Algoritmos de Búsqueda y Ordenamiento

Alumnos

Matías Boyer

matiasboyer7@gmail.com

Flavio Bravo

flaviobravo55@gmail.com

Materia

Programación I

Profesor

Cinthia Rigoni

Entrega

9 de junio de 2025

Índice

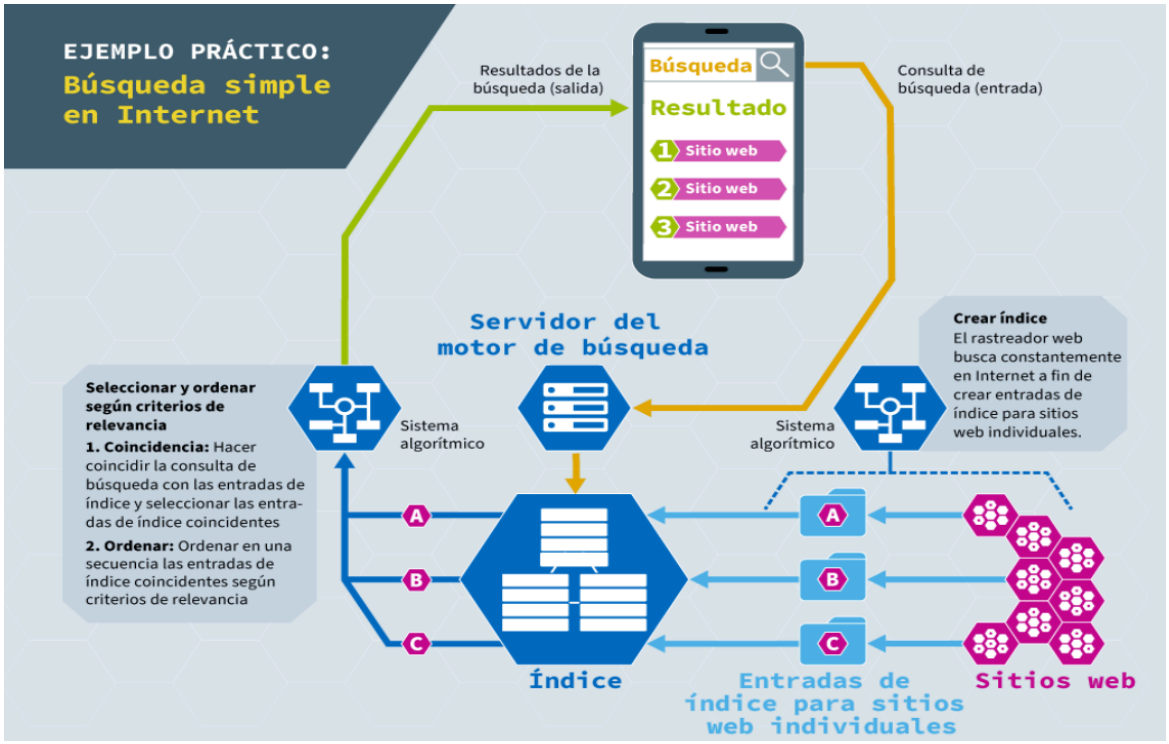
Introducción

Dentro del campo de la programación, es frecuente trabajar con estructuras de datos como listas o tablas. Este tipo de datos, normalmente contienen volúmenes enormes de información, por lo que ordenarlas y buscar dentro de ellas es una tarea fundamental.
El trabajo aborda este tema: los métodos de búsqueda y ordenamiento aplicados a estructuras de datos.
Se elige este tema, por ser esencial dentro de la programación habitual. Resulta indispensable tener que aprender e interiorizarse en el tema, sabiendo los métodos existentes, cómo funcionan y las formas de implementación en distintos entornos, para poder realizar una elección informada al momento de programar.
El objetivo principal del trabajo, es analizar e implementar algoritmos de búsqueda y ordenamiento conocidos, evaluando funcionamiento, complejidad y aplicación en diferentes escenarios.

Marco Teórico

Algoritmos de Búsqueda

Un algoritmo de búsqueda es una serie de pasos que seguimos para encontrar un dato o un elemento dentro de una estructura, como una lista, un árbol, una base de datos, etc. “Es una forma organizada de buscar algo”.
Son importantes en programación y en informática ya que necesitamos encontrar datos rápidos y eficientes.
Ejemplo de búsqueda de algoritmo en internet:



-Se visualizan los procesos que se producen después de una consulta de búsqueda en el servidor del motor de búsqueda.
Información obtenida de © Siemens Stiftung 2020, CC BY-SA 4.0 international.

Existen dos tipos principales de algoritmos de búsqueda: Los algoritmos de búsqueda lineal y los algoritmos de búsqueda binaria. A Continuación se explica cada uno.

Busqueda Lineal

Es el algoritmo de búsqueda más simple, que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. No se necesita que la lista esté ordenada, pero puede ser una búsqueda lenta si la lista del algoritmo es muy larga.

Búsqueda Binaria

Algoritmo eficiente, que sólamente funciona en listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades, y comparando el valor del elemento central con el valor buscado.

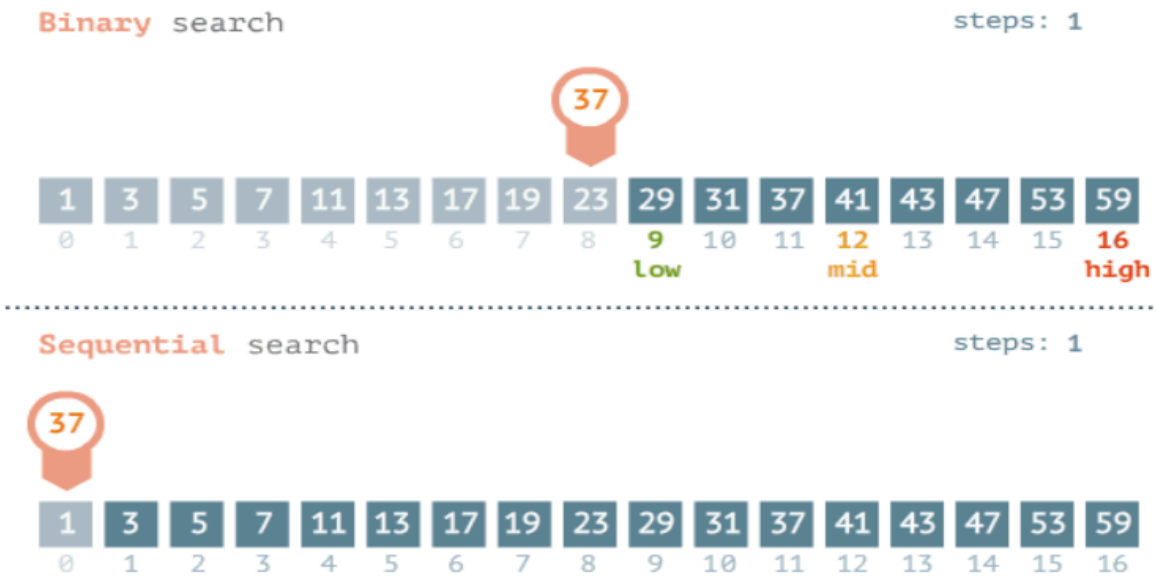
Ventajas

- Eficiencia
- Rapidez

Desventajas

- Requiere que los datos estén ordenados
- Ineficiente para grupos de datos pequeños

-Búsqueda secuencial y binaria del número 37



https://www.reddit.com/r/educationalgifs/comments/9rrk77/binary_vs_sequential_search/?tl=es-es

Complejidad de los algoritmos

La complejidad de un algoritmo describe cuánto tarda (o cuánto recurso usa) a medida que los datos crecen.
Es una medida de eficiencia que ayuda a determinar la efectividad de un algoritmo para resolver un problema. Su medida es la notación O(n) que expresa el tiempo en función a la cantidad de datos n.
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.

Búsqueda Lineal - Complejidad O(n)

La complejidad de la búsqueda lineal es O(n), significa que el tiempo de ejecución del algoritmo aumenta linealmente con el tamaño de la lista que se está buscando.

Búsqueda Binaria - Complejidad O(log n)

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.

A continuación vamos a explicar los tipos de complejidad temporal, pero vamos a excluir el lineal O(n) y el logarítmico O(log n), ya que lo hemos explicado anteriormente.

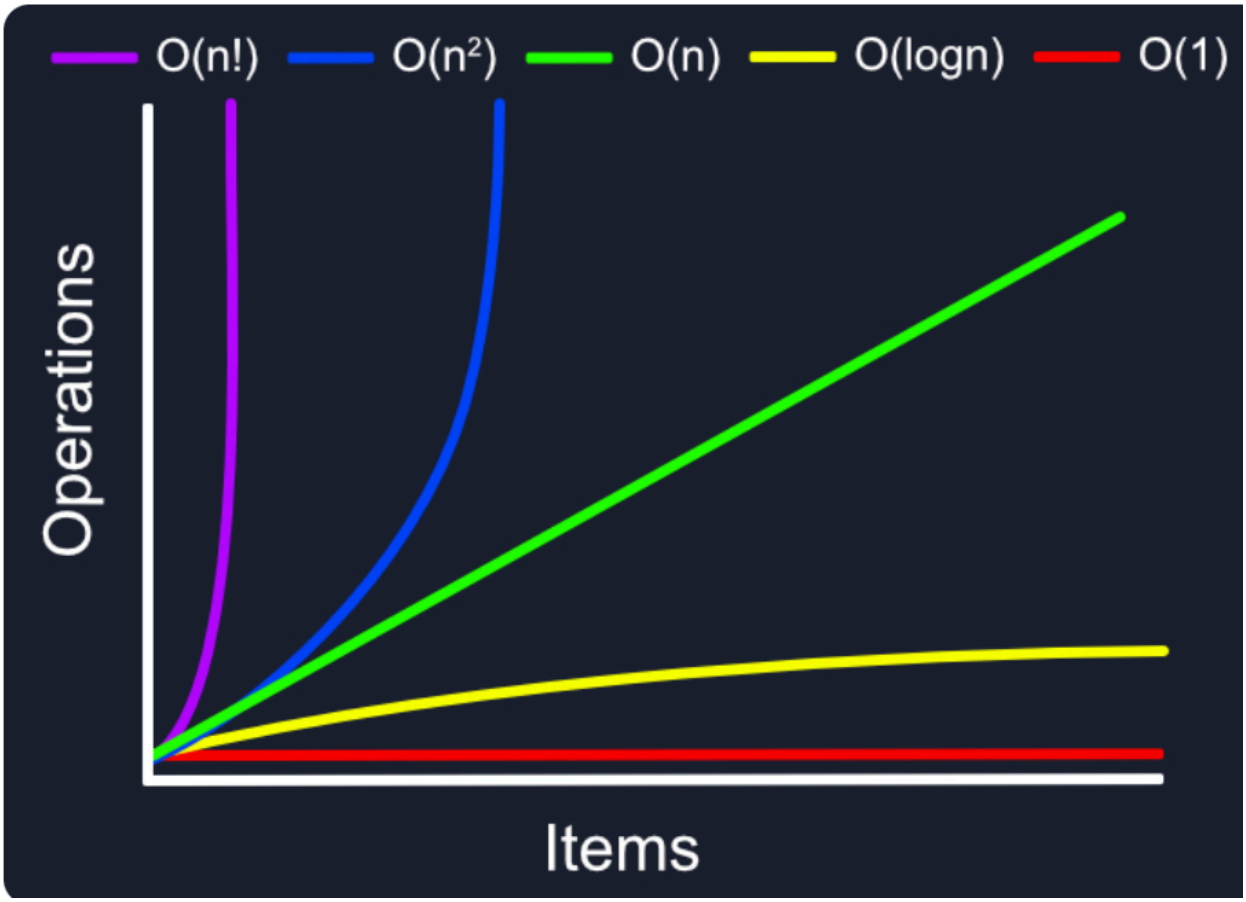
Tipos de complejidad temporal:

-Constante O(1): El tiempo de ejecución siempre será el mismo independientemente del tamaño de entrada. Es el mejor de los casos.

-Cuadrático O(n^2): Cuando se realiza una iteración anidada, es decir un bucle dentro de otro bucle, la complejidad del tiempo es cuadrática, lo cual es muy malo.

-Cúbico O(n^3):Cuando se realizan bucles anidados triples. La complejidad del tiempo es muy mala.

Imagen obtenida de https://www.undefinedworld.com/blog/analisis-de-algoritmos



Algoritmos de Ordenamiento

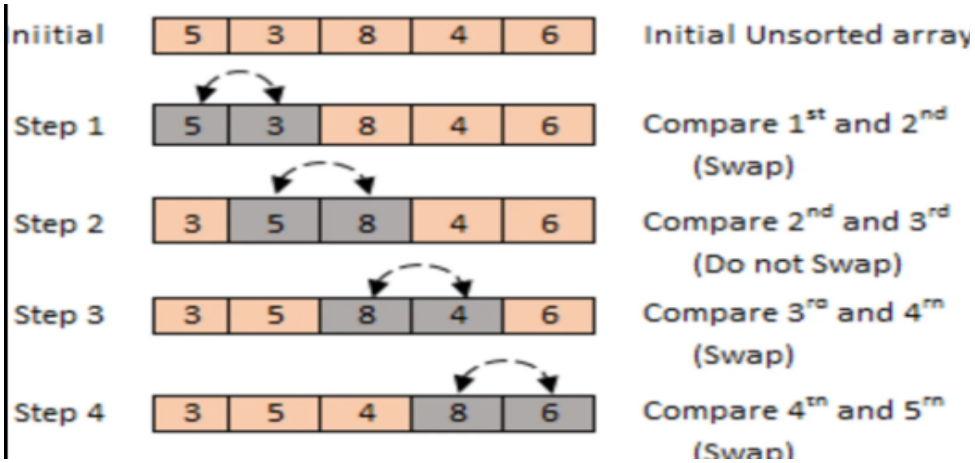
Los algoritmos de ordenamiento son un conjunto de instrucciones que toman una lista como entrada y organizan los elementos en un orden particular. Como de menor a mayor o alfabéticamente.
Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Beneficios de utilizar Algoritmos de Ordenamiento:
-Organización y búsqueda eficiente: Permiten organizar los datos, lo que facilita la búsqueda de información y reduce la búsqueda.
-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.
-Reducción de errores: Al ordenar los datos, se minimizan los errores de búsqueda y el procesamiento de información.
-Mejora la experiencia de usuario: Al ordenar los datos, la información se presenta de manera más clara y fácil de entender.

A continuación presentaremos los tipos de algoritmos de ordenamiento:

Bubble Sort (Ordenamiento por burbuja) - O(n^2)

El ordenamiento de burbuja es un algoritmo que nos permite ordenar valores de una lista. Funciona revisando cada elemento con su adyacente. Si ambos elementos no están ordenados, se procede a intercambiarlos, si por el contrario los elementos ya estaban ordenados se dejan tal como estaban. Este proceso sigue para cada elemento de la lista hasta que quede completamente ordenado.
Imagen que simula el ordenamiento por burbuja

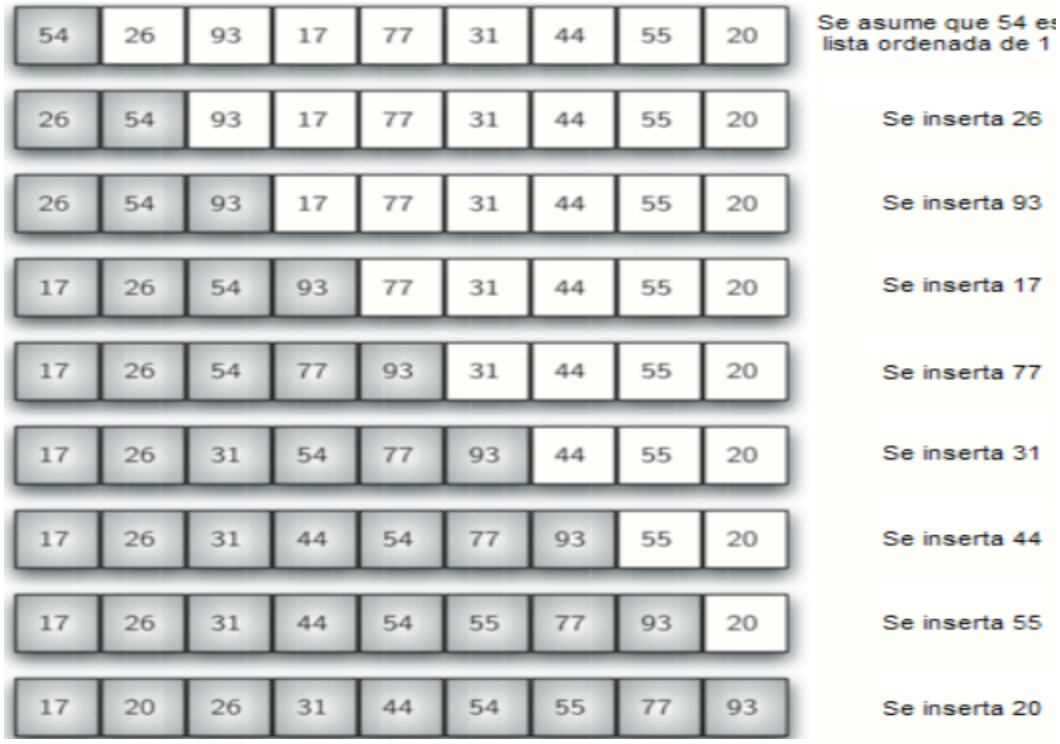


Ordenamiento por inserción (Insertion Sort) - Peor caso O(n^2) / Mejor caso O(n)

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

Compara los valores uno por uno comenzando con el segundo valor de la lista. Si este valor es mayor que el valor a su izquierda no se realizan cambios, de lo contrario este valor se desplaza rápidamente hacia la izquierda hasta que se encuentra con un valor menor.

Imagen ilustrativa de ordenamiento por inserción:



https://runestone.academy/

Selection Sort (Ordenamiento por selección) - O(n^2):

Algoritmo simple, funciona seleccionando repetidamente el elemento más pequeño (o más grande) de la parte no ordenada de una lista, y moviéndolo al inicio de la lista.

Este proceso se repite hasta que la lista esté ordenada

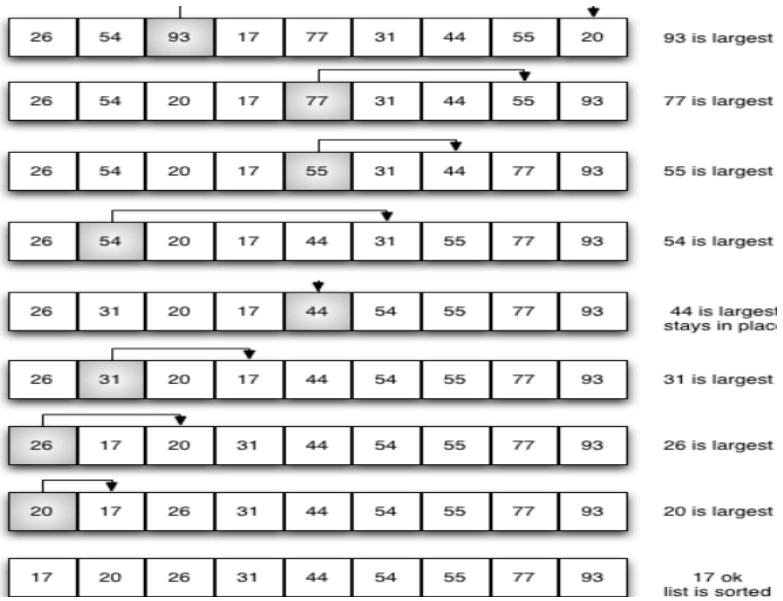
Ventajas

- Simplicidad
- Eficiencia
- No requiere memoria adicional

Desventajas

- El tiempo de ejecución aumenta con el tamaño del conjunto de datos
- NO es adecuado para grupos de datos grandes

Figura de ejemplo de ordenamiento por selección:



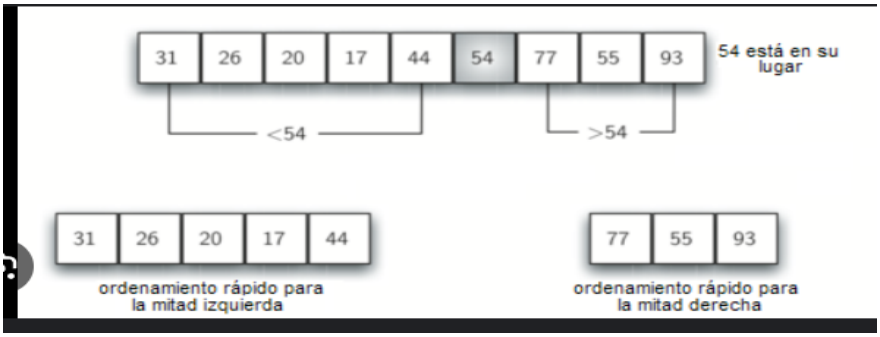
La figura muestra todo el proceso de ordenamiento. En cada paso, el ítem mayor restante se selecciona y luego se pone en su ubicación correcta. La primera pasada ubica el 93, la segunda pasada ubica el 77, la tercera ubica el 55, y así sucesivamente.

Quick Sort (Ordenamiento Rapido) - Peor caso O(n^2) / Mejor caso O(n log n)

Un ordenamiento rápido primero selecciona un valor, que se denomina el valor pivote. Luego organiza los elementos menores a un lado y los mayores al otro. Este tipo de ordenamiento usa el famoso divide y conquistaras.

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

Imagen ilustrativa de ordenamiento rápido:



https://runestone.academy/

Caso Práctico

Algoritmos de Búsqueda

En esta parte del trabajo integrador vamos a presentar un problema o situación concreta sobre cada uno de los algoritmos de búsqueda explicamos en el marco teórico.

Búsqueda lineal

Ahora vamos a programar en Python un ejemplo de búsqueda lineal. Estamos en una biblioteca gigante con un millón de libros numerados del 1 al 1.000.000 y queremos encontrar un libro en particular.
Vamos a simular que buscamos un número (libro) revisando uno por uno, desde el principio.

Implementación en python:

```python
import time

# Creamos la "biblioteca" con libros del 1 al 1.000.000
biblioteca = list(range(1, 1000001))  # lista de 1 a 1.000.000

# Libro que queremos encontrar
objetivo = 50

# Funcion de busqueda lineal
def busqueda_lineal(lista, objetivo):
    for indice, libro in enumerate(lista):
        if libro == objetivo:
            return indice  # posicion donde lo encontro
    return -1  # si no lo encuentra

# tiempo de busqueda
inicio_tiempo = time.time()
resultado = busqueda_lineal(biblioteca, objetivo)
fin_tiempo = time.time()

# Mostramos resultados
if resultado != -1:
    print(f"📘 Libro encontrado en la posición {resultado}")
else:
    print("❌ Libro no encontrado.")

print(f"⏱️ Tiempo de búsqueda lineal: {fin_tiempo - inicio_tiempo:.6f} segundos")
```

Pasos del script:

-Importamos el módulo `time` para poder medir cuánto tarda el algoritmo en ejecutarse.

-Creamos una lista que simula una biblioteca con libros numerados del 1 al 1.000.000.
Cada número representa un "libro". La lista está ordenada, pero la búsqueda lineal no necesita orden.

-Colocamos el objetivo, libro que queremos buscar, en este caso queremos buscar el libro 50
- Definimos la función que hace la búsqueda lineal: recorremos cada libro uno por uno, junto con su posición. si el libro actual es el que buscamos, devuelve su posición. Si llega al final y no lo encuentra, devuelve −1.
- Inicio tiempo: marca el momento antes de empezar a buscar.
-Resultado: guarda la posición del libro encontrado.
-Fin_tiempo: marca el momento después de terminar la búsqueda.
- Se muestra si se encontró el libro o no, y en qué posición.
- Se resta el tiempo final menos el inicial y muestra cuánto tardó la búsqueda y se muestra el número con decimales para ser más preciso.

Búsqueda binaria

Paso a paso:

1. Calcula el elemento central
2. Compara el elemento central con el valor buscado
    - Si es igual al valor buscado, la búsqueda terminó
    - Si es menor que el valor buscado, la búsqueda sigue en la mitad superior de la lista
    - Si es mayor que el valor buscado, la búsqueda sigue en la mitad inferior de la lista
3. Repite los pasos 1 y 2 hasta que la lista se reduzca a una longitud de 0
4. Si no encuentra el elemento, significa que no está en la lista

Implementación en Python

```python
import time

def busqueda_binaria(valor_a_buscar, lista):
    inicio = 0
    fin = len(lista) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == valor_a_buscar:
            return medio
        elif lista[medio] < valor_a_buscar:
            inicio = medio + 1
        else:
            fin = medio - 1

    return -1

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
print("La lista ordenada es:", lista)

valor = int(input("Ingrese el valor a buscar: "))

tiempo_inicial = time.time()
posicion = busqueda_binaria(valor, lista)
tiempo_final = time.time()
print(f"El valor {valor} se encuentra en la posición {posicion}")
print(f"Tiempo de ejecución: {tiempo_final - tiempo_inicial:.6f} segundos")
```

Algoritmos de Ordenamiento

En esta parte del trabajo integrador vamos a presentar un problema o situación concreta sobre cada uno de los algoritmos de ordenamiento explicamos en el marco teórico.

Bubble Sort (Ordenamiento por burbuja) - O(n^2)

Vamos a realizar un script en python que compare el tiempo de ejecución del ordenamiento burbuja (Bubble Sort) para una lista pequeña (por ejemplo, de 10 elementos) y una lista grande (por ejemplo, de 1000 elementos)

Implementación en python:

```python
import time
import random

# Función 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]
    return arr

# ////////////////////////////////
#  Lista pequeña
lista_pequeña = [10, 2, 8, 6, 7, 5, 3, 9, 1, 4]
print("Lista pequeña original:", lista_pequeña)

inicio_peq = time.time()
ordenada_pequeña = bubble_sort(lista_pequeña.copy())
fin_peq = time.time()

print("Lista pequeña ordenada:", ordenada_pequeña)
print("⏱️ Tiempo lista pequeña:", round(fin_peq - inicio_peq, 6), "segundos")

# ////////////////////////////////////////////////////////
# Lista grande (1000 elementos aleatorios)
lista_grande = random.sample(range(1, 1001), 1000)
print("\nLista grande generada con 1000 elementos (no se imprime ya que es muy grande)")

inicio_gra = time.time()
ordenada_grande = bubble_sort(lista_grande.copy())
fin_gra = time.time()

print("⏱️ Tiempo lista grande:", round(fin_gra - inicio_gra, 6), "segundos")
```

Insertion sort

Vamos a realizar un script en python que compare el tiempo de ejecución del ordenamiento por inserción (insertion sort) para una lista pequeña (por ejemplo, de 10 elementos) y una lista grande (por ejemplo, de 1000 elementos)

Implementación en python:

```python
import time
import random

# Función de ordenamiento por inserción
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
    return arr

# ////////////////////////////////
# Lista pequeña
lista_pequeña = [10, 2, 8, 6, 7, 5, 3, 9, 1, 4]
print("Lista pequeña original:", lista_pequeña)

inicio_peq = time.time()
ordenada_pequeña = insertion_sort(lista_pequeña.copy())
fin_peq = time.time()

print("Lista pequeña ordenada:", ordenada_pequeña)
print("⏱️ Tiempo lista pequeña:", round(fin_peq - inicio_peq, 6), "segundos")

#//////////////////////////////////////////
# Lista grande (1000 elementos aleatorios)
lista_grande = random.sample(range(1, 1001), 1000)
print("\nLista grande generada con 1000 elementos")

inicio_gra = time.time()
ordenada_grande = insertion_sort(lista_grande.copy())
fin_gra = time.time()

print("⏱️ Tiempo lista grande:", round(fin_gra - inicio_gra, 6), "segundos")
```

Selection Sort

Paso a paso:

1. Empiezo desde el índice X (al inicio de la ejecución, es 0)
2. Busco el elemento más pequeño desde X+1 hasta el final, dando una posición Y
3. Intercambia X con Y
4. Repito sumando 1 a X

Implementación en Python

```python
import time, random

def selection_sort(lista):
    # Empiezo desde el índice X
    for x in range(len(lista)):
        # Busco el elemento más pequeño desde X hasta el final, dando una posición Y
        for y in range(x + 1, len(lista)):
            if lista[y] < lista[x]:
                # En caso de que el elemento Y sea menor al elemento X, los intercambio
                lista[x], lista[y] = lista[y], lista[x]
    # Repito sumando 1 a X (for loop)

    return lista

lista = [random.randint(1, 100) for x in range(10)]
print(f"La lista desordenada tiene {len(lista)} elementos")
print(f"Los primeros 10 elementos son: {lista[:10]}")

tiempo_inicial = time.time()
lista_ordenada = selection_sort(lista)
tiempo_final = time.time()
print(f"La lista ordenada tiene {len(lista_ordenada)} elementos")
print(f"Los primeros 10 elementos son: {lista_ordenada[:10]}")
print(f"Tiempo de ejecución: {tiempo_final - tiempo_inicial:.6f} segundos")
```

Quick Sort

Paso a paso:

1. Al ser recursivo, reviso si la longitud de la lista es menor o igual a 1. En caso de serlo, devolvemos la lista.
2. Obtengo el pivote, valor central
3. Obtengo los valores a la izquierda del pivote (x < pivote)
4. Obtengo los valores al centro del pivote (x == pivote)
5. Obtengo los valores a la derecha del pivote (x > pivote)
6. Recursivamente:
   Ejecuto quick_sort con los valores de la izquierda
   Le agrego el valor central
   Ejecuto quick_sort con los valores de la derecha

Implementación en Python

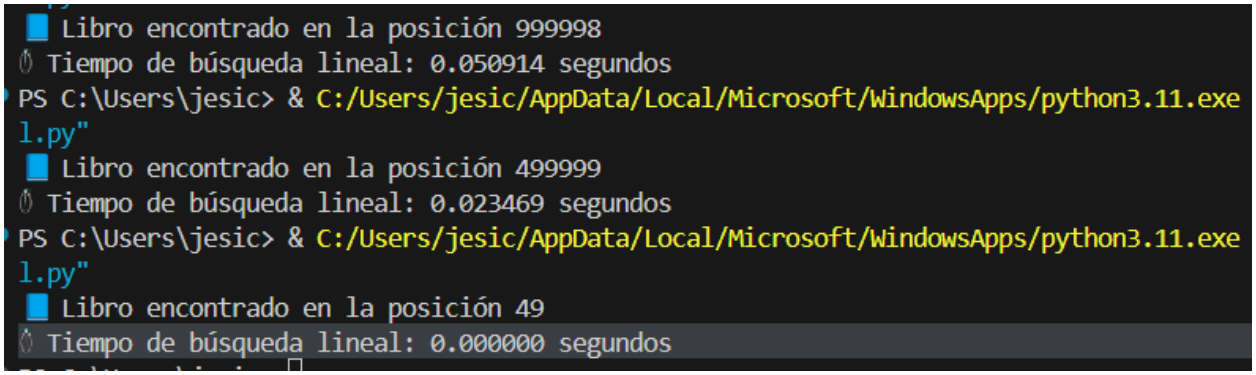
```python
import time, random

def quick_sort(lista):
    if len(lista) <= 1:
        return lista

    pivote = lista[len(lista) // 2]
    izquierda = [x for x in lista if x < pivote]
    centro = [x for x in lista if x == pivote]
    derecha = [x for x in lista if x > pivote]

    return quick_sort(izquierda) + centro + quick_sort(derecha)

lista = [random.randint(1, 100) for x in range(100)]
print(f"La lista desordenada tiene {len(lista)} elementos")
print(f"Los primeros 10 elementos son: {lista[:10]}")

tiempo_inicial = time.time()
lista_ordenada = quick_sort(lista)
tiempo_final = time.time()
print(f"La lista ordenada tiene {len(lista_ordenada)} elementos")
print(f"Los primeros 10 elementos son: {lista_ordenada[:10]}")
print(f"Tiempo de ejecución: {tiempo_final - tiempo_inicial:.6f} segundos")
```

Resultados Obtenidos

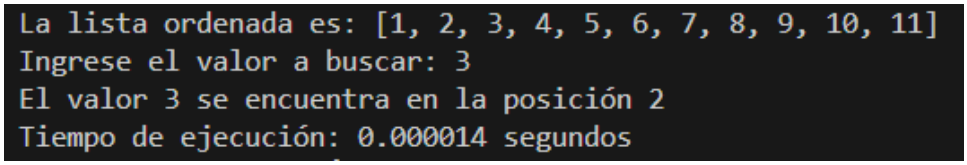
Algoritmos de Búsqueda

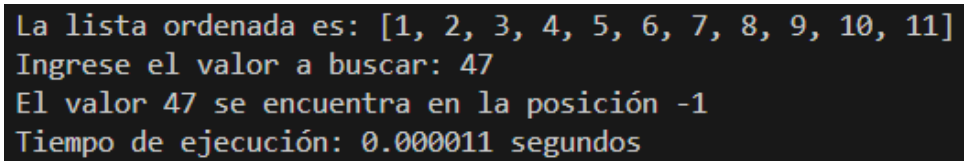
Búsqueda secuencial

Se realiza la prueba del script con diferentes números para corroborar su funcionamiento, primero buscamos la posición del libro 999999, después la posición del libro 500000 y por último la posición del libro 50. Logramos ver como decrece el tiempo de búsqueda si disminuimos la posición del libro que queremos buscar.



Búsqueda binaria

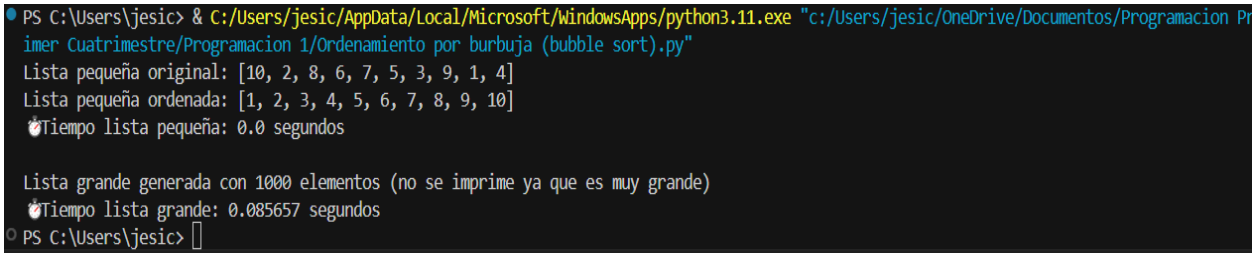




Algoritmos de Ordenamiento

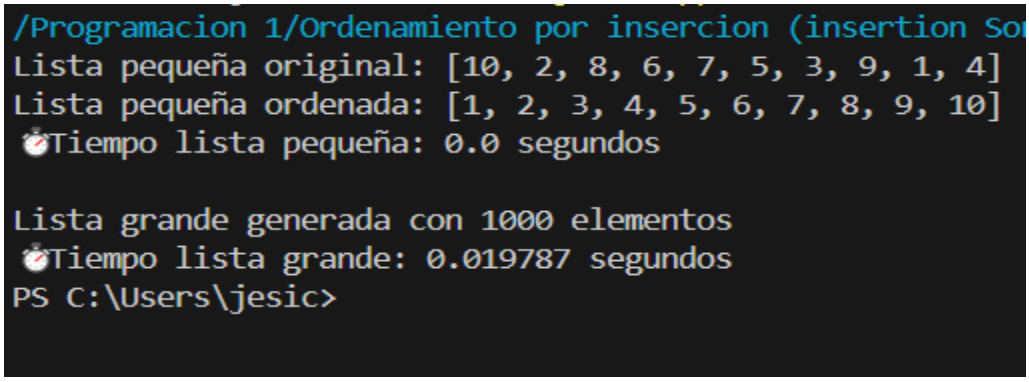
Bubble Sort

Acá podemos ver cómo el ordenamiento funciona muy bien para listas pequeñas, pero cuando el tamaño de los datos crece, la eficiencia decae.



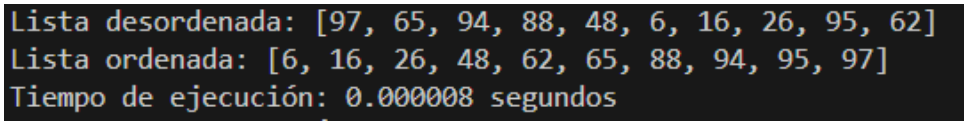
Insertion Sort

Este ordenamiento es muy eficiente para listas pequeñas y casi ordenadas

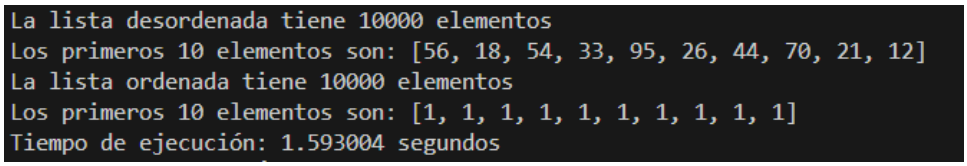


Selection Sort

Lista de 10 elementos



Lista de 10000 elementos



Quick Sort

Lista de 10 elementos

```
La lista desordenada tiene 10 elementos
Los primeros 10 elementos son: [22, 90, 75, 6, 86, 5, 10, 14, 9, 3]
La lista ordenada tiene 10 elementos
Los primeros 10 elementos son: [3, 5, 6, 9, 10, 14, 22, 75, 86, 90]
Tiempo de ejecución: 0.000014 segundos
```

Lista de 10000 elementos

```
La lista desordenada tiene 10000 elementos
Los primeros 10 elementos son: [83, 9, 67, 30, 9, 81, 75, 45, 75, 65]
La lista ordenada tiene 10000 elementos
Los primeros 10 elementos son: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
Tiempo de ejecución: 0.004450 segundos
```

Metodología Utilizada

- Decidimos trabajar en 2 algoritmos de búsqueda y 4 algoritmos de ordenamiento
- Cada uno trabajó en 1 algoritmo de búsqueda y 2 algoritmos de ordenamiento
- Buscamos la explicación teórica del algoritmo
- Buscamos el paso a paso en el caso práctico, y lo implementamos en python
- Mostramos las pruebas realizadas en la sección de Resultados Obtenidos

Conclusiones

- Aprendimos algunos algoritmos de búsqueda y ordenamiento conocidos
- Los implementamos en Python
- El tema elegido es de utilidad para el futuro, ya que nos permite saber cuáles son las mejores opciones al momento de tener que implementar una búsqueda en listas pequeñas o grandes