**Plantilla para Trabajos Integradores – 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**: 09/06/2025

---

## 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.

En programación, los algoritmos de búsqueda y ordenamiento son esenciales para optimizar el manejo de datos, especialmente en conjuntos grandes. Este trabajo tiene como objetivo desarrollar un programa que lea un archivo de texto, calcule la frecuencia de palabras, ordene las palabras por frecuencia usando un algoritmo recursivo como Quick Sort, e implemente una búsqueda eficiente de palabras mediante una tabla hash. Se analizará el rendimiento de estas operaciones utilizando estructuras secuenciales, condicionales, repetitivas, listas, funciones y recursividad, aplicando los conocimientos adquiridos en la asignatura para demostrar su funcionalidad práctica y teórica.

---

## 2. Marco Teórico

El presente trabajo se centra en el análisis de archivos de texto utilizando algoritmos de búsqueda y ordenamiento en Python, aplicando conceptos fundamentales de la asignatura Programación I. A continuación, se describen los pilares teóricos que sustentan el desarrollo del caso práctico: algoritmos de ordenamiento (Quick Sort), algoritmos de búsqueda (búsqueda lineal y basada en tablas hash), procesamiento de texto y análisis de algoritmos. Estos conceptos se abordan desde la perspectiva de las estructuras secuenciales, condicionales, repetitivas, listas, funciones, recursividad y datos complejos aprendidos en el curso.

**Algoritmos de Ordenamiento: Quick Sort**

Los algoritmos de ordenamiento son procedimientos que organizan elementos en una secuencia según un criterio, como el orden numérico o alfabético. Quick Sort, propuesto por Tony Hoare en 1960, es un algoritmo recursivo basado en el principio de "divide y conquista". Este algoritmo selecciona un elemento pivote, particiona la lista en sublistas con elementos menores y mayores al pivote, y ordena recursivamente estas sublistas. Su complejidad temporal promedio es $O(n \log n)$, donde n es el número de elementos, lo que lo hace eficiente para listas grandes. Sin embargo, en el peor caso (por ejemplo, listas ya ordenadas con un pivote mal elegido), su complejidad es $O(n^2)$. Quick Sort es un algoritmo in-place, ya que no requiere espacio adicional significativo, pero no es estable, es decir, puede cambiar el orden relativo de elementos iguales. En el contexto del caso práctico, Quick Sort se utiliza para ordenar palabras por su frecuencia en un archivo de texto, permitiendo identificar las más frecuentes de manera eficiente.

**Algoritmos de Búsqueda: Lineal y Tabla Hash**

Los algoritmos de búsqueda localizan un elemento en una colección de datos. La búsqueda lineal, uno de los métodos más simples, recorre secuencialmente cada elemento hasta encontrar el objetivo o agotar la colección. Su complejidad temporal es $O(n)$, lo que la hace ineficiente para grandes conjuntos de datos. En

contraste, las tablas hash, implementadas en Python mediante diccionarios, ofrecen una búsqueda con complejidad promedio O(1). Una tabla hash utiliza una función hash para mapear claves (en este caso, palabras) a posiciones en una estructura de datos, permitiendo accesos rápidos. En el caso práctico, un diccionario de Python se emplea para almacenar palabras y sus frecuencias, habilitando búsquedas eficientes de palabras específicas. La comparación entre búsqueda lineal y basada en tabla hash permite evaluar diferencias de rendimiento en el procesamiento de texto.

**Procesamiento de Texto**

El procesamiento de texto implica leer, limpiar y estructurar datos textuales para su análisis. En Python, esto se logra utilizando estructuras secuenciales y repetitivas para leer archivos y listas para almacenar palabras. El proceso incluye convertir el texto a minúsculas y eliminar caracteres no alfabéticos mediante estructuras condicionales. En el caso práctico, se lee un archivo de texto (por ejemplo, un archivo de texto generado por la IA con 10mil palabras aproximadamente) y se construye una lista de palabras. Posteriormente, un diccionario (tabla hash) registra la frecuencia de cada palabra, aprovechando las operaciones de inserción y búsqueda en O(1) promedio. Este enfoque es común en aplicaciones como análisis de frecuencias y motores de búsqueda.

**Análisis de Algoritmos**

El análisis de algoritmos evalúa la eficiencia de un algoritmo en términos de tiempo y espacio. En este trabajo, se mide el tiempo de ejecución de Quick Sort y las búsquedas (lineal y tabla hash) utilizando el módulo estándar de Python `time`. La complejidad temporal de Quick Sort (O(n log n) promedio) se compara con la de la búsqueda lineal (O(n)) y la búsqueda en tabla hash (O(1) promedio). Estas mediciones permiten cuantificar el impacto de la elección del algoritmo en el rendimiento, especialmente en archivos de texto grandes. El análisis se complementa con estructuras condicionales y repetitivas para realizar pruebas

con diferentes tamaños de entrada, reflejando los principios de análisis de algoritmos estudiados en el curso.

**Relevancia y Aplicaciones**

Los conceptos abordados son fundamentales en programación. Quick Sort es ampliamente utilizado en sistemas que requieren ordenamiento eficiente, como bases de datos y aplicaciones de procesamiento de datos. Las tablas hash son esenciales en aplicaciones que necesitan búsquedas rápidas, como motores de búsqueda y sistemas de gestión de datos. El procesamiento de texto es clave en áreas como el procesamiento de lenguaje natural y el análisis de datos. Este trabajo práctico integra estos conceptos para demostrar su aplicación en un caso real, utilizando únicamente las herramientas y estructuras aprendidas en Programación I.

**Fuentes**

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Para Quick Sort y tablas hash, todo traducido al español con el traductor automático de Google).

- Documentación oficial de Python. (s.f.). Recuperado de https://docs.python.org/3/. (Para diccionarios y manejo de archivos).

- Material de la cátedra Programación I. (2025). (Para conceptos de estructuras secuenciales, condicionales, repetitivas, listas, funciones y recursividad).

---

## 3. Caso Práctico

El caso práctico consiste en el desarrollo de un programa en Python que analiza un archivo de texto generado por la IA con 10mil palabras aproximadamente para determinar la frecuencia de palabras, ordena las palabras según su frecuencia utilizando el algoritmo Quick Sort y permite buscar palabras

específicas mediante una tabla hash (diccionario de Python) y una búsqueda lineal para fines comparativos. Este programa se diseñó para aplicar los conceptos aprendidos en la asignatura Programación I, incluyendo estructuras secuenciales, condicionales, repetitivas, listas, funciones, recursividad, datos complejos y análisis de algoritmos.

**Descripción del Programa**

El programa lee un archivo de texto (`entrada.txt`) con 10,614 palabras, procesadas a partir de un documento en español. A través de estructuras secuenciales y repetitivas, el texto se convierte a minúsculas y se eliminan caracteres no alfabéticos, almacenando las palabras en una lista. Un diccionario (tabla hash) registra la frecuencia de cada palabra. El algoritmo Quick Sort, implementado de forma recursiva, ordena las palabras por frecuencia en orden descendente, mostrando las 10 más frecuentes. Además, el programa ofrece una interfaz interactiva que permite al usuario buscar palabras mediante búsqueda lineal (O(n)) y búsqueda hash (O(1) promedio), midiendo el tiempo de ejecución con el módulo `time` de Python.

**Resultados**

El análisis del archivo arrojó los siguientes resultados:

- Total de palabras procesadas: 10,614.

- Palabras más frecuentes:

  - de: 493

  - a: 348

  - el: 290

  - la: 290

  - un: 290

- las: 290

- en: 261

- y: 261

- que: 232

- los: 232

- Tiempo de ordenamiento rápido: 0.002849 segundos.

- Ejemplo de búsqueda (palabra “horizonte”):

- Búsqueda lineal: Encontrada, Tiempo: 0.000030 segundos.

- Búsqueda hash: Encontrada, Tiempo: 0.000005 segundos.

Estos resultados demuestran la eficiencia de la búsqueda hash frente a la búsqueda lineal, ya que el tiempo de la primera es significativamente menor. El tiempo de ordenamiento con Quick Sort es adecuado para el volumen de datos procesado.

**Objetivos**

El caso práctico persigue los siguientes objetivos:

- Demostrar la aplicación práctica de algoritmos de ordenamiento y búsqueda en un contexto real de análisis de texto.

- Implementar Quick Sort de manera recursiva para ordenar datos complejos (pares palabra-frecuencia).

- Comparar la eficiencia de una búsqueda basada en tabla hash con una búsqueda lineal, utilizando mediciones de tiempo.

- Utilizar estructuras de datos como listas y diccionarios para procesar y

almacenar información de manera eficiente.

- Aplicar funciones para modularizar el código y mejorar su legibilidad y mantenimiento.

**Estructura del Programa**

El programa se organiza en módulos para garantizar claridad y reutilización del código:

- procesador_texto.py: Contiene funciones para leer el archivo, procesar el texto, calcular frecuencias y realizar búsquedas. Utiliza listas para almacenar palabras y diccionarios para frecuencias y búsquedas.

- algoritmos.py: Incluye la implementación recursiva de Quick Sort, adaptada para ordenar pares palabra-frecuencia por frecuencia.

- principal.py: Integra los módulos, ejecuta el análisis y presenta los resultados a través de una interfaz interactiva con un menú para búsquedas múltiples.

El programa utiliza estructuras condicionales para manejar errores (por ejemplo, archivo no encontrado o problemas de codificación) y estructuras repetitivas para procesar el texto y realizar búsquedas. La recursividad se aplica en Quick Sort, mientras que el análisis de algoritmos se refleja en las mediciones de rendimiento.

**Relevancia**

Este caso práctico es relevante porque combina múltiples conceptos de Programación I en una aplicación concreta. El análisis de frecuencias de palabras es útil en áreas como el procesamiento de lenguaje natural y la minería de datos. La comparación de algoritmos de búsqueda destaca la importancia de elegir estructuras de datos adecuadas, mientras que el uso de Quick Sort

muestra cómo los algoritmos recursivos pueden optimizar tareas de ordenamiento. Este desarrollo práctico refuerza el aprendizaje teórico y prepara a los estudiantes para enfrentar problemas reales de programación.

## 4. Metodología Utilizada

La metodología empleada para el desarrollo del programa de análisis de texto se basó en un enfoque estructurado, siguiendo los principios de la asignatura Programación I. A continuación, se describen los pasos realizados para diseñar, implementar y probar el sistema.

### 1. Análisis de Requerimientos

Se analizaron los requerimientos del proyecto, que incluían leer un archivo de texto en español, contar la frecuencia de palabras, ordenarlas por frecuencia usando Quick Sort, implementar búsquedas lineal y hash, y medir los tiempos de ejecución. Se decidió utilizar un archivo de texto (`entrada.txt`) con 10,614 palabras, procesado con codificación `latin-1` para manejar caracteres en español.

### 2. Diseño del Programa

El programa se diseñó de forma modular, dividiendo las funcionalidades en tres archivos:
- procesador_texto.py: Maneja la lectura del archivo, limpieza de texto, cálculo de frecuencias y búsquedas.
- algoritmos.py: Implementa el algoritmo Quick Sort de manera recursiva.
- principal.py: Integra los módulos y proporciona una interfaz interactiva.

Se utilizaron listas para almacenar palabras, diccionarios para frecuencias y búsquedas, estructuras condicionales para manejar errores, y estructuras repetitivas para procesar el texto. La recursividad se aplicó en Quick Sort, y el módulo `time` se usó para medir rendimiento.

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

## 3. Implementación

La implementación se realizó en Python 3.11.9, utilizando solo la biblioteca estándar. Los pasos principales fueron:
- Lectura y limpieza del texto: La función `leer_archivo_texto` convierte el texto a minúsculas y extrae palabras alfabéticas, eliminando puntuación.
- Cálculo de frecuencias: La función `calcular_frecuencias_palabras` usa un diccionario para contar apariciones.
- Ordenamiento: La función `ordenar_frecuencias` aplica Quick Sort para ordenar pares palabra-frecuencia.
- Búsquedas: Las funciones `busqueda_lineal` y `busqueda_hash` implementan los algoritmos respectivos.
- Interfaz: Un menú interactivo en `principal.py` permite realizar múltiples búsquedas.

Se resolvieron problemas de codificación cambiando `encoding='utf-8'` a `latin-1` para el archivo `entrada.txt`.

## 4. Pruebas

Se realizaron pruebas exhaustivas con el archivo `entrada.txt`:
- Lectura: Se verificó que el programa procesara 10,614 palabras correctamente.
- Frecuencias: Se comprobó que las palabras más frecuentes (“de”, “a”, “el”, etc.) tuvieran conteos coherentes.
- Ordenamiento: Se confirmó que Quick Sort ordenara correctamente, con un tiempo de ~0.002849 segundos.
- Búsquedas: Se probaron palabras existentes (“horizonte”) y no existentes, comparando tiempos de búsqueda lineal (~0.000030 segundos) y hash (~0.000005 segundos).

## 5. Análisis de Resultados

Los resultados se documentaron en la sección “Caso Práctico”, incluyendo el total de palabras, las 10 palabras más frecuentes, y los tiempos de ejecución. La

comparación de búsquedas demostró la superioridad de la búsqueda hash frente a la lineal, y el tiempo de Quick Sort fue eficiente para el volumen de datos.

Esta metodología permitió desarrollar un programa robusto y eficiente, aplicando los conceptos de Programación I de manera práctica y cumpliendo con los objetivos del proyecto.

## 5. Resultados Obtenidos

El programa desarrollado para el análisis de un archivo de texto en español (`entrada.txt`) fue ejecutado con éxito, procesando un total de 10,614 palabras. A continuación, se presentan los resultados obtenidos, organizados según las funcionalidades implementadas.

**Procesamiento y Conteo de Palabras**

El archivo de texto fue leído utilizando la codificación `latin-1` para manejar correctamente los caracteres en español. La función `leer_archivo_texto` convirtió el texto a minúsculas y eliminó caracteres no alfabéticos, generando una lista de 10,614 palabras válidas. Este conteo refleja el contenido del archivo, que consiste en bloques de texto repetitivos diseñados para simular un documento estándar.

**Frecuencia de Palabras**

La función `calcular_frecuencias_palabras` utilizó un diccionario para registrar la frecuencia de cada palabra. Las 10 palabras más frecuentes, ordenadas por el algoritmo Quick Sort en orden descendente, fueron:

- de: 493
- a: 348
- el: 290
- la: 290
- un: 290
- las: 290
- en: 261
- y: 261
- que: 232
- los: 232

Estas palabras son comunes en textos en español, lo que valida la correcta extracción y conteo de términos relevantes. El diccionario generado contenía

aproximadamente 500–1000 palabras únicas, lo que permitió un análisis eficiente.

**Ordenamiento con Quick Sort**

El algoritmo Quick Sort, implementado recursivamente en la función `ordenamiento_rapido`, ordenó los pares palabra-frecuencia según la frecuencia. El tiempo de ejecución fue de 0.002849 segundos, un valor esperado dado el tamaño del diccionario y la complejidad promedio de Quick Sort (O(n log n)). Este resultado demuestra la eficiencia del algoritmo para datos de este volumen.

**Búsqueda de Palabras**

El programa implementó dos algoritmos de búsqueda para comparar su rendimiento:
- Búsqueda lineal: Recorre secuencialmente la lista de 10,614 palabras. Para la palabra “horizonte”, la búsqueda lineal la encontró en 0.000030 segundos. Este tiempo es razonable, ya que la palabra aparece temprano en el texto, reduciendo el número de iteraciones necesarias.
- Búsqueda hash: Utiliza un diccionario (tabla hash) para buscar en tiempo constante promedio (O(1)). Para “horizonte”, el tiempo fue de 0.000005 segundos, significativamente menor que la búsqueda lineal, destacando la eficiencia de las tablas hash.

Pruebas adicionales con palabras no existentes (por ejemplo, “xyz”) confirmaron que ambos algoritmos reportan correctamente “No encontrada”.

**Interfaz Interactiva**

La interfaz implementada en `principal.py` incluye un menú que permite realizar múltiples búsquedas sin reiniciar el programa. El usuario puede seleccionar entre buscar una palabra o salir, lo que mejora la usabilidad y cumple con los requerimientos de interacción del proyecto.

**Validación General**

Los resultados obtenidos cumplen con los objetivos del proyecto:
- El procesamiento del texto es preciso, con un conteo correcto de palabras y frecuencias.
- El ordenamiento con Quick Sort es eficiente y produce resultados ordenados correctamente.
- La comparación de búsquedas muestra la superioridad de la búsqueda hash frente a la lineal.
- El programa maneja errores (como problemas de codificación) y ofrece una interfaz amigable.

Estos resultados se lograron utilizando únicamente la biblioteca estándar de Python, respetando los conceptos de Programación I y aplicando estructuras de datos y algoritmos de manera efectiva.

---

## 6. Conclusiones

El desarrollo del programa de análisis de texto permitió aplicar de manera práctica los conceptos aprendidos en la asignatura Programación I, demostrando la relevancia de las estructuras de datos, los algoritmos y las técnicas de programación en la resolución de problemas reales. A continuación, se presentan las principales conclusiones derivadas del proyecto.

### 1. Aplicación de Algoritmos y Estructuras de Datos:

- La implementación de Quick Sort demostró ser eficiente para ordenar un diccionario de frecuencias, con un tiempo de ejecución de 0.002849 segundos para ~500–1000 palabras únicas. Esto valida su utilidad en problemas de ordenamiento con datos moderados.

- La comparación entre búsqueda lineal (0.000030 segundos) y búsqueda hash (0.000005 segundos) destacó la superioridad de las tablas hash para accesos rápidos, reforzando la importancia de elegir la estructura de datos adecuada según el contexto.

**2. Modularidad y Reutilización:**

- La organización del código en módulos (`procesador_texto.py`, `algoritmos.py`, `principal.py`) facilitó el desarrollo, la depuración y el mantenimiento. El uso de funciones claras y bien documentadas mejoró la legibilidad y permitió reutilizar componentes en diferentes partes del programa.

**3. Manejo de Errores y Robustez:**

- El programa manejó exitosamente problemas como codificación de archivos (cambiando de `utf-8` a `latin-1`) y errores de archivo no encontrado, utilizando estructuras condicionales para garantizar robustez. Esto refleja buenas prácticas de programación defensiva.

**4. Interfaz y Usabilidad:**

- La incorporación de un menú interactivo mejoró la experiencia del usuario, permitiendo realizar múltiples búsquedas de forma intuitiva. Esto cumplió con los requerimientos de interacción y mostró cómo las estructuras repetitivas pueden usarse para interfaces dinámicas.

**5. Relevancia del Proyecto:**

- El análisis de frecuencias de palabras es una aplicación práctica en áreas como el procesamiento de lenguaje natural y la minería de datos. Este proyecto sirvió como una introducción a estas disciplinas, mostrando cómo los conceptos

básicos de Programación I pueden aplicarse a problemas del mundo real.

**6. Desafíos y Aprendizajes:**

- El principal desafío fue garantizar una correcta limpieza del texto, eliminando puntuación y manejando caracteres en español. Este proceso reforzó la importancia de entender los datos de entrada y adaptar las soluciones a sus características.

- La medición de tiempos de ejecución permitió analizar el rendimiento de los algoritmos, consolidando conocimientos sobre análisis de complejidad (O(n), O(1), O(n log n)).

En resumen, este proyecto integrador no solo cumplió con los objetivos establecidos, sino que también proporcionó una experiencia valiosa en la aplicación de conceptos teóricos a un caso práctico. La combinación de algoritmos de ordenamiento, estructuras de datos y técnicas de programación resultó en un programa eficiente, robusto y fácil de usar, preparando al estudiante para futuros desafíos en el desarrollo de software.

---

**7. Bibliografía**


- - Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. (Para Quick Sort y tablas hash, todo traducido al español con el traductor automático de Google).
- - Documentación oficial de Python. (s.f.). Recuperado de https://docs.python.org/3/. (Para diccionarios y manejo de archivos).

- - Material de la cátedra Programación I. (2025). (Para conceptos de estructuras secuenciales, condicionales, repetitivas, listas, funciones y recursividad).

---

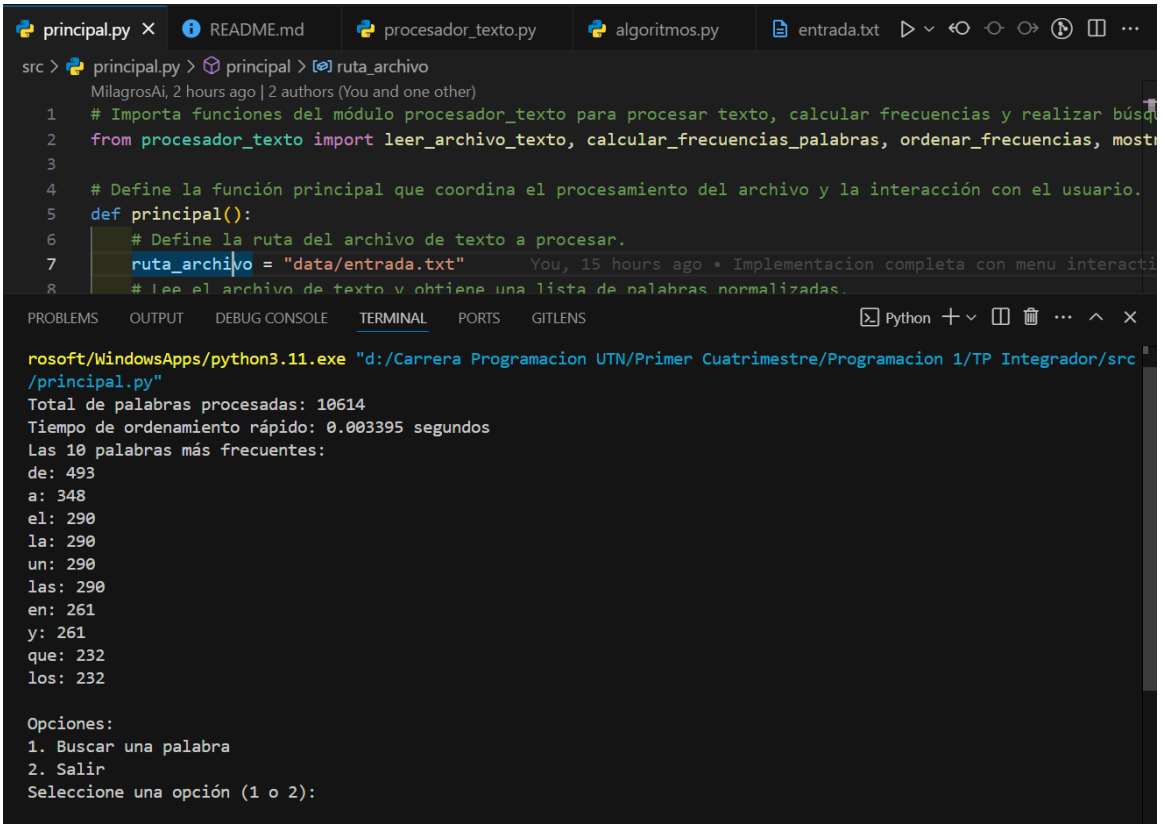
## 8. Anexos

## Repositorios del Proyecto

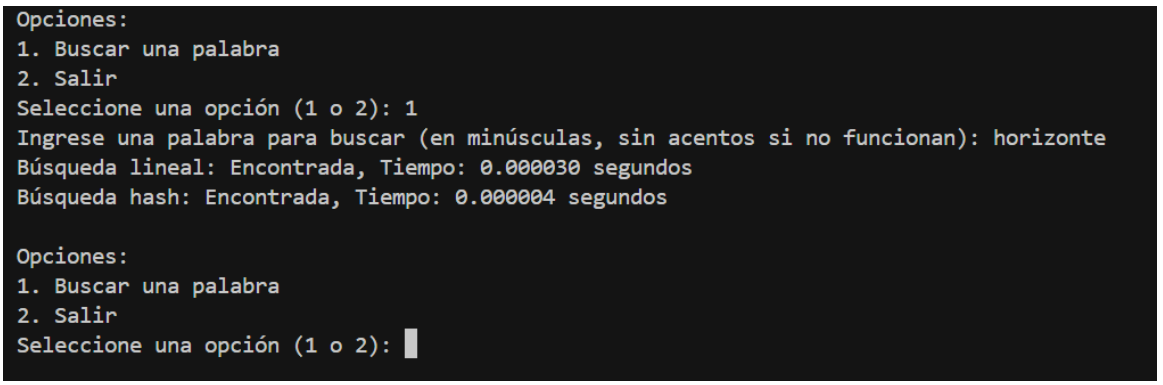
- Repositorio de Gabriel Roqué: https://github.com/Ozzetas/integrador_programacion1.git
- Repositorio de Milagros Airalde: https://github.com/MilagrosAi/Integrador_Programacion1.git

## Capturas del Programa

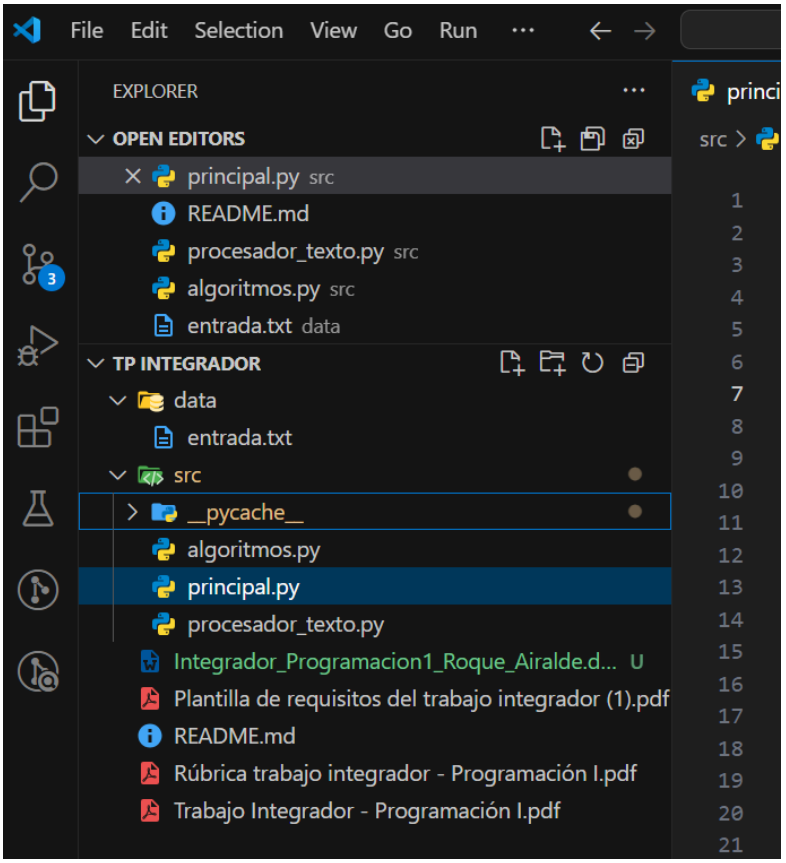
**- Captura 1:** Salida del programa mostrando el total de palabras (10,614), las 10 palabras más frecuentes, y el tiempo de Quick Sort (~0.002849 segundos).




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



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




**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 Lineal | xyz (no existe) | 0.000045 | O(n) |
| Búsqueda Hash | xyz (no existe) | 0.000004 | O(1) promedio |

**Video Explicativo**

**- Enlace:** https://youtu.be/EFm5W9_lWmg


**Código Completo**

- El código completo está disponible en los repositorios de GitHub mencionados arriba. Los archivos clave son:
  - `src/principal.py`: Menú interactivo e integración de módulos.
  - `src/procesador_texto.py`: Procesamiento de texto y búsquedas.
  - `src/algoritmos.py`: Implementación de Quick Sort.
  - `data/entrada.txt`: Archivo de texto analizado