COMPARACIÓN ENTRE LOS ALGORITMOS DE ORDENACIÓN INSERCIÓN BINARIA Y RADIX SORT Y SU MEDICIÓN EN TÉRMINOS DE EFICIENCIA

Comparison between binary insertion sort and radix sort algorithms and their measurement in terms of efficiency

Elvis Juan Yana Cayo
ejyanac@unjbg.edu.pe
Universidad Nacional Jorge Basadre Grohmann

Josué Adrián Sosa Cruz
jasosac@unjbg.edu.pe
Universidad Nacional Jorge Basadre Grohmann

Diesdanderson Dudu Vela Laurente
dvelal@unjbg.edu.pe
Universidad Nacional Jorge Basadre Grohmann

Resumen: Se llevará a cabo una comparación entre los métodos de ordenación Binaria y Radix Sort, aplicando criterio de análisis de algoritmos en términos de eficiencia en tiempo y operaciones. La Inserción Binaria usa búsqueda binaria para ubicar elementos, reduciendo comparaciones teniendo una complejidad de $O(n^2)$. En cambio Radix Sort ordena segun los digitos, sin comparaciones directas, con una complejidad de $O(nk)$, lo que lo hace más rápido en grandes volúmenes de datos. Las pruebas en C + + con arreglos aleatorios, descendentes y ascendentes mostraron que Radix Sort tiene tiempos casi constantes y significativos que la Inserción Binaria, lo cual resulta más adecuado para grandes conjuntos de datos.

Palabras Claves: Algoritmos de ordenación, comparación C + +, Inserción Binaria, Radix Sort, rendimiento.

Abstract: *A comparison will be carried out between Binary and Radix Sort sorting methods, applying algorithm analysis criteria in terms of time and operation efficiency. Binary Insertion uses binary search to locate elements, reducing comparisons and having a complexity of O(n2). On the other hand, Radix Sort sorts according to digits, without direct comparisons, with a complexity of O(nk), making it faster on large volumes of data. Tests in C++ with random, descending and ascending arrays showed that Radix Sort has almost constant and significant times than Binary Insertion, making it more suitable for large data sets.*

Keywords: *Sorting algorithms, C++ comparison, Binary Insertion, Radix Sort, performance.*

1. Introducción

El análisis de algoritmos nos permite entender que tan eficientes son las soluciones que usamos al programar, tanto en el tiempo que tardan como en la memoria que utilizan. Entre ellos los metodos de ordenacion son muy comunes, ya que sirven para organizar datos dentro de un programa

Los métodos de ordenación son importantes para la informática puesto que nos ayuda a organizar y optimizar la búsqueda, análisis y la manipulación de información. Entre los diversos algoritmos existentes, el método de ordenación Binaria y el Radix sort tienen dos enfoques distintos, el primero se basa en comparaciones y el segundo en el procesamiento por sus dígitos.

En este trabajo se buscó comparar el rendimiento de dos métodos de ordenamiento interno: la inserción binaria y el radix sort. Para ello se midieron aspectos como el tiempo de ejecución, la cantidad de comparaciones y los intercambios realizados durante el proceso. Con la finalidad de observar cómo se relaciona la teoría con los resultados prácticos, para entender mejor el comportamiento de cada algoritmo según el tamaño y tipo de los datos que se ordenan.

2. Descripción de método

En la presente investigación se analizaron dos algoritmos de ordenamiento Radix Sort e Inserción Binaria. Ambos fueron seleccionados por caso al azar, aun así estos métodos presentan un paradigma distintos dentro de la computación: mientras que Radix Sort pertenece a los métodos no comparativos basados en el procesamiento de dígitos, Inserción Binaria corresponde a los algoritmo comparativos basados en la busque y colocación secuencial optimizada.

2.1. Inserción Binaria

El algoritmo de inserción binaria constituye una mejora del método de inserción directa (insertion sort). La diferencia principal radica en que, en lugar de recorrer secuencialmente la parte ya ordenada del arreglo para encontrar la posición de inserción del elemento actual, se aplica una búsqueda binaria (binary search) para determinar la posición correcta, reduciendo así el número de comparaciones necesarias (GeeksforGeeks, 2025).

Este enfoque mantiene una complejidad temporal promedio y en el peor caso de $O(n^2)$, aunque el uso de la búsqueda binaria disminuye el número de comparaciones respecto a la inserción simple. Es un algoritmo estable, in situ y eficiente para conjuntos pequeños o parcialmente ordenados.

2.2. Radix Sort

El Radix Sort es un algoritmo de ordenamiento no comparativo que organiza los datos según los dígitos individuales de los numero. En cada iteración, los elementos se agrupan de acuerdo con el valor de un dígito —ya sea de menor a mayor significancia (*Least Significant Digit, LCD*) o en sentido inverso (*Most Significant Digit, MSD*)—, y luego se combinan nuevamente. Este proceso se repite hasta procesar todos los dígitos (Ixbalanque, 2024)

Su complejidad temporal se aproxima a $O(nk)$ donde n representa el número de elemento y k cantidad de digitos del numero mas grande. Al no realizar comparaciones directas entre elementos, el Radix Sort suele superar a los algoritmos comparativos en grandes volúmenes de datos numéricos, siempre que el rango de valores sea limitado. Es especialmente útil en contextos donde la estabilidad y la linealidad del tiempo de ejecución son deseables.

2.3. Escenarios de entrada

Se generaron arreglos numéricos bajo tres patrones representativos:

- Aleatorio uniforme: valores enteros generados aleatoriamente dentro de una rango definido.
- Ordenado ascendente: arreglo previamente ordenado de menor a mayor.
- Ordenado descendente: arreglo ordenado de mayor a menor (peor caso de algunos algoritmos comparativos)

2.4. Tamaños de entrada y repeticiones

Se usaron 2 tamaños de entrada: 1000, 10000 elementos, ejecutándose 30 repeticiones por condición (algoritmo * patrón * tamaño), con el objetivo de reducir la variabilidad estadística.

Los tiempos de ejecución se midieron utilizando la librería <chrono> de C + +, registrando los resultados en microsegundo y excluyendo el tiempo de entradas y salida de disco.

2.5. Métricas registradas

Durante las ejecuciones experimentales se recopilaron las siguiente métricas:

1. Tiempo de ejecución(μs): medido con alta resolución.
2. Número de comparaciones realizadas por cada algoritmo.
3. Número de intercambios efectuados durante el proceso de ordenamiento.

2.6 Implementación y entorno experimental

El estudio incluyó la implementación, ejecución y evaluación experimental de ambos algoritmos mediante un programa desarrollado en C++, el cual permitió medir el rendimiento de cada método considerando las métricas en cada tamaño de entrada. Esta aproximación permitió observar empíricamente el comportamiento de cada algoritmo bajo condiciones controladas, comparando su eficiencia y estabilidad de manera objetiva.

Los experimentos se llevaron a cabo en un equipo Acer Aspire 3 con procesador Intel Core i5, de arquitectura x64, empleando un núcleo para la ejecución de las pruebas. El equipo cuenta con 8 GB de memoria RAM (ampliada desde una configuración original de 4 GB). El sistema operativo utilizado fue Windows 11 de 64 bits.

El código fuente fue compilado mediante el compilador G++ (GNU Compiler Collection) en su versión 15.1.0 (Rev5, MSYS2 project), respetando las convenciones del estándar C++17.

3. Resultados

3.1 Comparación de eficiencia temporal

Los resultados obtenidos muestran claramente diferencias significativas entre Inserción Binaria y Radix Sort, dependiendo del patrón de los datos y del tamaño de los arreglos.

3.1.1 Arreglos de 1000 elementos

Tabla 1. Resumen de la eficiencia de los algoritmos para el ordenamiento de 1000 números distribuidos aleatoriamente.

Algoritmo	n	Patrón	Tiempo medio (µs)	Mediana (µs)	Desviación estándar	Mínimo	Máximo
Inserción Binaria	1000	Aleatorio	669.3666667	622	102.5221535	577	1020
Radix sort	1000	Aleatorio	37.73333333	34	11.17242089	27	75

Radix Sort tarda 37,73 µs en promedio, pero la Inserción Binaria se demora 669,37 µs. Esto señala que Radix Sort es mucho más efectivo en términos de tiempo si los datos no siguen un orden determinado, a pesar de que se hagan modificaciones menores.

Tabla 2. Resumen de la eficiencia de los algoritmos para el ordenamiento de 1000 números Ordenados Ascendentemente.

Algoritmo	n	Patrón	Tiempo medio (µs)	Mediana (µs)	Desviación estándar	Mínimo	Máximo
Inserción Binaria	1000	Ordenado Ascendente	59.06666667	33	151.5483156	20	861
Radix sort	1000	Ordernado Ascendente	33.55555556	28	22.62118363	27	130

La inserción binaria se beneficia del orden anterior y su tiempo promedio disminuye a 59,07 µs; sin embargo, continúa siendo más elevado que el de Radix Sort (33,56 µs).

Tabla 3. Resumen de la eficiencia de los algoritmos para el ordenamiento de 1000 números Ordenados Descendentemente

Algoritmo	n	Patrón	Tiempo medio (µs)	Mediana (µs)	Desviación estándar	Mínimo	Máximo
Inserción Binaria	1000	Ordenado Descendente	1533.866667	1046.5	928.4811005	1030	5029
Radix sort	1000	Ordenado Descendente	28.96551724	27	7.575726078	27	67

Aquí se puede notar la diferencia más significativa: Radix Sort conserva un tiempo estable (28,97 µs), lo que demuestra su estabilidad ante cualquier tipo de patrón de entrada; en cambio, la Inserción Binaria llega a 1533,87 µs por la cantidad de intercambios requeridos para revertir el orden.

3.1.2 Arreglos de 10000 elementos

Tabla 4. Resumen de la eficiencia de los algoritmos para el ordenamiento de 10000 números distribuidos aleatoriamente.

Algoritmo	n	Patrón	Tiempo medio (µs)	Mediana (µs)	Desviación estándar	Mínimo	Máximo
Inserción Binaria	10000	Aleatorio	57540	55077	5824.697014	53483	77814
Radix sort	10000	Aleatorio	380.7	350	78.79793013	347	772

La diferencia es incluso mayor. Radix Sort, que necesita 380,7 µs, es más eficiente que la Inserción Binaria, que requiere 57.540 µs; esto demuestra que los algoritmos de ordenamiento no comparativos son mejores para procesar grandes cantidades de datos que los basados en comparaciones.

Tabla 5. Resumen de la eficiencia de los algoritmos para el ordenamiento de 10000 números Ordenados Ascendentemente.

Algoritmo	n	Patrón	Tiempo medio (µs)	Mediana (µs)	Desviación estándar	Mínimo	Máximo
Inserción Binaria	10000	Ordernado Ascendente	801.3	502	103	259	5983
Radix sort	10000	Ordernado Ascendente	397.3666667	352.5	11	344	1058

Tabla 6. Resumen de la eficiencia de los algoritmos para el ordenamiento de 10000 números Ordenados Descendentemente

Algoritmo	n	Patrón	Tiempo medio (µs)	Mediana (µs)	Desviación estándar	Mínimo	Máximo
Inserción Binaria	10000	Ordernad Descendente	192680.6	173160.5	52237.70747	135153	335230
Radix sort	10000	OrdernadoDescendente	704.7666667	352.5	1234.437934	343	6635

La inserción binaria tiene un comportamiento que depende mucho del orden: 801,3 µs si está en orden ascendente y 192.680,6 µs si está en orden descendente. Radix Sort exhibe un comportamiento estable, entre 397 µs y 705 µs respectivamente, lo que confirma que su comportamiento es lineal ante cambios en el patrón de entrada.

Con el objetivo de complementar la interpretación de los resultados numéricos, se presentan a continuación 3 gráficos comparativos del tiempo de ejecución de los algoritmos de Inserción Binaria y Radix Sort. En ellos se observa el comportamiento de cada método bajo diferentes patrones de entrada (aleatorio, ascendente y descendente) y tamaños de arreglo (0, 1000 y 10000 elementos). Estas representaciones permiten visualizar de manera más clara la diferencia de eficiencia entre ambos algoritmos.

Figura 1. Gráfica comparativa de algoritmos de ordenamiento cuando los elementos son de orden aleatorio (t vs n)

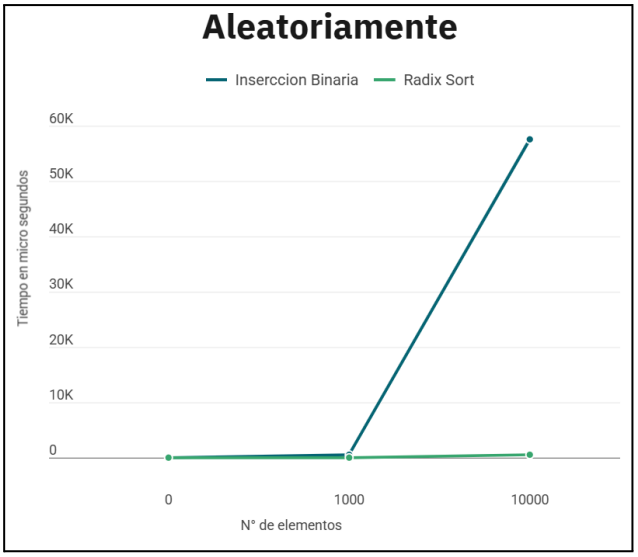


Figura 2. Gráfica comparativa de algoritmos de ordenamiento cuando los elementos son de orden ascendente (t vs n)

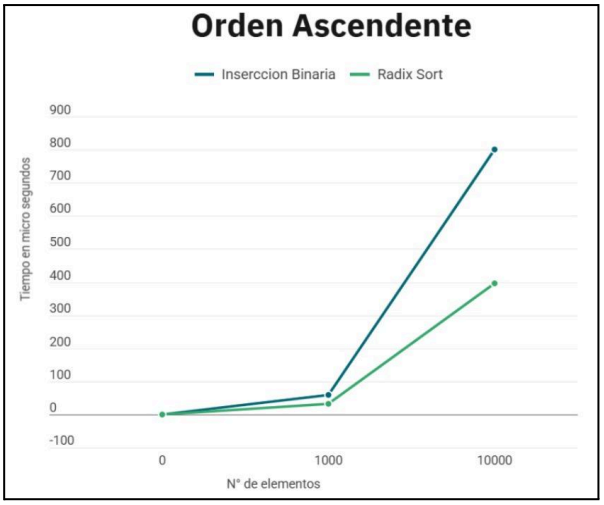
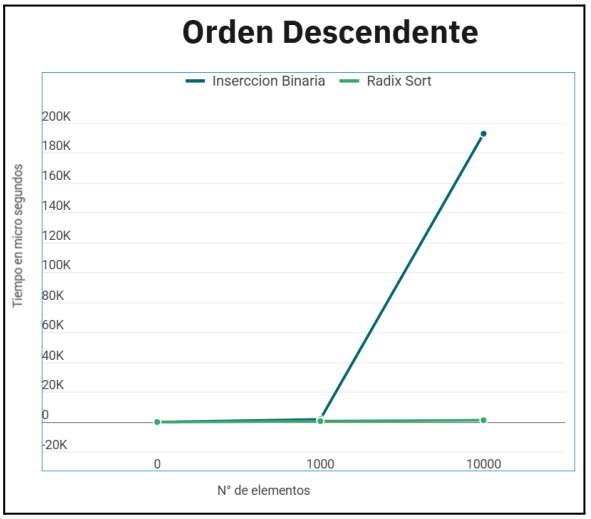


Figura 3. Gráfica comparativa de algoritmos de ordenamiento cuando los elementos son de orden descendente (t vs n)



3.2 Comparación de número de comparaciones e intercambios

Además del tiempo de ejecución, es fundamental analizar la cantidad de operaciones realizadas, ya que estas reflejan la eficiencia interna de cada algoritmo. En particular, se consideran comparaciones e intercambios, que son indicadores directos del esfuerzo computacional.

Resultados obtenidos:

Tabla 7. Comparaciones.

Algoritmo	Tamaño	Aleatorio	Ascendente	Descendente
Inserción Binaria	1000	8,571	8,977	7,987
Inserción Binaria	10000	118,938	123,617	113,631
Radix Sort	1000	5,003	5,003	5,003
Radix Sort	10000	60,004	60,004	60,004

La inserción binaria, en general, lleva a cabo más comparaciones, en particular con datos aleatorios o decrecientes, y menos cuando los datos están ordenados de manera ascendente. Radix Sort, por otro lado, mantiene una cantidad invariable de comparaciones sin tener en cuenta el tamaño del arreglo o el patrón de los datos, lo que demuestra que su eficacia es más constante en comparación con el orden original.

Tabla 8. Intercambios:

Algoritmo	Tamaño	Aleatorio	Ascendente	Descendente
Inserción Binaria	1000	251,381	999	500,499
Inserción Binaria	10000	25,058,436	9,999	50,004,999
Radix Sort	1000	8,000	8,000	8,000
Radix Sort	10000	100,000	100,000	100,000

La Inserción Binaria, respecto a los intercambios, tiene un comportamiento altamente dependiente de cómo están ordenados los datos: el mínimo se da cuando están dispuestos de manera ascendente y el máximo cuando lo están en forma descendente. Radix Sort tiene una cantidad fija de intercambios, lo que indica su estabilidad y menor dependencia con respecto al patrón de entrada, en contraste con la Inserción Binaria.

4. Discusión

Los hallazgos experimentales obtenidos validan las diferencias teóricas entre la Inserción Binaria y Radix Sort en lo que respecta a eficiencia, estabilidad y complejidad. La inserción binaria es un algoritmo comparativo cuya complejidad en el peor de los casos es $O(n^2)$. Su rendimiento es muy sensible al patrón de datos: cuando están organizados de forma ascendente, las comparaciones e intercambios son mucho menores; en cambio, con datos organizados de manera descendente, se llega al máximo, lo que muestra su comportamiento cuadrático. En cambio, Radix Sort, siendo un algoritmo no comparativo con complejidad $O(nk)$, muestra un comportamiento estable y lineal frente a cualquier patrón de entrada. Esto se manifiesta en los tiempos de ejecución y en las constantes comparaciones e intercambios de manera independiente al tamaño de los arreglos.

Respecto a la estabilidad, Radix Sort garantiza mantener el orden relativo de los elementos con igual valor, lo que resulta favorable en aplicaciones donde esto es importante, mientras que la Inserción Binaria también es estable, aunque su eficiencia se ve comprometida en casos desfavorables. En términos de limitaciones, la Inserción Binaria depende de operaciones de intercambio repetitivas, lo que aumenta el costo computacional en grandes volúmenes de datos. Radix Sort, a su vez, requiere espacio auxiliar para agrupar los dígitos, lo que podría limitar su uso en sistemas con memoria restringida. Además, en implementaciones de gran tamaño, como se planeó originalmente con arreglos de 100 000 elementos, se presentaron problemas de ejecución, probablemente relacionados con overflow o saturación de variables, lo que impidió completar las pruebas en dichos casos.

En relación con los peligros que amenazan la validez de los resultados, es importante tener en cuenta que las mediciones se llevaron a cabo en un único núcleo de un equipo particular, lo cual puede afectar los tiempos absolutos obtenidos. No obstante, al utilizar 30 repeticiones en cada condición, se disminuye la variabilidad estadística y se obtiene una tendencia fiable del comportamiento relativo de los algoritmos. Además, el hecho de seleccionar patrones representativos (aleatorio, ascendente o descendente) hace posible la evaluación de la sensibilidad de cada algoritmo ante diversos escenarios de entrada. Sin embargo, esto no abarca todos los posibles casos del mundo real, por lo que se aconseja tener precaución al generalizar los resultados.

En definitiva, los ensayos demuestran que la Inserción Binaria puede ser apropiada para volúmenes de datos reducidos o cuando se prevé que los datos estén en parte ordenados. Por otro lado, Radix Sort es sin duda más eficaz y estable para conjuntos de datos numéricos amplios, exhibiendo un comportamiento menos dependiente del patrón inicial y más predecible.

5. Conclusiones

Radix Sort exhibió una eficacia estable y superior en comparación con todos los tipos de patrones de datos (ascendentes, descendentes y aleatorios), conservando tiempos de ejecución y cantidad de operaciones casi invariables. Esto corrobora su complejidad de tipo lineal y su superioridad sobre los algoritmos comparativos cuando se trata de grandes cantidades de datos.

La inserción binaria mostró una fuerte dependencia del patrón de entrada, con un desempeño positivo cuando los datos se ordenaron de manera ascendente y un comportamiento cuadrático cuando eran descendentes o aleatorios. Esto la vuelve más apropiada para grupos pequeños o parcialmente ordenados.

Aunque se hallaron limitaciones prácticas, como la posibilidad de desbordamiento de memoria (overflow) al trabajar con tamaños más grandes, los ensayos experimentales respaldan la teoría de complejidad y estabilidad de ambos algoritmos. Los resultados, en general, señalan que Radix Sort es más eficaz y escalable. Por otro lado, la inserción binaria es ventajosa para circunstancias con menos escala y controladas.

6. Bibliografía

GeeksforGeeks. (2025, 23 julio). *Binary insertion sort*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/binary-insertion-sort/

Ixbalanque, D. (1 de abril de 2024). *¿Qué es el algoritmo Radix Sort?* Asimov Ingeniería. Recuperado de https://asimov.cloud/blog/programacion-5/que-es-el-algoritmo-radix-sort-272