Trabajo Práctico - Búsqueda y Ordenamiento

Alumnos:

Erika Samantha Gonzalez - erikasamanthagonzalez@gmail.com
Karen Yanet Guardia - karen.yanet.guardia@gmail.com

Materia: Programación 1

Profesor: Ariel Enferrel

Fecha de Entrega: 9 de Junio 2025

Índice

1. Introducción

En este trabajo se decidió abordar el tema de los algoritmos de búsqueda y ordenamiento aplicados a un sistema de gestión de notas de estudiantes. La elección surgió porque estos algoritmos son herramientas que ayudan a los programadores, son muy utilizadas tanto en la programación como en las tareas cotidianas como organizar datos o encontrar información dentro de una lista.

La búsqueda y el ordenamiento no solo son parte de los contenidos fundamentales de cualquier carrera de programación, sino que también se usan todo el tiempo en el desarrollo de software real, desde aplicaciones simples hasta sistemas complejos. Entender cómo funcionan y en qué casos conviene usar uno u otro permite escribir programas más eficientes y ordenados.

El objetivo principal del trabajo es poner en práctica los conceptos aprendidos no sólo durante la investigación para el trabajo, sino también durante la cursada, aplicándolos a una situación concreta y útil cargar estudiantes con sus notas, ordenarlos por nombre o por calificación, y buscar a una persona específica dentro del listado. Además, se busca comparar distintos algoritmos para ver cuál funciona mejor en diferentes escenarios, según la cantidad de datos o el tipo de ordenamiento.

En resumen, se trata de un ejercicio práctico que une teoría y programación, y que permite afianzar conocimientos clave para cualquier desarrollador.

2. Marco Teórico

Los algoritmos de búsqueda y ordenamiento se utilizan para localizar elementos dentro de una lista de datos y para organizarlos en un orden específico respectivamente. Mientras que los algoritmos de ordenamiento modifican la lista de datos, los de búsqueda no realizan cambios.

Algoritmos de Búsqueda

La búsqueda es una operación fundamental en programación que se utiliza para encontrar un elemento específico dentro de un conjunto de datos. La capacidad de buscar eficientemente a través de esta información es fundamental.

Los algoritmos de búsqueda intentan resolver el problema:

“Dada una secuencia de valores y un valor, devolver el índice del valor en la secuencia si se encuentra, de no encontrarse el valor en la secuencia señalizarlo apropiadamente.”

Existen varios algoritmos de búsqueda, entre los más comunes se encuentran:

- Búsqueda Lineal: recorre secuencialmente cada elemento de la lista hasta encontrar el objetivo o llegar al final.
- Búsqueda Binaria: un algoritmo que permite encontrar un elemento en una lista ordenada dividiendo el espacio de búsqueda en mitades
- Búsqueda de interpolación: utiliza la distribución de valores para hacer predicciones más inteligentes sobre dónde buscar el elemento.
- Búsqueda de hash: utiliza una función hash que a cada elemento lo convierte en un índice numérico que apunta directamente a la posición donde está almacenado el valor correspondiente.

En este trabajo nos vamos a enfocar en los algoritmos de búsqueda lineal y búsqueda binaria.

El algoritmo de búsqueda lineal recorre cada elemento de la lista hasta encontrar el deseado o llegar al final.

Su función en python sería

```
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

Este algoritmo se vuelve ineficiente ante un tamaño grande de la lista a la que se realiza la búsqueda, ya que la cantidad de comparaciones a realizar es proporcional a la longitud de la lista.

Por otro lado se encuentra el algoritmo de búsqueda binaria, el cual tiene como condición inicial que la lista a la que se realiza la búsqueda tiene que estar ordenada. A partir de esto el algoritmo analiza segmentos más chicos que la lista original y los va descartando hasta llegar al valor buscado. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento buscado con el elemento central. Si el elemento buscado es menor que el central, se descarta la mitad derecha; si es mayor, se descarta la mitad izquierda.

Su función en python sería:

```
def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

Al realizar la búsqueda en una lista ordenada es un algoritmo muy eficiente, ya que en cada paso se descarta la mitad de los elementos a inspeccionar.

Algoritmos de Ordenamiento

Ordenar es el proceso de reorganizar una secuencia de objetos para ponerlos en algún orden lógico. Los algoritmos de ordenamiento son un conjunto de instrucciones que toman una lista como entrada y organizan los elementos en un orden particular.

Los algoritmos de ordenamiento más comunes son:

- Ordenamiento por burbuja (Bubble Sort): es un algoritmo simple que ordena los elementos de menor a mayor o viceversa. Compara cada elemento con su adyacente intercambiando su posición si no están en el orden deseado.
- Ordenamiento por selección (Selection Sort): Uno de los algoritmos más simples. Itera en una lista y selecciona el elemento más pequeño actual y lo cambia al primer lugar.
- Ordenamiento por inserción (Insertion Sort): Es un algoritmo simple que inserta cada elemento de la lista en su posición correcta en la lista ordenada.
- Ordenamiento rápido (Quick Sort): Es un algoritmo eficiente que funciona dividiendo la lista en dos partes y luego ordenando cada parte de forma recursiva.
- Ordenamiento por mezcla (Merge Sort): Es un algoritmo eficiente que funciona dividiendo la lista en dos partes, ordenando cada parte y luego fusionando las dos partes ordenadas

Al elegir cual utilizar va a depender de varios factores como el tamaño de la lista, el tipo de dato y el rendimiento ya que van a diferenciarse en la memoria para ejecutarse o la cantidad de recursos.

En este trabajo se implementó el algoritmo de ordenamiento por burbuja, el cual compara cada elemento con su adyacente que lo hace un algoritmo lento que no es eficiente en listas grandes.

Su función en python sería:

```
def bubbleSort(lista):
    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]
    print(lista)
```

En el programa desarrollado en el caso práctico utilizamos el ordenamiento por burbuja para ordenar por nombre, apellido o nota.

3. Caso Práctico

Como caso práctico decidimos realizar un sistema de gestión de notas de estudiantes, muy completo y fácil de expandir, este sistema es un programa en Python donde se cargan nombres de estudiantes y sus calificaciones.

Para ordenar por nota, por nombre o por apellido usamos el algoritmo Bubble Sort.

```python
#Funcion para ordenar por burbuja - puede ser nota o nombre - por apellido la lista ya esta ordenada por defecto
def bubble_sort(lista_estudiantes, clave, descendente=False):
    n = len(lista_estudiantes)
    for i in range(n):
        for j in range(0, n - i - 1):
            a = lista_estudiantes[j][clave]
            b = lista_estudiantes[j + 1][clave]
            if (a < b and descendente) or (a > b and not descendente):
                lista_estudiantes[j], lista_estudiantes[j + 1] = lista_estudiantes[j + 1], lista_estudiantes[j]
# Mostrar la lista ordenada
    for estudiante in lista_estudiantes:
        print(f"{estudiante['nombre']} {estudiante['apellido']} - Nota: {estudiante['nota']}")
```

Para buscar a un estudiante específico por nombre utilizamos el algoritmo de Búsqueda Binaria.

```python
#Búsqueda binaria por apellido (requiere la lista ordenada por apellido)
def busqueda_binaria(lista_estudiantes, apellido):
    izquierda = 0
    derecha = len(lista_estudiantes) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        apellido_medio = lista_estudiantes[medio]["apellido"]

        if apellido_medio == apellido:
            est = lista_estudiantes[medio]
            print(f"Estudiante encontrado: {est['nombre']} {est['apellido']} - Nota: {est['nota']}")
            return
        elif apellido_medio < apellido:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    print(f"El apellido '{apellido}' no fue encontrado")
```

Y para buscar los estudiantes con una nota específica utilizamos el algoritmo de búsqueda lineal.

```python
#Funcion para busqueda lineal - puede ser por nota o por nombre
def busqueda_lineal(lista_estudiantes, nota):
    estudiantes = []

    for estudiante in lista_estudiantes:
        if estudiante["nota"] == nota:
            estudiantes.append({
                "nombre": estudiante["nombre"],
                "apellido": estudiante["apellido"],
                "nota": estudiante["nota"],
            })

    if estudiantes:
        for e in estudiantes:
            print(f'{e["nombre"]} {e["apellido"]} - Nota: {e["nota"]}')
    else:
        print("No se encontraron estudiantes con esa nota.")

    return estudiantes
```

Y de manera adicional agregamos que permitiera agregar o eliminar estudiantes desde consola y mostrar estadísticas simples: promedio, nota más alta, nota más baja.

```python
#Función para agregar estudiante
def agregar_estudiante(lista_estudiantes):
    nuevo_estudiante = crear_estudiante()
    insercion_ordenada(lista_estudiantes, nuevo_estudiante)


def crear_estudiante():
    nombre = input("\nIngrese el nombre del estudiante: ")
    apellido = input("Ingrese el apellido del estudiante: ")
    nota = float(input("Ingrese la nota: "))
    estudiante = {"nombre": nombre, "apellido": apellido, "nota": nota}
    return estudiante

def insercion_ordenada(lista_estudiantes, nuevo_estudiante):
    apellido_nuevo_estudiante = nuevo_estudiante['apellido']

    for indice in range(len(lista_estudiantes)):
        apellido_estudiante_actual = lista_estudiantes[indice]['apellido']

        if apellido_estudiante_actual.lower() > apellido_nuevo_estudiante.lower():
            lista_estudiantes.insert(indice,nuevo_estudiante)
            break

        if indice == len(lista_estudiantes) - 1:
            lista_estudiantes.append(nuevo_estudiante)
```

Este código aplica la modularización al estar compuesto de tres funciones siendo agregar_estudiante() la principal.

La función eliminar estudiante también utiliza el algoritmo de búsqueda binaria.

```python
#función eliminar_estudiante para eliminar estudiante
def eliminar_estudiante(lista_estudiantes):
    apellido = input("\nIngrese el apellido del estudiante a eliminar: ")
    izquierda = 0
    derecha = len(lista_estudiantes) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        apellido_medio = lista_estudiantes[medio]["apellido"]

        if apellido_medio == apellido:
            estudiante = lista_estudiantes[medio]
            confirmacion = input(f"¿Deseás eliminar a {estudiante['nombre']} {estudiante['apellido']} (s/n)? ").lower()
            if confirmacion == "s":
                lista_estudiantes.pop(medio)
                print("Estudiante eliminado correctamente.")
            else:
                print("Operación cancelada.")
            return
        elif apellido_medio < apellido:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    print(f"No se encontró ningún estudiante con el apellido '{apellido}'.")
```

```python
#Funcion promedio total
def promedio_total(lista_estudiantes):
    suma_notas = 0
    for estudiante in lista_estudiantes:
        suma_notas += estudiante["nota"]

    promedio = suma_notas / len(lista_estudiantes)

    return promedio

#Funcion clasificacion alumnos
def clasificacion_alumnos(lista_estudiantes, estado):
    resultados = []

    for estudiante in lista_estudiantes:
        if estado == "aprobados" and estudiante["nota"] >= 6:
            resultados.append(estudiante)
        elif estado == "desaprobados" and estudiante["nota"] < 6:
            resultados.append(estudiante)

    if resultados:
        for r in resultados:
            print(f'{r["nombre"]} {r["apellido"]} - Nota: {r["nota"]}')
    else:
        if estado == "aprobados":
            print("No hay alumnos aprobados.")
        else:
            print("No hay alumnos desaprobados.")

    return resultados
```

Y a todas estas opciones se accede a partir de un menú donde el usuario interactúa con la consola de comandos

```python
def menu_principal():
    print("\n¡Bienvenido/a a nuestro Sistema de Notas de la Universidad Tecnológica Nacional - Materia Programación 1!")
    print("Elija una de las opciones del menú.")
    print("1. Ver lista de estudiantes (ordenada alfabéticamente por apellido)")
    print("2. Agregar estudiante")
    print("3. Eliminar estudiante")
    print("4. Buscar un estudiante por apellido")
    print("5. Buscar nota")
    print("6. Ordenar lista")
    print("7. Mostrar estadísticas")
    print("8. Salir\n")
```

La estructura principal del programa consiste de un menú donde, dependiendo la opción elegida, se llama a la función correspondiente haciendo uso de la modularidad propia de Python.

```python
def main():
    while True:
        menu_principal()
        opcion = int(input("Ingrese la opcion a trabajar: "))

        if opcion == 1:
            print("\nLista de estudiantes")
            print("-"*20)
            listar_estudiantes(lista_estudiantes)
        elif opcion == 2:
            agregar_estudiante(lista_estudiantes)
            print("\nLista actualizada:")
            print("-"*20)
            listar_estudiantes(lista_estudiantes)
        elif opcion == 3:
            eliminar_estudiante(lista_estudiantes)
            print("\nLista actualizada:")
            print("-"*20)
            listar_estudiantes(lista_estudiantes)
        elif opcion == 4:
            apellido = input("\nIngrese el apellido a buscar: ")
            print("-"*20)
            busqueda_binaria(lista_estudiantes, apellido)
        elif opcion == 5:
            nota = float(input("Ingrese la nota a buscar: "))
```

```
            busqueda_lineal(lista_estudiantes, nota)
        elif opcion == 6:
            print("\nElija una de las opciones del menú.")
            print("1. Ordenar por nota (mayor a menor)")
            print("2. Ordenar por nombre (A-Z)")
            subopcion = int(input("\nIngrese la opcion a trabajar:
"))
            if subopcion == 1:
                print("Lista de estudiantes ordenado por notas (de
mayor a menor)\n")
                print("-"*20)
                bubble_sort(lista_estudiantes, "nota",
descendente=True)
            elif subopcion == 2:
                print("Lista de estudiantes ordenado por nombre
(A-Z)\n")
                print("-"*20)
                bubble_sort(lista_estudiantes, "nombre")
            else:
                print("Ingrese una opcion válida")
        elif opcion == 7:
            print("\nElija una de las opciones del menú.")
            print("1. Promedio total")
            print("2. Alumnos desaprobados")
            print("3. Alumnos aprobados")
            subopcion = int(input("\nIngrese la opcion a trabajar:
"))
            if subopcion == 1:
                print("-"*20)
                print(f"La nota promedio es
{promedio_total(lista_estudiantes):.2f}")
            elif subopcion == 2:
                print("-"*20)
                print("La lista de alumnos desaprobados son:")
                clasificacion_alumnos(lista_estudiantes,
"desaprobados")
            elif subopcion == 3:
                print("-"*20)
```

```
                print("La lista de alumnos aprobados son:")
                clasificacion_alumnos(lista_estudiantes, "aprobados")
            else:
                print("Ingrese una opcion válida")
        elif opcion == 8:
            print("Cerrando programa")
            break
        else:
            print("Ingrese una opcion válida")
```

4. Metodología Utilizada

Para el desarrollo de este trabajo integrador se siguió una metodología estructurada en diferentes etapas: combinando la investigación teórica, el diseño lógico del programa, la implementación en Python y sus pruebas funcionales.

El objetivo fue aplicar los conceptos de búsqueda y ordenamiento en un caso práctico, usando herramientas que nos permitieran comprender su utilidad y rendimiento.

1. Investigación teórica:

   La primera fase de la elaboración del trabajo constó de la investigación de los diferentes tipos de algoritmos de búsqueda y ordenamiento que existen para luego poder elegir a los más adecuados de acuerdo a los requerimientos del programa desarrollado. Consultamos en los vídeos y documentos dejados en la plataforma en la Unidad de Búsqueda y Ordenamiento del campus, leímos y analizamos documentación de otras fuentes adicionales mencionadas en la bibliografía que nos permitió terminar de comprender el funcionamiento, la eficiencia y las aplicaciones de cada algoritmo. Priorizamos el estudio de algoritmos que fueran accesibles desde lo conceptual pero útiles para resolver problemas cotidianos, como Bubble Sort, Búsqueda Lineal y Búsqueda Binaria.

   Analizamos las ventajas, desventajas y complejidades de cada uno, lo que sirvió como base para elegir cuáles implementar en nuestro sistema.

2. Diseño del sistema:

   Una vez hecho esto, pensamos en casos prácticos para realizar y poner en aplicación estos algoritmos. Nos decidimos por simular un sistema que carga nombres de estudiantes y sus calificaciones a la lista de estudiantes ordenadas alfabéticamente, donde también se pudiera ordenar la lista de estudiantes por nota o por nombre, buscar un estudiante específico para consultar su nota, permitiera también agregar y eliminar estudiantes desde la consola y mostrar estadísticas simples como el promedio de la lista, o la lista de los alumnos aprobados y desaprobados. Cada estudiante está representado por un diccionario con su nombre, apellido y nota. Estos diccionarios se almacenan en una lista principal que actúa como base de datos.

El diseño fue pensado para que sea un menú de opciones en la consola y que el usuario pudiera elegir entre las opciones descritas en el párrafo anterior. Dividido en dos archivos (main.py y funciones.py) para mejorar la organización del código.

3. Implementación:

   La implementación se realizó en Python donde programamos los algoritmos seleccionados de forma manual, sin utilizar funciones predefinidas como sort() o index(), para entender en profundidad su lógica.

   Los algoritmos implementados, como expresamos en el marco teórico, fueron:

   -Búsqueda Lineal, utilizada para buscar estudiantes por nota.

   -Búsqueda Binaria, usada para encontrar un estudiante por apellido en una lista previamente ordenada.

   -Bubble Sort, aplicado para ordenar la lista completa según diferentes criterios (nombre y nota).

4. Pruebas y validación:

   Una vez finalizado el desarrollo, realizamos pruebas funcionales para validar cada funcionalidad del menú de opciones de nuestro sistema.

5. Resultados Obtenidos

Los resultados obtenidos al finalizar la ejecución del programa son:

- La creación exitosa de un estudiante y su asignación en el orden que le corresponde según su apellido.
- La eliminación de un estudiante
- La búsqueda exitosa a un estudiante por su apellido al utilizar una funcion de busqueda binaria
- La búsqueda exitosa de una nota con la impresión por consola de los estudiantes que la obtienen a partir de una función de búsqueda lineal.
- El ordenamiento exitoso de los estudiantes por su nota, apellido y nombre utilizando una función de ordenamiento burbuja
- Y la impresión por pantalla de estadísticas como promedio general de notas y estudiantes aprobados o desaprobados.
- Se pudieron implementar los conceptos adquiridos a lo largo de la cursada al utilizar estructuras repetitivas y condicionales y listas a nuestras funciones.

6. Conclusiones

Este trabajo nos permitió comprender en profundidad el funcionamiento y la utilidad de los algoritmos de búsqueda y ordenamiento, aplicándolos a un caso práctico, como lo es nuestro sistema de gestión de notas, demostrando su importancia en el desarrollo de software. A través de la implementación en Python logramos poner en práctica los conceptos teóricos estudiados, cumpliendo los objetivos planteados.

La experiencia de programar algoritmos como búsqueda lineal, búsqueda binaria y ordenamiento por burbuja, nos brindó un entendimiento más concreto de su lógica interna y su impacto en la eficiencia del sistema. Este ejercicio también reforzó nuestras habilidades en programación modular, permitiendo estructurar el código en funciones independientes para facilitar su mantenimiento y escalabilidad.

Además, el uso de herramientas de control de versiones como Git y la gestión de un repositorio en Github nos permitió familiarizarnos con el trabajo colaborativo, al resolver conflictos y coordinar los cambios entre los integrantes del equipo. Al no sólo optimizar el desarrollo del proyecto, sino que también reflejó prácticas esenciales en el entorno profesional del desarrollo de software.

7. Bibliografía

Tecnicatura Universitaria en Programación. Programación I. Unidad Búsqueda y Ordenamiento en programación (2025). (1° ed.). Universidad Tecnológica Nacional.

Tecnicatura Universitaria en Programación. Programación I. Implementación de Algoritmos de Búsqueda Binaria en Python (2025). (1° ed.). Universidad Tecnológica Nacional.

Tecnicatura Universitaria en Programación. Programación I. Vídeos y PDF de la Unidad 7 - Datos Complejos - Diccionarios (2025). (1° ed.). Universidad Tecnológica Nacional.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

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

freeCodeCamp. (2019). *Sorting algorithms explained with examples in Python, Java, and C++.* freeCodeCamp.

https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/

freeCodeCampmp. (2019). *Search Algorithms Explained with Examples in Java, Python, and C++.* freeCodeCamp. https://www.freecodecamp.org/news/search-algorithms-explained-with-examples-in-java-python-and-c/

w3schools. *Data Structures and Algorithms, Bubble Sort.* w3schools.

https://www.w3schools.com/dsa/dsa_algo_bubblesort.php

w3schools. *Data Structures and Algorithms, Linear Search.* w3schools.

https://www.w3schools.com/dsa/dsa_algo_linearsearch.php

w3schools. *Data Structures and Algorithms, Binary Search.* w3schools.

https://www.w3schools.com/dsa/dsa_algo_binarysearch.php

8. Anexos

- Link al repositorio de github: https://github.com/samgnz/integrador-bus-ord
- Link al video de youtube: https://youtu.be/n0l70GtlZg8