

# Taller 3-4

Andrés Salcedo Vera

*Fundacion Universitaria Konrad Lorenz*

**Resumen**—A continuación, se presentarán los resultados obtenidos a partir de los algoritmos desarrollados para completar el Taller 3-4. Además, se detallará el proceso empleado para abordar los requisitos establecidos.

El programa se implementó en Python, aprovechando conceptos de caché, listas, diccionarios e índices invertidos. Es importante destacar que no se recurrió al uso de bibliotecas o software externo para resolver los problemas planteados.

## I. INTRODUCCIÓN

El primer requisito consistía en desarrollar un algoritmo que permitiera generar un ranking de palabras a partir de una lista en Python con más de 60 oraciones, utilizando Memorization como fundamento para optimizar el programa.

El siguiente requisito implicaba la creación de un algoritmo que, mediante el uso de índices invertidos en la misma lista, generara una nueva lista que contuviera cada una de las palabras y aparición en cada una de las oraciones.

A continuación se mostrará la lista que se usó para el programa:

- La programación en Python es clave para el trabajo con datos.
- Los programadores en Java tienen un alto interés en pasar a Python.
- La optimización de algoritmos es fundamental en el desarrollo de software.
- Las bases de datos relacionales son esenciales para muchas aplicaciones.
- El paradigma de programación funcional gana popularidad.
- La seguridad informática es un tema crucial en el desarrollo de aplicaciones web.
- Los lenguajes de programación modernos ofrecen abstracciones poderosas.
- La inteligencia artificial está transformando diversas industrias.
- El aprendizaje automático es una rama clave de la ciencia de datos.
- Las interfaces de usuario intuitivas mejoran la experiencia del usuario.
- La calidad del código es esencial para mantener un proyecto exitoso.
- La agilidad en el desarrollo de software permite adaptarse a cambios rápidamente.
- Las pruebas automatizadas son cruciales para garantizar la estabilidad del software.
- La modularización del código facilita la colaboración en equipos de programadores.
- El control de versiones es necesario para rastrear cambios en el código.
- La documentación clara es fundamental para que otros entiendan el código.
- La programación orientada a objetos promueve la reutilización de código.
- La resolución de problemas es una habilidad esencial en la programación.
- La optimización prematura puede llevar a código complicado y difícil de mantener.
- El diseño de interfaces de usuario atractivas mejora la usabilidad de las aplicaciones.
- El código limpio es esencial para facilitar el mantenimiento.
- Los patrones de diseño son soluciones probadas para problemas comunes.
- Las pruebas unitarias garantizan el correcto funcionamiento de las partes del código.
- El desarrollo ágil prioriza la entrega continua de valor al cliente.
- Los comentarios en el código deben ser claros y útiles.
- La recursividad es una técnica poderosa en la programación.
- Las bibliotecas de código abierto aceleran el desarrollo de software.
- La virtualización permite una mejor utilización de los recursos de hardware.
- La seguridad en la programación web es fundamental para prevenir ataques.
- Los principios SOLID son fundamentales para el diseño de software robusto.
- La arquitectura de microservicios permite escalar componentes individualmente.
- La refactorización mejora la calidad del código sin cambiar su comportamiento.
- Los sistemas distribuidos presentan desafíos en la sincronización de datos.
- El enfoque DevOps une el desarrollo y las operaciones para una entrega eficiente.
- Las bases de datos NoSQL son útiles para manejar datos no estructurados.
- La agilidad en el desarrollo permite adaptarse a cambios del mercado.
- Las buenas prácticas en el control de versiones facilitan la colaboración.
- La programación concurrente mejora la eficiencia en sistemas multiusuario.
- Los marcos de trabajo MVC separan la lógica de la interfaz de usuario.
- La interacción entre aplicaciones se logra a través de

APIs.

- El machine learning permite a las máquinas aprender de los datos.
- La analítica de datos ayuda a tomar decisiones basadas en información.
- El diseño responsivo garantiza una experiencia consistente en diferentes dispositivos.
- Las pruebas de carga verifican el rendimiento de las aplicaciones.
- El enfoque centrado en el usuario mejora la usabilidad de las aplicaciones.
- La programación reactiva es útil para manejar flujos de datos asincrónicos.
- Los contenedores facilitan la implementación y el despliegue de aplicaciones.
- La gestión de dependencias es esencial para administrar las bibliotecas externas.
- La integración continua automatiza la verificación de cambios en el código.
- El aprendizaje profundo es una rama avanzada del machine learning.
- La depuración es una habilidad crucial para encontrar y corregir errores.
- La criptografía protege la información sensible en aplicaciones.
- El desarrollo full-stack abarca tanto el frontend como el backend.
- Las pruebas de seguridad ayudan a identificar vulnerabilidades en el software.
- La agilidad cultural es clave para adoptar prácticas ágiles de manera efectiva.
- La infraestructura como código permite automatizar la gestión de servidores.
- Los patrones arquitectónicos guían la estructura general de una aplicación.
- El análisis predictivo utiliza datos históricos para predecir tendencias.
- Las interfaces API REST son ampliamente utilizadas para comunicarse con aplicaciones.
- El rendimiento de las aplicaciones es esencial para brindar una buena experiencia.
- La virtualización de servidores reduce costos y facilita la administración.
- La ingeniería de software implica la aplicación de métodos sistemáticos.
- El código autodocumentado es claro y fácil de entender para otros programadores.
- La integración de sistemas conecta diferentes aplicaciones para trabajar juntas.
- Las metodologías ágiles promueven la adaptación y la colaboración continua.
- El monitoreo de aplicaciones permite identificar y resolver problemas en tiempo real.
- El análisis de datos masivos (big data) abre oportunidades para obtener insights.
- El diseño de interfaces de usuario es crucial para la experiencia del usuario.
- La seguridad en el desarrollo es un proceso constante de

mitigación de riesgos.

## II. PRIMER PUNTO

Como mencionamos anteriormente, el primer objetivo era determinar la frecuencia de aparición de cada palabra en una lista. Para lograr esto, se empleó Python como la herramienta de resolución. En primer lugar, se creó una función que toma dos parámetros: la lista que se ingresará y una caché. Dentro de esta función, se utilizó un diccionario llamado "diccionario de palabras" para almacenar cada palabra como clave y su frecuencia como valor. Para llevar a cabo esta tarea, se realizaron dos iteraciones: la primera recorre cada una de las frases en la lista, mientras que la segunda itera a través de las palabras, separadas utilizando la función `.split`.

Hasta este punto, solo se han ingresado los datos y se ha extraído la información de la lista. Ahora, es necesario gestionar dos aspectos clave: 1) guardar los datos en la caché y 2) asignar el valor necesario a cada palabra, que actuará como nuestro contador. Para abordar esto, se implementó un filtro inicial con una declaración `if`. Si la palabra ya se encontraba en la caché, el programa simplemente buscaría la palabra en el diccionario y aumentaría su valor en 1. En caso contrario, se crearía una nueva entrada en el diccionario con la palabra como clave y se le asignaría un valor de 1.

Finalmente, para mantener una estructura organizada, se creó otro diccionario que se presentaría al usuario del programa. Aquí, se organizó la información utilizando la función `sorted`, que toma como parámetro cada uno de los elementos de la lista previamente creada. Se utilizó una función `lambda` como iterador, que recorrería los elementos uno a uno y los organizaría según el valor de la palabra, asignándoles una posición correspondiente.

Este sería el algoritmo correspondiente:

```
def cantidad_palabras(lista, cache={}):
    lista_palabras = {}
    for column in my_documents:
        for palabra in column.split():
            if palabra in lista_palabras:
                lista_palabras[palabra] += 1
            else:
                lista_palabras[palabra] = 1
    lista_ordenada = dict(sorted(lista_palabras.items(), key=lambda item: item[1], reverse=True))
    return lista_ordenada
```

Figura 1. Algoritmo


### II-A. Big O

En el código, podemos observar la presencia de varios bucles, los cuales incrementan la complejidad de la función en términos de rendimiento. Para ilustrar este punto, ejecutamos el código en un ordenador ASUS TUF Gaming F15 con un procesador Core i5, una tarjeta gráfica GTX 1660 Ti y 16 GB de RAM, lo que resultó en un tiempo de ejecución total de tan solo 0,20 milisegundos. En este caso particular, este tiempo es insignificante y no afecta el rendimiento de manera apreciable.

En cuanto a la complejidad del algoritmo, se puede caracterizar como  $O(n^3)$ . Esta complejidad se deriva de la presencia de dos bucles `'for'` y de la función `'lambda'`, que también realiza un recorrido sobre la lista previamente creada.

## II-B. Funcionalidad y resultados

y así se ejecutaría:



```
cantidad_palabras(my_documents[1])
```

Figura 2. Ejecutar

y este sería el resultado del top 10 de las palabras:

- 'de': 59
- 'La': 31
- 'la': 27
- 'en': 24
- 'para': 23
- 'el': 22
- 'es': 20
- 'El': 18
- 'código': 14
- 'aplicaciones': 12

Como se puede observar, en el código no se ha implementado la distinción entre mayúsculas y minúsculas. Esto depende de las necesidades específicas del proyecto, pero si fuera necesario, se podría agregar la función `.lower()` al código. Esto permitiría que las palabras se agrupen sin importar si están escritas en mayúsculas o minúsculas, adaptándose así a las necesidades particulares del usuario.

### Segunda parte

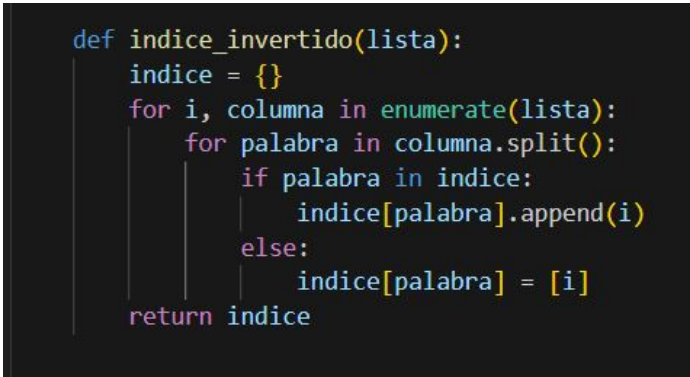
El segundo objetivo en el taller consistía en identificar la ubicación de cada palabra en las 69 oraciones. Por ejemplo, si la palabra "Python" aparecía en la primera y la última oración, el resultado debía indicar: Python [0, 68].

Para resolver este problema, se implementaron índices invertidos, lo que permite este tipo de búsquedas en los documentos disponibles.

Nuevamente, se aplicó la estructura de función en Python para abordar este problema. En este caso, la función toma como único parámetro la lista de oraciones. Dentro del código, se crea una lista vacía que actuará como un diccionario para almacenar las posiciones de las palabras. Se inicia un bucle for que recorre las oraciones, utilizando dos contadores: `i` para el número de identificación de la oración y `columna` como el iterador de cada elemento en nuestra lista. Luego, se utiliza la función `enumerate` para obtener un contador y la lista como parámetros.

A continuación, se aplica la función `.split` para separar las palabras en esa oración. Similar a la función anterior, en lugar de utilizar una caché, se utiliza un contador para cada elemento en la lista. Luego, se implementa una estructura condicional (if) para verificar si la palabra ya está en el diccionario de "índices". Si la palabra ya existe, se agrega el número de identificación de la oración a la lista de posiciones de esa palabra. En caso contrario, se crea una entrada para la palabra en el diccionario y se le asigna la posición actual en la lista de oraciones.

Finalmente, la función retorna la lista de índices invertidos, que proporciona la ubicación de cada palabra en las oraciones. Este enfoque permite una rápida búsqueda de las ubicaciones de las palabras en los documentos.



```
def indice_invertido(lista):
    indice = {}
    for i, columna in enumerate(lista):
        for palabra in columna.split():
            if palabra in indice:
                indice[palabra].append(i)
            else:
                indice[palabra] = [i]
    return indice
```

Figura 3. Segundo Punto

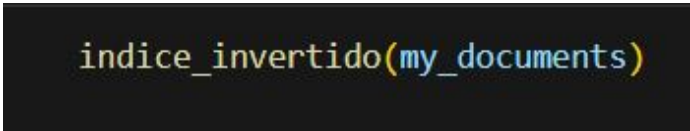
## II-C. Big O

En el código, podemos observar la presencia de varios bucles, los cuales incrementan la complejidad de la función en términos de rendimiento. Para ilustrar este punto, ejecutamos el código en un ordenador ASUS TUF Gaming F15 con un procesador Core i5, una tarjeta gráfica GTX 1660 Ti y 16 GB de RAM, lo que resultó en un tiempo de ejecución total de tan solo 0,35 milisegundos. En este caso particular, este tiempo es insignificante y no afecta el rendimiento de manera apreciable.

En cuanto a la complejidad del algoritmo, se puede caracterizar como  $O(n^2)$ . Esta complejidad se deriva de la presencia de dos bucles 'for' y en este caso no se hizo una organización del código, por lo que no aumento la complejidad en otro aspecto.

## II-D. Ejecutable y resultados

Este sería el ejecutable y unos ejemplos de las palabras que pueden salir:



```
indice_invertido(my_documents)
```

Figura 4. Selección de país

- 'programación': [0, 4, 6, 16, 17, 25, 28, 37, 45]
- 'clave': [0, 8, 54]
- 'trabajo': [0, 38]
- 'con': [0, 58]
- 'datos': [0, 3, 8, 32, 34, 34, 40, 41, 45, 57, 66]
- 'Los': [1, 6, 21, 24, 29, 32, 38, 46, 56]
- 'programadores': [1, 13, 62]
- 'Java': [1]
- 'tienen': [1]
- 'un': [1, 5, 10, 68]
- 'alto': [1]

### III. CONCLUSIONES

El desarrollo de este taller brinda la oportunidad de adquirir una comprensión más profunda sobre el uso de herramientas y métodos que contribuyen a la optimización de procesos. Conceptos como el *cache* y los "índices invertidos" suelen ser desconocidos para muchas personas, a pesar de que tienen un impacto significativo en nuestra vida cotidiana y en el ámbito de la programación. Por lo tanto, entender cómo funcionan estos conceptos puede ser sumamente beneficioso para programadores que deseen alcanzar un alto nivel de eficiencia en la optimización de sus programas.

El *cache* se refiere a una técnica de almacenamiento temporal de datos que permite acceder a ellos de manera más rápida y eficiente. Esto resulta esencial en el desarrollo de software y en la mejora del rendimiento de las aplicaciones, ya que acelera la recuperación de información previamente procesada, reduciendo así la carga de trabajo en el sistema.

Por otro lado, los "índices invertidos" son herramientas fundamentales para la búsqueda y recuperación de información en grandes conjuntos de datos. Estos índices almacenan información sobre la ubicación de palabras o elementos específicos en un conjunto de documentos, lo que facilita la búsqueda rápida y precisa de información relevante. Son esenciales en motores de búsqueda, bases de datos y sistemas de recuperación de información.

La adquisición de conocimientos en estas áreas no solo enriquece la habilidad técnica de un programador, sino que también tiene un impacto directo en la eficiencia de los programas que desarrollan. Al comprender y aplicar eficazmente el *cache* y los índices invertidos, los programadores pueden mejorar significativamente la velocidad y el rendimiento de sus aplicaciones, lo que a su vez se traduce en una mejor experiencia para los usuarios finales. En resumen, este taller proporciona una base sólida para aquellos que buscan perfeccionar sus habilidades de programación y contribuir a la optimización de la tecnología que utilizamos en nuestra vida cotidiana.

### IV. BIBLIOGRAFIA

- OpenAI. (2021). ChatGPT: A large language model.  
Andrés Salcedo Vera. (2023). Taller 3-4.