Trabajo Integrador: Análisis de Algoritmos en Python

Integrantes

Matías G. Perdigues	Alejandro Raúl Pereyra
30.408.387	32.260.891
matperdigues@gmail.com	escorturo86@gmail.com

Materia: Programación I

Profesor: Nicolás Quirós / Carla Bustos

Fecha de Entrega: 09 de junio de 2025

Índice

Introducción

El presente trabajo tiene como finalidad analizar algoritmos creados para una tarea concreta y específica para obtener estimaciones teóricas y empíricas sobre el consumo de recursos en tiempo y espacio y determinar así sus comportamientos e identificar el algoritmo más eficiente.

Para lograr lo mencionado anteriormente, haremos un recorrido conceptual sobre análisis de algoritmos e implementaremos en lenguaje Python tres funciones que realizarán la misma tarea a fin de medir su eficiencia. Nuestro análisis se apoyará

tanto en el método empírico, demostrando los registros de tiempo obtenidos para cada función, y a través del método teórico que permitirá obtener la complejidad matemática de cada algoritmo.

Marco Teórico

Algoritmo

Un Algoritmo es un conjunto ordenado y finito de operaciones que permiten encontrar la solución a un problema. En el contexto de la computación un algoritmo se usa para denominar a la secuencia de pasos a seguir para resolver un problema usando una computadora. En definitiva, un algoritmo es un procedimiento para resolver un problema cuyos pasos son concretos y no ambiguos. El algoritmo debe ser correcto, de longitud finita y debe entregar un resultado para todas las entradas.

Análisis de algoritmos

El análisis de algoritmos provee estimaciones acerca de los recursos que necesita cualquier algoritmo que resuelva un problema computacional dado. Teniendo en cuenta que para ciertos problemas existe más de una solución posible, estas estimaciones se convierten en una herramienta importante para encontrar algoritmos eficientes en la resolución de tareas o problemas. Un algoritmo es eficiente cuando logra llegar a sus objetivos planteados utilizando la menor cantidad de recursos posibles, es decir, minimizando el uso memoria, de pasos y de esfuerzo.

Análisis empírico

El análisis empírico de algoritmos consiste en ejecutar una implementación concreta del algoritmo sobre entradas específicas, generalmente crecientes, y medir experimentalmente su rendimiento en tiempo y espacio, con el fin de observar su comportamiento práctico y compararlo con otros algoritmos. Sin embargo, es importante tener en cuenta varios factores. En primer lugar, el tiempo de un sistema no suele ser muy preciso, y podrían obtenerse resultados ligeramente diferentes al ejecutar repetidamente el mismo programa con las mismas entradas. Una solución es realizar varias mediciones de este tipo y luego tomar su promedio como punto de observación de la muestra. En segundo lugar, dependiendo de la

velocidad de la computadora en la que se esté ejecutando el algoritmo, el tiempo de ejecución puede no registrarse y reportarse como cero. Por lo tanto, medir el tiempo de ejecución física tiene varias desventajas ya que el tiempo de ejecución física proporciona información muy específica sobre el rendimiento de un algoritmo en un entorno computacional particular.

Análisis teórico (asintótico)

El objetivo del análisis de algoritmos es comprender cómo crece la demanda de recursos, principalmente tiempo y espacio, a medida que aumenta el tamaño del problema. Esto se conoce como eficiencia asintótica del algoritmo. Se denomina así porque se estudia el comportamiento del algoritmo cuando el tamaño de entrada tiende a valores muy grandes, es decir, en el "límite". Este enfoque permite comparar algoritmos de forma independiente del hardware o lenguaje de programación utilizado, dejando de lado las desventajas detalladas en la descripción del análisis empírico.

En este trabajo emplearemos la notación Big-O para describir la complejidad de los algoritmos. Esta notación expresa cómo escala el tiempo de ejecución (o el uso de memoria) en función del tamaño de entrada n, lo que permite representar matemáticamente su eficiencia. La notación Big-O abstrae los detalles de implementación, sistema operativo o entorno, y se enfoca en el comportamiento fundamental del algoritmo cuando los datos crecen indefinidamente.

- **Ordenes de complejidad**
    a) O(1) – Constante: Los algoritmos de complejidad constate ejecutan siempre el mismo número de pasos sin importar cuan grande sea n.

    b) O(log n) – Logarítmica: estos algoritmos generalmente son algoritmos que resuelven un problema transformándolo en problemas menores como es el caso de búsqueda binaria en estructuras ordenadas.

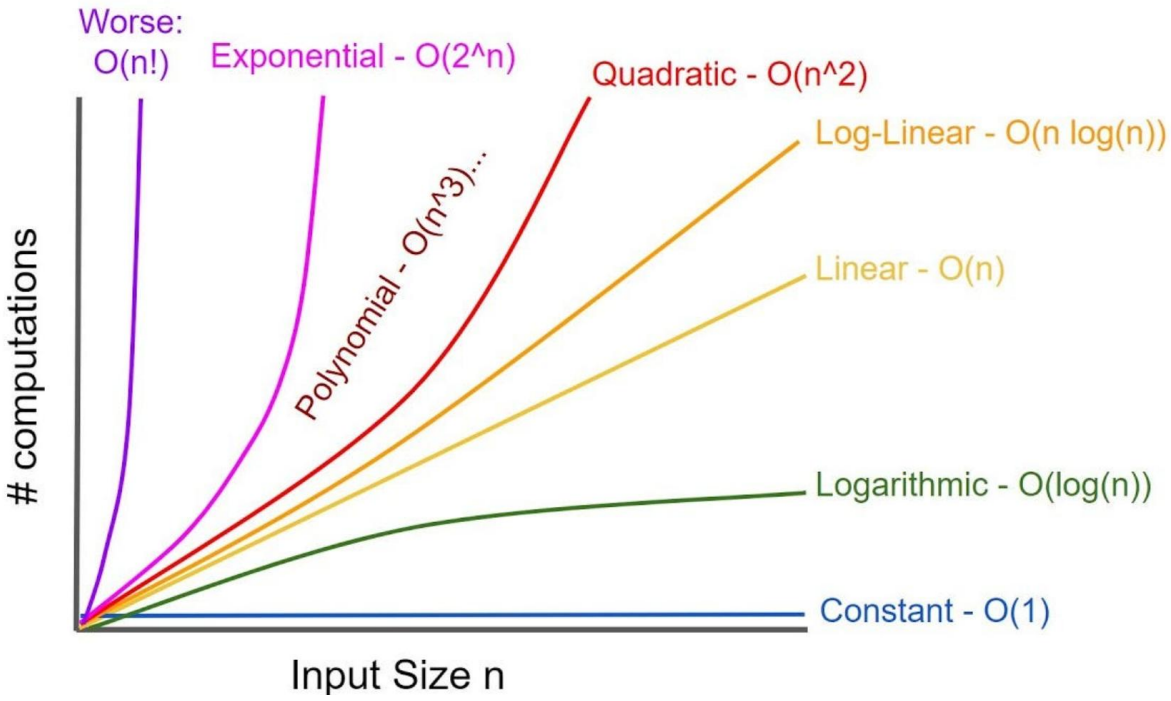
    c) O(n) – Lineal: los algoritmos de complejidad lineal generalmente tratan de manera constante cada n del problema por lo que si n dobla su tamaño el algoritmo también dobla el número de pasos. El tiempo de ejecución es proporcional al tamaño. Un ejemplo de esto es el recorrido completo de una lista.

d) $O(n^2)$ – Cuadrática: los algoritmos de complejidad cuadrática aparecen cuando los datos se procesan por parejas, en la mayoría de los casos en bucles anidados.

e) O(n log n) – Cuasilineal: Los algoritmos de complejidad “n log n” generalmente dividen un problema en problemas más sencillos de resolver para finalmente combinar las soluciones obtenidas.

f) $O(2^n)$ – Exponencial: Los algoritmos de complejidad exponencial no son útiles desde el punto de vista práctico, aparecen cuando un problema se soluciona empleando fuerza bruta.

g) O(n!) – Factorial: estos algoritmos generalmente aparecen en problemas de permutación completa



Caso Práctico

El trabajo consistió en el diseño de tres funciones que tuviesen como tarea asignada la comparación de dos listas, ingresadas como parámetros, para incorporar en una tercera lista los elementos comunes. Para estudiar la eficiencia de cada una de estas funciones, el tamaño de ambas listas se duplica mediante un ciclo for.

La primera de estas funciones compara los elementos de ambas listas mediante ciclos anidados, evaluando cada elemento de la primera lista contra todos los elementos de la segunda. Esto genera una búsqueda que verifica todas las posibles combinaciones entre ambas listas. Como resultado, la complejidad temporal de esta función es $O(n^2)$, lo que implica un crecimiento cuadrático del tiempo de ejecución a medida que aumenta el tamaño de entrada.

```
import time


def busqueda_ciclos(lista1,lista2):
    start=time.time()
    repetidos=[]
    for i in lista1:
        for x in lista2:
            if i == x:
                repetidos.append(i)
    end=time.time()
    return (end-start)*1000


for n in [100,200,400,800,1600,3200,6400,12800,25600]:
    print(busqueda_ciclos(list(range(n)),list(range(n))))
```

La segunda función realiza la comparación utilizando el operador de pertenencia in, que verifica si un elemento existe dentro de una lista. Internamente, este operador realiza una búsqueda lineal, evaluando secuencialmente cada elemento de la lista hasta encontrar una coincidencia o recorrer toda la lista. La ejecución de este operador tiene una complejidad temporal de O(n), lo que sumado a la

complejidad temporal del ciclo que lo contiene da como resultado una complejidad temporal total de $O(n^2)$. Por lo tanto, esta segunda función también presenta un crecimiento cuadrático del tiempo a medida que aumentan los datos de entrada.

```python
import time

def busqueda_in(lista1,lista2):
    start=time.time()
    repetidos=[]
    for z in lista1:
        if z in lista2:
            repetidos.append(z)
    end=time.time()
    return (end-start)*1000

for n in [100,200,400,800,1600,3200,6400,12800,25600]:
    print(busqueda_in(list(range(n)),list(range(n))))
```

Por último, la tercera función optimiza la comparación entre listas mediante el uso de un conjunto (set). En primer lugar, convierte la segunda lista ingresada como parámetro en un conjunto, lo cual tiene un costo de O(n). Luego, itera sobre los elementos de la primera lista y utiliza el operador de pertenencia in para verificar si cada elemento existe en el conjunto. A diferencia de las listas, los conjuntos permiten búsquedas promedio en O(1) gracias a su implementación basada en tablas hash.

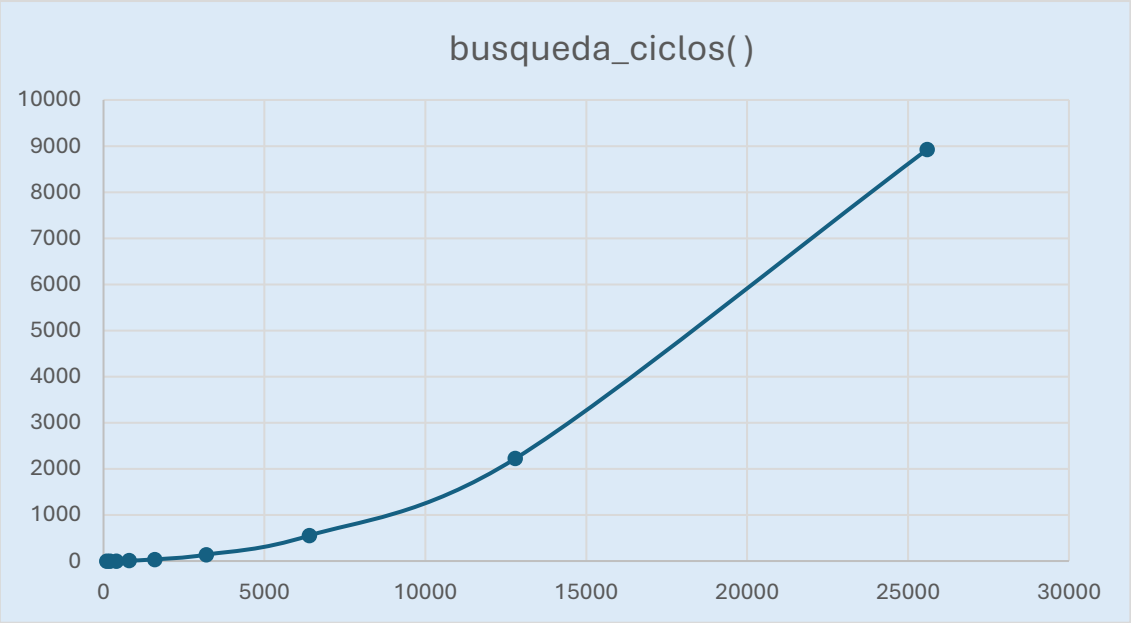
```python
def busqueda_set(lista1, lista2):
    start=time.time()
    set2 = set(lista2)
    repetidos = []
    for z in lista1:
        if z in set2:
            repetidos.append(z)
    end=time.time()
    return (end-start)*1000

for n in [100,200,400,800,1600,3200,6400,12800,25600]:
    print(busqueda_set(list(range(n)),list(range(n))))
```

Resultados Obtenidos

A continuación, se presentan los registros obtenidos durante las comprobaciones empíricas mediante tablas de relación *tamaño de entrada (n) - tiempos de procesamiento*, con sus correspondientes gráficos de comportamiento.

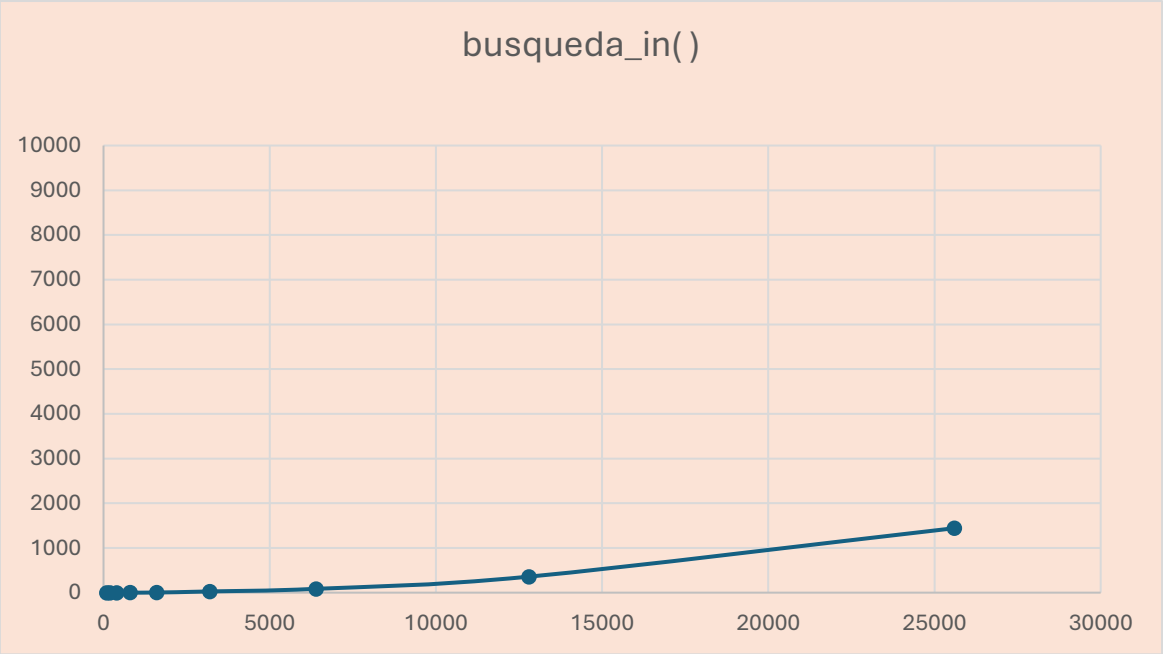
Función busqueda_ciclos( )

busqueda_ciclos( )	
Tamaño (n)	Tiempo
100	0
200	0,689029694
400	2,022266388
800	9,663581848
1600	36,74268723
3200	139,1069889
6400	553,5547733
12800	2224,391699
25600	8927,782774



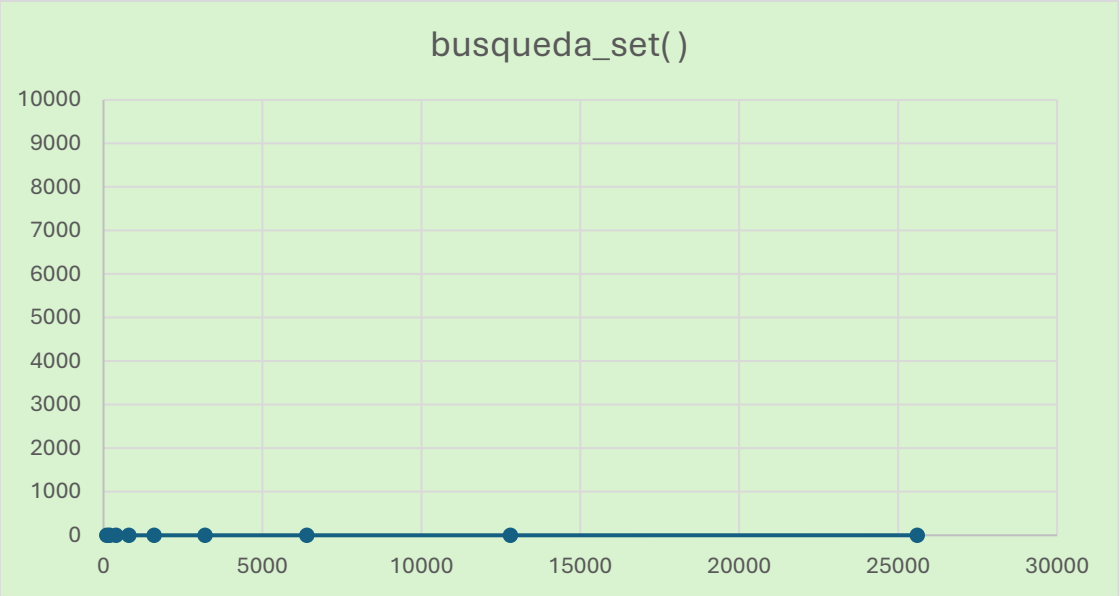
Función busqueda_in( )

busqueda_in( )	
Tamaño (n)	Tiempo
100	0
200	0
400	0
800	1,007795334
1600	2,996206284
3200	27,35042572
6400	86,11011505
12800	358,1047058
25600	1441,347122



Función busqueda_set( )

busqueda_set( )	
Tamaño (n)	Tiempo
100	0
200	0
400	0
800	0
1600	0
3200	0
6400	0
12800	0
25600	1,010417938



Conclusiones

A partir de estos resultados, podemos concluir que la comparación de listas mediante ciclos anidados es el camino menos eficiente. Por tratarse de una función con una complejidad $O(n^2)$ por el peso relativo que tiene el funcionamiento de cada ciclo for, el consumo de tiempo experimenta un crecimiento cuadrático a medida que aumenta el tamaño de entrada.

Si comparamos las funciones búsqueda_ciclos() y búsqueda_in(), resulta llamativo que, a pesar de presentar ambas funciones una complejidad cuadrática, la segunda sea bastante más eficiente. Esto último puede explicarse debido a que la implementación de la operación in en listas está hecha en lenguaje C, y suele ser ligeramente más rápida que un ciclo for escrito estrictamente en Python, al evitarse la sobrecarga de la interpretación línea a línea. Esto no cambia la complejidad, pero sí reduce la variable de tiempo.

Por último, la evidente eficiencia de la función búsqueda_set() puede explicarse a partir de que optimiza la comparación entre listas mediante el uso de un conjunto (set). La verificación de pertenencia se realiza en tiempo constante, gracias a su implementación basada en tablas hash. El bucle principal incorpora una complejidad de O(n) que sumada al costo de construcción del conjunto O(n), da una complejidad total de O(n). Por lo tanto, esta versión representa una mejora significativa en eficiencia frente a las funciones anteriores, mostrando un crecimiento lineal del tiempo de ejecución a medida que aumentan los datos de entrada.

Bibliografía

- Análisis de algortimos – Prof. Edgardo Marínez, 2019.
- Introduction to algorithms - Cormen, T. Leiserson, 2003.
- Introducción al Análisis de Algoritmos - Ariel Enferrel
- Análisis Teórico de Algoritmos - Ariel Enferrel