

Algoritmos de Búsqueda y Ordenamiento

Alumno: Ruff Luca Genaro – lucaruff2001@gmail.com

Materia: Programación 1

Profesor/a: Nicolás Quirós

Tutor: Neyén Bianchi

Fecha de Entrega: 25 de julio de 2025

Índice

Algoritmos de Búsqueda y Ordenamiento	1
Introducción	1
Marco Teórico	2
Caso practico	3
Metodología Utilizada	6
Resultados Obtenidos:	7
Conclusiones	7
Bibliografía	8
Anexos	9

Introducción

En programación utilizamos los algoritmos de búsqueda y ordenamiento para organizar y localizar datos de manera eficiente, siendo importantes para el desarrollo del software y la resolución de problemas computacionales.

Se eligió el tema debido a la importancia que tiene en el desarrollo de los softwares modernos, se busca comprender como funcionan estos algoritmos y cuando conviene utilizarlos para que el programador tome decisiones mas acertadas y mejorar la calidad de sus soluciones

El objetivo es analizar estos algoritmos y demostrar como pueden combinarse para resolver problemas de forma mas eficiente, utilizando la búsqueda lineal y binaria junto con métodos de ordenamiento de Burbuja y Quicksort.

Marco Teórico

Los algoritmos de búsqueda y ordenamiento son herramientas esenciales en programación, ya que permiten localizar y organizar información de manera eficiente.

La búsqueda consiste en localizar un elemento específico dentro de la estructura de datos, en los algoritmos que utilizamos se encuentra:

- **Búsqueda lineal:** Recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Su complejidad es $O(n)$
- **Búsqueda binaria:** Solo funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente, repite el proceso hasta encontrar el elemento o determina que no esta en el conjunto de datos, su complejidad es $O(\log n)$

En la practica del trabajo se utiliza la búsqueda binaria, pero para aplicarla correctamente requerimos previamente que la lista esta ordenada.

El ordenamiento organiza los datos en un cierto orden (Ej. De menor a mayor), lo cual facilita la búsqueda y otras operaciones, en los algoritmos que utilizamos se encuentra:

- **Bubble Sort (burbuja):** Compara elementos adyacentes y los intercambia si están desordenados. Es sencillo de implementar, pero poco eficiente en listas grandes, complejidad: $(O(n^2))$.
- **Quicksort:** Utiliza el enfoque "divide y conquista". Selecciona un pivote, divide la lista en menores y mayores, y aplica el mismo proceso recursivamente. Su complejidad: $O(n \log n)$ y suele ser mucho más rápido que otros métodos básicos.

El funcionamiento en el caso practico es el siguiente:

1. Se define una lista desordenada y un número a buscar (ejemplo: 32).
2. Se aplica **búsqueda lineal** sobre la lista original.
3. Se ordena la lista con **Bubble Sort**.
4. Se aplica **búsqueda binaria** sobre esa lista ordenada.

5. Se vuelve a ordenar la lista original usando **Quicksort**.
6. Se aplica **búsqueda binaria** sobre la lista ordenada con Quicksort.

Caso practico

En el desarrollo del programa en Python nos encontramos una lista desordenada de números enteros y necesitamos buscar un determinado numero que se encuentra en la lista.

Estructura del código:

```
# 1. Ordenamiento por burbuja
def bubble_sort(lista): # Función de ordenamiento burbuja
    n = len(lista) # Obtiene la longitud de la lista
    # Recorre toda la lista
    for i in range(n):
        # Últimos i elementos ya están en su lugar
        for j in range(0, n-i-1):
            # Si el elemento actual es mayor que el siguiente
            if lista[j] > lista[j+1]:
                # Intercambia los elementos
                lista[j], lista[j+1] = lista[j+1], lista[j]
    return lista # Devuelve la lista ordenada

# 2. Ordenamiento quicksort
def quicksort(lista): # Función de ordenamiento quicksort
    # Caso base: lista vacía o de un solo elemento
    if len(lista) <= 1:
        return lista # Devuelve la lista tal cual
    else:
        pivote = lista[0] # Elegimos el primer elemento como pivote
        # Elementos menores al pivote
        menores = [x for x in lista[1:] if x < pivote]
        # Elementos mayores o iguales al pivote
        mayores = [x for x in lista[1:] if x >= pivote]
        # Ordena recursivamente y combina
        return quicksort(menores) + [pivote] + quicksort(mayores)

# 3. Búsqueda lineal
def busqueda_lineal(lista, objetivo): # Función de búsqueda lineal
    # Recorre la lista
    for i in range(len(lista)):
        # Si encuentra el objetivo
        if lista[i] == objetivo:
            return i # Devuelve la posición
    return -1 # Devuelve -1 si no lo encuentra
```

```

# 4. Búsqueda binaria
def busqueda_binaria(lista, objetivo): # Función de búsqueda binaria
    izquierda = 0 # Límite izquierdo
    derecha = len(lista) - 1 # Límite derecho
    # Mientras el rango sea válido
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2 # Calcula el punto medio
        # Si encuentra el objetivo
        if lista[medio] == objetivo:
            return medio # Devuelve la posición
        # Si el medio es menor que el objetivo
        elif lista[medio] < objetivo:
            izquierda = medio + 1 # Busca en la mitad derecha
        else:
            derecha = medio - 1 # Busca en la mitad izquierda
    return -1 # Devuelve -1 si no lo encuentra

# Datos de prueba
datos = [34, 7, 23, 32, 5, 62, 19, 45] # Lista de números para probar
objetivo = 34 # Número a buscar

# Resultados
print("Lista original:", datos) # Muestra la lista original

# Búsqueda lineal en la lista original
resultado_lineal = busqueda_lineal(datos, objetivo) # Busca el objetivo
con búsqueda lineal
if resultado_lineal != -1: # Si lo encuentra
    print("Búsqueda lineal: Encontrado en posición", resultado_lineal) #
Muestra resultado
else: # Si no lo encuentra
    print("Búsqueda lineal: número no encontrado dentro de la lista") #
Mensaje personalizado

# Ordenar con Bubble Sort
ordenada_bubble = bubble_sort(datos.copy()) # Ordena la lista con bubble
sort
print("Lista ordenada con Bubble Sort:", ordenada_bubble) # Muestra la
lista ordenada

# Ordenar con Quicksort
ordenada_quicksort = quicksort(datos.copy()) # Ordena la lista con
quicksort
print("Lista ordenada con Quicksort:", ordenada_quicksort) # Muestra la
lista ordenada

# Búsqueda binaria sobre ambas listas ordenadas

```

```

resultado_binaria_bubble = busqueda_binaria(ordenada_bubble, objetivo) #
Busca con binaria en bubble sort
if resultado_binaria_bubble != -1: # Si lo encuentra
    print("Búsqueda binaria (lista ordenada con Bubble):",
resultado_binaria_bubble) # Muestra resultado
else: # Si no lo encuentra
    print("Búsqueda binaria (lista ordenada con Bubble): número no
encontrado dentro de la lista") # Mensaje

resultado_binaria_quicksort = busqueda_binaria(ordenada_quicksort,
objetivo) # Busca con binaria en quicksort
if resultado_binaria_quicksort != -1: # Si lo encuentra
    print("Búsqueda binaria (lista ordenada con Quicksort):",
resultado_binaria_quicksort) # Muestra resultado
else: # Si no lo encuentra
    print("Búsqueda binaria (lista ordenada con Quicksort): número no
encontrado dentro de la lista") # Mensaje

```

Para este trabajo se decidió implementar dos algoritmos de búsqueda (lineal y binaria) y dos algoritmos de ordenamiento (Bubble Sort y Quicksort) con el objetivo de comparar sus funcionamientos y eficiencia.

Elección de los algoritmos de búsqueda fueron:

- **Búsqueda lineal:** Fue elegida por ser simple y fácil a la hora de implementarlo ya que no requiere de una lista ordenada y permite ver un enfoque más básico para encontrar un elemento dentro de la lista
- **Búsqueda binaria:** Se lo eligió por ser mucho mas eficiente en listas ordenadas, ya que reduce mucho la cantidad de comparaciones necesarias, pero para su correcto funcionamiento se necesita ordenar la lista previamente.

Elección de los algoritmos de ordenamiento:

- **Bubble Sort:** Fue elegido por ser un algoritmo simple y fácil de entender, ideal para mostrar el paso a paso de como funciona el ordenamiento mediante comparaciones e intercambios sucesivos, su defecto es que para listas grandes no es muy eficiente.
- **Quicksort:** Fue elegido por su alta eficiencia y su uso en la programación, permite comparar su rendimiento con el algoritmo anteriormente nombrado y ver como el mismo resultado se puede lograr con menos pasos

Metodología Utilizada

Para el desarrollo del trabajo se siguió una estructura de varias etapas, que permitió llegar a los aspectos teóricos como prácticos de los algoritmos de búsqueda y ordenamiento.

- **Investigación previa:**
 - Uso del material teórico de “Búsqueda y Ordenamiento” que fue utilizado para definiciones y comparativas de los diferentes métodos
 - Videos donde se explica el uso de los algoritmos de “Búsqueda y Ordenamiento”
 - Uso del contenido de los módulos vistos en la materia de “Programación 1” para comprender el uso correcto del lenguaje Python
- **Etapa de diseño y prueba:**
 - Se define una lista desordenada de prueba y un número objetivo a buscar.
 - Se implementaron manualmente los algoritmos seleccionados: Búsqueda lineal y binaria y Ordenamiento: Bubble Sort y Quicksort.
 - Se utilizó el lenguaje de Python para el desarrollo del código
 - Se observan los resultados obtenidos para validar el funcionamiento
 - Se evalúa la efectividad de cada método utilizado
- **Herramientas y recursos utilizados:**
 - Lenguaje de programación: Python 3.11.9
 - Software de desarrollo (IDE): Visual Studio Code (Versión: 1.102.0)
 - Sistema operativo: Windows 11
- **Trabajo:** El trabajo fue realizado de forma individual, aplicando los conocimientos adquiridos en clase, reforzando la comprensión mediante ejemplos prácticos y documentación técnica.

Resultados Obtenidos:

- **Resultado:**

```

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

● PS C:\Users\lucar> & C:/Users/lucar/AppData/Local/Microsoft/Windows
Lista original: [34, 7, 23, 32, 5, 62, 19, 45]
Búsqueda lineal: Encontrado en posición 3
Lista ordenada con Bubble Sort: [5, 7, 19, 23, 32, 34, 45, 62]
Lista ordenada con Quicksort: [5, 7, 19, 23, 32, 34, 45, 62]
Búsqueda binaria (lista ordenada con Bubble): 4
Búsqueda binaria (lista ordenada con Quicksort): 4
○ PS C:\Users\lucar>

```

- Búsqueda Lineal: Encontró el numero 32 en la posición 3 de la lista original.
- Bubble Sort: Ordeno correctamente la lista
- Quicksort: Ordeno correctamente la lista (en estructuras mas grandes es más eficiente)
- Búsqueda Binaria: Encontró el numero 32 en la posición 3 sobre ambas listas ya ordenadas

Los cuatro algoritmos funcionaron de forma correcta y coherente entre sí.

Errores corregidos durante la implementación inicial:

- Se identifico un error lógico de la búsqueda binaria que no calculaba correctamente el índice medio, se corrigió con la formula: $(\text{izquierda} + \text{derecha}) // 2$.
- Se revisaron los "return" de las funciones para asegurar que se devolviera el índice correcto o "-1" en caso de no encontrar el valor

Repositorio del trabajo: <https://github.com/LucaR0801/T.P-Integrador-Programacion-1.git>

Conclusiones

El desarrollo del trabajo permite el estudio y comprensión de los algoritmos de búsqueda y ordenamiento, estos son fundamentales dentro de la programación, a través de la implementación manual de los métodos vistos, se logra comprender como funcionan internamente estos algoritmos y como la elección de estos afecta en el rendimiento del programa.

Se aprendió que la eficiencia no siempre depende de un único tipo de algoritmo, sino también del contexto en el que se aplica: el tamaño de la lista, la necesidad del orden previo y la claridad del código influyen para el método mas adecuado para el problema dado.

El tema tiene gran utilidad en el desarrollo de software, ya que estos procesos aparecen en las aplicaciones como sistema de bases de datos, motores de búsqueda, visualización de datos, etc. Al elegir e implementar correctamente estos algoritmos mejorara el rendimiento y el crecimiento de las soluciones programadas.

Como futura mejora, se podría medir los tiempos de ejecución reales con la librería “time” para así comprar el rendimiento de cada algoritmo con diferentes tamaños de entrada.

La dificultad que surgió fue la implementación correcta del “Quicksort”, ya que, al tratarse de un método recurrente, fue necesario comprender como se divide la lista y como se combinan los subresultados, los resultados obtenidos no eran correctos ya que daban errores lógicos, esto se resolvió repasando la teoría y mirando videos explicativos.

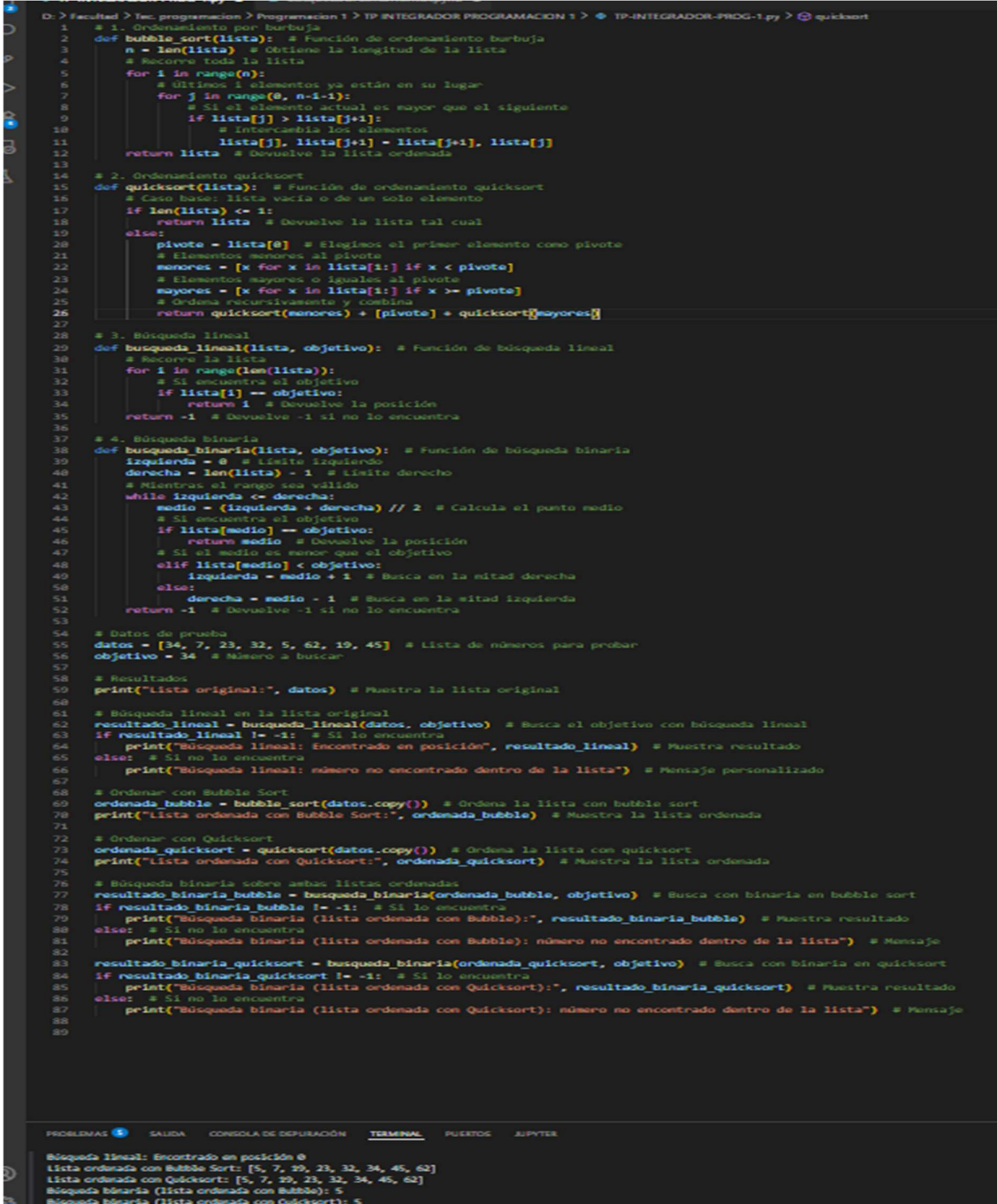
En resumen, el trabajo fue una experiencia valiosa tanto en el aspecto teórico como práctico. El conocimiento adquirido me permitirá comprender mejor el uso de estos algoritmos y aplicarlos de manera más eficiente en el desarrollo de programas más extensos y complejos.

Bibliografía

- Python Oficial: <https://docs.python.org/3/>
- UTN – Tecnicatura Universitaria en Programación: Material de la materia “Programacion 1”
- Material orientativo: (Integ. Búsqueda y ordenamiento)
- Video sobre Búsqueda: <https://www.youtube.com/watch?v=gJlQTq80llg>
- Video sobre Ordenamiento: <https://www.youtube.com/watch?v=xntUhrhtLaw>
- Material complementario: Búsqueda y Ordenamiento en Programación.pdf.
(Documento utilizado para el desarrollo teórico del trabajo)
- Notebook Búsqueda y ordenamiento:
<https://colab.research.google.com/drive/1KVqiJSzYLTPDFRwTYjN8CP7G4LPred9J?usp=sharing>

Anexos

- Captura de pantalla del programa funcionando:



```

D:\Facultad > Iac.programacion > Programacion 1 > TP INTEGRADOR PROGRAMACION 1 > TP-INTEGRADOR-PROG-1.py > quicksort
1 # 1. Ordenamiento por burbuja
2 def bubble_sort(lista): # Función de ordenamiento burbuja
3     n = len(lista) # Obtiene la longitud de la lista
4     # Recorre toda la lista
5     for i in range(n):
6         # Últimos i elementos ya están en su lugar
7         for j in range(0, n-i-1):
8             # Si el elemento actual es mayor que el siguiente
9             if lista[j] > lista[j+1]:
10                 # Intercambia los elementos
11                 lista[j], lista[j+1] = lista[j+1], lista[j]
12     return lista # Devuelve la lista ordenada
13
14 # 2. Ordenamiento quicksort
15 def quicksort(lista): # Función de ordenamiento quicksort
16     # Caso base: lista vacía o de un solo elemento
17     if len(lista) <= 1:
18         return lista # Devuelve la lista tal cual
19     else:
20         pivote = lista[0] # Elegimos el primer elemento como pivote
21         # Elementos menores al pivote
22         menores = [x for x in lista[1:] if x < pivote]
23         # Elementos mayores o iguales al pivote
24         mayores = [x for x in lista[1:] if x >= pivote]
25         # Ordena recursivamente y combina
26         return quicksort(menores) + [pivote] + quicksort(mayores)
27
28 # 3. Búsqueda lineal
29 def busqueda_lineal(lista, objetivo): # Función de búsqueda lineal
30     # Recorre la lista
31     for i in range(len(lista)):
32         # Si encuentra el objetivo
33         if lista[i] == objetivo:
34             return i # Devuelve la posición
35     return -1 # Devuelve -1 si no lo encuentra
36
37 # 4. Búsqueda binaria
38 def busqueda_binaria(lista, objetivo): # Función de búsqueda binaria
39     izquierda = 0 # Límite izquierdo
40     derecha = len(lista) - 1 # Límite derecho
41     # Mientras el rango sea válido
42     while izquierda <= derecha:
43         medio = (izquierda + derecha) // 2 # Calcula el punto medio
44         # Si encuentra el objetivo
45         if lista[medio] == objetivo:
46             return medio # Devuelve la posición
47         # Si el medio es menor que el objetivo
48         elif lista[medio] < objetivo:
49             izquierda = medio + 1 # Busca en la mitad derecha
50         else:
51             derecha = medio - 1 # Busca en la mitad izquierda
52     return -1 # Devuelve -1 si no lo encuentra
53
54 # Datos de prueba
55 datos = [34, 7, 23, 32, 5, 62, 19, 45] # Lista de números para probar
56 objetivo = 34 # Número a buscar
57
58 # Resultados
59 print("Lista original:", datos) # Muestra la lista original
60
61 # Búsqueda lineal en la lista original
62 resultado_lineal = busqueda_lineal(datos, objetivo) # Busca el objetivo con búsqueda lineal
63 if resultado_lineal != -1: # Si lo encuentra
64     print("Búsqueda lineal: Encontrado en posición", resultado_lineal) # Muestra resultado
65 else: # Si no lo encuentra
66     print("Búsqueda lineal: número no encontrado dentro de la lista") # Mensaje personalizado
67
68 # Ordenar con Bubble Sort
69 ordenada_bubble = bubble_sort(datos.copy()) # Ordena la lista con bubble sort
70 print("Lista ordenada con Bubble Sort:", ordenada_bubble) # Muestra la lista ordenada
71
72 # Ordenar con Quicksort
73 ordenada_quicksort = quicksort(datos.copy()) # Ordena la lista con quicksort
74 print("Lista ordenada con Quicksort:", ordenada_quicksort) # Muestra la lista ordenada
75
76 # Búsqueda binaria sobre ambas listas ordenadas
77 resultado_binaria_bubble = busqueda_binaria(ordenada_bubble, objetivo) # Busca con binaria en bubble sort
78 if resultado_binaria_bubble != -1: # Si lo encuentra
79     print("Búsqueda binaria (lista ordenada con Bubble):", resultado_binaria_bubble) # Muestra resultado
80 else: # Si no lo encuentra
81     print("Búsqueda binaria (lista ordenada con Bubble): número no encontrado dentro de la lista") # Mensaje
82
83 resultado_binaria_quicksort = busqueda_binaria(ordenada_quicksort, objetivo) # Busca con binaria en quicksort
84 if resultado_binaria_quicksort != -1: # Si lo encuentra
85     print("Búsqueda binaria (lista ordenada con Quicksort):", resultado_binaria_quicksort) # Muestra resultado
86 else: # Si no lo encuentra
87     print("Búsqueda binaria (lista ordenada con Quicksort): número no encontrado dentro de la lista") # Mensaje
88
89
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS JUPYTER
Búsqueda lineal: Encontrado en posición 0
Lista ordenada con Bubble Sort: [5, 7, 19, 23, 32, 34, 45, 62]
Lista ordenada con Quicksort: [5, 7, 19, 23, 32, 34, 45, 62]
Búsqueda binaria (lista ordenada con Bubble): 5
Búsqueda binaria (lista ordenada con Quicksort): 5

```

- Repositorio en GitHub: <https://github.com/LucaR0801/T.P-Integrador-Programaci-n-1.git>

- Link del video: <https://www.youtube.com/watch?v=QMeTZQ2bJ5I>