Trabajo Integrador - Programación I: “Algoritmos de Búsqueda y Ordenamiento en Python”

Alumnos:
Tomás José Buforn - tomasbuforn@gmail.com
Facundo Butti - facu_butti@hotmail.com

Materia:
Programación I

Profesor/a:
Ing. Laura Fernández

Fecha de entrega:
09/06/2025

Índice

Introducción

En el mundo de la programación y la informática, la eficiencia con la que manejamos y procesamos la información es muy importante. A diario nos encontramos con cantidades enormes de datos que necesitan ser organizados para que el acceso y manipulación sobre ellos sea lo más fácil y accesible posible. Aquí es donde los algoritmos de búsqueda y ordenamiento entran como herramientas indispensables. Este tipo de código o proceso lógico nos permite encontrar elementos o datos específicos dentro de una lista/colección o acomodarlos en un orden específico y predefinido. Este proceso no solo es la base de muchísimas aplicaciones que usamos a diario, como motores de búsqueda, bases de datos o sistemas de recomendación, sino que también son esenciales para la ciencia de la computación.

Se eligió este tema por su relevancia en casi cualquier software que manipule datos. El objetivo es comprender su funcionamiento, en el marco teórico podremos sentar las bases conceptuales del tema para luego aplicarlos en un caso práctico usando Python y reflexionar sobre su uso en distintos contextos.

Marco Teórico

Dentro de los algoritmos de búsqueda y ordenamiento, la búsqueda consiste en localizar un elemento en un conjunto de datos. La eficiencia de este proceso es fundamental, especialmente en sistemas que manejan grandes volúmenes de información. A continuación vamos a describir más profundamente dos de los algoritmos de búsqueda más importantes: la búsqueda lineal y la búsqueda binaria, además analizaremos sus ventajas y desventajas y cuando es conveniente usar uno y cuando es conveniente usar otro.

Búsqueda Lineal.

Es el algoritmo mas directo, consiste en recorrer una colección de datos, elemento por elemento, desde el principio hasta el final, comparando cada elemento con el valor que se desea encontrar. Su funcionamiento consiste en iniciar su exploración en la primera posición de la estructura de datos y en cada iteración comparar el elemento actual con el valor buscado. Si este coincide retorna la posición del elemento y finaliza. En caso de no coincidir avanza al siguiente elemento, esto continúa hasta que se encuentra el valor deseado o llegar al final de la colección de datos, indicando así que el elemento buscado no está presente en la lista.

Ventajas, desventajas y recomendaciones de uso.

Dentro de las ventajas de usar este algoritmo de búsqueda podemos decir que tiene una implementación sencilla no requiere un codigo extenso o dificil de hacer, lo que la hace accesible para todos los niveles de programadores. Otra de sus ventajas es que esta funciona perfectamente sobre datos desordenados y ademas puede ser aplicada sobre cualquier estructura de datos que permita la iteracion lineal de sus elementos.
Su principal desventaja esta en su rendimiento ya que en el peor de los casos que seria que esta llegue al ultimo elemento o este no este presente el algoritmo debe examinar todos los elementos lo cual lo hace poco practico para colecciones de gran tamaño.
Por eso es que la busqueda lineal se recomienda usar para colecciones de datos pequeñas, datos desordenados, o cuando se necesita encontrar la primera instancia de un valor que esta duplicado en una lista.

Busqueda Binaria.

A diferencia de la busqueda lineal, la busqueda binaria es mucho mas eficiente, pero esta tiene una condicion fundamental: la coleccion de datos debe estar ordenada. Podriamos decir que este algoritmo funciona bajo el lema de “divide y venceras”. Si este elemento coincide con el valor buscado, la búsqueda finaliza. Si el valor buscado es menor que el elemento central, el algoritmo descarta la mitad superior de la colección y repite el proceso en la mitad inferior. Si el valor buscado es mayor, descarta la mitad inferior y continúa la búsqueda en la mitad superior. Este proceso de división sucesiva reduce el espacio de búsqueda a la mitad en cada paso, hasta que el elemento es encontrado o el espacio de búsqueda se agota.

Ventajas, desventajas y recomendaciones de uso.

Su principal virtud es la alta eficiencia para grandes conjuntos de datos, al dividir a la mitad el espacio de busqueda en cada paso este hace que sea mucho mas eficiente que la busqueda lineal. Y en cuanto a desventajas que este tipo de busqueda tiene es que su mayor limitacion esta en que la coleccion de datos debe estar rigurosamente ordenada, si esto no es asi habria que considerar si el trabajo de ordenarla anula la ventaja de la busqueda binaria. Ademas tiene una mayor complejidad de implementacion que la busqueda lineal.
Se recomienda su uso para grandes conjuntos de datos ordenados, operaciones de busquedas frecuentes como pueden ser diccionarios, y en la optimizacion de recursos.

Algoritmos de Ordenamiento

El ordenamiento es una operación fundamental que consiste en organizar los elementos de una colección (como una lista) en una secuencia específica, ya sea ascendente, descendente o alfabéticamente. La importancia de los algoritmos de ordenamiento radica en que un conjunto de datos ordenado facilita enormemente otras operaciones, como la búsqueda (tal como se vio con la búsqueda binaria) y la fusión de datos. La eficiencia de un algoritmo de ordenamiento se mide por su complejidad temporal (cuánto tiempo tarda en ordenar) y su complejidad espacial (cuánta memoria adicional requiere). A continuación, se explorarán algunos algoritmos de ordenamiento representativos que ilustran distintas estrategias y eficiencias.

Bubble Sort (Ordenamiento Burbuja)

El Bubble Sort es uno de los algoritmos de ordenamiento más simples de entender e implementar.El algoritmo funciona comparando elementos que están uno al lado del otro, y si estos están el orden incorrecto los intercambia. Este proceso se repite iterativamente a través de la lista. En cada pasada completa, el elemento más grande (o más pequeño) que aún no está en su posición final, "burbujea" hasta el final (o principio) de la parte no ordenada de la lista. El proceso continúa hasta que una pasada completa se realiza sin ningún intercambio, indicando que la lista ya está ordenada.

Ventajas, desventajas y recomendaciones de uso

Empezando con sus ventajas el bubble sort es simple conceptualmente hablando y también lo es a la hora de su implementación, este es muy fácil de comprender y escribir. Este también es estable lo que quiere decir que si dos elementos tienen el mismo valor, su orden original se conserva y por último este conlleva un bajo consumo de memoria por lo que requiere una cantidad mínima de memoria adicional más allá de la lista original. Su principal desventaja es la extrema ineficiencia para grandes conjuntos de datos ya que tiene un rendimiento muy pobre. Este tipo de ordenamiento tiene aplicaciones muy limitadas por lo que su uso real está casi limitado a propósitos educativos y listas muy pequeñas.

Merge Sort (Ordenamiento por Mezcla/Fusión)

El Merge Sort es un algoritmo de ordenamiento eficiente y estable, es considerado uno de los algoritmos de ordenamiento más robustos.
Este algoritmo se divide en dos fases principales:

- División: La lista no ordenada se divide en dos mitades, hasta que cada sublista contiene un solo elemento. Una lista de un solo elemento se considera, por definición, ordenada.
- Fusión: Las sublistas se fusionan de forma ordenada. En cada paso de la fusión, dos sublistas ya ordenadas se combinan para producir una sublista más grande también ordenada. Este proceso de fusión es muy importante y se realiza comparando los primeros elementos de cada sublista y colocando el menor en la nueva lista, repitiendo hasta que todos los elementos de ambas sublistas se han incorporado.

Ventajas, desventajas y recomendaciones de uso

En sus ventajas nos encontramos con su alta eficiencia, es significativamente mas rapida que algoritmos como Bubble Sort para colecciones de datos grandes, ademas esta tambien es estable. Su lado negativo es que tiene un mayor consumo de memoria ya que para su proceso de fusion requiere un espacio adiciona significativo. Su uso esta recomendado para grandes conjuntos de datos, ordenamiento de archivos grandes, algoritmos basados en fusion y cuando la estabilidad es importante.

Insertion Sort (Ordenamiento por Inserción)

El Insertion Sort es un algoritmo de ordenamiento simple que construye la lista final ordenada elemento por elemento, tomando los elementos de la lista de entrada y reubicándolos en la posición correcta dentro de la sublista ya ordenada.
El algoritmo divide la lista en dos partes: una sublista ordenada (inicialmente, el primer elemento de la lista) y una sublista no ordenada. En cada iteración, toma el primer elemento de la sublista no ordenada y lo inserta en su posición correcta dentro de la sublista ordenada. Para encontrar esta posición, se compara el elemento a insertar con los elementos de la sublista ordenada, moviendo los elementos mayores (o menores, según el orden deseado) una posición hacia adelante para hacer espacio. Este proceso se repite hasta que todos los elementos de la lista original han sido insertados en la sublista ordenada.

Ventajas, desventajas y recomendaciones de uso

El insertion sort es simple y facil de implementar, es eficiente para listas pequeñas o casi ordenadas, es estable, requiere poca memoria adicional y ademas puede ordenar a medida que se reciben nuevos elementos. Como en todos los casos tambien hay algunas desventajas como que es ineficiente para grandes conjuntos de datos desordenados y necesita multiple desplazamientos de elementos para hacer espacio para la insercion, lo que puede ser costoso.

Se recomienda su uso para lista muy pequeñas o que esten casi ordenadas, tambien para propositos educativos y como parte de otros algoritmos.

Quick Sort (Ordenamiento Rápido)

Quick Sort es uno de los algoritmos de ordenamiento más eficientes y ampliamente utilizados en la práctica. También se basa en la division, pero lo hace de una manera diferente a Merge Sort, fragmentando la lista en lugar de simplemente dividirla.
El algoritmo Quick Sort funciona seleccionando un elemento de la lista, llamado **pivote**. Luego, fragmenta la lista en dos sublistas: una que contiene todos los elementos menores que el pivote y otra con todos los elementos mayores que el pivote. Los elementos iguales al pivote pueden ir a cualquiera de las dos sublistas o formar una tercera. Una vez fragmentada, el algoritmo se aplica repetitivamente a cada una de las sublistas hasta que la lista completa está ordenada. La elección del pivote es crucial para el rendimiento.

Ventajas, desventajas y recomendaciones de uso

Estamos hablando de un ordenamiento extremadamente eficiente en promedio, lo que lo convierte en uno de los algoritmos mas rapido para grandes volumenes de datos, tambien a diferencia de Merge Sort y Quick sort requiere una cantidad minima de memoria adicional. Dentro de sus desventajas se encuentra que si la eleccion del pivote no es buena el programa puede llegar a tener un mal rendimiento, este tambien no es estable y es mas complejo de implementar. Se recomienda usar este tipo de ordenamiento cuando hay grandes conjuntos de datos, cuando la eficiencia es crucial y para librerias de ordenamiento en general.

Selection Sort (Ordenamiento por Selección)

El Selection Sort es un algoritmo de ordenamiento que construye la lista ordenada seleccionando repetidamente el elemento más pequeño (o más grande) de la porción no ordenada de la lista y colocándolo al principio (o al final) de la porción ordenada.
El algoritmo funciona de la siguiente manera:

1. **Encuentra el Mínimo:** En la primera pasada, busca el elemento más pequeño en toda la lista no ordenada.
2. **Intercambia:** Una vez encontrado, lo intercambia con el elemento que se encuentra en la primera posición de la lista no ordenada (es decir, el primer elemento de la lista completa si es la primera pasada). Esto coloca el elemento más pequeño en su posición final ordenada.
3. **Repite:** Luego, el algoritmo considera la sublista restante (excluyendo el elemento ya ordenado) y repite el proceso: busca el elemento más pequeño en esta sublista y lo intercambia con el primer elemento de esta sublista.

Este proceso se continúa hasta que todos los elementos están en su posición final ordenada. En cada iteración, un elemento se coloca en su posición definitiva.

Ventajas, desventajas y recomendaciones de uso

Este ordenamiento es de facil implementacion, tiene un minimo numero de intercambios esto puede ser una ventaja en situaciones donde las operaciones de intercambio de datos son costosas y ademas no requiere memoria adicional significativa más allá de la lista original.
Su lado negativo es su ineficiencia para grandes conjuntos de datos, su rendimiento es consistente pero lento y no es estable. Se recomienda su uso para propositos educativos, listas extremadamente pequeñas y cuando el costo de intercambio es muy elevado.

Caso Práctico
Creamos un programa en Python que permite al usuario ingresar una lista de números, ordenarla con Bubble Sort/Ordenamiento de burbuja o Selection Sort/Ordenamiento de selección, y buscar un número usando búsqueda lineal o binaria.

Código fuente:

```python
def bubble_sort(arr):
    for i in range(len(arr)):
        for j in range(0, len(arr) - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

def selection_sort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i + 1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

def linear_search(arr, target):
    for i, val in enumerate(arr):
        if val == target:
            return i
    return -1

def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
```

```
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1

# Inicio del programa
print("---- Algoritmos de Búsqueda y Ordenamiento ----")

# Ingreso de números
entrada = input("Ingrese números separados por comas: ")
datos = [int(x.strip()) for x in entrada.split(",")]
print("Lista ingresada:", datos)

# Elección del método de ordenamiento
print("Métodos de ordenamiento disponibles:")
print("1. Bubble Sort")
print("2. Selection Sort")
op_orden = input("Seleccione una opción (1 o 2): ")

if op_orden == "1":
    datos_ordenados = bubble_sort(datos.copy())
    print("Lista ordenada con Bubble Sort:", datos_ordenados)
elif op_orden == "2":
    datos_ordenados = selection_sort(datos.copy())
    print("Lista ordenada con Selection Sort:", datos_ordenados)
else:
    print("Opción inválida. Se ordenará con Bubble Sort por defecto.")
    datos_ordenados = bubble_sort(datos.copy())

# Ingreso del número a buscar
buscar = int(input("Ingrese el número que desea buscar: "))

# Elección del método de búsqueda
print("Métodos de búsqueda disponibles:")
print("1. Búsqueda Lineal")
print("2. Búsqueda Binaria")
op_busqueda = input("Seleccione una opción (1 o 2): ")
```

```
if op_busqueda == "1":
    resultado = linear_search(datos_ordenados, buscar)
    metodo = "Búsqueda Lineal"
elif op_busqueda == "2":
    resultado = binary_search(datos_ordenados, buscar)
    metodo = "Búsqueda Binaria"
else:
    print("Opción inválida. Se usará Búsqueda Lineal por defecto.")
    resultado = linear_search(datos_ordenados, buscar)
    metodo = "Búsqueda Lineal"

# Resultado búsqueda
if resultado != -1:
    print(f"[{metodo}] El número {buscar} se encuentra en la posición
{resultado}.")
else:
    print(f"[{metodo}] El número {buscar} no fue encontrado en la lista.")
```

Metodología Utilizada

- Búsqueda y análisis de bibliografía confiable.
- Desarrollo progresivo de cada algoritmo.
- Pruebas con distintos datos para validar resultados.
- Publicación en GitHub.
- Elaboración del video explicativo.

Resultados Obtenidos

- Los algoritmos de ordenamiento funcionaron correctamente.
- Se observó que la búsqueda binaria fue más rápida.
- Se entendieron las ventajas y limitaciones de cada enfoque.
- El código se validó con pruebas sencillas.

Conclusiones

Este trabajo permitió profundizar en conceptos básicos y esenciales de los algoritmos de búsqueda y ordenamiento. Se comprobó la eficiencia de cada método según el contexto, y se valoró la importancia de ordenar antes de aplicar búsqueda binaria. Además, se reforzaron habilidades en Python, diseño de algoritmos y trabajo colaborativo.

Bibliografía

- Python Software Foundation. (2024). Documentación oficial de Python
- Documentos subidos al campus de la materia por el profesor.

Anexos

Capturas del programa funcionando.
Link al repositorio GitHub
Link al video explicativo en YouTube