Trabajo Práctico Integrador

Algoritmos de Búsqueda y Ordenamiento, Análisis de Algoritmos y Estructuras de Datos Avanzadas.

Alumnos

Yanela Luciana Mubilla

Alejandro Lagos

Carrera y materia

Universidad Tecnológica Nacional - Programación a Distancia

Programación 1

Profesor

Sebastián Bruselario

09 de junio de 2025

Índice

Introducción..... 1
Marco Teórico..... 1
Algoritmos de búsqueda..... 1
Algoritmos de Ordenamiento..... 2
Caso práctico..... 3
Metodología Utilizada..... 7
Resultados Obtenidos..... 7
Conclusiones..... 7
Bibliografía..... 8
Anexos..... 8

Introducción

En la programación, los algoritmos de búsqueda y ordenamiento son fundamentales para la gestión eficiente de la información. Su conocimiento permite desarrollar programas más rápidos, precisos y escalables.

Este trabajo explora los principales algoritmos de búsqueda y ordenamiento abordados en la materia, enfatizando sus ventajas, desventajas, complejidad computacional y aplicaciones. Se implementan en Python con un mismo caso práctico para ilustrar su funcionamiento.

Marco Teórico

Algoritmos de búsqueda

Los algoritmos de búsqueda permiten localizar un elemento dentro de un conjunto de

datos, como listas, bases de datos o estructuras más complejas. Se utilizan ampliamente en tareas como recuperación de información, búsquedas en sistemas y soluciones de problemas computacionales. (Khan Academy, s.f.).

- **Búsqueda Lineal:** Recorre la lista elemento por elemento hasta encontrar el objetivo. Es simple, aplicable a listas desordenadas, pero ineficiente para grandes volúmenes de datos.

  Complejidad: O(n) (KeepCoding, s.f.).

- **Búsqueda Binaria:** Sólo funciona en listas ordenadas. Divide el conjunto en mitades sucesivas, comparando con el elemento central.

  Complejidad: O(log n) (4Geeks, s.f.).

Algoritmos de Ordenamiento

Los algoritmos de ordenamiento organizan datos de acuerdo a un criterio determinado (por ejemplo, de menor a mayor), facilitando búsquedas, análisis y operaciones posteriores sobre esos datos. (Khan Academy, s.f.).

- **Bubble Sort:** Compara e intercambia pares adyacentes hasta ordenar toda la lista. Sencillo, pero lento.

  Complejidad: O(n^2) (4Geeks, s.f.).

- **Selection Sort:** Busca el mínimo en cada pasada y lo coloca en su lugar. Poco eficiente en listas grandes.

  Complejidad: O(n^2) (KeepCoding, s.f.).

- **Insertion Sort**: Inserta cada elemento en su posición correcta dentro de una lista

ordenada parcial.

Eficiente en listas pequeñas o semi ordenadas.

Complejidad: O(n^2) (Khan Academy, s.f.).

- **Quick Sort:** Selecciona un pivote y divide recursivamente en sublistas menores y mayores. Muy eficiente en la práctica.

  Complejidad promedio: O(n log n) (4Geeks, s.f.).

Existen otros algoritmos como Shell Sort, Heap Sort, Comb Sort, Merge Sort, entre otros, que por criterios del trabajo no se desarrollan aquí, pero vale la pena conocer su existencia y aplicaciones prácticas (Gbaudino, s.f.).

Caso práctico

Problema: Dada una lista de productos con su código numérico, se desea:

1. Ordenar los productos de menor a mayor código.
2. Buscar si un producto específico (por su código) está presente en la lista.

Datos iniciales:

```
productos = [4021, 1500, 3200, 873, 2999, 1002, 789, 4300, 2200, 1450, 5000, 3900, 2800, 600, 350, 7500, 3100]
buscar_codigo  =  2999
```

**Ejemplo aplicado en todos los algoritmos (ver Anexos para código completo).*

→ **Búsqueda Lineal:**

```python
# ESTILOS PARA DESTACAR LOS TÍTULOS DE LOS ALGORITMOS EN CONSOLA:

from colorama import Fore, Style, init
init(autoreset=True)


# LISTA BASE:

productos = [4021, 1500, 3200, 873, 2999, 1002, 789, 4300, 2200, 1450, 5000, 3900, 2800, 600, 350, 7500, 3100]
buscar_codigo = 2999


"""BÚSQUEDA LINEAL"""

def busqueda_lineal(lista,codigo):
    for i in range(len(lista)):
        if lista[i] == codigo:
            return i
    return -1 #indica que no se encontró el elemento buscado.

print(Style.BRIGHT + Fore.CYAN + "\nBÚSQUEDA LINEAL")
print(f"El código {buscar_codigo} se encuentra en la posición:", busqueda_lineal(productos, buscar_codigo))

```

```
PS D:\UTN_P1> & C:/Users/yane0/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/UTN_P1/Integrador.py

BÚSQUEDA LINEAL
El código 2999 se encuentra en la posición: 4
PS D:\UTN_P1>
```

*Este algoritmo de búsqueda es óptimo para buscar elementos en listas pequeñas.

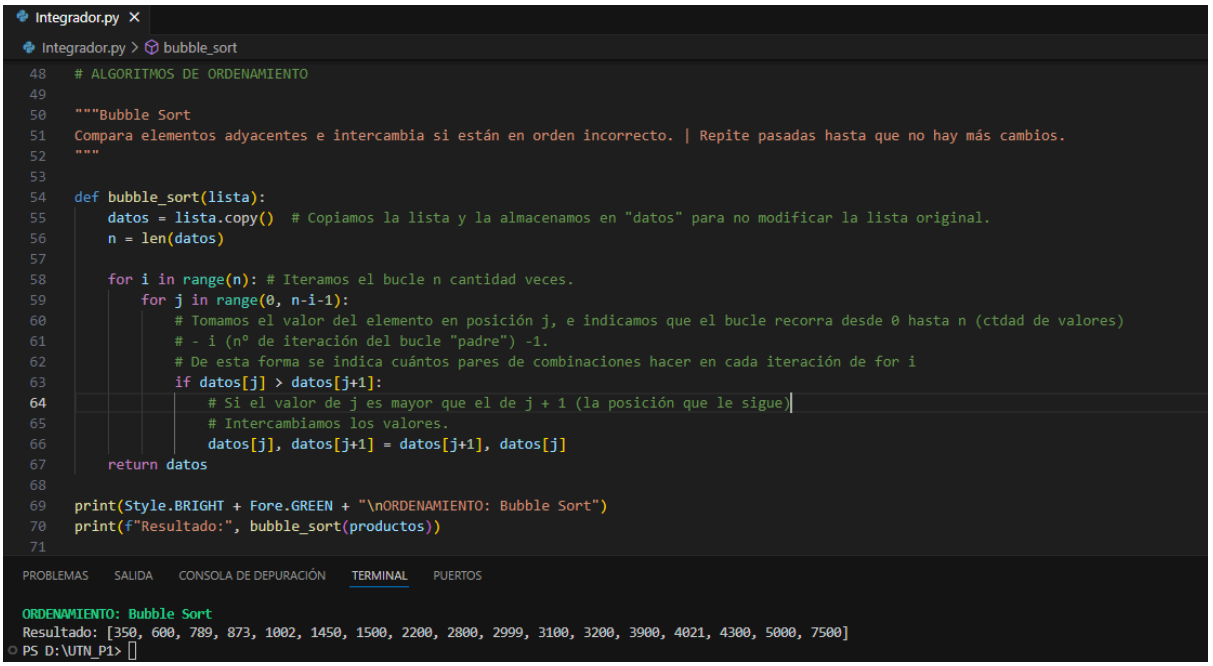
→ **Búsqueda Binaria:**

```python
"""BÚSQUEDA BINARIA"""

lista_ordenada = sorted(productos)

def busqueda_binaria(lista,codigo):
    izquierda, derecha = 0, len(lista) -1 #Utilizamos una asignación múltiple (en lugar de izquierda = 0 y derecha = len(lista) -1)
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == codigo:
            return medio
        elif lista[medio] < codigo:
            izquierda = medio + 1
        else:
            derecha = medio -1
    return -1 #indica que no se encontró el elemento buscado.

print(Style.BRIGHT + Fore.CYAN + "\nBÚSQUEDA BINARIA")
print(f"El código {buscar_codigo} se encuentra en la posición:", busqueda_binaria(lista_ordenada, buscar_codigo))

```

```
PS D:\UTN_P1> & C:/Users/yane0/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/UTN_P1/Integrador.py

BÚSQUEDA BINARIA
El código 2999 se encuentra en la posición: 9
PS D:\UTN_P1>
```

*El algoritmo de búsqueda binaria es más útil para listas más grandes.

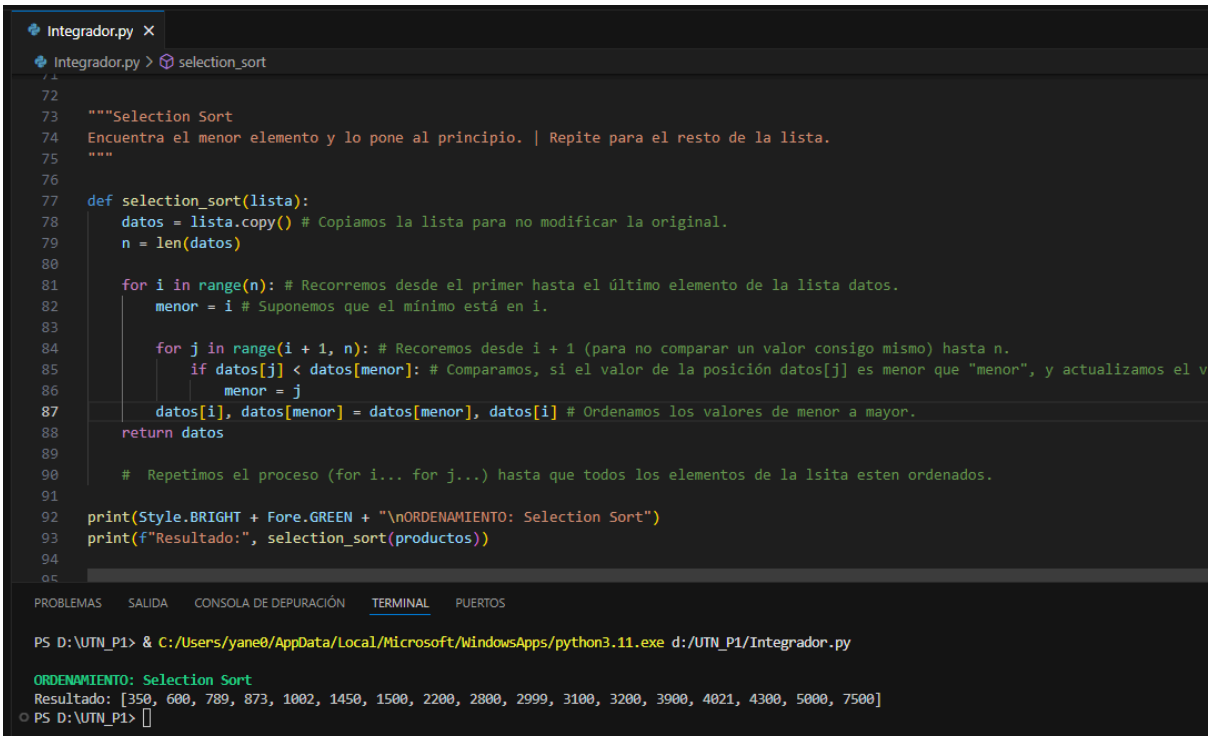
→ ORDENAMIENTO: Bubble Sort



```python
# ALGORITMOS DE ORDENAMIENTO

"""Bubble Sort
Compara elementos adyacentes e intercambia si están en orden incorrecto. | Repite pasadas hasta que no hay más cambios.
"""

def bubble_sort(lista):
    datos = lista.copy()  # Copiamos la lista y la almacenamos en "datos" para no modificar la lista original.
    n = len(datos)

    for i in range(n): # Iteramos el bucle n cantidad veces.
        for j in range(0, n-i-1):
            # Tomamos el valor del elemento en posición j, e indicamos que el bucle recorra desde 0 hasta n (ctdad de valores)
            # - i (nº de iteración del bucle "padre") -1.
            # De esta forma se indica cuántos pares de combinaciones hacer en cada iteración de for i
            if datos[j] > datos[j+1]:
                # Si el valor de j es mayor que el de j + 1 (la posición que le sigue)
                # Intercambiamos los valores.
                datos[j], datos[j+1] = datos[j+1], datos[j]
    return datos

print(Style.BRIGHT + Fore.GREEN + "\nORDENAMIENTO: Bubble Sort")
print(f"Resultado:", bubble_sort(productos))
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

ORDENAMIENTO: Bubble Sort
Resultado: [350, 600, 789, 873, 1002, 1450, 1500, 2200, 2800, 2999, 3100, 3200, 3900, 4021, 4300, 5000, 7500]
PS D:\UTN_P1>
```

→ ORDENAMIENTO: Selection Sort



```python
"""Selection Sort
Encuentra el menor elemento y lo pone al principio. | Repite para el resto de la lista.
"""

def selection_sort(lista):
    datos = lista.copy() # Copiamos la lista para no modificar la original.
    n = len(datos)

    for i in range(n): # Recorremos desde el primer hasta el último elemento de la lista datos.
        menor = i # Suponemos que el mínimo está en i.

        for j in range(i + 1, n): # Recoremos desde i + 1 (para no comparar un valor consigo mismo) hasta n.
            if datos[j] < datos[menor]: # Comparamos, si el valor de la posición datos[j] es menor que "menor", y actualizamos el v
                menor = j
        datos[i], datos[menor] = datos[menor], datos[i] # Ordenamos los valores de menor a mayor.
    return datos

    #  Repetimos el proceso (for i... for j...) hasta que todos los elementos de la lsita esten ordenados.

print(Style.BRIGHT + Fore.GREEN + "\nORDENAMIENTO: Selection Sort")
print(f"Resultado:", selection_sort(productos))
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS D:\UTN_P1> & C:/Users/yane0/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/UTN_P1/Integrador.py

ORDENAMIENTO: Selection Sort
Resultado: [350, 600, 789, 873, 1002, 1450, 1500, 2200, 2800, 2999, 3100, 3200, 3900, 4021, 4300, 5000, 7500]
PS D:\UTN_P1>
```

→ ORDENAMIENTO: Intersection Sort

Integrador.py

```python

"""Insertion Sort
Toma un elemento y lo inserta en su lugar correcto en la parte ya ordenada de la lista.
"""

def insertion_sort(lista):
    datos = lista.copy() # Duplicamos la lista para no afectar a la original.

    for i in range(1, len(datos)): # Recorremos todos los elementos de la lista comenzando en 1.
        valor = datos[i]
        j = i - 1 # Comparamos hacia atrás, desde el valor anterior.

        while j >= 0 and datos[j] > valor: # Si j es mayor que valor
            datos[j + 1] = datos[j] # Reemplazamos j + 1 (que sería la posición de i) por el valor de j (de esta forma de
            j -= 1 # Seguimos hacia la izquierda.
        datos[j + 1] = valor # Colocamos el valor menor en "valor".
    return datos

print(Style.BRIGHT + Fore.GREEN + "\nORDENAMIENTO: Insertion Sort")
print(f"Resultado:", insertion_sort(productos))


# """Quick Sort
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

```
PS D:\UTN_P1> & C:/Users/yane0/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/UTN_P1/Integrador.py

ORDENAMIENTO: Insertion Sort
Resultado: [350, 600, 789, 873, 1002, 1450, 1500, 2200, 2800, 2999, 3100, 3200, 3900, 4021, 4300, 5000, 7500]
PS D:\UTN_P1>
```

→ ORDENAMIENTO: Quick sort

Integrador.py

```python
"""Quick Sort
Divide la lista en menores y mayores que un pivote. | Ordena recursivamente.
"""

def quick_sort(lista):
    # Caso base: si tiene 1 o 0 elementos, ya está ordenada.
    if len(lista) <= 1:
        return lista
    else:
        # Elegimos el primer elemento como pivote
        pivote = lista[0]

        # Creamos dos listas:
        # 'menores' va a contener los elementos menores o iguales al pivote
        # 'mayores' va a contener los elementos mayores al pivote
        menores = [x for x in lista[1:] if x <= pivote] # x for x in... es una forma abreviada de escribir un bucle.
        mayores = [x for x in lista[1:] if x > pivote]  # lista[1:] es una sublista, desde el segundo elemento en adelante.

        # Llamamos recursivamente a quick_sort sobre cada sublista
        # y luego unimos: primero los menores ordenados, luego el pivote, luego los mayores ordenados
        return quick_sort(menores) + [pivote] + quick_sort(mayores)

print(Style.BRIGHT + Fore.GREEN + "\nORDENAMIENTO: Quick Sort")
print(f"Resultado:", quick_sort(productos))
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

```
PS D:\UTN_P1> & C:/Users/yane0/AppData/Local/Microsoft/WindowsApps/python3.11.exe d:/UTN_P1/Integrador.py

ORDENAMIENTO: Quick Sort
Resultado: [350, 600, 789, 873, 1002, 1450, 1500, 2200, 2800, 2999, 3100, 3200, 3900, 4021, 4300, 5000, 7500]
PS D:\UTN_P1>
```

Metodología Utilizada

1. Lectura y síntesis del material brindado por la facultad.
2. Investigación adicional para comprender en profundidad los algoritmos.
3. Implementación en Python de cada algoritmo.
4. Ejecución del mismo caso práctico con todos los algoritmos.
5. Registro de resultados y análisis comparativo.
6. Elaboración del presente informe.

Resultados Obtenidos

- Todos los algoritmos ordenaron correctamente la lista de productos.
- La búsqueda binaria fue más rápida, pero requirió ordenamiento previo.
- La búsqueda lineal fue funcional en la lista original, pero más lenta.
- Se observaron diferencias de rendimiento entre algoritmos de ordenamiento.
- QuickSort y MergeSort fueron los más eficientes para este tamaño de lista.

Conclusiones

Este trabajo permitió comprender cómo aplicar diversos algoritmos según el tipo de problema. El conocimiento de la complejidad computacional es clave para elegir el método adecuado.

Si bien para listas pequeñas las diferencias son sutiles, en entornos reales con millones de datos estas decisiones impactan de forma significativa.

Se destaca la utilidad de la búsqueda binaria combinada con algoritmos de ordenamiento eficientes.

Bibliografía

- KeepCoding. (s.f.). *Algoritmos de búsqueda en Python*. https://keepcoding.io/blog/algoritmos-de-busqueda-en-python/
- 4Geeks. (s.f.). *Algoritmos de ordenamiento y búsqueda en Python*. https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python
- Python Software Foundation. (s.f.). *The Python Standard Library*. https://docs.python.org/3/
- Khan Academy. (s.f.). *Algoritmos*. https://es.khanacademy.org/computing/computer-science/algorithms
- Gbaudino. (s.f.). *Métodos de ordenamiento con Python* [Repositorio GitHub]. https://github.com/GBaudino/MetodosDeOrdenamiento

---

Anexos

- Capturas del programa funcionando.

- Enlace al repositorio de GitHub:

https://github.com/YaneMub/TP-busqueda-ordenamiento-

- Enlace al video explicativo: (por completar)

- Código fuente (búsqueda y ordenamiento aplicado al caso).