# Algoritmos de Búsqueda y Ordenamiento

**Alumnos:** Meshler Andrés - andessemanuel@gmail.com \Molina Martín - martindanielmolina@gmail.com

**Materia:** Programación I

**Profesor/a:** Julieta Trapé

**Fecha de Entrega:** 9 de Junio de 2025

## Índice



## 1. Introducción

En programación, organizar y acceder a los datos de forma eficiente es una tarea clave.
Los algoritmos de búsqueda y ordenamiento son herramientas básicas para lograrlo.
En este trabajo analizamos cómo funcionan, cuándo conviene aplicarlos y qué ventajas tienen.
Además, hacemos foco en la comparación entre búsqueda lineal vs búsqueda binaria, ya que buscamos demostrar las diferencias entre ambos métodos según el tamaño de la lista sobre la cual se realiza la búsqueda.

## 2. Marco Teórico

### Tipos de Ordenamiento

Ordenar datos implica reorganizar una colección de elementos siguiendo alguna regla, como de menor a mayor o alfabéticamente. Aquí les presentamos algunos de los métodos más comunes, con un análisis de sus ventajas y desventajas:

- **Bubble Sort (Ordenamiento de Burbuja):** Este método compara cada par de elementos adyacentes y los permuta si están en el orden equivocado.
    - **Puntos a favor:** Es muy sencillo de entender y de codificar.
    - **Puntos en contra:** Su lentitud lo hace poco práctico para grandes volúmenes de datos.
- **Merge Sort (Ordenamiento por Mezcla):** Este algoritmo funciona dividiendo la lista en partes más pequeñas hasta que solo queda un elemento por segmento. Luego, esos segmentos se van uniendo de forma ordenada hasta reconstruir la lista completa.
    - **Puntos a favor:** Es un método consistente (mantiene el orden relativo de elementos idénticos) y tiene un rendimiento predecible y eficiente en cualquier escenario.
    - **Puntos en contra:** Requiere espacio extra en la memoria para las copias temporales de las sublistas, lo cual podría ser un inconveniente en sistemas con recursos limitados.
- **Quick Sort (Ordenamiento Rápido):** Selecciona un elemento clave, llamado "pivote", y organiza el resto de los elementos en dos grupos: los que son menores que el pivote y los que son mayores. Después, el proceso se repite en cada uno de esos grupos.
    - **Puntos a favor:** Suele ser el más veloz en la práctica para grandes conjuntos de datos, y no necesita tanto espacio de almacenamiento adicional.
    - **Puntos en contra:** Su rendimiento puede bajar considerablemente en casos específicos (aunque esto se evita con buenas elecciones de pivote), y no es un método consistente.

Para nuestro proyecto, aplicamos los tres métodos sobre una misma lista generada con valores aleatorios. Con el objetivo de comparar el rendimiento de cada uno.

**Estrategias de Búsqueda**

Los algoritmos de búsqueda tienen como meta principal localizar un valor específico dentro de una colección. Destacamos dos enfoques:

- **Búsqueda Lineal:** Consiste en revisar cada elemento de la colección uno por uno hasta dar con el valor buscado.
    - **Puntos a favor:** Su implementación es muy directa y no exige que la lista esté ordenada.
    - **Puntos en contra:** Su eficacia disminuye drásticamente con el tamaño de los datos, ya que, en el peor de los casos, podría tener que revisar toda la lista.

- **Búsqueda Binaria:** Este método es sumamente eficiente, pero solo funciona con listas que ya están ordenadas. Opera dividiendo el espacio de búsqueda en mitades sucesivas.
    - **Puntos a favor:** Es mucho más rápida que la búsqueda lineal para grandes volúmenes de datos ya ordenados.
    - **Puntos en contra:** La condición obligatoria de que la lista esté previamente ordenada puede implicar un paso adicional si los datos no lo están.

En este proyecto aplicamos y comparamos ambos métodos de búsqueda. Lo que nos permite realizar análisis y comparaciones entre ambos, para distintos tamaños de lista.

## 3. Implementación Práctica

Desarrollamos un programa en Python que permite generar listas de números aleatorios y aplicar distintos algoritmos de ordenamiento y búsqueda, con el objetivo de comparar su rendimiento en tiempo de ejecución. Está orientado a brindar una herramienta práctica para observar las diferencias de eficiencia entre los métodos de búsqueda y ordenamiento.

### Objetivo principal

El propósito del código es permitir al usuario realizar pruebas de rendimiento entre tres algoritmos de ordenamiento clásicos: Bubble Sort, Merge Sort y Quick Sort. También permite experimentar con dos algoritmos de búsqueda: Búsqueda Lineal y Búsqueda Binaria. De esta manera, se pueden observar de forma práctica los tiempos de ejecución de cada uno, y así comprender sus comportamientos según el tamaño de la lista.

### Funcionamiento general

1. **Generación de lista aleatoria:**
    - Se le solicita al usuario ingresar un tamaño para la lista.
    - A partir de ese número, se genera una lista de valores enteros únicos, de forma aleatoria.
2. **Aplicación de algoritmos de ordenamiento:**
    - Se aplican tres algoritmos sobre una copia de la lista original:
        - **Bubble Sort** (ordenamiento burbuja): Algoritmo simple pero ineficiente para listas grandes.
        - **Merge Sort** (ordenamiento por mezcla): Algoritmo eficiente de tipo "divide y vencerás".
        - **Quick Sort** (ordenamiento rápido): Muy eficiente en la práctica, también basado en "divide y vencerás".

- Se mide y muestra el tiempo de ejecución de cada algoritmo.

3. **Interacción con el usuario para búsqueda:**
   - Se permite elegir entre dos formas de determinar el valor a buscar:
     - **Por posición**: el usuario ingresa una posición dentro del rango de la lista generada (por ejemplo, entre 0 y 99 si la lista tiene 100 elementos). El programa verifica que la posición sea válida y, si lo es, muestra el valor que se encuentra en esa posición.
     - **Por valor**: se busca la posición de un valor ingresado por el usuario.
   - Con el valor a buscar, sobre la lista ordenada y luego desordenada, se realizan:
     - **Búsqueda Lineal**: recorre la lista elemento por elemento.
     - **Búsqueda Binaria**: requiere que la lista esté ordenada previamente.
   - Para ambas, se mide el tiempo de ejecución y se informa si el valor fue encontrado o no.

**Conclusión**

Este programa es útil como herramienta de experimentación y análisis de algoritmos, ya que permite observar cómo varía la eficiencia de cada método dependiendo del tamaño de la lista y del tipo de búsqueda.

## 4. Proceso de Trabajo

Para llevar a cabo este trabajo, seguimos una serie de pasos que nos ayudaron a organizarnos y completar el proyecto:

- Primero, nos dedicamos a recopilar información teórica de fuentes confiables.
- Después, implementamos en Python los algoritmos que decidimos usar.
- Realizamos pruebas con diferentes conjuntos de datos para verificar que todo funcionara bien.
- Registramos los resultados y confirmamos la correcta funcionalidad de lo desarrollado.
- Finalmente, preparamos este informe y todos los materiales adicionales.

## 5. Resultados y Aprendizajes

Este proyecto nos dejó varios puntos clave:

- Nuestro programa fue capaz de ordenar la lista de números, aplicando los distintos métodos de ordenamiento, sin problemas.

- La búsqueda binaria y la lineal tuvieron un comportamiento similar para listas de pequeño tamaño. Pero, a mayor tamaño de la lista se pudo observar una mayor eficiencia por parte del método de búsqueda binaria frente a la búsqueda lineal.
- Pudimos entender a fondo las diferencias en la "complejidad" de los distintos algoritmos que analizamos.
- Confirmamos la importancia de tener los datos ordenados para poder aplicar la búsqueda binaria de forma efectiva.

## 6. Conclusiones Generales

Los algoritmos de búsqueda y ordenamiento son, sin duda, pilares esenciales en el campo de la computación. Estudiarlos no solo nos ayuda a mejorar nuestras habilidades técnicas, sino que también nos da una perspectiva más profunda sobre cómo optimizar la manera en que buscamos y organizamos información.

La búsqueda binaria se destacó particularmente por su eficiencia, aunque siempre con la condición de que la lista debe estar previamente ordenada. Este trabajo nos reforzó la idea de que es fundamental elegir el algoritmo adecuado según el contexto y el tipo de datos que se vayan a manejar.

Para nuestro caso práctico, donde los elementos eran pocos, la combinación de la sencillez de Bubble Sort y de la Búsqueda Lineal fue ideal. Pero cuando aumentamos el tamaño de la lista pudimos observar la caída en la eficiencia de ambos métodos. Resultando más eficientes Merge Sort y Quick Sort y Búsqueda Binaria.

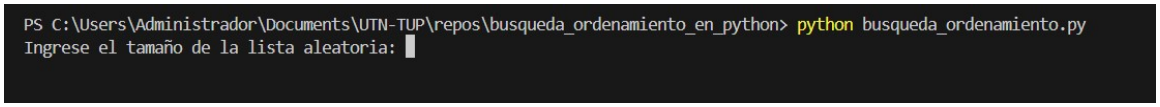
## 7. Referencias Bibliográficas


- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*.
- Documentación oficial de Python: https://docs.python.org/3/library/
- Khan Academy: https://es.khanacademy.org/computing/computer-science/algorithms


## 8. Material Adicional

- Capturas de pantalla de nuestro programa en acción.

  Pide al usuario un tamaño para generar la lista:



Una vez ingresado el tamaño genera la lista con valores no repetidos y aleatorios. Y luego aplica los distintos métodos de ordenamiento. Se observa que Quick Sort realizó el ordenamiento de manera más rápida, seguido de Merge Sort y por último el más lento fue Bubble Sort :

```
PS C:\Users\Administrador\Documents\UTN-TUP\repos\busqueda_ordenamiento_en_python> python busqueda_ordenamiento.py
Ingrese el tamaño de la lista aleatoria: 35000

Lista generada (primeros 10 valores): [27238, 39245, 57073, 46819, 63854, 40220, 15834, 50708, 19818, 6244] ...
Aplicando ordenamientos...
bubble_sort tomó 58.917527 segundos.
Lista ordenada (primeros 10 valores): [3, 5, 8, 10, 11, 13, 14, 15, 17, 18] ...
merge_sort tomó 0.096462 segundos.
Lista ordenada (primeros 10 valores): [3, 5, 8, 10, 11, 13, 14, 15, 17, 18] ...
quick_sort tomó 0.049472 segundos.
Lista ordenada (primeros 10 valores): [3, 5, 8, 10, 11, 13, 14, 15, 17, 18] ...

Buscando con lista ordenada...
--- Búsqueda ---
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

Se determina que el programa busque el elemento de la posición 19000. Ambos métodos de búsqueda encuentran el elemento, pero la búsqueda binaria resultó ser más eficiente:

```
Buscando con lista ordenada...
--- Búsqueda ---
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? p

Ingrese una posición (0 a 34999): 19000
El valor en la posición 19000 es: 38115
Búsqueda lineal: índice 19000, valor 38115, tiempo: 0.001081 s
Búsqueda binaria: índice 19000, valor 38115,tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

Al buscar un elemento que se encuentra en la posición 50 (al comienzo de la lista) no se observan diferencias entre los dos métodos de búsqueda:

```
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? p

Ingrese una posición (0 a 34999): 50
El valor en la posición 50 es: 100
Búsqueda lineal: índice 50, valor 100, tiempo: 0.000000 s
Búsqueda binaria: índice 50, valor 100,tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

Al buscar un elemento que se encuentra en la posición 34900 (al final de la lista) se observa una vez más que la búsqueda binaria fue más rápida:

```
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? p

Ingrese una posición (0 a 34999): 34900
El valor en la posición 34900 es: 69814
Búsqueda lineal: índice 34900, valor 69814, tiempo: 0.001351 s
Búsqueda binaria: índice 34900, valor 69814,tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

Indicamos al programa que realice la búsqueda de un valor que sabemos de antemano que no existe en la lista. Se observa que la búsqueda binaria devuelve el resultado de manera más rápida.

```
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? v

Ingrese el valor a buscar: 16
Búsqueda lineal: el valor 16 no fue encontrado, tiempo: 0.001980 s
Búsqueda binaria: el valor 16 no fue encontrado, tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

Listas no ordenadas:

Sobre la misma lista, pero ahora sin ordenar, indicamos nuevamente al programa que busque el elemento de la posición19000. Observamos que la búsqueda binaria no funciona en listas desordenadas. La búsqueda lineal si funciona, tardando un poco más en devolver el resultado que cuando la lista se encontraba ordenada.

```
Buscando con lista no ordenada...
--- Búsqueda ---
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? p

Ingrese una posición (0 a 34999): 19000
El valor en la posición 19000 es: 10793
Búsqueda lineal: índice 19000, valor 10793, tiempo: 0.002001 s
Búsqueda binaria: el valor 10793 no fue encontrado, tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

Al buscar el elemento ubicado en la posición 50 vemos nuevamente que la búsqueda binaria no funciona por encontrarse la lista desordenada. Por otro lado, vemos que la búsqueda lineal si funciona devolviendo el resultado correcto:

```
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? p

Ingrese una posición (0 a 34999): 50
El valor en la posición 50 es: 67808
Búsqueda lineal: índice 50, valor 67808, tiempo: 0.000000 s
Búsqueda binaria: el valor 67808 no fue encontrado, tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

La misma situación se repite cuando se busca un elemento al final de la lista no ordenada. Solo la búsqueda lineal funciona para listas no ordenadas. Y lo hace en un

tiempo mayor respecto a la misma búsqueda realizada sobre la misma lista, pero ordenada:

```
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? p

Ingrese una posición (0 a 34999): 34900
El valor en la posición 34900 es: 36080
Búsqueda lineal: índice 34900, valor 36080, tiempo: 0.002079 s
Búsqueda binaria: el valor 36080 no fue encontrado, tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

En el caso de la búsqueda de un valor que no existe, la búsqueda lineal responde correctamente tomándole más tiempo que con la lista ordenada. Una vez más el resultado de la búsqueda binaria no es válido ya que requiere que la lista se encuentre ordenada para funcionar correctamente:

```
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ? v

Ingrese el valor a buscar: 16
Búsqueda lineal: el valor 16 no fue encontrado, tiempo: 0.003006 s
Búsqueda binaria: el valor 16 no fue encontrado, tiempo: 0.000000 s
¿Desea determinar el valor a buscar por posición (p), por valor (v), o para salir (s) ?
```

- Enlace a nuestro repositorio en GitHub: ameshler/busqueda_ordenamiento_en_python
- Enlace a video explicativo en GoogleDrive: video