



Universidad Tecnológica Nacional

Trabajo Práctico – Búsqueda y Ordenamiento: “Análisis y aplicación de búsqueda y ordenamiento en Python sobre datos de la PokéAPI”

Alumnos:

Arturo Kaadú – kaaduarturo4@gmail.com – Comisión 15

Belén Yarde Buller – belenyardebuller@gmail.com – Comisión 23

Materia:

Programación I

Comisión 15

Docente Titular: Ariel Enferrel

Docente Tutor: Maximiliano Sar Fernández

Comisión 23

Docente Titular: Nicolás Quirós

Docente Tutor: Flor Camila Gubiotti

Fecha de Entrega: 09/06/2025

Índice

1. Introducción.....	3
2. Marco Teórico.....	4
3. Caso Práctico.....	7
4. Metodología Utilizada.....	22
5. Resultados Obtenidos.....	23
6. Conclusiones.....	33
7. Bibliografía.....	35
8. Anexos.....	36

1. Introducción

El presente trabajo se centra en el estudio e implementación de los algoritmos de búsqueda y ordenamiento, herramientas esenciales que permiten localizar y organizar información de forma optimizada. La elección de este tema tiene como punto central la importancia de estos algoritmos en casi cualquier aplicación de software, desde bases de datos masivas hasta el funcionamiento interno de interfaces de usuario. Comprender y conocer cómo implementar estos algoritmos no solo mejora el rendimiento de los programas, sino que también sienta las bases para resolver problemas complejos de forma más estructurada y lógica.

Este trabajo se propone alcanzar tres objetivos principales:

- ❖ Implementar y comprender el funcionamiento de los algoritmos de búsqueda lineal y binaria, y los algoritmos de ordenamiento bubble sort y quick sort. Se utilizará el lenguaje de programación python.
- ❖ Aplicar estos algoritmos a un conjunto de datos real y dinámico. Para esto se utilizará la Poké API de la cual se alimentarán los algoritmos para demostrar su funcionalidad en un caso práctico real y concreto.
- ❖ Evaluar y comparar el rendimiento de los algoritmos implementados y destacar su comportamiento ante diferentes escenarios de ordenamiento de pokemones (numérico y alfabético), lo cual se logrará al medir el tiempo de ejecución de cada algoritmo.

2. Marco teórico

Los algoritmos de búsqueda son procedimientos diseñados para localizar uno o más elementos dentro de una estructura de datos. Su eficiencia se centra en la capacidad de minimizar el número de comparaciones necesarias para encontrar el dato deseado.

Búsqueda Lineal: Es el método de búsqueda más simple. Consiste en recorrer secuencialmente cada elemento de una lista hasta encontrar el valor buscado o hasta que se haya revisado toda la lista.

- **Funcionamiento:** Comienza en el primer elemento y compara cada uno con el valor objetivo.
- **Ventajas:** Es fácil de implementar y no requiere que la lista esté ordenada.
- **Desventajas:** Su eficiencia es baja para listas grandes, ya que en el peor de los casos (elemento no encontrado o al final de la lista) debe recorrer toda la lista. Su complejidad temporal promedio y en el peor caso es $O(N)$, donde N es el número de elementos.
- **Implementación en Python:** Se puede lograr fácilmente con un bucle `for` o `while` que itere sobre los elementos de la lista. En este trabajo, se implementa para encontrar un pokemon específico en la lista obtenida de la API, buscando por su nombre o por su altura.

Búsqueda Binaria (Binary Search): Es un algoritmo mucho más eficiente que la búsqueda lineal, pero requiere que la lista esté previamente ordenada.

- **Funcionamiento:** Divide repetidamente por la mitad la porción de la lista donde podría estar el elemento buscado. Compara el valor objetivo con el elemento central de la porción actual: si coinciden, lo encuentra; si el objetivo es menor, busca en la mitad izquierda; si es mayor, busca en la mitad derecha.
- **Ventajas:** Mucho más rápido para listas grandes. Su complejidad temporal es $O(\log N)$.
- **Desventajas:** Requiere una lista ordenada.

Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permiten que diferentes aplicaciones de software se comuniquen entre sí. Para este trabajo, y a fin de demostrar claramente el rendimiento de los algoritmos de búsqueda y

ordenamiento se utilizó la PokèAPI, una API RESTful que proporciona acceso a una vasta base de datos de información sobre pokemon.

- **Funcionamiento:** Se realizan solicitudes HTTP (GET) a URLs específicas de la API para obtener datos en formato JSON.
- **requests:** Es una librería de Python que simplifica la realización de solicitudes HTTP, facilitando la interacción con APIs REST.
- **JSON (JavaScript Object Notation):** Es un formato de intercambio de datos ligero, fácil de leer y escribir por humanos, y fácil de analizar y generar por máquinas. Es el formato estándar para la comunicación en la mayoría de las APIs REST.

Los algoritmos de ordenamiento son métodos que reorganizan los elementos de una lista o array en una secuencia específica que puede ser ascendente o descendente, basándose en un criterio de comparación. La elección del algoritmo impacta directamente en la eficiencia de la escalabilidad del procesamiento de datos. Para este trabajo se utilizarán los siguientes:

Bubble Sort (Ordenamiento de Burbuja)

- **Funcionamiento:** Es un algoritmo que recorre la lista repetidamente comparando pares de elementos adyacentes y los intercambia si están en el orden incorrecto. Las burbujas de elementos más grandes o más pequeños van ascendiendo o descendiendo a su posición final en cada pasada.
- **Ventajas:** Sencillo de implementar.
- **Desventajas:** Extremadamente ineficiente para listas grandes debido a su alta cantidad de comparaciones e intercambios. Su complejidad temporal promedio y en el peor caso es $O(N^2)$, donde N es el número de elementos.
- **Implementación en Python:** En este proyecto, se aplica a la lista de pokemon para ordenarla por altura o nombre, demostrando su funcionalidad básica en la manipulación de diccionarios dentro de una lista.

Quick Sort (Ordenamiento Rápido)

Es un algoritmo de ordenamiento eficiente basado en la técnica de "divide y vencerás".

- **Funcionamiento:** Selecciona un elemento llamado pivote de la lista, en el caso de este trabajo se eligió como pivote el resultado de dividir en dos la lista. Esto crea dos sublistas, las que son menores que el pivote, y las que son mayores. Recursivamente, Quicksort se aplica a estas dos sublistas y finalmente se concatenan las sub listas ordenadas con el pivote en el medio.

- Ventajas: En promedio, es uno de los algoritmos de ordenamiento más rápidos y es muy eficiente. Su complejidad temporal promedio es $O(N\log N)$.
- Desventajas: En el peor de los casos (ej. lista ya ordenada o inversa, dependiendo de la elección del pivote), su complejidad puede degradarse a $O(N^2)$.
- Implementación en Python: Se implementa en este trabajo para ordenar la lista de pokemon por altura o nombre, aprovechando su naturaleza recursiva.

3. Caso práctico

El objetivo central que este trabajo abordará es la aplicación, medición y comparación de algoritmos de búsqueda y ordenamiento en los datos de pokemon obtenidos a través de la PokéApi. En un escenario donde se requiere tanto la recuperación rápida de elementos específicos como la organización sistemática de la colección según diferentes criterios (en el caso de este trabajo será por nombre y altura), es fundamental contar con algoritmos que permitan agilizar y medir de forma optimizada estos tiempos.

Para ello se plantea la tarea de:

- ❖ Obtener una lista de pokemon mediante una solicitud GET a la PokèAPI para realizar la ingesta de datos de una fuente externa y dinámica.
- ❖ Permitir la búsqueda de pokemon. Particularmente estas dos acciones:
 - Identificar si un pokemon específico existe en la lista mediante la búsqueda de su nombre.
 - Encontrar pokemons según su altura en decímetros.
- ❖ Habilitar el ordenamiento de pokemon: Organizar la lista completa en función de atributos numéricos (altura) o textuales (nombre).
- ❖ Medir el rendimiento: Cuantificar el tiempo que tardan los algoritmos elegidos en ejecutar sus operaciones para poder comparar su eficiencia.

Este escenario es representativo de muchos problemas reales en desarrollo de software, donde la gestión de grandes volúmenes de datos requiere soluciones eficientes de búsqueda y ordenamiento. Para abordar el tema planteado, se estructuró el proyecto en varias funciones que envuelven la lógica de obtención de datos, de búsqueda y también de ordenamiento.

Ejecución principal del programa

```
--
70 ~ if __name__ == "__main__":
71 ~     GET_POKEMONS_LIMIT = 50
72 ~     runProgram = True
73 ~
74 ~     # Llamamos a la función para obtener los datos y guardarlos en pokemons_list
75 ~     pokemons_list = get_pokemons(GET_POKEMONS_LIMIT)
76 ~     pprint.pprint(pokemons_list)
77 ~
78 ~     if not pokemons_list:
79 ~         print("Error al obtener los pokemons.")
80 ~     else:
81 ~         while runProgram:
82 ~             # Acá ejecutamos los algoritmos de búsqueda y ordenamiento
83 ~             user_option = input("Ingrese la operación que desea realizar: \n0. Salir \n1. Búsqueda por nombre \n2. Búsqueda por altura \n3. Ordenamiento por altura \n4. Ordenamiento por nombre: ")
84 ~             if (user_option == "1"):
85 ~
86 ~                 # Búsqueda por nombre
87 ~                 desired_pokemon = input("Por favor, ingrese el pokemon que desea buscar: ")
88 ~
89 ~                 start_time = timeit.default_timer()
90 ~                 linear_search(pokemons_list, 'name', desired_pokemon)
91 ~                 end_time = timeit.default_timer()
92 ~                 print("Tiempo de ejecución para linear_search por nombre:", end_time - start_time, "segundos")
93 ~
94 ~                 start_time = timeit.default_timer()
95 ~                 binary_search(pokemons_list, 'name', desired_pokemon)
96 ~                 end_time = timeit.default_timer()
97 ~                 print("Tiempo de ejecución para binary_search:", end_time - start_time, "segundos")
98 ~
99 ~                 print(f"\n\n--- Fin de la ejecución de la opción {user_option} ---\n")
100 ~
101 ~             elif (user_option == "2"):
102 ~
103 ~                 # Búsqueda por altura
104 ~                 desired_height = input("Por favor, ingrese la altura que desea buscar (en decímetros): ")
105 ~
106 ~                 start_time = timeit.default_timer()
107 ~                 linear_search(pokemons_list, 'height', desired_height)
108 ~                 end_time = timeit.default_timer()
109 ~                 print("Tiempo de ejecución para linear_search por altura:", end_time - start_time, "segundos")
110 ~
111 ~                 start_time = timeit.default_timer()
112 ~                 binary_search(pokemons_list, 'height', desired_height)
113 ~                 end_time = timeit.default_timer()
114 ~                 print("Tiempo de ejecución para binary_search:", end_time - start_time, "segundos")
115 ~
116 ~                 print(f"\n\n--- Fin de la ejecución de la opción {user_option} ---\n")
--
```

El programa comienza en la línea 70 del archivo main.py, donde se setea la constante GET_POKEMONS_LIMIT con el objetivo de indicar el número total de pokemons a obtener desde la PokèAPI. En la siguiente línea se setea la variable runProgram, que servirá de bandera para ejecutar el menú de opciones que tendrá el usuario. En la línea 75 se lleva a cabo la llamada a la API externa, y se guarda el resultado en la variable pokemons_list. Para mayor legibilidad, se opta por el uso de la librería pprint para mostrar el listado de pokemons en consola. En las líneas 78 y 79 se maneja un posible error en la petición.


```

117 ..... elif (user_option == "3"):
118 .....     # Bubble sort
119 .....     start_time = timeit.default_timer()
120 .....     sorted_pokemons_bubble = bubble_sort(pokemons_list, 'height', ascending=True)
121 .....     end_time = timeit.default_timer()
122 .....     print_pokemon_list(sorted_pokemons_bubble, "Pokémon ordenados por altura (ascendente) con Bubble Sort", limit=GET_POKEMONS_LIMIT)
123 .....     print("Tiempo de ejecución para bubble_sort:", end_time - start_time, "segundos")
124 .....
125 .....     # Quick sort
126 .....     start_time = timeit.default_timer()
127 .....     sorted_pokemons_quick = quick_sort(pokemons_list, 'height', ascending=True)
128 .....     end_time = timeit.default_timer()
129 .....     print_pokemon_list(sorted_pokemons_quick, "Pokémon ordenados por altura (ascendente) con Quick Sort", limit=GET_POKEMONS_LIMIT)
130 .....     print("Tiempo de ejecución para quick_sort:", end_time - start_time, "segundos")
131 .....
132 ..... elif (user_option == "4"):
133 .....     # Ordenamiento por nombre
134 .....     start_time = timeit.default_timer()
135 .....     sorted_pokemons_name = bubble_sort(pokemons_list, 'name', ascending=True)
136 .....     end_time = timeit.default_timer()
137 .....     print_pokemon_list(sorted_pokemons_name, "Pokémon ordenados por nombre (ascendente) con Bubble Sort", limit=GET_POKEMONS_LIMIT)
138 .....
139 .....     # Ordenamiento por nombre descendente
140 .....     start_time = timeit.default_timer()
141 .....     sorted_pokemons_name_desc = quick_sort(pokemons_list, 'name', ascending=False)
142 .....     end_time = timeit.default_timer()
143 .....     print_pokemon_list(sorted_pokemons_name_desc, "Pokémon ordenados por nombre (descendente) con Quick Sort", limit=GET_POKEMONS_LIMIT)
144 .....     print("Tiempo de ejecución para quick_sort:", end_time - start_time, "segundos")
145 .....
146 ..... elif (user_option == "0"):
147 .....     print("Saliendo del programa...")
148 .....     runProgram = False
149 .....
150 ..... else:
151 .....     print("Por favor, ingrese una opción válida.")
152 .....
153 .....

```

Dentro de las líneas 81 y 153 se lleva a cabo el bucle while, cuya bandera fue mencionada anteriormente, brindándole al usuario la posibilidad de realizar 4 operaciones:

1. Búsqueda por nombre
2. Búsqueda por altura
3. Ordenamiento por altura
4. Ordenamiento por nombre

En cada una de esas opciones se realizarán operaciones comparativas y se imprimirán en pantalla los tiempos de ejecución. Para el caso de los algoritmos de búsqueda, se ejecutarán funciones de búsqueda lineal y binaria, solicitando al usuario ingresar el nombre o altura deseada, acorde a la opción escogida. Las variables `desired_pokemon` y `desired_height` guardarán estos valores, que luego serán utilizados al llamar a cada función. Para el caso de los algoritmos de ordenamiento, se procederá de igual forma, corriendo ambos (`quick_sort` y `bubble_sort`) en cada opción y mostrando por consola los tiempos de ejecución en cada caso.

El usuario también tendrá la opción de finalizar la ejecución del programa ingresando la opción 0 en consola, tal como se indica en la línea 148 del código. En caso de no ingresar ninguna de las opciones brindadas por el programa, se le informará que ingrese una opción válida (línea 153).

Llamada a la PokéAPI

```
11 def get_pokemons(limit):
12     print("Obteniendo pokemons...")
13     # Obtenemos los primeros n (limit) pokemons de la PokéAPI, extrayendo solo su nombre y altura.
14     # Devolvemos los datos directamente como una lista de diccionarios.
15
16     # Definimos la url base de la PokéAPI
17     base_url = "https://pokeapi.co/api/v2/"
18
19     # Primera petición a endpoint para obtener un listado de los primeros 50 pokemons
20     pokemons_list_url = f"{base_url}pokemon?offset=0&limit={limit}"
21
22     # Inicializamos una lista vacía que luego retornaremos al final de la función
23     get_pokemons_list = []
24
25     # Hacemos la petición a la PokéAPI usando try except para capturar posibles errores y excepciones
26     try:
27         response_list = requests.get(pokemons_list_url)
28         response_list.raise_for_status()
29         list_info = response_list.json()
30
31         # Del json devuelto en la petición, obtenemos el campo "results"
32         pokemon_entries = list_info['results']
33
34     except requests.exceptions.RequestException as e:
35         # Si hay un error en la request, se imprime un mensaje de error por consola y se devuelve una lista vacía
36         print(f"Error en get_pokemons(): {e}")
37         return []
38
39     # Segunda petición a endpoint para obtener la altura de cada pokemon
40     # Iteramos sobre cada pokemon obtenido en la lista inicial, y de cada uno de ellos, tomamos la url: entry['url']
41     for i, entry in enumerate(pokemon_entries):
42         pokemon_detail_url = entry['url']
43
44         # Volvemos a utilizar un bloque try except para capturar posibles errores
45         try:
46             response_detail = requests.get(pokemon_detail_url)
47             response_detail.raise_for_status()
48             pokemon_info = response_detail.json()
49
50             # Creamos un diccionario pokemon_data para guardar la información que nos interesa: nombre y altura
51             pokemon_data = {
52                 "name": pokemon_info['name'],
53                 "height": pokemon_info['height']
54             }
55
56             # Agregamos el diccionario de cada pokemon a la lista que retornará la función
57             get_pokemons_list.append(pokemon_data)
58
59     except requests.exceptions.RequestException as e:
60         # Si hay un error en la request, se imprime un mensaje de error por consola y se devuelve una lista vacía
61         print(f"Error al obtener detalles para '{entry['name']}': ({pokemon_detail_url}): {e}")
62         continue # Continúa con el siguiente Pokémon si hay un error
63
64     # Devolvemos la lista obtenida mediante las llamadas a la API
65     print("Listado de pokemons obtenido exitosamente")
66     return get_pokemons_list
```

Esta función se encarga de hacer dos peticiones a la PokéAPI para traer los datos necesarios para ejecutar los algoritmos de búsqueda y ordenamiento. Luego de realizar un print para informar al usuario que se está ejecutando la búsqueda, se procede a definir las urls a utilizar para las solicitudes. A través del parámetro limit, recibido mediante la constante GET_POKEMONS_LIMIT mencionada anteriormente, se realiza la primera petición, que nos devolverá en la llave 'results' un listado de pokemons con sus nombres y una url de detalle para cada uno. Esa url de detalle servirá para realizar la segunda petición, en la línea 46, con el fin de obtener información sobre la altura de cada pokemon (contenida en la llave 'height'). Con ambos datos ('name', conseguido en la primera petición, y 'height', en la

segunda), se crea el diccionario `pokemon_data`. Cada diccionario, referido a cada pokemon individual, será añadido a la lista gracias al método `append`.

Finalmente, se devolverá la lista, contenida en la variable `get_pokemons_list` (líneas 65 y 66) junto con un mensaje de éxito que se mostrará por consola. Cabe mencionar que en ambas peticiones se utilizan bloques `try-except` para capturar posibles errores y manejarlos de forma segura (líneas 26 a 37 y 45 a 62).

Algoritmos de búsqueda

Dentro del archivo `funciones_busqueda.py` se realizaron dos funciones orientadas a desarrollar algoritmos de búsqueda lineal (`lineal_search`) y binaria (`binary_search`). Ambas reciben tres parámetros:

1. La lista de pokemon (`pokemons_list`), proveniente de la llamada a la PokèAPI dentro de la ejecución principal del programa, en el archivo `main.py`.
2. La clave de búsqueda (`search_key`), que puede ser 'name' o 'height'. Esto está determinado por la opción elegida por el usuario al ejecutar el programa.
3. El valor de búsqueda (`search_query`), que será el valor de búsqueda ingresado por el usuario en consola. Se guarda en las variables `desired_pokemon` o `desired_height` dentro del archivo `main.py`.

Búsqueda lineal

```
13 # Búsqueda Lineal
14 def linear_search(pokemons_list, search_key, search_query):
15
16     print(f"\n\n--- Ejecutando Búsqueda Lineal por {getKeyTranslation(search_key)}: ---\n")
17
18     # Búsqueda por nombre
19     if search_key == 'name':
20         search_query = normalizeInput(search_query)
21         for pokemon in pokemons_list:
22             if pokemon.get(search_key) and pokemon[search_key].lower() == search_query:
23                 print(f"'{search_query}' encontrado: {pokemon}")
24                 return pokemon
25
26     # Búsqueda por altura
27     elif search_key == 'height':
28         # Intentamos convertir el valor de búsqueda a un tipo numérico para comparar alturas
29         try:
30             search_query = float(search_query)
31         except ValueError:
32             # Si la conversión falla, capturamos el error, imprimimos un mensaje informativo y devolvemos None
33             print("Error: El valor de altura debe ser un número.")
34             return None
35
36         for pokemon in pokemons_list:
37             if pokemon.get(search_key) and pokemon[search_key] == search_query:
38                 print(f"pokemon con altura '{search_query}' encontrado: {pokemon}")
39                 return pokemon
40
41     else:
42         # Si la clave de búsqueda no es 'name' ni 'height', imprimimos un mensaje informativo y devolvemos None
43         # Esto no ocurrirá ya que el programa manejará las claves de búsqueda 'name' y 'height', pero se incluye para evitar
44         # posibles errores y asegurarse de que el programa funcione correctamente
45         print(f"Criterio de búsqueda '{search_key}' no válido. Opciones válidas: 'name' o 'height'.")
46         return None
47
48     print(f"'{search_query}' no encontrado por {getKeyTranslation(search_key)}.")
49     return None
```

Esta función realiza una búsqueda secuencial del elemento a encontrar. En la línea 19 se desarrolla el caso de búsqueda por nombre, haciendo una previa normalización del dato ingresado por el usuario a través de la función `normalizeInput`. Se procede a recorrer la lista de pokemons, elemento por elemento, hasta encontrar la coincidencia deseada. Esto se produce cuando el pokemon con índice `search_key` (en el caso de búsqueda por nombre, esa llave será 'name') coincide con el nombre del pokemon ingresado en consola por el usuario (`search_query`). Esto se representa en la línea 22 mediante un `if` que, en su segunda condición dentro del bloque `and`, indica: `pokemon[search_key] == search_query`.

En la línea 27 se desarrolla la búsqueda por altura. Lo primero que se hace es convertir el input ingresado por el usuario en un `float`, capturando posibles errores en el proceso. Una vez convertido el valor, se recorre la lista de pokemons de manera secuencial, buscando una coincidencia entre el valor de `pokemon['height']`, es decir, la altura de cada pokemon en cada iteración, y el número de altura deseada ingresado por el usuario, que se encuentra en la variable `search_query`. Si hay coincidencia, se retorna el pokemon encontrado. De lo contrario, se devuelve `None` y se imprime por consola un mensaje informativo. Para dicho mensaje, se usa la función `getKeyTranslation`.

Búsqueda binaria

```
52 def binary_search(pokemons_list, search_key, search_query):
53
54     print(f"\n--- Ejecutando Búsqueda Binaria por {getKeyTranslation(search_key)}: ---\n")
55
56     # Tomamos en cuenta el caso de que la función reciba una lista vacía
57     if not pokemons_list:
58         print("La lista de pokemon está vacía.")
59         return None
60
61     # Ordenamos la lista de pokemons utilizando la función quick_sort del archivo funciones_ordenamiento. Esta función
62     # ordena la lista de forma ascendente por defecto. Guardamos el resultado en la variable pokemons_list_sorted.
63     pokemons_list_sorted = quick_sort(pokemons_list[:], search_key)
64
65     # Inicializamos los punteros low y high para el algoritmo de búsqueda binaria
66     # 'low' es el índice más bajo de la lista, el primer elemento.
67     low = 0
68     # 'high' es el índice más alto, es decir, el último elemento de la lista ordenada.
69     # Se le resta 1 a len() ya que el rango de la lista comienza en 0.
70     high = len(pokemons_list_sorted) - 1
71
72     # Normalizamos el valor de search_query que proviene del input ingresado por el usuario en consola
73     search_query_normalized = None
74     if search_key == 'name':
75         # Si se trata de un string, usamos nuestra función normalizeUserInput de /utils
76         search_query_normalized = normalizeInput(search_query)
77     elif search_key == 'height':
78         try:
79             # Si se trata de un valor referido a altura, lo convertimos a float
80             search_query_normalized = float(search_query)
81         except ValueError:
82             print("Error: El valor de altura para búsqueda binaria debe ser un número válido.")
83             return None
84     else:
85         print(f"Criterio de búsqueda '{search_key}' no válido para búsqueda binaria. Use 'name' o 'height'.")
86         return None
```

Esta función comienza con una comprobación, en la línea 57, que revisa si la lista de pokemons viene vacía. Si este escenario ocurre, se imprime un mensaje informativo por consola y se retorna el None, finalizando la ejecución de la función. Posteriormente, en la línea 63, se realiza el ordenamiento necesario y crítico para el correcto funcionamiento de cualquier algoritmo de búsqueda binaria. Para ello, se aprovecha la función de quick_sort presente en el archivo funciones_ordenamiento.py. Dicha función tiene una variable seteada en True por defecto, para generar orden ascendente. La lista ordenada se guarda dentro de la variable pokemons_list_sorted.

Una vez ordenado, se procede a establecer los límites del bucle que se realizará posteriormente. Para ello, se inicializan las variables low, en cero, y high, en len(pokemons_list_sorted) - 1. Esto significa que se obtiene el tamaño total de la lista y luego se le resta 1, ya que los índices de las listas comienzan en 0. De esta forma, tenemos nuestros punteros para comenzar con el bucle while.

Previo a dicha iteración, optamos por normalizar el valor de search_query, que corresponde al dato ingresado por el usuario dentro de la ejecución principal del programa. En el caso de búsquedas por nombre, se usa la función normalizeInput y, en el caso de búsquedas por

altura, se convierte el valor a float. En la línea 84 se considera el caso de que las llaves ingresadas sean inválidas. Si bien en nuestro programa esto no ocurre, ya que se pasan las llaves al llamar a la función, se realizó esta comprobación contemplando el escenario de futuras mejoras que impliquen la utilización de esta función para otros casos de uso, con otras keys.

```
88 ... # ----- Bucle Principal de la Búsqueda Binaria -----
89
90 ... # El bucle 'while' continuará mientras 'low' sea menor o igual que 'high'.
91 ... # Esto significa que todavía hay un rango de elementos para examinar:
92 ✓ ... while low <= high:
93 ...     # Paso 1: Calcular el punto medio (índice 'mid')
94 ...     # 'mid' es el índice del elemento central en el segmento actual de la lista.
95 ...     # Usamos división entera '//' para asegurar que 'mid' sea un índice válido
96 ...     mid = (low + high) // 2
97
98 ...     # Paso 2: Obtener el pokemon en la posición media
99 ...     current_pokemon = pokemons_list_sorted[mid]
100
101 ...     # Extraemos el valor del pokemon actual basándonos en la clave de búsqueda
102 ...     # Usamos .get() para manejar el caso de que la clave no exista.
103 ...     current_value = current_pokemon.get(search_key)
104
105 ...     # Manejamos el caso de que el valor no exista para este pokemon
106 ✓ ...     if current_value is None:
107 ...         print(f"Advertencia: pokemon en índice {mid} no tiene la clave '{search_key}'. No se puede continuar la búsqueda.")
108 ...         return None
109
110 ...     # Paso 3: Normalizar el valor del pokemon actual para la comparación.
111 ...     normalized_current_value = current_value
112 ✓ ...     if search_key == 'name':
113 ...         normalized_current_value = normalizeInput(str(current_value))
114
115 ...     # Si search_key es 'height', el valor de current_value ya es numérico (int/float),
116 ...     # por lo que no necesita una normalización adicional.
117
118 ...     # Acá decidimos qué mitad descartar
119 ✓ ...     if normalized_current_value == search_query_normalized:
120 ...         # El valor del medio coincide con lo que buscamos. Devolvemos el resultado.
121 ...         print(f"{search_query} encontrado por {getKeyTranslation(search_key)}: {current_pokemon}")
122 ...         return current_pokemon
123
124 ✓ ...     elif normalized_current_value < search_query_normalized:
125 ...         # El valor en el medio es MENOR que lo que buscamos.
126 ...         # Esto significa que el pokemon buscado (si existe) debe estar en la mitad DERECHA de la lista.
127 ...         # Por lo tanto, movemos el límite inferior ('low') para empezar la búsqueda después del medio.
128 ...         low = mid + 1
129
130 ✓ ...     else: # normalized_current_value > search_query_normalized
131 ...         # El valor en el medio es MAYOR que lo que buscamos.
132 ...         # Esto significa que el pokemon buscado (si existe) debe estar en la mitad IZQUIERDA de la lista.
133 ...         # Por lo tanto, movemos el límite superior ('high') para terminar la búsqueda antes del medio.
134 ...         high = mid - 1
135
136 ...     # Si el bucle 'while' termina, significa que 'low' se volvió mayor que 'high'.
137 ...     # Esto ocurre cuando el rango de búsqueda se ha reducido a cero o menos,
138 ...     # lo que indica que el elemento buscado no fue encontrado en la lista.
139 ...     print(f"{search_query} no encontrado por {getKeyTranslation(search_key)}.")
140 ...     return None
141 ...
```

A partir de la línea 92, comienza la ejecución del bucle while, que permanecerá activo mientras 'low' sea menor o igual a 'high'. Esto significa que se seguirá iterando mientras haya al menos un elemento dentro de la lista. En la línea 96, establecemos el valor de 'mid', variable que contiene el índice del elemento central. Nos permite dividir la lista por la mitad, para hacer las búsquedas en grupos más reducidos, de forma más eficiente. En la línea 103 se guarda el valor del pokemon que marca la mitad de la lista, y se analiza si coincide con el pokemon buscado, manejando errores derivados del uso incorrecto de las keys (líneas

106-108). Antes de realizar las comprobaciones, normalizamos los valores utilizando funciones de utils creadas específicamente con ese fin.

En la línea 119 se revisa si el valor medio es el valor buscado. De ser el caso, se devuelve el pokemon, de lo contrario, se pasa al bloque elif de la línea 124. Se realiza la comprobación para saber si el valor medio es menor que el valor buscado. Si esto es correcto, se mueve el límite inferior a $\text{mid} + 1$, para empezar la búsqueda después del medio. Si esto no es correcto, se pasa al bloque else de la línea 130, y se reduce el límite high a $\text{mid} - 1$, para terminar la búsqueda antes del medio.

El bucle terminará cuando se encuentre una coincidencia, o cuando 'low' sea mayor que 'high', implicando que no quedan más elementos por recorrer y que no se ha hallado el resultado. En este último escenario, se mostrará un mensaje informativo por consola, aprovechando la función getKeyTranslation para traducir las llaves al castellano.

Funciones dentro de directorio utils

```
utils > string_utils.py > ...  
You, 1 second ago | 1 author (You)  
1  def normalizeInput(input):  
2      # Normaliza la entrada del usuario pasando todo a minúsculas y eliminando espacios en blanco  
3      return input.lower().strip()  
4  
5  def getKeyTranslation(key):  
6      # Recibe una key y devuelve su traducción en castellano  
7      if key == 'name':  
8          return 'Nombre'  
9      elif key == 'height':  
10         return 'Altura'
```

normalizeInput

Esta función recibe un string, lo transforma en minúsculas y le quita espacios en blanco, y devuelve el resultado de dicha transformación. Permite normalizar las entradas del usuario para simplificar las comparaciones dentro de las funciones de búsqueda.

getKeyTranslation

Esta función recibe una llave del diccionario de pokemons y devuelve su traducción al castellano. Es útil para los prints informativos dentro de las funciones de búsqueda y ordenamiento.

Algoritmos de ordenamiento

A continuación se explicarán a detalle las funciones de ordenamiento elegidas: ordenamiento por burbuja (Bubble Sort) y ordenamiento rápido (Quick Sort). Para

implementar los ordenamientos elegidos se trabajó en el archivo funciones_ordenamiento.py.

Bubble Sort

El algoritmo Bubble Sort se diseñó para reordenar una lista de elementos que compara pares adyacentes y los intercambia si no están en el orden deseado. El propósito principal es ordenar una lista de diccionarios de pokemon según una clave específica, en este caso se decidió por name o height (nombre o altura) y en orden ascendente. En cada "iteración", el elemento más grande (o más pequeño) se "burbujea" hasta su posición final, reduciendo la porción desordenada de la lista. A pesar de su facilidad de implementación, el Bubble Sort presenta una eficiencia limitada, especialmente con grandes volúmenes de datos, debido a su complejidad temporal de $O(N^2)$.

A continuación se adjunta la función con comentarios.

```
def bubble_sort(arr, key, ascending=True):
    n = len(arr) # para obtener el numero total de la lista. Basicamente la longitud de la lista que usaremos en la busqueda
    arr_copy = arr[:] #copiamos la lista por las dudas no interfiera con la lista principal de los pokemon
    # for exterior Define hasta dónde tiene que trabajar el bucle j.

    for i in range(n - 1): # Controla cuántas pasadas principales se necesitan para que todos los grandes lleguen a su lugar final
        for j in range(n - 1 - i): # El n - 1 es porque siempre comparamos un Pokémon con el de al lado (j con j+1)
            #accedemos a los elementos que tiene al lado el pokemon encontrado al recorrer
            pokemon1 = arr_copy[j]
            pokemon2 = arr_copy[j + 1]
            #despues extraemos los valores para comparar usando el argumento key
            # manejamos errores con try except en caso de que la clave no exista
            try:
                if key == 'name':
                    value1 = pokemon1[key].lower() # Convertimos a minúsculas para evitar problemas de mayúsculas/minúsculas
                    value2 = pokemon2[key].lower()
                else:
                    value1 = int(pokemon1[key])
                    value2 = int(pokemon2[key]) #si no convertimos a int los datos vienen como strings haciendolos difícil de comparar
            except KeyError:
                print(f" Error: la clave {key} no se encontró en uno o ambos diccionarios de pokemones")
                return arr_copy # devolvemos lo que hay hasta el momento si hay un error

            #Realizar la comparación e intercambio
            if (ascending and value1 > value2) or (not ascending and value1 < value2):
                # Intercambio de los elementos en la lista
                # (el lado derecho) son los nuevos valores que quiero poner
                # en esas posiciones, pero en el orden inverso.
                # valúa ambos valores del lado derecho. Una vez que tiene arr_copy[j + 1] y arr_copy[j] listos
                # los asigna a las posiciones correspondientes en el lado izquierdo.
                arr_copy[j], arr_copy[j + 1] = arr_copy[j + 1], arr_copy[j]

    return arr_copy # se devuelve la lista ordenada
```

La función toma tres parámetros:

- arr: la lista de diccionarios de pokemon a ordenar.
- key: La clave del diccionario por la que se realiza el ordenamiento
- ascending: un booleano, True para ascendente y False para descendente.

Primero se obtiene la cantidad de elementos en la lista 'arr'. Con este valor se puede controlar los bucles de iteración. Luego con `arr_copy = arr[:]` se crea una copia de la lista original para asegurar que se trabajará solo sobre ella. Luego se define el bucle externo `for i in range (n -1)`. En este bucle externo se controla el número de pasadas completas que se realizará sobre la lista. Luego en el bucle interior se realizan los intercambios. En cada pasada 'i', el elemento más grande (o más pequeño dependiendo del orden) se ubica (burbujea) hasta su posición correcta al final de la parte no ordenada de la lista. Se usa `n -1` porque no necesitamos comparar el último elemento de la lista.

En el bucle interno `for j in range (n-1-i)` se comparan los pares de elementos adyacentes y se hacen los intercambios. El `n -1` evita el error de índice fuera de rango porque compara `j` con `j + 1`. `-i` se utiliza porque en cada pasada del bucle externo los `i` últimos ya están en su lugar y no necesitan ser revisados de nuevo, por lo que el rango se acorta.

Dentro de `j` se accede al primer pokémon del par actual a comparar. (`pokemon1 = arr_copy[j]`). Después se accede al segundo pokémon del par, el que está al lado de `pokemon1`: `pokemon2 = arr_copy[j + 1]`

Luego se manejan los errores con un `try-except` si la `key` no existe en ningún diccionario. Dentro se encuentra la validación de `key`. Si la clave es `name`, se obtiene el nombre y lo convierte a minúsculas para asegurar que la comparación sea insensible a mayúsculas y minúsculas. Se hace lo mismo para el segundo valor.

En `else`, si la clave `key` se asume es un valor numérico, como `height`, se convierte el valor de la `key` del primer pokémon a un entero. Esto es necesario porque los datos de la API pueden venir como strings numéricos y se necesita compararlos como números. Se hace lo mismo para el `value2`.

El `except KeyError` imprime un mensaje de error claro si la clave no existe. Después se devuelve la copia de la lista tal como está en ese momento para evitar errores mayores. El próximo error imprime un mensaje de error si la clave no es convertible a número.

`if (ascending and value1 > value2) or (not ascending and value1 < value2)`: Aquí se determina si se debe hacer un intercambio. Si `ascending` es `True` se intercambia si `value1` es mayor que `value2` y viceversa.

En `arr_copy[j], arr_copy[j + 1] = arr_copy[j + 1], arr_copy[j]` se intercambian los valores.

Una vez que todos los bucles han terminado, la lista 'arr_copy' está completamente ordenada y es devuelta por la función.

Quick Sort

El algoritmo Quick Sort (Ordenamiento Rápido) se eligió porque es uno de los algoritmos de ordenamiento más eficientes y ampliamente utilizados, especialmente para grandes volúmenes de datos. Utiliza 'divide y vencerás', lo que significa que divide un problema grande (listas grandes, de pokemon en este caso) en subproblemas más pequeños y fáciles de resolver y luego combina las soluciones. Su proceso implica seleccionar un elemento como pivote y particionar la lista en dos sublistas: una con elementos menores que el pivote y otra con elementos mayores. La clave de su funcionamiento reside en la recursión, ya que el algoritmo se aplica a estas sublistas de forma repetida hasta que cada sublista se reduce a un solo elemento o queda vacía, momento en el cual se considera ordenada y se combinan las sublistas para formar el resultado final. Gracias a esta aproximación recursiva, Quick Sort logra una complejidad temporal promedio de $O(N\log N)$.

Se define la función con los mismos parámetros que bubble sort, arr, key, ascending true. Se utiliza `if len(arr) <= 1: return arr` como caso base de la recursión que se implementará después. Si la lista tiene 0 o 1 elemento significa que ya está ordenada. Esto hace que la recursión se detenga. Luego se devuelve la lista tal cual ya que no queda nada que ordenar.

Uno de los elementos más importantes para la eficiencia del algoritmo de Quick Sort es el pivote. Se selecciona un pivote y el elemento del medio de la lista. Se selecciona `pivot = arr[len(arr) // 2]` como el pivote al partir la lista a la mitad. Con el manejo de errores `Try except` se asegura de extraer de forma segura el valor del pivote según la key.

Las líneas `if key == 'name': else: pivot_value = int(float(pivot[key]))` determinan si la clave es name y se obtiene el nombre del pivote y lo convierte en minúsculas como en bubble sort. Si la clave no es name convierte el valor a flotante y luego a entero. Los `except` manejan los mensajes de error si la clave no existe en el elemento pivote o si el elemento no es convertible al tipo numérico esperado. Después `return: arr` devuelve la lista sin ordenar si el valor del pivote es invalido.

A continuación se crean las listas para almacenar los elementos menores iguales o mayores que el pivote. Después se itera sobre cada pokémon en la lista original para clasificarlo en left, middle o right según la posición del pivote.

Después se itera sobre cada pokémon en la lista original para clasificarlo en left, middle o right. El `try-except` siguiente asegura que se puedan obtener y preparar el valor de cada pokémon para la comparación y se maneja cualquier dato inconsistente.

En las siguientes líneas se obtiene y normaliza el nombre del pokemon y si no se obtiene y convierte el valor numérico del pokémon actual.

```
if key == 'name':  
    current_value = pokemon[key].lower()  
else:  
    current_value = int(float(pokemon[key]))
```

Luego el except determina un mensaje de error si un elemento no puede ser procesado y se devuelve la lista sin ordenar. Finalmente se pasa a las comparaciones. Si el valor del pokémon actual es igual al valor del pivote, lo añade a la lista middle, si el valor es menor que el del pivote va a left y si es mayor (de forma ascendente) va a right. Se usa .append() para agregar los pokemones al final de la lista existente.

El último paso es la llamada recursiva final que combina las sub listas ordenadas. Al llamar a quick_sort(left, key, ascending) + middle + quick_sort(right, key, ascending) se ordena recursivamente la sublista de elementos 'left'. El operador '+' concatena estas tres listas en el orden correcto formando la lista final ordenada.

Se adjunta el código comentado de la función:

```
def quick_sort(arr, key, ascending=True):
    #caso base: si la lista es vacía o tiene un solo elemento, ya está ordenada
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2] # Elegimos el pivote como el elemento del medio
    try:
        if key == 'name':
            pivot_value = pivot[key].lower()
        else:
            pivot_value = int(float(pivot[key]))
    except KeyError:
        print(f"Error: la clave '{key}' no se encontró en el pivote ({pivot.get('name', 'desconocido')}).")
        return arr
    except (ValueError, TypeError):
        print(f"Error: el valor del pivote para la clave '{key}' no es numérico en '{pivot.get('name', 'desconocido')}'.")
        return arr

    left = [] #pokemones a la izquierda del pivote
    middle = [] # pokemones que son iguales al pivote
    right = [] # pokemones a la derecha del pivote

    for pokemon in arr:
        try:
            if key == 'name':
                current_value = pokemon[key].lower() # Convertimos a minúsculas para evitar problemas de mayúsculas/minúsculas
            else:
                # Convertimos el valor a int, manejando posibles errores de conversión
                current_value = int(float(pokemon[key])) # el valor actual del pokemon que se comparará
        except (KeyError, ValueError, TypeError):
            print(f"Error en Quicksort: la clave {key} no se encontró o su valor no es numérico para el elemento ({pokemon.get('name', 'desconocido')}).")
            return arr

    # Comparamos el valor actual con el valor del pivote si no pasamos al siguiente pokemon.
    if current_value == pivot_value:
        middle.append(pokemon)
    elif (ascending and current_value < pivot_value) or (not ascending and current_value > pivot_value):
        left.append(pokemon)
    else:
        right.append(pokemon)

    return quick_sort(left, key, ascending) + middle + quick_sort(right, key, ascending)
```

Función print_pokemon_list

Además de los algoritmos de búsqueda y ordenamiento, el proyecto utiliza una función auxiliar `print_pokemon_list`. Esta función ayuda a visualizar de forma clara los resultados de las operaciones, especialmente las listas de pokémon después de haber sido ordenadas. En esta función también se realiza la conversión de decímetros a centímetros para mostrar el orden por altura de los pokemones ya que la lista tiene las alturas en decímetros originalmente. Tener esta función también ayuda a mantener claridad en el archivo principal `main.py`. También permitió hacer pruebas en las funciones de ordenamiento sin interferir en la ejecución principal.

Se adjunta el código de la función:

```
def print_pokemon_list(pokemon_list, title="lista de pokemon", limit=10):
    print(f"\n--- {title} ---") #
    if not pokemon_list:
        print("La lista de Pokémon está vacía.")
        return
    #le decimos hasta donde el limite para la devolucion y convertimos a cm porque los datos vienen en decimetros.
    for i, p in enumerate(pokemon_list[:limit]):
        name = p.get('name', 'nombre desconocido') #el segundo valor es para asegurarnos que muestre algo si hay error
        height_dm = p.get('height', 'altura desconocida')
        height_cm = 'altura desconocida'
        if isinstance(height_dm, (int,float)): #preguntamos si heightdm es int o float
            height_cm = height_dm * 10
        elif isinstance(height_dm, str) and height_dm.isdigit(): #preguntamos si es string y si ese string son digitos
            height_cm = int(height_dm) * 10
    print(f" - {name}: {height_cm} cm")
```

4. Metodología Utilizada

El proceso de desarrollo de este proyecto se inició con una fase de investigación previa, centrada en la comprensión y profundización de los algoritmos de búsqueda (lineal y binaria) y de ordenamiento (Bubble Sort y Quick Sort). Para ello, se consultaron fuentes académicas de la Universidad Tecnológica Nacional, junto con la investigación de recursos específicos necesarios para el desarrollo del programa, que incluyen documentación de la Poké API y de las librerías `timeit`, `requests` y `pprint` de Python.

Para la implementación, se utilizó Python en su última versión, Visual Studio Code como Entorno de Desarrollo Integrado (IDE), Git y Github como herramientas para el control de versiones, Poké API como API proveedora de datos, y las librerías `timeit` (para medir tiempos de ejecución), `requests` (para hacer las llamadas a la API) y `pprint` (para imprimir más ordenadamente resultados en consola).

El diseño del desarrollo se llevó a cabo con un código base inicial, creado de manera conjunta, orientado a dar un funcionamiento básico al programa. Posteriormente, se optó por la división del trabajo en dos partes, una referida a las funciones de ordenamiento y otra dedicada a los algoritmos de búsqueda. Para dicha etapa se eligió el trabajo colaborativo en simultáneo, con la creación de dos ramas que se bifurcaban de la principal (`main`), para continuar los desarrollos respectivos en paralelo y de forma ordenada.

Commits

main	All users	All time
Commits on Jun 7, 2025		
Ajustes del menu de usuario y ordenamientos belenyb committed 3 hours ago	b0944e8	🔗 <>
Merge rama-busqueda y conflictos resueltos en main.py belenyb committed 3 hours ago	2263c90	🔗 <>
merge ordenamientos y menu arturokaadu committed 3 hours ago	42e26fe	🔗 <>
Merge branch 'rama-ordenamientos' arturokaadu committed 3 hours ago	1ba9d2c	🔗 <>
imported time arturokaadu committed 3 hours ago	2a4de0c	🔗 <>
README.md ajustado segun requerimientos - falta detalle de funciones y reflexiones belenyb committed 3 hours ago	b32c1d6	🔗 <>
funciones 5 y 6 arturokaadu committed 4 hours ago	a53c0a0	🔗 <>
Commits on Jun 5, 2025		

Una vez alcanzada la totalidad del código, se procedió a unir ambos desarrollos en la rama principal del repositorio, y realizar las pruebas correspondientes, con el objetivo de evaluar el rendimiento de los algoritmos en distintas condiciones de tamaño y naturaleza de los datos.

5. Resultados Obtenidos

Casos de prueba realizados

A. Pruebas de algoritmos de búsqueda por nombre

Las funciones de búsqueda por nombre ofrecen resultados de interpretación más directa, ya que realizan operaciones con identificadores únicos, a diferencia de las búsquedas por altura, que pueden tener más de una coincidencia encontrada. Esta característica permite que el análisis se centre exclusivamente en la duración temporal de cada algoritmo, sin presentar variaciones en los resultados que interfieran con la evaluación integral del rendimiento.

1. La primera prueba realizada para analizar el comportamiento de estos algoritmos fue sobre un total de 30 elementos, logrando un tiempo de ejecución de 0.0006 segundos para la búsqueda lineal y 0.0003 segundos para la búsqueda binaria. Esta última fue aproximadamente dos veces más rápida que la lineal para hallar el mismo elemento. Observando la posición del elemento correspondiente a 'arbok' en la lista original, este representa un escenario cercano al peor caso, ya que la función debe recorrer casi todos los elementos hasta encontrar la coincidencia. La búsqueda binaria, por el contrario, al operar sobre una lista previamente ordenada, logra ubicar el elemento mucho más eficientemente, llegando al resultado en la mitad del tiempo.

```
tp-integrador-programacion git:(rama-busqueda) x /usr/local/bin/python /Users/belenyardebuller/code/UTN/ayso/tp-integrador-programaci
Obteniendo pokemons...
Listado de pokemons obtenido exitosamente
[{'height': 7, 'name': 'bulbasaur'},
 {'height': 10, 'name': 'ivysaur'},
 {'height': 20, 'name': 'venusaur'},
 {'height': 6, 'name': 'charmander'},
 {'height': 11, 'name': 'charmeleon'},
 {'height': 17, 'name': 'charizard'},
 {'height': 5, 'name': 'squirtle'},
 {'height': 10, 'name': 'wartortle'},
 {'height': 16, 'name': 'blastoise'},
 {'height': 3, 'name': 'caterpie'},
 {'height': 7, 'name': 'metapod'},
 {'height': 11, 'name': 'butterfree'},
 {'height': 3, 'name': 'weedle'},
 {'height': 6, 'name': 'kakuna'},
 {'height': 10, 'name': 'beedrill'},
 {'height': 3, 'name': 'pidgey'},
 {'height': 11, 'name': 'pidgeotto'},
 {'height': 15, 'name': 'pidgeot'},
 {'height': 3, 'name': 'rattata'},
 {'height': 7, 'name': 'raticate'},
 {'height': 3, 'name': 'spearow'},
 {'height': 12, 'name': 'fearow'},
 {'height': 20, 'name': 'ekans'},
 {'height': 35, 'name': 'arbok'},
 {'height': 4, 'name': 'pikachu'},
 {'height': 8, 'name': 'raichu'},
 {'height': 6, 'name': 'sandshrew'},
 {'height': 10, 'name': 'sandslash'},
 {'height': 4, 'name': 'nidoran-f'},
 {'height': 8, 'name': 'nidorina'}]
Ingrese la operacion que desea realizar:
0. Salir
1. Búsqueda por nombre
2. Búsqueda por altura
3. Ordenamiento por altura
4. Ordenamiento por nombre: 1
Por favor, ingrese el pokemon que desea buscar: arbok

--- Ejecutando Búsqueda Lineal por Nombre: ---
'arbok' encontrado: {'name': 'arbok', 'height': 35}
Tiempo de ejecución para linear_search por nombre: 0.0006068040096324682 segundos

--- Ejecutando Búsqueda Binaria por Nombre: ---
'arbok' encontrado por Nombre: {'name': 'arbok', 'height': 35}
Tiempo de ejecución para binary_search: 0.00030899999546818435 segundos

--- Fin de la ejecución de la opción 1 ---
```

2. En la segunda prueba realizada, se optó por subir a 100 el número total de elementos de la lista, para entender el impacto que tiene el tamaño de la lista en el tiempo que tardan los algoritmos en encontrar el objetivo. Los resultados evidencian que, cuanto más grande sea la lista, más tiempo tardará el algoritmo de búsqueda lineal en encontrar el elemento deseado, ya que tendrá que comprobar cada elemento de la lista hasta encontrar el que busca. Contrariamente, la búsqueda binaria será más eficiente puesto que el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista.

```
--- Ejecutando Búsqueda Lineal por Nombre: ---  
'dodrio' encontrado: {'name': 'dodrio', 'height': 18}  
Tiempo de ejecución para linear_search por nombre: 0.003217374993255362 segundos  
  
--- Ejecutando Búsqueda Binaria por Nombre: ---  
'dodrio' encontrado por Nombre: {'name': 'dodrio', 'height': 18}  
Tiempo de ejecución para binary_search: 0.0007408750243484974 segundos  
  
--- Fin de la ejecución de la opción 1 ---
```

3. En la tercera prueba realizada, sobre el mismo total de pokemons (100), se optó por usar el parámetro de búsqueda 'charmander' para analizar el mejor caso posible de las búsquedas lineales. El resultado se condice con la teoría, que indica que la búsqueda lineal es más efectiva cuando el elemento a buscar es cercano a las primeras posiciones de la lista, es decir, cercano al mejor caso. En este ejemplo, la búsqueda lineal fue notoriamente más rápida, ejecutándose en 0.0005 segundos, contra los 0.0013 segundos de ejecución de la búsqueda binaria.

```
--- Ejecutando Búsqueda Lineal por Nombre: ---  
'charmander' encontrado: {'name': 'charmander', 'height': 6}  
Tiempo de ejecución para linear_search por nombre: 0.0005455419886857271 segundos  
  
--- Ejecutando Búsqueda Binaria por Nombre: ---  
'charmander' encontrado por Nombre: {'name': 'charmander', 'height': 6}  
Tiempo de ejecución para binary_search: 0.0013169999874662608 segundos
```

Estos resultados se encuentran detallados en la tabla comparativa del “Anexo III: Búsquedas por nombre: comparativa de tiempos de ejecución (en segundos)”.

B. Pruebas de algoritmos de búsqueda por altura

Al realizar búsquedas por altura, es común encontrar más de una coincidencia posible, ya que varios pueden compartir la misma estatura. Esta característica, visiblemente diferente a

buscar por nombre, un valor que es único, explica por qué los resultados de la búsqueda lineal y la búsqueda binaria pueden diferir para una misma consulta.

1. En este primer caso, se observa que, al buscar un pokemon con una altura de 10 decímetros dentro de un total de 30 elementos, la búsqueda lineal devolvió a 'ivysaur', mientras que la búsqueda binaria halló a 'beedrill'. La razón de esta diferencia radica en los requisitos de ordenamiento de cada algoritmo. La búsqueda lineal inspecciona la lista secuencialmente desde el principio, y devuelve la primera coincidencia que encuentra. En la lista original, 'ivysaur' era el primer pokemon con altura 10. La búsqueda binaria, en cambio, exige que la lista esté previamente ordenada por el criterio de búsqueda; en este caso, altura. Al ordenar la lista, la posición de los pokemon cambia, devolviendo 'beedrill' como la primera coincidencia encontrada.

```
tp-integrador-programacion git:(rama-busqueda) x /usr/local/bin/python /Users/belenyardebuller/code/UTN/ayso/tp-integrador-programacion
Obteniendo pokemons...
Listado de pokemons obtenido exitosamente
[{'height': 7, 'name': 'bulbasaur'},
 {'height': 10, 'name': 'ivysaur'},
 {'height': 20, 'name': 'venusaur'},
 {'height': 6, 'name': 'charmander'},
 {'height': 11, 'name': 'charmeleon'},
 {'height': 17, 'name': 'charizard'},
 {'height': 5, 'name': 'squirtle'},
 {'height': 10, 'name': 'wartortle'},
 {'height': 16, 'name': 'blastoise'},
 {'height': 3, 'name': 'caterpie'},
 {'height': 7, 'name': 'metapod'},
 {'height': 11, 'name': 'butterfree'},
 {'height': 3, 'name': 'weedle'},
 {'height': 6, 'name': 'kakuna'},
 {'height': 10, 'name': 'beedrill'},
 {'height': 3, 'name': 'pidgey'},
 {'height': 11, 'name': 'pidgeotto'},
 {'height': 15, 'name': 'pidgeot'},
 {'height': 3, 'name': 'rattata'},
 {'height': 7, 'name': 'raticate'},
 {'height': 3, 'name': 'spearow'},
 {'height': 12, 'name': 'fearow'},
 {'height': 20, 'name': 'ekans'},
 {'height': 35, 'name': 'arbok'},
 {'height': 4, 'name': 'pikachu'},
 {'height': 8, 'name': 'raichu'},
 {'height': 6, 'name': 'sandshrew'},
 {'height': 10, 'name': 'sandslash'},
 {'height': 4, 'name': 'nidoran-f'},
 {'height': 8, 'name': 'nidorina'}]
Ingrese la operacion que desea realizar:
0. Salir
1. Búsqueda por nombre
2. Búsqueda por altura
3. Ordenamiento por altura
4. Ordenamiento por nombre: 2
Por favor, ingrese la altura que desea buscar (en decímetros): 10

--- Ejecutando Búsqueda Lineal por Altura: ---
Pokémon con altura '10.0' encontrado: {'name': 'ivysaur', 'height': 10}
Tiempo de ejecución para linear_search por altura: 0.0001435000158380717 segundos

--- Ejecutando Búsqueda Binaria por Altura: ---
'10' encontrado por Altura: {'name': 'beedrill', 'height': 10}
Tiempo de ejecución para binary_search: 0.0002930420159827918 segundos
```

Respecto a los tiempos de ejecución, se puede observar que la búsqueda lineal fue ligeramente más rápida, aunque esto fue claramente influenciado por la posición del primer elemento coincidente, muy cercana a la posición inicial del bucle.

2. Para analizar en mayor profundidad los tiempos de ejecución de cada operación, se realizó una nueva búsqueda, ingresando el valor 4 como altura. Mediante esta

búsqueda, y teniendo en cuenta que el primer pokemon de altura 4 en la lista original se encontraba en una posición más alejada de la inicial, se esperaba que la búsqueda binaria fuera más veloz. Los resultados obtenidos fueron los siguientes:

```
/usr/local/bin/python /Users/belenyardebuller/code/UTN/ayso/tp-integrador-programacion/main.py
○ → tp-integrador-programacion git:(rama-busqueda) x /usr/local/bin/python /Users/belenyardebuller/code/UTN/ayso/tp-integrador-programacion
Obteniendo pokemons...
Listado de pokemons obtenido exitosamente
[{'height': 7, 'name': 'bulbasaur'},
 {'height': 10, 'name': 'ivysaur'},
 {'height': 20, 'name': 'venusaur'},
 {'height': 6, 'name': 'charmander'},
 {'height': 11, 'name': 'charmeleon'},
 {'height': 17, 'name': 'charizard'},
 {'height': 5, 'name': 'squirtle'},
 {'height': 10, 'name': 'wartortle'},
 {'height': 16, 'name': 'blastoise'},
 {'height': 3, 'name': 'caterpie'},
 {'height': 7, 'name': 'metapod'},
 {'height': 11, 'name': 'butterfree'},
 {'height': 3, 'name': 'weedle'},
 {'height': 6, 'name': 'kakuna'},
 {'height': 10, 'name': 'beedrill'},
 {'height': 3, 'name': 'pidgey'},
 {'height': 11, 'name': 'pidgeotto'},
 {'height': 15, 'name': 'pidgeot'},
 {'height': 3, 'name': 'rattata'},
 {'height': 7, 'name': 'raticate'},
 {'height': 3, 'name': 'spearow'},
 {'height': 12, 'name': 'fearow'},
 {'height': 20, 'name': 'ekans'},
 {'height': 35, 'name': 'arbok'},
 {'height': 4, 'name': 'pikachu'},
 {'height': 8, 'name': 'raichu'},
 {'height': 6, 'name': 'sandshrew'},
 {'height': 10, 'name': 'sandslash'},
 {'height': 4, 'name': 'nidoran-f'},
 {'height': 8, 'name': 'nidorina'}]
Ingrese la operacion que desea realizar:
0. Salir
1. Búsqueda por nombre
2. Búsqueda por altura
3. Ordenamiento por altura
4. Ordenamiento por nombre: 2
Por favor, ingrese la altura que desea buscar (en decímetros): 4

--- Ejecutando Búsqueda Lineal por Altura: ---

Pokémon con altura '4.0' encontrado: {'name': 'pikachu', 'height': 4}
Tiempo de ejecución para linear_search por altura: 0.0006747920124325901 segundos

--- Ejecutando Búsqueda Binaria por Altura: ---

'4' encontrado por Altura: {'name': 'nidoran-f', 'height': 4}
Tiempo de ejecución para binary_search: 0.000310208008158952 segundos
```

La información impresa en consola validó la hipótesis inicial, mostrando un tiempo de ejecución menor en el caso de la búsqueda binaria por altura. La búsqueda lineal, por su parte, tardó el doble de tiempo en hallar el primer pokemon con estatura de 4 decímetros.

C. Algoritmos de ordenamiento por altura

Para llevar a cabo las pruebas de comparación de ordenamientos por altura se decidió por hacer el llamado a ambas funciones de ordenamiento, Bubble Sort y Quick Sort para así medir adecuadamente el tiempo de ejecución de cada una. Los resultados serán devueltos de forma ascendente en centímetros, gracias a que los datos se convirtieron de decímetros (como vienen por defecto en la lista) a centímetros, lo que permite evidenciar con más claridad los resultados obtenidos.

Prueba 1: 150 pokemon

Primero se procede a hacer pruebas con 150 pokemon pedidos a la API. Esto traerá el listado de pokémon pedido para cada algoritmo de ordenamiento ejecutado y demostrará cuál es el más conveniente para un conjunto de datos lo suficientemente amplio para marcar las diferencias en tiempos de ejecución de cada uno.

```
- raticate: 70 cm
- dugtrio: 70 cm
- growlithe: 70 cm
- bellspout: 70 cm
- raichu: 80 cm
- nidorina: 80 cm
- zubat: 80 cm
- gloom: 80 cm
- psyduck: 80 cm
- machop: 80 cm
- farfetchd: 80 cm
- staryu: 80 cm
- jolteon: 80 cm
- porygon: 80 cm
- nidorino: 90 cm
- abra: 90 cm
- tentacool: 90 cm
- grimer: 90 cm
- magikarp: 90 cm
- flareon: 90 cm
- ivysaur: 100 cm
- wartortle: 100 cm
- beedrill: 100 cm
- sandslash: 100 cm
- wigglytuff: 100 cm
- parasect: 100 cm
- venonat: 100 cm
- persian: 100 cm
- primeape: 100 cm
- poliwhirl: 100 cm
- weepinbell: 100 cm
- graveler: 100 cm
- ponyta: 100 cm
- magneton: 100 cm
- drowzee: 100 cm
- marowak: 100 cm
- rhyhorn: 100 cm
- tangeela: 100 cm
```

```
--- Pokémon ordenados por altura (ascendente) con Bubble Sort ---
- diglett: 20 cm
- caterpie: 30 cm
- weedle: 30 cm
- pidgey: 30 cm
- rattata: 30 cm
- spearow: 30 cm
- paras: 30 cm
- magnemite: 30 cm
- shelder: 30 cm
- ditto: 30 cm
- eevee: 30 cm
- pikachu: 40 cm
- nidoran-f: 40 cm
- meowth: 40 cm
- geodude: 40 cm
- krabby: 40 cm
- exeggute: 40 cm
- cubone: 40 cm
- horsea: 40 cm
- omanyte: 40 cm
- squirtle: 50 cm
- nidoran-m: 50 cm
- jigglypuff: 50 cm
- oddish: 50 cm
- mankey: 50 cm
- voltorb: 50 cm
- kabuto: 50 cm
- charmander: 60 cm
- kakuna: 60 cm
- sandshrew: 60 cm
- clefairy: 60 cm
- vulpix: 60 cm
- poliwag: 60 cm
- koffing: 60 cm
- goldeen: 60 cm
- bulbasaur: 70 cm
```

```
- gengar: 150 cm
- hitmonlee: 150 cm
- scyther: 150 cm
- pinsir: 150 cm
- blastoise: 160 cm
- golbat: 160 cm
- machop: 160 cm
- tentacruel: 160 cm
- slowbro: 160 cm
- haunter: 160 cm
- hypno: 160 cm
- zapdos: 160 cm
- charizard: 170 cm
- golduck: 170 cm
- victreebel: 170 cm
- rapidash: 170 cm
- dewgong: 170 cm
- articuno: 170 cm
- dodrio: 180 cm
- aerodactyl: 180 cm
- dratini: 180 cm
- arcanine: 190 cm
- rhydon: 190 cm
- venusaur: 200 cm
- ekans: 200 cm
- exeggutor: 200 cm
- moltres: 200 cm
- mewtwo: 200 cm
- snorlax: 210 cm
- kangaskhan: 220 cm
- dragonite: 220 cm
- lapras: 250 cm
- arbok: 350 cm
- dragonair: 400 cm
- gyarados: 650 cm
- onix: 880 cm
```

Tiempo de ejecución para bubble_sort: 0.0016162999672815204 segundos

Tiempo de ejecución para bubble_sort: 0.0016162999672815204 segundos

```

--- Pokémon ordenados por altura (ascendente) con Quick Sort ---
- diglett: 20 cm
- caterpie: 30 cm
- weedle: 30 cm
- pidgey: 30 cm
- rattata: 30 cm
- spearow: 30 cm
- paras: 30 cm
- magnemite: 30 cm
- shelder: 30 cm
- ditto: 30 cm
- eevee: 30 cm
- pikachu: 40 cm
- nidoran-f: 40 cm
- meowth: 40 cm
- geodude: 40 cm
- krabby: 40 cm
- exeggcute: 40 cm
- cubone: 40 cm
- horsea: 40 cm
- omanyte: 40 cm
- squirtle: 50 cm
- nidoran-m: 50 cm
- jigglypuff: 50 cm
- oddish: 50 cm
- mankey: 50 cm
- voltorb: 50 cm
- kabuto: 50 cm
- charmander: 60 cm
- kakuna: 60 cm
- sandshrew: 60 cm
- clefairy: 60 cm
- vulpix: 60 cm
- poliwag: 60 cm
- koffing: 60 cm
- goldeen: 60 cm
- bulbasaur: 70 cm

```

```

- gengar: 150 cm
- hitmonlee: 150 cm
- scyther: 150 cm
- pinsir: 150 cm
- blastoise: 160 cm
- golbat: 160 cm
- machop: 160 cm
- tentacruel: 160 cm
- slowbro: 160 cm
- haunter: 160 cm
- hypno: 160 cm
- zapdos: 160 cm
- charizard: 170 cm
- golduck: 170 cm
- victreebel: 170 cm
- rapidash: 170 cm
- dewgong: 170 cm
- articuno: 170 cm
- dodrio: 180 cm
- aerodactyl: 180 cm
- dratini: 180 cm
- arcanine: 190 cm
- rhydon: 190 cm
- venusaur: 200 cm
- ekans: 200 cm
- exeggutor: 200 cm
- exeggutor: 200 cm
- moltres: 200 cm
- mewtwo: 200 cm
- snorlax: 210 cm
- kangaskhan: 220 cm
- dragonite: 220 cm
- lapras: 250 cm
- arbok: 350 cm
- dragonair: 400 cm
- gyarados: 650 cm
- onix: 880 cm
Tiempo de ejecución para quick_sort: 0.00011220003943890333 segundos

```

Tiempo de ejecución para quick_sort: 0.00011220003943890333 segundos

Los resultados obtenidos para los ordenamientos por altura fueron los siguientes:

- ❖ Tiempo de ejecución para Quick Sort: 0.00011220003943890333 segundos
- ❖ Tiempo de ejecución para Bubble Sort: 0.0016162999672815204 segundos

Al comparar los tiempos de ejecución obtenidos para 150 pokemon, se observa una diferencia clara y significativa entre ambos algoritmos:

Quick Sort fue aproximadamente 14 veces más rápido que Bubble Sort para este conjunto de datos. Específicamente, Quick Sort completó la tarea en alrededor de 0.00011 segundos, mientras que Bubble Sort necesitó aproximadamente 0.00161 segundos. Esta disparidad en el rendimiento se alinea directamente con la complejidad temporal teórica de cada algoritmo. Estos resultados confirman la teoría al demostrar que Quick Sort es considerablemente más eficiente que Bubble Sort para ordenar un listado de pokemon, incluso con un volumen de datos moderado. La diferencia en tiempos se volvería aún más pronunciada si se aumentara la cantidad de pokemon a ordenar.

Prueba 2: 30 pokemon

Se realiza la misma prueba pero con un número considerablemente menor que la primera vez para comparar si la diferencia entre un algoritmo y otro sigue siendo alta.

```
- squirtle: 50 cm
- charmander: 60 cm
- kakuna: 60 cm
- sandshrew: 60 cm
- bulbasaur: 70 cm
- metapod: 70 cm
- raticate: 70 cm
- raichu: 80 cm
- nidorina: 80 cm
- ivysaur: 100 cm
- wartortle: 100 cm
- beedrill: 100 cm
- sandslash: 100 cm
- charmeleon: 110 cm
- butterfree: 110 cm
- pidgeotto: 110 cm
- fearow: 120 cm
- pidgeot: 150 cm
- blastoise: 160 cm
- charizard: 170 cm
- venusaur: 200 cm
- ekans: 200 cm
- arbok: 350 cm
```

Tiempo de ejecución para bubble_sort: 8.420000085607171e-05 segundos

```
- squirtle: 50 cm
- charmander: 60 cm
- kakuna: 60 cm
- sandshrew: 60 cm
- bulbasaur: 70 cm
- metapod: 70 cm
- raticate: 70 cm
- raichu: 80 cm
- nidorina: 80 cm
- ivysaur: 100 cm
- wartortle: 100 cm
- beedrill: 100 cm
- sandslash: 100 cm
- charmeleon: 110 cm
- butterfree: 110 cm
- pidgeotto: 110 cm
- fearow: 120 cm
- pidgeot: 150 cm
- blastoise: 160 cm
- charizard: 170 cm
- venusaur: 200 cm
- ekans: 200 cm
- arbok: 350 cm
```

Tiempo de ejecución para quick_sort: 4.8200017772614956e-05 segundos

Los resultados obtenidos fueron los siguientes:

- ❖ Tiempo de ejecución para Bubble Sort: $8.420000085607171e-05$ segundos (0.0000842 segundos)
- ❖ Tiempo de ejecución para Quick Sort: $4.8200017772614956e-05$ segundos (0.0000482 segundos)

Quick Sort fue aproximadamente 1.75 veces más rápido que Bubble Sort. Aunque la diferencia absoluta en tiempo es muy pequeña (ambos son extremadamente rápidos para un volumen de datos tan reducido), la tendencia se mantiene: Quick Sort sigue siendo el algoritmo más eficiente. Para este tamaño de lista, la sobrecarga de la recursión en Quick Sort es mínima y su eficiencia inherente sigue siendo superior.

C. Algoritmos de ordenamiento por nombre

Las pruebas por nombre se realizaron de la misma manera que por altura, de forma ascendente y primero eligiendo 150 pokemon y luego 30, para poder evidenciar los resultados en un número moderadamente amplio como 150 y uno mucho menor como 30.

Prueba 1: 150 pokemon

```
- seadra: 120 cm
- seaking: 130 cm
- seel: 110 cm
- sheldler: 30 cm
- slowbro: 160 cm
- slowpoke: 120 cm
- snorlax: 210 cm
- spearow: 30 cm
- squirtle: 50 cm
- starmie: 110 cm
- staryu: 80 cm
- tangela: 100 cm
- tauros: 140 cm
- tentacool: 90 cm
- tentacruel: 160 cm
- vaporeon: 100 cm
- venomoth: 150 cm
- venomat: 100 cm
- venusaur: 200 cm
- victreebel: 170 cm
- vileplume: 120 cm
- voltorb: 50 cm
- vulpix: 60 cm
- wartortle: 100 cm
- weedle: 30 cm
- weepinbell: 100 cm
- weezing: 120 cm
- wigglytuff: 100 cm
- zapdos: 160 cm
- zubat: 80 cm
Tiempo de ejecución para bubble_sort: 0.0025790000217966735 segundos
```

```
- sandslrew: 60 cm
- sandslash: 100 cm
- scyther: 150 cm
- seadra: 120 cm
- seaking: 130 cm
- seel: 110 cm
- shelder: 30 cm
- slowbro: 160 cm
- slowpoke: 120 cm
- snorlax: 210 cm
- spearow: 30 cm
- squirtle: 50 cm
- starmie: 110 cm
- staryu: 80 cm
- tanglea: 100 cm
- tauros: 140 cm
- tentacool: 90 cm
- tentacruel: 160 cm
- vapoleon: 100 cm
- venomoth: 150 cm
- venomat: 100 cm
- venusaur: 200 cm
- victreebel: 170 cm
- vipleume: 120 cm
- voltorb: 50 cm
- vulpix: 60 cm
- wartortle: 100 cm
- weedle: 30 cm
- weepinbell: 100 cm
- weezing: 120 cm
- wigglytuff: 100 cm
- zapdos: 160 cm
- zubat: 80 cm
Tiempo de ejecución para quick sort: 0.0002081999552301407 segundos
```

Al analizar los resultados del ordenamiento por nombre, se observa un patrón de rendimiento consistente con las pruebas por altura. En este caso, Quick Sort fue aproximadamente 12.3 veces más rápido que Bubble Sort. Quick Sort completó la tarea en aproximadamente 0.00020 segundos, mientras que Bubble Sort tardó alrededor de 0.00257 segundos.

Estos resultados reafirman que la complejidad algorítmica es un factor determinante en el rendimiento, independientemente del tipo de dato específico por el cual se realice el ordenamiento (numérico o textual).

Prueba 2: 30 pokemon

```
- charmander: 60 cm
- charmeleon: 110 cm
- ekans: 200 cm
- fearow: 120 cm
- ivysaur: 100 cm
- kakuna: 60 cm
- metapod: 70 cm
- nidoran-f: 40 cm
- nidorina: 80 cm
- pidgeot: 150 cm
- pidgeotto: 110 cm
- pidgey: 30 cm
- pikachu: 40 cm
- raichu: 80 cm
- raticate: 70 cm
- rattata: 30 cm
- sandslrew: 60 cm
- sandslash: 100 cm
- spearow: 30 cm
- squirtle: 50 cm
- venusaur: 200 cm
- wartortle: 100 cm
- weedle: 30 cm
Tiempo de ejecución para quick_sort: 3.489997470751405e-05 segundos
```

```
- charmander: 60 cm
- charmeleon: 110 cm
- ekans: 200 cm
- fearow: 120 cm
- ivysaur: 100 cm
- kakuna: 60 cm
- metapod: 70 cm
- nidoran-f: 40 cm
- nidorina: 80 cm
- pidgeot: 150 cm
- pidgeotto: 110 cm
- pidgey: 30 cm
- pikachu: 40 cm
- raichu: 80 cm
- raticate: 70 cm
- rattata: 30 cm
- sandshrew: 60 cm
- sandslash: 100 cm
- spearow: 30 cm
- squirtle: 50 cm
- venusaur: 200 cm
- wartortle: 100 cm
- weedle: 30 cm
Tiempo de ejecución para bubble_sort: 8.74000252224505e-05 segundos
```

Con el mismo conjunto de 30 pokemon, se ejecutaron ambos algoritmos para ordenar la lista por el atributo name. Los tiempos registrados fueron los siguientes:

- ❖ Tiempo de ejecución para Bubble Sort: 8.74000252224505e-05 segundos (0.0000874 segundos)
- ❖ Tiempo de ejecución para Quick Sort: 3.489997470751405e-05 segundos (0.0000349 segundos)

La diferencia en los tiempos absolutos entre el ordenamiento por altura y por nombre es mínima, lo cual indica que las operaciones de normalización de cadenas de texto no añaden una carga significativa para listas tan pequeñas.

6. Conclusiones

Las pruebas realizadas sobre las funciones de búsqueda por nombre revelaron el comportamiento típico de estos algoritmos. La búsqueda binaria, por su parte, demostró una superioridad en eficiencia temporal en escenarios donde el elemento buscado se encontraba alejado del inicio de la lista, como en el caso de 'arbok', o en búsquedas con listas de mayor tamaño (100 elementos). La búsqueda lineal, por su parte, se observa notoriamente más rápida en escenarios cercanos al mejor caso, cuando el elemento buscado se ubica en las primeras posiciones de la lista. Conjuntamente, las pruebas realizadas sobre algoritmos de búsqueda por nombre validan las características teóricas de cada algoritmo, evidenciando que la elección más conveniente depende del volumen de datos a analizar y de la posición del elemento dentro del conjunto.

Las pruebas sobre los algoritmos de búsqueda utilizando la altura como criterio revelaron complejidades adicionales debido a la posibilidad de múltiples coincidencias, a diferencia de identificadores únicos como el nombre. Esta particularidad llevó a que la búsqueda lineal y la binaria retornaran diferentes resultados (ivysaur y beedrill respectivamente, para la altura '10' en una lista de 30 elementos), ilustrando cómo cada algoritmo selecciona una coincidencia según su lógica interna (la primera encontrada secuencialmente, en el caso de la búsqueda lineal, versus la primera en la lista ordenada, en la búsqueda binaria). En términos de rendimiento, se observó que la búsqueda lineal fue inicialmente más rápida para valores con coincidencias cercanas al inicio. Sin embargo, al extender la búsqueda a un valor (altura 4) cuya primera ocurrencia estaba más alejada en la lista original y en un conjunto de 100 elementos, la búsqueda binaria demostró su superioridad en eficiencia temporal, ejecutándose en la mitad del tiempo que la lineal. Estos resultados confirman que, aunque la presencia de duplicados puede variar el pokemon específico hallado, la ventaja de la búsqueda binaria en listas de mayor tamaño o cuando el elemento objetivo no está en las primeras posiciones se mantiene consistente, compensando el costo inicial del ordenamiento.

Las pruebas realizadas sobre los algoritmos de ordenamiento, Bubble Sort y Quick Sort, proporcionaron una validación importante sobre sus complejidades. Se observó consistentemente que Quick Sort superó a Bubble Sort en términos de velocidad de ejecución, tanto al ordenar por altura como por nombre, incluso con conjuntos de datos pequeños (30 pokemon). Para 30 pokemon, Quick Sort fue aproximadamente 1.75 veces más rápido al ordenar por altura y 2.5 veces más rápido al ordenar por nombre. Este comportamiento es crucial, debido a que la brecha de rendimiento entre ambos algoritmos se volvería exponencialmente más grande a medida que el volumen de datos aumentara, lo

que reafirma que Quick Sort es la opción preferible para escenarios donde la eficiencia es una prioridad, mientras que Bubble Sort podría considerarse solo en casos donde la simplicidad de implementación es el único factor crítico y el tamaño de la lista es insignificante. Esto subraya la importancia de elegir algoritmos eficientes, ya que su ventaja de rendimiento se vuelve exponencialmente mayor a medida que el volumen de datos crece.

La realización del presente trabajo contribuyó a una comprensión más profunda de los algoritmos de búsqueda lineal y binaria, y de los métodos de ordenamiento Bubble Sort y Quick Sort. Permitió, a su vez, entender las complejidades temporales de cada uno de ellos, comprobando los mejores y peores casos posibles, y percibir la influencia que tiene el tamaño de la muestra a analizar en la eficiencia de cada algoritmo.

Un obstáculo inesperado surgió al observar que las búsquedas binarias y lineales por altura, a pesar de operar correctamente, arrojaban resultados distintos para la misma consulta. Este escenario impulsó un análisis más profundo sobre el comportamiento de los algoritmos ante la presencia de valores duplicados en los datos, llevando a una comprensión más en detalle de sus estrategias de búsqueda y el análisis de sus resultados.

Como posibles mejoras, se podrían implementar las llamadas asíncronas para el método de obtención inicial de los pokemones, con el objetivo de procesar múltiples solicitudes de forma paralela, y contribuir a la reducción de los tiempos de carga. A su vez, sería conveniente guardar los datos obtenidos en un archivo local en formato JSON para los casos en que se realizarán varias pruebas con el mismo volumen de datos, de manera que se puedan acceder sin tener que reconectar con el servicio externo.

Considerando extensiones futuras, el proyecto podría beneficiarse de una utilización de estructuras de datos más avanzadas, como árboles o hash, brindando también la posibilidad de integrar funcionalidades ampliadas para los algoritmos de búsqueda y ordenamiento, con atributos adicionales como tipo, peso y habilidades.

Para concluir, el tema trabajado es sumamente útil para desarrollar la capacidad de seleccionar el algoritmo de búsqueda u ordenamiento más adecuado en diferentes escenarios, considerando esta habilidad crucial para crear aplicaciones eficientes y escalables, que optimicen el uso de recursos.

7. Bibliografía

- ❖ Python Software Foundation. (2025). Python 3 Documentation.
<https://docs.python.org/3>
- ❖ RESTful API: PokèAPI. <https://pokeapi.co/docs/v2>
- ❖ Librería requests versión 2.32.3 (2024). Apache Software License (Apache-2.0).
<https://pypi.org/project/requests>
- ❖ Librería timeit (2025). Python Software Foundation.
<https://docs.python.org/3/library/timeit.html>
- ❖ Librería pprint (2025). Python Software Foundation.
<https://docs.python.org/3/library/pprint.html>

8. Anexos

Anexo I: Enlace al video explicativo

[Enlace al video explicativo](#)

Anexo II: Código completo

[Enlace al repositorio del proyecto en Github.](#)

Anexo III: Búsquedas por nombre: comparativa de tiempos de ejecución (en segundos)

Búsquedas por nombre				
Elemento buscado	Total de elementos	Búsqueda Lineal	Búsqueda Binaria	Observaciones
"arbok"	30	0.000606s	0.000308s	La búsqueda binaria fue dos veces más rápida que la lineal.
				"arbok" representa un escenario cercano al "peor caso" para la lineal.
"dodrio"	100	0.003217s	0.000740s	La ventaja de la búsqueda binaria se acentúa con el incremento del tamaño de la lista.
"charmander"	100	0.000545s	0.001316s	La búsqueda lineal fue más rápida en su "mejor caso" (elemento cercano al inicio).

Anexo IV: Presentación con diapositivas

[Enlace a la presentación](#)