**Trabajo Práctico Integrador**

**Programación I**

**Algoritmos de Búsqueda y Ordenamiento en Python**

**Alumnos**

Franco Perez – francoemmanuelperez2016@gmail.com

Maximiliano Rao – rao.maximiliano.l@gmail.com

**Profesor**: Prof. Nicolás Quirós

**Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.**

Junio 2025

índice

[1- Introducción 3](#_Toc199260268)

[2- Marco Teórico 4](#_Toc199260269)

[3- Caso Práctico 5](#_Toc199260270)

[4- Metodología Utilizada 5](#_Toc199260271)

[5- Resultados Obtenidos 5](#_Toc199260272)

[6- Conclusiones 5](#_Toc199260273)

[7- Bibliografía 5](#_Toc199260274)

[8- Anexos 5](#_Toc199260275)

# Introducción

El presente trabajo se centra en el estudio de los algoritmos de búsqueda y ordenamiento, herramientas fundamentales dentro del área de la programación. Se eligió este tema debido a su importancia en la resolución de problemas cotidianos en el desarrollo de software, como la gestión eficiente de grandes volúmenes de datos, la optimización del rendimiento de aplicaciones y la mejora en la experiencia del usuario.

En programación, la búsqueda permite localizar información específica dentro de un conjunto de datos, mientras que el ordenamiento organiza dichos datos para facilitar su acceso y análisis. Estos procesos son esenciales en una amplia variedad de contextos, desde motores de búsqueda, bases de datos y sistemas de archivos, hasta algoritmos de inteligencia artificial y visualización de datos.

El objetivo de este trabajo es comprender el funcionamiento de distintos algoritmos de búsqueda y ordenamiento, analizar sus ventajas y desventajas en términos de eficiencia, uso de memoria y escalabilidad, y determinar en qué situaciones resulta más conveniente utilizar cada uno. Además, se busca aplicar estos conocimientos en la implementación de un programa práctico en Python que combine ambos tipos de algoritmos.

A través de este análisis, se pretende destacar la relevancia de seleccionar adecuadamente los métodos de búsqueda y ordenamiento en función del problema a resolver, contribuyendo así al diseño de soluciones más efectivas y robustas en el ámbito del desarrollo de software.

# Marco Teórico

Los algoritmos de búsqueda y ordenamiento son pilares fundamentales en ciencias de la computación y programación. Su correcta aplicación permite mejorar el rendimiento de los programas, optimizar recursos y garantizar la escalabilidad de las soluciones desarrolladas.

1. **Concepto de Búsqueda**

La búsqueda es el proceso mediante el cual se localiza un elemento dentro de un conjunto de datos. En programación, este conjunto suele estar representado por listas, arreglos u otras estructuras.

* **Búsqueda lineal**

La búsqueda lineal, también llamada secuencial, recorre uno a uno los elementos de la lista hasta encontrar el valor buscado o llegar al final sin hallarlo.

|  |  |
| --- | --- |
| 1  2  3  4  5 | **def** busqueda\_lineal(lista, objetivo):  **for** i **in** range(len(lista)):  **if** lista[i] == objetivo:  **return** i  **return** -1 |

**Complejidad**:

* **Peor caso: O(n),** donde n es el número de elementos en la lista. Esto ocurre cuando el elemento buscado está al final de la lista o no está presente.
* **Mejor caso: O(1),** cuando el elemento buscado es el primero de la lista.
* **Caso promedio: O(n),** porque en promedio se recorren la mitad de los elementos.
* **Búsqueda binaria**

La búsqueda binaria requiere que los datos estén ordenados. Divide la lista a la mitad en cada iteración, descartando la mitad que no contiene el objetivo.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **def** busqueda\_binaria(lista, objetivo):  izquierda, derecha = 0, len(lista) - 1  **while** izquierda <= derecha:  medio = (izquierda + derecha) // 2  **if** lista[medio] == objetivo:  **return** medio  **elif** lista[medio] < objetivo:  izquierda = medio + 1  **else**:  derecha = medio - 1  **return** -1 |

**Complejidad:**

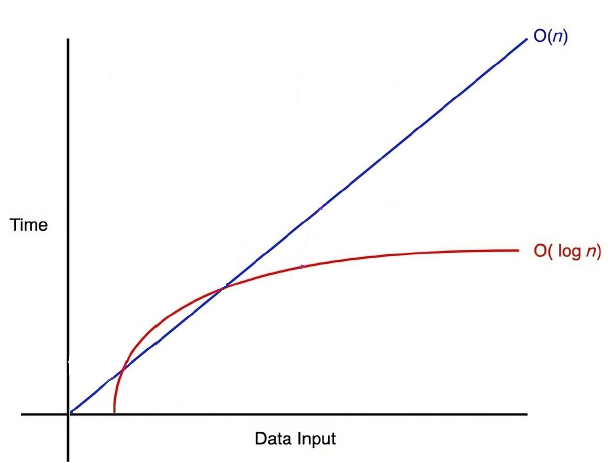
* **Peor caso: O(log n),** donde n es el número de elementos en la lista. Esto ocurre cuando el elemento no está presente o está en una de las divisiones finales.
* **Mejor caso: O(1),** cuando el elemento buscado está justo en el centro de la lista.
* **Caso promedio: O(log n),** porque el algoritmo divide la lista en mitades en cada iteración.

**Comparación entre algoritmos de búsquedas:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algoritmo | Complejidad | Ventajas | Desventajas | Uso recomendado |
| Búsqueda lineal | O(n) | * No requiere lista ordenada. * Implementación simple. | * Poco eficiente con grandes volúmenes de datos. | Listas pequeñas o desordenadas |
| Búsqueda binaria | O(log n) | * Eficiente para grandes listas ordenadas. | * Requiere datos previamente ordenados * Más compleja de implementar | Listas grandes y ordenadas |

Los algoritmos de búsqueda lineal tienen un tiempo de ejecución de O(n), lo que significa que el tiempo de búsqueda es directamente proporcional al tamaño de la lista. Esto significa que si la lista tiene el doble de elementos, el algoritmo tardará el doble de tiempo en encontrar el elemento deseado.

Los algoritmos de búsqueda binaria tienen un tiempo de ejecución de O(log n), lo que significa que el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. Esto significa que si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado.



1. **Concepto de Ordenamiento**

El ordenamiento organiza datos según un criterio (por ejemplo, de menor a mayor). Es crucial para acelerar búsquedas, detectar duplicados o visualizar información.

* **Bubble Sort (Ordenamiento por burbuja):**

Algoritmo simple que compara elementos adyacentes y los intercambia si están en orden incorrecto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **def** bubble\_sort(arr):  n = len(arr)  **for** i **in** range(n):  **for** j **in** range(0, n - i - 1):  **if** arr[j] > arr[j + 1]:  arr[j], arr[j + 1] = arr[j + 1], arr[j] |

**Complejidad:**

* + **Peor caso: O(n²),** donde n es el número de elementos en la lista. Esto ocurre cuando la lista está en orden inverso.
  + **Mejor caso: O(n),** cuando la lista ya está ordenada (con una optimización que detecta si no hubo intercambios).
  + **Caso promedio: O(n²).**
* **Quick Sort (Ordenamiento rápido):**

Divide la lista en dos sublistas, menores y mayores al pivote, y ordena recursivamente.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **def** quicksort(arr):  **if** len(arr) <= 1:  **return** arr  **else**:  pivot = arr[0]  less = [x **for** x **in** arr[1:] **if** x <= pivot]  greater = [x **for** x **in** arr[1:] **if** x > pivot]  **return** quicksort(less) + [pivot] + quicksort(greater) |

**Complejidad:**

* + **Peor caso: O(n²),** cuando el pivote seleccionado es el menor o mayor elemento en cada partición (por ejemplo, si la lista ya está ordenada).
  + **Mejor caso:O(n log n),** cuando el pivote divide la lista en dos partes aproximadamente iguales.
  + **Caso promedio: O(n log n).**
* **Selection Sort (Ordenamiento por selección):**

El algoritmo de selección funciona encontrando el elemento más pequeño en la lista no ordenada y colocándolo en la primera posición. Luego, repite este proceso para el resto de la lista, encontrando el siguiente elemento más pequeño y colocándolo en la siguiente posición, y así sucesivamente hasta que toda la lista esté ordenada. Puede ser más eficiente cuando la lista está casi ordenada, ya que realiza menos comparaciones en ese caso.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **def** selection\_sort(arr):  n = len(arr)  **for** i **in** range(n):  *# Encontrar el índice del elemento mínimo*  min\_index = i  **for** j **in** range(i+1, n):  **if** arr[j] < arr[min\_index]:  min\_index = j  *# Intercambiar el elemento mínimo con el elemento actual*  arr[i], arr[min\_index] = arr[min\_index], arr[i] |

**Complejidad:**

* **Peor caso: O(n²),** ya que siempre realiza comparaciones para encontrar el mínimo.
* **Mejor caso: O(n²),** incluso si la lista está ordenada.
* **Caso promedio: O(n²)**
* **Insertion Sort (Ordenamiento por inserción):**

Construye una lista ordenada insertando cada elemento en su posición correcta.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **def** insertion\_sort(arr):  **for** i **in** range(1, len(arr)):  key = arr[i]  j = i-1  **while** j >=0 **and** key < arr[j] :  arr[j+1] = arr[j]  j -= 1  arr[j+1] = key |

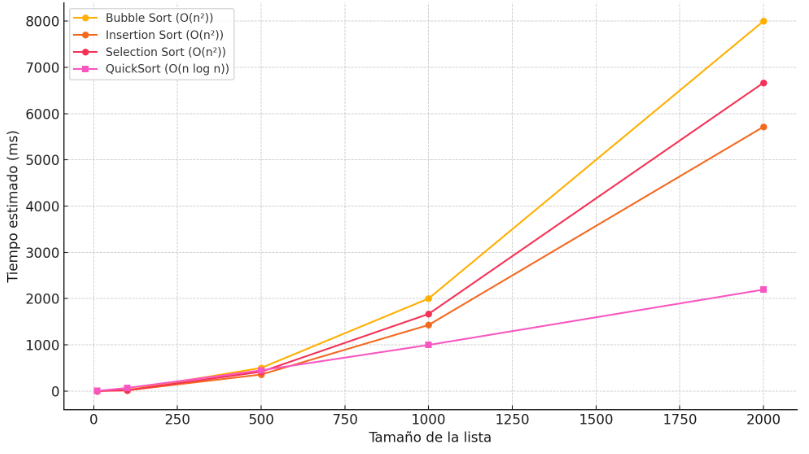
**Complejidad:**

* **Peor caso: O(n²),** cuando la lista está en orden inverso.
* **Mejor caso: O(n),** cuando la lista ya está ordenada
* **Caso promedio: O(n²).**

**Comparación entre algoritmos de ordenamiento**

|  |  |  |  |
| --- | --- | --- | --- |
| Algoritmo | Caso Promedio | Eficiencia | Uso recomendado |
| Burbuja | O(n²) | Baja | Listas pequeñas o casi ordenadas |
| Selección | O(n²) | Baja | Listas pequeñas |
| Inserción | O(n²) | Media | Listas pequeñas, casi ordenadas |
| QuickSort | O(n log n) | Alta | Listas grandes, alto rendimiento |

Gráfico comparativo de tiempo estimado de ejecución para diferentes algoritmos de ordenamiento según el tamaño de la lista:



# Caso Práctico

# Metodología Utilizada

# Resultados Obtenidos

# Conclusiones

# Bibliografía

# Anexos