Trabajo Práctico – Algoritmos de Búsqueda y Ordenamiento en Python

- Alumnos:
  - Mariano Rodriguez Arce - rodriguezarce.mariano@gmail.com
  - Valentin Rodriguez Arce - valentinn.rodriguez05@gmail.com

- Materia: Programación
- Profesor/a: Nicolás Quirós
- Fecha de Entrega: 09/06/2025

Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

Introducción

El presente trabajo integrador aborda la implementación y análisis de algoritmos de búsqueda y ordenamiento utilizando el lenguaje de programación Python. Se busca comprender cómo distintos algoritmos se comportan frente a listas numéricas con distintos tamaños y configuraciones (aleatoria, ordenada e invertida), evaluando su rendimiento mediante la medición precisa de tiempos de ejecución.

A lo largo del trabajo se exploran algoritmos fundamentales como burbuja, selección, inserción, quicksort y mergesort para el ordenamiento, y búsqueda lineal y binaria para la localización de elementos. Estos algoritmos son aplicados a datos generados de manera controlada, lo que permite comparar de forma clara su eficiencia y comportamiento en distintas condiciones.

El objetivo es aplicar los conceptos teóricos estudiados en clase de manera práctica, fortaleciendo la lógica algorítmica y analizando los resultados obtenidos para comprender cuál algoritmo resulta más conveniente en función del contexto de uso.

Marco Teórico

Un algoritmo es una secuencia finita, ordenada y definida de pasos que permiten resolver un problema o realizar una tarea específica. En términos sencillos, es como una receta lógica que transforma una entrada (input) en una salida (output) siguiendo un conjunto claro de instrucciones.

En el contexto de la programación, los algoritmos constituyen la base sobre la que se construyen funciones, módulos y aplicaciones completas. Cada acción automatizada en un programa —desde filtrar datos hasta realizar cálculos complejos— se basa en la aplicación de uno o más algoritmos.

De acuerdo con Cormen et al. (2009), para que una secuencia de instrucciones pueda considerarse un algoritmo, debe cumplir cinco propiedades esenciales:

- **Entrada**: debe recibir uno o más datos iniciales.
- **Salida**: debe generar al menos un resultado.
- **Claridad**: cada paso debe estar definido de forma precisa.
- **Finitud**: debe finalizar luego de un número limitado de pasos.
- **Eficiencia**: debe usar los recursos (tiempo y memoria) de forma razonable

Estas características permiten evaluar la calidad de un algoritmo en términos de correctitud, claridad y rendimiento, aspectos clave al abordar problemas reales y trabajar con estructuras de datos.

Clasificación general de algoritmos

Los algoritmos pueden clasificarse de diversas formas según el tipo de problema que resuelven, la técnica que emplean o su eficiencia computacional. Esta clasificación permite comprender qué tipo de estrategia es más adecuada para abordar distintos desafíos en programación.

Algoritmos de búsqueda: permiten encontrar un valor específico dentro de una estructura de datos, como una lista o un diccionario.
Algoritmos de ordenamiento: organizan un conjunto de datos según un criterio, por ejemplo, de menor a mayor.
Algoritmos recursivos: se llaman a sí mismos para resolver problemas dividiéndolos en subproblemas más pequeños.
Algoritmos greedy: toman decisiones óptimas en cada paso, sin revisar todas las posibilidades.

Algoritmos de divide y vencerás: dividen el problema en partes, las resuelven y combinan las soluciones.
Algoritmos dinámicos: resuelven problemas complejos almacenando resultados parciales (memorización).

Entre estas categorías, **los algoritmos de búsqueda y ordenamiento** tienen un rol central en la programación. Se aplican en tareas básicas como localizar información, clasificar datos, optimizar consultas o mejorar la experiencia del usuario. Además, sirven como base para algoritmos más complejos presentes en áreas como inteligencia artificial, big data y sistemas de recomendación.

Por su amplia aplicación y por ser fundamentales en la manipulación de estructuras de datos, el estudio de estos algoritmos resulta clave para desarrollar habilidades de lógica algorítmica y análisis de eficiencia. En los siguientes apartados se profundizará en ellos, explicando sus características, ventajas y casos de uso.

Algoritmos de Búsqueda

Los algoritmos de búsqueda son procedimientos diseñados para encontrar un elemento específico dentro de una estructura de datos, como una lista o un arreglo. Su objetivo principal es determinar si el elemento está presente y, en caso afirmativo, ubicar su posición dentro de la estructura.

Estos algoritmos son fundamentales en programación porque permiten acceder a la información de forma eficiente, lo que resulta clave en el desarrollo de sistemas que gestionan grandes volúmenes de datos. La elección del algoritmo de búsqueda adecuado influye directamente en el rendimiento general del programa, especialmente cuando se trabaja con estructuras de datos extensas.

Existen distintos tipos de algoritmos de búsqueda, cada uno con características, ventajas y desventajas particulares. A continuación, se presentan dos de los más utilizados en la práctica:

Búsqueda Lineal

También conocida como búsqueda secuencial, consiste en recorrer la estructura de datos elemento por elemento, desde el inicio hasta el final, comparando cada uno con el valor que se desea encontrar. Si se halla una coincidencia, se detiene el proceso; si no, continúa hasta agotar todos los elementos.

Ventajas:

- Fácil de implementar y comprender.
- No requiere que los datos estén ordenados.

Desventajas:

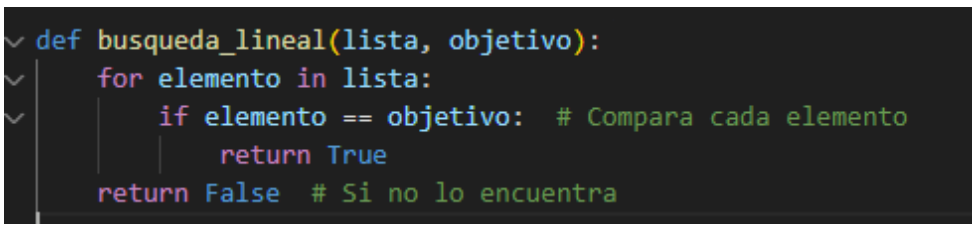
- Puede ser lenta si la lista es muy extensa, ya que se compara uno por uno.

Complejidad temporal:

- Peor caso: $O(n)$
- Mejor caso: $O(1)$

pseudocódigo:

función buscar_lineal(lista, objetivo)
  para cada elemento en lista:
    si elemento == objetivo:
      retornar verdadero
  retornar falso



```
def busqueda_lineal(lista, objetivo):
    for elemento in lista:
        if elemento == objetivo:  # Compara cada elemento
            return True
    return False  # Si no lo encuentra
```

Búsqueda Binaria

La búsqueda binaria es un método eficiente que se aplica sobre estructuras previamente ordenadas. El algoritmo divide el conjunto por la mitad y compara el valor medio con el buscado, descartando la mitad irrelevante en cada paso. Como señalan Cormen et al. (2009), este enfoque reduce significativamente la cantidad de comparaciones, lo que lo convierte en una herramienta clave para optimizar búsquedas en listas extensas.

Ventajas:

- Requiere muchas menos comparaciones que la búsqueda lineal.
- Es mucho más rápida en listas grandes.

Desventajas:

- Solo puede aplicarse si los datos están ordenados de antemano.

Complejidad temporal:

- Peor caso: $O(\log n)$
- Mejor caso: $O(1)$

pseudocódigo:

```
función buscar_binaria(lista_ordenada, objetivo)
  inicio ← 0
  fin ← longitud(lista_ordenada) - 1

  mientras inicio ≤ fin:
    medio ← (inicio + fin) // 2
    si lista_ordenada[medio] == objetivo:
      retornar verdadero
    sino si lista_ordenada[medio] < objetivo:
      inicio ← medio + 1
    sino:
      fin ← medio - 1

  retornar falso
```

```python
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2  # Encuentra el punto medio
        if lista[medio] == objetivo:
            return True
        elif lista[medio] < objetivo:
            inicio = medio + 1  # Busca en la mitad derecha
        else:
            fin = medio - 1  # Busca en la mitad izquierda
    return False
```

Algoritmos de Ordenamiento

Ordenar datos significa reorganizar una colección de elementos según un criterio, como de menor a mayor. Esta operación es clave en programación porque facilita tareas como la búsqueda, el análisis y la visualización de información.

Un conjunto de datos ordenado permite aplicar algoritmos más eficientes, como la búsqueda binaria, y mejora el rendimiento general de los programas. Por eso, aprender a ordenar correctamente es fundamental para optimizar procesos y trabajar con grandes volúmenes de información.

Existen distintos algoritmos de ordenamiento, cada uno con ventajas y desventajas según el contexto. A continuación, se describen algunos de los más representativos:

Ordenamiento Burbuja

El ordenamiento burbuja es uno de los algoritmos más simples e intuitivos para ordenar datos. Consiste en comparar pares de elementos adyacentes y, si están en el orden incorrecto, se intercambian. Este proceso se repite varias veces hasta que toda la lista queda ordenada.

Aunque su implementación es fácil de entender y útil para fines educativos, su eficiencia es baja en comparación con otros métodos, ya que tiene una **complejidad temporal de $O(n^2)$** en el peor y promedio de los casos. Esto significa que su rendimiento se degrada rápidamente con listas grandes.

pseudocódigo:

Para i desde 0 hasta n-1:

  Para j desde 0 hasta n-i-1:

    Si elemento[j] > elemento[j+1]:

      Intercambiar elemento[j] con elemento[j+1]

```python
def burbuja(lista):
    n = len(lista)
    for i in range(n):
        for j in range(0, n - i - 1):
            # Compara elementos adyacentes
            if lista[j] > lista[j + 1]:
                # Intercambia si están en orden incorrecto
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    return lista

# Ejemplo de uso:
numeros = [5, 3, 8, 2, 1]
ordenados = burbuja(numeros)
print(ordenados)  # Resultado: [1, 2, 3, 5, 8]
```

Ordenamiento por Selección

El ordenamiento por selección recorre la lista buscando el elemento más pequeño (o más grande, según el orden deseado) y lo coloca en la posición correcta. Luego repite el proceso con el resto de los elementos.

Aunque mejora la cantidad de intercambios respecto al burbuja, su **complejidad también es $O(n^2)$**, por lo que no es eficiente para listas grandes. Aun así, es útil para entender el funcionamiento de algoritmos de ordenamiento paso a paso.

pseudocódigo:

Para i desde 0 hasta n-1:
  Establecer mínimo = i
  Para j desde i+1 hasta n:
    Si lista[j] < lista[mínimo]:
      mínimo = j
  Intercambiar lista[i] con lista[mínimo]

```python
def seleccion(lista):
    n = len(lista)
    for i in range(n):
        min_idx = i
        # Buscar el índice del valor mínimo restante
        for j in range(i + 1, n):
            if lista[j] < lista[min_idx]:
                min_idx = j
        # Intercambiar el actual con el mínimo encontrado
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista

# Ejemplo de uso:
numeros = [64, 25, 12, 22, 11]
ordenados = seleccion(numeros)
print(ordenados)  # Resultado: [11, 12, 22, 25, 64]
```

Ordenamiento por Inserción

El ordenamiento por inserción es un algoritmo que organiza los elementos de forma similar a cómo se acomodan cartas en la mano: toma cada elemento y lo posiciona en el lugar adecuado dentro de una sublista ya ordenada. Es especialmente eficaz en listas pequeñas o parcialmente ordenadas, donde logra buenos resultados con pocas comparaciones.

Aunque su eficiencia decrece con listas extensas debido a su complejidad temporal de $O(n^2)$, su sencillez lo convierte en una herramienta didáctica fundamental.

Según Knuth (1998), este método destaca por su claridad conceptual y utilidad en situaciones donde los datos ya están casi ordenados o se insertan de forma incremental.

Pseudocódigo:

Para i desde 1 hasta n-1:
  valor_actual = lista[i]
  j = i - 1
  Mientras j >= 0 y lista[j] > valor_actual:
    lista[j + 1] = lista[j]
    j = j - 1
  lista[j + 1] = valor_actual

```python
def insercion(lista):
    for i in range(1, len(lista)):
        valor = lista[i]
        j = i - 1
        # Desplazar los elementos mayores hacia la derecha
        while j >= 0 and lista[j] > valor:
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = valor
    return lista

# Ejemplo:
numeros = [8, 4, 3, 7]
ordenados = insercion(numeros)
print(ordenados)  # Resultado: [3, 4, 7, 8]
```

Quicksort

Quicksort es un algoritmo de ordenamiento muy eficiente que se basa en la estrategia de **divide y vencerás**. Consiste en dividir la lista en partes más pequeñas para ordenarlas por separado. Utiliza un **elemento pivote** para comparar el resto de los elementos: los que son menores al pivote se agrupan a un lado, y los mayores al otro. Luego, se aplica el mismo proceso recursivamente a cada sublista.

Gracias a esta división, Quicksort suele ser más rápido que otros métodos simples, especialmente en listas grandes.
En promedio, tiene una **complejidad temporal de O(n log n)**, pero en el peor de los casos —cuando las particiones son muy desequilibradas— puede degradarse a **O(n^2)**.

pseudocódigo:

función quicksort(lista):
  si la longitud de lista es menor o igual a 1:
    devolver lista
  pivote = último elemento de la lista
  menores = elementos < pivote
  iguales = elementos == pivote
  mayores = elementos > pivote
  devolver quicksort(menores) + iguales + quicksort(mayores)

```python
def quicksort(lista):
    if len(lista) <= 1:
        return lista
    else:
        pivote = lista[-1]
        menores = [x for x in lista if x < pivote]
        iguales = [x for x in lista if x == pivote]
        mayores = [x for x in lista if x > pivote]
        return quicksort(menores) + iguales + quicksort(mayores)

# Ejemplo:
datos = [9, 3, 7, 1, 4]
ordenados = quicksort(datos)
print(ordenados)  # Resultado: [1, 3, 4, 7, 9]
```

Mergesort

Mergesort es un algoritmo de ordenamiento eficiente y estable que se apoya en la estrategia de *divide y vencerás*. Su funcionamiento se basa en dividir recursivamente la lista en mitades hasta que cada sublista contiene un solo elemento, y luego fusionarlas en orden para reconstruir la lista completa de forma ordenada.

Una de sus principales ventajas es que mantiene una complejidad temporal de $O(n \log n)$ en todos los casos (mejor, peor y promedio), lo que lo hace predecible y confiable incluso con grandes volúmenes de datos. Además, al conservar el orden relativo de elementos iguales, es una opción ideal en

contextos donde esa estabilidad es importante, como en ordenamientos secundarios o múltiples criterios.

pseudocódigo:

Si la lista tiene más de un elemento:

Dividir la lista en dos mitades

Aplicar Mergesort a cada mitad

Combinar (fusionar) ambas mitades ordenadamente

```python
def mergesort(lista):
    if len(lista) > 1:
        medio = len(lista) // 2
        izquierda = lista[:medio]
        derecha = lista[medio:]
        # Ordenar recursivamente cada mitad
        mergesort(izquierda)
        mergesort(derecha)
        # Fusionar las dos mitades ordenadas
        i = j = k = 0
        while i < len(izquierda) and j < len(derecha):
            if izquierda[i] < derecha[j]:
                lista[k] = izquierda[i]
                i += 1
            else:
                lista[k] = derecha[j]
                j += 1
            k += 1
        # Agregar lo que quede en cada mitad
        while i < len(izquierda):
            lista[k] = izquierda[i]
            i += 1
            k += 1
        while j < len(derecha):
            lista[k] = derecha[j]
            j += 1
            k += 1
    return lista
# Ejemplo de uso:
numeros = [38, 27, 43, 3, 9, 82, 10]
print("Lista original:", numeros)
mergesort(numeros)
print("Lista ordenada con mergesort:", numeros)  # Resultado: [3, 9, 10, 27, 38, 43, 82]
```

Eficiencia y notación Big-O

La eficiencia de un algoritmo se refiere a los recursos computacionales que requiere, principalmente el tiempo de ejecución y el uso de memoria. Para analizar esta eficiencia de manera objetiva, se utiliza la **notación Big-O**, una herramienta matemática que describe el crecimiento asintótico del tiempo de ejecución en función del tamaño de la entrada (n).

Los algoritmos pueden comportarse de forma diferente según el contexto:

- **Mejor caso**: cuando se resuelve el problema con el mínimo esfuerzo posible.
- **Peor caso**: cuando se necesitan la mayor cantidad de operaciones.
- **Caso promedio**: comportamiento habitual en situaciones generales.

Algunos ejemplos comunes de notación Big-O son:

- **O(1)**: tiempo constante. No cambia con el tamaño de la entrada.
- **O(n)**: tiempo lineal. Crece proporcionalmente al número de elementos.
- **O(log n)**: tiempo logarítmico. Muy eficiente, ya que reduce el problema a la mitad en cada paso.
- **O(n^2)**: tiempo cuadrático. Aparece en algoritmos simples, pero se vuelve ineficiente con grandes volúmenes de datos.
- **O(n log n)**: eficiencia intermedia. Característica de algoritmos como Mergesort y Quicksort (en promedio).

Comprender esta notación es clave para elegir el algoritmo más adecuado, optimizando así el rendimiento general del programa.
Según **Cormen et al. (2009)**, la notación Big-O proporciona una base teórica robusta para comparar algoritmos en términos de escalabilidad y comportamiento frente a diferentes tamaños de entrada.

Comprender los algoritmos de búsqueda y ordenamiento es esencial en el desarrollo de software, ya que permiten manipular grandes volúmenes de datos de manera más eficiente y organizada. Estos algoritmos no solo optimizan el rendimiento de las aplicaciones, sino que también fortalecen habilidades fundamentales como la resolución lógica de problemas, el uso adecuado de estructuras de datos y el análisis de eficiencia computacional.

El caso práctico desarrollado en Python tiene como objetivo aplicar estos conceptos teóricos en situaciones reales, analizando cómo se comportan diferentes algoritmos frente a conjuntos de datos

concretos. Esta articulación entre teoría e implementación práctica busca consolidar una comprensión integral, crítica y aplicada de los algoritmos abordados.

Caso Práctico

El caso práctico desarrollado consiste en la implementación y análisis comparativo de distintos algoritmos de búsqueda y ordenamiento utilizando el lenguaje Python. El objetivo principal es evaluar el rendimiento de cada algoritmo frente a listas de diferentes tamaños y formas, a través de la medición de sus tiempos de ejecución.

Para los algoritmos de ordenamiento, se trabajó con cinco técnicas clásicas: burbuja, selección, inserción, quicksort y mergesort. Para los algoritmos de búsqueda, se compararon los métodos lineal y binario. La evaluación se realizó generando listas numéricas de tres formas distintas:

Aleatoria: números enteros generados al azar.

Ordenada: secuencia ascendente de enteros consecutivos.

Invertida: secuencia descendente de enteros consecutivos.

En cuanto a los tamaños de entrada, se utilizaron listas de 500, 1000 y 2000 elementos para los algoritmos de ordenamiento, y listas de 50, 100, 1000 y 100.000 elementos para los algoritmos de búsqueda. En el caso de la búsqueda, el elemento buscado fue intencionalmente inexistente (−1), lo que fuerza a los algoritmos a ejecutar su recorrido completo (peor caso).

La función medir_tiempo permitió registrar con precisión el tiempo de ejecución en milisegundos de cada algoritmo al procesar las listas generadas. Los resultados fueron impresos por consola y permitieron observar diferencias significativas entre los métodos, especialmente en listas grandes.

Este enfoque permitió identificar claramente las ventajas y desventajas de cada técnica. Por ejemplo, algoritmos como burbuja o inserción mostraron un rendimiento significativamente más bajo en listas extensas, mientras que quicksort y mergesort mantuvieron tiempos estables y eficientes. En cuanto a las búsquedas, se comprobó que la búsqueda binaria es más rápida, pero requiere una lista previamente ordenada.

En síntesis, el caso práctico permitió aplicar de forma efectiva los conceptos teóricos estudiados en clase y reflexionar sobre la importancia de elegir el algoritmo adecuado según el contexto y las características del conjunto de datos.

Metodología Utilizada

Para el desarrollo de este trabajo integrador se siguió una metodología basada en la combinación de investigación teórica, diseño algorítmico y programación práctica. A continuación se detallan los recursos consultados, las etapas del desarrollo y las herramientas empleadas.

Fuentes consultadas

- Introduction to Algorithms – Cormen, Leiserson, Rivest y Stein (2009).
- The Art of Computer Programming – Donald Knuth (1998).
- Sitio web GeeksforGeeks: artículos sobre búsqueda lineal, binaria y algoritmos de ordenamiento.
- Documentación oficial de Python (docs.python.org).
- Apuntes de clase de la cátedra y material teórico proporcionado por la universidad.

Etapas del desarrollo

- Investigación teórica: Se estudiaron los conceptos fundamentales de algoritmos, su clasificación, funcionamiento y complejidad computacional.
- Diseño del programa: Se definieron los algoritmos a implementar y las pruebas a realizar para comparar su rendimiento.
- Implementación en Python: Se programaron cinco algoritmos de ordenamiento y dos de búsqueda, además de una función para medir tiempos de ejecución.
- Pruebas y validaciones: Se generaron listas de diferentes tamaños y formas (aleatoria, ordenada, invertida) para evaluar el comportamiento de cada algoritmo.
- Documentación y análisis: Se organizaron los resultados, se tomaron capturas de pantalla de las salidas del programa y se redactaron las conclusiones en base al análisis comparativo.

Herramientas utilizadas

- Lenguaje: Python 3.11
- Entorno de desarrollo (IDE): Visual Studio Code
- Control de versiones: Git y GitHub (para organización y entrega del proyecto)
- Sistema operativo: Windows

Resultados Obtenidos

Durante la ejecución del programa se observaron comportamientos diferenciados entre los algoritmos de búsqueda y ordenamiento, en función del tamaño y la disposición de los datos de entrada. A continuación se detallan los principales resultados obtenidos:

Algoritmos de ordenamiento

Se utilizaron listas de 500, 1000 y 2000 elementos en tres configuraciones: aleatoria, ordenada e invertida. Los resultados muestran lo siguiente:

- Burbuja, selección e inserción funcionaron correctamente, pero mostraron tiempos de ejecución significativamente más altos en listas grandes, especialmente en las invertidas.
- Quicksort y mergesort ofrecieron los mejores tiempos de ejecución en todos los casos, incluso en listas invertidas o aleatorias, demostrando su eficiencia.
- El orden de los datos influye: los algoritmos simples mejoran ligeramente con listas ordenadas, pero su complejidad sigue siendo alta.

Algoritmos de búsqueda

Se utilizaron listas ordenadas de 50, 100, 1000 y 100000 elementos. El objetivo buscado fue el número -1, garantizando que no esté presente (peor caso).

- La búsqueda lineal funcionó correctamente en todos los tamaños, pero fue progresivamente más lenta en listas más grandes.
- La búsqueda binaria mostró gran eficiencia, con tiempos muy bajos incluso en listas de 100000 elementos, siempre que estén ordenadas.

Consideraciones generales

- Todos los algoritmos fueron implementados de forma manual, sin librerías externas.
- Se confirmó que la búsqueda binaria no puede aplicarse sobre listas no ordenadas, ya que da resultados incorrectos o irrelevantes.
- Las funciones de medición fueron confiables, con resultados consistentes en múltiples ejecuciones.
- No se presentaron errores en la ejecución del programa.
- Enlace a repositorio (GitHub, por ejemplo), si es posible.

Conclusiones

El trabajo permitió aplicar y comparar distintos algoritmos de búsqueda y ordenamiento, observando su rendimiento con listas de diversos tamaños y formas. Se comprobó que los algoritmos simples como burbuja o inserción funcionan bien en listas pequeñas, pero son ineficientes en casos grandes, mientras que quicksort y mergesort ofrecen mejores resultados generales.

En cuanto a la búsqueda, la binaria fue la más eficiente, aunque requiere que los datos estén ordenados, a diferencia de la búsqueda lineal que funciona en cualquier caso, pero con mayor tiempo de ejecución en listas grandes.

En resumen, la experiencia reforzó la importancia de elegir el algoritmo adecuado según el contexto, destacando cómo una buena decisión algorítmica mejora significativamente el rendimiento de un programa.

Bibliografía

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Knuth, D. E. (1998). The Art of Computer Programming: Volume 3 - Sorting and Searching (2nd ed.). Addison-Wesley.

GeeksforGeeks. (n.d.). *Linear search*. https://www.geeksforgeeks.org/linear-search

GeeksforGeeks. (n.d.). *Binary search*. https://www.geeksforgeeks.org/binary-search

Python Software Foundation. (n.d.). *Sorting HOW TO — Python 3.13.0 documentation*. https://docs.python.org/es/3.13/howto/sorting.html

W3Schools. (n.d.). *Python lists and algorithms*. https://www.w3schools.com/python

Programiz. (n.d.). *Python programming examples*. https://www.programiz.com/python-programming/examples