Trabajo Integrador: Programación I Algoritmos de Búsqueda y Ordenamiento

Alumnos:
Grosso Belen - belugrosso98@gmail.com COM-15
Moyano Rocío - rco.myno@gmail.com COM-18

Materia: Programación I

Profesores: Maximiliano Sar Fernandez, Julieta Trapé

Fecha de Entrega: 9 de junio de 2025

Índice

Introducción - Rocio

Un algoritmo es un set de instrucciones para completar una tarea específica de la secuencia de operaciones a realizar, como si se tratara de una receta.
En el área de la programación se utilizan los algoritmos de ordenamiento para optimizar la tarea de búsqueda y organizar los datos de manera más eficiente. En ese caso, un algoritmo de ordenamiento permite reorganizar una lista de elementos o nodos en un orden específico, lo que es esencial para optimizar el rendimiento de las aplicaciones.

Dependiendo de la lista que queramos programar, utilizaremos búsqueda lineal o binaria. Si la lista tiene elementos ordenados, podemos usar un algoritmo de búsqueda binaria, pero si la lista contiene los elementos de forma desordenada este algoritmo no nos servirá, para buscar un elemento en una lista desordenada debemos utilizar un algoritmo de búsqueda lineal.

En el presente trabajo, desarrollaremos por qué los algoritmos de ordenamiento y búsqueda son fundamentales para la manipulación y búsqueda, y cuáles nos permiten organizar los conjuntos de datos.

Marco Teórico

Búsqueda binaria es un algoritmo cuyo input es una lista ordenada de elementos. Si el elemento que se está buscando está presente en la lista, la búsqueda binaria devuelve la posición donde se encuentra. De lo contrario, devuelve null.

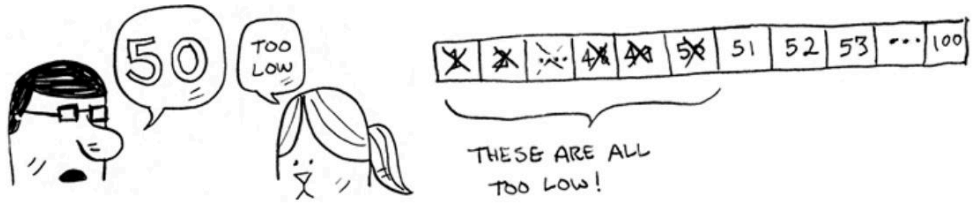
En general, para una lista de *n* elementos, a la búsqueda binaria le toma log2 *n* pasos para encontrar el elemento en el peor de los casos, donde a un a búsqueda simple, donde se busca elemento a elemento, le toma *n* pasos.

El siguiente es un ejemplo con imágenes que fue tomado del libro Grokking Algorithms (2024) de Aditya Bhargava que compara la búsqueda simple y la búsqueda binaria en el marco de un juego: adivinar un número del 1 al 100, en el menor número de intentos posibles. A cada intento, quien piensa el número dice si este es muy bajo, muy alto o correcto. Si intentáramos adivinar de la siguiente manera: 1, 2, 3, 4 … se vería de esta manera:

A bad approach to number guessing

Eso es una búsqueda simple, con cada intento se elimina sólo un número. Si el número a adivinar es el 99, podría llevar 99 intentos descubrirlo.

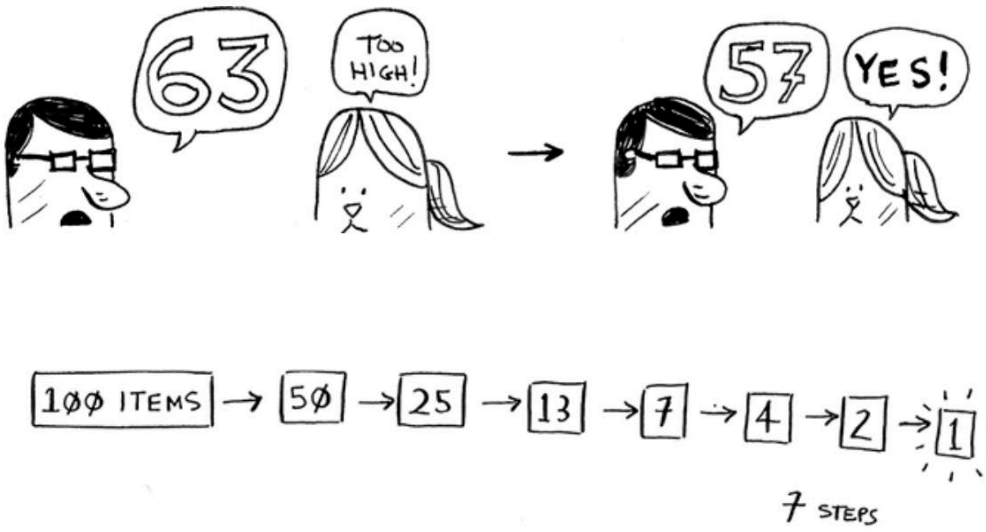
Una técnica mejor es aplicar búsqueda binaria, empezando por el número 50.



Resulta ser muy bajo, pero se eliminaron la mitad de las posibilidades. La siguiente suposición es 75.

Fue un número muy alto, pero otra vez se eliminaron la mitad de los números restantes.

Con la búsqueda binaria, en cada intento de adivinar el número correcto, se toma el número del medio y así eliminar la mitad de las opciones.



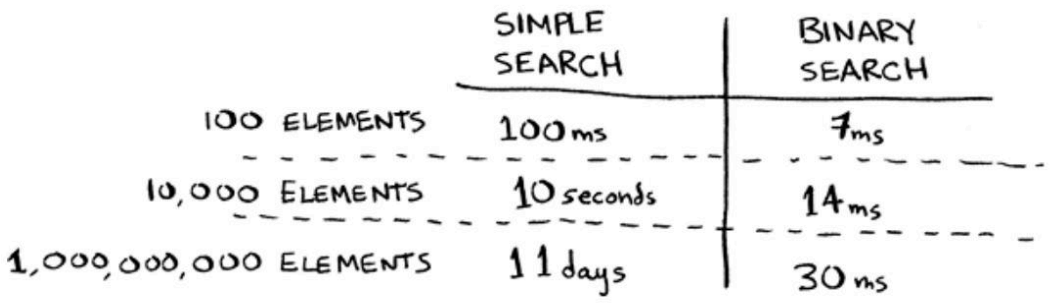
Como se observa en el ejemplo, la búsqueda binaria sólo funciona cuando la lista de elementos está ordenada.

Comparación de eficiencia: búsqueda binaria vs búsqueda lineal - Rocio

La búsqueda simple, en comparación con la binaria es más simple y no necesita que la lista a buscar esté ordenada, por lo que es más eficiente si el número de items es pequeño y si varían con el tiempo (ya que no será necesario ordenarla para cada búsqueda).
El tiempo de ejecución de una búsqueda binaria y una búsqueda simple no crece siguiendo el mismo ratio.

En la búsqueda simple, el número máximo de intentos de adivinar es el mismo que el tamaño de la lista. A esto se le llama *tiempo linear*. En cambio, la búsqueda binaria funciona en *tiempo logarítmico*.

En la siguiente tabla se puede ver cómo crece el tiempo de ejecución en las dos búsquedas.



Run times grow at very different speeds!

Cuando el número de ítems de la lista crece, a la búsqueda binaria le toma un poco más de tiempo encontrarlo, pero a la búsqueda simple le toma mucho más tiempo de ejecución llegar al resultado. Es por esto que es esencial conocer cómo aumenta el tiempo de ejecución de un algoritmo cuando el tamaño de la lista aumenta también.

Notación Big O

La notación Big O indica qué tan rápido es un algoritmo para resolver un problema. El tiempo de ejecución de un algoritmo crece a ritmos diferentes: no es suficiente conocer cuánto tiempo tarda su ejecución, es necesario saber cómo este tiempo se incrementa a medida que el tamaño de la lista crece para poder elegir el que mejor se ajuste a nuestro caso a resolver.

Esta notación indica la cantidad de operaciones que el algoritmo va a realizar. Se basa en el análisis del peor escenario posible: en la búsqueda lineal esto significa que el ítem buscado se encuentra en la última posición.

Limitaciones de la búsqueda binaria - Rocio

El algoritmo de búsqueda binaria es muy eficiente pero presenta algunas limitaciones que son importantes destacar al momento de analizar su idoneidad para resolver un problema.

Solo funciona en listas ordenadas: Si la lista no está ordenada, primero se debe ordenar. Esto puede agregar un costo de procesamiento significativo.
No es ideal para estructuras de datos dinámica: Si los datos cambian frecuentemente, ya sea insertando nuevos o eliminandolos, mantener la lista ordenada puede ser costoso.
Depende del acceso rápido a los elementos: funciona mejor en estructuras que permiten el acceso aleatorio, como arrays en memoria. Si los datos están en estructuras como listas enlazadas (linked lists), la búsqueda binaria pierde eficiencia porque no se puede acceder directamente al elemento central.
No funciona bien con elementos distribuidos: En algunos casos, los datos pueden estar almacenados en múltiples ubicaciones o hasta en diferentes sistemas, donde no es tarea fácil acceder al elemento central.

Ordenamiento

El ordenamiento es un proceso esencial en programación que consiste en organizar un conjunto de elementos (como números, letras u objetos) según un criterio determinado, ya sea de forma ascendente o descendente. A continuación expondremos los distintos tipos de ordenamiento.

Burbuja - Rocio

El ordenamiento por burbuja, también conocido como Sinking Sort, es un algoritmo simple que consiste en recorrer repetidamente la lista a ordenar, comparando pares de elementos adyacentes y realizando intercambios cuando están en el orden incorrecto.

Este proceso se repite hasta que no se necesiten más intercambios, lo que indica que la lista ya está ordenada. Dado que en cada pasada al menos un elemento alcanza su posición final, es posible reducir el rango de comparación en las siguientes iteraciones.

Su complejidad es simple pero poco efectiva para listas de mayor cantidad de elementos.

Inserción

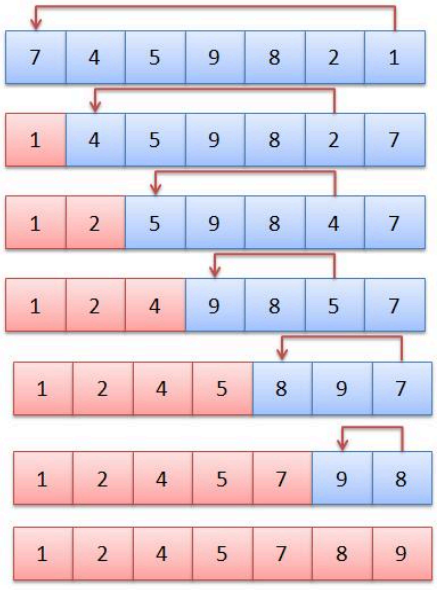
El algoritmo de ordenamiento por inserción es una técnica sencilla y fácil de implementar para ordenar una lista. Su funcionamiento se basa en recorrer la lista, seleccionando en cada iteración un elemento como clave, y comparándolo con los elementos anteriores para insertarlo en la posición correcta dentro de la porción ya ordenada.

El algoritmo comienza con una lista de elementos desordenados. Se toma el segundo elemento como clave y se lo compara con los elementos situados a su izquierda. Si es menor, se lo inserta en la posición correspondiente. Luego, se selecciona el siguiente elemento como nueva clave y se repite el mismo proceso, comparándolo con todos los elementos anteriores. Esta comparación continúa hasta encontrar un valor menor o hasta llegar al inicio de la lista. Este procedimiento se repite hasta seleccionar la última clave.

Al finalizar, todos los elementos han sido insertados en su posición correcta, y la lista queda completamente ordenada.

Selección - Rocio

Este algoritmo de ordenamiento es un algoritmo en el lugar (*in-place*) basado en comparaciones, en el que la lista se divide en dos partes: una parte ordenada al inicio (izquierda) y una parte desordenada al final (derecha). Al comenzar, la parte ordenada está vacía y la parte desordenada contiene toda la lista.



En cada iteración, se selecciona el elemento más pequeño de la parte desordenada y se intercambia con el primer elemento de esa sección, incorporándolo así a la parte ordenada. Este proceso se repite, desplazando el límite de la parte desordenada un elemento hacia la derecha en cada paso.

Quicksort

Quicksort es un algoritmo de ordenamiento que utiliza dividir y conquistar.

Dividir y conquistar es una manera de pensar un problema, es un algoritmo recursivo que usa dos pasos para resolver un problema: descubrir el caso base (el caso más sencillo) y dividir o disminuir el problema hasta que se convierte en el caso base.

Veamos este ejemplo de dividir y conquistar:

Una lista de números que hay que sumarlos entre ellos para devolver el total. El caso base sería una lista vacía o una lista con un solo elemento, donde sólo se devuelve 0 o el número.

Con cada llamada recursiva, la lista se debe acercar a un array vacío porque, según dividir y conquistar, con cada llamada recursiva se tiene que reducir el problema.

Como la recursión guarda el estado de la función en cada llamada, al llegar al caso base, las soluciones parciales se combinan para obtener el resultado final. En el caso de la suma, cada llamada devuelve la suma de su elemento actual más el resultado de la siguiente llamada, acumulando el total de manera progresiva.

La visualización gráfica sería:

```
sum_array([2, 4, 6])
    /       \
 sum([2])   sum([4, 6])
          /     \
       sum([4])  sum([6])
```

Donde en cada llamada a sum(número) se va descomponiendo la lista en partes más pequeñas.

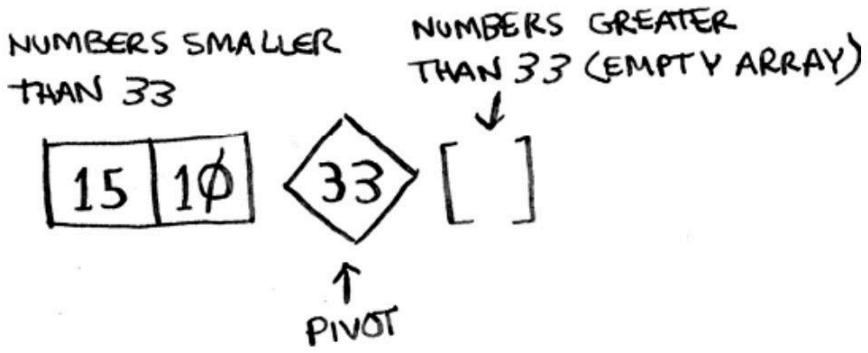
Para implementar quicksort es necesario seguir 3 pasos clave:

1. **Seleccionar un pivote**: Un elemento del array (puede ser el primero, el último, uno aleatorio o la mediana).
2. **Particionar el array**: Reorganizar los elementos para que:
    - Los *menores* que el pivote queden a su *izquierda*.
    - Los *mayores* que el pivote queden a su *derecha*.
    - El pivote queda en su posición final correcta.
3. **Repetir recursivamente**: Aplicar el mismo proceso a las subarrays izquierda y derecha del pivote.

Quicksort es ideal cuando se necesita velocidad y eficiencia en memoria, siempre que se evite el peor caso con una buena selección de pivotes.

Veamos otro ejemplo para ordenar un array usando quick sort. Ejemplo tomado del libro Grokking Algorithms (2024) de Aditya Bhargava.

Para ordenar el siguiente array [33,15,10] de menor a mayor elegimos un pivote, encontramos el elemento menor que el pivote y el elemento mayor, es decir, hacemos particiones.



Ahora tenemos un subarray de los números menores que el pivote, el pivote y un subarray de los números mayores al pivote. Los sub-arrays no están ordenados, solamente están particionados.

Si los sub-arrays estuvieran ordenados, solamente habría que combinar array izquierdo + pivote + array derecho. Para ordenarlos y llegar al resultado, el caso base ya conoce cómo ordenar arrays vacíos (sub-array derecho) y puede ordenar recursivamente arrays de 2 elementos (sub-array izquierdo). Entonces, si se llama a quicksort sobre los dos sub-arrays y se combina el resultado, se obtiene un array ordenado.

```
quicksort([15, 10]) + [33] + quicksort([])
> [10, 15, 33]                    ①
```

① A sorted array 1) Un array ordenado

Caso Práctico

Para nuestro caso práctico, se implementó la **búsqueda binaria junto con quicksort**, además incorporamos una pequeña demostración de como funcionaria en un programa orientado para usuarios y desglosaremos el código paso a paso para su entendimiento.

Las listas y diccionarios[1] (se investigó para este presente trabajo) que utilizaremos son:

```
lista_especialistas = ["Neurologo", "Pediatra", "Traumatologo", "Cardiologo", "Clinico", "Dermatologa", "Kinesiologo", "Obstetra", "Urologo", "Cirujano"]
```

[1] https://ellibrodepython.com/diccionarios-en-python

```python
doctores_por_especialidad = {
    'Cardiologo': ['Dr. Hernán Cabrera', 'Dr. Iván Romero', 'Dr. Marcelo Vargas'],
    'Cirujano': ['Dr. Leonardo Ibáñez', 'Dr. Gustavo Soto', 'Dr. Fabián Medina'],
    'Clinico': ['Dr. Ernesto Aguilar', 'Dra. Valeria Peña', 'Dr. Rodrigo Fernández'],
    'Dermatologa': ['Dra. Carolina Fuentes', 'Dra. Patricia Núñez', 'Dra. Daniela Rojas'],
    'Kinesiologo': ['Lic. Nicolás Salas', 'Lic. Andrés Molina', 'Lic. Franco Herrera'],
    'Neurologo': ['Dr. Ricardo Paredes', 'Dr. Sergio Quiroga', 'Dr. Pablo Leiva'],
    'Obstetra': ['Dra. Ana Martínez', 'Dra. Florencia Díaz', 'Dra. Elena Suárez'],
    'Pediatra': ['Dra. Laura Ramírez', 'Dra. Julieta Castro', 'Dra. Cecilia Gómez'],
    'Traumatologo': ['Dr. Diego Sánchez', 'Dr. Tomás Vera', 'Dr. Martín López'],
    'Urologo': ['Dr. Carlos Méndez', 'Dr. Javier Ruiz', 'Dr. Alberto Torres']
}
```

```python
especialistas_codigos = {
    "Neurologo": 4,
    "Pediatra": 8,
    "Traumatologo": 9,
    "Cardiologo": 10,
    "Clinico": 7,
    "Dermatologa": 5,
    "Kinesiologo": 6,
    "Obstetra": 1,
    "Urologo": 3,
    "Cirujano": 2
}
```

En primera instancia, se crea la función quicksort, donde podemos ver un bucle if empezando por condicionar si es que hay 1 o menos elementos en la lista, lo cual retorna “lista_especialistas”.

El pivote se usa como referencia, ya que la lista debe ser dividida en dos.

```python
def quicksort(lista_especialistas):
    if len(lista_especialistas) <= 1:
        return lista_especialistas
    pivote = lista_especialistas[0] #
```

Este bloque corresponde a la parte recursiva del algoritmo quicksort, que ordena listas dividiéndolas en elementos menores o iguales y mayores a un "pivote". Se crea una lista con todos los elementos menores o iguales al pivote donde “lista_especialistas[1:]” toma todos los elementos excepto el primero, porque ese primero es el pivote.

Se usa una lista por comprensión para filtrar y finalmente se llama recursivamente a quicksort con las listas menor y mayor uniendo los resultados en el orden: menores + pivote + mayores.

```python
menor = [x for x in lista_especialistas[1:] if x <= pivote]
mayor = [x for x in lista_especialistas[1:] if x > pivote]
return quicksort(menor) + [pivote] + quicksort(mayor) # ret
```

Definimos la búsqueda binaria, utilizando como parámetros la lista y el objetivo, es decir lo que queremos encontrar dentro de la lista.

Izquierda y derecha representan los índices que marcan los extremos del segmento de la lista que se está examinando. Al principio, se empieza con todo el rango:

```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
```

desde el primer (0) al último elemento (lista) - 1).

Con el bucle while se ejecuta mientras haya elementos por revisar. Si la izquierda supera a la derecha, significa que el objetivo no está en la lista.

El elemento medio calcula el índice medio del segmento actual de la lista con una división (//) entera.

Con el bucle if, si el elemento en la posición medio es el objetivo, lo encontró y devuelve su índice; si el valor medio es menor que el objetivo, el valor debe estar a la derecha y se actualiza izquierda; si el valor medio es mayor que el objetivo, el valor debe estar a la izquierda y se actualiza derecha. Por último, si el bucle termina sin encontrar el objetivo, se devuelve -1, que indica que el valor no está en la lista.

Finalmente utilizamos las funciones anteriormente definidas y unificamos el código para el programa.

En primera instancia solicitamos al usuario su nombre y correo electrónico que serán guardados en la variables “nombre” y “mail” respectivamente. Como vemos en la captura, se despliega una breve explicación de los pasos a seguir que ayudarán al usuario a entender el programa.

Cada número equivale a la especialidad médica y dicha disponibilidad estará dictaminada por el ordenamiento de números de la lista “especialistas_medicos”.

```
nombre = (input("¡Bienvenido/a! Por favor ingrese su nombre y apellido: "))
mail = (input("Por favor ingrese su mail: "))
print("------------------------------")

print(f"Estimado/a paciente {nombre}, a continuación se mostrarán los especialistas médicos disponibles hasta la fecha en orden descendente.")
print("Cada numero quivale a la especializacion disponible en nuestro sistema. Por ejemplo, el número **4** equivaldrá a la cuarta especialidad en la lista de especialistas médicos")
print("Mayor a 5: equivale a las especialidades con menor disponibilidad hasta el momento.")
print()
```

Luego se despliega por consola el resultado de las listas previamente ordenadas con quicksort y búsqueda binaria:

```
print("===== Disponibilidad =====")

print(quicksort(numeros)) # traemos la lista de

print()
print("===== Especialistas Médicos =====")
print(quicksort(lista_especialistas_ordenada)) #
```

El código anterior, es asistido por la variable “numeros”, por cada x en la lista “lista_especialistas_ordenada” toma el valor correspondiente a x dentro del diccionario especialistas_codigos y lo agrega a la lista de números.

```
numeros = [especialistas_codigos[x] for x in lista_especialistas_ordenada]
```

Resultado:

```
===== Disponibilidad =====
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

===== Especialistas Médicos =====
['Cardiologo', 'Cirujano', 'Clinico', 'Dermatologa', 'Kinesiologo', 'Neurologo', 'Obstetra', 'Pediatra', 'Traumatologo', 'Urologo']
```

Agregamos la variable “búsqueda” donde el usuario ingresa la especialidad elegida. El .capitalize() convierte la primera letra en mayúscula y el resto en minúscula, para ayudar a igualar con el contenido de la lista. Esto ayudará a que el usuario pueda ingresar con más libertad las palabras y no resulte en un error.

La variable “indice” usa la función de búsqueda binaria para encontrar el índice, donde luego con el bucle if verificará si fue encontrada y está en el diccionario de doctores.

```
print()
busqueda = input("Ingrese la especialidad que desea buscar (ej. Pediatra): ").capitalize()

# Realizar la búsqueda
indice = busqueda_binaria(lista_especialistas_ordenada, busqueda)
```

Por último, utilizamos el bucle if para mostrar la posición de la especialidad elegida y una condición si esta es mayor a 5, mostrará un mensaje para el usuario. Con el bucle for muestra los doctores disponibles para esa especialidad y no encuentra otorga un mensaje de advertencia al usuario.

```
if indice != -1 and busqueda in doctores_por_especialidad:
    print(f"\nEspecialidad encontrada en la posición {indice}.")

# Mensaje aclarando que si la posición es 5 o mayor saldra un me
    if indice >= 5:
        print("Atención: Esta especialidad presenta menor dispon

    print("\nDoctores disponibles:")
    for doctor in doctores_por_especialidad[busqueda]:
        print(f"- {doctor}")
else:
# Si no coindice con el contenido de la lista
    print("Especialidad no encontrada.")
```

Resultado:

```
Ingrese la especialidad que desea buscar (ej. Pediatra): pediatra

Especialidad encontrada en la posición 7.
```

Metodología Utilizada - Rocio

La elaboración del trabajo se realizó en las siguientes etapas:

- Recolección de información teórica en documentación confiable.
- Implementación en Python de los algoritmos estudiados.

- Redacción de código aplicando la teoría.
- Ejemplo práctico para el entendimiento de los conceptos.

Resultados Obtenidos - Rocio

- El programa ordenó correctamente la lista de números ingresada.
- La búsqueda binaria localizó de forma eficiente el número especificado.
- Se comprendieron las diferencias en complejidad entre los algoritmos estudiados.
- Se valoró la importancia de tener los datos ordenados para aplicar búsqueda binaria.

Conclusiones - Rocio

El estudio de los algoritmos de búsqueda y ordenamiento es fundamental en programación, ya que constituyen la base para el manejo eficiente de datos. Comprender cómo y cuándo aplicar métodos como búsqueda lineal, binaria, así como ordenamientos como burbuja, inserción, selección, quicksort o mergesort, permite optimizar el rendimiento de los programas, especialmente cuando se trabaja con grandes volúmenes de información.

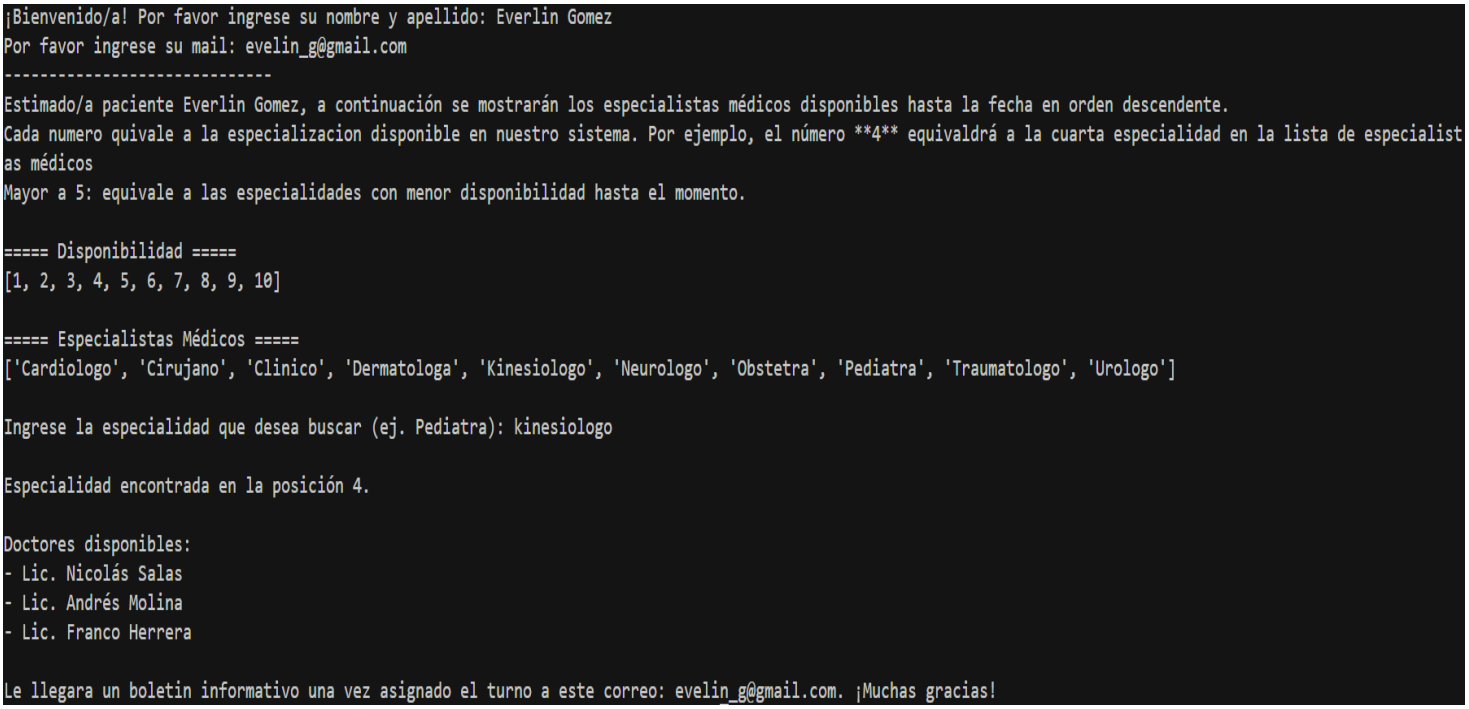
Más allá de su aplicación técnica, estos algoritmos fomentan el pensamiento lógico, la resolución de problemas y la toma de decisiones basada en la eficiencia computacional. Dominar estas herramientas no solo mejora nuestras habilidades como programadores, sino que también sienta las bases para enfrentar desafíos más complejos en estructuras de datos y algoritmos avanzados.

Bibliografía

- https://ellibrodepython.com/diccionarios-en-python
- Aditya Bhargava, Grokkin Algotithms, second edition (2024). Consultado en linea: https://learning.oreilly.com/library/view/grokking-algorithms-second
- https://lwh-21.github.io/Algos/algoritmos%20de%20ordenaci%C3%B3n/bubblesort_es.html#
- https://juncotic.com/ordenamiento-por-insercion-algoritmos-de-ordenamiento/
- https://medium.com/@teamtechsis/selection-sort-algorithm-74dfcc7b4c87
- Apunte Google Colab “BusquedaOrdenamiento” - TUPaD

Anexos

- Captura de pantalla del programa funcionando.



- Repositorio en GitHub: https://github.com/Belu1498/Trabajo_Integrador_Programacion_I_TUPaD/tree/main
- Video explicativo: https://youtu.be/ml4aovoTRw8
- Presentacion Power Point: https://gamma.app/docs/Algoritmos-de-Busqueda-y-Ordenamiento-m0b2e4zk3ys9frs?mode=doc