Trabajo Práctico Integrador N° 2 – Programación

Alumnos:

Lara Malena Pérez Hernández – larahernandez.edu@gmail.com

Kevin Antonio Pelaez Dioses – pelaezkevin@hotmail.com

Materia: Programación I

Tema: Algoritmos de Búsqueda y Ordenamiento

Profesor: Nicolas Quirós

Tutora: Carla Bustos

Fecha de Entrega: 9 de junio de 2025

Índice

1. Introducción

En el ámbito de la informática y el desarrollo de software, la capacidad de manejar y procesar grandes volúmenes de datos de manera eficiente es un pilar fundamental. Al programar, debemos tener en cuenta la arquitectura de lo que hagamos, ya que el manejo de grandes datos nos lleva a querer simplificar y realizar un procesamiento de los mismos de la manera más eficaz posible. Dentro de este contexto, los algoritmos de búsqueda y ordenamiento emergen como herramientas esenciales que permiten organizar la información para su posterior consulta o análisis.

Hemos elegido este tema porque ofrece una puerta de entrada directa a los principios de la eficiencia algorítmica y el análisis de la complejidad. La elección de un algoritmo adecuado puede marcar una diferencia drástica en el rendimiento de una aplicación, especialmente cuando se trabaja con conjuntos de datos extensos. En la programación, estos algoritmos están por todas partes: los encontramos al organizar listas de productos en una tienda online, al buscar información en una base de datos, o incluso al optimizar rutas en un GPS.

El objetivo principal de este trabajo es llevar a cabo una investigación práctica y aplicada sobre diversos algoritmos de búsqueda y ordenamiento en Python. Pondremos un énfasis especial en la búsqueda binaria por su reconocida eficiencia en datos ordenados, y contrastaremos la complejidad temporal de distintos métodos de ordenamiento para entender su impacto real.

2. Marco Teórico

Este apartado contiene la fundamentación conceptual del tema tratado. Incluiremos definiciones, clasificaciones y cómo se implementan los conceptos en Python.

2.1. Algoritmos de Ordenamiento

Los algoritmos de ordenamiento son procedimientos computacionales que se utilizan para reorganizar una colección de elementos (como números o cadenas de texto) en una secuencia específica, basada en un criterio definido (por ejemplo, de menor a mayor, alfabéticamente, etc.). La eficiencia se mide comúnmente utilizando la Notación **Big O (O)**, que describe cómo el tiempo de ejecución (o el espacio de memoria) del algoritmo crece en relación con el tamaño de la entrada (n).

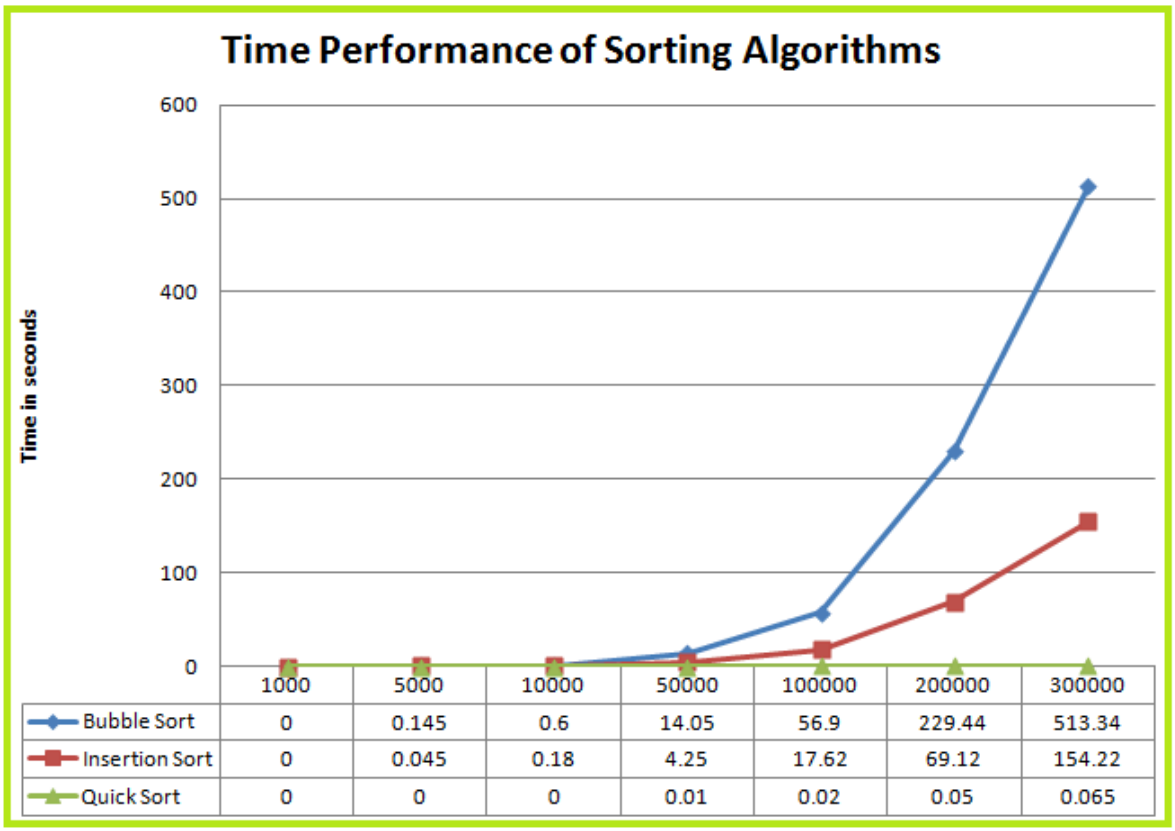
A continuación, describimos algunos de los algoritmos de ordenamiento más comunes y relevantes para nuestro análisis:

2.1.1. Bubble Sort: Este algoritmo compara elementos adyacentes y los intercambia si están en el orden incorrecto. Repite este proceso hasta que todos los elementos "burbujeen" (GeeksforGeeks, 2025) a su posición final, dejando la lista ordenada. Es muy ineficiente para listas grandes, con una complejidad de $O(n^2)$.

2.1.2. Insertion Sort: Este algoritmo de ordenamiento va construyendo una lista ordenada al tomar un elemento de la entrada y lo inserta en su posición correcta dentro de la porción de la lista que ya está ordenada. Es eficiente para listas pequeñas o para aquellas que ya se encuentran parcialmente ordenadas, presentando una complejidad de O(n2) en el caso promedio y peor, pero de O(n) en el mejor caso (cuando la lista ya está ordenada).

2.1.3. Quick Sort: un algoritmo eficiente que utiliza la técnica de "divide y vencerás" (GeeksforGeeks, 2025). Funciona seleccionando un elemento pivot y particionando la lista en dos sublistas: una con elementos menores que el pivot y otra con elementos mayores. Luego, se aplica Quicksort recursivamente a cada sublista. Es muy eficiente para grandes volúmenes de datos, con una complejidad de O(nlogn) en todos los casos.

2.1.4. Comparación de algoritmos de ordenamientos



	1000	5000	10000	50000	100000	200000	300000
Bubble Sort	0	0.145	0.6	14.05	56.9	229.44	513.34
Insertion Sort	0	0.045	0.18	4.25	17.62	69.12	154.22
Quick Sort	0	0	0	0.01	0.02	0.05	0.065

Figura 1. Comparación de algoritmos de ordenamiento (Bubble Sort, Insertion Sort y Quick Sort). Fuente: Workpress: , https://vinayakgarg.wordpress.com/wp-content/uploads/2011/10/comparison_thumb.png?w=650&h=456 (consultado el 9 de junio de 2025).

En la Figura 1, se visualiza el rendimiento de los algoritmos; el **eje Y** representa el tiempo de ejecución (en segundos) y el **eje X**, el número de elementos ordenados. Se observa cómo la complejidad O(n2) de Bubble Sort e Insertion Sort resulta en un crecimiento exponencial del tiempo, mientras que Quick Sort, con su complejidad O(nlogn), mantiene una curva significativamente más plana, demostrando su mayor eficiencia para grandes volúmenes de datos.

2.2. Algoritmos de Búsqueda

2.2.1. Búsqueda Lineal (Secuencial): La búsqueda lineal examina cada elemento de la lista secuencialmente hasta encontrar el valor buscado.

2.2.2. Búsqueda Binaria: La búsqueda binaria es un algoritmo eficiente que funciona exclusivamente en listas ordenadas. Esto quiere decir que la lista tiene que ser ordenada primero.

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.

3. Caso Práctico

Para demostrar la aplicación y la eficiencia de los algoritmos de búsqueda y ordenamiento, hemos desarrollado un programa en Python. Este programa nos permitirá:

1. Implementar los algoritmos de ordenamiento: Bubble Sort y Quick Sort.
2. Implementar los algoritmos de búsqueda: Búsqueda Lineal y Búsqueda Binaria.

3.1 Bubble Sort y Quick Sort

```python
def bubble_sort(arr):
    # El bucle externo recorre la lista para cada elemento
    n = len(arr)
    for i in range(n):
        # El bucle interno compara y mueve el elemento más grande a su posición final
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    #Este algoritmo es O(n^2) por los bucles anidados

def quick_sort(arr):
    # Caso base: si la lista tiene 0 o 1 elemento, ya está ordenada
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0] # Se elige un pivote (elemento inicial en este caso)
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        # Se ordenan recursivamente las sublistas y se concatenan
        return quick_sort(less) + [pivot] + quick_sort(greater)
```

3.2 Búsqueda Lineal y Búsqueda Binaria

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

def busqueda_binaria(lista, objetivo):
    izquierda = 0
    derecha = 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
```

4. Metodología Utilizada

- Investigación teórica sobre algoritmos fundamentales: se investigó de material encontrado de forma online como también de orientación de videos explicativos del tema.
- Implementación en Python con medición de rendimiento: Se calcularon mediciones del tiempo con la funcion time.time() empleado en cada algoritmo para su comparación y análisis.
- Prueba del código: Después de decidir comparar los algoritmos Quicksort y Bubble en cuanto a ordenamiento y en cuanto a búsqueda los algoritmos Lineal y Binaria. Se desarrolló un programa en Python para mostrar los resultados y compararlos.

Al momento de realizar las pruebas finales, se había decidido en un inicio probar con un volumen de 2.000.000 de datos, sin embargo este nivel de datos tenía una demora significativa en su ejecución, por lo tanto se fue tanteando bajar la cantidad de datos a fin de tener un tiempo de ejecución más rápido para la demostración en video.

- Pruebas con diferentes volúmenes de datos: Para la comparación de datos se tomó una muestra de pocos datos (11) contra otra muestra mucho más extensa de datos (20.000).
- Herramientas y recursos utilizados:
    - Visual Studio Code
    - Bibliotecas de Python random y time
    - Github para generar el repositorio digital

5. Resultados Obtenidos

Luego de testear el programa y ver los resultados (en capturas anexadas a la carpeta digital) se llegó a los siguientes conclusiones

1. Ordenamiento:

- Quick Sort demuestra superioridad absoluta en términos de eficiencia
- Bubble Sort solo es viable para datasets muy pequeños o fines educativos
- La diferencia de rendimiento se amplifica exponencialmente con el tamaño de datos

2. Búsqueda:

- Búsqueda Binaria justifica el costo de ordenar los datos previamente
- La escalabilidad de búsqueda binaria la hace ideal para aplicaciones reales
- Búsqueda Lineal mantiene su utilidad cuando los datos no están ordenados

Como dificultad a mencionar, en un principio se pensó en un volumen de 2.000.000 para realizar una prueba de los algoritmos, pero eso trajo (al menos en nuestras computadoras hogareñas) un gran tiempo de ejecución, por el cual se decidió bajar el volumen de datos a 20.000 elementos.

6. Conclusiones

La realización de este trabajo práctico integrador nos proporcionó una comprensión profunda que va mucho más allá de la teoría vista en clase. Lo más impactante fue poder ver realmente el tiempo de ejecución de los algoritmos. Observar que Quicksort ejecuta en milisegundos lo que Bubble Sort tarda varios segundos nos hizo comprender el verdadero impacto de la complejidad algorítmica.

Durante nuestra investigación descubrimos que estos algoritmos realmente se usan en el desarrollo de programas actuales: desde bases de datos que utilizan algoritmos de ordenamiento optimizados para indexar información, hasta motores de búsqueda como Google que emplean variaciones de estos algoritmos para presentar resultados, pasando por plataformas de e-commerce que ordenan productos por precio o relevancia. Esta conexión entre lo académico y lo profesional nos mostró la utilidad práctica de estos algoritmos en el ámbito profesional..

Las principales dificultades que enfrentamos incluyeron el ajuste del volumen de datos de prueba - inicialmente planificamos 2,000,000 elementos pero los tiempos de ejecución resultaron excesivos, por lo que adoptamos un enfoque pragmático reduciendo a 20,000 elementos para mantener la viabilidad de las demostraciones. También tuvimos desafíos técnicos como el manejo correcto de Quicksort (que retorna una nueva lista versus Bubble Sort que modifica in-place), lo cual nos enseñó las diferencias entre tipos de implementación algorítmica.

Este trabajo nos ha motivado a continuar combinando teoría con implementación práctica, desarrollando una habilidad de investigación que nos servirá tanto en proyectos académicos como profesionales.

7. Bibliografía

GeeksforGeeks. (2025, 17 de abril). Bubble Sort Algorithm. Recuperado de **https://www.geeksforgeeks.org/bubble-sort-algorithm/**

GeeksforGeeks. (2025, 17 de abril). Quick Sort Algorithm. Recuperado de **https://www.geeksforgeeks.org/quick-sort-algorithm/**

"Comparación de algoritmos de ordenamiento (Bubble Sort, Insertion Sort y Quick Sort)." Imagen. Google Images. **https://vinayakgarg.wordpress.com/wp-content/uploads/2011/10/comparison_thumb.png?w=650&h=456** (consultado el 9 de junio de 2025).

@dubiouscode4802. (2021, 14 de febrero). Quicksort vs Bubblesort [Video]. YouTube. **https://youtube.com/shorts/UPfOSmX6gRk?si=sbZKhAf_gc0WlO6d**

8. Anexos

Enlace al repositorio: https://github.com/kpelaez/TUPaD_P1_TP_Integrador.git

Enlace al video explicativo: https://youtu.be/vwID-LQWfLQ