TRABAJO PRÁCTICO INTEGRADOR

PROGRAMACIÓN

Análisis de algoritmos

Alumnos:

Facundo Auciello (Comisión 24) - facundoauciello@gmail.com

Ayelen Etchegoyen (Comisión 13) - aye.etche@gmail.com

Materia: Programación

Profesor: Enferrel, Ariel (Comisión 13)

Tutor: Gonzalez, Franco (Comisión 13)

Profesor: AUS Bruselario, Sebastián (Comisión 24)

Tutor: Gubiotti, Florencia (Comisión 24)

Fecha de Entrega: 9 de Junio de 2025

Introducción

Este trabajo tiene como eje central el análisis de algoritmos, un algoritmo es un conjunto de pasos o instrucciones bien definidas diseñadas para resolver un problema específico. En programación, los algoritmos son la base de cualquier solución, y su eficiencia es crucial para el rendimiento de las aplicaciones.

Se investigará la eficiencia de dos estrategias para abordar un problema planteado por los estudiantes, el conteo de palabras en un texto. Para esto, se desarrollarán dos versiones del mismo algoritmo: uno basado en listas y otro en diccionarios. Este problema es común en áreas de procesamiento de lenguaje natural, análisis de grandes volúmenes de datos y programación de motores de búsqueda. Estas opciones requieren algoritmos capaces de un buen análisis y rendimiento.

El objetivo es analizar y comparar ambas soluciones utilizando herramientas de medición empírica (tiempos reales de ejecución) y notación Big-O (análisis teórico), con el fin de evaluar el comportamiento frente a diferentes tamaños de entrada. De este modo, se pretende fomentar una comprensión más profunda sobre cómo se estudia y trabaja la eficiencia de los algoritmos y por qué es una habilidad esencial en el desarrollo de software.

Marco Teórico

Características de un algoritmo

- Correcto: resuelve el problema planteado de forma precisa, sin errores.
- Eficiente: aprovecha al máximo los recursos disponibles, especialmente tiempo y memoria.
- Robusto: es capaz de manejar situaciones inesperadas sin fallar.
- Legible: se encuentra escrito de forma clara, comprensible y mantenible.
- Modular: puede convertirse en partes reutilizables.

¿Por qué analizar algoritmos?

Cuando programamos existen múltiples soluciones a un mismo problema que debemos analizar. Evaluar cuál de las diferentes soluciones ofrece el mejor rendimiento es fundamental, especialmente si son aplicaciones con grandes volúmenes de datos o alta demanda de procesamiento.

Por eso, la complejidad temporal (cuanto tarda un algoritmo en ejecutarse) y la complejidad espacial (cuanta memoria requiere) son clave en la toma de decisiones durante el desarrollo.

Análisis empírico

El análisis empírico es una técnica que consiste en evaluar el rendimiento real de un algoritmo mediante su ejecución práctica en un entorno controlado. Se basa en datos obtenidos de pruebas reales, como el tiempo de ejecución o el uso de memoria.

Se lleva a cabo implementando un algoritmo en un lenguaje de programación (en este caso Python), ejecutándolo con distintos tamaños de entrada y midiendo cuánto tarda o cuántos recursos utiliza en cada caso. Esta metodología permite observar si el algoritmo se comporta como se esperaría según su complejidad teórica y también si hay diferencias debido al entorno (hardware, sistema operativo).

Este tipo de análisis es fundamental para:

- Validar el comportamiento técnico de un algoritmo.
- Obtener gráficas que muestran el tiempo de ejecución de un algoritmo.
- Facilitar la comparación visual de los tiempos de ejecución de diferentes algoritmos para un mismo problema

Desventajas:

- Los resultados pueden variar según el hardware, sistema operativo o condiciones de ejecución.
- Requiere implementar código y preparar entornos de prueba.
- Los tiempos de ejecución sirven como una aproximación bajo ciertas condiciones, no como verdad universal.

Análisis teórico

El análisis teórico se basa en la predicción del comportamiento del algoritmo (un enfoque matemático), sin la necesidad de ejecutarlo. La idea principal de este análisis es obtener una medida de la eficiencia del algoritmo abstrayéndose de los recursos computacionales. Se calcula una función temporal T(n) que representa el número de operaciones que realiza el algoritmo para una entrada de tamaño “n”.

El orden de crecimiento se refiere al tiempo o espacio que tarda un algoritmo en base a su tamaño de entrada.

Ventajas:

- Permite comparar dos algoritmos.
- Permite identificar la complejidad computacional.
- Permite predecir el rendimiento con distintos volúmenes de entradas.

Análisis según el caso

- Peor caso: se calcula el límite superior del tiempo de ejecución de un algoritmo considerando el caso que genera el máximo de operaciones.
- Mejor caso: se calcula el límite inferior del tiempo de ejecución de un algoritmo tomando la ejecución con el menor número de operaciones realizadas.
- Caso promedio: se toman todas las entradas posibles realizando un promedio de los resultados suponiendo una distribución uniforme.

Notaciones asintóticas más comunes:

- Big O (O): representa el límite superior del crecimiento reflejando el peor caso.
- Big Theta (Θ): representa tanto el límite superior como el inferior. Es el tipo de caso promedio.
- Big Omega(Ω): representa el límite inferior del crecimiento. Es el mejor caso.

Notación Big O

La notación Big O es una forma de describir el comportamiento asintótico de una función, es decir, cómo crece cuando el tamaño de la entrada tiende a infinito. Se utiliza para simplificar la comparación de algoritmos eliminando constantes y términos de menor orden.

- El análisis teórico nos permite calcular la función temporal T(n) de un algoritmo.
- La notación Big-O simplifica la comparación de algoritmos al enfocarse en el término de mayor crecimiento.
- Es una herramienta esencial para elegir el algoritmo más eficiente en función del tamaño de la entrada.

Para obtener el Big O debemos calcular la T(n) sumando el número de operaciones. Luego identificar el término de mayor crecimiento y retirar su coeficiente.

Ejemplo:
$T(n) = 5n^3 + 3n + 1$
Big O = n^3

Principales órdenes de complejidad

El orden de eficiencia es decreciente.

Orden	Nombre	Comportamiento
O(1)	Constante	El tiempo de ejecución no varía conforme aumenta el tamaño de los datos de entrada.
O(log n)	Logarítmica	El algoritmo crece al inicio y luego llega a un punto de estabilización.
O(n)	Lineal	El tiempo de ejecución es proporcional a su entrada.
O(n log n)	Lineal logarítmica	El algoritmo realiza una operación logarítmica por cada elemento de la entrada.
O(n²)	Cuadrática	El tiempo de ejecución aumenta proporcionalmente al cuadrado del tamaño de entrada.
O(n³)	Cúbica	El tiempo de ejecución aumenta proporcionalmente al cubo del tamaño de entrada.
O(a^n)	Exponencial	El tiempo de ejecución crece exponencialmente; el algoritmo se vuelve ineficiente incluso para entradas pequeñas.
O(n!)	Factorial	El tiempo de ejecución aumenta exponencialmente siguiendo el crecimiento factorial de n.

Gráfico de órdenes de complejidad

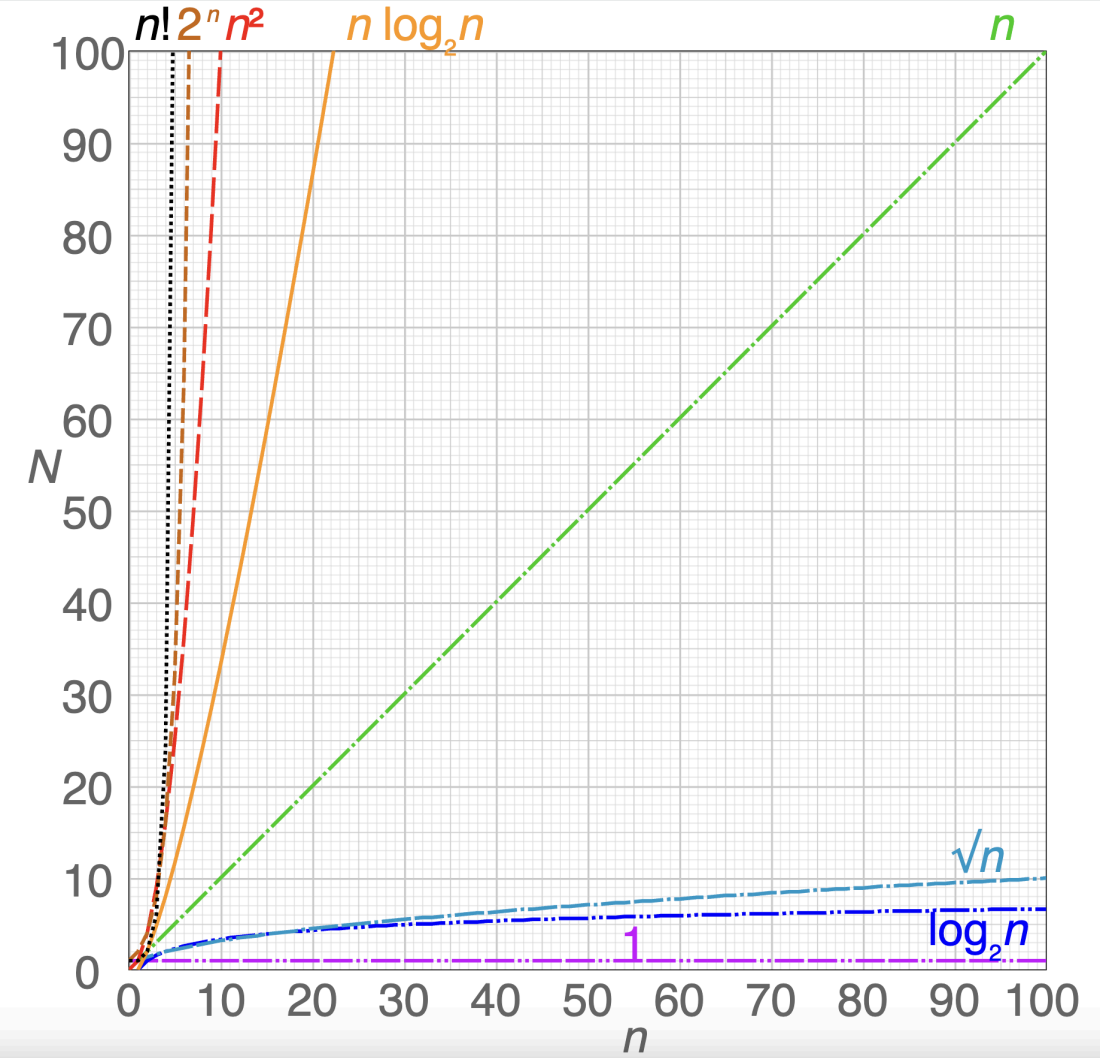


Gráfico que muestra las funciones comúnmente utilizadas en el análisis de algoritmos representando el número de operaciones *N* versus el tamaño de entrada *n*. Wikipedia https://es.wikipedia.org/wiki/An%C3%A1lisis_de_algoritmos

Rendimiento y complejidad de algoritmos clásicos

Algorithm	Big O	Big Ω
linear search	$O(n)$	$\Omega(1)$
binary search	$O(\log(n))$	$\Omega(1)$
bubble sort	$O(n^2)$	$\Omega(n)$
insertion sort	$O(n^2)$	$\Omega(n)$
selection sort	$O(n^2)$	$\Omega(n^2)$

CS50 Computational Complexity (2018) Harvard
https://cs50.harvard.edu/ap/2020/assets/pdfs/computational_complexity.pdf

Listas y diccionarios en Python:

Una lista en Python es una estructura de datos ordenada y mutable que puede almacenar elementos de cualquier tipo (números, cadenas, objetos, etc). Se accede a los elementos por su posición (índice) y es ideal para recorrer y almacenar datos. En este proyecto, se utilizará una lista para guardar tuplas que contienen palabras y su frecuencia, esto requiere recorrer toda la lista para buscar coincidencias y actualizar los valores, lo que puede resultar costoso a medida que crece la cantidad de datos.

Un diccionario en python es una colección no ordenada de pares clave:valor, donde cada clave es única y se accede directamente a los valores mediante estas claves, lo que permite búsquedas y actualizaciones rápidas. En este proyecto, se usará un diccionario para contar cuantas veces aparece cada palabra en un texto, aprovechando su capacidad de acceso rápido y constante, lo cual mejora notablemente el rendimiento.

Caso Práctico

En este caso se plantea un programa que genera listas de palabras aleatorias a partir de un conjunto predefinido y aplica dos algoritmos para contar la frecuencia de aparición de cada palabra:

- Uno basado en una lista de tuplas
- Otro utilizando un diccionario

Ambos métodos son ejecutados con diferentes tamaños de entrada, y se mide el tiempo que tardan en completarse.

Fragmentos del código

Generación de texto aleatorio:

La función **`generarTexto(cantidad)`** genera una lista de palabras aleatorias para simular un texto de entrada. Se utilizará para probar ambos algoritmos.

```
2  import time
3  import random
4
5  def generarTexto(cantidad):
6      palabras = ["programacion", "python", "utn", "variables", "ayso", "oe", "matematica", "funciones"]
7      resultado = []
8      for i in range(cantidad):
9          palabrasAzar = random.choice(palabras)
10         resultado.append(palabrasAzar)
11     return resultado
```

Algoritmo con lista de tuplas:

La función **`contarPalabrasLista(texto)`** recorre una lista acumulando palabras y contando su frecuencia usando tuplas.

```
13 #algoritmo con lista de tuplas
14 def contarPalabrasLista(texto):
15     conteo = []
16     for palabra in texto:
17         encontrado = False
18         for i in range(len(conteo)):
19             if conteo[i][0] == palabra:        You, 2 day
20                 conteo[i] = (palabra, conteo[i][1] + 1)
21                 encontrado = True
22         if not encontrado:
23             conteo.append((palabra, 1))
24     return conteo
```

Algoritmo con diccionario:

La función **`contarPalabrasDiccionario(texto)`** utiliza un diccionario clave:valor para contar palabras. Accede directamente a cada palabra.

```python
#algoritmo con diccionario
def contarPalabrasDiccionario(texto):
    conteo = {}
    for palabra in texto:
        if palabra in conteo:
            conteo[palabra] += 1
        else:
            conteo[palabra] = 1
    return conteo
```

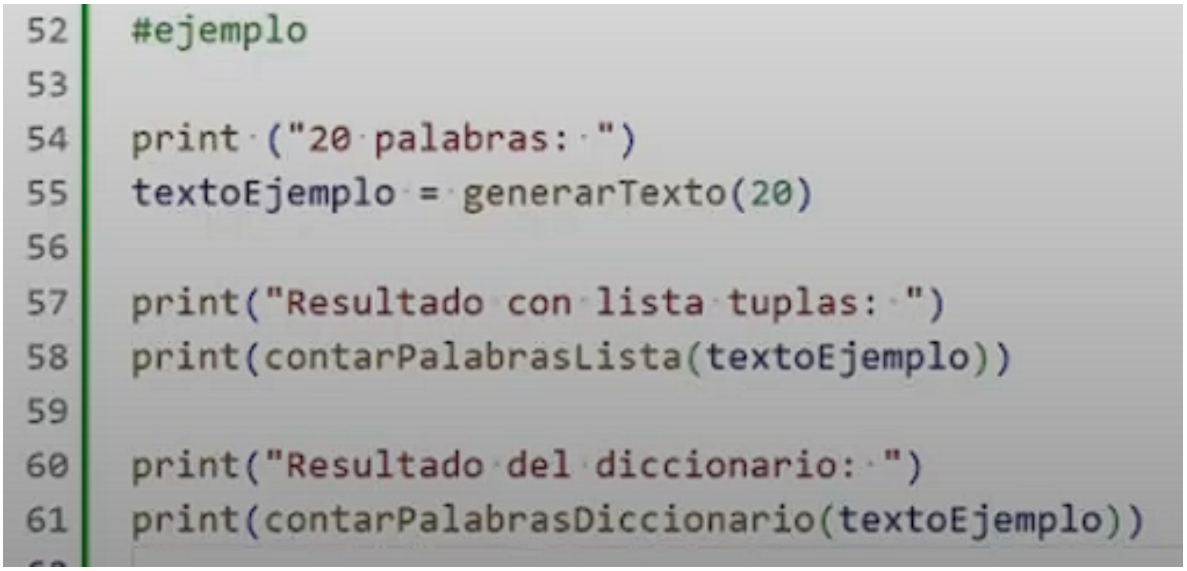
Medición de tiempos de ejecución:

Bucle que prueba ambos algoritmos con diferentes tamaños de texto (1.000, 5.000, 10.000, 50.000, 100.000), y mide el tiempo real que tarda cada uno en ejecutarse.

```python
tamaniosEntrada = [1000, 5000, 10000, 50000, 100000]

for numeros in tamaniosEntrada:
    texto = generarTexto(numeros)
        You, 2 days ago • codigo completo
    inicio = time.time()
    contarPalabrasLista(texto)
    fin = time.time()
    print(f"Lista - {numeros} palabras: {round((fin - inicio) * 1000, 2)} ms")

    inicio = time.time()
    contarPalabrasDiccionario(texto)
    fin = time.time()
    print(f"Diccionario - {numeros} palabras: {round((fin - inicio) * 1000, 2)} ms")
```

Ejemplo de salida con 20 palabras:

Pequeña prueba visual para mostrar cómo se almacenan los resultados del conteo, tanto en la lista como en el diccionario, facilitando la comprensión de la estructura final del código.



```
#ejemplo

print ("20 palabras: ")
textoEjemplo = generarTexto(20)

print("Resultado con lista tuplas: ")
print(contarPalabrasLista(textoEjemplo))

print("Resultado del diccionario: ")
print(contarPalabrasDiccionario(textoEjemplo))
```

Resultado del ejemplo en consola:

```
20 palabras:

Resultado con lista tuplas:

[(ayso, 3), ('utn', 3), ('oe', 2), ('variables', 3),
('programacion', 2), ('funciones', 1), ('matematica', 2),
('python', 4)]

Resultado del diccionario:

{'ayso': 3, 'utn': 3, 'oe': 2, 'variables': 3, 'programacion': 2,
'funciones': 1, 'matematica': 2, 'python': 4}
```

Tiempos en consola

```
Lista - 1000 palabras: 1.0 ms

Diccionario - 1000 palabras: 0.0 ms

Lista - 5000 palabras: 3.0 ms

Diccionario - 5000 palabras: 1.0 ms
```

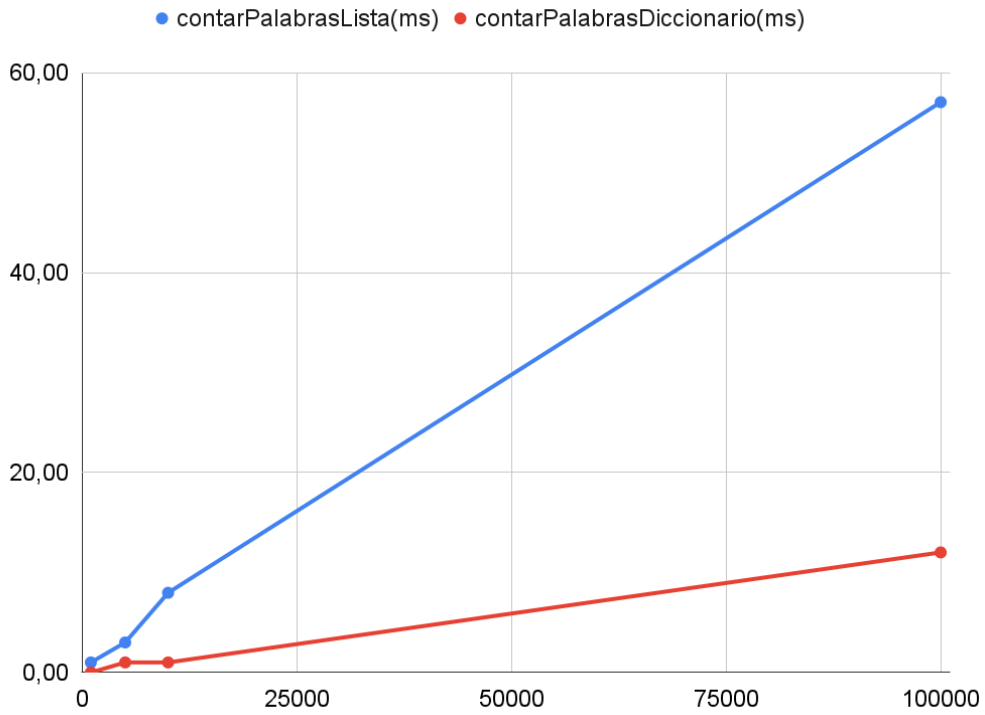
```
Lista - 10000 palabras: 7.97 ms

Diccionario - 10000 palabras: 1.0 ms

Lista - 100000 palabras: 57.06 ms

Diccionario - 100000 palabras: 12.01 ms
```

Análisis empírico

Tiempos de ejecución según el tamaño de entrada para ambos algoritmos:

n	contarPalabrasLista(ms)	contarPalabrasDiccionario(ms)
1000	1,00	0,00
5000	3,00	1,00
10000	7,97	1,00
100000	57,06	12,01

Resultado de representar los algoritmos en el gráfico:



Eje X: cantidad de palabras.
Eje Y: tiempo de ejecución en milisegundos.

Interpretación del gráfico:

Se observa el crecimiento de ambas funciones en relación a las entradas introducidas.

Con las primeras entradas más pequeñas, las líneas se encuentran muy cerca. Luego al aumentar el tamaño de las entradas las líneas del gráfico se alejan significativamente indicando que la función ***contarPalabrasLista*** crece mucho más rápido que la de ***contarPalabrasDiccionario***.

También se observa que la línea roja se mantiene siempre por debajo de la línea azul a lo largo de toda la muestra.

Se concluye que, empíricamente, la función ***contarPalabrasDiccionario*** es la más eficiente.

Análisis teórico

En esta sección abordamos el comportamiento matemático de los algoritmos sin ejecutarlos. Se calculará la función temporal para obtener el mayor término que dará el Big O.

Análisis del peor caso `contarPalabrasLista`

Como se puede observar en el siguiente algoritmo existen dos bucles for anidados: el externo recorre cada palabra del texto y el interno recorre la lista conteo en busca de coincidencias. A medida que el número de palabras distintas que son contabilizadas crece, el bucle interno debe hacer más recorrido, por lo que el peor caso se produce cuando todas las palabras son diferentes.

```
#algoritmo con lista de tuplas
1   def contarPalabrasLista(texto):
2       conteo = [] 1 Operación
3       for palabra in texto: n veces
4           encontrado = False 1 Operación
5           for i in range(len(conteo)): (n-1)/2 veces
6               if conteo[i][0] == palabra: 2 Operaciones
7                   conteo[i] = (palabra, conteo[i][1] + 1) 3 Oper.
8                   encontrado = True 1 Operación
```

```
9           if not encontrado: 1 Operación
10              conteo.append((palabra, 1)) 1 Operación
11      return conteo 1 Operación
```

Cálculo de T(n) y Big O:

1. Línea 2: 1 operación
2. Línea 3 (n veces):
    - Línea 4: 1 * n = n operaciones
    - Línea 5: Suma de 0 + 1 + 2 + ... + (n-1) = n(n-1)/2 veces
    - Línea 6: 3 * n(n-1)/2 = 3n(n-1)/2 operaciones
    - Línea 7: Se ejecuta cuando se encuentra la palabra (0 veces)
    - Línea 8: Se ejecuta cuando se encuentra la palabra (0 veces)
    - Línea 9: 1 * n = n operaciones
    - Línea 10: 1 * n = n operaciones
3. Línea 11: 1 operación

T(n) = 1 + n + 2n(n-1)/2 + n + n + 1

T(n) = 1 + n + $2n^2/2$ - 3n/2 + 2n + 1

T(n) = 2 + 3n - 3n/2 + $2n^2/2$

T(n) = 2 + 3n/2 + $2n^2/2$

T(n) = (4 + 3n + $2n^2$)/2

T(n) = n^2 + 1.5n + 2

Peor caso: T(n) = O(n^2) - cuando todas las palabras son diferentes.

Análisis del peor caso para `contarPalabrasDiccionario`

Este algoritmo presenta un único bucle for, por lo que debemos analizar cuando se producen más cantidad de operaciones observando cada línea del código.

Se infiere que el peor caso sucede cuando las palabras se repiten. Esto se produce porque en la línea 5 tenemos 3 operaciones y en la línea 7 solo 2 operaciones.

#algoritmo con diccionario

```
1   def contarPalabrasDiccionario(texto):
2       conteo = {} 1 Operación
3       for palabra in texto: n veces
4           if palabra in conteo: 1 Operación
5               conteo[palabra] += 1 3 Operaciones
6           else:
7               conteo[palabra] = 1 2 Operaciones
8       return conteo 1 Operación
```

Cálculo de T(n) y Big O:

1. Línea 2: 1 operación
2. Línea 3 (n veces):
    - Línea 4: 1 * n = n operaciones
    - Línea 5: 3 * n = 3n operaciones
    - Línea 7: Se ejecuta cuando no se encuentra la palabra (0 veces)
3. Línea 8: 1 operación

T(n) = 1 + n + 3n + 1

T(n) = 4n + 2

Peor caso: T(n) = O(n) - cuando todas las palabras son iguales

Metodología Utilizada

Para el desarrollo del marco teórico se llevó a cabo una investigación sobre los conceptos fundamentales del análisis de algoritmos, incluyendo la notación Big O, complejidad temporal y espacial, análisis empírico y estructuras de datos.

Se consultaron diferentes fuentes, tanto la documentación propuesta por la materia como libros y artículos sobre la temática junto con la documentación oficial de python 3 (ver Bibliografía).

Esta etapa permitió establecer un mejor entendimiento de las bases teóricas y contextualizar el caso práctico del proyecto propuesto, asegurando una correcta interpretación de los resultados obtenidos.

Implementación del caso práctico

Una vez definido el problema (conteo de palabras en un texto), se diseñaron dos algoritmos para resolverlo: uno utilizando listas y otro utilizando diccionarios. Al ser programados se pensó una lógica clara para facilitar su análisis.

Se generaron textos artificiales de diferentes tamaños con palabras aleatorias ya dadas por nosotros para simular las entradas reales. A cada versión del algoritmo se le midió el tiempo de ejecución para distintas cantidad de palabras (1000, 5000, 10000, 100000), repitiendo el procedimiento varias veces para asegurar la consistencia de los resultados.

Todas las pruebas se realizaron en la misma computadora, bajo condiciones controladas, para minimizar la variabilidad del entorno en el análisis empírico.

Herramientas y recursos utilizados

Visual Studio Code

https://code.visualstudio.com/

Python

https://www.python.org/downloads/

Google spreadsheets

https://workspace.google.com/products/sheets/?hl=es

Trabajo colaborativo

El trabajo fue realizado en equipo, con una división de tareas y comunicación constante. Facundo se encargó de la primera parte del video, realiza una introducción y explica conceptos fundamentales. Explica el desarrollo del código y luego ejecuta las pruebas explicando los tiempos de ejecución obtenidos.

Ayelen realizó la segunda parte del video, enfocada en la representación de los algoritmos en el gráfico. Luego desarrolla el análisis teórico con la función T(n) y la notación Big O para cada caso. Al final cierra con una conclusión.

Ambos trabajamos en la idea principal del proyecto, poniéndonos de acuerdo con el trabajo a realizar y colaborando mutuamente en parte de cada uno, además de realizar encuentros virtuales para desarrollar el código y debatir sobre las pruebas y resultados para escribir el trabajo a la par.

Resultados Obtenidos

- Desarrollo y ejecución correcta del código, obteniendo distintos tiempos de ejecución para cada algoritmo.

- Representación de los resultados en un gráfico, lo que permitió su análisis comparativo.
- Se calculó la función T(n) y su correspondiente notación Big O para cada algoritmo.
- Los análisis arrojaron resultados consistentes y acordes con el comportamiento esperado de los algoritmos.

Podemos afirmar que el algoritmo más eficiente es el que utiliza un diccionario (el segundo analizado). Está afirmación se sostiene en los distintos enfoques aplicados a lo largo del trabajo. Desde el análisis empírico, el segundo algoritmo presentó tiempos de ejecución menores. Luego en el gráfico, donde la línea correspondiente al algoritmo con diccionario (línea roja) permaneció por debajo del algoritmo de lista.

En el análisis teórico, se comprobó que el primer algoritmo posee una complejidad cuadrática $O(n^2)$, mientras que el segundo presenta una complejidad lineal $O(n)$.
Por lo tanto, los resultados fueron coherentes y consistentes.

Conclusiones

Contar la frecuencia de palabras es una operación que se realiza constantemente en sistemas como correctores ortográficos, filtros de spam, análisis de redes sociales o motores de búsqueda como Google. Elegir una estructura poco eficiente, como una lista, en contextos reales puede multiplicar los tiempos de procesamiento de manera crítica.

Entender cómo las decisiones que tomamos como programadores afectan el rendimiento de un programa, y por qué elegir una estructura de datos adecuada puede marcar la diferencia.

Un error común al desarrollar algoritmos es suponer que si un programa funciona, entonces es eficiente. Este trabajo demuestra que el rendimiento no depende solo del resultado correcto, sino también de las decisiones de implementación y estructura de datos.

El trabajo permitió evidenciar las diferencias de eficiencia entre implementaciones y entender la importancia del análisis teórico como herramienta para la toma de decisiones en programación.

Anexo

● Video de la presentación
https://www.youtube.com/watch?v=Ln369Jf7w9c
● Repositorio del proyecto - README

https://github.com/FacuAuciello/progra-integrador/edit/main/README.md

Bibliografía

- Enferrel Ariel (2025), *Introducción al Análisis de Algoritmos*. Universidad Tecnológica Nacional. https://tup.sied.utn.edu.ar/pluginfile.php/10244/mod_label/intro/Introduccion%20al%20Analisis%20de%20Algoritmos.pdf
- Enferrel Ariel (2025), *Análisis Teórico de Algoritmos*. Universidad Tecnológica Nacional. https://tup.sied.utn.edu.ar/pluginfile.php/10251/mod_label/intro/Analisis-Teorico-de-Algoritmos.pdf
- Enferrel Ariel (2025), *Análisis Teórico y Notación Big-O*. Universidad Tecnológica Nacional. https://tup.sied.utn.edu.ar/pluginfile.php/10251/mod_label/intro/Analisis%20de%20algoritmo%20Teorico%20y%20Big%20O.pdf
- Enferrel Ariel (2025), *Notación Big-O*. Universidad Tecnológica Nacional. https://tup.sied.utn.edu.ar/pluginfile.php/10247/mod_label/intro/Notacion-Big-O.pdf
- GeeksForGeeks (12 de Mayo de 2025), *Analysis of Algorithms* https://www.geeksforgeeks.org/analysis-of-algorithms/
- G. Brassard / P. Bratley (n.d.), *FUNDAMENTOS DE ALGORITMIA.* Départament d'informatique et de recherche opérationelle Université de Montréal https://bonetblai.github.io/courses/ci2525/Brassard_Bratley_Fundamentals_of_Algorithmics_ES.pdf
- Thomas H. Cormen, Charles E. Leiserson, Clifford Stein, Ronald L. Rivest (2009), *Introduction to Algorithms, Third Edition*. MIT Press. https://www.google.com.ar/books/edition/Introduction_to_Algorithms_third_edition/F3anBQAAQBAJ?hl=en&gbpv=1&dq=introduction+to+algorithms&printsec=frontcover