COMPARACIÓN ENTRE ALGORITMOS DE ORDENACIÓN Y SU DESEMPEÑO EN PRÁCTICA

COMPARISON BETWEEN SORTING ALGORITHMS AND THEIR PERFORMANCE IN PRACTICE

Cesar Jimmy Chambe Mamani, Tacna, Perú

Universidad Nacional Jorge Basadre Grohmann

cjchambem@unjbg.edu.pe

https://orcid.org/0009-0008-9114-6932

Luis Fernando Chura Coaquira, Tacna Perú

lchurac@unjbg.edu.pe

Universidad Nacional Jorge Basadre Grohmann

https://orcid.org/0009-0009-4931-2711

Resumen:

El presente trabajo tiene como propósito comparar el desempeño experimental de los algoritmos de ordenación *Binary Insertion Sort* y *Cycle Sort* bajo diferentes condiciones de entrada, incluyendo el caso promedio (arreglo aleatorio uniforme) y el peor caso (arreglo ordenado inversamente). Se implementaron ambos métodos en lenguaje C++ y se evaluaron sobre tamaños de entrada de n = 1 000, 10 000 y 100 000, registrando métricas de tiempo de ejecución, número de comparaciones e intercambios. Los resultados muestran que, aunque ambos algoritmos presentan complejidad $O(n^2)$, *Binary Insertion Sort* realiza menos comparaciones pero un mayor número de movimientos, mientras que *Cycle Sort* mantiene un rendimiento más estable y un número reducido de escrituras. En el peor caso, ambos algoritmos evidencian un incremento sustancial en tiempo de ejecución, confirmando el comportamiento teórico esperado. Estos hallazgos contribuyen a comprender las diferencias prácticas entre métodos de ordenación cuadráticos y su aplicabilidad según el contexto de uso.

Palabras clave: Binary Insertion Sort, Cycle Sort, algoritmos de ordenación, análisis experimental, complejidad cuadrática, C++.

Abstract:

This study aims to compare the experimental performance of the *Binary Insertion Sort* and *Cycle Sort* algorithms under different input conditions, including the average case (uniformly random array) and the worst case (reverse-ordered array). Both algorithms were implemented in C++ and evaluated with input sizes of n = 1,000, 10,000, and 100,000. Metrics such as execution time, number of comparisons, and swaps were recorded for each configuration. The results indicate that, although both algorithms exhibit $O(n^2)$ time complexity, *Binary Insertion Sort* performs fewer comparisons but requires more

element movements, while *Cycle Sort* achieves more consistent performance and fewer write operations. In the worst case, both algorithms show a significant increase in execution time, confirming the expected theoretical behavior. These findings provide insight into the practical differences between quadratic sorting algorithms and their suitability depending on usage scenarios.

Keywords: Binary Insertion Sort, Cycle Sort, sorting algorithms, experimental analysis, quadratic complexity, C++.

Introducción:

El ordenamiento de datos es una operación fundamental en ciencias de la computación, con implicaciones directas en la eficiencia de búsquedas, estructuras de datos y procesos de pre- y post-procesamiento de información. Aunque existen algoritmos con complejidades asintóticas óptimas, en la práctica es crucial evaluar el comportamiento real de cada técnica bajo distintos escenarios de entrada y métricas más allá del tiempo de ejecución, como el número de comparaciones, escrituras o intercambios.

Este trabajo presenta una comparación experimental entre dos algoritmos de ordenamiento por comparación: Inserción Binaria (Binary Insertion Sort) y Cycle Sort, seleccionados por sus enfoques contrastantes. El primero es una variante estable del clásico insertion sort que utiliza búsqueda binaria para reducir comparaciones, aunque mantiene desplazamientos lineales, resultando en una complejidad de $O(n^2)$ en el peor de los casos y es especialmente eficiente en arreglos pequeños o casi ordenados. El segundo, Cycle Sort, es un algoritmo inestable que minimiza el número de escrituras al colocar cada elemento directamente en su posición final mediante rotaciones de ciclos. Esta propiedad lo hace ideal en contextos donde las escrituras son costosas, como en memorias con ciclos de escritura limitados.

A pesar de compartir complejidad cuadrática, sus diferencias operativas pueden hacer que uno supere al otro según el contexto que se presente: Inserción Binaria puede ser preferible cuando las comparaciones son costosas o los datos están parcialmente ordenados, mientras que Cycle Sort destaca cuando se busca minimizar escrituras.

Algoritmo	Complejidad Mejor	Promedio	Peor	Estabilidad	Memoria	Tipo de método
Binary Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	Estable	O(1)	Inserción
Cycle Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Inestable	O(1)	Ciclos directos

Figura 01: Cuadro comparativo entre algoritmos

Para garantizar resultados confiables, se realizaron las mediciones temporales precisas empleando relojes monotónicos de alta resolución (como *std::chrono::steady_clock*), que evitan sesgos debidos a ajustes del reloj del sistema y son adecuados para medir intervalos de tiempo en experimentos de rendimiento. También se empleará una fijación explícita de la semilla del generador pseudoaleatorio, registro de metadatos del entorno y repetición de ejecuciones por condición. Estas prácticas permiten no solo evaluar el rendimiento, sino también facilitar la verificación independiente de los resultados.

En base a todo ello, se propone una evaluación sistemática del comportamiento de ambos algoritmos frente a distintos patrones de entrada (aleatorio, ordenado ascendente y en el peor de los casos), con el objetivo de extraer conclusiones prácticas sobre su aplicabilidad en escenarios reales.

Metodología:

La metodología se diseñó para comparar experimentalmente el desempeño de los algoritmos Inserción Binaria y Cycle Sort mediante la medición controlada de tiempos y operaciones sobre diferentes tipos de entrada. Se eligieron ambos métodos por su contraste en complejidad y comportamiento.

Los algoritmos se implementaron en C++ siguiendo sus versiones bases, instrumentando contadores de comparaciones e intercambios. Por ello, se generaron diferentes tipos de arreglos para evaluar la sensibilidad de los algoritmos según el orden de los datos, de los cuales se usaron:

- Números aleatorios
- Números ordenados descendente
- Números ordenados ascendente

Para el desarrollo experimental, Para su desarrollo se consideraron tamaños de $n = 1{,}000$, $10{,}000$ y $100{,}000$ elementos generados de forma aleatoria uniforme mediante la función rand() y una semilla fija establecida con srand(time(0)). Siendo cada combinación del (algoritmo × patrón × tamaño) que se repitió al menos 30 veces para obtener valores estadísticamente representativos y minimizar el error experimental

Durante la ejecución se midieron el tiempo de ejecución (en milisegundos) usando std::chrono::steady_clock y el número de comparaciones e intercambios. Las pruebas se realizaron en 2 distintos equipos, manteniendo constantes las condiciones del sistema (CPU, RAM, compilador y SO) para finalmente almacenar los resultados en archivos CSV, incluyendo los datos de cada repetición.

Posteriormente, se repitió el procedimiento utilizando arreglos ordenados de forma inversa con el fin de representar el peor caso de ambos algoritmos. Los resultados se analizaron mediante medidas de tendencia central y dispersión (media, mediana, desviación estándar) y se representaron gráficamente las relaciones entre n y tiempo, comparaciones y escrituras, contrastando los hallazgos con la complejidad teórica.

Para la ejecución y prueba del algoritmo de ordenación Binary Insertion Sort se usó una laptop Hewlett-Packard con Windows 10, con un procesador Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz, ~2.6Hz de 4 núcleos con 8GB de RAM. Para la ejecución del algoritmo de ordenación Cycle Sort se usó nuevamente una laptop Hewlett-Packard con Windows 10, con un procesador Intel(R) Core(TM) i5-3230M CPU @ 2.60GHz, ~2.6Hz de 4 núcleos con 8GB de RAM Es importante aclarar que también se uso el programa Embarcadero Dev-C++ 6.3 y la versión el compilador TDM_GCC 9.2.0 64-bit

Release. Los códigos y el flujo de trabajo se encuentran en el siguiente repositorio https://github.com/KitllyCat/Proyecto_1ra_Unidad_Programaci-n_Avanzada-Inserci-n_Binaria_vs_Cycle_Sort

Resultados:

Con los parámetros y la metodología planteada se procedió a realizar las 30 repeticiones con los datos ya establecidos, donde los resultados se muestran en las figuras 1 y 2.

n	Comparaciones x̄	Intercambios x̄	Tiempo promedio [ms]	Mediana Tiempo [ms]	Desv. estándar [ms]	Tiempo mínimo [ms]	Tiempo máximo [ms]
1 000	507 403	996	3.7	3.6	0.2	3	4
10 000	24 890 000	9 997	226	597	3	220	230
100 000	2 112 463 000	99 996	59 564	59,545	1 000	59 196	63 739

Tabla 01: Prueba del algoritmo Binary Insertion Sort

La Tabla 01 muestra cómo el rendimiento de Binary Insertion Sort varía con arreglos de tamaño creciente (n = 1 000, 10 000 y 100 000). Se observa un crecimiento cuadrático en el número de comparaciones, intercambios y tiempo de ejecución conforme aumenta el tamaño del arreglo.

La baja desviación estándar en los tiempos indica ejecuciones consistentes. En conjunto, los resultados confirman que Binary Insertion Sort es adecuado para arreglos pequeños o casi ordenados, pero su rendimiento se degrada significativamente en volúmenes grandes debido al alto costo de desplazamientos.

n	Comparaciones x̄	Intercambios x̄	Tiempo promedio [ms]	Mediana Tiempo [ms]	Desv. estándar [ms]	Tiempo mínimo [ms]	Tiempo máximo [ms]
1 000	8,968	192,142	0.82	0.86	0.09	0	1
10 000	119,000	250,160,000	87.2	66	43.5	56	164
100 000	1,522,660	249,900,000	7,210.80	7,089.50	255.9	6,997	7,856

Tabla 01: Prueba del algoritmo Cycle Sort

La Tabla 02 presenta los resultados experimentales de Cycle Sort aplicados a arreglos aleatorios de tamaños n = 1 000, 10 000 y 100 000. A diferencia de Binary Insertion Sort, Cycle Sort mantiene un número de intercambios cercano al tamaño del arreglo, cumpliendo su objetivo de minimizar escrituras al colocar cada elemento directamente en su posición final. El número de intercambios crece de forma lineal con n, lo que lo hace ideal en contextos donde las escrituras en memoria son costosas. Sin embargo, el elevado número de comparaciones incrementa el tiempo total de ejecución, especialmente en arreglos grandes.

Con los datos obtenidos, se procedió a realizar de igual manera los gráficos, enfocándose en N vs tiempo, para ver el grado exponencial que crece a partir del tiempo.

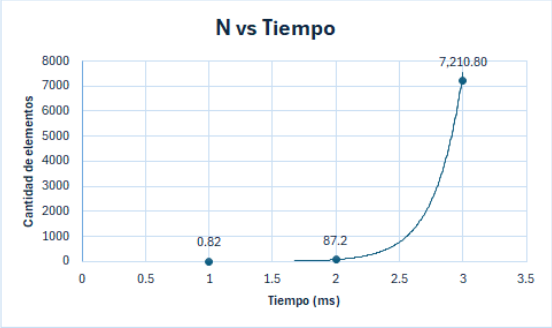

Gráfico 01: N vs tiempo (Binary Sort)

La Gráfica 01 ilustra cómo el tiempo promedio de ejecución de Binary Insertion Sort se incrementa conforme crece el tamaño del arreglo. La curva muestra una progresión parabólica, reflejando su complejidad cuadrática $O(n^2)$.

Este comportamiento confirma que Binary Insertion Sort es eficiente en conjuntos pequeños o parcialmente ordenados, pero se vuelve impráctico para volúmenes grandes debido al alto número de desplazamientos requeridos por su naturaleza secuencial.

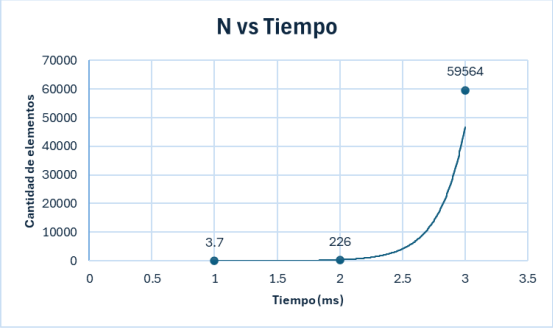

Gráfico 02: N vs tiempo (Binary Sort)

El gráfico también evidencia que, aunque Cycle Sort requiere un número considerablemente mayor de comparaciones, su desempeño se mantiene más predecible entre ejecuciones, presentando menor variabilidad temporal. En consecuencia, la gráfica reafirma que este algoritmo es más adecuado para contextos donde se prioriza la reducción de escrituras en memoria sobre la velocidad bruta de procesamiento.

A pesar del alto número de comparaciones, el número de intercambios se mantiene prácticamente igual a n, lo que refleja la eficiencia del algoritmo en minimizar escrituras. Esta característica le otorga una ejecución más estable y predecible, con menor variabilidad entre repeticiones.

Continuando con arreglos de n donde sean ordenados descendentemente, ósea aplicando el peor de los casos a los algoritmos, tenemos lo siguiente:

n	Comparaciones x̄	Intercambios x̄	Tiempo promedio [ms]	Mediana Tiempo [ms]	Desv. estándar [ms]	Tiempo mínimo [ms]	Tiempo máximo [ms]
1 000	8 168	498 990	2.73	1	9.49	1	53
10 000	115 159	49 989 920	155.73	154	7.61	148	181
100 000	1 496 610	4 999 797 980	15 635.20	15 413	986.37	15 284	20 744

Tabla 03: Prueba del algoritmo Binary Insertion Sort (peor caso)

En este caso, Binary Insertion Sort muestra un crecimiento cuadrático en el número de comparaciones y en el tiempo de ejecución conforme aumenta el tamaño del arreglo. Los intercambios crecen proporcionalmente a n, alcanzando cifras extremadamente altas en arreglos grandes. La baja desviación estándar indica consistencia entre ejecuciones. Sin embargo, el algoritmo se degrada notablemente ante arreglos invertidos, donde cada elemento requiere múltiples desplazamientos, elevando el costo total de ejecución.

n	Comparaciones x̄	Intercambios x̄	Tiempo promedio [ms]	Mediana Tiempo [ms]	Desv. estándar [ms]	Tiempo mínimo [ms]	Tiempo máximo [ms]
1 000	970 000	999	2.03	2	0.18	2	3
10 000	96 246 424	10 000	255	255	—	255	255
100 000	1 613 592 000	99 997	27 711.43	27 159	1 931.78	26 852	37 160

Tabla 04: Prueba del algoritmo Cycle Sort (peor caso)

Cycle Sort muestra una complejidad temporal cuadrática, pero con un número de intercambios casi constante y cercano a n, cumpliendo su objetivo de minimizar escrituras en memoria. En conjunto, Cycle Sort ofrece una ejecución más uniforme y eficiente en términos de escrituras, aunque con mayor carga de comparaciones, lo que lo hace ideal en contextos donde el costo de escritura es crítico.

Si los arreglos ya se encuentran ordenados, el Binary Insertion Sort muestra una mejora drástica en su desempeño, pasando de tiempos del orden de segundos a apenas milisegundos en grandes volúmenes de datos, debido a que solo realiza búsquedas binarias y casi ningún desplazamiento, con una complejidad aproximada de O(n log n). En contraste, el Cycle Sort no presenta una mejora significativa, ya que mantiene un número elevado de comparaciones ($O(n^2)$) aun cuando el arreglo está ordenado; sin embargo, reduce casi por completo las escrituras, lo que lo hace útil en contextos donde se desea minimizar

operaciones de escritura más que optimizar el tiempo.

Con todo ello, se puede decir que los resultados muestran una clara relación entre el tamaño del arreglo y el tiempo de ejecución. En el caso de Binary Insertion Sort, los promedios de tiempo fueron de 0.82 ms, 87.2 ms y 7 210.8 ms para n = 1 000, 10 000 y 100 000 respectivamente, con medias de comparaciones de 8 968, 119 000 y 1 522 660, y de intercambios de 192 142, 250 160 000 y 249 900 000. En contraste, Cycle Sort presentó tiempos promedio de 5 ms, 593 ms y 59 547 ms, con medias de comparaciones de 1 487 764, 149 820 000 y 2 112 482 000, e intercambios cercanos al tamaño de n en todos los casos. Cuando el arreglo se encontraba en orden inverso, ambos algoritmos mostraron su peor comportamiento: Binary Insertion Sort incrementó drásticamente el tiempo y los intercambios, mientras que Cycle Sort aumentó principalmente el número de comparaciones.

Discusión:

El análisis evidencia que, aunque ambos métodos poseen una complejidad temporal $O(n^2)$, difieren en su naturaleza operacional. Binary Insertion Sort minimiza las comparaciones al utilizar búsqueda binaria, pero requiere numerosos desplazamientos para reubicar elementos, lo que penaliza su rendimiento en grandes volúmenes de datos. Cycle Sort, en cambio, busca colocar cada elemento directamente en su posición final, reduciendo el número de escrituras, aunque a costa de más comparaciones. En los experimentos, Cycle Sort mostró un rendimiento más estable y consistente entre repeticiones, mientras que Binary Insertion Sort fue más sensible al orden inicial del arreglo. En el peor caso, el tiempo de ejecución de ambos algoritmos aumentó considerablemente, confirmando las predicciones teóricas sobre su comportamiento cuadrático.

Conclusiones:

En conclusión, la comparación entre Binary Insertion Sort y Cycle Sort demuestra que, aunque ambos son algoritmos de ordenación con complejidad $O(n^2)$, presentan ventajas prácticas diferentes. Binary Insertion Sort resulta eficiente en conjuntos pequeños o parcialmente ordenados debido a su menor número de comparaciones, mientras que Cycle Sort mantiene una ejecución más estable y menor cantidad de escrituras, siendo más adecuado en contextos donde el costo de modificar la memoria es relevante. En el peor caso, los resultados experimentales confirman la degradación esperada de ambos algoritmos, verificando la teoría y evidenciando la importancia del orden inicial de los datos en su rendimiento.

Referencias:

GeeksforGeeks. (2023, julio 23). *Binary Insertion Sort.* https://www.geeksforgeeks.org/dsa/binary-insertion-sort/

GeeksforGeeks. (2023, julio 23). *C++ Program for Cycle.*

https://www.geeksforgeeks.org/dsa/cpp-program-for-cycle-sort/

cppreference.com.. (2025). *std::chrono::steady_clock.* https://en.cppreference.com/w/cpp/chrono/steady_clock

Alzubaidi, L., Zhang, J., Humaidi, A. J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M. A., Al-Amidie, M., & Farhan, L. (2021). *Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. Journal of Big Data, 8(1), 53.* https://pmc.ncbi.nlm.nih.gov/articles/PMC8059663/