

# Taller 3 count word

Jose David Ramirez Beltran - 506222723

27 de agosto de 2023

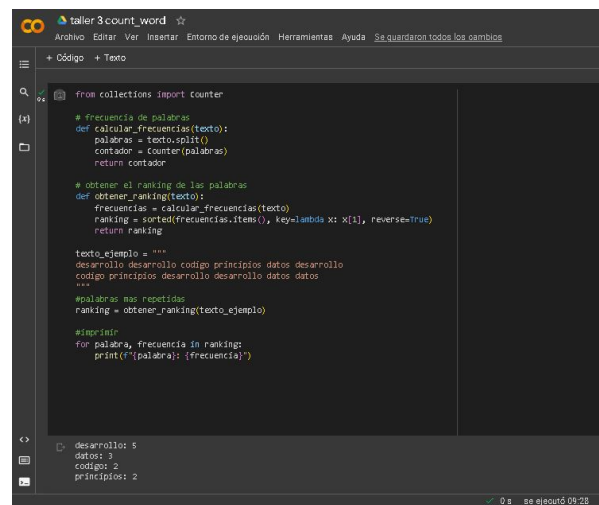
## 1. Introducción

En el mundo de la programación, el conteo de palabras se refiere a una tarea común que consiste en contar las palabras en un texto, utilizando código informático. Esta tarea puede ser útil en diversas aplicaciones, como procesamiento de lenguaje natural, análisis de textos, generación de estadísticas y muchos más ámbitos.

Los programadores suelen tomar este concepto, utilizando algoritmos y funciones que detectan espacios en blanco o caracteres especiales para así poder identificar los límites entre palabras, y esto puede variar según el lenguaje de programación en el que se trabaje y el problema al que se le quiera dar solución.

Dando un ejemplo, en Python, se pueden usar funciones de cadenas para dividir el texto en palabras utilizando espacios como separadores y así poder contar la cantidad total o que resulta de palabras. En otros lenguajes, como C++ o Java, es posible utilizar iteraciones y condicionales para llegar a lo mismo.

## 2. Código 1



```
from collections import Counter

# frecuencia de palabras
def calcular_frecuencias(texto):
    palabras = texto.split()
    contador = Counter(palabras)
    return contador

# obtener el ranking de las palabras
def obtener_ranking(texto):
    frecuencias = calcular_frecuencias(texto)
    ranking = sorted(frecuencias.items(), key=lambda x: x[1], reverse=True)
    return ranking

texto_ejemplo = """
desarrollo desarrollo codigo principios datos desarrollo
codigo principios desarrollo desarrollo datos datos
"""

# palabras mas repetidas
ranking = obtener_ranking(texto_ejemplo)

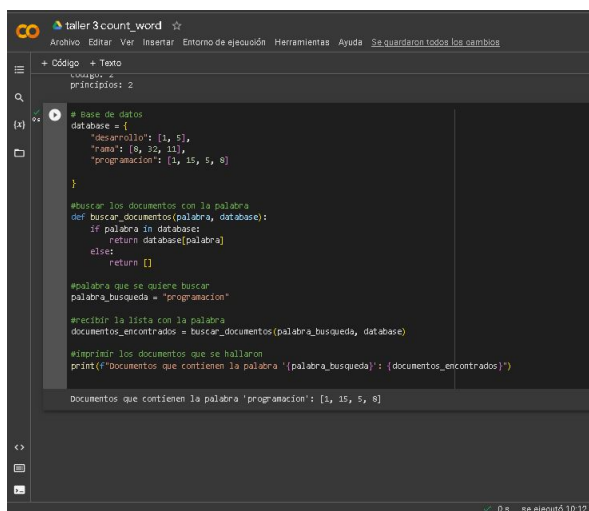
# imprimir
for palabra, frecuencia in ranking:
    print(f"{palabra}: {frecuencia}")
```

- Complejidad Temporal
  - La función `calcular_frecuencias` tiene una complejidad temporal de  $O(n)$ , donde  $n$  es la longitud del texto en palabras, ya que recorre el texto una vez para contar las frecuencias de las palabras.
  - La función `obtener_ranking` realiza una ordenación de las frecuencias, por ende tiene una complejidad temporal de  $O(m \log m)$ , donde  $m$  es el número de palabras únicas en el texto.
  - Por lo tanto, la complejidad temporal total del algoritmo sería  $O(n + m \log m)$ , siendo la parte de ordenación que tiene mayor relevancia en el código.
- Complejidad Espacial
  - La función `calcular_frecuencias` utiliza el diccionario (`Counter`) ahí es donde almacena

las frecuencias de las palabras, lo que requiere espacio que sea proporcional al número de palabras únicas en el texto, es decir,  $O(m)$ .

- La función `obtener_ranking` crea una lista de *tuplas* parecido a un array para almacenar el ranking, que requiere un espacio proporcional al número de palabras en el texto, es decir,  $O(m)$ .
- Es decir que la complejidad espacial del algoritmo es mayor por el almacenamiento de las frecuencias y el ranking, y eso llevaría a decir que tiene una complejidad espacial de  $O(m)$ .

### 3. Código 2



```

# Base de datos
database = {
    "desarrollo": [1, 5],
    "rama": [8, 32, 11],
    "programacion": [1, 15, 5, 8]
}

# Buscar los documentos con la palabra
def buscar_documentos(palabra, database):
    if palabra in database:
        return database[palabra]
    else:
        return []

# palabra que se quiere buscar
palabra_busqueda = "programacion"

# escribir la lista con la palabra
documentos_encontrados = buscar_documentos(palabra_busqueda, database)

# imprimir los documentos que se hallaron
print(f"documentos que contienen la palabra '{palabra_busqueda}': {documentos_encontrados}")

documentos que contienen la palabra 'programacion': [1, 15, 5, 8]

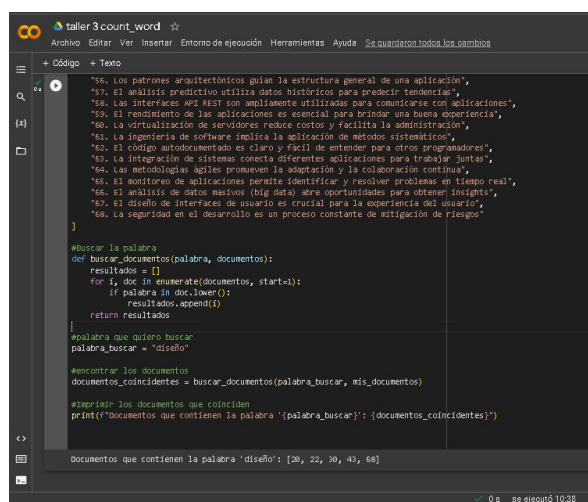
```

- Complejidad Temporal
- La función `buscar_documentos` realiza una búsqueda en una base de datos para encontrar los documentos que contienen la palabra.
- La búsqueda en una base de datos tiene una complejidad promedio de  $O(1)$  en el caso ideal. Pero, en el peor caso, cuando hay colisiones en la función hash, la complejidad puede aumentar a  $O(n)$ , donde  $n$  sería el número de palabras registradas en la base de datos.
- Ahora, la complejidad temporal en este caso depende de cómo se manejen las colisiones en la función hash y de la estructura interna de

la base de datos. Si asumimos una implementación eficiente de la base de datos (como un hash map que se vio en clase), se podría considerar la complejidad como  $O(1)$ .

- Complejidad Espacial
- La base de datos `database` almacena las palabras como claves y los documentos como listas de valores que les son asociados.
- La cantidad total de espacio requerido para la base de datos depende del número total de palabras únicas y de la cantidad de documentos que le pertenecen a cada palabra.
- Si hay  $m$  palabras únicas y cada palabra tiene  $k$  documentos asociados, la complejidad espacial sería  $O(m + k \cdot m)$ , es decir,  $O(k \cdot m)$ , ya que se espera que el número de palabras únicas sea menor que la cantidad total de documentos en el caso más general.

### 4. Código 3



```

#66. Los patrones arquitectónicos guían la estructura general de una aplicación",
#67. El análisis predictivo utiliza datos históricos para predecir tendencias",
#68. Los interfaces API son ampliamente utilizados para comunicarse con aplicaciones",
#69. El rendimiento de las aplicaciones es esencial para brindar una buena experiencia",
#70. La virtualización de servidores reduce costos y facilita la administración",
#71. La ingeniería de software implica la aplicación de métodos sistemáticos",
#72. El código autocompletado es claro y fácil de entender para otros programadores",
#73. La integración de sistemas conecta diferentes aplicaciones para trabajar juntas",
#74. Las metodologías ágiles promueven la adaptación y la colaboración continua",
#75. El monitoreo de aplicaciones permite identificar y resolver problemas en tiempo real",
#76. El análisis de datos masivos (big data) abre oportunidades para obtener insights",
#77. El diseño de interfaces de usuario es crucial para la experiencia del usuario",
#78. La seguridad en el desarrollo es un proceso constante de mitigación de riesgos"

# Buscar la palabra
def buscar_documentos(palabra, documentos):
    resultados = []
    for i, doc in enumerate(documentos, start=1):
        if palabra in doc.lower():
            resultados.append(i)
    return resultados

# palabra que quiero buscar
palabra_buscar = "diseno"

# encontrar los documentos
documentos_coincidentes = buscar_documentos(palabra_buscar, mis_documentos)

# imprimir los documentos que coinciden
print(f"documentos que contienen la palabra '{palabra_buscar}': {documentos_coincidentes}")

documentos que contienen la palabra 'diseno': [20, 22, 30, 43, 68]

```

- Complejidad Temporal
- La función `buscar_documentos` recorre la lista de documentos y verifica si la palabra buscada está en cada documento. En el peor caso, tiene que recorrer todos los documentos para verificar si la palabra está presente.
- Si hay  $n$  documentos en total y en cada documento tiene una longitud de  $m$  palabras, entonces la complejidad temporal sería  $O(n \times m)$ .

- En este caso,  $n$  es el número total de documentos y  $m$  es el promedio de palabras en cada documento.
- Dado que en cada documento se realiza una operación de verificación de la palabra, la complejidad temporal depende del tamaño promedio de los documentos y del número total de documentos.
- Complejidad Espacial
- La función `buscar_documentos` crea una lista `resultados` para almacenar los números de documentos que contienen la palabra buscada.
- La cantidad máxima de elementos en esta lista sería igual al número total de documentos.
- Si hay  $n$  documentos en total, la complejidad espacial sería  $O(n)$  debido a la lista de resultados.

<https://colab.research.google.com/drive/1dsacKQCMP5q24xzxij8Esharing>

<https://github.com>

Indices Invertidos

<https://www.youtube.com/watch?v=TvHv2UZvx74>

## 5. Conclusiones

En general, la complejidad temporal de un algoritmo se refiere a la cantidad de tiempo que tarda en ejecutarse en función del tamaño de entrada y la complejidad espacial hace referencia a la cantidad de espacio en memoria que utiliza teniendo en cuenta el tamaño de entrada. Por otro lado, la memorización es la técnica que se usa para optimizar algoritmos recursivos, almacenando resultados previos y reutilizándolos para evitar cálculos repetitivos, esto ayudando a reducir la complejidad temporal de algoritmos recursivos al evitar hacer un cálculo nuevamente que ya se había realizado. Por último, es importante analizar la complejidad temporal y espacial de los algoritmos para entender cómo se comportarán en diferentes escenarios, y el análisis de algoritmos ayuda a seleccionar el enfoque adecuado para desarrollar y tratar de solucionar un problema, pero también a tomar decisiones sobre el rendimiento algorítmico. Por ende, esto también nos ayuda a comprender como funciona un código y como poder mostrar varios datos que se encuentran estipulados en una base de datos.

## 6. Bibliografía

Ejercicios mostrados en el documento