**Algoritmos de Búsqueda y Ordenamiento en Python**

Alumnos:

Balbuena Shirley – Perez Doti Martin

Materia: Programación I

Profesor/a:

Sebastian Bruselario - Veronica Carbonari

Mails institucionales:  
[martin.perez@tupad.utn.edu.ar](mailto:martin.perez@tupad.utn.edu.ar) - [shirley.balbuena@tupad.utn.edu.ar](mailto:shirley.balbuena@tupad.utn.edu.ar)

Fecha de Entrega:

09/06/2025

[Link de repositorio](https://github.com/ShirleyBal/TPIntegradorProgramacionI)

[**Link al video en YouTube**](https://youtu.be/NTh1Q4qFreU)

[**Link al video en Google Drive**](https://drive.google.com/file/d/1hz3aX_zyu5GX6_Be8zkwCUdY55UKRE5O/view?usp=sharing)

**Índice**

1. Introducción

2. Marco Teórico

3. Caso Práctico

4. Metodología Utilizada

5. Resultados Obtenidos

6. Conclusiones

7. Bibliografía

8. Anexos

# 1. Introducción

El presente trabajo aborda el estudio de los algoritmos de búsqueda y ordenamiento, dos pilares fundamentales en la programación y el manejo de datos. La elección de este tema se debe a su aplicabilidad innegable en múltiples contextos informáticos y un pequeño conocimiento previo de los autores, desde la gestión de bases de datos hasta el desarrollo de aplicaciones complejas que requieren rapidez y precisión en el acceso a la información.

En el campo de la programación y Data Analytics, la capacidad de organizar y recuperar datos de forma eficiente es clave para optimizar el rendimiento de los sistemas, pudiendo generar diferencias abismales entre un sistema optimizado y uno que no lo está.

Comprender y aplicar adecuadamente los algoritmos de búsqueda y ordenamiento no solo permite mejorar la eficiencia computacional, sino también desarrollar un pensamiento algorítmico sólido que resulta esencial para la resolución de cualquier tipo de problema.

El objetivo principal de este trabajo es analizar, comparar e implementar distintos algoritmos de búsqueda y ordenamiento en Python, evaluando su rendimiento y aplicabilidad según el contexto, intentando siempre acercarnos lo máximo posible a un desarrollo de sistema ideal. Asimismo, se busca fomentar la comprensión crítica de las diferencias entre los algoritmos más conocidos, como búsqueda lineal y binaria, o bubble sort y quick sort, reconociendo sus ventajas y limitaciones. Esperamos que este enfoque nos permita solidificar los conceptos teóricos estudiados mediante una aplicación práctica orientada a casos reales.

# 2. Marco Teórico

2.1 ¿Qué es la búsqueda?

La **búsqueda** es un proceso mediante el cual se intenta localizar un elemento dentro de una colección de datos. Es una operación fundamental en programación y estructuras de datos, ya que permite acceder, modificar o eliminar elementos de manera eficiente.

Se utiliza la notación O(n) para el peor caso de búsqueda, es decir, a cuanto equivale el tiempo máximo que puede demorar el algoritmo.  
Se utiliza la notación Ω(n) (alternativamente se puede utilizar o(n))para el mejor caso de búsqueda, es decir, a cuanto equivale el tiempo mínimo en condiciones ideales.  
También puede usarse Θ(n), para definir el tiempo promedio de un caso típico.

#### Algoritmos comunes de búsqueda:

* **Búsqueda lineal**
  + Recorre cada elemento de forma secuencial hasta encontrar el objetivo.
  + **Complejidad promedio y peor caso**: O(n)
  + **Mejor caso (Ω)**: Ω(1), si el elemento está en la primera posición.
* **Búsqueda binaria**
  + Solo funciona con listas previamente ordenadas. Divide la lista en mitades sucesivas.
  + **Mejor caso**: Ω(1)
  + **Promedio**: Θ(log n)
  + **Peor caso**: O(log n)
* **Búsqueda por interpolación**
  + Estima la posición del elemento según su valor (sólo útil si los valores están distribuidos uniformemente).
  + **Mejor caso**: Ω(1)
  + **Peor caso**: O(n)
* **Búsqueda hash**
  + Usa una función hash para acceder directamente al valor buscado.
  + **Complejidad promedio**: O(1), **peor caso**: O(n), **mejor caso**: Ω(1)
* **Búsqueda aleatoria**
  + Selecciona elementos al azar hasta encontrar el correcto.
  + No garantiza eficiencia ni evita repeticiones.
  + **Mejor caso**: Ω(1) (si se elige el elemento correcto en el primer intento)
  + **Promedio**: O(n!)
  + **Peor caso**: potencialmente infinito si no se controlan duplicaciones.

2.2 ¿Qué es el ordenamiento?

El **ordenamiento** organiza los datos siguiendo un criterio definido, como de menor a mayor. Ordenar correctamente los datos permite realizar operaciones como búsquedas, comparaciones y agrupamientos de manera mucho más eficiente.

#### Algoritmos comunes de ordenamiento:

* **Bubble Sort**
  + Compara elementos adyacentes e intercambia si están en orden incorrecto.
  + **Mejor caso**: Ω(n) (si se agrega verificación de orden en cada pasada)
  + **Peor caso**: O(n²)
* **Selection Sort**
  + Encuentra el mínimo de la lista y lo coloca al principio.
  + **Mejor y peor caso**: Θ(n²) (no mejora aunque esté ordenada)
* **Insertion Sort**
  + Inserta elementos en la posición correcta dentro de una lista ya ordenada.
  + **Mejor caso**: Ω(n) (si la lista ya está ordenada)
  + **Peor caso**: O(n²)
* **Merge Sort**
  + Divide y conquista: divide la lista en partes, las ordena y luego las fusiona.
  + **Complejidad en todos los casos**: Θ(n log n)
* **Quick Sort**
  + Elige un pivote, separa menores y mayores, y ordena recursivamente.
  + **Mejor caso**: Ω(n log n), **promedio**: Θ(n log n), **peor caso**: O(n²) (si el pivote es mal elegido)
* **BogoSort (ordenamiento aleatorio)**
  + Mezcla la lista al azar y verifica si está ordenada. Repite hasta que lo esté.
  + **Mejor caso**: Ω(n) (si la lista ya está ordenada)
  + **Promedio**: O(n!)
  + **Peor caso**: Tiempo no acotado
  + Se utiliza como ejemplo de lo que **no** se debe hacer en programación eficiente.

2.3 La importancia de los casos extremos

Se decide incluir algoritmos como BogoSort o búsqueda aleatoria debido a que permite visualizar de manera extrema por qué el diseño de algoritmos importa. Son casos útiles para enseñanza, ya que:

* Ilustran la ineficiencia absoluta de algoritmos mal diseñados.
* Muestran la diferencia real entre el mejor y el peor caso, donde el tiempo de ejecucion teóricamente puede estar en cualquier lugar entre 1 e infinito.
* Refuerzan la necesidad de evaluar la entrada antes de elegir un algoritmo.

Por ejemplo, un algoritmo como Merge Sort garantiza eficiencia estable (Θ(n log n)), mientras que BogoSort es impredecible y totalmente impráctico. De forma similar, la búsqueda binaria puede ser excelente, pero inútil si se aplica sobre listas desordenadas.

# 3. Caso Práctico

En esta sección se presenta un caso de implementación y comparación funcional de distintos algoritmos de búsqueda y ordenamiento, incluyendo tanto estrategias eficientes como ineficientes. El objetivo es demostrar las diferencias de rendimiento entre ellos a través de un enfoque experimental utilizando el lenguaje Python.

#### Descripción del problema

Se parte de una lista desordenada de 10 números enteros generados aleatoriamente entre 1 y 100. Se selecciona uno de ellos como "objetivo de búsqueda" para ser localizado mediante distintos métodos. Asimismo, se ordena la lista con varios algoritmos para observar y comparar su comportamiento.

#### Algoritmos implementados

**Búsqueda:**

* Búsqueda lineal
* Búsqueda binaria (requiere lista ordenada)
* Búsqueda aleatoria (ineficiente)

**Ordenamiento:**

* Bubble Sort
* Insertion Sort
* Quick Sort
* BogoSort (ordenamiento aleatorio)

Durante la ejecución el programa muestra la lista desordenada generada aleatoriamente, imprime los resultados de los métodos de búsqueda implementados y luego la ordena con todos los métodos disponibles, incluyendo la cantidad de intentos que fueron necesarios para que BogoSort logre ordenar la lista.

Se decide implementar métodos absurdamente ineficientes para tener un claro ejemplo de malas prácticas de diseño. Si bien el algoritmo BogoSort logra ordenar la lista de forma correcta en cualquier circunstancia es muy ineficiente, siendo normal que tome varios millones de intentos hasta lograr su objetivo, y su tiempo promedio de ejecución incrementa exponencialmente junto con n.

# 4. Metodología Utilizada

**Herramientas empleadas:**

IDE: Se utilizó Visual Studio Code por su integración con Python, autocompletado inteligente y facilidad para ejecutar scripts.

Control de versiones: Se usó Git para el control de versiones y GitHub como plataforma de trabajo colaborativo para almacenar los archivos del proyecto y mantener el historial de cambios.

Link de repositorio: <https://github.com/ShirleyBal/TPIntegradorProgramacionI>

**Pasos seguidos:**

1. Investigación teórica inicial sobre algoritmos de búsqueda y ordenamiento.
2. Implementación de los algoritmos de búsqueda y de ordenamiento previamente investigados, el código fue escrito en Python.
3. Pruebas con listas generadas aleatoriamente.
4. Comparación de resultados.
5. Documentación y análisis final, donde se consolidan todos los hallazgos en el documento presente.

**Fases de prueba:**

Se probaron los algoritmos con listas pequeñas (10 elementos) para observar el comportamiento y facilitar su seguimiento.

Se midieron intentos (en BogoSort y búsqueda aleatoria) y posiciones retornadas para validar la lógica.

**Trabajo colaborativo:**

* La investigación teórica se realizó de forma independiente por ambos integrantes, con el objetivo de lograr una mayor profundidad teórica.
* La parte teórica fue desarrollada por ambos integrantes en conjunto.
* El código fue escrito por un integrante, testeado extensivamente por el otro y validado en conjunto.

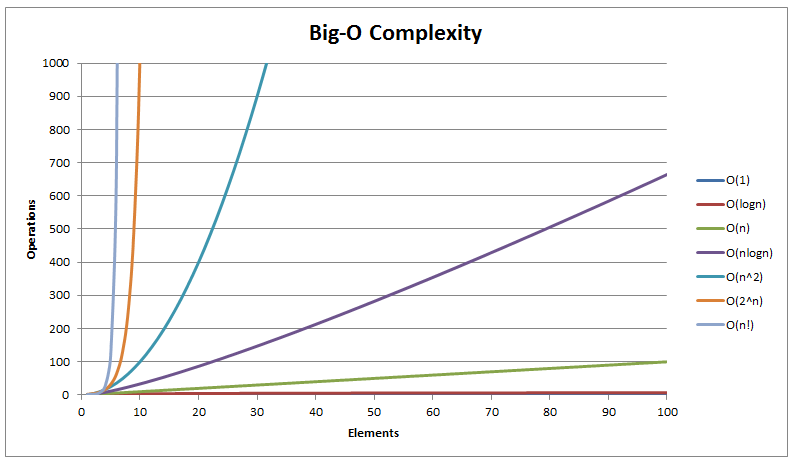
# 5. Resultados Obtenidos

Cuadro comparativo:

| **Algoritmo** | **Tipo** | **Eficiencia Esperada** | **Resultados Obtenidos** |
| --- | --- | --- | --- |
| **Búsqueda lineal** | Búsqueda | O(n) | Siempre encuentra el elemento, tarda más con listas grandes. |
| **Búsqueda Binaria** | Búsqueda | O(log n) (requiere lista ordenada) | Más rápida y eficiente. |
| **Búsqueda Aleatoria** | Búsqueda | O(n) en promedio | Poco eficiente, los resultados pueden variar. |
| **Bubble Sort** | Ordenamiento | O(n²) | Funciona bien en listas cortas. |
| **Insertion Sort** | Ordenamiento | O(n²) | Funciona bien cuando la lista ya está casi ordenada. |
| **Quick Sort** | Ordenamiento | O(n log n) | El más eficiente de todos. |
| **BogoSort** | Ordenamiento | O(n!) | Muy ineficiente; útil solo con listas pequeñas. |

Al ejecutar los distintos algoritmos de búsqueda y ordenamiento sobre las listas de números aleatorios, se observaron las diferencias en su eficacia y eficiencia.

A continuación se adjunta una imagen mostrando los tiempos de ejecución promedio que pueden tomar los distintos metodos de busqueda:



Los algoritmos de búsqueda, Búsqueda Lineal funciona correctamente en todos los casos, aunque su eficiencia disminuye con listas largas, ya que revisa elemento por elemento. Búsqueda Binaria fue mucho más rápida, pero sólo pudo aplicarse a listas previamente ordenadas. Cuando se aplicó correctamente, encontró los elementos en mucho menos tiempo. Búsqueda Aleatoria, en cambio, fue altamente ineficiente y poco confiable, ya que elige posiciones al azar y puede tardar mucho en encontrar el valor, o incluso no hallarlo en un tiempo razonable.

En los algoritmos de ordenamiento, por ejemplo, Bubble Sort mostró un rendimiento bajo con listas desordenadas, ya que realiza múltiples pasadas comparando e intercambiando elementos, lo que genera un alto consumo de tiempo a medida que la lista crece. En cambio, Insertion Sort se desempeñó mejor especialmente cuando la lista ya estaba parcialmente ordenada, ya que requiere menos movimientos en esos casos. Quick Sort, por su parte, fue el algoritmo más eficiente en la mayoría de las pruebas, especialmente con listas grandes, gracias a su enfoque de divide y conquista. Por último, Bogo Sort demostró ser completamente ineficiente y caótico, tardando mucho tiempo en ordenar incluso listas pequeñas, lo que sirvió como ejemplo de un algoritmo no práctico.

Estos resultados permitieron comparar no sólo la eficiencia de los algoritmos en distintos contextos, sino también su adecuación a diferentes tipos de listas. Quedó en evidencia que no todos los algoritmos son apropiados para cualquier situación, y que elegir el correcto depende del tipo de datos y de la operación deseada.

# 6. Conclusiones

El desarrollo del trabajo nos permitió comprender en profundidad la importancia que tienen los algoritmos de búsqueda y ordenamiento desde un contexto de programación y experimentación práctica.

A través de la implementación, prueba y comparación de múltiples algoritmos se evidenció cómo las decisiones y hábitos del programador a cargo de la información influyen directamente en la eficiencia y escalabilidad de un sistema.

Es muy importante reconocer que no todos los algoritmos son adecuados para cualquier situación debido a las grandes diferencias en escalabilidad, requisitos, etc.

La incorporación de algoritmos aleatorios sin ningún tipo de control adicional sirvió como contraste pedagógico, demostrando efectivamente la importancia de la toma de decisiones, llevando al extremo los indicadores de mejor caso, peor caso y caso promedio.

Más allá del caso específico de los métodos de búsqueda y ordenamiento aplicados, el presente proyecto integrador permitió afianzar habilidades de programación en Python, modularización del código (al apartar cada algoritmo en su propia función), trabajo con Git y Github, además de fomentar el trabajo en equipo, habilidades de investigación, documentación y redacción académica.

# 7. Bibliografía

- Python Oficial: <https://docs.python.org/es/3.13/tutorial/datastructures.html>

- Khan Academy: <https://es.khanacademy.org/computing/computer-science/algorithms>

- DataCamp: <https://www.datacamp.com/es/tutorial/linear-search-python>

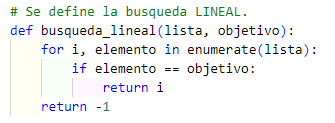
- Freecodecamp: <https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/>

-HackerEarth: <https://www.hackerearth.com/practice/algorithms/sorting/bubble-sort/tutorial/>

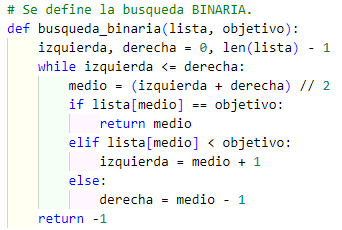
# 8. Anexos

A continuación, se presentan pequeños fragmentos relevantes del código python utilizado;

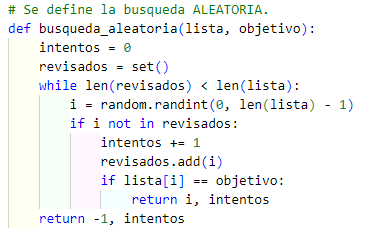
Aquí se define la función de búsqueda lineal:



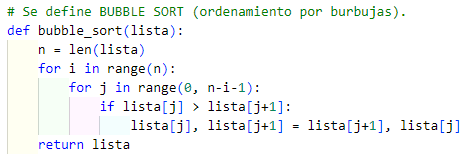
Aquí se define la función de búsqueda binaria:



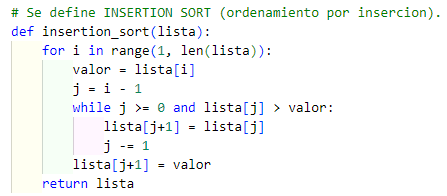
Aquí se define la función de búsqueda aleatoria:



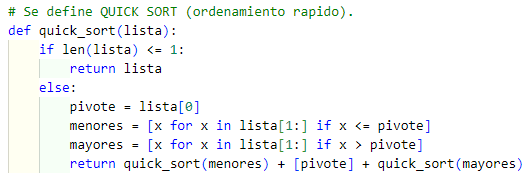
Aquí se define la función de Bubble Sort:



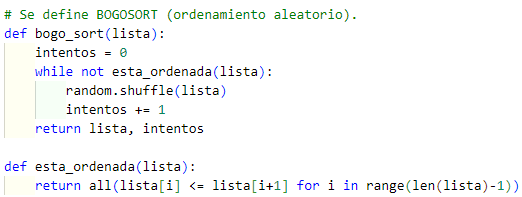
Aquí se define la función de Insertion Sort:



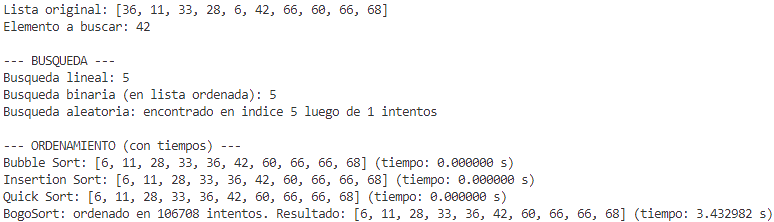
Aquí se define la función de Quick Sort:



Aquí se define la función de Ordenamiento Aleatorio (BogoSort):



Se adjunta una captura de pantalla con un ejemplo de los resultados obtenidos luego de la ejecución del código(con los resultados más bajos para las pruebas aleatorias encontrados en la serie de pruebas realizadas).



El hecho de que el récord de velocidad de BogoSort sea de 100.000 intentos demuestra su increíble ineficiencia, y como observadores suponemos que el número excepcionalmente bajo que se obtuvo se debe a suerte y a que hay 2 instancias del mismo número en la lista, haciendo que estos puedan permutar su lugar sin desordenar la lista (en este caso 66).