UNIVERSIDAD TECNOLOGICA NACIONAL

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN

UNIVERSIDAD TECNOLÓGICA NACIONAL

Trabajo Práctico Integrador

Programación I

“Algoritmos de Búsqueda y Ordenamiento en Python”

Alumnos:

Pablo Sebastián Ventura – (DNI 28.942.841)

Nicolás Humberto Visintin – (DNI 33.599.495)

Comision 23

Profesor Titular

Prof. Nicolás Quirós

Profesora Tutora

Flor Gubiotti

Fecha de Entrega: 09/06/2025

UNIVERSIDAD TECNOLOGICA NACIONAL

Índice

Video Youtube:

https://youtu.be/S3QFeVwDdBg

Drive:

https://drive.google.com/drive/folders/1XTcbF9Lzd3x7c-1r-rX7qQE9Th97awVJ?usp=sharing

Repositorio Github:

https://github.com/PaSe1982/UTN-TUPaD-P1/tree/main/TP%20INTEGRADOR%20PROGAMACION

https://github.com/Nvis88/UTN-TUPaD-P1-2025/tree/main/INTEGRADOR%20Programacion

UNIVERSIDAD TECNOLOGICA NACIONAL

1. Introducción

Este trabajo práctico tiene como objetivo comparar la eficiencia de dos algoritmos de ordenamiento (Bubble Sort y Selection Sort) y dos algoritmos de búsqueda (Lineal y Binaria) aplicados sobre una lista de números de DNI generados aleatoriamente. Se medirá el tiempo de ejecución de cada algoritmo en diferentes tamaños de lista para ilustrar cómo se comportan en escenarios de datos pequeños y grandes.

- **Bubble Sort:** Algoritmo de ordenamiento de complejidad temporal $O(n^2)$ que recorre repetidamente la lista intercambiando pares adyacentes fuera de orden.
- **Selection Sort:** Algoritmo de ordenamiento de complejidad $O(n^2)$ que en cada pasada selecciona el elemento mínimo restante y lo coloca al inicio.
- **Búsqueda Lineal:** Recorre secuencialmente la lista hasta encontrar el elemento buscado; complejidad $O(n)$.
- **Búsqueda Binaria:** Requiere lista ordenada; divide el espacio de búsqueda a la mitad en cada paso; complejidad $O(\log n)$.

UNIVERSIDAD TECNOLOGICA NACIONAL

2. Marco Teórico

Los algoritmos de búsqueda y ordenamiento son fundamentales en el desarrollo de software, ya que permiten localizar información de forma eficiente y organizar datos para su posterior uso. Python ofrece múltiples herramientas nativas para implementar estos algoritmos de manera sencilla y legible.

La búsqueda lineal consiste en recorrer secuencialmente una lista hasta encontrar el elemento deseado. Es fácil de implementar pero poco eficiente para listas grandes. Por otro lado, la búsqueda binaria requiere que la lista esté ordenada previamente y permite encontrar elementos dividiendo el espacio de búsqueda en cada paso, lo que mejora considerablemente la eficiencia (O(log n)).

En cuanto al ordenamiento, uno de los algoritmos más conocidos es el Bubble Sort, que compara elementos adyacentes y los intercambia si están en el orden incorrecto. Aunque no es el más eficiente, es útil para fines educativos. Python también proporciona métodos internos como sort() o sorted(), basados en algoritmos más optimizados como Timsort.

La elección del algoritmo adecuado depende del tipo y tamaño de los datos, así como de los requerimientos de tiempo y rendimiento.

Referencias:
- Python Software Foundation. (2024). Python 3 Documentation. https://docs.python.org/3/
- Sweigart, A. (2019). Automate the Boring Stuff with Python. No Starch Press.

2. Caso Práctico

Nuestro práctico parte de una lista de DNI's, generados en forma aleatoria con "n" cantidad de elementos, dónde luego seleccionamos un número de dicha lista al azar para realizar la búsqueda, y comparamos el tiempo de procesamiento tanto del ordenamiento cómo de la búsqueda.

Este proceso se realizó varias veces, con distinta cantidad de elementos para comparar los tiempos de procesamientos a medida que aumentan los elementos.

Se implementaron las siguientes etapas:

1. Generación aleatoria de DNIs: genera_lista_DNIs(n) crea una lista de "n" DNI's aleatorios, únicos y desordenados.
2. Selección de elemento de búsqueda: dni_aleatorio(lista) elige un DNI al azar de la lista.
3. Ordenamiento con Bubble Sort: ordenamiento_bubble_sort(lista) y medición de tiempo.
4. Ordenamiento con Selection Sort: ordenamiento_selection_sort(lista) y medición de tiempo.
5. Búsqueda Lineal: busqueda_lineal(lista_desordenada, dni) y medición de tiempo.
6. Búsqueda Binaria: busqueda_binaria(lista_ordenada, dni) y medición de tiempo.
7. Comparación de casos: Se repite la medición para tamaños de lista: 2, 5, 10, 50, 100, 500, 1000, 2000 y 4000.

UNIVERSIDAD TECNOLOGICA NACIONAL

4. Metodología Utilizada

- Se definieron funciones en Python con docstrings claros para cada algoritmo.
- Para cada tamaño de lista, se generó la lista y se capturó el tiempo de ejecución usando el módulo time (desde time.time() en segundos).
- Cada medición se realizó de manera independiente y análoga para asegurar consistencia.
- Los resultados se registraron en milisegundos para facilitar la comparación.

UNIVERSIDAD TECNOLOGICA NACIONAL

5. Resultados Obtenidos

A continuación se presentan los tiempos de ejecución obtenidos para cada algoritmo y tamaño de lista, medidos en milisegundos:

n (DNI's)	Tiempo de Ordenamiento (en milisegundos)		Tiempo de Búsqueda (en milisegundos)	
	Bubble Sort	Selection Sort	Búsqueda Lineal	Búsqueda Binaria
2	0,00262	0,00215	0,00095	0,00167
5	0,00334	0,00215	0,00072	0,00143
10	0,00477	0,00358	0,00048	0,00072
50	0,07105	0,03958	0,00095	0,00048
100	0,26608	0,14830	0,00143	0,00167
500	7,81894	4,47488	0,00620	0,00477
1000	58,35295	35,39562	0,03219	0,00596
2000	165,02309	67,25383	0,01884	0,00429
4000	657,58085	270,02621	0,05579	0,00477
8000	2980,22103	1074,62335	0,19670	0,00501

Observaciones:

- Estos valores cambian con cada ejecución del programa, ya que dependen de la lista creada aleatoriamente.

- **Bubble Sort** muestra un crecimiento drástico del tiempo al aumentar n, consistente con su complejidad $O(n^2)$.
- **Selection Sort** es más eficiente en intercambios, presentando tiempos menores para todos los tamaños.

- **Búsqueda Lineal** aumenta linealmente con el tamaño de la lista, aunque sus tiempos siguen siendo muy bajos comparados con el ordenamiento.
- **Búsqueda Binaria** mantiene tiempos casi constantes y muy reducidos, validando su complejidad $O(\log n)$.

6. Conclusiones

Ordenamiento:

Ambos algoritmos de ordenamiento (Bubble Sort y Selection Sort) exhiben un comportamiento cuadrático: al duplicar el tamaño de la lista, el tiempo de ejecución se multiplica por aproximadamente 4.

Para listas muy pequeñas, la diferencia entre Bubble Sort y Selection Sort es marginal; no obstante, Selection Sort suele ser ligeramente más eficiente debido a la menor cantidad de intercambios.

A medida que el tamaño supera algunos cientos de elementos, ambos algoritmos se vuelven poco prácticos, por lo que es recomendable utilizar algoritmos de ordenamiento con complejidad $O(n \log n)$ (por ejemplo, Merge Sort o el Timsort implementado en list.sort() de Python).

Búsqueda:

La búsqueda lineal recorre la lista elemento a elemento con complejidad $O(n)$, por lo que su tiempo crece proporcionalmente al número de elementos; sin embargo, sus tiempos absolutos siguen siendo muy bajos en comparación con los órdenes de magnitud del ordenamiento.

La búsqueda binaria ofrece un rendimiento significativamente superior en listas grandes, con complejidad $O(\log n)$ y tiempos casi constantes, pero requiere que la lista esté previamente ordenada. Este coste adicional de ordenamiento debe tenerse en cuenta si se realizan múltiples búsquedas.

UNIVERSIDAD TECNOLOGICA NACIONAL

7. Bibliografía

- Python Software Foundation. Python 3 Documentation.
- Material de lectura de la clase (Programación I).
- ChatGPT, análisis de datos.

8. Anexos: Funcionamiento del Programa.

1. El programa inicia definiendo una lista llamada “ejecuciones”, que contiene la cantidad de elementos (N° de DNI), las que luego generamos aleatoriamente para el análisis de ordenamiento y búsqueda, en donde seleccionamos un elemento al azar que será el que buscaremos posteriormente:

```
##############################
### - PROGRAMA PRINCIPAL - ###
##############################

# Definimos una lista de ejecuciones con diferentes cantidades de elementos:
ejecuciones = [2, 5, 10, 50, 100, 500, 1000, 2000, 4000, 8000]

print("Elementos \t Bubble Sort \t Selection Sort \t Búsqueda Lineal \t Búsqueda Binaria")
for elementos in ejecuciones:

    # Creamos una lista de "n" cantidad de DNI's:
    lista_desordenada = genera_lista_DNIs(elementos)

    # Seleccionamos un DNI al azar de la lista y muestro por consola:
    dni_a_buscar = dni_aleatorio(lista_desordenada)
```

2. Continuamos con el ordenamiento de los datos de la lista.
    a. Se definen las variables tiempo_inicio y tiempo_fin que representan la hora antes y después del ordenamiento.
    b. Se llama la función ordenamiento_bubble_sort(lista_desordenada) y ordenamiento_selection_sort(lista_desordenada).

```
    ################################
    ### ORDENAMIENTO DE LA LISTA ###
    ################################

    # Ordenamiento por BUBBLE SORT
    tiempo_inicio_BS = time.time() # Inicio del cronómetro
    lista_ordenada_BS = ordenamiento_bubble_sort(lista_desordenada)
    tiempo_fin_BS = time.time() # Fin del cronómetro

    # Ordenamiento por SELECTION SORT
    tiempo_inicio_SS = time.time() # Inicio del cronómetro
    lista_ordenada_SS = ordenamiento_selection_sort(lista_desordenada)
    tiempo_fin_SS = time.time() # Fin del cronómetro
```

3. Continuamos con los algoritmos de búsqueda, lineal y binaria:
    a. Se definen las variables tiempo_inicio y tiempo_fin que representan la hora antes y después de la búsqueda.

b. Se llama a la función busqueda_lineal(lista_desordenada, dni_a_buscar) y busqueda_binaria(lista_ordenada_BS, dni_a_buscar).

```
# BUSQUEDA LINEAL
tiempo_inicio_BL = time.time() # Inicio del cronómetro
posicion_dni_BL = busqueda_lineal(lista_desordenada, dni_a_buscar)
tiempo_fin_BL = time.time() # Fin del cronómetro

# BUSQUEDA BINARIA - Con lista ordenada.
tiempo_inicio_BB = time.time() # Inicio del cronómetro
posicion_dni_BB = busqueda_binaria(lista_ordenada_BS, dni_a_buscar)
tiempo_fin_BB = time.time() # Fin del cronómetro
```

4. Finalmente se imprime por consola un resumen de la información.

```
# Imprimo los resultados de las ejecuciones
print(f"{elementos}\t\t{(tiempo_fin_BS - tiempo_inicio_BS)*1000:.5f}\t\t"
    f"{(tiempo_fin_SS - tiempo_inicio_SS)*1000:.5f}\t\t"
    f"{(tiempo_fin_BL - tiempo_inicio_BL)*1000:.5f}\t\t"
    f"{(tiempo_fin_BB - tiempo_inicio_BB)*1000:.5f}")
```

Funcionamiento Funciones:

1) Esta función permite generar una lista con la cantidad de números de DNI que se desee. Cada número es aleatorio y único dentro de la lista, garantizando que no haya duplicados. Esta lista es la base sobre la cual se aplicarán posteriormente los algoritmos de ordenamiento y búsqueda.

```python
def genera_lista_DNIs(elementos):
    """
    Genera una lista de números de DNI's aleatorios y desordenados.

    Args:
        elementos (int): Cantidad de DNIs a generar.

    Returns:
        list: Lista de DNIs aleatorios y desordenados.
    """
    from random import randint
    lista = []
    for _ in range(elementos):
        dni = randint(10000000, 50000000)  # Genera un DNI entre 10.000.000 y 50.000.000
        while dni in lista:  # Asegura que el DNI sea único
            dni = randint(10000000, 50000000)
        lista.append(dni)
    return lista
```

2) Una vez generada la lista de DNIs, esta función selecciona uno al azar. Ese valor es el que se utilizará como “DNI a buscar” durante las pruebas de los algoritmos de búsqueda (lineal y binaria).

```python
def dni_aleatorio(lista):
    """
    Selecciona un DNI al azar de la lista.

    Args:
        lista (list): Lista de DNIs.

    Returns:
        int: Un DNI seleccionado al azar de la lista.
    """
    from random import choice
    return choice(lista)
```

3) Este procedimiento ordena la lista de DNIs utilizando el algoritmo ***Bubble Sort*** el cual compara elementos adyacentes y los intercambia si están en el orden incorrecto. Aunque no es eficiente para grandes volúmenes.

```python
def ordenamiento_bubble_sort(lista):
    """
    Ordena una lista de DNIs utilizando el algoritmo Bubble Sort.

    Args:
        lista (list): Lista de DNIs a ordenar.

    Returns:
        list: Lista de DNIs ordenada.
    """
    n = len(lista)
    for i in range(n):
        for j in range(0, n-i-1):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]  # Intercambio
    return lista
```

4) Este método de ordenamiento de lista, implementa el algoritmo ***Selection Sort***, que selecciona el elemento más pequeño en cada pasada y lo coloca al principio de la lista. Al igual que Bubble Sort, es útil para mostrar cómo funciona un ordenamiento básico paso a paso.

```python
def ordenamiento_selection_sort(lista):
    """
    Ordena una lista de DNIs utilizando el algoritmo Selection Sort.

    Args:
        lista (list): Lista de DNIs a ordenar.

    Returns:
        list: Lista de DNIs ordenada.
    """
    n = len(lista)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if lista[j] < lista[min_idx]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]  # Intercambio
    return lista
```

5) La función busqueda_lineal, busca un DNI recorriendo toda la lista, uno por uno, hasta encontrar el valor deseado. Esta búsqueda se realiza sobre la lista original, sin necesidad de ordenarla previamente.

```python
def busqueda_lineal(lista, dni):
    """
    Realiza una búsqueda lineal para encontrar un DNI en la lista.

    Args:
        lista (list): Lista de DNIs.
        dni (int): DNI a buscar.

    Returns:
        int: Índice del DNI en la lista, o -1 si no se encuentra.
    """
    for i in range(len(lista)):
        if lista[i] == dni:
            return i
    return -1  # Si no se encuentra el DNI
```

6) Este algoritmo requiere (a diferencia de la búsqueda lineal), que la lista esté ordenada previamente. Funciona dividiendo la lista por la mitad en cada paso, lo cual mejora significativamente el tiempo de respuesta en listas grandes.

```python
def busqueda_binaria(lista, dni):
    """
    Realiza una búsqueda binaria para encontrar un DNI en la lista ordenada.

    Args:
        lista (list): Lista de DNIs ordenados.
        dni (int): DNI a buscar.

    Returns:
        int: Índice del DNI en la lista, o -1 si no se encuentra.
    """
    izquierda = 0
    derecha = len(lista) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == dni:
            return medio
        elif lista[medio] < dni:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    return -1  # Si no se encuentra el DNI
```