UNIVERSIDAD TECNOLOGICA NACIONAL

TRABAJO INTEGRADOR

PROGRAMACIÓN I

TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A DISTANCIA

Título del trabajo:

Análisis de eficiencia en algoritmos de validación de IDs duplicados entre registros y formularios en Python

Alumnos:

Cristian Emmanuel Rivero Corradi - cristianemmanuelrivero@gmail.com

Julio Cesar Roja - julioroja987@gmail.com

Docente Titular:

Nicolas Quiros

Docente Tutor:

Francisco Quarñolo

Fecha de entrega:

09/06/2025

ÍNDICE

INTRODUCCIÓN

El estudio de la eficiencia algorítmica es fundamental para desarrollar programas escalables y con buen rendimiento. En este trabajo analizamos un problema común en sistemas informáticos: la detección de identificadores numéricos duplicados entre una base de datos de usuarios registrados y un conjunto de nuevos registros provenientes de un formulario.

Este tema fue elegido por su aplicabilidad en escenarios reales, como la validación de formularios, la sincronización de datos y el control de integridad en bases de datos. Comprender y comparar distintas estrategias algorítmicas para este tipo de problema es clave en la formación de cualquier programador.

El objetivo principal del trabajo es aplicar los conceptos de análisis algorítmico para comparar dos soluciones al mismo problema: una con complejidad cuadrática, basada en bucles anidados, y otra optimizada mediante el uso de estructuras de datos eficientes, como diccionarios en Python. Además, se busca resaltar el valor de la notación Big-O como herramienta para anticipar el comportamiento de los algoritmos ante diferentes volúmenes de datos, más allá de los resultados prácticos inmediatos.

MARCO TEÓRICO

Análisis de algoritmos:

Es una disciplina que permite describir y comparar el rendimiento de diferentes soluciones algorítmicas. Se enfoca principalmente en dos aspectos:

- **Eficiencia temporal:** cuánto tiempo tarda un algoritmo en ejecutarse.
- **Eficiencia espacial:** cuánta memoria adicional necesita durante su ejecución.

Notación Big-O:

Es una herramienta que describe cómo crece el tiempo o el uso de memoria en función del tamaño de la entrada (n). No mide tiempo real, sino que establece una cota superior del crecimiento, es decir, cómo se comporta un algoritmo en el peor de los casos:

- O(1): tiempo constante
- O(log n): logarítmico
- O(n): lineal
- $O(n^2)$: cuadrático
- $O(2^n)$: exponencial
- O(n!): factorial

Medición empírica:

Para comprobar el rendimiento en la práctica, se puede usar una función como **time.time()** que mide el tiempo de ejecución en segundos.
Sin embargo, estos valores pueden variar según el sistema operativo, la versión del lenguaje, el entorno de ejecución e incluso el hardware. Por eso, los resultados empíricos deben interpretarse como complementarios al análisis teórico, no como reemplazo.

Importancia de Big-O frente al tiempo real:

Aunque medir el tiempo real ayuda a observar el comportamiento de un algoritmo en un caso concreto, no es suficiente para evaluar su eficiencia general.
La notación Big-O permite comparar algoritmos en términos de escalabilidad, sin depender de factores externos como el hardware o el lenguaje de programación. Es clave para tomar decisiones informadas cuando se trabaja con grandes volúmenes de datos o sistemas complejos.

CASO PRÁCTICO

Problema: Dadas dos listas de identificadores numéricos, una que representa usuarios ya registrados en la base de datos (**ids_registrados**) y otra que representa una nueva carga de usuarios desde un formulario (**ids_formulario**). El objetivo es verificar si alguno de los nuevos IDs ya está presente entre los registrados.

Se plantea una primera solución utilizando bucles anidados. El algoritmo recorre, mediante un primer bucle, la lista (**ids_registrados**) y, por cada uno de sus elementos, ejecuta un segundo bucle que recorre la lista (**id_formulario**). En cada iteración del segundo bucle se evalúa, mediante una estructura condicional **if**, si los valores de ambas listas son iguales. En caso de encontrar una coincidencia, la función finaliza de inmediato y retorna **True**, indicando la existencia de un identificador duplicado. Si no se encuentra ninguna coincidencia luego de recorrer ambas listas, se devuelve **False**.

```python
# Algoritmo con bucles anidados: compara cada elemento del formulario con cada ID registrado
def id_duplicado_bucles_anidados(ids_registrados, ids_formulario):
    for id1 in ids_registrados:
        for id2 in ids_formulario:
            if id1 == id2:              # Se ejecuta n x m veces en el peor de los casos
                return True
    return False
# Complejidad total O(n*m) -> convirtiendose en el peor de los casos a O(n^2)
```

Una vez implementada la solución inicial, se busca optimizar el algoritmo utilizando diccionarios.
Para ello, se crea un diccionario llamado (**registrados_dict**), donde cada clave representa un ID ya registrado, y el valor asociado puede ser simplemente **True**.

Esta estructura permite consultar de forma directa si un ID ya fue registrado, sin necesidad de recorrer toda la lista de manera secuencial.
A continuación, se recorre la segunda lista (**ids_formulario**) con un ciclo **for**, y mediante una condición **if**, se verifica si cada ID nuevo está presente como clave en el diccionario.

Esta operación resulta altamente eficiente, ya que los diccionarios en Python están implementados internamente como tablas hash, lo cual permite realizar búsquedas en tiempo constante promedio (O(1)).
De esta manera, se elimina por completo el uso de bucles anidados.
En cuanto se detecta una coincidencia, la función retorna **True** y finaliza su ejecución inmediatamente.

```python
# Algoritmo con dict: convierte la lista de registrados en un diccionario para verificar duplicados
def id_duplicado_dict(ids_registrados, ids_formulario):
    registrados_dict = {}
    for id in ids_registrados:                # O(n)
        registrados_dict[id] = True
    for id in ids_formulario:                 # O(m)
        if id in registrados_dict:                # O(1) ejecuta la búsqueda
            return True
    return False
# Complejidad total: O(n + m). Se reduce a O(n)
```

Luego se implementa una función auxiliar denominada (**medir_tiempo**), cuya finalidad es registrar el tiempo de ejecución de cada uno de los algoritmos analizados.

Esta función permite cuantificar de manera empírica el rendimiento de cada enfoque, lo cual resulta útil para complementar el análisis teórico previamente realizado mediante notación Big-O.

```python
# Función auxiliar para medir el tiempo de ejecución de cada algoritmo
def medir_tiempo(funcion, lista1, lista2):
    inicio = time.time()
    resultado = funcion(lista1, lista2)
    fin = time.time()
    return resultado, fin - inicio
```

Para evaluar la eficiencia de los algoritmos, se generaron listas de prueba con identificadores numéricos aleatorios de seis dígitos.
Esto se realizó mediante la función (**random.randint()**) dentro de una comprensión de listas, como puede observarse en el bloque de código.
La constante **n**, definida en 10.000, determina la cantidad de elementos de cada lista: (**ids_registrados**) e (**ids_formulario**).

A continuación, se ejecuta cada uno de los algoritmos, el de bucles anidados y el optimizado con diccionario utilizando como entrada estas dos listas.
Ambas ejecuciones son evaluadas mediante la función (**medir_tiempo**), que permite capturar el resultado de cada algoritmo (si hay duplicados o no) junto con el tiempo exacto que tarda en completarse.

Los resultados se imprimen por consola, mostrando para cada algoritmo si se detectaron IDs duplicados y cuánto tiempo tardó en procesar la información.
Esta metodología permite observar, de forma objetiva, cómo se comporta cada enfoque en términos de rendimiento cuando se enfrentan a un conjunto de datos de tamaño considerable.

```python
if __name__ == "__main__":
    # Generación de listas de prueba con IDs aleatorios de 6 dígitos
    n = 10000
    ids_registrados = [random.randint(100000, 999999) for _ in range(n)]
    ids_formulario = [random.randint(100000, 999999) for _ in range(n)]

    # Ejecución y medición del algoritmo lento (con posibles duplicados)
    resultado1, tiempo1 = medir_tiempo(id_duplicado_bucles_anidados, ids_registrados, ids_formulario)
    print(f"[Algoritmo bucles anidados] -- Hay duplicados?: {resultado1} -- Tiempo: {tiempo1:.6f} segundos")

    # Ejecución y medición del algoritmo rápido (con posibles duplicados)
    resultado2, tiempo2 = medir_tiempo(id_duplicado_dict, ids_registrados, ids_formulario)
    print(f"[Algoritmo con diccionario] -- Hay duplicados?: {resultado2} -- Tiempo: {tiempo2:.6f} segundos")
```

Por último, se generaron listas diseñadas específicamente para simular el peor caso posible, que no exista ningún ID duplicado entre los registros.
Para ello, se utilizaron rangos numéricos no superpuestos al construir cada lista, asegurando que no haya coincidencias entre los valores de (**ids_registrados**) e (**ids_formulario**).
Esta estrategia garantiza que los algoritmos deban recorrer por completo ambas listas sin encontrar coincidencias, lo que representa el escenario más costoso en términos de rendimiento.

Luego, al igual que en la prueba anterior, se ejecutan ambos algoritmos utilizando estas listas como entrada.
La función (**medir_tiempo**) permite medir de forma precisa cuánto tarda cada algoritmo en completarse bajo esta condición adversa.
Los resultados se imprimen en pantalla, mostrando si se detectaron duplicados (lo cual no ocurrirá en este caso) y el tiempo total de ejecución.

Este experimento permite observar con claridad cómo el algoritmo basado en bucles anidados se ve afectado negativamente por el tamaño de las listas, mientras que el algoritmo optimizado con diccionario mantiene un rendimiento estable incluso en el peor escenario.

```python
    # Generación de listas SIN duplicados para simular el peor caso
    ids_registrados = [random.randint(100000, 200000) for _ in range(n)]
    ids_formulario = [random.randint(300000, 400000) for _ in range(n)]

    # Ejecución y medición del algoritmo lento (sin duplicados)
    resultado1, tiempo1 = medir_tiempo(id_duplicado_bucles_anidados, ids_registrados, ids_formulario)
    print(f"[Algoritmo bucles anidados] -- Hay duplicados?: {resultado1} -- Tiempo: {tiempo1:.6f} segundos")

    # Ejecución y medición del algoritmo rápido (sin duplicados)
    resultado2, tiempo2 = medir_tiempo(id_duplicado_dict, ids_registrados, ids_formulario)
    print(f"[Algoritmo con diccionario] -- Hay duplicados?: {resultado2} -- Tiempo: {tiempo2:.6f} segundos")
```

METODOLOGÍA UTILIZADA

1) **Reformulación del problema:**
   Se planteó el problema como la detección de identificadores numéricos duplicados entre dos listas, representando registros antiguos (**ids_registrados**) y nuevos (**ids_formulario**).

2) **Análisis teórico de complejidad algorítmica:**
   Se evaluaron dos enfoques:
   - **id_duplicado_bucles_anidados**: presenta una complejidad temporal de $O(n^2)$, debido al uso de bucles anidados que comparan todos los elementos de ambas listas.
   - **id_duplicado_dict**: utiliza un diccionario para realizar búsquedas en tiempo constante, alcanzando una complejidad $O(n + m)$, que se simplifica a $O(n)$ cuando ambas listas tienen tamaños similares.

3) **Implementación en Python:**
   Ambos algoritmos fueron codificados en Python, junto con una función auxiliar (**medir_tiempo**) que permite evaluar el tiempo real de ejecución.

4) **Generación de datos de prueba:**
   Se generaron listas de 10.000 IDs aleatorios de seis dígitos mediante la función (**random.randint()**), simulando un entorno de alta carga.

5) **Ejecución y medición empírica:**
   Se ejecutaron ambos algoritmos con las mismas listas y se midió su rendimiento usando (**time.time()**), que calcula el tiempo transcurrido en segundos.

6) **Comparación de resultados:**
   Finalmente, se compararon los tiempos de ejecución obtenidos en distintos escenarios, incluyendo uno con posibles duplicados y otro que simula el peor caso (sin duplicados), para observar de manera empírica la diferencia de eficiencia entre ambas soluciones.

RESULTADOS OBTENIDOS

Ambos algoritmos implementados arrojaron la misma respuesta lógica ante los distintos conjuntos de datos: identificaron correctamente la presencia o ausencia de IDs duplicados.

Sin embargo, las diferencias en rendimiento fueron notorias.
El algoritmo basado en bucles anidados presentó tiempos de ejecución considerablemente más altos, especialmente en el peor caso (listas sin elementos en común), donde puede tardar varios segundos en completarse debido a la alta cantidad de comparaciones.

En contraste, el algoritmo optimizado con diccionario respondió casi de forma instantánea, incluso con listas de gran tamaño, completando su ejecución en milisegundos.

Este comportamiento observado en las pruebas empíricas confirma lo anticipado por el análisis teórico de complejidad algorítmica, validando que una buena elección de estructura de datos y enfoque puede mejorar significativamente el rendimiento de un programa.

CONCLUSIONES

El análisis algorítmico es una herramienta clave para anticipar el rendimiento de un programa, incluso sin necesidad de ejecutarlo.
Si bien los tiempos de ejecución reales aportan información útil en contextos específicos, la notación Big-O permite evaluar la escalabilidad de una solución ante volúmenes crecientes de datos.

En este trabajo, se demostró que un algoritmo con complejidad **O(n)** basado en un diccionario supera ampliamente a otro de **O(n²)** que utiliza bucles anidados.
Esto validó la importancia de seleccionar estructuras de datos adecuadas, como diccionarios implementados mediante tablas hash, que permiten búsquedas en tiempo constante.

Analizar la complejidad algorítmica desde el inicio del desarrollo es esencial, especialmente en proyectos que procesan grandes cantidades de datos, como validaciones masivas, control de duplicados o sincronización de registros.

BIBLIOGRAFÍA

- Grokking Algorithms, Aditya Bhargava.
- Documentación oficial Python time: https://docs.python.org/3/library/time.html
- Big O Cheat Sheet: https://www.bigocheatsheet.com/

ANEXOS

- Repositorio: https://github.com/EmaRivero/trabajo-integrador-programacion.git
- Video explicativo: https://www.youtube.com/watch?v=xH90jmq37H0
- Presentación: https://drive.google.com/file/d/1yyXApN7CBWTsbftrhyOw49wGpOl6-8lg/view?usp=sharing