consistente. Aunque sus tiempos promedio son generalmente más altos que los casos óptimos de QuickSort, la variabilidad es menor en todos los patrones, destacando su predictibilidad y robustez ante diferentes distribuciones de datos.

Discusiones: Velocidad vs. Consistencia

Los resultados confirman que QuickSort aprovecha mejor la localidad espacial (más rápido en promedio), mientras que HeapSort ofrece un rendimiento más seguro y predecible.



Implicaciones y Futuras Direcciones

Los hallazgos sugieren la necesidad de estrategias de ordenamiento más sofisticadas para el procesamiento de datos moderno.

Optimización de Algoritmos

La elección debe basarse en el contexto de uso y la naturaleza de los datos, no solo en la complejidad teórica.

Desarrollo de Estrategias Híbridas

Implementar QuickSort modificado que cambie a HeapSort cuando se detecten patrones problemáticos (datos ordenados o con duplicados).

Análisis en Hardware Moderno

Evaluar el rendimiento en arquitecturas contemporáneas (GPUs, sistemas distribuidos) para obtener información valiosa.

Impacto del Aprendizaje Automático

Utilizar técnicas de ML para optimizar la selección de algoritmos en función de las características de los datos de entrada.

Conclusiones Finales

El estudio reafirma que la eficiencia algorítmica es un diálogo complejo entre la teoría, la implementación y el hardware.

1

Fortalezas Específicas

QuickSort: Velocidad media y eficiencia en memoria. HeapSort: Mayor estabilidad y menor sensibilidad a la distribución de datos.

2

Recomendación Práctica

QuickSort para datos de rango medio y velocidad prioritaria. HeapSort para entornos que exigen predictibilidad y consistencia.

3

Base para el Futuro

Los resultados sientan las bases para investigar algoritmos híbridos o adaptativos que combinen las ventajas de ambos métodos.