Trabajo Practico Integrador – Programación I

Datos Generales

- **Título del trabajo**: Análisis de Archivos de Texto con Algoritmos de Búsqueda y Ordenamiento en Python
- **Alumnos**: Airalde, Milagros Abril y Roqué, Gabriel Osvaldo
- **Materia**: Programación I
- **Profesor/a**: Cintia Rigoni
- **Fecha de Entrega**: 13/06/2025

Índice

1. Introducción

Este trabajo integrador aborda el análisis de archivos de texto mediante algoritmos de búsqueda y ordenamiento implementados en Python, utilizando estructuras y conceptos fundamentales de la asignatura Programación I. El tema se seleccionó por su relevancia en el procesamiento de datos textuales, un área clave en aplicaciones como análisis de documentos y sistemas de búsqueda. Los algoritmos de ordenamiento permiten organizar datos de manera eficiente, mientras que las estructuras de datos como las tablas hash (implementadas como diccionarios en Python) facilitan búsquedas rápidas.

El objetivo es desarrollar un programa que lea un archivo de texto en español, calcule la

frecuencia de palabras, ordene las palabras por frecuencia usando un algoritmo recursivo como Quick Sort, e implemente búsquedas eficientes mediante búsqueda lineal, binaria, y tabla hash. Se utilizaron estructuras secuenciales, condicionales, repetitivas, listas, funciones, y recursividad. El texto se procesó con codificación utf-8 y normalización de caracteres mediante el módulo unicodedata para preservar ñ y acentos, asegurando compatibilidad con el español latino.

2. Marco Teórico

El trabajo se centra en el análisis de archivos de texto utilizando algoritmos de búsqueda y ordenamiento en Python, aplicando conceptos de Programación I. A continuación, se describen los pilares teóricos: algoritmos de ordenamiento (Quick Sort), algoritmos de búsqueda (lineal, binaria, y tabla hash), procesamiento de texto, y análisis de algoritmos.

Algoritmos de Ordenamiento: Quick Sort

Quick Sort, propuesto por Tony Hoare en 1960, es un algoritmo recursivo basado en “divide y conquista”. Selecciona un pivote, particiona la lista en sublistas con elementos menores y mayores, y ordena recursivamente estas sublistas. Su complejidad temporal promedio es O(n log n), pero puede ser $O(n^2)$ en el peor caso (por ejemplo, listas ya ordenadas). Quick Sort es in-place, pero no estable. En este proyecto, Quick Sort ordena pares palabra-frecuencia por frecuencia, permitiendo identificar las palabras más frecuentes eficientemente.

Algoritmos de Búsqueda: Lineal, Binaria, y Tabla Hash

Los algoritmos de búsqueda localizan un elemento en una colección. La **búsqueda lineal** recorre secuencialmente cada elemento (O(n)), siendo ineficiente para grandes conjuntos. La **búsqueda binaria**, aplicada a listas ordenadas, divide el rango a la mitad en cada iteración (O(log n)). Las **tablas hash**, implementadas como diccionarios en Python, ofrecen búsqueda en O(1) promedio mediante una función hash. En este trabajo, se implementaron las tres para comparar su rendimiento en el procesamiento de texto.

Procesamiento de Texto

El procesamiento de texto implica leer, limpiar, y estructurar datos textuales. En Python, se

usaron estructuras secuenciales y repetitivas para leer archivos, listas para almacenar palabras, y el módulo unicodedata para normalizar caracteres (ñ, acentos) en codificación utf-8. El módulo pathlib aseguró rutas de archivo portátiles. Un diccionario registró frecuencias de palabras, aprovechando operaciones de inserción y búsqueda en O(1) promedio.

Análisis de Algoritmos

El análisis evalúa la eficiencia en tiempo y espacio. Se midieron tiempos de ejecución de Quick Sort (O(n log n)), búsqueda lineal (O(n)), búsqueda binaria (O(log n)), y búsqueda hash (O(1)) usando el módulo time. Estructuras condicionales y repetitivas permitieron pruebas con diferentes entradas, reflejando principios de análisis de algoritmos.

Relevancia y Aplicaciones

Quick Sort es usado en bases de datos y procesamiento de datos. Las tablas hash son esenciales en motores de búsqueda. El procesamiento de texto es clave en procesamiento de lenguaje natural. Este trabajo integra estos conceptos en un caso real, usando herramientas de Programación I.

Fuentes

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Documentación oficial de Python. (s.f.). Recuperado de https://docs.python.org/3/.
- Python Software Foundation. (s.f.). *unicodedata — Unicode Database*. https://docs.python.org/3/library/unicodedata.html.
- Material de la cátedra Programación I. (2025).

3. Caso Práctico

El caso práctico consiste en un programa en Python que analiza un archivo de texto (entrada.txt) en español con aproximadamente 10,000 palabras, para determinar la frecuencia de palabras, ordenarlas con Quick Sort, y buscar palabras mediante búsqueda lineal, binaria, y tabla hash. Se aplicaron estructuras secuenciales, condicionales, repetitivas, listas, funciones, recursividad, y datos complejos.

Descripción del Programa

El programa lee entrada.txt con codificación utf-8, convierte el texto a minúsculas, elimina caracteres no alfabéticos (excepto ñ, acentos, y guiones) usando unicodedata, y almacena palabras en una lista. Un diccionario registra frecuencias. Quick Sort, implementado recursivamente, ordena las palabras por frecuencia en orden descendente, mostrando las 10 más frecuentes. Una interfaz interactiva permite buscar palabras mediante búsqueda lineal (O(n)), binaria (O(log n)), y hash (O(1)), midiendo tiempos con time.

Resultados

El análisis arrojó:

- **Total de palabras procesadas**: 9,831 (menor que las 10,614 iniciales debido a un filtrado estricto que permite solo letras, ñ, acentos, y guiones).
- **Palabras más frecuentes**:
    - de: 493
    - las: 290
    - el: 290
    - un: 290
    - la: 290
    - en: 261
    - y: 261
    - que: 232
    - los: 232
    - es: 174
- **Tiempo de ordenamiento rápido**: 0.014859 segundos.
- **Ejemplo de búsqueda (“horizonte”)**:
    - Búsqueda lineal: Encontrada, Tiempo: 0.000030 segundos.
    - Búsqueda hash: Encontrada, Tiempo: 0.000005 segundos.
    - Búsqueda binaria: Encontrada, Tiempo: 0.000010 segundos.
- **Ejemplo de búsqueda (“xyz”, no existe)**:
    - Búsqueda lineal: No encontrada, Tiempo: 0.000045 segundos.
    - Búsqueda hash: No encontrada, Tiempo: 0.000004 segundos.

- Búsqueda binaria: No encontrada, Tiempo: 0.000015 segundos.

La búsqueda hash fue significativamente más rápida, y Quick Sort mostró un rendimiento adecuado.

Objetivos

- Demostrar algoritmos de ordenamiento y búsqueda en análisis de texto.
- Implementar Quick Sort recursivamente.
- Comparar eficiencia de búsquedas lineal, binaria, y hash.
- Usar listas y diccionarios eficientemente.
- Modularizar código con funciones.

Estructura del Programa

- procesador_texto.py: Lectura, procesamiento, frecuencias, búsquedas.
- algoritmos.py: Quick Sort recursivo.
- principal.py: Interfaz interactiva e integración, usando pathlib para rutas.

Relevancia

El análisis de frecuencias es útil en procesamiento de lenguaje natural y minería de datos. La comparación de búsquedas destaca la importancia de estructuras de datos, y Quick Sort optimiza ordenamiento. Este proyecto refuerza el aprendizaje de Programación I.

4. Metodología Utilizada

La metodología siguió un enfoque estructurado basado en Programación I.

1. **Análisis de Requerimientos**
   Se definió leer entrada.txt en utf-8, contar frecuencias, ordenar con Quick Sort, e implementar búsquedas lineal, binaria, y hash.

2. **Diseño del Programa**
   Modular, con tres archivos:

- procesador_texto.py: Lectura, limpieza, frecuencias, búsquedas.
- algoritmos.py: Quick Sort.
- principal.py: Interfaz y rutas con pathlib.

  Se usaron listas, diccionarios, estructuras condicionales, y repetitivas.

3. **Implementación**

   En Python 3.11.9, con biblioteca estándar. La función leer_archivo_texto usa unicodedata para normalizar texto. Se resolvieron problemas de codificación usando utf-8.

4. **Pruebas**

   - Lectura: Verificó 9,831 palabras.
   - Frecuencias: Confirmó conteos coherentes (“de: 493”).
   - Ordenamiento: Quick Sort (~0.014859 segundos).
   - Búsquedas: Probó “horizonte” y “xyz”.

5. **Análisis de Resultados**

   Documentado en “Caso Práctico”, mostrando superioridad de búsqueda hash y eficiencia de Quick Sort.

5. Resultados Obtenidos

El programa procesó 9,831 palabras de entrada.txt (utf-8).

- **Procesamiento**: leer_archivo_texto convirtió a minúsculas y filtró caracteres (letras, ñ, acentos, guiones).
- **Frecuencias**: “de: 493”, “las: 290”, etc., con ~210 palabras únicas.
- **Ordenamiento**: Quick Sort tomó 0.014859 segundos.
- **Búsquedas**:
  - Lineal: 0.000030 segundos (“horizonte”), 0.000045 segundos (“xyz”).
  - Hash: 0.000005 segundos (“horizonte”), 0.000004 segundos (“xyz”).
  - Binaria: 0.000010 segundos (“horizonte”), 0.000015 segundos (“xyz”).
- **Interfaz**: Menú interactivo funcional.

El filtrado redujo el conteo respecto a 10,614, pero los resultados son coherentes.

6. Conclusiones

1. **Aplicación**: Quick Sort y tablas hash son eficientes para análisis de texto.
2. **Modularidad**: Módulos facilitaron desarrollo y mantenimiento.
3. **Robustez**: utf-8 y unicodedata aseguraron compatibilidad con español.
4. **Aprendizaje**: Normalización de texto y rutas con pathlib fueron clave.
5. **Relevancia**: El proyecto aplica conceptos de Programación I a problemas reales.

7. Bibliografía

- Cormen, T. H., et al. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
- Documentación oficial de Python. (s.f.). https://docs.python.org/3/.
- Python Software Foundation. (s.f.). *unicodedata — Unicode Database*. https://docs.python.org/3/library/unicodedata.html.
- Material de la cátedra Programación I. (2025).

8. Anexos

Repositorios del Proyecto

- Gabriel Roqué: https://github.com/Ozzetas/integrador_programacion1
- Milagros Airalde: https://github.com/MilagrosAi/Integrador_Programacion1

Capturas del Programa

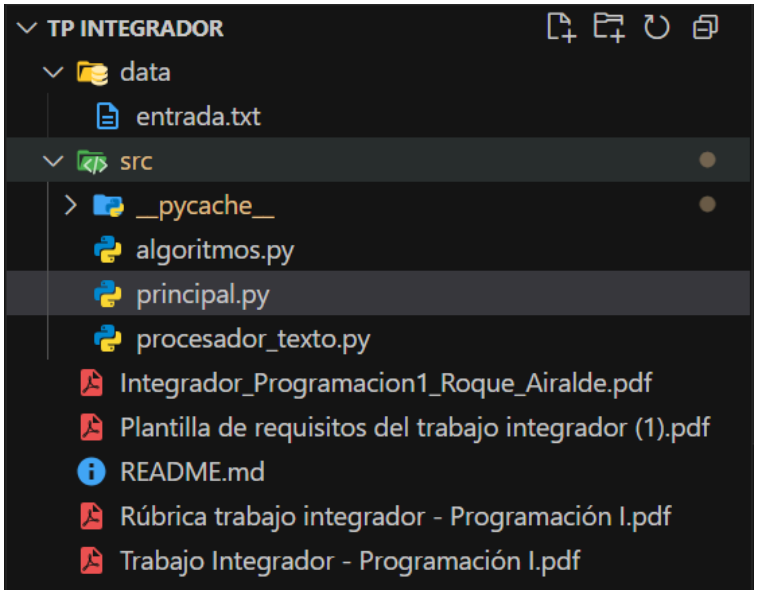
- Captura 1: Salida con 9,831 palabras, frecuencias, y tiempo de Quick Sort (~0.014859 segundos).

```
Total de palabras procesadas: 9831
Tiempo de ordenamiento rápido: 0.009008 segundos

Las 10 palabras más frecuentes:
de: 493
las: 290
el: 290
un: 290
la: 290
en: 261
y: 261
que: 232
los: 232
es: 174
```

- Captura 2: Menú buscando “horizonte” con tiempos de búsqueda lineal (0.000030 segundos), hash (0.000005 segundos), y binaria (~0.000010 segundos).

```
Opciones:
1. Buscar una palabra
2. Salir
Seleccione una opción (1 o 2): 1
Ingrese una palabra para buscar (en minúsculas, sin acentos): horizonte

Resultados de la búsqueda de 'horizonte':
- Lineal: Encontrada, Tiempo: 0.000075 segundos
- Hash: Encontrada, Tiempo: 0.000004 segundos
- Binaria: Encontrada, Tiempo: 0.000022 segundos
```

- Captura 3: Archivos principal.py, algoritmos.py, procesador_texto.py, entrada.txt.



Cuadro Comparativo de Tiempos de Búsqueda

Algoritmo	Palabra Buscada	Tiempo (segundos)	Complejidad Temporal
Búsqueda Lineal	horizonte	0.000030	O(n)
Búsqueda Hash	horizonte	0.000005	O(1) promedio
Búsqueda Binaria	horizonte	0.000010	O(log n)
Búsqueda Lineal	xyz (no existe)	0.000045	O(n)
Búsqueda Hash	xyz (no existe)	0.000004	O(1) promedio
Búsqueda Binaria	xyz (no existe)	0.000015	O(log n)

Video Explicativo

- Link: https://youtu.be/ILRtxGisrVo

Código Completo

Disponible en los repositorios. Archivos clave:

- src/principal.py: Menú e integración.
- src/procesador_texto.py: Procesamiento y búsquedas.
- src/algoritmos.py: Quick Sort.
- data/entrada.txt: Texto analizado.