UNIVERSIDAD TECNOLOGICA NACIONAL

Programación I

Trabajo Integrador

Algoritmos de Búsqueda y Ordenamiento en Python.

Alumnos

Enderson Eduardo Suarez Porras (endrsn.srz.grs@gmail.com)

Damián Eduardo Tristant (st_demian@hotmail.com)

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación I

Docentes

Nicolás Quirós

Matias Santiago Torres

20 de Junio de 2025

Índice

1. Introducción.

Este trabajo se centra en la aplicación de algoritmos de búsqueda y ordenamiento utilizando el lenguaje de programación Python. Se eligió el tema debido a su relevancia dentro del desarrollo de software, ya que estos algoritmos son fundamentales para la manipulación eficiente de datos.

A través del desarrollo de una agenda de contactos, se busca ilustrar de forma práctica cómo se pueden implementar distintas técnicas para buscar y organizar información.

El objetivo principal es comparar el funcionamiento de distintos algoritmos, comprender sus ventajas y limitaciones, y aplicarlos en un problema cotidiano y sencillo como la gestión de contactos en dicha agenda.

2. Marco teórico

Procesar eficientemente los datos es uno de los pilares fundamentales en el desarrollo de software. Entre las operaciones más comunes en la gestión de datos se encuentran la búsqueda y el ordenamiento, dos procesos que permiten localizar elementos dentro dentro de una estructura y organizarlos según ciertos criterios.

La búsqueda es el proceso mediante el cual se intenta encontrar un valor específico dentro de una estructura de datos. En Python, este proceso puede realizarse de distintas formas, entre ellas las más comunes son:

Búsqueda lineal (o secuencial): Recorre los elementos uno a uno hasta encontrar el buscado o hasta agotar todos los elementos. Este método no requiere que los datos se encuentren arreglados

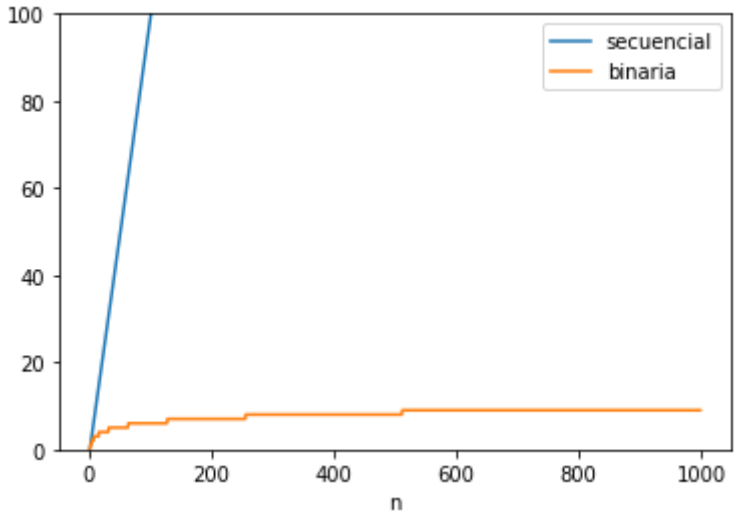
Es fácil de programar pero frente a las estructuras de datos grandes puede ser lento, lo que la hace un método ideal para fuentes de datos pequeños.

Su complejidad es O(n), lo que significa que el tiempo de ejecución crece proporcionalmente con el tamaño de la lista. Esto explica porqué ante el crecimiento de, por ejemplo, una lista de contactos, que pasa de tener 5 contactos a tener 100.000, el tiempo que tarda el algoritmo crece en paralelo.

Búsqueda binaria: Es un algoritmo más eficiente, con complejidad O(log n), pero solo funciona sobre estructuras fueron sometidas a un proceso de ordenamiento.

Consiste en dividir el rango de búsqueda, a la mitad en cada paso, comparando el elemento central con el valor buscado y descartando la mitad que no puede contenerlo. Esto hace que sea más rápido este método.

En el siguiente gráfico, se observa, la línea azul correspondiente a la búsqueda lineal, que crece con el tamaño de la lista, mientras que la búsqueda binaria correspondiente a la línea verde, lo hace de manera logarítmica O(log n).



El ordenamiento, por su parte, consiste en organizar los elementos de nuestro conjunto según criterios funcionales para nuestros programas.

El principal beneficio de ordenar nuestros datos, es la eficiencia, ya que al contar con un conjunto organizado se pueden realizar búsquedas de manera más rápida y sencilla, lo que se hace más conveniente cuanto más grandes son los conjuntos.

También se facilita el análisis de datos, lo que permite encontrar patrones o tendencias, a la vez que el uso de herramientas de visualización de datos se potencia.

Al momento de ordenar, hay varios algoritmos de los que nos podemos servir, mencionamos algunos de ellos:

Ordenamiento por burbuja (Bubble Sort): Compara pares de elementos adyacentes y los intercambia si están desordenados. Sencilla pero muy lenta.

Ordenamiento rápido (Quick Sort): Toma la lista original y la divide en dos para después ordenarlos en forma recursiva, una vez ordenados los combina para tener la lista original pero ordenada. Ideal para grandes listas de datos, es mucho más rápido que el Bubble Sort en la mayoría de los casos.

Insertion Sort: Construye una lista ordenada insertando cada elemento en su posición correcta. Ideal para listas pequeñas y casi ordenadas.

De esta manera, la búsqueda y el ordenamiento nos abren posibilidades cuyos beneficios conviene aprovechar para nuestros conjuntos de datos.

La eficiencia, la organización y escalabilidad, por mencionar algunos de estos beneficios, no son características que aparezcan de manera casual y azarosa en nuestro código y conjuntos, sino que debemos hacer uso de las buenas prácticas de programación y de las herramientas disponibles para dotar a nuestro codigo de ellas, y es lo que procederemos a desarrollar en nuestro caso práctico.

3. Caso práctico y metodología.

Se llevó a cabo el desarrollo una agenda de contactos simulada utilizando listas de diccionarios. Cada contacto contiene tres campos: nombre, teléfono y email.

a) Comenzando por la carga de datos, nuestra agenda se construyó como una lista de diez contactos ficticios. Cada contacto es un diccionario, y la lista entera representa nuestra base de datos sobre la cual se aplican los algoritmos.

```
agenda = [
    {"nombre": "Luis", "telefono": "123", "email": "luis@mail.com"},
    {"nombre": "Ana", "telefono": "456", "email": "ana@mail.com"},
    {"nombre": "Carlos", "telefono": "789", "email": "carlos@mail.com"},
    {"nombre": "Bruno", "telefono": "321", "email": "bruno@mail.com"},
    {"nombre": "Sofía", "telefono": "654", "email": "sofia@mail.com"},
    {"nombre": "Valentina", "telefono": "987", "email": "valentina@mail.com"},
    {"nombre": "Diego", "telefono": "159", "email": "diego@mail.com"},
    {"nombre": "Martina", "telefono": "753", "email": "martina@mail.com"},
    {"nombre": "Pedro", "telefono": "852", "email": "pedro@mail.com"},
    {"nombre": "Elena", "telefono": "951", "email": "elena@mail.com"}
]
```

Esta estructura es ideal para probar búsquedas lineales y binarias, ya que nos permite recorrer los elementos y acceder a sus datos de forma clara.

b) Luego pasamos a crear una función que nombramos obtener_nombre( ) que devuelve el valor de la clave “nombre” de cada contacto. Esta función es la que luego vamos a utilizar como criterio para ordenar la agenda alfabéticamente.

```
17  def obtener_nombre(contacto):
18      return contacto["nombre"]
19
```

c) Antes de realizar las búsquedas, aplicamos un ordenamiento con el método sort () de listas, utilizando como clave la función mencionada (obtener_nombre). Esto es indispensable para que la búsqueda binaria funcione correctamente, aunque no es determinante para la búsqueda lineal.

```
31  agenda.sort(key=obtener_nombre)
```

d) Una vez ordenada, pasamos a imprimir la agenda completa por pantalla, con nombre, teléfono y correo electrónico.

```
35 for contacto in agenda:
36     print(f"Nombre: {contacto['nombre']}, Teléfono: {contacto['telefono']}, Email: {contacto['email']}")
```

e) Luego, para la función busqueda_lineal () la cual recorre cada elemento de la lista comparando el nombre buscado con el elemento nombre en cada contacto, usamos la funcionalidad .lower() en ambas cadenas, para evitar ningún tipo de problemas con mayúsculas y minúsculas.

Lo que hace esta lista es devolver el primer contacto que coincida con el nombre de búsqueda.

```python
# Búsqueda lineal
def busqueda_lineal(agenda, nombre):
    #Recorre con ciclo for.
    #Por cada contacto en la agenda, si coincide con el nombre pasado por parametro retorna el contacto
    for contacto in agenda:
        #Para simplicar la prueba utilizamos lower para que no distinga entre mayusculas o minusculas
        if contacto["nombre"].lower() == nombre.lower():
            return contacto
    return None
```

f) Búsqueda binaria: después de la búsqueda lineal, pasamos a implementar el algoritmo de búsqueda binaria. En este caso se define un rango de búsqueda delimitado por los extremos de la lista (izquierda y derecha) para calcular repetidamente un punto medio. Según el valor en ese punto medio, se descarta una mitad del rango. Este proceso se repite hasta encontrar el nombre o agotar las posibilidades.

```python
def buscar_binaria(agenda, nombre):

    #Declaro "Las puntas del indice como izquierda y derecha"
    izquierda = 0
    derecha = len(agenda) - 1

    #Mientras izquierda sea menor que derecha
    while izquierda <= derecha:
        #calcula el medio y guarda en variale medio
        medio = (izquierda + derecha) // 2
        #variable nombre_medio es igual a la agenda [con el medio calculado arriba] y el nombre que este en ese indi
        nombre_medio = agenda[medio]["nombre"]

        #Si el nombre en ese indice medio es igual al nombre traido como parametro devuelve el mismo valor.
        if nombre_medio.lower() == nombre.lower():
            return agenda[medio]
        #Si el nombre como parametro es menor al nombre medio. Derecha va a ser igual a medio - 1
        elif nombre.lower() < nombre_medio.lower():
            derecha = medio - 1
            #Sino izquierda va a ser igual a medio + 1
        else:
            izquierda = medio + 1

            #En resumen divide la lista en 2 y busca en cada mitad por separado comparando con un elemento central.
            #Si en menor se descarta la mitad izquierda y si es mayor la mitad derecha.
```

g) Mediciones de tiempos: teniendo nuestra lista ordenada y definidas las funciones para la búsqueda lineal y para la búsqueda binaria, pasamos a comparar el rendimiento de ambos métodos, medimos el tiempo de ejecución importante

con time.time ( ). Se registran los tiempos antes y después de cada búsqueda, y se calcula la diferencia.

```
67  #Medicion tiempo busqueda lineal
68  import time
69  inicio_lineal = time.time()
70  resultado_lineal = busqueda_lineal(agenda,"Carlos")
71  fin_lineal = time.time()
72  tiempo_lineal = fin_lineal - inicio_lineal
73
74  #Medicion tiempo busqueda binaria
75  inicio_binaria = time.time()
76  resultado_binaria = buscar_binaria(agenda,"Carlos")
77  fin_binaria = time.time()
78  tiempo_binaria = fin_binaria - inicio_binaria
```

h) Como paso final, imprimimos los datos del contacto encontrado en cada tipo de búsqueda, junto con el tiempo que tardó en encontrarse. Lo que nos permite observar las diferencias de rendimiento entre ambos métodos, y aunque el tamaño de nuestra lista tan pequeña no permite notar la diferencia, contamos con el sustento teórico y el desarrollo del código para justificar su implementación en las pruebas.

```
80  # Prueba búsqueda lineal y resultado por pantalla
81  resultado_lineal = busqueda_lineal(agenda, "Carlos")
82  print("Resultado búsqueda lineal:")
83  print(f"Nombre: {resultado_lineal['nombre']}, Teléfono: {resultado_lineal['telefono']}, Email: {resultado_l
84  print(f"Tiempo de busqueda: {tiempo_lineal:.6f} segundos")
85
86  print("")
87  # Prueba búsqueda binaria y resultado por pantalla
88  resultado_binaria = buscar_binaria(agenda, "Carlos")
89  print("Resultado búsqueda binaria:")
90  print(f"Nombre: {resultado_binaria['nombre']}, Teléfono: {resultado_binaria['telefono']}, Email: {resultado
91  print(f"Tiempo de busqueda: {tiempo_binaria:.6f} segundos")
```

4. Resultados obtenidos.

La ejecución del programa mostró la agenda ordenada alfabéticamente por nombre, confirmando el correcto funcionamiento de la función sort() con clave personalizada.

Ambas funciones de búsqueda, tanto lineal como binaria, lograron encontrar el contacto buscado (Carlos) correctamente. En ambos casos, se imprimió por pantalla el nombre, teléfono y correo electrónico del contacto encontrado, junto con el tiempo que tardó la búsqueda en ejecutarse.

Dado que se trata de una agenda pequeña, de tan solo 10 elementos, los tiempos de ejecución registrados para ambas búsquedas fueron extremadamente bajos y similares,lo que no permitió observar una diferencia entre uno y otro método. Esto era esperable, ya que las diferencias de rendimiento entre algoritmos se vuelven significativas cuando el tamaño de los datos es de diferencia mucho mayor.

En nuestro caso, sin embargo, la estructura del programa permitió evidenciar cómo cada algoritmo funciona internamente, como realizar su implementación y preparar el terreno para ampliar el experimento a listas de mayor tamaño en el futuro, incluso dentro de este código que consideramos escalable.

5. Conclusiones.

A través de este ejercicio pudimos implementar y comparar dos estrategias de búsqueda fundamentales: la lineal y la binaria, así como aplicar una técnica de ordenamiento previa a través del método sort(). El trabajo que antecede estos comentarios nos permitió entender cómo el ordenamiento puede mejorar la eficiencia de ciertos algoritmos.

Si bien con estructuras pequeñas las diferencias no son evidentes, comprender estas técnicas fue la finalidad de este trabajo, entendiéndose como herramientas esenciales para enfrentar problemas más complejos y trabajar con grandes volúmenes de datos.

También se hizo evidente la importancia de utilizar estructuras claras y funciones auxiliares que faciliten el mantenimiento y la

lectura del código. El trabajo nos sirvió no solo para ejercitar conceptos de búsqueda y ordenamiento, sino también para fortalecer la lógica algorítmica y modular en la programación con Python, en el marco de la materia Programación I, esperamos haber cumplido con lo esperado.