Datos Generales

- **Título del trabajo:** Búsqueda y ordenamiento
- **Alumnos:** Yanina Martin - yaninamicamartin@gmail.com
  Martín Ezequiel Toledo - metoledo94@gmail.com
- **Materia:** Programación I
- **Profesor/a:** Julieta Trapé
- **Fecha de Entrega:** 09/06/2025

Índice

---

1. Introducción

- ¿Por qué se eligió este tema?

Fue elegido este tema por su importancia en el ámbito de la programación.

Este tipo de algoritmos son fundamentales para el manejo de los datos y para poder manejar gran cantidad de volúmenes de información.

• ¿Qué importancia tiene en la programación?

Los algoritmos de búsqueda y ordenamiento son fundamentales en la programación. No podemos procesar o modificar información si no podemos encontrarla. Se usan a diario en casi todos los productos informáticos que consumimos (listado de productos en un e-commerce con filtros, barras de búsqueda en páginas web, contactos en nuestro celular, entre muchos otros ejemplos). También se implementan en varios aspectos de la programación que están por detrás de la interfaz del usuario como en los métodos que nos ofrecen los distintos lenguajes de programación para ordenar o buscar en listas, analizar datos, optimizar almacenamiento en bases de datos y recuperar valores de las mismas.

• ¿Qué objetivos se propone alcanzar con el desarrollo del trabajo?

Muchas veces desconocemos el funcionamiento de métodos de alto nivel que utilizamos a la hora de programar. La conveniencia de este tipo de programación puede dejar en segundo plano el entendimiento de que tipo de algoritmos estamos utilizando, y si son los más adecuados según el caso de uso en particular. Este trabajo busca realizar una comparación entre algoritmos de búsqueda lineal y binaria, examinar casos de uso práctico, cómo son implementados en la librería estándar de Python y cuando es beneficioso usar uno u otro.

---

2. Marco Teórico

En programación, la búsqueda y la ordenación son técnicas esenciales para organizar y manipular datos.

La búsqueda consiste en encontrar un elemento puntual en una lista de datos, y la ordenación busca organizar los elementos de un conjunto de datos, en un orden específico. Existen varios algoritmos de búsqueda y ordenación, cada uno con sus propias ventajas y desventajas.

Los más comunes en búsqueda son: Los buscadores lineales y los buscadores binarios.

Búsqueda Lineal

La búsqueda lineal es un algoritmo que compara cada uno de los elementos presentes en la lista hasta encontrar el valor deseado. Es uno de los más sencillos entre los algoritmos de búsqueda, lo cual puede ser una ventaja y una desventaja a la vez (Wikipedia, s.f.; DataCamp, s.f.). Se recomienda para datasets pequeños, desordenados, cuando tenemos estructuras de datos disímiles o muy complejas o cuando la eficiencia no es importante. Deja de ser útil cuando la complejidad temporal del algoritmo se vuelve un problema, ya que en el peor caso puede necesitar recorrer absolutamente todos los elementos de la lista (BBC Bitesize, s.f.). Python utiliza este tipo de búsqueda en algunos operadores de la librería estándar, por ejemplo:

El operador **in** realiza una búsqueda lineal en la lista o tupla hasta encontrar el valor X, de izquierda a derecha devolviendo un booleano (Codecademy, s.f.).

```
x = 5
cinco_presente = x in [1, 2, 3, 4, 5, 6, 7]
```

El método **index()** que pertenece al objeto de lista devuelve el índice de la primera ocurrencia del valor deseado, ejecutando una búsqueda lineal (Python Software Foundation, s.f.).

```
[10, 20, 30, 40].index(30)
```

Búsqueda binaria:

Según Khan Academy (s.f.), la búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos. Funciona al dividir repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una (“Binary Search”, Khan Academy, s.f.).

Por ejemplo, si se quisiera encontrar un número que se encuentra en el rango de 1 a 100, y se propone inicialmente el valor 28, al indicar que el número es mayor, se descartan todos los valores iguales o inferiores a 28. Si a continuación se sugiere el valor 80 y se señala que el número es menor, entonces se excluyen todos los valores iguales o superiores a 80. Como resultado, el rango de búsqueda se reduce únicamente a los valores comprendidos entre 29 y 79, representados en verde:

1	29	79	100

A partir de esta reducción, el proceso se repite aplicando la misma lógica de búsqueda binaria, acotando iterativamente el intervalo hasta encontrar el número objetivo.

Ordenamiento:

Es el proceso de organizar los elementos de una estructura de datos en un orden específico, siendo este fundamental ya que facilita la búsqueda eficiente, y permite analizar los datos de manera más clara optimizando el rendimiento de los algoritmos.

La elección del algoritmo va a depender de algunos factores como, el tamaño de la lista, el tipo de datos y los requisitos de rendimiento.

Algunos de los más comunes son:

Ordenamiento por burbuja: Compara e intercambia elementos repetidamente hasta que la lista esté ordenada.

Ordenamiento por selección: Selecciona el elemento más pequeño y lo coloca en su posición correcta en cada iteración.

Ordenamiento por inserción: Inserta cada elemento en el lugar correspondiente dentro de una sub lista ya ordenada.

Ordenamiento rápido: Divide la lista en partes usando un pivot y ordena recursivamente.

Ordenamiento por mezcla: Divide la lista en mitades, ordena cada una y luego las fusiona.

En conclusión, los algoritmos de búsqueda y ordenamiento son elementos esenciales en la programación, ya que permiten organizar y acceder a la información de manera eficiente

---

3. Caso Práctico

Se muestran dos implementaciones de búsqueda binaria y lineal a fin de poner en evidencia la utilidad de cada método.

En el primer caso se considera un dataset desordenado[1] que consta de 11 columnas y 4.484 filas, que podría considerarse un dataset pequeño.

```
import pandas as pd
import timeit

# Cargamos el dataset
```

[1] Sheta, Amrr. “ASAG data paraphrased.” *ASAG data paraphrased*, 2024, https://www.kaggle.com/datasets/amrrsheta/asag-data-paraphrased.

```
df = pd.read_csv("paraphrased_all_ASAG.csv")

# Buscamos algun valor que se encuentre repetido en el dataset, para
usar como ejemplo
valores_frecuencia = df['student_answer_paraphrased'].value_counts()
valor_objetivo = valores_frecuencia.index[0]

# Convertimos los valores a una lista para ejecutar las búsquedas
data_list = df['student_answer_paraphrased'].tolist()
sorted_list = sorted(data_list)
```

Como primer paso se elige una columna que contenga variedad de datos sobre donde ejecutaremos este caso de uso. Esta columna está compuesta por varias cadenas distintas.

Se utiliza pandas para facilitar la manipulación del dataset y se extrae el valor más frecuente en la lista. Es requisito que la lista se encuentre ordenada para ejecutar una búsqueda binaria, de modo que se almacena una copia de la lista ordenada en una variable separada.

```
# Definimos los métodos de búsqueda

def busqueda_lineal(data, objetivo):
  for i, val in enumerate(data):
    if val == objetivo:
      return i
  return -1

def busqueda_binaria(data, objetivo):
  izq, der = 0, len(data) - 1
  while izq <= der:
      med = (izq + der) // 2
      if data[med] == objetivo:
          return med
      elif data[med] < objetivo:
          izq = med + 1
      else:
          der = med - 1
  return -1

# Medimos el tiempo de cada método
```

```
tiempo_lineal = timeit.timeit(lambda: busqueda_lineal(data_list, valor_objetivo), number = 100)

tiempo_binario = timeit.timeit(lambda: busqueda_binaria(sorted_list, valor_objetivo), number=100)

# En el caso de búsqueda binaria, tenemos que considerar el tiempo que toma ordenar la lista
tiempo_ordenamiento = timeit.timeit(lambda: sorted(data_list), number=100)

tiempo_binario_total = tiempo_binario + tiempo_ordenamiento
```

Una vez implementada cada función de búsqueda, se utiliza la librería timeit para medir el tiempo de ejecución de cada método. Corremos la función 100 veces a fin de obtener un promedio y a su vez un número más legible. Debemos agregarle al tiempo de búsqueda binaria, el tiempo que tomó ordenar la lista para hacer una comparación justa.

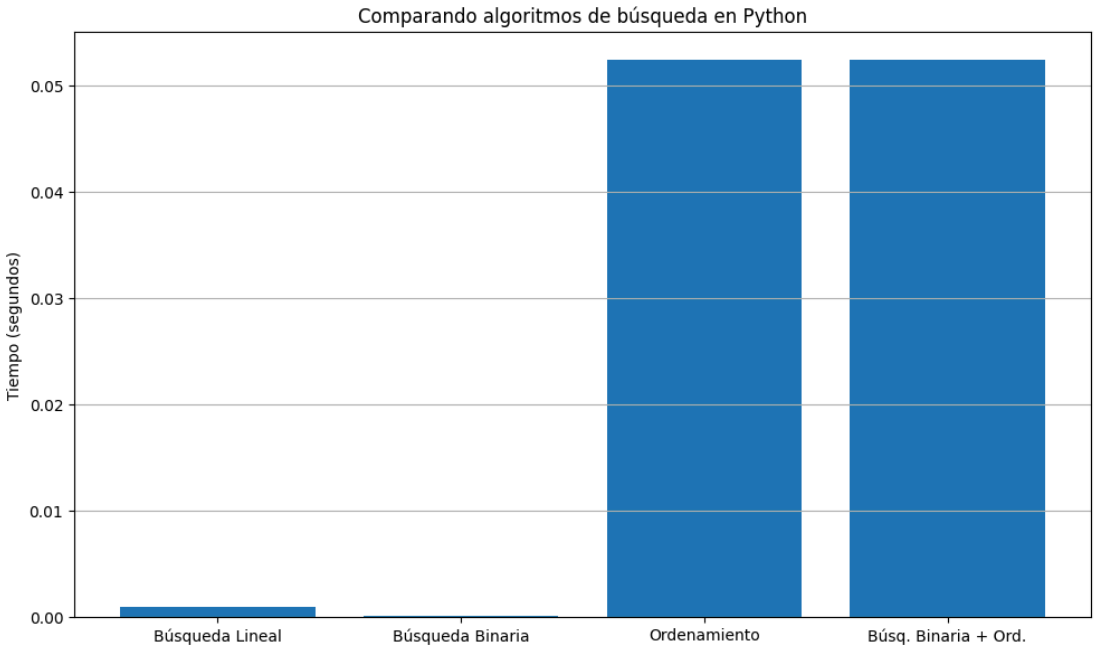
Por último, se despliegan los resultados:

```
# Imprimo resultados

print(f"Valor objetivo: {repr(valor_objetivo)}")
print(f"Tiempo de búsqueda lineal: {tiempo_lineal:.6f} segundos")
print(f"Tiempo de búsqueda binaria: {tiempo_binario:.6f} segundos")
print(f"Tiempo de ordenamiento: {tiempo_ordenamiento:.6f} segundos")
print(f"Total tiempo binario + tiempo de ordenamiento: {tiempo_binario_total:.6f} segundos")
```

```
Valor objetivo: 'No answer.'
Tiempo de busqueda lineal: 0.001002 segundos
Tiempo de busqueda binaria: 0.000072 segundos
Tiempo de ordenamiento: 0.046849 segundos
Total tiempo binario + tiempo de ordenamiento: 0.046921 seconds
```

Se utiliza la librería matplotlib para generar un gráfico simple que compara los tiempos de ejecución:

```python
import matplotlib.pyplot as plt

labels = ['Búsqueda Lineal', 'Búsqueda Binaria', 'Ordenamiento', 'Búsq. Binaria + Ord.']

tiempos = [tiempo_lineal, tiempo_binario, tiempo_ordenamiento, tiempo_binario_total]

# Se crea un gráfico de barras
plt.figure(figsize=(10, 6))
plt.bar(labels, tiempos)

plt.ylabel('Tiempo (segundos)')
plt.title('Comparando algoritmos de búsqueda en Python')
plt.grid(axis='y')

plt.tight_layout()

plt.show()
```



-Búsqueda binaria con función de ordenamiento por selección:

****Ordenamiento por selección****

(compara todos y pone el más chico al inicio)

-Primero importamos **timeit**, que sirve para medir el tiempo que demora la búsqueda binaria.

-Crea una lista de edades desordenadas

-Define la función **ordenar_edades**

-Guarda en **n** la cantidad de elementos que tiene la lista

-Luego la recorre con un **for i in range** buscando el valor menor, **pos_menor = i**

-Compara este valor con todos los que están a su derecha en la lista: **for j in range (i + 1, n):**

-Si encuentra un número más chico, guarda su posición como la nueva **pos_menor**

-Intercambia los valores, dejando la lista ordenada de menor a mayor

****Función binaria****

(Solo funciona con listas ordenadas)

-Se define una función que busca un valor en una lista.

```
import timeit

edades = [34, 21, 45, 19, 60, 27, 38, 22, 15, 8]

""" Función de ordenamiento por selección """

def ordenar_edades (lista):
    n = len(lista)
    for i in range (n):
        pos_menor = i
        for j in range (i + 1, n):
            if lista [j] < lista[pos_menor]:
                pos_menor = j
```

```
        lista [i], lista[pos_menor] = lista [pos_menor], lista[i]


""" Función de búsqueda binaria """

def busqueda_binaria(lista, valor):
    inicio = 0
    fin = len (lista) -1
```

-Definimos variables de inicio y fin y se repite mientra haya elementos por revisar

-Calcula la posición del medio de la lista actual **// 2**

-Si el número que está en el medio es el que se busca, devuelve esa posición, si es menor que el buscado, busca en la mitad derecha de la lista. Si es mayor, lo busca en la mitad izquierda. Si no lo encuentra, devuelve -1.

-Llama a la función para ordenar las edades, y luego las imprime ya ordenadas.

-Se le pide al usuario que ingrese un número.

-Para finalizar se mide el tiempo de ejecución.

```
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == valor:
            return medio
        elif lista[medio] < valor:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1

ordenar_edades(edades)
print("Edades ordenadas: ", edades)

valor_buscar = int(input("Que edad queres buscar? "))


""" Tiempo de busqueda """

inicio = timeit.default_timer()
resultado = busqueda_binaria(edades, valor_buscar)
fin = timeit.default_timer()
```

```
""" Mostrar resultado """

if resultado != -1:
      print(f"El valor {valor_buscar} fue encontrado en la posición
{resultado}")

else:
    print(f"El valor {valor_buscar} no se encuentra en la lista")

    print("Tiempo de busqueda:" , fin - inicio, "segundos")
```

---

4. Metodología utilizada

Durante el proceso, se siguieron las siguientes etapas:

Investigación teórica: Se consultaron sitios educativos y documentación técnica para comprender los fundamentos de los algoritmos utilizados. Las fuentes utilizadas están detalladas al final del informe.

Diseño y prueba del código: Cada integrante escribió su propio código en Python, lo probó y verificó su funcionamiento.

Herramientas utilizadas: Se empleó el entorno de desarrollo Visual Studio Code, el lenguaje de programación Python y el sistema de control de versiones Git para subir los archivos al repositorio compartido en GitHub.

Trabajo colaborativo: El trabajo fue realizado por dos participantes, en donde se acordó hacer un ejemplo práctico cada uno, para poder plantear dos formas de algoritmos.

---

5. Resultados obtenidos

El primer caso práctico pone en evidencia que, mientras que la búsqueda binaria es órdenes de magnitud más rápida que la búsqueda lineal, el tiempo dedicado en ordenar los datos es suficiente para perder el beneficio de velocidad que nos aporta. Se resalta la eficacia de la búsqueda lineal en el caso de uso y además la mayor sencillez de su implementación (nótese la diferencia entre la longitud y complejidad del código entre ambos tipos de búsqueda).

En el segundo caso, se demuestra que la búsqueda binaria es muy rápida y eficiente incluso con una lista relativamente corta. Además, el tiempo de ejecución medido con timeit fue mínimo, lo que confirma la eficiencia de búsqueda una vez que la lista ya se encuentra ordenada.

---

6. Conclusiones

Los casos prácticos en estudio dan a cuenta que no hay un mejor o peor método de búsqueda. Deben pensarse como herramientas diseñadas para casos particulares. Entender cómo funcionan y cómo se implementan nos da el conocimiento necesario para aplicarlos en el caso apropiado y no perder eficiencia. Existe un *trade off* entre conveniencia y eficiencia a la hora de usar estos métodos. Es por esto que vale la pena conocer qué tipo de algoritmos se encuentran detrás de las funciones de alto nivel que se invocan en los lenguajes de programación con los se trabajan *index()* en Python o *first(), any(), some()* en Javascript entre tantos otros.

---

7. Bibliografía

DataCamp. (s.f.). *Búsqueda lineal en Python: Tutorial paso a paso*. DataCamp. https://www.datacamp.com/es/tutorial/linear-search-python

Wikipedia. (s.f.). *Búsqueda lineal*. Wikipedia. https://es.wikipedia.org/wiki/B%C3%BAsqueda_lineal

Codecademy. (s.f.). *Linear and binary search cheatsheet*. Codecademy. https://www.codecademy.com/learn/search-algorithms/modules/linear-binary-search/cheatsheet

Sheta, A. R. (s.f.). *ASAG Data Paraphrased*. Kaggle. https://www.kaggle.com/datasets/amrrsheta/asag-data-paraphrased/data

BBC Bitesize. (s.f.). *Linear search - Searching algorithms*. BBC. https://www.bbc.co.uk/bitesize/guides/zgr2mp3/revision/2

Python Software Foundation. (s.f.). *array — Efficient arrays of numeric values*. En *The Python Standard Library*. https://docs.python.org/3/library/array.html#array.array.index

Khan Academy. (s.f.). Binary search. Recuperado de https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

8. Anexos

Repositorio en Github - https://github.com/martinetoledo/TUP-TPI-P1

Video explicativo - https://www.youtube.com/watch?v=VbsNx0dgkPc