UNIVERSIDAD TECNOLOGICA NACIONAL

Trabajo Integrador Programación I

Algoritmo de Búsqueda y Ordenamiento

- **Título del trabajo:** Algoritmos de Búsqueda y Ordenamiento en Python
- **Alumnos:**
    - Maria Celeste Reinaudo – maria.reinaudo@tupad.utn.edu.ar
    - Matias Romano – MatiasAntonioRomano@gmail.com
- **Materia:** Programación I
- **Profesor:** Nicolás Quirós
- **Tutor:** Francisco Quarñolo
- **Fecha de Entrega:** 09/06/2025

Índice

1. Introducción

Este trabajo trata sobre el estudio y la aplicación de algoritmos de búsqueda y ordenamiento usando Python. Elegimos este tema porque es muy importante en la programación y en la solución de problemas con computadoras.

Estos algoritmos son fundamentales porque nos ayudan a organizar grandes cantidades de datos y a encontrar información de forma rápida y eficiente. Con este trabajo, queremos analizar y comparar diferentes métodos, entender cómo funcionan y usarlos en un ejemplo práctico.

El objetivo principal es unir lo que aprendimos en la teoría con su uso en la práctica, viendo cómo se comportan los algoritmos en distintas situaciones y pensando en qué tan bien funcionan.

2. Marco Teórico

Algoritmos de Búsqueda

Un factor importante es la búsqueda, es decir, encontrar uno o varios datos que cumplan con un criterio dado.

- **Búsqueda Lineal:** Recorre una lista elemento por elemento hasta encontrar el objetivo.
    - **Complejidad temporal:** O(n)
- **Búsqueda Binaria:** Divide recursivamente una lista ordenada para encontrar el objetivo.
    - **Complejidad temporal:** O(log n)

Explicación detallada:

Búsqueda lineal: Las búsquedas lineales son las que utilizamos naturalmente, recorriendo uno por uno los elementos hasta encontrar el o los que cumplan con el criterio deseado. Esto usualmente se logra con un bucle que recorre desde el primero (o segundo) elemento comparándolo con otro elemento que cumple con el criterio dado.

Búsqueda binaria: Como la lista ya está ordenada, podemos aprovechar eso y usar un método diferente. En lugar de revisar todos los elementos uno por uno, vamos a ir achicando el grupo de elementos que estamos mirando.

La idea es ir descartando partes de la lista donde sabemos que el valor que buscamos no puede estar:

1. Al principio, miramos toda la lista.

2. Buscamos el valor que está en el medio de la lista. Si ese es el valor que buscamos, devolvemos su posición.

3. Si el valor del medio es más grande que el que buscamos, entonces el número que queremos está en la parte izquierda de la lista. Podemos ignorar la parte derecha.

4. Si el valor del medio es más chico que el que buscamos, entonces el número debe estar en la parte derecha. Podemos ignorar la parte izquierda.

5. Después de descartar una parte, repetimos el proceso con el segmento que queda.

6. Si llega un momento en que ya no quedan elementos por revisar (es decir, el segmento se vuelve vacío), eso quiere decir que el valor no está en la lista.

Algoritmos de Ordenamiento

Existen diversos algoritmos diseñados para ordenar datos, generalmente dentro de arreglos u otras estructuras de datos. Mantener la información organizada es fundamental, ya que resulta esencial para el funcionamiento de la mayoría de las aplicaciones con las que interactuamos a diario.

Métodos de ordenamiento de métodos *iterativos*:

- **Ordenamiento por selección (Selection Sort):** se recorre el vector buscando el valor mínimo entre los elementos no ordenados y se lo intercambia con el elemento que está en la posición actual. Este proceso se repite, avanzando una posición a la vez, hasta que todo el vector queda ordenado. Aunque su lógica es sencilla y fácil de entender, no es de los métodos más eficientes para grandes volúmenes de datos.

- **Ordenamiento por inserción (Insertion Sort):** En este algoritmo, se recorre el vector tomando cada valor y comparándolo con los elementos anteriores, hasta encontrar su posición correcta e insertarlo allí. Aunque esta lógica resulta intuitiva para una persona al ordenar manualmente, su implementación en código es algo más compleja.

- **Ordenación por burbujeo (Bubble Sort):** Este algoritmo de ordenamiento funciona recorriendo el vector y comparando elementos adyacentes. Si el elemento de la derecha no cumple con el criterio deseado (por ejemplo, si es mayor o menor que el de la izquierda, según se requiera), se realiza un intercambio. Este proceso se repite varias veces sobre el vector hasta que todos los elementos quedan ordenados.

Métodos de ordenamiento, basados éstos en un planteo *recursivo* del problema, que nos permitirán obtener el mismo resultado de forma más eficiente.

- **Ordenamiento por mezcla (Merge Sort):** Divide y mezcla de forma ordenada.

  Se basa en el siguiente enfoque:

1. Si la lista está vacía o tiene un solo elemento, ya se considera ordenada, por lo que no requiere procesamiento.

2. En caso contrario, se divide la lista en dos sublistas de tamaños aproximadamente iguales.

3. Cada sublista se ordena de forma recursiva aplicando el mismo método.

4. Finalmente, se combinan ambas sublistas de manera ordenada para formar la lista final.

- **Quick Sort:** Usa un pivote para dividir y ordenar.

  Su popularidad se debe a que, en la práctica, suele ser uno de los métodos más eficientes para ordenar datos reales.

  El funcionamiento básico del algoritmo es el siguiente:

1. Si la lista está vacía o tiene un solo elemento, ya se encuentra ordenada y no requiere más acciones.

2. En caso contrario, se selecciona un elemento de la lista (por ejemplo, el primero) y se lo toma como pivote. A partir de él, se generan tres sublistas: una con los elementos menores al pivote, otra que contiene únicamente al pivote, y una tercera con los elementos mayores o iguales (excluyendo el pivote).

3. Cada una de estas sublistas se ordena de forma recursiva utilizando el mismo procedimiento.

4. Finalmente, se concatenan las tres sublistas ordenadas para obtener la lista final.

Tabla Comparativa

Algoritmo	Mejor caso	Peor caso	Promedio	Estabilidad
Bubble Sort	O(n)	O(n²)	O(n²)	Estable
Selection Sort	O(n²)	O(n²)	O(n²)	Inestable
Insertion Sort	O(n)	O(n²)	O(n²)	Estable
Merge Sort	O(n log n)	O(n log n)	O(n log n)	Estable
Quick Sort	O(n log n)	O(n²)	O(n log n)	Inestable

3. Caso Práctico

Descripción del problema

Se desarrolló una aplicación en Python que permite al usuario cargar una lista de números enteros, elegir un algoritmo de búsqueda u ordenamiento y ver el resultado y el tiempo de ejecución.

Código fuente

https://github.com/Titacele/Integrador_programacion

Decisiones de diseño

Se eligieron algoritmos conocidos por su claridad didáctica. Se evitó el uso de funciones embebidas como sorted() para poder analizar el comportamiento interno de los algoritmos.

Validación del funcionamiento

El programa fue probado con múltiples listas de datos de diferente tamaño y orden.

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 3, 6, 10
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 1
Lista ordenada: [3, 6, 10]
Tiempo de ejecución: 0.000049 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 3, 6, 10
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 2
Lista ordenada: [3, 6, 10]
Tiempo de ejecución: 0.000030 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 5, 9, 90
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 4
Lista ordenada: [5, 9, 90]
Tiempo de ejecución: 0.000025 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 5, 9, 90
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 5
Lista ordenada: [5, 9, 90]
Tiempo de ejecución: 0.000084 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 2
Ingrese números separados por comas: 1, 2, 3, 4, 5, 6, 7, 8
Ingrese el número a buscar: 1
Seleccione método de búsqueda:
1. Lineal
2. Binaria (requiere lista ordenada)
Opción: 2
Elemento encontrado en la posición 0
Tiempo de ejecución: 0.000031 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 1, 4, 6, 8, 2, 3, 5, 7, 9, 10, 15, 13, 14, 11, 12
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 1
Lista ordenada: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Tiempo de ejecución: 0.000086 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 1, 4, 6, 8, 2, 3, 5, 7, 9, 10, 15, 13, 14, 11, 12
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 2
Lista ordenada: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Tiempo de ejecución: 0.000056 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 3, 4, 1, 6, 3, 8, 9, 11, 24, 23, 15, 17, 19, 20, 31, 56, 76
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 4
Lista ordenada: [1, 3, 3, 4, 6, 8, 9, 11, 15, 17, 19, 20, 23, 24, 31, 56, 76]
Tiempo de ejecución: 0.000152 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 1
Ingrese números separados por comas: 3, 4, 1, 6, 3, 8, 9, 11, 24, 23, 15, 17, 19, 20, 31, 56, 76
Seleccione el método de ordenamiento:
1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
Opción: 5
Lista ordenada: [1, 3, 3, 4, 6, 8, 9, 11, 15, 17, 19, 20, 23, 24, 31, 56, 76]
Tiempo de ejecución: 0.000110 segundos
```

```
--- MENÚ PRINCIPAL ---
1. Ordenar lista
2. Buscar elemento
3. Salir
Seleccione una opción: 2
Ingrese números separados por comas: 3, 2, 6, 5, 9, 0, 1, 4
Ingrese el número a buscar: 4
Seleccione método de búsqueda:
1. Lineal
2. Binaria (requiere lista ordenada)
Opción: 1
Elemento encontrado en la posición 7
Tiempo de ejecución: 0.000010 segundos
```

4. Metodología Utilizada

1. Revisión bibliográfica de algoritmos básicos y su análisis.
2. Implementación de cada algoritmo desde cero en Python.
3. Pruebas comparativas de tiempo de ejecución con time y timeit.
4. Uso de GitHub como repositorio compartido.

5. Resultados Obtenidos

- Se comprobó que **Bubble Sort** y **Selection Sort** son ineficientes para listas grandes.
- **Merge Sort** y **Quick Sort** mostraron mejor rendimiento con listas de más de 1.000 elementos.
- La **búsqueda binaria** fue mucho más rápida, pero solo aplicable a listas ordenadas.

6. Conclusiones

El trabajo permitió comprender en profundidad el funcionamiento interno de los algoritmos clásicos. Se evidenció la importancia de elegir el algoritmo correcto según el tipo de problema.

También se reforzó el uso de Git, la documentación colaborativa y el análisis de eficiencia.

7. Bibliografía

- Python Software Foundation. (2024). *Python 3 Documentation*. https://docs.python.org/3/

- Libro: Algoritmo y Programación I - Autor: Varios. Bajo las licencias Creative Commons.

- Geeks for Geeks. (2024). *Data Structures and Algorithms in Python*. https://www.geeksforgeeks.org