**INDICE**

[**INTRODUCCIÓN** 1](#_Toc177658955)

[**OJJETIVOS** 2](#_Toc177658956)

[**Implementación de algoritmo de búsqueda secuencial** 3](#_Toc177658957)

[**Implementación de búsqueda binaria** 4](#_Toc177658958)

[**Comparación entre algoritmo de búsqueda secuencial y búsqueda binaria con diferentes tamaños de arreglos, obteniendo el tiempo de ejecución.** 5](#_Toc177658959)

[**Implementación de algoritmo de ordenamiento burbuja** 6](#_Toc177658960)

[**Implementación de algoritmo de ordenamiento por inserción** 7](#_Toc177658961)

[**Implementación del algoritmo de ordenamiento por selección** 8](#_Toc177658962)

[**Comparación entre algoritmos de ordenamiento burbuja, inserción y selección** 9](#_Toc177658963)

[**CONCLUSIONES** 10](#_Toc177658964)

# **INTRODUCCIÓN**

En el universo de la programación, la implementación y diseño de algoritmos eficientes es esencial para desarrollar aplicaciones robustas y rápidas. Los algoritmos de búsqueda y ordenamiento ocupan un lugar destacado por su amplia aplicabilidad e impacto significativo en el rendimiento de los sistemas. El lenguaje de programación Java, proporciona un entorno ideal para poner en práctica estos algoritmos.

**Algoritmos de búsqueda:**

La búsqueda es el proceso de localizar un elemento específico dentro de una estructura de datos. Existen varios algoritmos para lograrlo, con ventajas y desventajas según el caso que se esté explorando. Los dos más comunes son el de búsqueda lineal y búsqueda binaria. La búsqueda lineal es sencilla y eficaz mientras que la búsqueda binaria, requiere una lista ordenada para un mejor rendimiento en términos de complejidad temporal.

**Algoritmos de ordenamiento:**

El ordenamiento, es el proceso de organizar datos en una secuencia específica, comúnmente orden ascendente o descendente. Entre los algoritmos más conocidos se encuentran el ordenamiento por burbuja, ordenamiento rápido (quicksort) y el ordenamiento por mezcla (mergesort), además de estos están dos de los que vamos a estudiar como el ordenamiento por inserción y el ordenamiento por selección.

A través de la implementación y el análisis detallado, este documento pretende proporcionar una base para la comprensión y el uso de los algoritmos de búsqueda y ordenamiento en Java.

# **OJJETIVOS**

**OBJETIVO GENERAL**

* Implementación y análisis de algoritmos de búsqueda y ordenamiento en Java.

**OBJETIVOS ESPECIFICOS**

* Implementar algoritmos de búsqueda secuencial y búsqueda binaria.
* Implementar algoritmos de ordenamiento burbuja, inserción y selección.
* Describir en pseudocódigo la lógica de funcionamientos de los algoritmos y utilizar. la notación Big O para analizar la complejidad temporal en el mejor, caso promedio y peor caso.
* Comparar el rendimiento de los algoritmos con casos de prueba para calcular tiempos de ejecución y su variación con diferentes tamaños de datos.

# **Implementación de algoritmo de búsqueda secuencial**

Para ejemplificarlo en este documento se utiliza un array como estructura de datos que se utiliza como parámetro de entrada junto con el elemento a buscar dentro de ese arreglo.

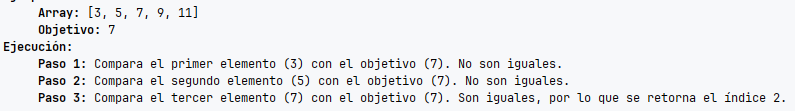
Texto

Descripción generada automáticamente

En el ejemplo anterior, se recorre el array para buscar el elemento (target) que se desea encontrar, devuelve -1 si no le encuentra y en pseudocódigo los pasos serían los siguientes:

Búsqueda secuencial:  
 Para cada elemento en el array:  
 Si el elemento es igual al objetivo:  
 Retorna el índice  
Retorna -1 si no lo encuentra

**Ejemplo de ejecución:**



La complejidad temporal de este algoritmo es la siguiente:

* Mejor caso: O(1) (si el elemento está al principio)
* Peor caso: O(n) (si el elemento está al final o no está en el array)
* Caso promedio: O(n)

En el ejemplo anterior la complejidad temporal es el caso promedio O(n) ya que tiene que recorrer el arreglo en 3 iteraciones, si el elemento a buscar es (target=3) la complejidad sería O(1) ya que está al principio y recorre el arreglo 1 vez y el peor caso es cuando el elemento a buscar no está (retorna -1) o si está en la última posición (target=11) ya que debe recorrer todo el arreglo.

**Ventajas:** Simple de implementar y funciona con arrays no ordenados.

**Desventajas:** Ineficiente para arrays grandes, ya que en el peor caso requiere revisar todos los elementos.

# **Implementación de búsqueda binaria**

Para ejemplificarlo en este documento, se muestra abajo un método Java que recibe como entrada un array de enteros y el elemento objetivo que se desea buscar.

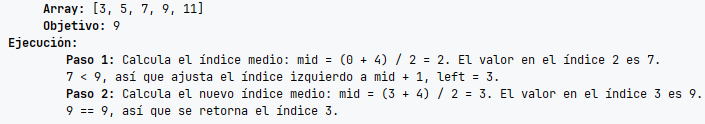
Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

En pseudocódigo se resume así:

Mientras left <= right:  
 mid = (left + right) / 2  
 Si array[mid] es igual al objetivo:  
 Retorna mid  
 Si array[mid] es menor que el objetivo:  
 Ajusta left a mid + 1  
 De lo contrario:  
 Ajusta right a mid - 1  
Retorna -1 si no se encuentra

**Ejemplo de ejecución:**



La complejidad temporal de este algoritmo es la siguiente:

* Mejor caso: O(1) (si el objetivo está en el medio)
* Peor caso: O(log n) (se divide el array en mitades)
* Caso promedio: O(log n)

Para calcular cuántos pasos como máximo se requieren, se usa logaritmo base 2 de 5:

Log2(5) ≈2.32 Esto significa que puede tomar un máximo de 3 pasos en el peor caso y 2 a 3 en el caso promedio.

**Ventajas:** Alta eficiencia, especialmente en grandes conjuntos de datos, y simplicidad.

**Desventajas:** Requiere listas ordenadas y no es adecuada para listas dinámicas o estructuras no secuenciales.

# **Comparación entre algoritmo de búsqueda secuencial y búsqueda binaria con diferentes tamaños de arreglos, obteniendo el tiempo de ejecución.**

Para el método de la búsqueda binaria se sumará el tiempo de ordenamiento (Arrays.sort() que utiliza el algoritmo de búsqueda Quicksort para tipos primitivos como int y Timsort para objetos).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Tamaño del arreglo** | **Secuencial (ns)** | **Ordenamiento (ns)** | **Binaria (ns)** | **Total, binaria (ns)** | **Mejor tiempo** | **Peor tiempo** |
| **10** | 2326200 | 2366500 | 4400 | 2370900 | SECUENCIAL | BINARIA |
| **20010** | 577800 | 12660300 | 2500 | 12662800 | SECUENCIAL | BINARIA |
| **40010** | 895900 | 8314600 | 3100 | 8317700 | SECUENCIAL | BINARIA |
| **60010** | 438500 | 8126100 | 1300 | 8127400 | SECUENCIAL | BINARIA |
| **80010** | 233700 | 7497700 | 1800 | 7499500 | SECUENCIAL | BINARIA |
| **100010** | 70600 | 8447900 | 1900 | 8449800 | SECUENCIAL | BINARIA |
| **120010** | 44400 | 10632900 | 1600 | 10634500 | SECUENCIAL | BINARIA |
| **140010** | 67700 | 8832900 | 1400 | 8834300 | SECUENCIAL | BINARIA |
| **160010** | 31400 | 9935500 | 1100 | 9936600 | SECUENCIAL | BINARIA |
| **180010** | 34900 | 11200200 | 1100 | 11201300 | SECUENCIAL | BINARIA |
| **200010** | 185500 | 13322000 | 1600 | 13323600 | SECUENCIAL | BINARIA |
| **220010** | 66000 | 13911400 | 1600 | 13913000 | SECUENCIAL | BINARIA |
| **240010** | 44400 | 15787700 | 1300 | 15789000 | SECUENCIAL | BINARIA |
| **260010** | 113300 | 16456500 | 2200 | 16458700 | SECUENCIAL | BINARIA |
| **280010** | 143900 | 16932200 | 2100 | 16934300 | SECUENCIAL | BINARIA |
| **300010** | 56400 | 17808400 | 1100 | 17809500 | SECUENCIAL | BINARIA |
| **320010** | 76100 | 19331300 | 1800 | 19333100 | SECUENCIAL | BINARIA |
| **340010** | 65200 | 21218200 | 4000 | 21222200 | SECUENCIAL | BINARIA |
| **360010** | 104300 | 21865600 | 1300 | 21866900 | SECUENCIAL | BINARIA |
| **380010** | 74800 | 23491800 | 2300 | 23494100 | SECUENCIAL | BINARIA |
| **400010** | 82100 | 27156200 | 2200 | 27158400 | SECUENCIAL | BINARIA |
| **420010** | 173000 | 26348700 | 2000 | 26350700 | SECUENCIAL | BINARIA |
| **440010** | 90500 | 27665700 | 1700 | 27667400 | SECUENCIAL | BINARIA |
| **460010** | 119600 | 29966000 | 1900 | 29967900 | SECUENCIAL | BINARIA |
| **480010** | 149800 | 29782000 | 5400 | 29787400 | SECUENCIAL | BINARIA |
| **500010** | 111900 | 31567100 | 2100 | 31569200 | SECUENCIAL | BINARIA |
| **520010** | 198400 | 31811800 | 3000 | 31814800 | SECUENCIAL | BINARIA |
| **540010** | 126100 | 33301900 | 1900 | 33303800 | SECUENCIAL | BINARIA |
| **560010** | 112200 | 34478100 | 2000 | 34480100 | SECUENCIAL | BINARIA |
| **580010** | 181700 | 36664800 | 3800 | 36668600 | SECUENCIAL | BINARIA |
| **600010** | 180800 | 37548000 | 2000 | 37550000 | SECUENCIAL | BINARIA |
| **620010** | 146600 | 38568300 | 2100 | 38570400 | SECUENCIAL | BINARIA |
| **640010** | 551500 | 39789500 | 2200 | 39791700 | SECUENCIAL | BINARIA |
| **660010** | 222000 | 43182300 | 3200 | 43185500 | SECUENCIAL | BINARIA |
| **680010** | 332200 | 43228900 | 2900 | 43231800 | SECUENCIAL | BINARIA |
| **700010** | 173500 | 44871000 | 2600 | 44873600 | SECUENCIAL | BINARIA |
| **720010** | 338400 | 46584100 | 2500 | 46586600 | SECUENCIAL | BINARIA |
| **740010** | 368800 | 46915700 | 3800 | 46919500 | SECUENCIAL | BINARIA |
| **760010** | 186500 | 48451300 | 4000 | 48455300 | SECUENCIAL | BINARIA |
| **780010** | 319100 | 50707300 | 2200 | 50709500 | SECUENCIAL | BINARIA |
| **800010** | 1063500 | 50946100 | 2600 | 50948700 | SECUENCIAL | BINARIA |
| **820010** | 391600 | 51741900 | 2500 | 51744400 | SECUENCIAL | BINARIA |
| **840010** | 651100 | 52958500 | 2300 | 52960800 | SECUENCIAL | BINARIA |
| **860010** | 431800 | 55402200 | 2300 | 55404500 | SECUENCIAL | BINARIA |
| **880010** | 220400 | 56785900 | 2700 | 56788600 | SECUENCIAL | BINARIA |
| **900010** | 335600 | 58613200 | 4500 | 58617700 | SECUENCIAL | BINARIA |
| **920010** | 421800 | 59021400 | 2100 | 59023500 | SECUENCIAL | BINARIA |
| **940010** | 1132600 | 61947600 | 5700 | 61953300 | SECUENCIAL | BINARIA |
| **960010** | 308800 | 64142100 | 2300 | 64144400 | SECUENCIAL | BINARIA |
| **980010** | 499500 | 65285500 | 2700 | 65288200 | SECUENCIAL | BINARIA |

Los tiempos de la tabla anterior están en nanosegundos, el tiempo de ejecución de la búsqueda binaria parece ser menor que el de la secuencial en la mayoría de los casos; pero sumándole el tiempo de ejecución del ordenamiento del arreglo a la búsqueda binaria el tiempo total es mayor.

# **Implementación de algoritmo de ordenamiento burbuja**

Abajo se encuentra un ejemplo con método java que recibe un array de enteros como parámetro de entrada y realiza el ordenamiento usando el algoritmo de la burbuja.

Texto

Descripción generada automáticamente

En pseudocódigo se resume así:

Repite mientras haya intercambios:  
 Para cada elemento en el array:  
 Si el elemento es mayor que el siguiente:  
 Intercambia los elementos

**Ejemplo:**

Texto, Tabla

Descripción generada automáticamente

La complejidad temporal de este algoritmo es la siguiente:

Mejor caso: O(n) (si el array ya está ordenado)

Peor caso: O(n^2) (cuando el array está en orden inverso)

Caso promedio: O(n^2)

Los pasos que tomara el ejemplo anterior se calculan así (n(n-1))/2, donde n es el tamaño del arreglo y quedaría como (5(5-1))/2 = ((5)(4))/2 = 10.

# **Implementación de algoritmo de ordenamiento por inserción**

**En el método Java siguiente se muestra su aplicación:**

Texto, Carta

Descripción generada automáticamente

En pseudocódigo se resumiría así:

Para cada elemento desde el segundo hasta el final:  
 Guarda el elemento actual  
 Mueve los elementos mayores al elemento actual a la derecha  
 Inserta el elemento en su posición correcta

**Ejemplo:**

Texto, Carta

Descripción generada automáticamente

**La complejidad temporal del algoritmo es:**

Mejor caso: O(n) (si el array ya está ordenado)

Peor caso: O(n^2) (cuando el array está en orden inverso)

Caso promedio: O(n^2)

El total de comparaciones que realiza el ejemplo anterior son las siguientes:

Primer paso: 1 comparación, segundo paso: 2 comparaciones, tercer paso: 3 comparaciones, cuarto paso: 3 comparaciones

Total = 1 + 2+ 3 + 3 = 9 comparaciones.

El algoritmo de ordenamiento por método de la burbuja y por inserción tienen el mismo orden de complejidad en el peor caso, pero el algoritmo de inserción tiende a ser más eficiente.

# **Implementación del algoritmo de ordenamiento por selección**

Ejemplo en Java:

Texto

Descripción generada automáticamente

Pseudocódigo:

Para cada posición en el array:  
 Encuentra el índice del elemento más pequeño desde la posición actual hasta el final  
 Intercambia el elemento más pequeño con el elemento en la posición actual

**Ejemplo:**

Texto, Carta

Descripción generada automáticamente

La complejidad de este algoritmo es la siguiente:

Mejor caso: O(n^2) (si el array está en cualquier orden)

Peor caso: O(n^2)

Caso promedio: O(n^2)

En este ejemplo se realizaron 10 comparaciones (n(n-1))/2 = (5(5-1))/2 = 10

La complejidad de este algoritmo es la misma para el mejor caso, caso promedio y peor caso.

# **Comparación entre algoritmos de ordenamiento burbuja, inserción y selección**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Tamaño del arreglo** | **Burbuja (ns)** | **Inserción (ns)** | **Selección (ns)** | **Mejor tiempo** | **Peor tiempo** |
| **10** | 2965600 | 6900 | 5300 | SELECCIÓN | BURBUJA |
| **3010** | 21143500 | 5805700 | 6879100 | INSERCIÓN | BURBUJA |
| **6010** | 32427400 | 9453400 | 16226600 | INSERCIÓN | BURBUJA |
| **9010** | 102228800 | 17694400 | 18182000 | INSERCIÓN | BURBUJA |
| **12010** | 185858700 | 9768900 | 65942900 | INSERCIÓN | BURBUJA |
| **15010** | 287191200 | 15159200 | 99854100 | INSERCIÓN | BURBUJA |
| **18010** | 417903600 | 23177400 | 144275200 | INSERCIÓN | BURBUJA |
| **21010** | 571089300 | 30434500 | 196704900 | INSERCIÓN | BURBUJA |
| **24010** | 763917200 | 40744500 | 266522300 | INSERCIÓN | BURBUJA |
| **27010** | 989920300 | 53047800 | 337273800 | INSERCIÓN | BURBUJA |
| **30010** | 1207671700 | 66309700 | 243619000 | INSERCIÓN | BURBUJA |
| **33010** | 1472369800 | 76827200 | 286819700 | INSERCIÓN | BURBUJA |
| **36010** | 2301917700 | 148354600 | 575186100 | INSERCIÓN | BURBUJA |
| **39010** | 2421080100 | 151605200 | 449593800 | INSERCIÓN | BURBUJA |
| **42010** | 2992010400 | 128015000 | 550384200 | INSERCIÓN | BURBUJA |
| **45010** | 3503114600 | 176437400 | 668749500 | INSERCIÓN | BURBUJA |
| **48010** | 4000492400 | 167594200 | 682054100 | INSERCIÓN | BURBUJA |
| **51010** | 4385724100 | 326678800 | 965054800 | INSERCIÓN | BURBUJA |
| **54010** | 5290578600 | 293612200 | 1034473900 | INSERCIÓN | BURBUJA |
| **57010** | 6058732100 | 363208100 | 1183165000 | INSERCIÓN | BURBUJA |
| **60010** | 7110280400 | 405478400 | 1287320600 | INSERCIÓN | BURBUJA |
| **63010** | 6098358400 | 281514100 | 1125303700 | INSERCIÓN | BURBUJA |
| **66010** | 5954439800 | 322059200 | 1182142400 | INSERCIÓN | BURBUJA |
| **69010** | 8768013100 | 514940800 | 1486832100 | INSERCIÓN | BURBUJA |
| **72010** | 7129720500 | 400839400 | 1413846400 | INSERCIÓN | BURBUJA |
| **75010** | 7773782800 | 415213700 | 1614500300 | INSERCIÓN | BURBUJA |
| **78010** | 8331135400 | 468018200 | 1698445600 | INSERCIÓN | BURBUJA |
| **81010** | 8864001800 | 494441300 | 1763405600 | INSERCIÓN | BURBUJA |
| **84010** | 12229519600 | 664536200 | 2619563600 | INSERCIÓN | BURBUJA |
| **87010** | 14347121800 | 742909100 | 2845457700 | INSERCIÓN | BURBUJA |
| **90010** | 15875050500 | 861610400 | 2984024000 | INSERCIÓN | BURBUJA |
| **93010** | 15968645800 | 890171700 | 2979650500 | INSERCIÓN | BURBUJA |
| **96010** | 16610249200 | 911692700 | 3180187200 | INSERCIÓN | BURBUJA |
| **99010** | 17906349600 | 991061500 | 3339919200 | INSERCIÓN | BURBUJA |

La tabla anterior muestra los resultados de tiempos de ejecución en nanosegundos al ordenar arreglos de diferentes tamaños y se encuentra que la mayoría tienen menor tiempo utilizando el algoritmo de ordenamiento por inserción, pero los 3 algoritmos no san tan eficientes para grandes volúmenes de datos debido a su complejidad cuadradita O(n^2) por lo que en la práctica se prefieren algoritmos más eficientes como QuickSort ó MergeSort, que tienen complejidades O(n log n).

La tabla siguiente muestra como crecería el tiempo de ejecución entre la complejidad cuadrática y logarítmica.

|  |  |  |
| --- | --- | --- |
| **Tamaño del arreglo** | **O(n^2)** | **O(log n)** |
| 10 | 100 | 3.32 |
| 100 | 10,000 | 6.64 |
| 1000 | 1,000,000 | 9.97 |

# **CONCLUSIONES**

Hemos analizado los algoritmos de búsqueda secuencial, binaria y los métodos de ordenamiento por burbuja, selección e inserción que nos revela la importancia de elegir el enfoque adecuado según el contexto y las características de los datos. La búsqueda secuencial es sencilla, se puede aplicar a diferentes colecciones y puede ser no tan eficiente para conjuntos de datos grandes. Por otro lado la búsqueda binaria es rápida, pero tiene la desventaja de que los datos deben estar previamente ordenados lo que suma tiempo si primero se ordenan los datos y luego se implementa, en el caso que analizamos concluimos que el secuencial tiene mejor tiempo debido a que la búsqueda binaria primero debe hacer un proceso de ordenamiento lo cual suma tiempo, por lo que se podría usar en los casos que sabemos que los datos están previamente ordenados para evitar el costo de ordenamiento.

En cuanto a los algoritmos de ordenamiento, cada método presenta sus propias ventajas y desventajas. El ordenamiento por burbuja es sencillo, pero poco eficiente en comparación con los algoritmos de inserción y selección, que como lo analizamos tienen mejores rendimientos sobre todo el de inserción. Todos tienen una complejidad cuadrática (el tiempo de ejecución crece cuadráticamente con el tamaño de los datos n) y en la practica son utilizados mejores algoritmos como QucikSort o MergeSort debido a que son de complejidad logarítmica (el tiempo de ejecución crece logarítmicamente con el tamaño de los datos n).

Estos algoritmos sirven en la comprensión de la eficiencia computacional, ya que permiten desarrollar habilidades críticas para abordar problemas más complejos.

El dominio de estos algoritmos es esencial para la programación y desarrollo de software, también proporciona una base para el aprendizaje de estructuras de datos y técnicas avanzadas en el ámbito de la computación. La elección del algoritmo adecuado puede significativamente hacer la diferencia entre un sistema eficiente y uno que se vuelve casi inoperante con grandes volúmenes de datos.