Algoritmos de Búsqueda y Ordenamiento en Python

Alumnos:

Matias Giraldo, matias.giraldo@tupad.utn.edu.ar

Marcos Glocker, marcosglocker2@gmail.com

Materia: Programación I

Profesor/a: Ariel Enferrel

Fecha de Entrega: 09/06/2025

Índice

1. Introducción

El tema elegido es Algoritmos de búsqueda y ordenamiento en Python por el motivo de cómo se realizan las masivas búsquedas en internet actualmente y cómo los resultados de estas son eficientes en lo que respecta a lo que busca el usuario. Estos algoritmos son fundamentales en el área de programación ya que son aplicables en casi todos los sistemas que manipulan datos, su estudio permite comprender cómo se manipulan y organizan los datos de manera eficiente.
Con el desarrollo del trabajo se pretende explorar algunos de los algoritmos más representativos implementados en Python, entendiendo su funcionamiento, ventajas y desventajas. Aunque se hará un énfasis especial en el algoritmo de búsqueda binaria, que es considerado uno de los métodos más eficientes y fundamentales en estructuras de datos ordenadas.

2. Marco Teórico

¿Qué son los algoritmos de búsqueda?

Los algoritmos de búsqueda son procedimientos que buscan un elemento particular dentro de una estructura de datos o espacio de búsqueda. Su objetivo es encontrar una solución o valor que cumpla con ciertas condiciones específicas.

Básicamente podemos decir, que es una secuencia de instrucciones que permite localizar un elemento dentro de una estructura de datos o un conjunto de datos.

¿Qué tipos de algoritmos de búsqueda existen? ¿Cuáles son sus ventajas y desventajas? Casos de uso. Describir el funcionamiento del algoritmo de búsqueda binaria."

Existen distintos tipos de algoritmos de búsqueda, entre los más utilizados tenemos:

- **Búsqueda Lineal:** son algoritmos de búsqueda sencillos que verifican cada elemento de una lista o matriz uno por uno hasta que encuentran el elemento objetivo o llegan al final de la lista. Son útiles cuando la lista no está ordenada o para listas pequeñas; tienen un tiempo de ejecución de O(n), lo que significa que el tiempo de búsqueda es directamente proporcional al tamaño de la lista.
- **Búsqueda Binaria:** son algoritmos de búsqueda eficiente que funciona en conjuntos de datos ordenados (en caso de que no estén ordenados dichos datos, la búsqueda puede devolver errores). Funciona dividiendo repetidamente la lista en dos mitades, descartando la mitad que no contiene el elemento buscado, hasta que se encuentra el elemento o se determina que no está en la lista; 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.
- **Búsqueda de Interpolación:** es un algoritmo de búsqueda que sirve para encontrar un valor específico en una matriz ordenada, en cada paso, estima la posición del elemento objetivo dentro de la matriz en función a su valor; esta puede llegar a ser más eficiente que la búsqueda binaria en conjuntos de datos grandes, con distribuciones uniformes de valores.
- **Búsqueda de Hash:** es un método para buscar datos rápidamente. Funciona asignando una "huella" o "clave" única a cada dato, lo que permite ubicarlo de forma

eficiente, sin necesidad de buscar uno por uno, (función hash), es muy eficiente para conjuntos de datos grandes.

- **Algoritmos de búsqueda por comparación:** Mejoran la búsqueda lineal eliminando progresivamente registros en función de propiedades y trabajando con estructuras de datos ordenadas.

Estos algoritmos recientemente mencionados, tienen varios métodos y/o tipos de uso, en cuanto al campo estudio y empleo de la programación podríamos mencionar:

- **Bases de Datos:** Para encontrar registros específicos, como usuarios o productos.
- **Motores de Búsqueda:** Para indexar y recuperar información de la web.
- **Inteligencia Artificial:** Para tomar decisiones, resolver problemas y analizar grandes cantidades de datos.
- **Redes y Gráficos:** Para encontrar rutas o caminos entre nodos.
- **Algoritmos de Ordenación:** Algunos algoritmos de ordenación, como Merge Sort, utilizan algoritmos de búsqueda como parte de su implementación.
- **Sistemas operativos:** Facilitan la búsqueda de archivos, procesos y recursos dentro del sistema.
- **Aplicaciones móviles:** Se utilizan para la búsqueda de contactos, aplicaciones, contenido multimedia y más.

En cuanto a las consideraciones sobre estos algoritmos de búsqueda, podemos tener en cuenta:

- **Eficiencia:** La elección del algoritmo de búsqueda adecuado depende de la estructura de los datos y la cantidad de elementos a buscar.
- **Complejidad:** Algunos algoritmos tienen mayor complejidad temporal, lo que puede afectar su rendimiento con grandes conjuntos de datos; es 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.
- **Optimización:** Se pueden utilizar técnicas de optimización para mejorar la eficiencia de los algoritmos de búsqueda.

¿Qué son los algoritmos de ordenamiento?

Los algoritmos de Ordenamiento son los procedimientos para ordenar elementos de una lista o conjunto de datos, repitiendo una secuencia de pasos, permiten ordenar

una lista de elementos para que esta sea más utilizable, generalmente colocando los elementos en orden numérico o lexicográfico , son un componente esencial de muchas otras aplicaciones, incluyendo herramientas de búsqueda , análisis de datos y comercio electrónico.

Estos tipos de algoritmos tienen varias aplicaciones, entre las principales están:

- **Búsqueda:** La ordenación es un paso crucial en algoritmos de búsqueda como la búsqueda binaria, que requiere datos ordenados.
- **Gestión de datos:** Permiten organizar la información de bases de datos, facilitar la búsqueda y recuperación de datos, y mejorar el análisis.
- **Análisis de datos:** La ordenación ayuda a identificar patrones, tendencias y outliers en conjuntos de datos.
- **Sistemas de información:** Se utilizan en sistemas de información para organizar y presentar datos de manera eficiente, como en informes financieros o historiales médicos.
- **Programación:** Son esenciales en el desarrollo de software para optimizar la búsqueda y recuperación de datos, mejorar la experiencia del usuario y facilitar la implementación de funcionalidades.

Si se habla de algoritmos de ordenamiento, podemos mencionar múltiples ejemplos diferentes, con sus pro y contras a la hora de su utilización, en este caso dejamos los más comunes:

- **Bubble Sort:** Recorre la lista repetidamente, intercambiando elementos adyacentes si están en el orden incorrecto, es fácil de entender, pero no muy eficiente para listas grandes.
- **Selection Sort:** Encuentra el elemento más pequeño en la lista y lo coloca en la posición correcta, repitiendo este proceso,es más eficiente que el Bubble Sort, pero sigue siendo lento para listas grandes
- **Insertion Sort:** Construye la lista ordenada de forma gradual, insertando cada elemento en la posición correcta, es eficiente para listas pequeñas o parcialmente ordenadas.
- **Merge Sort:** Divide la lista en sublistas, las ordena y las fusiona en una lista ordenada.
- **Quick Sort:** Utiliza un pivote para dividir la lista en sublistas, y luego las ordena.
- **Heap Sort:** Utiliza una estructura de datos de heap para ordenar la lista.
- **Counting Sort:** Ideal para ordenar listas de números enteros, contando la frecuencia de cada número.
- **Radix Sort:** Ordena elementos basados en sus dígitos.

3. Caso Práctico

Primeramente, se le pedirá al usuario que cree una lista de 10 números enteros y distintos entre sí, luego el problema sería el de ordenar la lista de números (usaremos Insertion Sort), el criterio para su ordenamiento será de menor a mayor. Después, se plantearía al usuario que ingrese un número, el programa empezará con la búsqueda del número en la lista, en el caso de encontrarse con él, devolverá la posición del número en la lista, en caso de que no esté, se devolverá el mensaje “El número no se encuentra en la lista” y finalizará el programa.

Código fuente:

```
#Funciones
def ordenamiento_por_insercion(lista):
    """Ordena la lista de números de menor a mayor usando el
algoritmo de inserción

    Este algoritmo construye la lista ordenada final un elemento
a la vez. Recorre
    la lista de entrada y, en cada iteración, toma un elemento y
lo inserta en la
    posición correcta dentro de la sublista ya ordenada.

    Parámetros:
        lista: La lista de números desordenada que se desea
ordenar.

    Retorna:
        lista: La misma lista de entrada, pero ordenada de menor
a mayor.
    """
    for i in range(1, len(lista)):
        temp = lista[i]
        j = i - 1
        while j >= 0 and temp < lista[j]:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = temp
    return lista

def busqueda_binaria(lista,num):
    """Busca un número en la lista previamente ordenada usando
búsqueda binaria
```

```
    Es un algoritmo eficiente de tipo "divide y vencerás".
Funciona dividiendo
    repetidamente el intervalo de búsqueda por la mitad hasta
encontrar el
    elemento o determinar que no está en la lista

    Requisito:
        La lista de entrada debe estar ordenada para que el
algoritmo funcione correctamente ()

    Parámetros:
        lista: La lista en la que se realizará la búsqueda.
        num: El número que se desea encontrar en la lista

    Retorna:
        int: El índice del número si se encuentra en la lista o
si el número no se encuentra, retorna -1
    """
    inicio = 0
    fin = len(lista) - 1
    while inicio <= fin:
        # valor del medio
        med = (inicio + fin) // 2
        if lista[med] == num:
            return med
        elif lista[med] < num:
            inicio = med + 1
        else:
            fin = med - 1
    return -1
# Principio del programa
print("Ingrese 10 numeros distintos, estos formaran parte de una
lista")
# Lista vacía
lista = []
# Usamos un bucle while que terminará cuando la lista tenga 10
elementos
while len(lista) < 10:
    try:
        numero_ingresado = int(input(f"Ingrese el número
{len(lista) + 1}: "))

        # Se verifica si el número ya existe en la lista
```

```
        if numero_ingresado in lista:
            # Si ya está, informamos al usuario y el bucle vuelve
a empezar.
            print(f"ERROR, el numero {numero_ingresado} ya fue
ingresado. Intente con otro.")
        else:
            # Si no está, lo agregamos a la lista.
            lista.append(numero_ingresado)
    # Se imprime esto si lo que se ingresa es diferente de un
numero entero o si se ingresa un numero que ya está en la lista.
    except ValueError:
        print("ERROR, ingrese solo numeros enteros")
# Se define variable lista_ordenada como una lista modificada por
una función
lista_ordenada = ordenamiento_por_insercion(lista)
print(f"\nLa lista ordenada de menor a mayor
es:\n{lista_ordenada}")
# Se pide al usuario un numero de la lista
numero = int(input("\nAhora, ingrese un numero de la lista para
encontrar su posición exacta: "))
# se define variable posicion con el retorno de la funcion
busqueda_binaria
posicion = busqueda_binaria(lista_ordenada,numero)
# Condicional que evalua si el numero esta o no en la lista
if posicion == -1:
    print("El numero no se encuentra en la lista")
else:
    print(f"Su posicion exacta es la {posicion}")
```

Se utilizó Insertion Sort debido a que es un método fácil de programar y además, al ser una lista pequeña (10 elementos) la que se ordena, es un método que se adecua a este tipo de tamaño pequeño. Lo que es el algoritmo de búsqueda, se implementó la búsqueda binaria, porque es más eficiente, en este caso, no se luce demasiado porque la lista es pequeña, pero su desarrollo y aplicación en otros contextos es de lejos mucho mejor que la lineal.

4. Metodología Utilizada

1) Vimos todo el contenido disponible en el aula virtual (videos, PP, etc.)
2) Complementamos con más información de otros sitios
3) Aplicamos en Python algunos de los conceptos (Insertion Sort y búsqueda binaria)
4) Hicimos varias pruebas del código y logró funcionar eficientemente.
5) Ingresamos los resultado obtenidos y la conclusión del trabajo
6) Creamos un PowerPoint para presentar el TP en un video.

Uno de los integrantes se enfocó más en la investigación y profundización de los temas para presentarlo en el marco teórico, el otro se enfocó más en el caso práctico, luego complementamos lo que hicimos para poder entender lo que hizo el otro. Para finalmente, completar las partes restantes (Resultados obtenidos, conclusión,etc.)

5. Resultados Obtenidos

Se probaron los algoritmos con listas de distintos tamaños (10, 20 y 50 elementos) ingresadas manualmente, asegurando que fueran valores numéricos. También se simularon búsquedas de valores presentes y ausentes en la lista.
Restringimos que los números ingresados sean enteros y distintos entre sí (para que no se repitan).
El rendimiento del programa es muy bueno ya sea si el tamaño es grande o pequeño, aunque posiblemente, habría que ajustar el algoritmo de ordenamiento, a uno que sea más eficiente para tamaños mayores.

6. Conclusiones

Este trabajo permitió comprender los conceptos sobre los algoritmos de búsqueda y ordenamiento, establecer la importancias de estos en la programación y actualidad. Se aprendió como cada algoritmo tiene un comportamiento distinto según el tamaño y tipos de datos, y por qué es necesario saber el contexto para implementarlos.
Hubiese sido una gran mejora haber hecho un análisis con otro programa parecido, pero en vez de búsqueda binaria y Insertion Sort, utilizar otros tipos de algoritmos (ejemplo: Búsqueda lineal y bubble sort) y comparar la velocidad y eficiencia respecto uno del otro modelo.
Tuvimos dificultades por errores en la entrada de datos, por ejemplo, si el usuario ponía 2 números igual, el programa siempre retornaba una posición (si es que fue el número elegido a buscar), pero se sabe que está en 2 posiciones, entonces esto planteaba un error, lo corregimos usando la estructura de control try-except que nos permite restringir qué tipo de datos no deben validarse.

7. Bibliografía

Información complementaria de búsqueda binaria en python:
https://www.datacamp.com/es/tutorial/binary-search-python
Información complementaria de algoritmos de ordenación:
https://eiposgrados.com/blog-python/tipos-de-algoritmos-de-ordenacion-en-python/
Información complementaria de algoritmos de búsqueda:
https://www.geeksforgeeks.org/interpolation-search/
https://www.luigisbox.es/glosario-de-busqueda/algoritmo-de-busqueda/
Información complementaria de algoritmos de Ordenamiento, tipos y aplicaciones:
https://www.britannica.com/technology/sorting-algorithm
https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-sorting-algorithm/

https://blog.fiverr.com/post/sorting-algorithms-what-are-they-and-how-to-use-them
https://www.wscubetech.com/resources/dsa/sorting-algorithms
https://www.ferrovial.com/es/stem/algoritmos/
Estructura de Control Try-except:
https://www.freecodecamp.org/espanol/news/sentencias-try-y-except-de-python-como-menejar-excepciones-en-python/

8. Anexo

Link del repositorio en Github:
https://github.com/MatiasGiraldo/TP-Integrador-Program1