Trabajo Práctico Integrador
Programación I

- **Título del trabajo**: Algoritmos de Búsqueda y ordenamiento

- **Alumnos**: Rousseaux Carolina, Rosales Gastón

- **Materia**: Programación I

- **Profesor/a**: Prof. Nicolás Quirós

- **Fecha de Entrega**: 09/06/2025

Índice


UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

1.Introducción

Los algoritmos de búsqueda y ordenamiento son herramientas fundamentales en la programación. La búsqueda es una operación clave y común en muchas aplicaciones que consiste en encontrar uno o varios datos que cumplan con un criterio dado dentro de un conjunto de datos. También se define simplemente como localizar un elemento en una lista o tupla. Por otro lado, el ordenamiento tiene el propósito de organizar los datos dentro de un arreglo u otra estructura de datos. El ordenamiento organiza los datos de acuerdo a un criterio, por ejemplo; de menor a mayor.

Cuando se trabaja con grandes cantidades de información almacenada, usualmente es necesario buscar un dato en enormes volúmenes, esto subraya la relevancia de este tipo de algoritmos en programación y a la vez nos da la pauta de lo importante que es la resolución de problemas relacionados con el orden de búsquedas y ordenamientos.

Este trabajo intentará identificar los distintos tipos de algoritmos propuestos para la búsqueda y ordenamiento de datos y cuál es su importancia ante problemas que se puedan presentar, diagnosticando cual es más eficaz que otro, según el contexto que lo requiera. En él, incluimos conceptos como búsqueda lineal, búsqueda binaria, tipos de ordenamiento como burbuja, por inserción, quicksort, o por mezcla(merge sort) y la notación Big O.

2. Marco Teórico

Podemos empezar a desarrollar este trabajo introduciéndonos en los algoritmos de búsqueda uno de los factores importantes para encontrar uno o varios datos según el criterio del programador.
Si bien existen varios algoritmos de búsqueda este grupo hará hincapié en los 3 que se consideraron más importantes.

2.1 Algoritmos de búsqueda

- **Búsqueda lineal:** También conocido como búsqueda secuencial, es el algoritmo de búsqueda más simple. Consiste en recorrer cada elemento

del conjunto de datos de forma secuencial, uno por uno hasta encontrar el elemento deseado o determinar que no está incluido en la lista. La eficacia de este algoritmo, está directamente relacionada al tamaño de la lista de datos.

Es un algoritmo fácil de implementar, pero por otro lado puede ser lento para conjuntos de grandes datos, y más aún cuando estos no están ordenados.

Sintaxis en Python de la búsqueda lineal.

```
''' Busqueda lineal del indice con el mayor valor '''
def obterner_indice_del_mayor(vector):
  indiceDelMayor = 0
  for i in range(1, len(vector)):
    # Si el valor actual (en la posición i) tiene
    # un valor mayor al anteriormente detectado se
    # actualiza el indiceDelMayor con el actual.
    if vector[i] > vector[indiceDelMayor]:
      indiceDelMayor = i

  return indiceDelMayor
```

- **Búsqueda binaria**: Es un algoritmo de búsqueda eficiente que funciona **solo en conjuntos de datos ordenados.** Se basa en la estrategia de dividir el problema en subproblemas, en este caso compara el elemento del medio entre dos extremos.

  Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente, repitiendo el proceso hasta encontrarlo o determinar que no se encuentra. Esto reduce el tamaño del problema con cada paso.

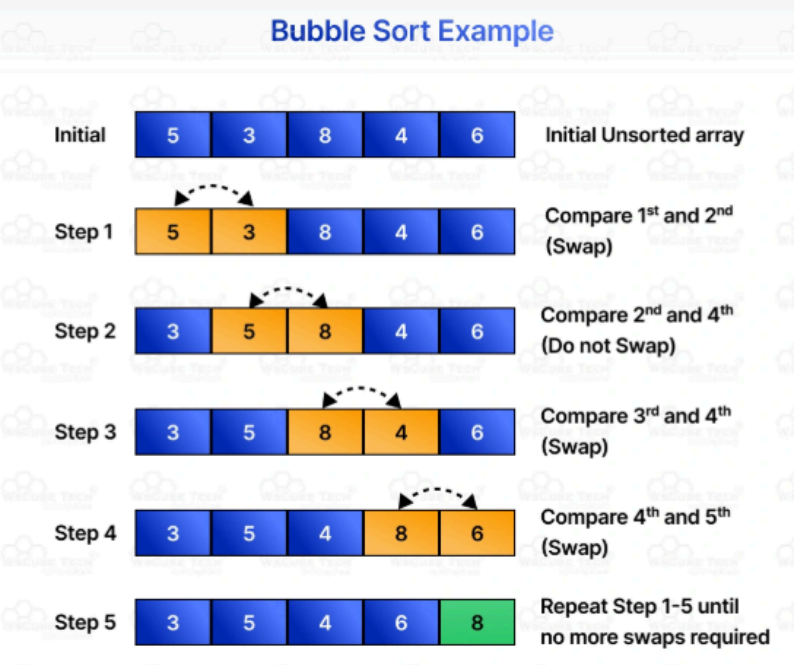
Sintaxis búsqueda Binaria en Python.

```python
def busqueda_binaria(vector, buscado):
  izq = 0
  der = len(vector)-1
  # Mientras sean válidos los límites:
  while izq <= der:
    # se calcula el índice de la mitad
    med = int((izq + der) / 2)
    # Si se encontró el elemento buscado se devuelve el índice:
    if vector[med] == buscado:
      return med
    # Se calcula el nuevo límite:
    if buscado < vector[med]:
      der = med - 1
    else:
      izq = med + 1
  # Devolver un valor de índice no válido:
  return -1
```

- **Búsqueda de hash**: Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes

2.2 Algoritmos de Ordenamiento.

- **Bubble Sort**: O el ordenamiento de burbuja, se basa en recorrer el conjunto comparando elementos adyacentes entre sí e intercambiandolos si están en el orden incorrecto según el criterio deseado por el programador. Esto debe repetirse una y otra vez sobre el conjunto hasta ordenarlo. Es simple y fácil de implementar, pero no es eficiente para conjuntos de grandes volúmenes.



- **Selection Sort**: Funciona encontrando el elemento más pequeño de la lista y luego intercambiándolo con el elemento de la posición actual. Este proceso se repite hasta que todos los elementos estén ordenados. Es más eficiente que bubble sort pero se considera lento para conjuntos grandes.

  Sintaxis del ordenamiento por selección en Python:

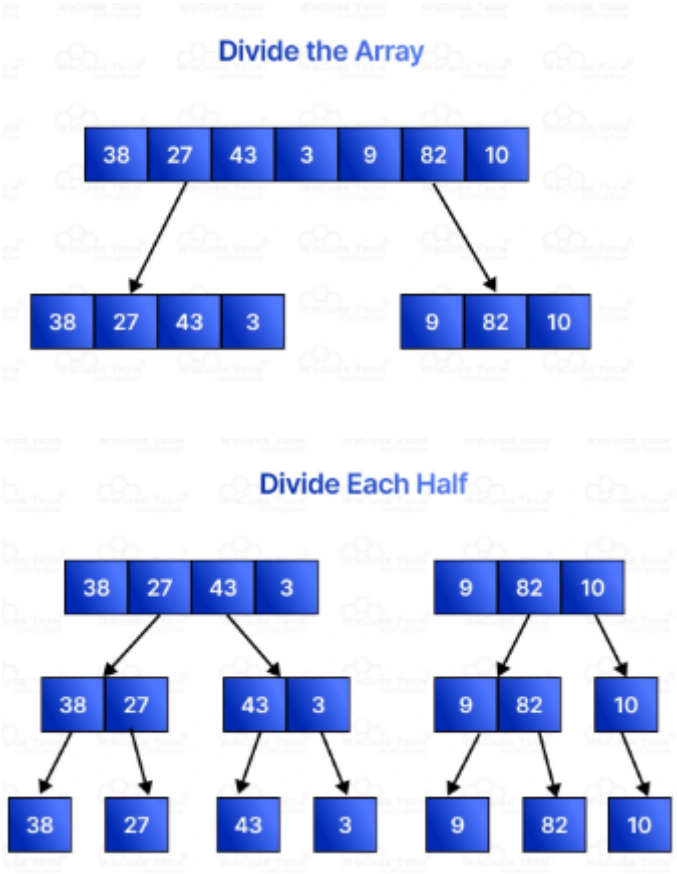
```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

# Example usage
arr = [64, 25, 12, 22, 11]
print("Sorted array is:", selection_sort(arr))
```

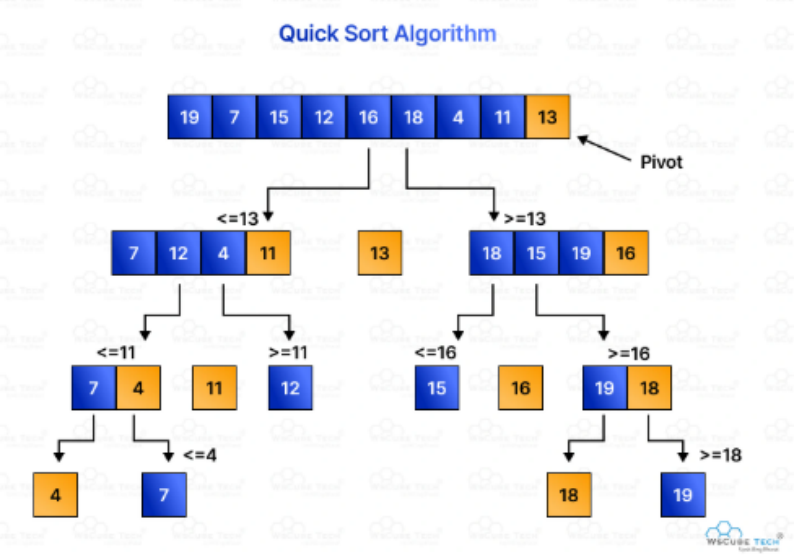
- **Insertion Sort**: En este algoritmo se recorre un vector y se toma el valor evaluado, este se compara con cada uno de los elementos anteriores hasta insertarlo en el lugar correcto, así, se construye una lista ordenada elemento por elemento.

Sintaxis del ordenamiento por inserción en Python:

- **Merge Sort:** Es un método de ordenamiento que se basa en dividir el stack o lista propuesta en pequeños grupos, ordenar cada grupo, y entonces reagruparlos en orden.



- **Quick Sort**: El ordenamiento rápido o quicksort es un algoritmo de los más potentes y eficientes. Basado en la técnica “divide y vencerás", se selecciona un “pivote” y organiza los elementos menores a un lado y los mayores al otro. Esto se aplica recursivamente a las sublistas.



2.3 Notación Big O

- La notación Big O describe la tasa de crecimiento de la complejidad de un algoritmo. Representa el tiempo que tarda uno en ejecutarse en función del tamaño de la entrada. Por ejemplo, si un algoritmo tiene una complejidad de O(n), tardará el doble de tiempo en ejecutarse si el tamaño de la entrada se duplica.

  Esta notación se utiliza específicamente para describir el peor caso de complejidad de tiempo en un algoritmo, indicando que el algoritmo nunca tardará más de O(n) tiempo, para una entrada de tamaño n.

  Por ejemplo, para los casos ya vistos anteriormente, podemos mencionar algunos como:

  Búsqueda lineal → O(n): El tiempo de búsqueda es directamente proporcional al tamaño de la lista

  Búsqueda Binaria → O(log n): El tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. Esto significa que para listas grandes es mucho más eficiente que la búsqueda lineal.

  Bubble Sort → O(n^2): Compara repetidamente pares de elementos adyacentes y los intercambia si están en el orden incorrecto, por lo que su rendimiento es lento, y lo hace improbable de usar en grandes aplicaciones.

3. Caso Práctico

En función de lo expuesto anteriormente, vamos a trabajar sobre una serie de algoritmos de búsqueda.

3.1 Algoritmo n°1: búsqueda lineal

Para el caso de los algoritmos de búsqueda lineal, vamos a crear un algoritmo donde se ingrese el nombre de un ciudad, y se acceda a un dato meteorológico, como la temperatura por ejemplo.

En este caso vamos a trabajar solo con las temperaturas, pero podrían incluirse otros datos, como presión, precipitaciones, media de temperaturas mínimas o máximas anuales, etc.

Bienvenido | algoritmo_meteorología.py

algoritmo_meteorología.py > ...

```python
#Creamos un diccionario con las ciudades y sus respectivas temperaturas
meteorologia = [
                {"ciudad": "Rio Gallegos", "temperatura": 5},
                {"ciudad" :"Paraná", "temperatura": 9},
                {"ciudad" : "Buenos Aires", "temperatura" : 10}
]

#Para buscar el dato, solicitamos el ingreso de un dato al usuario

consulta = input("Ingrese la ciudad que desea consultar: ")

# Acá lo que hacemos es buscar en nuestros diccionarios si está el dato ingresado por el usuario.
# Si está la ciudad y se corrobora que es igual al dato ingresado sale del bucle.
for i in meteorologia:
    if i["ciudad"].lower() == consulta.lower():
        resultado_busqueda = i
        break

#Aca una vez que tenemos un resultado para la búsqueda, lo mostramos por pantalla.
if resultado_busqueda:
    print(f"En la ciudad {resultado_busqueda['ciudad']}, hacen {resultado_busqueda['temperatura']} grados centígrados.")
else:
    print("Error. No se encontró la ciudad que estás buscando")
```

Terminal de salida

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  GITLENS                                                    Python

PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/PC/Desktop/programacion 1/Busqueda_y_Ordenamiento/algoritmo_meteorologia.py"
Ingrese la ciudad que desea consultar: Buenos aires
En la ciudad Buenos Aires, hacen 10 grados centígrados.
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/PC/Desktop/programacion 1/Busqueda_y_Ordenamiento/algoritmo_meteorologia.py"
Ingrese la ciudad que desea consultar: La Plata
Traceback (most recent call last):
  File "c:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento\algoritmo_meteorología.py", line 21, in <module>
    if resultado_busqueda:
       ^^^^^^^^^^^^^^^^
NameError: name 'resultado_busqueda' is not defined
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento>
```

3.2 Algoritmo n°2: Búsqueda binaria

Para este caso nos resultó conveniente trabajar tomando como ejemplo la selección de butacas de un cine, ya que son elementos que están numerados y en orden. De la misma forma, se podría aplicar este algoritmo de búsqueda, a butacas de avión, habitaciones de hotel, etc; ya que son grandes listas de datos, y están ordenadas.

Bienvenido | algoritmo_cine.py

algoritmo_cine.py > ...

```python
# En este algoritmo vamos a trabajar con una lista de butacas de cine, para ver si estan vacías u ocupadas

# Definimos la función para la busqueda binaria
def busqueda_binaria(butacas_ocupadas,consulta): #Defino función con parámetros
    izquierda = 0              #Posicion 0 de la lista
    derecha = len(butacas_ocupadas) - 1      #Ultima posicion de la lista (longitud de nuestra lista - 1)

    while izquierda <= derecha:    #Mientras se cumpla la condicion de busqueda en ese rango
        medio = (izquierda + derecha) // 2         #Buscamos el punto medio de la lista ordenada

        if butacas_ocupadas[medio] == consulta:  #Comparo el índice medio con el valor de consulta
            return medio                        #Si es igual, retorno ese valor
        elif butacas_ocupadas[medio] < consulta:   #Si el valor medio es menor del valor que queremos buscar
            izquierda = medio + 1               #Se actualizan las variables de izquierda y derecha
        else:                                   # Y se actualiza el índice
            derecha = medio - 1

    return None

# Programa principal

#Creamos una lista
butacas_ocupadas = []
for i in range(1,51,5):
    butacas_ocupadas.append(i)

#Hacemos la consulta de la butaca
consulta = int(input("Ingrese el número de butaca que desea consultar: "))

#Mostramos por pantalla
print("El número de butacas ocupadas es: ", butacas_ocupadas)

resultado = busqueda_binaria(butacas_ocupadas, consulta)

if resultado != None:  #Se evalúa si la butaca está libre u ocupada.
    print(f"La butaca {consulta} está ocupada.")
else:
    print(f"La butaca {consulta} está libre.")
```

Terminal de salida

```
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/PC/Desktop/programacion 1/Busqueda_y_Ordenamien
to/algoritmo_cine.py"
Ingrese el número de butaca que desea consultar: 15
El número de butacas ocupadas es:  [1, 6, 11, 16, 21, 26, 31, 36, 41, 46]
La butaca 15 está libre.
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> ^C
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/PC/Desktop/programacion 1/Busqueda_y_Ordenamien
to/algoritmo_cine.py"
La butaca 15 está libre.
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> ^C
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/PC/Desktop/programacion 1/Busqueda_y_Ordenamien
to/algoritmo_cine.py"
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/PC/Desktop/programacion 1/Busqueda_y_Ordenamien
to/algoritmo_cine.py"
to/algoritmo_cine.py"
Ingrese el número de butaca que desea consultar: 31
El número de butacas ocupadas es:  [1, 6, 11, 16, 21, 26, 31, 36, 41, 46]
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> ^C
PS C:\Users\PC\Desktop\programacion 1\Busqueda_y_Ordenamiento> & C:/Users/PC/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/PC/Desktop/programacion 1/Busqueda_y_Ordenamien
to/algoritmo_cine.py"
Ingrese el número de butaca que desea consultar:
```

4. Metodología Utilizada

Se trabajó con el material propuesto por la cátedra, y los códigos se fueron probando en Visual Studio Code. En un primer momento, se consultó el material disponible, tanto pdfs, power points y el libro Pensar en Python, para obtener la información teórica necesaria sobre los algoritmos de búsqueda y ordenamiento. Luego, se definieron los casos prácticos en los que trabajaríamos, para probarlos se utilizó Visual Studio Code. Se pensaron situaciones que suelen presentarse en la vida cotidiana, para implementar los algoritmos, y que de ésta forma sean más comprensibles. Por ejemplo, los casos que tomamos fueron para búsqueda lineal, utilizamos la búsqueda de una ciudad con su temperatura correspondiente; y para el caso de búsqueda binaria; utilizamos el caso de la selección de butacas de cine.

Como forma de trabajo colaborativo utilizamos docs.google, donde se fue trabajando en simultáneo, y editando el documento.

5. Resultados Obtenidos

Para el caso del algoritmo número 1, el algoritmo de búsqueda lineal, vimos que el algoritmo funciona correctamente al tener un bajo volúmen de datos. Se ingresa el dato, el nombre de la ciudad en este caso, y nos devuelve la temperatura correspondiente.

Para el algoritmo número 2, de búsqueda binaria, trabajamos con el ejemplo de selección de butacas en un cine. Ya que nos pareció apropiado porque son datos que se encuentran en orden. Y sería una forma sencilla de probar el funcionamiento de la estructura de búsqueda binaria.

Para este último caso, en función de la creación de una lista ordenada de números, solicitamos al usuario el ingreso de un número de butacas, el algoritmo procesa el dato de entrada comparándolo con lista general de butacas, y nos devuelve el valor de salida. Si el valor ingresado está en la lista, nos devuelve que la butaca está ocupada, sino que la butaca está libre.

6. Conclusiones

La búsqueda es importante en programación porque se utiliza en una amplia variedad de aplicaciones.
Con el algoritmo del caso número 1, de búsqueda lineal, vimos que como dice la definición de algoritmos de búsqueda lineal, solo son eficientes para situaciones donde se maneja un reducido volúmen de datos, ya que va comparando el dato ingresado para consultar 1 a 1 con los elementos de la lista. En el caso trabajado, si bien hicimos la prueba para unas pocas ciudades cargadas, el algoritmo funcionó; pero si ampliamos el área de prueba, a todas las ciudades de Argentina, por ejemplo; resultaría ser un algoritmo lento, es decir, ineficiente cuando el volúmen de datos que se maneja es muy grande. Sin embargo, nos pareció que para ese caso en particular, era el algoritmo de búsqueda apropiado, ya que los datos no se encuentran en orden, y al generar una búsqueda de una ciudad determinada, nos devuelve la temperatura correspondiente.

Para búsquedas complejas, con mucho volumen de datos, es recomendable utilizar un algoritmo de búsqueda binaria, antes que un algoritmo lineal, ya que es más rápido y eficiente. Como mostramos en el caso número 2, donde implementamos un algoritmo de búsqueda binaria para la selección de butacas de cine. Nos pareció que es un algoritmo mucho más eficiente para listas con grandes volúmenes de datos que se encuentran de forma más ordenada, porque la metodología con la que trabaja, implica que se reduce el espacio de búsqueda a la mitad, comparando el valor a consultar con el valor medio de la lista, lo que reduce el área de búsqueda.

7.Bibliografía

- Downey Allen. Pensar en Python. 2da. Edición, Versión 2.4.0. Green Tea Press
- https://www.wscubetech.com/resources/dsa/bubble-sort (consultada el 6/6/25)
- https://visualgo.net/en/sorting?slide=20(consultada el 6/6/25)
- https://algorithm-visualizer.seancoughlin.me/searching/linearS earch(consultada el 7/6/25)
- Búsqueda y ordenamiento - Taller de programación, Martín S. Avalos.

8. Anexos

Ilustración del tiempo que tardan los algoritmos abordados en el presente trabajo, en encontrar el dato a consultar.

Por ejemplo, en la búsqueda lineal, vemos que cuanto más grande es la lista de datos, mayor es el tiempo que tarda en consultar. Esto se ve en detalle en la gráfica de tipo exponencial, de color azul. En cambio, para los algoritmos de búsqueda binaria, el cual se ve representado en la gráfica naranja, de tipo logarítmica, vemos que no hay un aumento significativo en el tiempo que se emplea en buscar un elemento en listas de grandes volúmenes de datos.

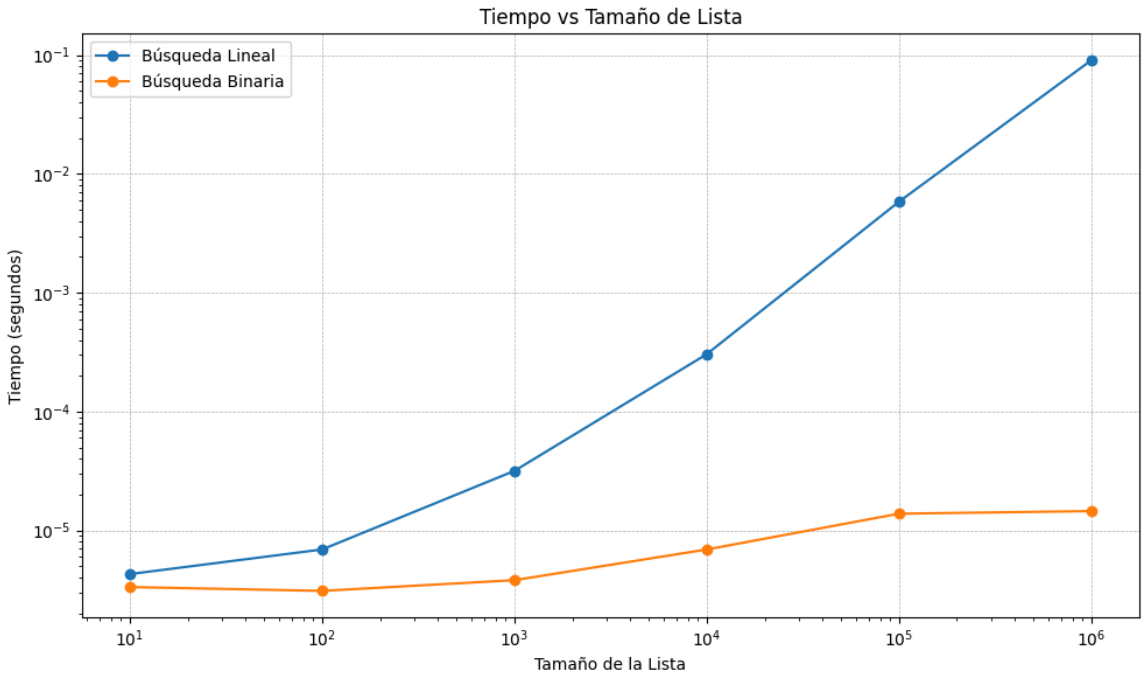


Imagen obtenida del material provisto por la cátedra.