# Trabajo Integrador Programación

## Listas, búsquedas y ordenamiento

**Alumnos**

Laura Mendez (comisión 17) y Shai Kohn (comisión 16)

**Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.**

**programación I**

**Docente Titular**

Julieta Trapé y Cinthia Rigoni

**Docente Tutor**

Angel David López y Ana Mutti

Junio de 2025

Índice

## 1-Introducción

Este trabajo práctico integrador tiene como objetivo implementar y analizar el funcionamiento de la búsqueda lineal, listas y ordenamiento utilizando Python. Crearemos una lista con distintos productos de un supermercado a través de un menú interactivo, el usuario podrá seleccionar una categoría según el producto que esté buscando (bebidas, alimentos o herramientas). También implementaremos estrategias de ordenamientos adaptativas (Bubble Sort y Quicksort) y métodos de búsqueda (Lineal y Binaria). Este trabajo ayuda a fortalecer la capacidad de crear soluciones algorítmicas y aplicar estructuras condicionales.

## 2-Marco teórico

### Algoritmos de búsqueda

Un algoritmo de búsqueda es una secuencia de pasos diseñada para encontrar un valor dentro de una estructura de datos. Hay varios tipos de algoritmos de búsqueda, se clasifican según el tipo de datos, la estructura en la que se almacenan y la eficiencia deseada. Los dos más comunes son la búsqueda línea y la búsqueda binaria.

### Búsqueda lineal

Una búsqueda lineal es un algoritmo que localiza un valor concreto dentro de una lista comprobando cada elemento uno por uno, es decir comienza por el primer elemento lo compara con el objetivo de búsqueda y sigue su curso hasta encontrarse con el objetivo. Es sencillo e intuitivo.

No necesita que los datos estén ordenados para funcionar, por lo que se utiliza principalmente en conjuntos de datos sin ordenar. Sin embargo, esto lo hace que no sea tan eficaz como otros algoritmos que requieren datos preclasificados.

**Búsqueda binaria**

Es un algoritmo que encuentra la posición de un valor objetivo dentro de una matriz ordenada dividiendo repetidamente el intervalo de búsqueda por la mitad. Tiene una complejidad temporal de O(log n), lo que la hace mucho más eficaz que la búsqueda lineal. Sin embargo, sólo funciona con datos ordenados. Cuando tu conjunto de datos está ordenado, la búsqueda binaria es casi siempre la mejor opción, porque reduce el espacio de búsqueda a la mitad en cada paso, reduciendo drásticamente el número de comparaciones necesarias para encontrar el objetivo.

**Ordenamiento**

**Bubble Sort**

Es un tipo de algoritmo de ordenamiento que puedes usar para organizar un conjunto de valores en orden ascendente. Si lo deseas, también puedes implementar el ordenamiento de burbuja para ordenar los valores en orden descendente.

Un ejemplo del mundo real de un algoritmo de ordenamiento de burbuja es cómo la lista de contactos en tu teléfono se ordena alfabéticamente. O el ordenamiento de archivos en tu teléfono según el momento en que fueron añadidos.

En este artículo, explicaré todo lo que necesitas saber sobre el algoritmo de ordenamiento de burbuja con algunas infografías que he preparado. Luego, te mostraré ejemplos de código del algoritmo de ordenamiento de burbuja en Python, Java y C++.

Para implementar un algoritmo Bubble sort, se suele escribir una función, y luego un bucle dentro de otro bucle: un bucle interno y un bucle externo.

Supongamos que queremos ordenar una serie de números: **5, 3, 4, 1 y 2**, de manera que queden organizados en orden ascendente.

El ordenamiento comienza en la primera iteración comparando los dos primeros valores. Si el primer valor es mayor que el segundo, el algoritmo empuja el primer valor al índice del segundo valor.

**Quicksort**

QuickSort es un algoritmo de ordenamiento eficiente que utiliza un enfoque de "divide y vencerás" para ordenar una lista de elementos. Fue desarrollado en 1959 por Tony Hoare cuando era estudiante visitante en la Universidad Estatal de Moscú. El algoritmo es altamente utilizado en la industria por su rapidez (tiempo de ejecución promedio), estabilidad con grandes cantidades de datos y uso moderado de memoria.

Funciona mediante la selección de un "pivote" de la lista, la partición de la lista en dos sublistas de elementos menores y mayores que el pivote, y luego la recursión de estos pasos en cada sublista hasta que todos los elementos estén ordenados.

Veamos un ejemplo detallado de cómo ordenar la lista [4, 2, 7, 11, 1, 3, 10, 0, 9, 8, 6, 5] :

Seleccionamos un pivote: por ejemplo, el primer elemento de la lista, el 4 (podría ser cualquiera otra posición, incluso aleatoria).

Partimos la lista en dos sublistas: una con los elementos menores que 4 y otra con los elementos mayores que 4. La lista se divide en [2, 1, 3, 0] y [7, 11, 10, 9, 8, 6, 5].

Ordenamos recursivamente cada sublista hasta que todos los elementos estén ordenados: a. Ordenamos la sublista[2, 1, 3, 0] siguiendo los mismos pasos. Seleccione el primer elemento, 2, como pivote. Particione la lista en [1, 0] y [3]. Ordenar recursivamente cada sublista hasta que

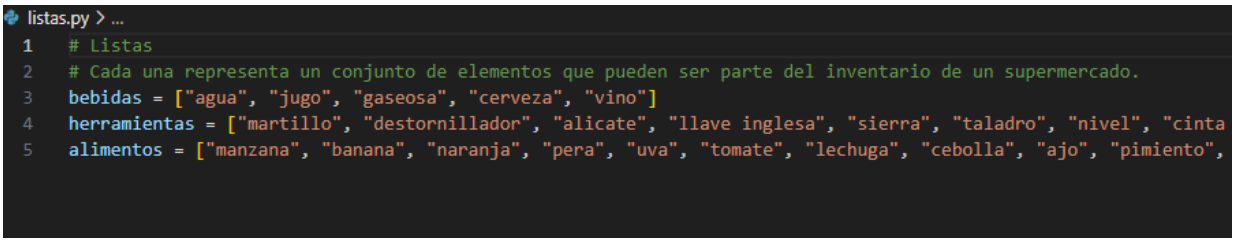
todos los elementos estén ordenados. b. Ordenamos la sublista [7, 11, 10, 9, 8, 6, 5] siguiendo los mismos pasos. Seleccione el primer elemento, 7, como pivote. Dividimos la lista en [4, 6, 5] y [11, 10, 9, 8]. Continuamos ordenando recursivamente cada sublista hasta que todos los elementos estén ordenados.

Una vez que todas las sublistas estén ordenadas, se combinarán entre ellas para formar la lista completa ordenada: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

**3-Caso practico**

Creamos una lista de un supermercado con distintas categorías: Bebidas, herramientas y alimentos

**Código de ejemplo: Listas**




**Ejemplo de búsqueda lineal:**

busqueda.py > busqueda_lineal

```python
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            print(f"El producto '{objetivo}' se encuentra en la posición {i}.")
            return i
    print(f"El producto '{objetivo}' no se encuentra en la lista.")
    return -1

def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            print(f"El producto '{objetivo}' se encuentra en la posición {medio} de la lista.")
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    print(f"El producto '{objetivo}' no se encuentra en la lista.")
    return -1
```

Se define una función llamada busqueda_lineal que recibe dos parámetros:

- lista: una lista de elementos (pueden ser números, palabras, etc.)
- objetivo: el elemento que queremos buscar en la lista.

Luego en la línea 2 se recorre la lista usando un bucle for donde i es el índice.

En la línea 3 en cada iteración, se compara el elemento actual (lista[i]) con el objetivo.

Línea 4 Si se encuentra el objetivo, se muestra un mensaje indicando la posición (índice) en que se encontró.

Termina la función devolviendo el índice donde se encontró el objetivo (línea 5).

Se define una función llamada busqueda_binaria con los mismos parámetros, pero en este caso se requiere que la lista este ordenada para poder funcionar.

Definimos 2 variables izquierda (inicio de la búsqueda) y derecha (final de la búsqueda).

Mientras (While) el rango de búsqueda sea válido (no se crucen los índices), sigue buscando.

Calcula el índice del medio del rango actual (Linea 12)

Si el valor en el medio es igual al objetivo imprime que lo encontró. Y se muestra la posición donde se encontró la búsqueda.

Si el valor en el medio es menor, busca en la mitad derecha (descarta la izquierda) y si el valor en el medio es mayor, busca en la mitad izquierda (descarta la derecha).

Si el bucle termina sin encontrar resultados, imprime que no lo encontró.

Devuelve -1 para indicar que no hubo resultados.

**4-Conclusiones**

El desarrollo de este programa nos ha permitido integrar conceptos fundamentales de programación mediante la interacción de distintas técnicas. Al simular un sistema de gestión de productos de supermercado, hemos demostrado cómo utilizar listas para almacenar datos dinámicamente y cómo aplicar distintos algoritmos de ordenamiento y búsqueda para mejorar la eficiencia en la gestión de información.

El uso del Bubble Sort permitió ilustrar un algoritmo de ordenamiento sencillo, fácil de entender y de implementar. Por otro lado, el Quicksort mostró su superioridad en cuanto a rendimiento, al ser un algoritmo de ordenamiento más avanzado y adecuado para listas extensas.

En cuanto a la búsqueda, la lineal resultó útil en escenarios donde los datos no están ordenados, ya que permite encontrar elementos sin requerir un orden previo. En contraste, la búsqueda binaria, mucho más eficiente, demostró su utilidad en listas ordenadas, destacando la importancia de aplicar el algoritmo correcto según el contexto de los datos.

Finalmente, el menú interactivo ofreció una interfaz amigable que permitió al usuario explorar las funcionalidades del programa de manera intuitiva, reforzando el aprendizaje de conceptos como la entrada del usuario, estructuras condicionales, bucles y funciones.

Este proyecto no solo fortaleció el conocimiento técnico en programación, sino que también permitió comprender mejor la importancia de la selección de algoritmos adecuados para resolver problemas reales de forma eficiente.

## 5-Referencias

Link de pagina (https://www.datacamp.com/es/tutorial/linear-search-python, s.f.)

(Programador web Valencia, s.f.)

*Tecnicatura Universitaria en Programación. Programación I.*

*Junio……* (2025). (1° ed.). Universidad Tecnológica Nacional.

*Link del repositorio (https://github.com/Shaikohn/Trabajo-Integrador-TUPaD-de-Kohn-y-Mendez)*

*Link del video en Youtube (https://www.youtube.com/watch?v=_ZAKUBSOmBo)*