

# Algoritmos de Búsqueda y Ordenamiento

*Mauro Zavatti - zavattimauro@gmail.com*

*Agustina Aguilera - Agustina\_aguilera@outlook.com.ar*

Programación I

Prof. Nicolás Quirós

\*Fecha de Entrega: 2 de mayo de 2025

Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía

Introducción

El presente trabajo se centra en el estudio de los algoritmos de búsqueda y ordenamiento, un tema fundamental dentro del campo de la programación y el desarrollo de software. Se eligió esta temática debido a su relevancia práctica y teórica, ya que permite comprender cómo se gestionan y manipulan grandes volúmenes de datos de forma eficiente.

En la programación, los procesos de búsqueda y ordenamiento son esenciales para optimizar el rendimiento de las aplicaciones, facilitar el acceso a la información y garantizar una mejor experiencia de usuario. Por ejemplo, al trabajar con bases de datos que contienen miles de registros, contar con algoritmos adecuados permite obtener resultados precisos en el menor tiempo posible, reduciendo el consumo de recursos y minimizando errores.

Mediante este trabajo aprenderemos a entender la lógica detrás de los algoritmos y el funcionamiento, siendo capaces de poder aplicarlos en las circunstancias necesarias.

## Marco Teórico

### Búsqueda y ordenamiento.

Los algoritmos de ordenamiento organizan los datos de acuerdo a un criterio, como de menor a mayor o alfabéticamente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

La búsqueda es una herramienta poderosa que se utiliza en una amplia variedad de aplicaciones de programación. Al comprender los diferentes algoritmos de búsqueda y cómo utilizarlos, puede mejorar el rendimiento y la eficiencia de sus programas. La búsqueda consiste en localizar un elemento en un conjunto de datos. Los dos métodos más comunes son la búsqueda lineal y la búsqueda binaria, pero en este trabajo investigaremos acerca de la búsqueda de interpolación, que es una forma un poco mas avanzada que la búsqueda binaria.

Si comparamos la búsqueda de interpolación con las búsquedas lineal y binaria podemos notar una gran diferencia entre el tiempo que llevo su ejecución, ya que la búsqueda de interpolación tiene intención de señalar un bloque donde posiblemente este la posición del valor dado a partir de una lista ordenada, funcionando mucho mejor cuando los elementos están distribuidos de manera uniforme.

Como ejemplo si intentara buscar el numero 19 en una lista ordenada con 15 numeros, lo mas probable es que este en el ultimo bloque. Asi reduciría el tiempo que se tardo en encontrar el bloque en comparación con la búsqueda binaria que comenzaría desde el centro.

¿Qué tan eficiente es?

Método	Mejor caso	Promedio	Peor caso	Requiere orden?
Búsqueda lineal	$O(1)$	$O(n)$	$O(n)$	No
Búsqueda binaria	$O(1)$	$O(\log n)$	$O(\log n)$	Si
Búsqueda de interpolación	$O(1)$	$O(\log \log n)$	$O(n)$	Si

## Caso Práctico

Se utilizó una lista y se aplicó el método de búsqueda de interpolación para encontrar un valor específico.

```
import time
```

```
lista = [1, 5, 57, 20, 34, 2, 8, 11, 45, 16, 30]
```

```
x = 22
```

```
arr= sorted(lista)
```

```
print ("Lista ordenada", arr)
```

```
print("Buscaremos el numero", x)
```

```
def busqueda_interpolacion(arr, x):
```

```
    i = 0  #Índice inicial
```

```
    j = len(arr) - 1  #índice final
```

```
    pos = 0 #posición estimada del elemento
```

```
    pasos = 0 # hacemos un contador para los pasos realizados
```

```
    #Bucle principal
```

```
    #controlamos que el índice bajo no sea mayor que el índice alto y
```

```
    # que el valor buscado este dentro del rango actual que es arr[i] a arr[j]
```

```
    while i <= j and arr[i] <= x <= arr[j]:
```

#Calculamos la posicion estimada (pos) donde podria estar x.

#evitamos division por cero si los extremos son iguales

```
if arr[i] == arr[j]:
```

```
    if arr[i] == x:
```

```
        return i
```

```
    else:
```

```
        return -1
```

#Estimamos la posicion probable de x usando interpolacion lineal.

#La formula calcula en que punto del interbalo (i a j) podria estar x,

#suponiendo que los datos estan distribuidos uniformemente.

#Esto ayuda a reducir la cantidad de pasos en comparacion a una busqueda binaria.

```
pos = i + (((x - arr[i]) * (j - i) // (arr[j] - arr[i])))
```

```
pasos += 1
```

```
if arr[pos] == x:
```

```
    print(f"Pasos realizados: {pasos}")
```

```
    return pos
```

```
elif arr[pos] < x:
```

```
    i = pos + 1
```

```
else:
```

```
    j = pos - 1
```

```
return -1 #Si no se encuentra
```

#Usamos time para medir cuanto tarda en ejecutarse la busqueda

inicio = time.time()

busqueda = busqueda\_interpolacion(arr, x)

fin = time.time()

if busqueda != -1:

print(f'El numero {x} esta en la posicion {busqueda}')

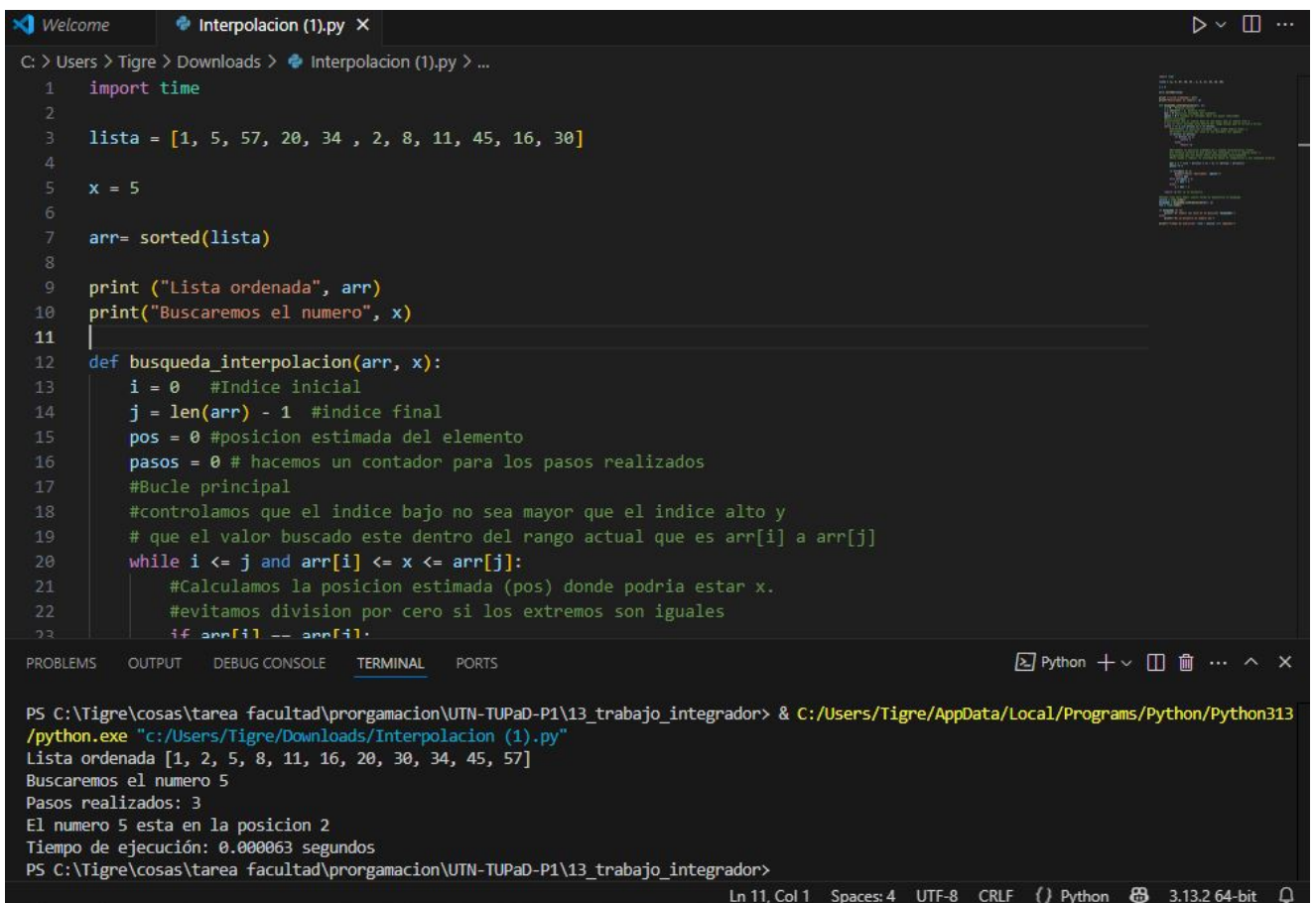
else:

print(f'No se encontro el numero {x}')

print(f'Tiempo de ejecución: {fin - inicio:.6f} segundos")

Capturas de los diferentes resultados:

Busqueda norma:



```
1 import time
2
3 lista = [1, 5, 57, 20, 34, 2, 8, 11, 45, 16, 30]
4
5 x = 5
6
7 arr= sorted(lista)
8
9 print ("Lista ordenada", arr)
10 print("Buscaremos el numero", x)
11
12 def busqueda_interpolacion(arr, x):
13     i = 0 #Indice inicial
14     j = len(arr) - 1 #indice final
15     pos = 0 #posicion estimada del elemento
16     pasos = 0 # hacemos un contador para los pasos realizados
17     #Bucle principal
18     #controlamos que el indice bajo no sea mayor que el indice alto y
19     # que el valor buscado este dentro del rango actual que es arr[i] a arr[j]
20     while i <= j and arr[i] <= x <= arr[j]:
21         #Calculamos la posicion estimada (pos) donde podria estar x.
22         #evitamos division por cero si los extremos son iguales
23         if arr[i] == arr[j]:
24             return -1
25         pos = i + ((x - arr[i]) * (j - i)) // (arr[j] - arr[i])
26         pasos += 1
27         if arr[pos] == x:
28             return pos
29         if arr[pos] < x:
30             i = pos
31         else:
32             j = pos
33     return -1
```

PS C:\Tigre\cosas\tarea facultad\prorgamacion\UTN-TUPaD-P1\13\_trabajo\_integrador> C:/Users/Tigre/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Tigre/Downloads/Interpolacion (1).py"

Lista ordenada [1, 2, 5, 8, 11, 16, 20, 30, 34, 45, 57]

Buscaremos el numero 5

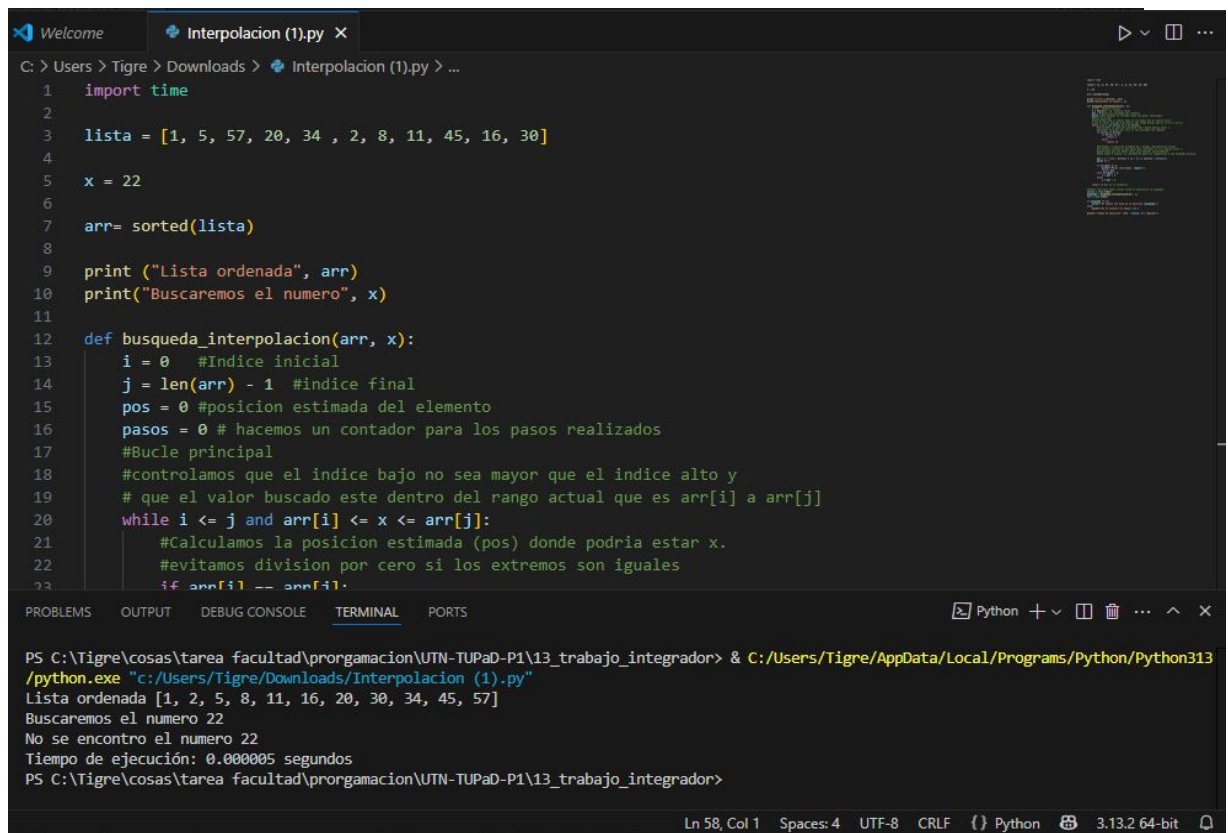
Pasos realizados: 3

El numero 5 esta en la posicion 2

Tiempo de ejecución: 0.000063 segundos

PS C:\Tigre\cosas\tarea facultad\prorgamacion\UTN-TUPaD-P1\13\_trabajo\_integrador>

Busqueda de elemento inexistente:



```
1 import time
2
3 lista = [1, 5, 57, 20, 34, 2, 8, 11, 45, 16, 30]
4
5 x = 22
6
7 arr= sorted(lista)
8
9 print ("Lista ordenada", arr)
10 print("Buscaremos el numero", x)
11
12 def busqueda_interpolacion(arr, x):
13     i = 0 #Indice inicial
14     j = len(arr) - 1 #indice final
15     pos = 0 #posicion estimada del elemento
16     pasos = 0 # hacemos un contador para los pasos realizados
17     #Bucle principal
18     #controlamos que el indice bajo no sea mayor que el indice alto y
19     # que el valor buscado este dentro del rango actual que es arr[i] a arr[j]
20     while i <= j and arr[i] <= x <= arr[j]:
21         #Calculamos la posicion estimada (pos) donde podria estar x.
22         #evitamos division por cero si los extremos son iguales
23         if arr[i] == arr[j]:
24             pos = i
25         else:
26             pos = int((j-i) * (x-arr[i]) / (arr[j]-arr[i]))
27         pasos += 1
28         if arr[pos] == x:
29             return pos, pasos
30         if arr[pos] < x:
31             i = pos + 1
32         else:
33             j = pos - 1
34     return -1, pasos
```

PS C:\Tigre\cosas\tarea facultad\prorgamacion\UTN-TUPaD-P1\13\_trabajo\_integrador> & C:/Users/Tigre/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Tigre/Downloads/Interpolacion (1).py"

Lista ordenada [1, 2, 5, 8, 11, 16, 20, 30, 34, 45, 57]

Buscaremos el numero 22

No se encontro el numero 22

Tiempo de ejecucion: 0.000005 segundos

PS C:\Tigre\cosas\tarea facultad\prorgamacion\UTN-TUPaD-P1\13\_trabajo\_integrador>

## Metodología Utilizada

La elaboración del trabajo se realizó en las siguientes etapas:

- Recolección de información teórica en documentación confiable.
- Implementación en Python de los algoritmos estudiados.
- Pruebas con diferentes conjuntos de datos.
- Registro de resultados y validación de funcionalidad.
- Elaboración de este informe y preparación de anexos.

## Resultados Obtenidos

## Conclusiones

Los algoritmos de búsqueda y ordenamiento son indispensables para el manejo de datos, ya que permiten al usuario acceder a la información de forma organizada y eficiente. Su correcta implementación puede mejorar significativamente el rendimiento de los sistemas, aumentando la velocidad de procesamiento y optimizando el uso de recursos.

En particular, la búsqueda por interpolación destaca por su eficiencia en arreglos ordenados y con distribución uniforme, ya que estima la posición probable del elemento buscado. Esto permite reducir considerablemente el tiempo de búsqueda al acceder directamente a la zona donde es más probable que se encuentre el objetivo.

#### Bibliografía –

Medium <https://medium.com/@diadasia-chilensis/b%C3%BAsqueda-por-interpolaci%C3%B3n-en-python-explicaci%C3%B3n-l%C3%ADnea-por-l%C3%ADnea-29f6980a3fdb>

Apunte de búsqueda y ordenamiento – tupad1