



Programación I

Trabajo Práctico Integrador

Tema: Algoritmos de Búsqueda y Ordenamiento

Alumnos:

Agustín Tejada

Franco Trabucci

Datos Generales

Alumnos:

Agustín Tejada – tejadaagustin60@gmail.com

Franco Trabucci - francotrabucci@gmail.com

Materia: Programación I

Profesor: Nicolás Quirós

Fecha de Entrega: 09/06/25

Índice

- Introducción	4
- Marco Teórico	5
- Caso Práctico	7
- Metodología Utilizada	10
- Resultados Obtenidos	11
- Conclusión	12
- Bibliografía	13
- Anexo	14

Introducción

El tema "Algoritmos de Búsqueda y Ordenamiento" fue seleccionado debido a su relevancia en la programación. Estos algoritmos representan herramientas esenciales para la gestión eficiente de datos, ya que permiten organizar la información y localizar elementos específicos dentro de un conjunto, optimizando el tiempo de procesamiento.

La importancia de los algoritmos de búsqueda y ordenamiento en programación radica en su aplicación a una amplia variedad de problemas y contextos, desde bases de datos y sistemas de archivos hasta inteligencia artificial y desarrollo web. La eficiencia en el ordenamiento y búsqueda de datos incide directamente en el rendimiento de los sistemas informáticos, haciendo que el dominio de estos algoritmos sea crucial para el rendimiento de los sistemas.

El presente trabajo tiene como objetivo explicar los tipos de algoritmos de búsqueda y ordenamiento, se explica de manera más detallada el "Algoritmo de Búsqueda Binaria" y el "Algoritmo de Ordenamiento Quicksort". También se explica el caso práctico de un programa que utiliza éstos algoritmos mencionados.

Marco Teórico

Algoritmos de Búsqueda

Un algoritmo de búsqueda es un conjunto de instrucciones que están diseñadas para localizar un elemento dentro de una estructura de datos.

Tipos de Algoritmos de Búsqueda (más comunes)

Búsqueda Binaria

Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos.

Búsqueda lineal

Es el algoritmo de búsqueda más simple, que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Es fácil de implementar, pero es lento para conjuntos de datos grandes.

Búsqueda 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.

Algoritmos de Ordenamiento

Instrucciones que organizan un conjunto de elementos siguiendo un criterio específico, como de menor a mayor o de mayor a menor. Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Tipos de Algoritmos de Ordenamiento (más comunes)

Ordenamiento de burbuja o bubble sort

Funciona comparando cada elemento de la lista con el siguiente elemento y luego intercambiando los elementos si están en el orden incorrecto. Es simple de entender, ideal para listas pequeñas pero ineficiente para grandes conjuntos de datos.

Quicksort

Es un algoritmo de ordenamiento basado en la estrategia de divide y vencerás. Consiste en elegir un elemento pivote dentro del conjunto de datos y reorganizar la lista de forma que todos los elementos menores al pivote queden a su izquierda y los mayores a su derecha. Luego, se repite el mismo proceso de forma recursiva con cada sublista.

Inserción

Construye la lista ordenada elemento por elemento, insertando cada nuevo elemento en la posición correcta. Es eficiente para listas pequeñas o parcialmente ordenadas.

Selección

Busca el elemento más pequeño de la lista y lo coloca al inicio. Repite el proceso con el resto de la lista, es decir, busca el siguiente elemento más pequeño en la parte no ordenada y lo intercambia con el elemento en la siguiente posición. Este procedimiento se repite hasta que todos los elementos estén de manera ordenada.

Caso Práctico

Se desarrolló un programa en Python que le solicita al usuario ingresar una lista de palabras separadas por comas. Posteriormente, se solicita al usuario una palabra para buscarla dentro de la lista. El programa ordena la lista alfabéticamente utilizando el algoritmo QuickSort y, una vez ordenada, realiza la búsqueda de la palabra ingresada empleando el algoritmo de búsqueda binaria. Si la palabra es encontrada, retorna el índice correspondiente; de lo contrario, devuelve -1 indicando que la palabra no se encuentra en la lista.

Programa de Python:

Función de ordenamiento usando Quicksort

```
def quick_sort(lista):
```

```
    # Si la lista es vacía o con un solo elemento retorna la lista
```

```
    if len(lista) <= 1:
```

```
        return lista
```

```
    else:
```

```
        # Selección del primer elemento como pivote
```

```
        pivote = lista[0]
```

```
        # Sublistas de elementos menores o iguales y mayores al pivote  
(ignorando mayúsculas/minúsculas)
```

```
        menores = [x for x in lista[1:] if x.lower() <= pivote.lower()]
```

```
        mayores = [x for x in lista[1:] if x.lower() > pivote.lower()]
```

```
        # Recursión para ordenar las sublistas
```

```
        return quick_sort(menores) + [pivote] + quick_sort(mayores)
```

Función de búsqueda usando Búsqueda Binaria

```
def busqueda_binaria(lista, objetivo):
```

```
    izquierda = 0
```

```
derecha = len(lista) - 1
```

```
# Mientras el rango sea válido
```

```
while izquierda <= derecha:
```

```
    medio = (izquierda + derecha) // 2
```

```
    # Comparaciones ignorando mayúsculas/minúsculas
```

```
    if lista[medio].lower() == objetivo.lower():
```

```
        return medio
```

```
    elif lista[medio].lower() < objetivo.lower():
```

```
        izquierda = medio + 1
```

```
    else:
```

```
        derecha = medio - 1
```

```
# Si no se encuentra el elemento
```

```
return -1
```

```
# Programa principal
```

```
# Solicita la lista de palabras al usuario
```

```
entrada = input("Ingresa una lista de palabras separadas por comas: ")
```

```
# Divide las palabras y elimina espacios extras
```

```
lista_usuario = [palabra.strip() for palabra in entrada.split(', ')]
```

```
# Solicita al usuario la palabra a buscar
```

```
palabra_a_buscar = input("Ingresa la palabra a buscar: ")
```



```
# Ordena la lista llamando a la función quick_sort

lista_ordenada = quick_sort(lista_usuario)

# Se muestra la lista ordenada

print("Lista ordenada:", lista_ordenada)


# Busca la palabra en la lista ordenada, llamando a la función
busqueda_binaria

indice = busqueda_binaria(lista_ordenada, palabra_a_buscar)


# Muestra el resultado

if indice != -1:

    print(f"La palabra '{palabra_a_buscar}' fue encontrada en la posición
    {indice}.")

else:

    print(f"La palabra '{palabra_a_buscar}' no está en la lista.")
```

Decisiones de diseño

QuickSort fue elegido como método de ordenamiento debido a su eficiencia, ideal para listas de tamaño mediano a grande. Además, su implementación es relativamente sencilla y adecuada para listas de cadenas de texto.

Búsqueda Binaria fue seleccionada por su eficiencia en listas ordenadas.

Ambas funciones fueron diseñadas para ignorar las diferencias entre mayúsculas y minúsculas utilizando `str.lower()`, mejorando la experiencia del usuario al no requerir coincidencias exactas de capitalización.

Metodología Utilizada

La elaboración del trabajo se realizó en las siguientes etapas:

- Selección del tema "Algoritmos de Búsqueda y Ordenamiento".
- Recolección e investigación de información sobre el tema.
- Creación de un programa en Python con el algoritmo de búsqueda binaria y el algoritmo de ordenamiento Quicksort.
- Pruebas del programa creado.
- Registro de resultados y validación de funcionalidad.
- Elaboración de este informe y preparación de anexos.
- Creación del video explicando el tema y la aplicación de los algoritmos en el programa creado.
- Publicación del programa en GitHub

Resultados Obtenidos

- Comprensión de algoritmos de Búsqueda y Ordenamiento.
- Creación de un programa en Python.
- Implementación de algoritmo Quicksort en un programa de Python.
- Implementación de algoritmo Búsqueda Binaria en un programa de Python.
- Prueba del programa
- El programa ordenó correctamente la lista de palabras.
- La búsqueda binaria localizó de forma eficiente la palabra buscada.

Conclusión

A lo largo del desarrollo del presente trabajo integrador, logramos aplicar de manera práctica los conceptos fundamentales relacionados con los algoritmos de búsqueda y ordenamiento. Aprendimos que la elección del algoritmo adecuado depende fuertemente del contexto, como la cantidad de datos y el entorno específico en el que se aplica.

Este proyecto no solo permitió afianzar y aplicar conocimientos teóricos, sino también desarrollar habilidades blandas, trabajando en equipo y compartiendo distintos puntos de vista con nuestro compañero.

En conclusión, sentimos que alcanzamos los objetivos que nos propone el Trabajo Práctico Integrador y pudimos aplicar los conocimientos que estamos adquiriendo a lo largo de este cuatrimestre pudiendo continuar avanzando en la formación como programadores.

Bibliografía

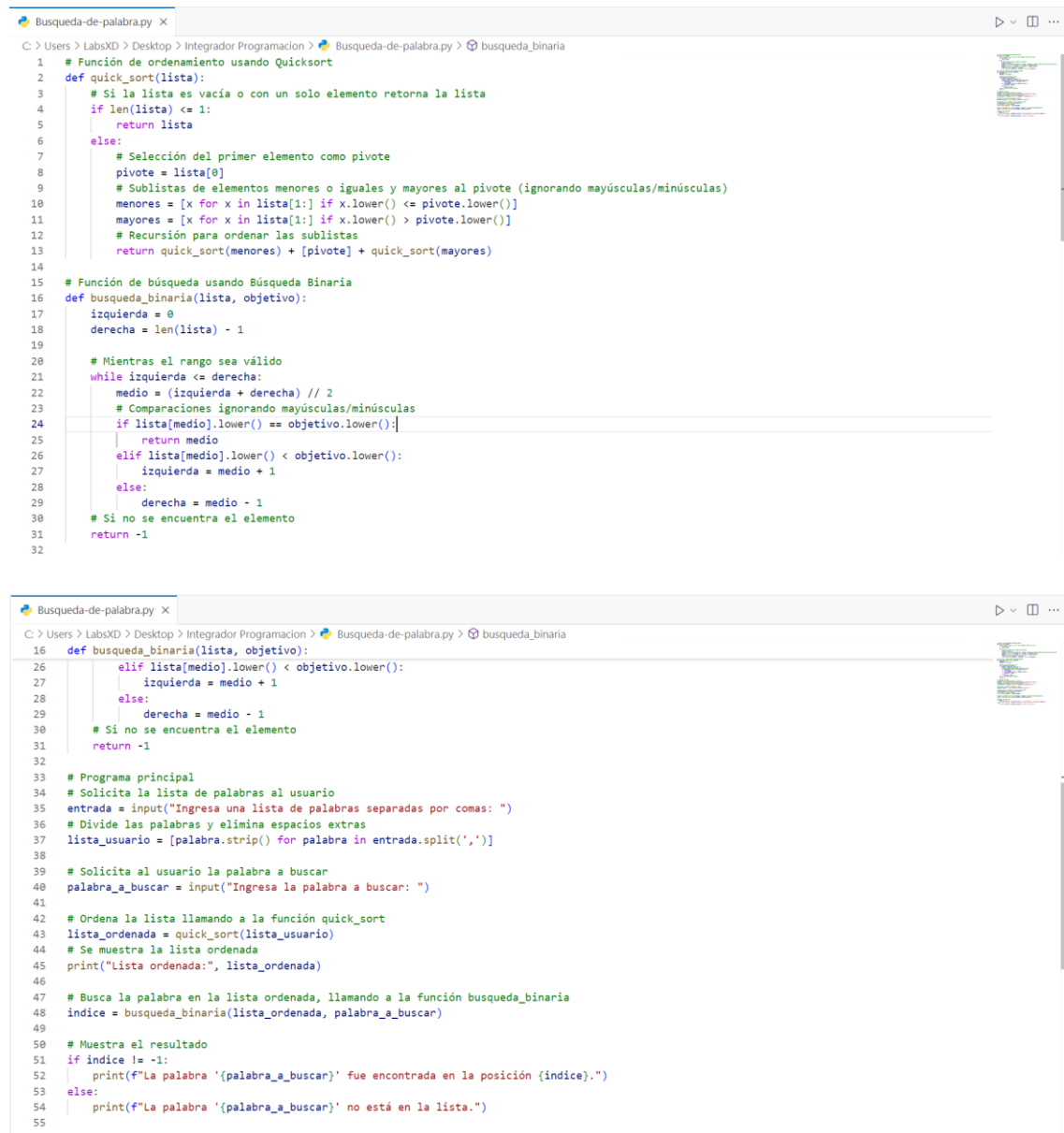
- <https://www.fing.edu.uy/tecnoinf/mvd/cursos/prinprog/material/teo/prinprog-teorico11.pdf>
- <https://www.freecodecamp.org/espanol/news/algoritmos-de-ordenacion-explicados-con-ejemplos-en-javascript-python-java-y-c/>
- https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda
- Material teórico publicado en el campus

Anexo

Link del vídeo: <https://www.youtube.com/watch?v=O8lgmd1YyTw>

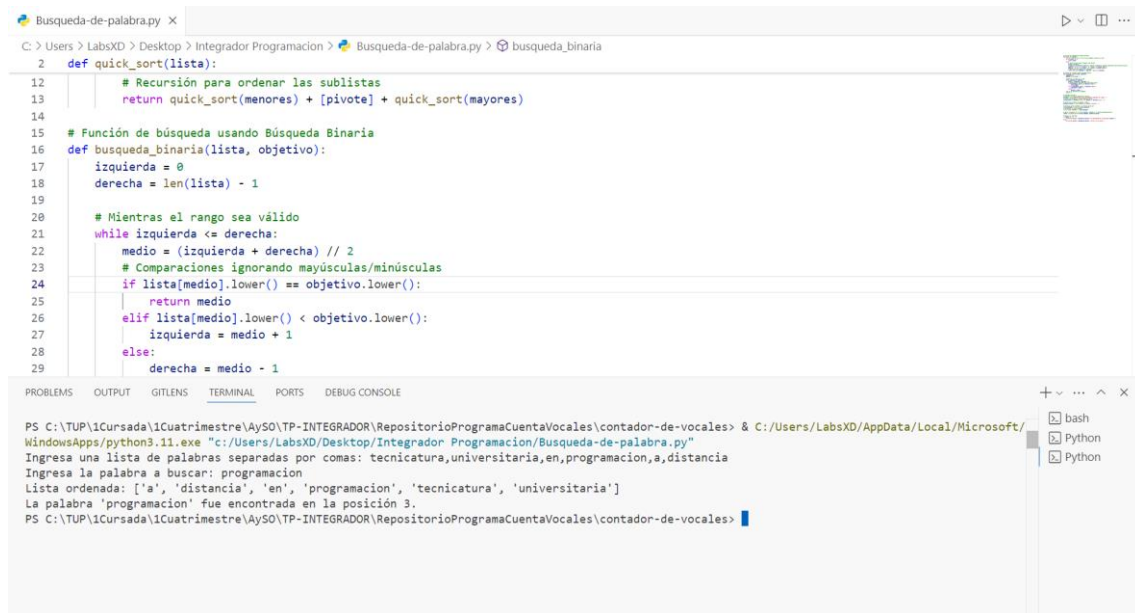
Link del repositorio: <https://github.com/AgustinTejada/proyecto-integrador-programacion-i>

Capturas de pantalla del programa:



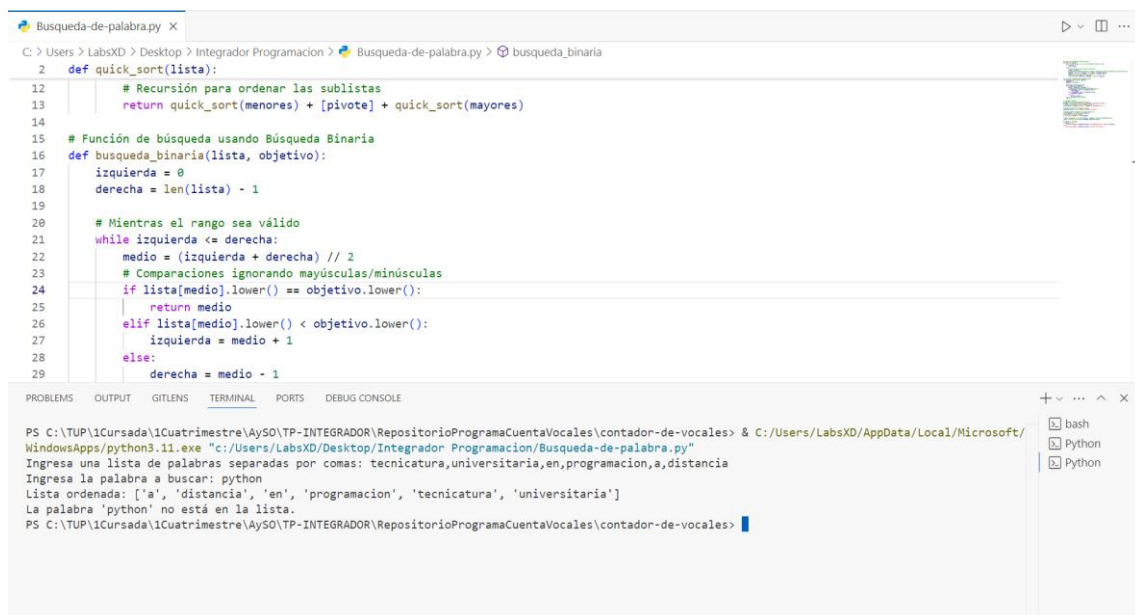
```
1 # Función de ordenamiento usando Quicksort
2 def quick_sort(lista):
3     # Si la lista es vacía o con un solo elemento retorna la lista
4     if len(lista) <= 1:
5         return lista
6     else:
7         # Selección del primer elemento como pivote
8         pivote = lista[0]
9         # Sublistas de elementos menores o iguales y mayores al pivote (ignorando mayúsculas/minúsculas)
10        menores = [x for x in lista[1:] if x.lower() <= pivote.lower()]
11        mayores = [x for x in lista[1:] if x.lower() > pivote.lower()]
12        # Recursión para ordenar las sublistas
13        return quick_sort(menores) + [pivote] + quick_sort(mayores)
14
15 # Función de búsqueda usando Búsqueda Binaria
16 def busqueda_binaria(lista, objetivo):
17     izquierda = 0
18     derecha = len(lista) - 1
19
20     # Mientras el rango sea válido
21     while izquierda <= derecha:
22         medio = (izquierda + derecha) // 2
23         # Comparaciones ignorando mayúsculas/minúsculas
24         if lista[medio].lower() == objetivo.lower():
25             return medio
26         elif lista[medio].lower() < objetivo.lower():
27             izquierda = medio + 1
28         else:
29             derecha = medio - 1
30     # Si no se encuentra el elemento
31     return -1
32
33 # Programa principal
34 # Solicita la lista de palabras al usuario
35 entrada = input("Ingresa una lista de palabras separadas por comas: ")
36 # Divide las palabras y elimina espacios extras
37 lista_usuario = [palabra.strip() for palabra in entrada.split(',')]
38
39 # Solicita al usuario la palabra a buscar
40 palabra_a_buscar = input("Ingresa la palabra a buscar: ")
41
42 # Ordena la lista llamando a la función quick_sort
43 lista_ordenada = quick_sort(lista_usuario)
44 # Se muestra la lista ordenada
45 print("Lista ordenada:", lista_ordenada)
46
47 # Busca la palabra en la lista ordenada, llamando a la función busqueda_binaria
48 indice = busqueda_binaria(lista_ordenada, palabra_a_buscar)
49
50 # Muestra el resultado
51 if indice != -1:
52     print(f"La palabra '{palabra_a_buscar}' fue encontrada en la posición {indice}.")
53 else:
54     print(f"La palabra '{palabra_a_buscar}' no está en la lista.")
55
```

Capturas de pantalla de pruebas del programa:



The screenshot shows a code editor with a Python file named `Busqueda-de-palabra.py`. The code implements a recursive function `quick_sort` and a binary search function `busqueda_binaria`. The terminal window shows the following output:

```
PS C:\TUP\1Cursada\1Cuatrimestre\AySO\TP-INTEGRADOR\RepositorioProgramaCuentaVocales\contador-de-vocales> & C:/Users/LabsXD/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/LabsXD/Desktop/Integrador Programacion/Busqueda-de-palabra.py"
Ingresa una lista de palabras separadas por comas: tecnicatura,universitaria,en,programacion,a,distancia
Ingresa la palabra a buscar: programacion
Lista ordenada: ['a', 'distancia', 'en', 'programacion', 'tecnicatura', 'universitaria']
La palabra 'programacion' fue encontrada en la posición 3.
PS C:\TUP\1Cursada\1Cuatrimestre\AySO\TP-INTEGRADOR\RepositorioProgramaCuentaVocales\contador-de-vocales>
```



The screenshot shows the same code editor with the Python file `Busqueda-de-palabra.py`. The terminal window shows the following output:

```
PS C:\TUP\1Cursada\1Cuatrimestre\AySO\TP-INTEGRADOR\RepositorioProgramaCuentaVocales\contador-de-vocales> & C:/Users/LabsXD/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/LabsXD/Desktop/Integrador Programacion/Busqueda-de-palabra.py"
Ingresa una lista de palabras separadas por comas: tecnicatura,universitaria,en,programacion,a,distancia
Ingresa la palabra a buscar: python
Lista ordenada: ['a', 'distancia', 'en', 'programacion', 'tecnicatura', 'universitaria']
La palabra 'python' no está en la lista.
PS C:\TUP\1Cursada\1Cuatrimestre\AySO\TP-INTEGRADOR\RepositorioProgramaCuentaVocales\contador-de-vocales>
```