Trabajo Integrador -Programación I

- Título del trabajo: "Algoritmos de Búsqueda y Ordenamiento"
- Alumnos:
- Agustin Federico Ras – federicoras19@gmail.com (comisio 11)
- Manuel Angel Dupuy – manueldupuy@gmail.com (comision 25)
- Materia: Programación I
Comision 25
- Profesor: AUS Bruselario, Sebastián
- Tutor: Carbonari, Verónica
Comision 19
- Profesor: Nicolás Quirós
- Tutor: Carla Bustos
- Fecha de Entrega: 09/06/2025

Indice

1. Introducción

En el ámbito de la programación, los algoritmos de búsqueda y ordenamiento son fundamentales para el manejo eficiente de datos. Este trabajo se centra en la implementación práctica de estos algoritmos en Python, permitiendo al usuario interactuar con diferentes métodos para ordenar y buscar información en una lista de estudiantes generada aleatoriamente.

Los objetivos principales son:

- Demostrar la aplicación de algoritmos de ordenamiento y búsqueda en un contexto real.
- Comparar el rendimiento de diferentes algoritmos mediante la medición de tiempos de ejecución.
- Facilitar la comprensión de estos conceptos a través de una interfaz interactiva.

---

2. Marco Teórico

Algoritmos de Ordenamiento

1. Bubble Sort: Compara pares continuos de una lista intercambiandolos si están en el orden incorrecto y repite este proceso hasta llegar al final de la lista. No es el más eficiente en listas grandes comparado con el resto de los algoritmos de ordenamiento .
2. Merge Sort: Primero divide la lista en mitades hasta que cada parte tiene un solo elemento. Luego la va combinando de forma ordenada, de a pares, hasta formar una lista final totalmente ordenada. Es eficiente tanto para listas grandes como para pequeñas.

3. Insertion Sort: Es eficiente en listas pequeñas. Básicamente recorre las lista y en cada paso toma el siguiente elemento y lo inserta en el lugar correcto. Es eficiente cuando se aplica en listas casi ordenadas.
4. Quick Sort: Elige un elemento como pivote y reorganiza la lista colocando a su izquierda los elementos menores y a su derecha los mayores. Luego repite el mismo proceso en cada mitad recursivamente es muy rápido en la práctica aunque puede existir un caso donde el pivote seleccionado sea justo un elemento del extremo de la lista y eso deriva en una menor eficiencia.

Algoritmos de Búsqueda

1. Búsqueda Lineal: Recorre la lista secuencialmente hasta encontrar el elemento.

2. Búsqueda Binaria: Divide repetidamente la lista ordenada por la mitad para encontrar el elemento.

---

3. Caso Práctico

Descripción del Problema

Se desarrolló un programa en Python que permite comparar el rendimiento de diferentes algoritmos de búsqueda y ordenamiento. El sistema genera una lista de estudiantes con nombres y promedios únicos, permite seleccionar criterios de ordenamiento (nombre o promedio) y ofrece diferentes métodos para realizar búsquedas en los datos.

Código Fuente

```
import time
```

```python
import random

import unicodedata



# Trabajo Integrador - Programación I

# Título: Comparador de Algoritmos de Búsqueda y Ordenamiento en Python

# Integrantes: Agustin Federico Ras, Manuel Angel Dupuy

# Fecha de entrega: 07/06/2025



# Algoritmos de ordenamiento



def bubble_sort(arr, key):

    n = len(arr)

    for i in range(n):

        for j in range(0, n - i - 1):

            if arr[j][key] > arr[j + 1][key]:

                arr[j], arr[j + 1] = arr[j + 1], arr[j]

    return arr



def insertion_sort(arr, key):

    for i in range(1, len(arr)):
```

```
        current = arr[i]
        j = i - 1
        while j >= 0 and current[key] < arr[j][key]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = current
    return arr


def quick_sort(arr, key):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2][key]
    left = [x for x in arr if x[key] < pivot]
    middle = [x for x in arr if x[key] == pivot]
    right = [x for x in arr if x[key] > pivot]
    return quick_sort(left, key) + middle + quick_sort(right, key)


def merge_sort(arr, key):
    if len(arr) <= 1:
        return arr
```

```
    mid = len(arr) // 2

    left = merge_sort(arr[:mid], key)

    right = merge_sort(arr[mid:], key)

    return merge(left, right, key)



def merge(left, right, key):

    result = []

    i = j = 0

    while i < len(left) and j < len(right):

        if left[i][key] < right[j][key]:

            result.append(left[i])

            i += 1

        else:

            result.append(right[j])

            j += 1

    result.extend(left[i:])

    result.extend(right[j:])

    return result



# Algoritmos de búsqueda
```

```
def busqueda_lineal(arr, key, target):
    for i in range(len(arr)):
        if normalize_value(arr[i][key]) == normalize_value(target):
            return i
    return -1


def busqueda_binaria(arr, key, target):
    target_norm = normalize_value(target)
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        current = normalize_value(arr[mid][key])
        if current == target_norm:
            return mid
        elif current < target_norm:
            low = mid + 1
        else:
            high = mid - 1
```

```
    return -1


# Función para medir el tiempo de ejecución
def medir_tiempo(algoritmo, arr, key, target=None):
    inicio = time.time()
    if target is not None:
        resultado = algoritmo(arr, key, target)
    else:
        resultado = algoritmo(arr.copy(), key)
    fin = time.time()
    return resultado, fin - inicio


# Normalizador de texto
def normalize_value(value):
    if isinstance(value, str):
        value = unicodedata.normalize('NFKD', value).encode('ASCII',
'ignore').decode('utf-8').lower()
    return value


# Generador de nombres únicos
```

```
def generar_nombres_unicos(cantidad):

    nombres = ["Lucia", "Mateo", "Valentina", "Santiago", "Martina",
"Benjamin", "Camila", "Thiago", "Julieta", "Lautaro", "Agustin", "Manuel"]

    apellidos = ["Gomez", "Fernandez", "Lopez", "Martinez", "Perez",
"Sanchez", "Romero", "Diaz", "Alvarez", "Torres", "Ras", "Dupuy"]

    combinaciones = [f"{n} {a}" for n in nombres for a in apellidos]

    random.shuffle(combinaciones)

    return combinaciones[:cantidad]



# Interfaz por consola

def main():

    tamanio = int(input("\nCantidad de estudiantes a generar (max. 100):
"))

    if tamanio > 100:

        print("Demasiados estudiantes. Maximo permitido: 100.")

        return



    nombres_unicos = generar_nombres_unicos(tamanio)

    promedios_unicos = random.sample(range(1, 101), tamanio)
```

```
estudiantes = [
    {"nombre": nombre, "promedio": promedio}
    for nombre, promedio in zip(nombres_unicos, promedios_unicos)
]


print(f"\nLista generada ({tamanio} estudiantes):")
for e in estudiantes:
    print(f"{e['nombre']}: {e['promedio']}")


print("\nOrdenar por:")
print("1. Nombre")
print("2. Promedio")
clave = input("Elige una opcion (1-2): ")
clave_orden = "nombre" if clave == "1" else "promedio"


print("\nAlgoritmos de Ordenamiento:")
print("1. Bubble Sort")
print("2. Insertion Sort")
print("3. Quick Sort")
print("4. Merge Sort")
```

```
    opcion = input("Elige una opcion (1-4): ")



    algoritmos = {

        "1": bubble_sort,

        "2": insertion_sort,

        "3": quick_sort,

        "4": merge_sort

    }



    if opcion not in algoritmos:

        print("Opcion invalida")

        return



    print("\nOrdenando estudiantes...")

    estudiantes_ordenados, tiempo_ordenamiento =
medir_tiempo(algoritmos[opcion], estudiantes, clave_orden)

    print(f"Tiempo de ordenamiento: {tiempo_ordenamiento:.6f} segundos")

    print("\nLista ordenada:")

    for e in estudiantes_ordenados:

        print(f"{e['nombre']}: {e['promedio']}")
```

```
    print("\nBuscar estudiante por:")
    print("1. Nombre")
    print("2. Promedio")
    opcion_busqueda = input("Elige una opcion (1-2): ")
    clave_busqueda = "nombre" if opcion_busqueda == "1" else "promedio"

    if clave_busqueda == "nombre":
        valor = input("Nombre completo a buscar: ")
    else:
        valor = int(input("Promedio a buscar (1-100): "))

    print("\nMetodo de busqueda:")
    print("1. Lineal")
    print("2. Binaria")
    metodo = input("Elige una opcion (1-2): ")

    if metodo == "1":
        index, tiempo_busqueda = medir_tiempo(busqueda_lineal,
estudiantes_ordenados, clave_busqueda, valor)
```

```
    else:

        index, tiempo_busqueda = medir_tiempo(busqueda_binaria,
estudiantes_ordenados, clave_busqueda, valor)



    print(f"Tiempo de busqueda: {tiempo_busqueda:.6f} segundos")



    if index != -1:

        e = estudiantes_ordenados[index]

        print(f"\nEstudiante encontrado: {e['nombre']}, promedio:
{e['promedio']}")

    else:

        print("\nEstudiante no encontrado.")



if __name__ == "__main__":

    main()
```

Explicación del Funcionamiento

1. Generación de Datos:
    - generar_nombres_unicos() crea combinaciones únicas de nombres y apellidos para los estudiantes.
    - Se generan promedios aleatorios únicos para cada estudiante.
2. Algoritmos de Ordenamiento:

   - Bubble Sort: Implementado en bubble_sort(), compara elementos adyacentes y los intercambia si están en el orden incorrecto.
   - Insertion Sort: En insertion_sort(), construye la lista ordenada insertando un elemento a la vez en su posición correcta.
   - Quick Sort: quick_sort() utiliza recursión y un pivote para dividir la lista en sublistas.
   - Merge Sort: Implementado en merge_sort(), divide la lista en mitades, las ordena recursivamente y luego las fusiona.
3. Algoritmos de Búsqueda:
   - Búsqueda Lineal: busqueda_lineal() recorre la lista secuencialmente hasta encontrar el elemento.
   - Búsqueda Binaria: busqueda_binaria() requiere una lista ordenada y divide repetidamente el espacio de búsqueda a la mitad.
4. Normalización de Texto:
   - normalize_value() estandariza los textos para hacer búsquedas insensibles a mayúsculas y caracteres especiales.
5. Medición de Tiempo:
   - medir_tiempo() registra el tiempo de ejecución de cualquier algoritmo, ya sea de ordenamiento o búsqueda.

Interfaz de Usuario

El programa solicita al usuario:

1. Cantidad de estudiantes a generar (máximo 100).
2. Criterio de ordenamiento (nombre o promedio).
3. Algoritmo de ordenamiento a utilizar.
4. Criterio de búsqueda (nombre o promedio).
5. Método de búsqueda (lineal o binaria).

Muestra los resultados ordenados, los tiempos de ejecución y el resultado de la búsqueda.

Ventajas del Enfoque

- Flexibilidad: Permite ordenar y buscar por diferentes criterios.
- Eficiencia: Incluye algoritmos avanzados como Quick Sort y Merge Sort.
- Robustez: Maneja correctamente búsquedas de texto con normalización.
- Medición: Proporciona datos concretos sobre el rendimiento de cada algoritmo.

---

4. Metodología Utilizada

1. Investigación: Revisión de documentación y recursos académicos sobre algoritmos de búsqueda y ordenamiento.
2. Desarrollo: Implementación de los algoritmos en Python, asegurando su correcto funcionamiento.
3. Pruebas: Ejecución del programa con diferentes tamaños de listas para validar los resultados y los tiempos de ejecución.
4. Documentación: Redacción del informe y preparación del video explicativo.

---

5. Resultados Obtenidos

- El programa generó listas de estudiantes con promedios aleatorios y las ordenó correctamente según el algoritmo seleccionado.
- Se midieron los tiempos de ejecución para cada algoritmo, observando que Quick Sort fue el más eficiente en listas grandes.
- La búsqueda binaria demostró ser significativamente más rápida que la lineal en listas ordenadas, especialmente con grandes volúmenes de datos.

---

6. Conclusiones

Este trabajo permitió aplicar y comparar algoritmos de búsqueda y ordenamiento en un contexto práctico. Se evidenció la importancia de elegir el algoritmo adecuado según el tamaño de los datos y los requisitos de eficiencia. Además, se reforzaron habilidades de programación en Python y el uso de mediciones de tiempo para evaluar el rendimiento.

---

7. Bibliografía

- Python Software https://docs.python.org/3/
- Algoritmo de ordenación https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/
- Algoritmo de ordenamiento https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento
  Algoritmo de búsqueda https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda