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.

Además, el proyecto incorpora el uso de módulos para organizar el código de manera más clara y estructurada, separando los datos, la lógica de búsqueda y la ejecución principal en archivos independientes. Esta modularización no solo mejora la legibilidad del programa, sino que también facilita su mantenimiento, reutilización y escalabilidad.

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

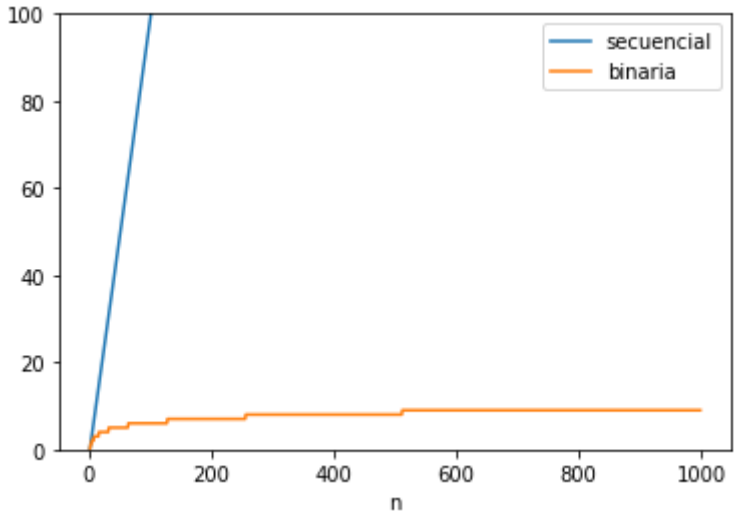
Es fácil de programar pero frente a las estructuras de datos grandes puede ser lento, lo que la hace un método ideal para fuentes de datos pequeños.

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.

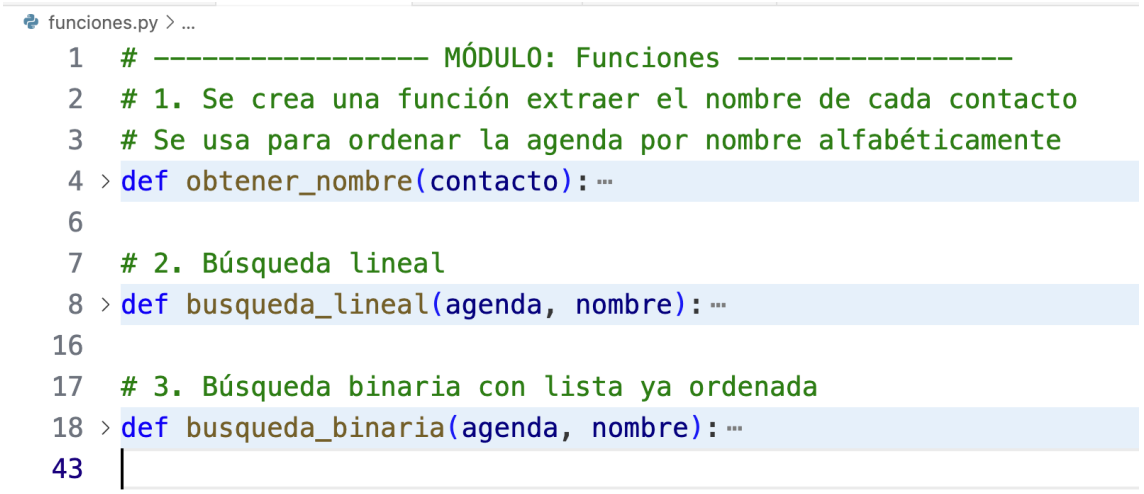
Módulo datos: datos.py

a) Comenzando por la carga de datos, nuestra agenda se construyó como una lista de 296 contactos ficticios con nombres únicos. Cada contacto es un diccionario, y la lista entera representa nuestra base de datos sobre la cual se aplican los algoritmos.

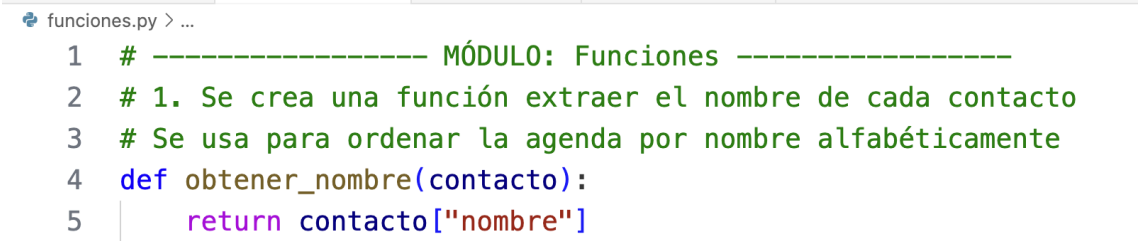
El uso de la estructura de diccionario es ideal para probar búsquedas lineales y binarias, ya que nos permite recorrer los elementos y acceder a sus datos de forma clara.

Módulo funciones: funciones.py

En este módulo hemos definido 3 funciones, obtener nombre, busqueda_lineal y busqueda_binaria.



b) Dentro de este módulo 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.



c) Luego, se definió la función busqueda_lineal () la cual recorre cada elemento de la lista comparando el nombre buscado con el elemento nombre en cada contacto y 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.

```
# 2. 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
```

d) 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.

funciones.py > ...

```python
# 3. Búsqueda binaria con lista ya ordenada
def busqueda_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
        nombre_medio = agenda[medio]["nombre"]

        #Si el nombre en ese indice medio es igual al nombre traido como parametro devuelve el mismo val
        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 elemen
            #Si en menor se descarta la mitad izquierda y si es mayor la mitad derecha.

    return None
```

Módulo de ejecución: TpIntegrador.py

Es el archivo principal. En él se ordena la agenda, se ejecutan las búsquedas y se miden los tiempos de ejecución. Por último se muestran los resultados por pantalla.

e) En este archivo .py empezamos haciendo los llamados al diccionario y a las funciones que hemos definido anteriormente en los módulos Agenda y Funciones, respectivamente.

   Para esto se usa la estructura from …. import … en ambos casos

TpIntegrador.py > main

```python
# ----------------- MÓDULO: Agenda -----------------
from datos import agenda
# ----------------- MÓDULO: Funciones -----------------
from funciones import obtener_nombre, busqueda_lineal, busqueda_binaria

# ----------------- MÓDULO: Aplicación principal -----------------
```

Y luego pasamos a las operaciones propias de este archivo.

f) Le expresamos a python que queremos usar la librería time.

```python
# ----------------- MÓDULO: Aplicación principal -----------------
def main():
    # Importamos la libreria time
    import time
```

g) 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

```python

    # Ordena la lista por nombre usando "sort" y la funcion obtener_nombre
    agenda.sort(key=obtener_nombre)

```

es determinante para la búsqueda lineal.

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

```python

    # Imprimo la agenda completa ordenada alfabeticamente
    print("Agenda ordenada:\n")
    for contacto in agenda:
        print(f"Nombre: {contacto['nombre']}, Teléfono: {contacto['telefono']}, Email: {contacto['email'
```

i) 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.

```
TpIntegrador.py > main
 7  def main():
19      #Medicion tiempo busqueda lineal
20      inicio_lineal = time.time()
21      resultado_lineal = busqueda_lineal(agenda,"Carlos")
22      fin_lineal = time.time()        (variable) inicio_lineal: float
23      tiempo_lineal = fin_lineal - inicio_lineal
24
25      #Medicion tiempo busqueda binaria
26      inicio_binaria = time.time()
27      resultado_binaria = busqueda_binaria(agenda,"Carlos")
28      fin_binaria = time.time()
29      tiempo_binaria = fin_binaria - inicio_binaria
```

j) 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 de búsqueda.

```
30
31      # Prueba búsqueda lineal y resultado por pantalla
32      print("Resultado búsqueda lineal:")
33      print(f"Nombre: {resultado_lineal['nombre']}, Teléfono: {resultado_lineal['telefono']}, Email: {resultado_l
34      print(f"Tiempo de busqueda: {tiempo_lineal:.6f} segundos")
35
36      print("")
37      # Prueba búsqueda binaria y resultado por pantalla
38      print("Resultado búsqueda binaria:")
39      print(f"Nombre: {resultado_binaria['nombre']}, Teléfono: {resultado_binaria['telefono']}, Email: {resultado_
40      print(f"Tiempo de busqueda: {tiempo_binaria:.6f} segundos")
41
42  if __name__ == "__main__":
43      main()
```

4. Resultados obtenidos.

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

```
(3.11.9) enderson@Escritorio-Enderson Trabajo_Integrador_Programaci-n % /Users/enderson/.pyenv/versions/3.11.9/bin/python /Users/enders
bajo_Integrador_Programaci-n/TpIntegrador.py
Agenda ordenada:

Nombre: Abel, Teléfono: 709, Email: abel@mail.com
Nombre: Abril, Teléfono: 807, Email: abril@correo.com
Nombre: Adela, Teléfono: 481, Email: adela@mail.com
Nombre: Adriana, Teléfono: 344, Email: adriana@example.com
Nombre: Adrián, Teléfono: 140, Email: adrián@email.com
Nombre: Adán, Teléfono: 297, Email: adán@email.com
Nombre: Agustina, Teléfono: 305, Email: agustina@example.com
Nombre: Agustín, Teléfono: 522, Email: agustín@correo.com
Nombre: Ahri, Teléfono: 639, Email: ahri@mail.com
Nombre: Aida, Teléfono: 177, Email: aida@mail.com
Nombre: Ailén, Teléfono: 701, Email: ailén@correo.com
Nombre: Alain, Teléfono: 608, Email: alain@mail.com
Nombre: Alan, Teléfono: 905, Email: alan@example.com
Nombre: Alba, Teléfono: 581, Email: alba@correo.com
Nombre: Alejandra, Teléfono: 335, Email: alejandra@example.com
Nombre: Alejandro, Teléfono: 364, Email: alejandro@correo.com
Nombre: Alexa, Teléfono: 748, Email: alexa@example.com
Nombre: Alfonso, Teléfono: 843, Email: alfonso@email.com
```

Así pudimos observar la eficacia de este método para ordenar datos, otorgandonos una lista que anteriormente se encontraba desordenada en una sobre la cual ahora se puede realizar una búsqueda binaria.

b) 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.

```
Resultado búsqueda lineal:
Nombre: Carlos, Teléfono: 513, Email: carlos@correo.com
Tiempo de busqueda: 0.000010 segundos

Resultado búsqueda binaria:
Nombre: Carlos, Teléfono: 513, Email: carlos@correo.com
Tiempo de busqueda: 0.000004 segundos
```

Los tiempos de ejecución registrados para ambas búsquedas aunque fueron bajos, permitieron observar una diferencia entre uno y otro método.

Esto era esperable, y dado que las diferencias de rendimiento entre estos dos algoritmos de búsqueda se vuelven más significativas a medida que el tamaño de los datos es muchos mayor, dimos por alcanzado el objetivo al obtener un tiempo de búsqueda binaria de menos de la mitad del tiempo de la búsqueda lineal. 0.000004 segundos vs 0.000010 segundos

Con la modularización de nuestro código hemos conseguido, separando los datos de las funciones y la ejecución, mejorar la legibilidad del programa y facilitar su reutilización, para lo cual incluso hemos incluido la función main en el archivo TpIntegrador.py pudiendo reutilizar tambien esta función con sus resultados, para expandir el código a futuro.

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. Comprender estas técnicas de búsqueda era la finalidad de este trabajo, y las entendimos como herramientas esenciales para enfrentar problemas más complejos y trabajar con grandes volúmenes de datos.

Pudimos observar incluso con un diccionario agenda de tan solo 296 contactos, diferencias de el triple del tiempo entre ellas, siendo la binaria el método más rápido. En testeos con listas más pequeñas las diferencias eran tan similares como irrelevantes, pero pasado el umbral de determinado volumen de datos las diferencias se irá haciendo mayor debido al funcionamiento diferencial de los métodos. Lo que permitió constatar la supremacía en la eficacia de la búsqueda binaria sobre la búsqueda lineal.

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.

5. Bibliografía.

1. Notebook Búsqueda y Ordenamiento. Material de la cátedra. Programación I. Disponible en: BusquedaOrdenamiento.ipynb

2. Difference Between Binary Search and Linear Search Algorithm Disponible en: https://www.wscubetech.com/resources/dsa/linear-vs-binary-search

2. Sorting Techniques Disponible en: https://docs.python.org/es/3.13/howto/sorting.html