Trabajo Práctico Integrador – Programación I

Integrantes:

Sanabria, Betina

betinasanabria2023@gmail.com

Rodríguez, Santiago David

Santir9515@gmail.com

Comisión: M2025-20

Tecnicatura Universitaria en Programación

Materia: Programación I

Docente: Cinthia Rigoni

Fecha: 9 de junio de 2025

1. Introducción

El presente trabajo se centra en el desarrollo de un motor de búsqueda de palabras utilizando el lenguaje de programación Python.

La elección de este tema responde a la necesidad de aplicar de forma práctica los conocimientos adquiridos a lo largo del semestre, especialmente en lo referente a estructura de datos, algoritmo de búsqueda y ordenamiento, y desarrollo de funciones. Además, se trata de un problema con aplicaciones concretas en el ámbito del análisis de textos, la gestión de información y la implementación de herramientas automatizadas.

La importancia de este tema en la programación radica en su aplicabilidad en múltiples áreas: desde los motores de búsqueda en internet hasta los sistemas de recomendación, pasando por el análisis de grandes volúmenes de datos.

La capacidad de recorrer, clasificar y buscar información de manera eficiente es una competencia fundamental en el desarrollo de software y en la resolución de problemas reales.

Asimismo, el trabajo permite integrar conceptos clave como el manejo de listas, el uso de diccionarios, la implementación de funciones y el ordenamiento mediante algoritmos, lo que lo convierte en un excelente ejercicio de síntesis.

El objetivo principal de este trabajo es diseñar e implementar un sistema capaz de procesar un archivo de texto, analizarlo y permitir al usuario buscar palabras y obtener estadísticas básicas sobre su frecuencia de aparición. Para ello, se utilizarán herramientas propias de la programación en Python, aplicando principios de eficiencia, claridad y reutilización de código.

A través de este desarrollo, se busca consolidar los contenidos abordados durante el semestre y demostrar su aplicación práctica en un problema realista.

2. Marco Teórico

El desarrollo de un motor de búsqueda de palabras requiere la comprensión e implementación de varios conceptos fundamentales en el campo de la programación, especialmente vinculados a estructuras de datos y algoritmos. En este apartado se presentan las definiciones, clasificaciones y sintaxis asociadas a los principales elementos utilizados, con ejemplo prácticos en Python.

Estructura de datos

Las estructuras de datos son formas organizadas de almacenar y gestionar información para facilitar su acceso y modificación. En este trabajo se utilizan principalmente listas, diccionarios y tuplas

A. Listas

Son colecciones ordenadas y mutables de elementos. Se definen con corchetes” []”, y permiten recorrer, agregar, eliminar y modificar elementos

```
palabras = ["hola", "mundo", "hola"]
```

Se utilizan para almacenar las palabras extraídas de un texto

B. Diccionarios

Estructuras clave-valor que permiten acceder rápidamente a los datos.

```python
frecuencias = {"hola": 2, "mundo": 1}
```

En este trabajo, los diccionarios se usan para contar cuantas veces aparece cada palabra

C. Tuplas

Son estructuras inmutables, útiles para representar pares de datos. Se utilizan, por ejemplo, para ordenar las palabras por frecuencia:

```python
tupla = ("hola", 2)
```

Algoritmos de búsqueda

Un algoritmo de búsqueda permite localizar un valor especifico dentro de una estructura. Los mas relevantes para este trabajo son:

A. Búsqueda Lineal

Consiste en recorrer todos los elementos hasta encontrar el valor deseado. Es simple pero ineficiente para grandes volúmenes:

```python
def buscar_lineal(lista, objetivo):
    for i in lista:
        if i == objetivo:
            return True
    return False
```

B. Búsqueda binaria

Requiere que la lista esté ordenada previamente. Divide el conjunto en mitades sucesivas:

```python
def buscar_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return True
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return False
```

Algoritmos de Ordenamiento

El ordenamiento de datos permite organizar la información para facilitar búsquedas o visualización. En este trabajo se puede implementar:

A. Ordenamiento por frecuencia

Utilizando la función *sorted()* de Python con una función lambda:

```
ordenado = sorted(frecuencias.items(), key=lambda x: x[1], reverse=True)
```

B. Ordenamiento alfabético

Usando la clave por defecto (la palabra):

```
alfabetico = sorted(frecuencias.items())
```

Recursividad

La recursividad es una técnica en la que una función se llama a si misma para resolver un problema más pequeño. Se puede aplicar en algoritmos de ordenamiento como el Quicksort, aunque en este trabajo se priorizó el uso de funciones internas de Python para simplificar la implementación

Python

Python es un lenguaje interpretado, dinámico y con una sintaxis clara que facilita la implementación de este tipo de soluciones. Su biblioteca estándar incluye herramientas como *collections.Counter,* aunque en este trabajo se optó por una implementación manual para comprender mejor la lógica subyacente

Referencias utilizadas

- Python Documentation: https://docs.python.org/3/
- Cormen, T. H., et al. (2009). Introduction to Algorithms.
- Downey, A. (2015). Think Python.

3. Caso practico

Descripción del problema

El problema a resolver consiste en crear un motor de búsqueda de palabras que analice un archivo de texto y devuelva información útil sobre las palabras contenidas en él. El sistema debe:

- Leer un archivo .txt
- Contar cuantas veces aparece cada palabra
- Permitir buscar palabras especificas
- Mostrar el ranking de las palabras mas usadas
- Ordenar alfabéticamente el vocabulario del texto

Decisiones de diseño

- Diccionario (*dict):* se usó para contar palabras por su eficiencia en búsquedas
- Ordenamiento: se utilizó la función *sorted()* de Python con funciones *lambda,* por su claridad y eficiencia. Se evito usar algoritmos como bubble sort por su bajo rendimiento en textos grandes
- Limpieza de texto: Se eliminaron signos comunes para asegurar un conteo limpio.
- Modularización: cada función tiene una tarea especifica para facilitar la lectura, reutilización y pruebas

Validación del funcionamiento

Para validar el motor de búsqueda se utilizó un archivo de prueba llamado texto_prueba.txt con el siguiente contenido:

“Aunque parecía imposible, Python finalmente ha conseguido superar a Java, y la inteligencia artificial podría ser la causante. C# también se quedan atrás en cuanto a popularidad en 2025.

No cabe duda de que uno de los lenguajes más utilizados de todos los tiempos ha sido Java, siendo el que ha liderado la industria durante varios años, pero Python también se ha mantenido como una de las opciones más relevantes, versátiles y eficientes.

Con la integración de la inteligencia artificial para generar código con texto, como lo permite ChatGPT o Claude, la industria está cambiando drásticamente. Si bien significa que hay una probabilidad de que muchos programadores se queden sin trabajo, también abre las puertas a una nueva era llena de oportunidades y mayor productividad.

La implementación de estas herramientas en el mencionado lenguaje de programación probablemente haya sido una de las razones por la que ha aumentado su popularidad durante este 2025, al igual que su facilidad de uso y los numerosos proyectos para los que sirve.

Lo impresionante de esto es que desde 2001 no se había visto una participación tan grande en la industria y Python ha logrado un hito que lleva más de un par de décadas sin ser alcanzado.”

A continuacion mostramos la salida del procesamiento

```
PS C:\Users\santi> & C:/Users/santi/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/santi/TP-Integrador.py
Ruta del archivo de texto (.txt): texto_ejemplo.txt

Se procesaron 204 palabras.

Que palabra querés buscar?: Python
La palabra 'Python' aparece 3 veces.

Top 10 palabras mas frecuentes:
de: 14
la: 9
que: 9
ha: 6
una: 5
y: 5
en: 4
los: 4
se: 4
a: 3

Primeras 10 palabras ordenadas alfabeticamente:
2001: 1
2025: 2
a: 3
abre: 1
al: 1
alcanzado: 1
artificial: 2
atrás: 1
aumentado: 1
aunque: 1
```

4. Pasos seguidos durante el desarrollo del trabajo

Durante el desarrollo del presente trabajo integrador se siguieron varias etapas, desde la investigación conceptual hasta la validación del código implementado. A continuación, se detallan los pasos más relevantes:

A. Investigación previa

Se inició con una investigación teórica sobre los algoritmos de búsqueda, estructuras de datos asociadas (como diccionarios y listas) y técnicas de ordenamiento. Se consultaron fuentes como la documentación oficial de Python (docs.python.org), recursos educativos como W3Schools, y apuntes del curso. También se revisaron ejemplos de análisis de texto y motores de búsqueda simples para comprender mejor la lógica del problema.

B. Etapas de diseño y prueba del código

Diseño del problema: Se definió el objetivo del programa: desarrollar un motor de búsqueda capaz de leer texto, contar palabras, buscar términos específicos y ordenarlos por frecuencia o alfabéticamente.

Desarrollo modular: Se implementaron funciones separadas para cada parte del proceso: carga del texto, conteo de frecuencias, búsqueda de palabras y ordenamiento.

Pruebas unitarias: Se probaron las funciones individualmente usando textos cortos.

Prueba completa: Se utilizó un archivo de prueba real con más de 200 palabras para validar el funcionamiento general del programa.

C. Herramientas y recursos utilizados

Lenguaje de programación: Python 3.12

Entorno de desarrollo (IDE): Visual Studio Code y terminal de Visual Studio Code

Librerías: No se utilizaron librerías externas, solo funciones integradas de Python.

Editor de texto: Bloc de notas y VS Code para crear el archivo de entrada .txt

D. Trabajo colaborativo

El trabajo fue realizado de forma grupal, el planteamiento del tema a desarrollar fue elegido por ambos, teníamos varias ideas para realizar, por ejemplo, sistema de gestión de inventario, sistema de atención prioritaria en emergencias. Nos decidimos por el motor de búsqueda, ya que puede ayudar a identificar en un texto presentado si hay repeticiones de palabras, lo cual genera una informalidad en un trabajo a presentar.

5. Resultados obtenidos

A. Errores:

En este ejemplo el procesamiento no nos ordenada de manera correcta las palabras.

```
PS C:\Users\santi> & C:/Users/santi/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/santi/TP-Integrador.py
Ruta del archivo de texto (.txt): texto_ejemplo.txt

Se procesaron 204 palabras.

Que palabra querés buscar?Python
La palabra 'Python' aparece 3 veces.

Top 10 palabras mas frecuentes:
y: 5
visto: 1
versátiles: 1
varios: 1
utilizados: 1
uso: 1
uno: 1
una: 5
un: 2
trabajo: 1
```

Se soluciono implementando en la función x:(-x[1], x[0])) Esto ayuda a ordenar de mayor a menor la frecuencia de las palabras y además si hay empate en la repetición las ordena alfabéticamente para tener una mejor visibilidad

```
def ordenar_por_frecuencia(frecuencias):
    return sorted(frecuencias.items(), key=lambda x: (-x[1], x[0]))
```

También tuvimos el siguiente error en la misma línea de código porque dejamos la expresión reverse = True, lo cual nos invertía la sentencia anterior y daba el siguiente proceso.

```
Ruta del archivo de texto (.txt): texto_ejemplo.txt

Se procesaron 204 palabras.

Que palabra querés buscar?Python
La palabra 'Python' aparece 3 veces.

Top 10 palabras mas frecuentes:
visto: 1
versátiles: 1
varios: 1
utilizados: 1
uso: 1
uno: 1
trabajo: 1
todos: 1
tiempos: 1
texto: 1

Primeras 10 palabras ordenadas alfabeticamente:
2001: 1
2025: 2
a: 3
abre: 1
al: 1
alcanzado: 1
artificial: 2
atrás: 1
aumentado: 1
```

Otro de los errores iniciales fue un FileNotFoundError, produjo al no ingresar correctamente la ruta del archivo de texto. Se resolvió verificando la correcta ruta del archivo usando la función pwd en la terminal.

También se corrigió un detalle menor con el uso de mayúsculas, ya que el sistema no identificaba mayúsculas y minúsculas. Lo solucionamos aplicando la función .lower() para estandarizar todo el texto a minúscula.

Por último, eliminamos los signos de puntuación, ya que afectaban el conteo correcto de palabras. Para esto se uso la librería string para limpiar correctamente el texto antes del análisis.

B. Evaluación de rendimiento:

Dado que se utilizó un único algoritmo de búsqueda y ordenamiento en estructuras simples como diccionarios y listas, no se hizo una comparación directa de rendimiento entre distintos métodos. Sin embargo, se comprobó que el programa responde rápidamente incluso con archivos de más de 200 palabras, lo que demuestra un rendimiento adecuado para el objetivo del trabajo.

6. **Conclusiones**

El desarrollo de este trabajo integrador permitió profundizar los conocimientos sobre algoritmos de búsqueda, estructura de datos y análisis de texto, aplicados de manera concreta mediante el lenguaje Python. A través del diseño e implementación de un motor de búsqueda simple, se logro aplicar conceptos clave como el uso de diccionarios para el conteo de palabras, ordenamiento por frecuencia, y manipulación de cadenas de texto.

Una de las principales enseñanzas del trabajo fue comprender como elegir la estructura de datos adecuada impacta directamente en la eficiencia y claridad del código. Además, se reforzó la importancia de dividir el problema en funciones cortas y específicas, lo que favorece el mantenimiento y pruebas del programa.

El tema trabajado tiene una gran utilidad práctica tanto en programación como en otros proyectos relacionados con el análisis de datos, IA, motores de búsqueda y desarrollo de herramientas de texto automatizadas. Incluso este modelo básico, ya puede ser aplicado en contextos reales tales como análisis de documentos, clasificación de contenido y filtros de texto.

Como comentamos, entre las dificultades surgidas, se destacan problemas con la lectura del archivo de entrada, manejo de mayúsculas y signos de puntuación. Estos problemas fueron resueltos mediante ajustes en el código, aplicando funciones de estandarización y limpieza de texto.

Como posibles mejoras, se podría incorporar un filtro por tipo de palabra, posibilidad de reemplazar la palabra con algún sinónimo utilizando una IA o implementarlo en una web, para así facilitar el uso.

7. Bibliografía

- Python Software Foundation. (2024). *Python 3 Documentation*. https://docs.python.org/3/
- StackOverflow https://es.stackoverflow.com/
- Material de apoyo provisto por la catedra

8. Anexo

Por último, dejamos el link a github para descargar el repositorio mas una captura del código completo comentado
Link: https://github.com/Santir9515/trabajo-colaborativo/tree/main/Trabajo-Integrador
Captura:

```python
def cargar_texto(ruta_archivo):
    #Lee el archivo y devuelve una lista de palabras en minúsculas, sin signos de puntuación
    with open(ruta_archivo, "r", encoding="utf-8") as archivo:
        texto = archivo.read().lower()

    palabras = texto.split()
    #Aca limpiamos los signos de puntuacion para evitar errores
    palabras_limpias = [palabra.strip(".,:;()¿?!¡\"'") for palabra in palabras]
    return palabras_limpias

#Vamos a contar la frecuencia de cada palabra usando un diccionario
def contar_frecuencias(lista_palabras):
    frecuencias = {}
    for palabra in lista_palabras:
        #evita palabras vacias
        if palabra:
            if palabra in frecuencias:
                frecuencias[palabra] += 1
            else:
                frecuencias[palabra] = 1
    return frecuencias

#La siguiente función devuelve la frecuencia de una palabra, Si no existe, retorna 0.
def buscar_palabra(frecuencias, palabra):
    return frecuencias.get(palabra.lower(), 0)

#Esta funcion va a devolver una lista de tuplas (palabra, frecuencia) ordenadas de mayor a menor
def ordenar_por_frecuencia(frecuencias):
    #Esto ayuda a ordenar de mayor a menor la frecuencia de las palabras y ademas si hay empate las ordena alfabeticamente
    return sorted(frecuencias.items(), key=lambda x: (-x[1], x[0]))

#Esta funcion nos va a ordenar las tuplas alfabeticamente por palabra
def ordenar_alfabeticamente(frecuencias):
    return sorted(frecuencias.items(), key=lambda x: x[0])

if __name__ == "__main__":
    archivo = input("Ruta del archivo de texto (.txt): ")
    palabras = cargar_texto(archivo)
    print(f"\nSe procesaron {len(palabras)} palabras.\n")

    frecuencias = contar_frecuencias(palabras)

    palabra_buscada = input("Que palabra querés buscar?: ")
    resultado = buscar_palabra(frecuencias, palabra_buscada)
    print(f"La palabra '{palabra_buscada}' aparece {resultado} veces. \n")

    print("Top 10 palabras mas frecuentes:")
    top10 = ordenar_por_frecuencia(frecuencias)[:10]
    for palabra, cantidad in top10:
        print(f"{palabra}: {cantidad}")

    print("\nPrimeras 10 palabras ordenadas alfabeticamente:")
    orden_alfabetico = ordenar_alfabeticamente(frecuencias)[:10]
    for palabra, cantidad in orden_alfabetico:
        print(f"{palabra}: {cantidad}")
```