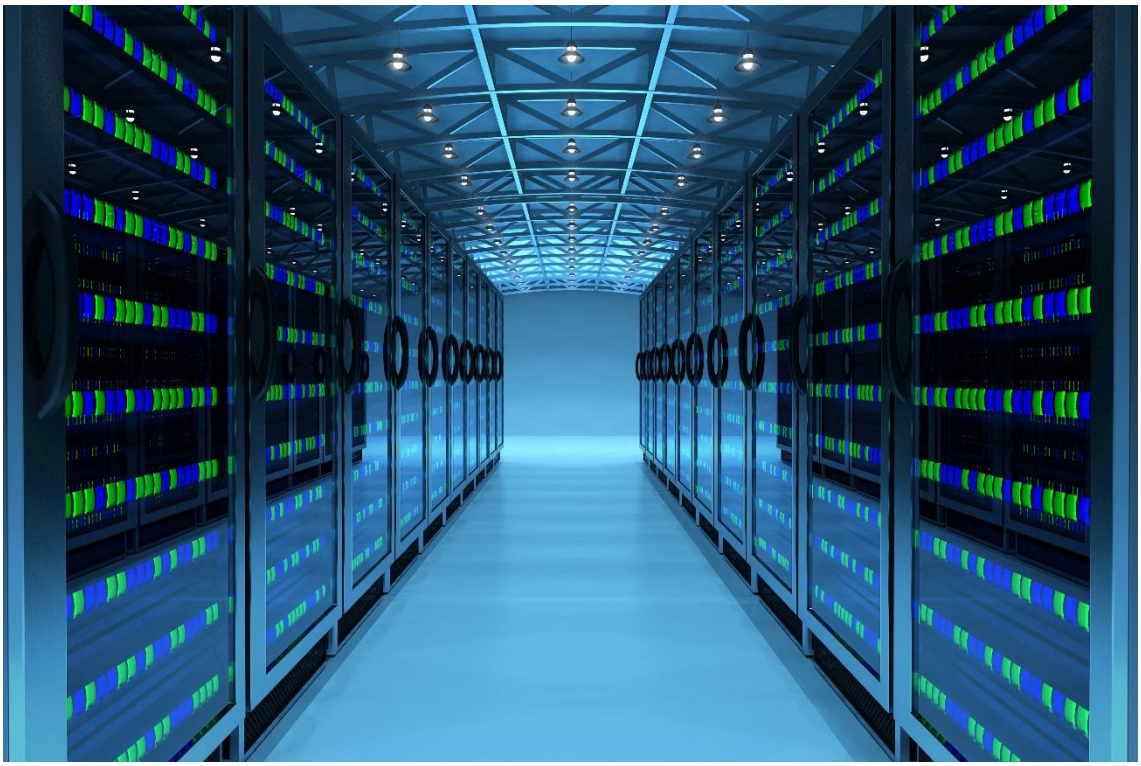


Algoritmos de Búsqueda y Ordenamiento en Python

Trabajo practico Integrador
Programación I

Alumnos: Gabriel Alejandro Ledesma - gledesma10@gmail.com

Tomas Lucas - tomasemmanuellucas96@gmail.com

Materia: Programación 1

Profesora: Cinthia Rigoni

Fecha de Entrega: 09/06/2025

Índice

Introducción

El tema de búsqueda y ordenamiento de algoritmos en Python fue elegido debido a su relevancia fundamental en el campo de la programación. Estos algoritmos forman parte esencial de múltiples procesos computacionales, desde la organización eficiente de datos hasta la mejora del rendimiento en aplicaciones complejas. Python, por su sintaxis clara y su amplia adopción en entornos educativos y profesionales, se presenta como un lenguaje ideal para abordar y comprender estos conceptos.

La importancia de los algoritmos de búsqueda y ordenamiento radica en que son herramientas básicas para la manipulación y estructuración de datos. Su correcto uso permite optimizar tiempos de ejecución y recursos, lo cual es crucial en el desarrollo de software eficiente.

Este trabajo se propone como objetivo principal analizar y comprender el funcionamiento de los algoritmos más representativos en estas categorías, como la búsqueda lineal y binaria, así como los métodos de ordenamiento como burbuja, inserción, selección y Quicksort. A través de ejemplos prácticos en Python, se busca facilitar la comprensión de su lógica interna y evaluar sus ventajas y limitaciones. De este modo, se pretende fortalecer el pensamiento algorítmico y la capacidad de aplicar soluciones eficientes a problemas comunes en programación.

Marco Teórico

¿Qué son los Algoritmos de Búsqueda? 📊🔍

Los algoritmos de búsqueda son métodos que nos permiten encontrar la ubicación de un elemento específico dentro de una lista de elementos. Dependiendo de la lista necesitarás utilizar un algoritmo u otro, por ejemplo, si la lista tiene elementos ordenados, puedes usar un algoritmo de búsqueda binaria, pero si la lista contiene los elementos de forma desordenada este algoritmo no te servirá, para buscar un elemento en una lista desordenada deberás utilizar un algoritmo de búsqueda lineal.

Búsqueda Lineal

Los algoritmos de búsqueda lineal, también conocidos como búsqueda secuencial, implican recorrer una lista de elementos uno por uno hasta encontrar un elemento específico. Este algoritmo es muy sencillo de implementar en código, pero puede

ser muy ineficiente dependiendo del largo de la lista y la ubicación donde está el elemento. A continuación, veremos un pequeño ejemplo de código en Python.

```python

def linear_search(lista, objetivo):

    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i

    return -1


lista = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
numero_objetivo = 39
resultado = linear_search(lista, numero_objetivo)

if resultado != -1:
    print(f"El número {numero_objetivo} se encuentra en la posición: {resultado}")
else:
    print(f"El número {numero_objetivo} NO se encuentra en la lista.")
```

En este ejemplo de código, necesitamos buscar el número 39, para buscarlo de forma lineal simplemente recorremos la lista con la ayuda de una estructura de bucle *for* y luego preguntamos si el elemento actual es igual a el elemento que estamos buscando, de ser así retornamos el índice del elemento y terminamos el bucle, pero si el bucle termina y no retorno ningún elemento significa que el número que buscamos no se encuentra en la lista por lo que retornamos *-1*. Este algoritmo puede ser útil para recorrer listas pequeñas o listas desordenadas, pero no es eficiente para recorrer listas demasiado largas.

Búsqueda Binaria

El algoritmo de búsqueda binaria es un algoritmo muy eficiente que se aplica solo a listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento objetivo con el elemento del medio, esto reduce significativamente la cantidad de comparaciones necesarias.

A continuación, veremos un pequeño ejemplo de búsqueda binaria con Python.

```python
def binary_search(lista, objetivo, inicio, fin ):
    if inicio > fin:
        return -1

    centro = (inicio + fin) // 2
    if lista[centro] == objetivo:
        return centro
    elif lista[centro] < objetivo:
        return binary_search(lista, objetivo, centro + 1, fin)
    else:
        return binary_search(lista, objetivo, inicio, centro - 1)


# Ejemplo de uso
lista = [1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 15, 20, 27, 34, 39, 50]
numero_objetivo = 27
inicio_busqueda = 0
fin_busqueda = len(lista) - 1

resultado = binary_search(lista, numero_objetivo, inicio_busqueda, fin_busqueda)

if resultado != -1:
    print(f"El número {numero_objetivo} se encuentra en la posición {resultado}.")
else:
    print(f"El número {numero_objetivo} NO se encuentra en la lista.")
```

En este ejemplo, hacemos uso de un algoritmo de búsqueda binario para encontrar el número 27 en una lista de elementos ordenados, para poder encontrar el elemento que buscamos podemos hacer uso de una función recursiva, en esta función el caso base sería si el número de la lista en la posición *centro* es igual al número que buscamos, de ser así retornamos el valor de la variable *centro* este sería el índice del número, de lo contrario, dividimos la lista en dos mitades y hacemos el llamado recursivo hasta encontrar el número que buscamos pero si el número no se encuentra en la lista retornamos *-1*.

¿Qué es el ordenamiento?

El ordenamiento organiza los datos de acuerdo con un criterio, como de menor a mayor o alfabéticamente.

Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla. Algunos de los beneficios de utilizar algoritmos de ordenamiento incluyen:

● Búsqueda más eficiente: Una vez que los datos están ordenados, es mucho más fácil buscar un elemento específico. Esto se debe a que se puede utilizar la búsqueda binaria, que es un algoritmo de búsqueda mucho más eficiente que la búsqueda lineal.

● Análisis de datos más fácil: Los datos ordenados pueden ser analizados más fácilmente para identificar patrones y tendencias. Por ejemplo, si se tienen datos sobre las ventas de una empresa, se pueden ordenar por producto, región o fecha para ver qué productos se venden mejor, en qué regiones se venden más productos o cómo cambian las ventas con el tiempo.

● Operaciones más rápidas: Muchas operaciones, como fusionar dos conjuntos de datos o eliminar elementos duplicados, se pueden realizar de manera más rápida y eficiente en datos ordenados.

Existen muchos algoritmos de ordenamiento diferentes, cada uno con sus propias ventajas y desventajas. Algunos de los algoritmos de ordenamiento más comunes incluyen:

1. Bubble Sort (Ordenamiento por burbuja):

○ Compara elementos adyacentes y los intercambia si están en el orden incorrecto.

○ Es fácil de entender, pero no muy eficiente para listas grandes.

2. Quick Sort (Ordenamiento rápido):

○ Divide y conquista: selecciona un “pivote” y organiza los elementos menores a un lado y los mayores al otro.

○ Es mucho más rápido que el Bubble Sort en la mayoría de los casos.

3. Selection Sort (Ordenamiento por selección):

○ Encuentra el elemento más pequeño de la lista y lo coloca al inicio. Repite el proceso con el resto de la lista.

○ Es más eficiente que el Bubble Sort, pero sigue siendo lento para listas grandes.

4. Insertion Sort (Ordenamiento por inserción):

○ Construye la lista ordenada elemento por elemento, insertando cada nuevo elemento en la posición correcta.

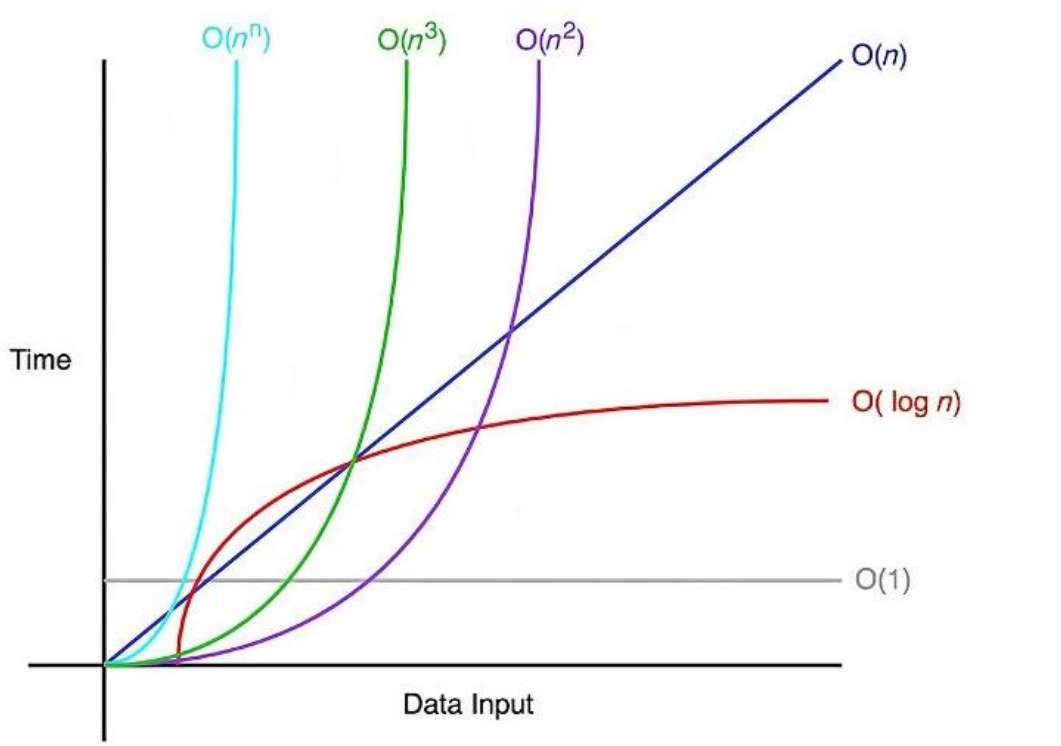
○ Es eficiente para listas pequeñas o parcialmente ordenadas.

Complejidad de los algoritmos

La complejidad medida con O(n) es una forma de medir la eficiencia de un algoritmo. Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada. Por ejemplo, un algoritmo con una complejidad de O(n) tardará el doble de tiempo en ejecutarse si el tamaño de la entrada se duplica.

La notación O(n) se utiliza para describir el peor caso de complejidad de tiempo de un algoritmo. Esto significa que el algoritmo nunca tardará más de O(n) tiempo en ejecutarse para cualquier entrada de tamaño n.

La complejidad medida con O(n) es una herramienta útil para comparar diferentes algoritmos y elegir el más eficiente para una tarea determinada.



Caso Práctico

Código utilizado:

```
import random
import time

#Método Bubble Sort:

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]

#Metodo Quick Sort:

def quick_sort(arr):
    def _quick_sort(items, low, high):
        if low < high:
            pi = particion(items, low, high)
            _quick_sort(items, low, pi - 1)
            _quick_sort(items, pi + 1, high)

    def particion(items, low, high):
        pivot = items[high]
        i = low - 1
        for j in range(low, high):
            if items[j] <= pivot:
                i += 1
                items[i], items[j] = items[j], items[i]
        items[i + 1], items[high] = items[high], items[i + 1]
        return i + 1

    _quick_sort(arr, 0, len(arr) - 1)

#Método Selection Sort:

def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
```

```
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

#Método Insertion Sort:

def insertion_sort(arr):
    for i in range(1, len(arr)):
        actual = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > actual:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = actual

#Función para medir el tiempo de cada método:

def medir_tiempo(algoritmo, lista):
    copia = lista.copy()
    inicio = time.time()
    algoritmo(copia)
    fin = time.time()
    return fin - inicio

#Comparación de tiempos:

tamaños = [100, 500, 1000]
algoritmos = [
    ("Bubble Sort", bubble_sort),
    ("Quick Sort", quick_sort),
    ("Selection Sort", selection_sort),
    ("Insertion Sort", insertion_sort)
]

for n in tamaños:
    print(f"\n📏 Tamaño de lista: {n}")
    lista = [random.randint(0, 10000) for _ in range(n)]
    for nombre, funcion in algoritmos:
        tiempo = medir_tiempo(funcion, lista)
        print(f"{nombre:<18}: {tiempo:.6f} segundos")
```

Descripción general del funcionamiento del código

El presente código en Python tiene como objetivo implementar y comparar el rendimiento de cuatro algoritmos clásicos de ordenamiento: Bubble Sort, Quick Sort, Selection Sort e Insertion Sort. Para ello, se genera una lista de números aleatorios de distintos tamaños y se mide el tiempo que tarda cada algoritmo en ordenarla.

Cada algoritmo está definido en una función independiente:

- **Bubble Sort** recorre repetidamente la lista comparando elementos adyacentes y los intercambia si están en el orden incorrecto. Este proceso se repite hasta que la lista está ordenada.
- **Quick Sort** utiliza el enfoque de "divide y vencerás". Selecciona un elemento como pivote y reordena los elementos de forma que los menores al pivote queden a su izquierda y los mayores a su derecha. Luego aplica el mismo procedimiento recursivamente a las sublistas resultantes.
- **Selection Sort** busca el valor mínimo en la lista y lo coloca en la primera posición, luego repite el proceso con el resto de la lista hasta ordenarla por completo.
- **Insertion Sort** construye la lista ordenada de forma incremental. Para cada elemento, lo compara con los anteriores y lo inserta en la posición adecuada desplazando los demás elementos si es necesario.

Para evaluar el rendimiento de cada algoritmo, se define la función *medir_tiempo*, que mide el tiempo de ejecución utilizando el módulo *time*. Esta función recibe como parámetros un algoritmo de ordenamiento y una lista, crea una copia de la lista para no modificar la original, y devuelve el tiempo que tardó en completarse la ordenación.

Finalmente, el código genera listas aleatorias de tres tamaños distintos (100, 500 y 1000 elementos) y aplica sobre ellas cada uno de los algoritmos. Luego imprime en pantalla el tiempo que cada uno tardó en ordenarlas, lo que permite comparar su eficiencia en distintos escenarios.

Metodología Utilizada

Para la realización de este trabajo seguimos una serie de etapas que nos permitieron avanzar de forma organizada y efectiva:

- **Investigación teórica:** Recopilamos información confiable sobre los algoritmos de búsqueda y ordenamiento, enfocándonos en sus características, funcionamiento y eficiencia.
- **Implementación práctica:** Programamos en Python los algoritmos seleccionados (Bubble Sort, Quick Sort, Selection Sort e Insertion Sort), asegurándonos de que funcionaran correctamente.
- **Pruebas y experimentación:** Realizamos pruebas utilizando listas de diferentes tamaños para observar el comportamiento de cada algoritmo en distintos escenarios.
- **Medición y análisis de resultados:** Registramos los tiempos de ejecución de cada algoritmo para compararlos y analizar su rendimiento.
- **Redacción del informe:** Finalmente, organizamos toda la información y conclusiones obtenidas en este informe, reflejando tanto la parte teórica como la práctica del trabajo.

Resultados obtenidos

Al ejecutar el código con listas de 100, 500 y 1000 elementos generadas aleatoriamente, se observaron diferencias significativas en los tiempos de ejecución de los algoritmos. Quick Sort fue consistentemente el más rápido, gracias a su enfoque recursivo y eficiente incluso en listas grandes, con una complejidad promedio de **O(n log n)**. Insertion Sort y Selection Sort mostraron un rendimiento aceptable en listas pequeñas, pero su desempeño disminuyó al aumentar el tamaño de la lista, ya que ambos tienen una complejidad de **$O(n^2)$** en el peor de los casos. Bubble Sort fue el más lento de los cuatro, también con una complejidad de **$O(n^2)$**,

lo que se reflejó en tiempos de ejecución considerablemente mayores, especialmente con 1000 elementos.

Estos resultados confirman que la elección del algoritmo de ordenamiento influye directamente en el rendimiento del programa, y que algoritmos más eficientes como Quick Sort son preferibles en contextos donde se manejan grandes volúmenes de datos.

Conclusiones

Con este trabajo aprendimos cómo funcionan diferentes formas de **buscar y ordenar datos** usando Python. Pudimos ver que, aunque todos los algoritmos hacen lo mismo —buscar o acomodar datos—, **algunos son más rápidos que otros**, sobre todo cuando la cantidad de datos es grande.

Nos dimos cuenta de que es muy importante **elegir bien qué método usar**, porque eso puede hacer que un programa funcione mucho mejor y tarde menos en dar resultados.

También entendimos que los algoritmos más simples, como el **Bubble Sort**, son buenos para aprender, pero no tan eficientes. En cambio, otros como el **Quick Sort** son mucho más rápidos, aunque un poco más difíciles de entender al principio.

Además, este trabajo nos ayudó a **practicar lógica de programación**, a escribir código y a probar cómo se comportan las soluciones en distintos casos. Eso nos dio una idea más clara de cómo se aplican estos temas en la vida real.

Como mejora, podríamos sumar más algoritmos en el futuro o probar con más datos para seguir comparando.

En resumen, fue un trabajo muy útil para aprender, practicar y trabajar en equipo.

Bibliografía

- Autor desconocido. (2023, diciembre). *Algoritmos de ordenamiento y búsqueda en Python: Optimizando la gestión de datos*. 4Geeks. https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python

- Universidad Tecnológica Nacional (s.f.). Material teórico provisto en el aula virtual. Plataforma SIED UTN. https://tup.sied.utn.edu.ar/mod/resource/view.php?id=3434

- Python Software Foundation. (n.d.). The Python Standard Library. Python.org. https://docs.python.org/3/library/

Anexos

Asistencia de inteligencia artificial (IA) mediante ChatGPT, utilizada para mejorar la redacción, claridad y organización del informe.

Repositorio en GitHub: insertar enlace

Video explicativo: (insertar enlace al video en YouTube)