Trabajo Práctico Integrador:

Algoritmos de búsqueda y ordenamiento en Python

Alumnos: Brian Gutierrez Colque, Franco Leonel Herrera

Materia: Programación I

Profesor: Ariel Enferrel

Fecha de entrega: 09/06/2025

Índice

1. Introducción

Los algoritmos de búsqueda y ordenamiento son herramientas fundamentales en el mundo de la programación. Estos permiten manejar datos de manera eficiente y efectiva. En un entorno donde la información se genera a ritmo vertiginoso, comprender estos algoritmos no solo es útil, sino esencial para optimizar el rendimiento de cualquier aplicación.

Este trabajo buscará abordar los conceptos básicos de estos algoritmos, explorando su funcionamiento interno y su puesta en práctica en el desarrollo de software.

2. Marco teórico

Algoritmos de búsqueda

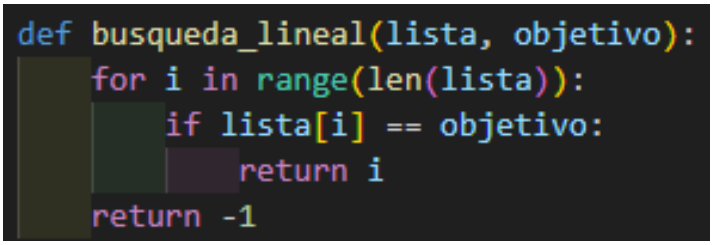
Un algoritmo de búsqueda es un conjunto de instrucciones diseñadas para encontrar un elemento específico dentro de una colección de datos. Dependiendo del tipo de estructura en la que se busque y la naturaleza del dato, existen diferentes algoritmos adecuados para cada situación. Estos algoritmos son comunes cuando se necesita encontrar el menor o mayor elemento (cuando se puede establecer un orden), o buscar el índice de un elemento determinado. Son usados por casi todas las aplicaciones.

Tipos comunes de algoritmos de búsqueda

Búsqueda Lineal

Es el método más sencillo. Consiste en recorrer secuencialmente todos los elementos hasta encontrar el objetivo o determinar que no está presente.

Ejemplo en Python:



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

Este código es funcional pero poco eficiente, tardaría mucho tiempo en buscar un valor dentro de una lista muy larga.

Búsqueda Binaria

Este algoritmo se utiliza sobre estructuras ordenadas, es decir, se debe ordenar la estructura como precondición. Divide repetidamente el espacio de búsqueda a la mitad, lo que reduce significativamente el tiempo necesario para encontrar un elemento.

Ejemplo en Python:

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

Este código es funcional y más rápido que el de búsqueda lineal, pero requiere de una lista ordenada.

Búsqueda de interpolación

Es similar a la búsqueda binaria, pero utiliza una fórmula para estimar dónde puede encontrarse el valor buscado, lo que puede mejorar la eficiencia en ciertas distribuciones.

Ejemplo en Python:

```python
def busqueda_interpolacional(lista, objetivo):
    inicio, fin = 0, len(lista) - 1

    while inicio <= fin and objetivo >= lista[inicio] and objetivo <= lista[fin]:

        if inicio == fin:
            if lista[inicio] == objetivo:
                return inicio
            return -1

        # FORMULA DE INTERPOLACION
        pos = inicio + ((objetivo - lista[inicio]) * (fin - inicio) // (lista[fin] - lista[inicio]))

        if lista[pos] == objetivo:
            return pos
        if lista[pos] < objetivo:
            inicio = pos + 1
        else:
            fin = pos - 1

    return -1
```

Este algoritmo es más eficiente que la búsqueda binaria en aquellos casos en que la estructura está ordenada y uniformemente distribuida.

Búsqueda Exponencial

Este algoritmo combina estrategias lineales y exponenciales, siendo útil cuando se desconoce el tamaño del rango en el que se busca.

Ejemplo en Python:

```python
def busqueda_exponencial(lista, objetivo):

    if lista[0] == objetivo:
        return 0

    i = 1
    while i < len(lista) and lista[i] <= objetivo:
        i *= 2

    return busqueda_binaria(lista, objetivo, i // 2, min(i, len(lista) -1))
```

Este algoritmo es útil para buscar en listas ordenadas y combina crecimiento exponencial con búsqueda binaria. Se puede notar que se hace uso de la función **busqueda_binaria**, sin embargo, esta no hace referencia a la vista anteriormente, sino a una nueva función que permite indicar inicio y fin de la estructura pasándoselos como argumentos.

Búsqueda por Saltos

Este algoritmo se basa en saltar algunos elementos y realizar una búsqueda lineal dentro del bloque donde probablemente se encuentre el elemento deseado.

Ejemplo en Python:

```python
def busqueda_por_saltos(lista, objetivo):
    n = len(lista)
    salto = int(math.sqrt(n)) # TAMAÑO DEL SALTO
    inicio = 0
    fin = salto

    # SALTAR BLOQUES HASTA QUE ENCONTREMOS UNO DONDE PODRIA ESTAR EL OBJETIVO
    while inicio < n and lista[min(fin, n) - 1] < objetivo:

        inicio = fin
        fin += salto

        if inicio >= n:
            return -1

    # BUSQUEDA LINEAL DENTRO DEL BLOQUE
    for i in range(inicio, min(fin, n)):
        if lista[i] == objetivo:
            return i

    return -1
```

Este algoritmo hace saltos de tamaño raíz cuadrada hasta encontrar un bloque donde podría estar el elemento. Para usarlo se debe importar el módulo **math**.

Complejidad de los algoritmos

La complejidad medida con O(n) es una forma de medir la eficiencia de un algoritmo. Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada. Por ejemplo, un algoritmo con una complejidad de O(n) tardará el doble de tiempo en ejecutarse si el tamaño de la entrada se duplica.

La notación O(n) se utiliza para describir el peor caso de complejidad de tiempo de un algoritmo. Esto significa que el algoritmo nunca tardará más de O(n) tiempo en ejecutarse para cualquier entrada de tamaño n.

La complejidad medida en O(n) es una herramienta útil para comparar diferentes algoritmos y elegir el más eficiente para una tarea determinada.

Los algoritmos de **búsqueda lineal** tienen un tiempo de ejecución de O(n), lo que significa que el tiempo de búsqueda es directamente proporcional al tamaño de la lista. Esto significa que, si la lista tiene el doble de elementos, el algoritmo tardará el doble de tiempo en encontrar el elemento deseado.

Los algoritmos de **búsqueda binaria** tienen un tiempo de ejecución de O(log n), lo que significa que el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. Esto significa que, si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado.

Tiempo de ejecución de los algoritmos de búsqueda lineal y binaria para diferentes tamaños de la lista:

Tamaño de la lista	Búsqueda lineal	Búsqueda binaria
10	10	3
100	100	7
1000	1000	10
10000	10000	13
100000	100000	16

Se puede observar que el tiempo de ejecución de la búsqueda lineal aumenta mucho más rápidamente que el tiempo de ejecución de la búsqueda binaria a medida que aumenta el tamaño de la lista. Esto hace que la búsqueda binaria sea mucho más eficiente para listas grandes.

Aplicaciones prácticas de algoritmos de búsqueda

Los algoritmos de búsqueda tienen diversas aplicaciones prácticas:

- **Sistemas de Base de Datos**: Permiten realizar consultas eficientes para extraer información específica mediante búsquedas rápidas.
- **Motores de Búsqueda**: Optimizan cómo recuperar resultados relevantes para consultas realizadas por usuarios.
- **Inteligencia Artificial**: Se utilizan en algoritmos que requieren encontrar soluciones eficientes a problemas complejos, como juegos o rutas óptimas.
- **Procesamiento de Lenguaje Natural (PLN)**: Facilitan tareas como la búsqueda semántica o recuperación eficiente en grandes conjuntos textuales.

Algoritmos de ordenamiento

Los algoritmos de ordenamiento son un conjunto de procedimientos usados para organizar datos en una secuencia específica, ya sea ascendente o descendente. La ordenación es una operación fundamental en programación que permite mejorar la eficiencia y optimización en la búsqueda de datos, facilitando tareas como la visualización y análisis de información.

Estos algoritmos son importantes por su capacidad para manejar conjuntos de datos grandes y complejos, permitiendo a los programadores crear aplicaciones más eficientes.

Tipos comunes de algoritmos de ordenamiento

Ordenamiento por selección

Este algoritmo divide el conjunto de datos en dos partes: la parte ordenada y la parte no ordenada. En cada iteración, se busca el elemento más chico (o más grande) en la parte no ordenada y se intercambia con el primer elemento no ordenado.

Ejemplo en Python:

```python
def ordenamiento_por_seleccion(lista):
    n = len(lista)

    for i in range(n):
        min_idx = i

        # BUSCO EL INDICE DEL VALOR MINIMO EN EL RESTO DE LA LISTA
        for j in range(i + 1, n):
            if lista[j] < lista[min_idx]:
                min_idx = j

        # INTERCAMBIO EL MINIMO ENCONTRADO CON EL VALOR EN LA POSICION I
        lista[i], lista[min_idx] = lista[min_idx], lista[i]

    return lista
```

Este algoritmo es simple, pero no muy eficiente para listas grandes.

Ordenamiento por inserción

Funciona construyendo una secuencia ordenada al insertar elementos uno a uno. Se toma cada elemento del conjunto no ordenado y se inserta en la posición correcta dentro del conjunto ya ordenado.

Ejemplo en Python:

```python
def ordenamiento_por_insercion(lista):

    for i in range(1, len(lista)):
        clave = lista[i]
        j = i - 1

        # MOVER ELEMENTOS MAYORES QUE LA CLAVE UNA POSICION MAS ADELANTE
        while j >= 0 and lista[j] > clave:
            lista[j + 1] = lista[j]
            j -= 1

        lista[j + 1] = clave

    return lista
```

Este algoritmo va insertando cada elemento en su posición correcta respecto a los anteriores. Es muy eficiente para listas pequeñas o casi ordenadas.

Ordenamiento rápido (Quicksort)

Utiliza el principio de divide y vencerás. Se selecciona un “pivote” y se reorganizan los elementos para que todos los menores al pivote están a su izquierda y los mayores a su derecha. Luego, se aplica recursivamente este proceso a las sublistas.

Ejemplo en Python:

```python
def quicksort(lista):

    if len(lista) <= 1:
        return lista
    else:
        pivot = lista[0]
        menor = [x for x in lista[1:] if x <= pivot]
        mayor = [x for x in lista[1:] if x > pivot]

        return quicksort(menor) + [pivot] + quicksort(mayor)
```

Generalmente es más eficiente que inserción y selección para listas grandes, especialmente cuando se implementa de manera eficiente.

Ordenamiento por mezcla (Mergesort)

Al igual que el anterior, este algoritmo está basado en divide y vencerás. Divide el conjunto original hasta que cada sublista tenga un solo elemento. Luego, combina las sublistas para crear listas ordenadas.

Ejemplo en Python:

```python
def mergesort(lista):

    if len(lista) > 1:
        medio = len(lista) // 2
        izquierda = lista[:medio]
        derecha = lista[medio:]

        # LLAMADAS RECURSIVAS
        mergesort(izquierda)
        mergesort(derecha)

        # MEZCLA
        i = j = k = 0

        while i < len(izquierda) and j < len(derecha):

            if izquierda[i] < derecha[j]:
                lista[k] = izquierda[i]
                i += 1
            else:
                lista[k] = derecha[j]
                j += 1

            k += 1

        # AGREGAR ELEMENTOS RESTANTES
        while i < len(izquierda):
            lista[k] = izquierda[i]
            i += 1
            k += 1

        while j < len(derecha):
            lista[k] = derecha[j]
            j += 1
            k += 1

    return lista
```

Este algoritmo divide la lista en dos mitades recursivamente y luego mezcla ambas mitades ordenadas. Tiene un rendimiento muy bueno.

Ordenamiento burbuja

Es un método sencillo que compara pares adyacentes e intercambia elementos si están en el orden incorrecto. Este proceso se repite hasta que no hay más intercambios necesarios, indicando que la lista está ordenada.

Ejemplo en Python:

```python
def bubble_sort(lista):

    n = len(lista)

    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

Este algoritmo es ideal para listas chicas o cuando se necesita encontrar el k-ésimo más chico (k-ésimo estadístico).

Aplicaciones prácticas de algoritmos de ordenamiento

- **Búsqueda eficiente**: Algunos algoritmos de búsqueda funcionan mejor en conjuntos de datos que ya están ordenados, como la búsqueda binaria.
- **Análisis estadístico**: La organización de datos permite realizar análisis como cálculos de promedios o medianas con mayor facilidad.
- **Visualización**: La presentación clara de datos mediante gráficos o tablas organizadas se logra gracias a los algoritmos de ordenamiento.

3. Caso práctico y metodología utilizada

El caso práctico que llevamos a cabo consiste en un programa que permite al usuario hacer una búsqueda sobre una lista de dos maneras posibles: **Búsqueda lineal** y **Búsqueda binaria**. Además, se le permite al usuario ordenar la lista a través de dos algoritmos de ordenamiento: **Ordenamiento rápido (Quicksort)** y **Ordenamiento burbuja**. El programa muestra el tiempo de ejecución de cada algoritmo seleccionado.

El tipo de dato que decidimos usar es **diccionario**, por lo que de acá surgieron las primeras investigaciones que tuvimos que realizar. Elegimos usar este tipo de dato para darle más sentido al programa y que no se trate simplemente de ordenar números o palabras, sino de ordenar **alumnos**. Antes de mostrar cómo se compone cada alumno, debemos definir qué es un diccionario: Un diccionario es un tipo de dato estructurado que permite guardar pares clave-valor.

Entonces, en nuestro caso, un alumno está compuesto por un **nombre**, **legajo** y **promedio**:

```
alumno = {
  “nombre”: “Franco”,
  “legajo”: 1,
  “promedio”: 87
}
```

Uno de los mayores desafíos al trabajar con diccionarios, fue manejar el acceso a los campos de cada alumno, ya sea para ordenar o para hacer una búsqueda, y la adaptación de los algoritmos al uso de diccionarios.

Finalmente, decidimos que el usuario podrá realizar la búsqueda de alumnos a través del legajo, ya que es el único campo que no va a repetirse. Sin embargo, para el ordenamiento, le brindamos al usuario posibilidad de decidir qué campo utilizar, quedando el programa principal de la siguiente manera:

```python
#----- PROGRAMA PRINCIPAL -----#
print("Trabajo Practico Integrador - Algoritmos de Busqueda y Ordenamiento")
print("Brian Gutierrez Colque - Franco Herrera")
print("---------------------------------------")
opcion = input("""¿Qué desea hacer?

1. Buscar alumno por legajo.
2. Ordenar alumnos por clave.

Seleccione una opción: """)

if opcion == "1":

    legajo_a_buscar = int(input("Ingrese el legajo del alumno a buscar: "))
    busca_alumnos(alumnos, legajo_a_buscar)

elif opcion == "2":

    clave = input("Ingrese la clave por la que va a ordenar [nombre], [legajo] o [promedio]: ")
    ordena_alumnos(alumnos, clave)

else:
    print ("Opción inválida")
```

A partir de la creación de la estructura del programa principal, empezamos a trabajar en las funciones **busca_alumnos()** y **ordena_alumnos()** que se van a encargar de aplicar el uso de los distintos algoritmos según sea el caso.

Otro gran desafío fue la creación de una lista con la cantidad de datos suficiente para poder dar cuenta de la diferencia de los tiempos de ejecución entre un algoritmo y otro. Para esto, acudimos a la IA, de manera tal que cree una función para la generación de la lista de manera dinámica, indicando la cantidad de alumnos que tendrá y el legajo inicial:

```python
#----- INFORMACION DE LA BASE DE DATOS -----#

# FUNCION PARA GENERAR UNA LISTA
def generar_alumnos(n, legajo_inicial=1):
    nombres_base = ["Franco", "Brian", "Martina", "Mateo", "Lucia", "Lucas", "Martin", "Luciana",
    "Brenda", "Ana", "Marcos", "David", "Rocio", "Luz", "Valentina", "Ulises", "Roman", "Emiliano"]
    alumnos_generados = []

    for i in range(n):
        nombre = random.choice(nombres_base)
        legajo = legajo_inicial + i
        promedio = random.randint(0, 100)
        alumnos_generados.append({"nombre": nombre, "legajo": legajo, "promedio": promedio})

    return alumnos_generados

alumnos = generar_alumnos(500, 1)
```

A grandes rasgos, las funciones **busca_alumnos()** y **ordena_alumnos()** le solicitan al usuario que indique el tipo de búsqueda/ordenamiento que desea utilizar y en base a esa decisión usan un algoritmo u otro. Ambas funciones muestran el tiempo transcurrido en la ejecución del algoritmo en milisegundos.

En el caso de la función de búsqueda de alumnos, también le solicita al usuario que ingrese el número de legajo del alumno que desea buscar y muestra la información del alumno encontrado. En caso de que el legajo ingresado no se encuentre, se muestra el mensaje “No se encontró el alumno con el legajo XXX”.

En el caso de la función de ordenamiento, ésta le solicita al usuario el campo por el cual desea ordenar la lista, pudiendo elegir **nombre**, **legajo** y **promedio**. A modo de verificación del funcionamiento, también muestra la lista ordenada.

Para la realización del caso práctico se utilizó **Visual Studio Code** y utilizamos las librerías **random** para la generación dinámica de la lista y **time** para el registro de los tiempos de ejecución de cada algoritmo.

4. Resultados obtenidos

Ejecución del programa:

```
PS C:\Users\franc> & C:/Users/franc/AppData/Local/Microsoft/WindowsA
Trabajo Practico Integrador - Algoritmos de Busqueda y Ordenamiento
Brian Gutierrez Colque - Franco Herrera
---------------------------------------
¿Qué desea hacer?

1. Buscar alumno por legajo.
2. Ordenar alumnos por clave.

Seleccione una opción:
```

Búsqueda Lineal:

```
Seleccione una opción: 1
Ingrese el legajo del alumno a buscar: 499
¿Qué tipo de búsqueda quiere hacer?

    1. Busqueda Lineal.
    2. Busqueda Binaria.

    Seleccione una opción: 1
INICIO BUSQUEDA LINEAL
------------------------
Tiempo transcurrido: 0.039 milisegundos
Alumno encontrado: {'nombre': 'Franco', 'legajo': 499, 'promedio': 32}
------------------------
FIN BUSQUEDA LINEAL
```

Búsqueda Binaria:

```
Seleccione una opción: 1
Ingrese el legajo del alumno a buscar: 499
¿Qué tipo de búsqueda quiere hacer?

    1. Busqueda Lineal.
    2. Busqueda Binaria.

    Seleccione una opción: 2
INICIO BUSQUEDA BINARIA
------------------------
Tiempo transcurrido: 0.007 milisegundos
Alumno encontrado: {'nombre': 'David', 'legajo': 499, 'promedio': 73}
------------------------
FIN BUSQUEDA BINARIA
```

Ordenamiento rápido (Quicksort)

NOTA: Para esta prueba decidimos comentar el print que muestra la lista ordenada, ya que la misma contiene 500 alumnos.

```
Seleccione una opción: 2
Ingrese la clave por la que va a ordenar [nombre], [legajo] o [promedio]: legajo
¿Qué tipo de ordenamiento quiere hacer?

    1. Ordenamiento rapido.
    2. Ordenamiento burbuja.

    Seleccione una opción: 1
INICIO ORDENAMIENTO RAPIDO
------------------------
Tiempo transcurrido: 12.611 milisegundos
------------------------
FIN ORDENAMIENTO RAPIDO
```

Ordenamiento burbuja

NOTA: Para esta prueba decidimos comentar el print que muestra la lista ordenada, ya que la misma contiene 500 alumnos.

```
Seleccione una opción: 2
Ingrese la clave por la que va a ordenar [nombre], [legajo] o [promedio]: legajo
¿Qué tipo de ordenamiento quiere hacer?

    1. Ordenamiento rapido.
    2. Ordenamiento burbuja.

    Seleccione una opción: 2
INICIO ORDENAMIENTO BURBUJA
------------------------
Tiempo transcurrido: 6.612 milisegundos
------------------------
FIN ORDENAMIENTO BURBUJA
```

Errores corregidos:

En un principio, usábamos **time()** para calcular los tiempos de ejecución, sin embargo, al no usar una lista con muchos datos, siempre nos daba el tiempo de 0 segundos, por lo que decidimos utilizar **perf_counter()**, que es mucho más precisa, y multiplicamos el resultado por mil, pasando el tiempo a milisegundos.

Otro de los inconvenientes que tuvimos, fue cuando quisimos probar con una lista de 1000 alumnos. Esto ocurrió porque el algoritmo de ordenamiento rápido utiliza recursión, por lo que nos dio el conocido **RecursionError: maximum recursion depth exceeded**:

```
Seleccione una opción: 2
Ingrese la clave por la que va a ordenar [nombre], [legajo] o [promedio]: legajo
¿Qué tipo de ordenamiento quiere hacer?

    1. Ordenamiento rapido.
    2. Ordenamiento burbuja.

    Seleccione una opción: 1
INICIO ORDENAMIENTO RAPIDO
------------------------
Traceback (most recent call last):
  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 195, in <module>
    ordena_alumnos(alumnos, clave)
  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 144, in ordena_alumnos
    lista_ordenada = ordenamiento_rapido(lista, clave)
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 58, in ordenamiento_rapido
    return ordenamiento_rapido(menores, clave) + [pivote] + ordenamiento_rapido(mayores, clave)
                                                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 58, in ordenamiento_rapido
    return ordenamiento_rapido(menores, clave) + [pivote] + ordenamiento_rapido(mayores, clave)
                                                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 58, in ordenamiento_rapido
    return ordenamiento_rapido(menores, clave) + [pivote] + ordenamiento_rapido(mayores, clave)
                                                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

  [Previous line repeated 994 more times]
  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 55, in ordenamiento_rapido
    menores = [x for x in lista[1:] if x[clave] <= pivote[clave]]
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
RecursionError: maximum recursion depth exceeded
```

Finalmente, decidimos utilizar una lista con 500 alumnos, ya que modificar la función para que utilice iteración iba a significar no utilizar la estructura de los ejemplos mostrados en el punto 2.

Por último, en un principio olvidamos validar el ingreso de clave por la que el usuario desea ordenar la lista, por lo que si el usuario ingresaba un valor distinto a nombre, legajo o promedio nos daba el error de tipo **KeyError**:

```
Seleccione una opción: 2
Ingrese la clave por la que va a ordenar [nombre], [legajo] o [promedio]: apellido
¿Qué tipo de ordenamiento quiere hacer?

    1. Ordenamiento rapido.
    2. Ordenamiento burbuja.

    Seleccione una opción: 1
INICIO ORDENAMIENTO RAPIDO
------------------------
Traceback (most recent call last):
  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 196, in <module>
    ordena_alumnos(alumnos, clave)
  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 145, in ordena_alumnos
    lista_ordenada = ordenamiento_rapido(lista, clave)
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 55, in ordenamiento_rapido
    menores = [x for x in lista[1:] if x[clave] <= pivote[clave]]
              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "c:\Users\franc\OneDrive\Facultad\Programacion I\TP INTEGRADOR\PROGRAMA_TP_INTEGRADOR.py", line 55, in <listcomp>
    menores = [x for x in lista[1:] if x[clave] <= pivote[clave]]
                                       ~^^^^^^^
KeyError: 'apellido'
```

Evaluación de rendimiento:

Algoritmos de búsqueda:

- Búsqueda Lineal: 0,039 ms
- Búsqueda Binaria: 0,007 ms

Algoritmos de ordenamiento:

- Ordenamiento rápido: 12,611 ms
- Ordenamiento burbuja: 6,612 ms

NOTA: Si bien la lista de alumnos se genera de manera dinámica por cada ejecución (es decir, no usamos la misma lista para todas las pruebas), la cantidad de elementos siempre es la misma, y tras ejecutar múltiples veces el programa, llegamos a la conclusión de que los tiempos son muy similares.

Repositorio en GitHub:

https://github.com/fleonelherrera/TP-Integrador-Progral

5. Conclusiones

Los algoritmos de búsqueda y ordenamiento son pilares fundamentales en el campo de la programación, ya que permiten gestionar y manipular datos de manera eficiente. La capacidad de buscar un elemento específico en una colección o de ordenar datos para facilitar su análisis es esencial en la mayoría de las aplicaciones informáticas.

En este trabajo, aplicamos en un caso práctico cuatro tipos de algoritmos diferentes (dos de ordenamiento y dos de búsqueda). Pudimos observar cómo la búsqueda binaria se impone ante la búsqueda lineal, con la desventaja de que la primera necesita recibir una lista ordenada, y cómo el ordenamiento burbuja reduce el tiempo de ejecución a la mitad en comparación con el ordenamiento rápido. Así, dimos cuenta de que cada algoritmo tiene sus propias ventajas y desventajas en términos de complejidad temporal y espacial.

Como posibles mejoras del programa, consideramos fundamental que se pueda incorporar el uso de una misma lista para todos los casos de prueba y, en medida de lo posible, que la misma tenga una mayor cantidad de registros. Esto será fundamental para hacer más notable la diferencia de tiempos de ejecución entre un algoritmo y otro. También, se podrían agregar otros tipos de algoritmos, como la búsqueda exponencial o Mergesort.

En resumen y a modo de cierre, podemos decir que tanto los algoritmos de búsqueda como los de ordenamiento son herramientas básicas que todo programador debe dominar. Su comprensión no solo facilita la resolución efectiva de problemas complejos, sino que también sienta las bases para el desarrollo de software robusto y eficiente.

6. Bibliografía

Algoritmos de Búsqueda y Ordenamiento: Fundamentos y Aplicaciones Clave en Programación. *Wolf Diseño Web*. https://wolfdisenoweb.com/2025/04/14/algoritmos-de-busqueda-y-ordenamiento/

Algoritmos de Búsqueda y Ordenamiento. *Andrés Mise Olivera (Medium)*. https://medium.com/@mise/algoritmos-de-b%C3%BAsqueda-y-ordenamiento-7116bcea03d0

ChatGPT. *OpenAI*. https://chatgpt.com/

time – Time Access and conversions. *Documentación de Python*. https://docs.python.org/3/library/time.html

Teoría sobre Busqueda y Ordenamiento. *Universidad Tecnológica Nacional*. https://tup.sied.utn.edu.ar/

7. Anexos

Tabla comparativa de los resultados para las distintas pruebas de los algoritmos de búsqueda:

	Búsqueda lineal	Búsqueda binaria
Primera prueba	0,026 ms	0,005 ms
Segunda prueba	0,027 ms	0,005 ms
Tercera prueba	0,023 ms	0,004 ms
Cuarta prueba	0,025 ms	0,005 ms

Tabla comparativa de los resultados para las distintas pruebas de los algoritmos de ordenamiento (el ordenamiento se hizo por legajo):

	Ordenamiento rápido	Ordenamiento burbuja
Primera prueba	14,765 ms	6,501 ms
Segunda prueba	12,329 ms	6,429 ms
Tercera prueba	13,479 ms	6,740 ms
Cuarta prueba	12,551 ms	6,448 ms

Ordenamiento rápido (tamaño de lista: 15, ordenamiento por promedio)

```
Seleccione una opción: 2
Ingrese la clave por la que va a ordenar [nombre], [legajo] o [promedio]: promedio
¿Qué tipo de ordenamiento quiere hacer?

    1. Ordenamiento rapido.
    2. Ordenamiento burbuja.

    Seleccione una opción: 1
INICIO ORDENAMIENTO RAPIDO
------------------------
Lista ordenada: [{'nombre': 'Brenda', 'legajo': 2, 'promedio': 4}, {'nombre': 'Lucia', 'legajo': 13, 'promedio': 6}, {'nom
bre': 'Luz', 'legajo': 11, 'promedio': 10}, {'nombre': 'Martina', 'legajo': 10, 'promedio': 14}, {'nombre': 'Brian', 'lega
jo': 4, 'promedio': 15}, {'nombre': 'Mateo', 'legajo': 1, 'promedio': 18}, {'nombre': 'Luz', 'legajo': 15, 'promedio': 20}
, {'nombre': 'David', 'legajo': 12, 'promedio': 25}, {'nombre': 'Emiliano', 'legajo': 3, 'promedio': 25}, {'nombre': 'Marc
os', 'legajo': 7, 'promedio': 44}, {'nombre': 'Lucia', 'legajo': 6, 'promedio': 58}, {'nombre': 'Emiliano', 'legajo': 5, '
promedio': 60}, {'nombre': 'Ana', 'legajo': 8, 'promedio': 65}, {'nombre': 'Luz', 'legajo': 9, 'promedio': 76}, {'nombre':
 'Luciana', 'legajo': 14, 'promedio': 82}]
------------------------
FIN ORDENAMIENTO RAPIDO
```

Ordenamiento burbuja (tamaño de lista: 15, ordenamiento por promedio)

```
Seleccione una opción: 2
Ingrese la clave por la que va a ordenar [nombre], [legajo] o [promedio]: promedio
¿Qué tipo de ordenamiento quiere hacer?

    1. Ordenamiento rapido.
    2. Ordenamiento burbuja.

    Seleccione una opción: 2
INICIO ORDENAMIENTO BURBUJA
------------------------
Lista ordenada: [{'nombre': 'Martin', 'legajo': 3, 'promedio': 9}, {'nombre': 'Ana', 'legajo': 6, 'promedio': 11}, {'nombr
e': 'Marcos', 'legajo': 4, 'promedio': 15}, {'nombre': 'Mateo', 'legajo': 11, 'promedio': 15}, {'nombre': 'Luciana', 'lega
jo': 10, 'promedio': 17}, {'nombre': 'Ana', 'legajo': 9, 'promedio': 26}, {'nombre': 'Rocio', 'legajo': 5, 'promedio': 36}
, {'nombre': 'Valentina', 'legajo': 12, 'promedio': 43}, {'nombre': 'Lucas', 'legajo': 13, 'promedio': 44}, {'nombre': 'Lu
z', 'legajo': 7, 'promedio': 47}, {'nombre': 'Roman', 'legajo': 14, 'promedio': 47}, {'nombre': 'David', 'legajo': 1, 'pro
medio': 54}, {'nombre': 'Martin', 'legajo': 8, 'promedio': 62}, {'nombre': 'Luz', 'legajo': 2, 'promedio': 67}, {'nombre':
 'Emiliano', 'legajo': 15, 'promedio': 85}]
------------------------
FIN ORDENAMIENTO BURBUJA
```