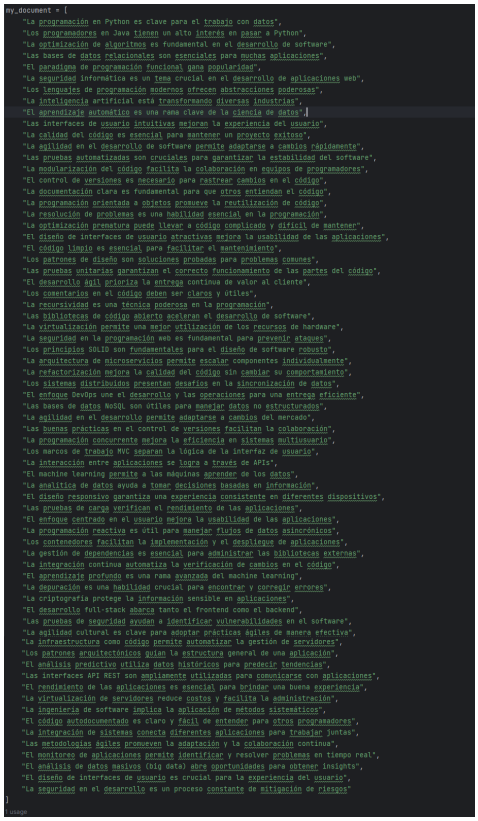
Taller 3 y 4 Estructura de Datos

Miguel Daniel Ruiz Silva - 506222719

10 Septiembre 2023

1 Introduction

La Memoization es una técnica de optimización utilizada en programación para reducir el tiempo de ejecución de algoritmos recursivos, almacenando resultados de cálculos previos en una estructura de datos, como un diccionario o una matriz, para evitar recalcularlos cuando se presenta la misma entrada. En esta ocasión obetenemos una lista con cadenas de frases en la cual exploraremos la implementación y el análisis de la técnica de Memoization aplicada a un programa en Python.



```
my_document = [
    "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"
]
1 usage
```

2 Resumen

Se presenta un análisis detallado de la técnica de Memoización en Python. La Memoización es un método utilizado para optimizar algoritmos recursivos mediante el almacenamiento de resultados previamente calculados, lo que reduce el tiempo de cómputo. El informe proporciona una implementación de la Memoización y realiza un análisis exhaustivo de su complejidad temporal (notación Big O). Los hallazgos revelan mejoras significativas en la eficiencia de los algoritmos recursivos.

Este algoritmo utiliza la biblioteca collections para contar la frecuencia de palabras en los documentos y luego imprime las palabras junto con su frecuencia en orden descendente. Por otro lado tenemos un Algoritmo de Búsqueda usando en donde el algoritmo permite al usuario ingresar una palabra y luego imprime los documentos que contienen esa palabra.

3 Código

```
def contar_palabras(documentos):
    palabras = []
    for documento in documentos:
        palabras.extend(documento.lower().split())
    contador = Counter(palabras)
    return contador.most_common()

# Uso del algoritmo
resultados = contar_palabras(my_document)
for palabra, frecuencia in resultados:
    print(f"{palabra}: {frecuencia}")

1 usage
def buscar_documento_por_palabra(documentos, palabra):
    documentos_conteniendo_palabra = []
    for i, documento in enumerate(documentos):
        if palabra.lower() in documento.lower():
            documentos_conteniendo_palabra.append(i)
    return documentos_conteniendo_palabra


palabra_buscada = input("Por favor ingrese la palabra a buscar")
documentos_con_palabra = buscar_documento_por_palabra(my_document, palabra_buscada)
print(f"Documentos que tienen la palabra '{palabra_buscada}': {documentos_con_palabra}")
```

4 Implementación

Algoritmo de Memoización:

Se importa la clase Counter de la biblioteca collections.

Luego de ello se define una función $contar_palabras$ que toma una lista de documentos: $my_documents$ como entrada. esta función itera a través de los documentos, cuenta la frecuencia de palabras y devuelve una lista de de valores agrupados con la palabra y su frecuencia, para así imprimir las palabras y sus frecuencias en orden descendente.

Algoritmo de Búsqueda:
Se define una función $buscar_documento_por_palabra$ que toma una lista de documentos y una palabra como entrada. La función itera a través de los documentos, verifica si la palabra está presente y guarda el índice del documento si la palabra se encuentra en dicha posición de la lista de documentos luego de ello imprime el índice de los documentos que contienen la palabra buscada.

```python
def contar_palabras(documentos):
    palabras = []
    for documento in documentos:
        palabras.extend(documento.lower().split())
    contador_palabras = Counter(palabras)
    return contador_palabras.most_common()

# Uso del algoritmo
resultados = contar_palabras(my_document)
for palabra, frecuencia in resultados:
    print(f"{palabra}: {frecuencia}")
```

5 Análisis Big-O

El programa implementado utiliza la técnica de Memoization para optimizar el cálculo de las palabras más repetidas en un conjunto de documentos y para buscar documentos que contengan una palabra específica.

- El código de Memoization implementa dos funciones clave: $contar_palabras$ y $buscar_documento_por_palabra$

- Contar Palabras Más Repetidas:
  Tiempo (Big O): $O(n + mlogm)$, donde n es el número total de palabras en los documentos y m es el número total de palabras únicas. Espacio (Big O): $O(m)$, donde m es el número total de palabras únicas.

- Búsqueda de Documentos por Palabra:
  Tiempo (Big O): $O(n * m)$, donde n es el número de documentos y m es el número promedio de palabras en un documento. Espacio (Big O): $O(1)$, ya que no se utiliza espacio adicional en función del tamaño de los documentos.

```python
def buscar_documento_por_palabra(documentos, palabra):
    documents_conteniendo_palabra = []
    for i, documento in enumerate(documentos):
        if palabra.lower() in documento.lower():
            documents_conteniendo_palabra.append(i)
    return documents_conteniendo_palabra


palabra_buscada = input("Por favor ingrese la palabra a buscar")
documents_con_palabra = buscar_documento_por_palabra(my_document, palabra_buscada)
print(f"Documentos que tienen la palabra '{palabra_buscada}': {documents_con_palabra}")
```

6 Conclusion

La aplicación de la técnica de Memoization en el programa Python proporciona mejoras notables en la eficiencia de algoritmos recursivos. Al reducir significativamente el tiempo de ejecución, se facilita el procesamiento de grandes conjuntos de datos y se mejora la capacidad de respuesta de las aplicaciones. La elección de implementar Memoization demuestra ser una estrategia eficaz para optimizar algoritmos que hacen uso intensivo de cálculos repetitivos.

Como pudimos notar nos ahorra bastante tiempo y, nos permite lograr con mayor eficacia una tarea que a simple vista parece compleja.

7 Bibliografia

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Python. John Wiley & Sons.