

CARPETA DE ANÁLISIS DE ALGORITMOS EN PYTHON

ALUMNOS

Ortellado, Karen (ortelladokaren@gmail.com)

Nicolas, Rodrigo (renicolas7@gmail.com)

MATERIA

Programación I

PROFESORA:

Julieta Trapé

TUTOR:

Marcos Vega

FECHA DE ENTREGA:

Lunes 9 de junio de 2025

Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Resultados Obtenidos
5. Conclusiones
6. Bibliografía
7. Anexos
8. Video explicativo

INTRODUCCIÓN:

El presente proyecto integrador de la materia Programación I, aborda al “Análisis de Algoritmos” como su tema de estudio, ya que es una rama de la programación que invita a poner en práctica a los estudiantes, nuevas y diferentes formas de analizar código, midiendo y comparando la eficiencia de los programas teniendo en cuenta tiempos de ejecución y uso de memoria. El hecho de comprender cómo se comportan los algoritmos frente al aumento de datos es crucial para el desarrollo de aplicaciones, garantizando eficiencia y escalabilidad de nuestras soluciones.

En paralelo, los conceptos de eficiencia temporal y espacial serán implementados a través de ejercicios prácticos y luego se presentarán las conclusiones.

Finalmente, es prioridad que los alumnos logren, a través de la siguiente investigación, poder explicar los conceptos fundamentales, demostrar a través de ejemplos prácticos la teoría y poder desenvolverse de forma natural.

MARCO TEÓRICO:

- ¿Qué es un algoritmo?

Es un conjunto de instrucciones precisas y ordenadas que resuelven un problema en específico. Es correcto, robusto y eficiente. Busca producir el resultado deseado, funcionando correctamente en diferentes condiciones y usos.

- ¿Qué es el análisis de algoritmos?

Es el estudio formal de rendimiento de algoritmos que busca optimizar el **Tiempo de ejecución** (Eficiencia temporal), el **Uso de memoria** (Eficiencia espacial) y otros recursos como ancho de banda o energía consumida

- ¿Por qué es importante analizarlos?

Es importante analizarlos para comparar la eficiencia de diferentes soluciones para un mismo problema. Para optimizar el uso de la memoria y el tiempo de ejecución, fundamental para el rendimiento de aplicaciones (como en sistemas de gran escala). Y además nos ayuda a poder elegir que algoritmo es mas adecuado para cada situación, garantizando **eficiencia** y **escalabilidad** en nuestras soluciones.

ANÁLISIS EMPÍRICO	ANÁLISIS TEÓRICO
<p>Implica poner a prueba cada tipo de solución en código y probarlas dentro de un mismo ambiente controlado, para que las condiciones de comparación sean similares.</p> <p>Interesa saber el tiempo de ejecución que le toma al algoritmo analizar diferentes tamaños de entradas.</p>	<p>Se enfoca en obtener una función matemática $T(n)$ que describa el número de operaciones que un algoritmo realiza para una entrada n. Esta función nos da una visión general de cómo funciona el algoritmo. Lo que permite identificar y comparar algoritmos independientemente del software y del hardware.</p>
VENTAJAS	VENTAJAS
<ul style="list-style-type: none"> • Proporciona resultados reales en un entorno específico. • Detecta constantes ocultas y factores del sistema. • Útil para validar el análisis teórico 	<ul style="list-style-type: none"> • Independiente del hardware / software • Permite comparar algoritmos sin implementarlos • Identifica el comportamiento en el peor caso • Más económico que el empírico
DESVENTAJAS	DESVENTAJAS

<ul style="list-style-type: none">• Dependiente del hardware y software usado.• Requiere la implementación completa del algoritmo.• Puede no cubrir todos los casos posibles.• Costoso en tiempo y recursos	<ul style="list-style-type: none">• Ignora constantes y factores de menor orden.• Puede no reflejar exactamente el rendimiento real.• Requiere conocimientos matemáticos.
--	---

Conceptos Clave:

• Notación Big-O: Describe cómo crece el tiempo de ejecución de un algoritmo en función del tamaño de la entrada de datos. Generalmente se usa para expresar el peor caso de crecimiento del algoritmo (por ejemplo:

$O(n)$, $O(\log n)$, $O(n^2)$).

- Ayuda a predecir escalabilidad
- Permite elegir el algoritmo más eficiente para grandes volúmenes de datos
- Fundamental en el diseño de sistemas a gran escala

• Tiempo de ejecución real: Se puede medir usando módulos como time en Python.

• Complejidad espacial: Se refiere a la cantidad de memoria adicional que utiliza el


Algoritmo.

CASO PRÁCTICO:

Elegimos como caso práctico el análisis de un código en Python que contiene una lista de datos desordenados (nombres y apellidos), utilizar un algoritmo de ordenamiento y analizar cuál es más eficiente para éste caso.

RESULTADOS OBTENIDOS:

- ANÁLISIS TEÓRICO: BUBBLE SORT:



```
def bubble_sort(array):  
    n = len(array)  
    for i in range(n):  
        swapped = False  
        for j in range(0, n-i-1):  
            if array[j][0] > array[j+1][0]:  
                array[j], array[j+1] = array[j+1], array[j]  
                swapped = True # 1  
        if (swapped == False):  
            break
```

Podemos determinar que este algoritmo será de la notación Big $O(n^2)$ ya que cuenta con dos for loops anidados en su código. Por lo tanto el tiempo de ejecución crece proporcionalmente al tamaño del cuadrado de la entrada.

MERGE SORT:

```
def merge_sort(array):
    if len(array) > 1:
        # Divide el array en dos mitades
        left_array = array[:len(array)//2]
        right_array = array[len(array)//2:]

        # Recursivamente
        merge_sort(left_array)
        merge_sort(right_array)

    # Merge
    # Comparamos los elementos
    i = 0 # compara el elemento de la izquierda del array izquierdo
    j = 0 # compara el elemento de la izquierda del array derecho
    k = 0 # el index del array mergeado
    while i < len(left_array) and j < len(right_array):
        # Dividie entre apellido y nombre y compara los apellidos
        if left_array[i].split()[0] < right_array[j].split()[0]:
            array[k] = left_array[i]
            i += 1
        else:
            array[k] = right_array[j]
            j += 1
        k += 1

    while i < len(left_array):
        array[k] = left_array[i]
        i += 1
        k += 1

    while j < len(right_array):
        array[k] = right_array[j]
        j += 1
        k += 1
```

Se trata de un algoritmo “divide y conquista”. Soluciona los problemas de manera recursiva hasta que sean más simples de solucionar. Ya que este algoritmo funciona de manera recursiva dividiendo el arreglo en mitades hasta que cada subarreglo tenga un solo elemento, y luego combina estas sublistas en una única lista, el tiempo de ejecución será de tipo $O(n \cdot \log(n))$. Es decir, que podemos esperar un tiempo de ejecución más corto.

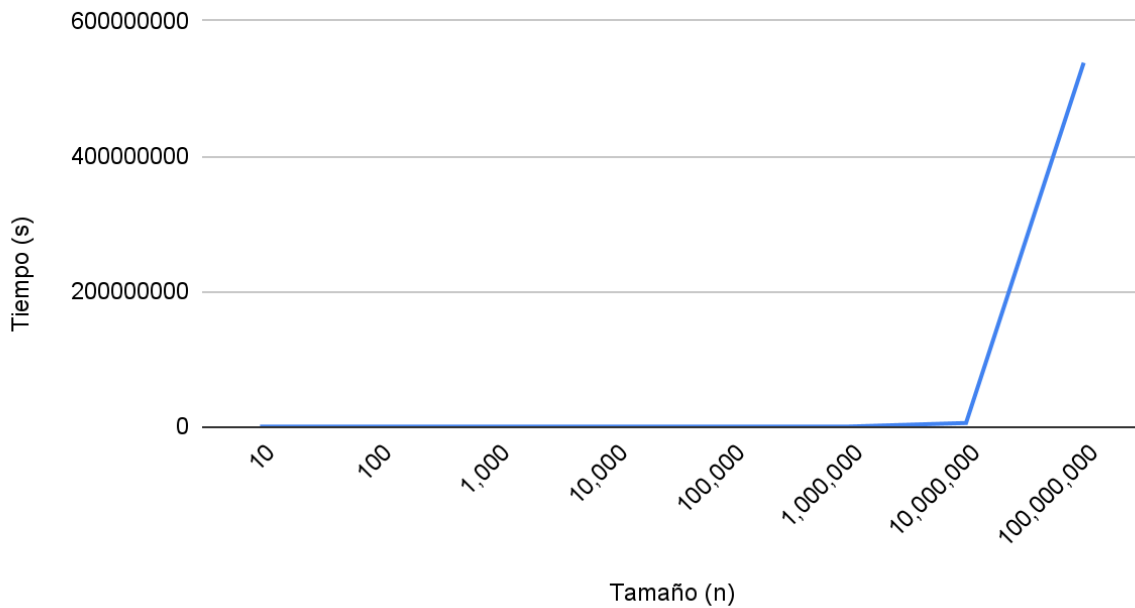
QUICKSORT:

```
def quicksort(array, left, right):  
    if left < right:  
        partition_pos = partition(array, left, right)  
        quicksort(array, left, partition_pos - 1)  
        quicksort(array, partition_pos + 1, right)  
  
def partition(array, left, right):  
    i = left  
    j = right - 1  
    pivot = array[right]  
  
    while i < j:  
        while i < right and array[i] < pivot:  
            i += 1  
        while j > left and array[j] >= pivot:  
            j -= 1  
        if i < j:  
            array[i], array[j] = array[j], array[i]  
    if array[i] > pivot:  
        array[i], array[right] = array[right], array[i]  
  
    return i
```

Este algoritmo también funciona cómo divide y conquista, pero no divide en partes iguales. Se elige el punto de pivote de partida para comparar los elementos. Todos los elementos menores al pivote quedan a la izquierda y todos los elementos mayores o iguales a la derecha. Se aplica de manera recursiva por lo que tenemos $O(\log n)$. Luego el código de la función `partition` se encarga reorganizar los elementos alrededor del pivote en tiempo $O(n)$. Por lo que el tiempo final de complejidad de este algoritmo será $O(n \cdot \log n)$. Es decir el producto del tiempo del número de niveles de recursión por el tiempo de trabajo realizado por nivel. En el mejor caso: el pivote divide a la lista en partes casi iguales, y el tiempo de ejecución es $O(n \cdot \log n)$. En el peor caso: Se elige el último elemento cómo punto de pivote y la lista ya está ordenada lo que lleva a una complejidad $O(n^2)$. Para evitar esto, se debería elegir el pivote de manera aleatoria. Lo cual mejora el rendimiento promedio del algoritmo. Podemos deducir entonces que este algoritmo será más eficiente para los casos promedios pero será igual de eficiente que bubble sort en el peor de los casos.

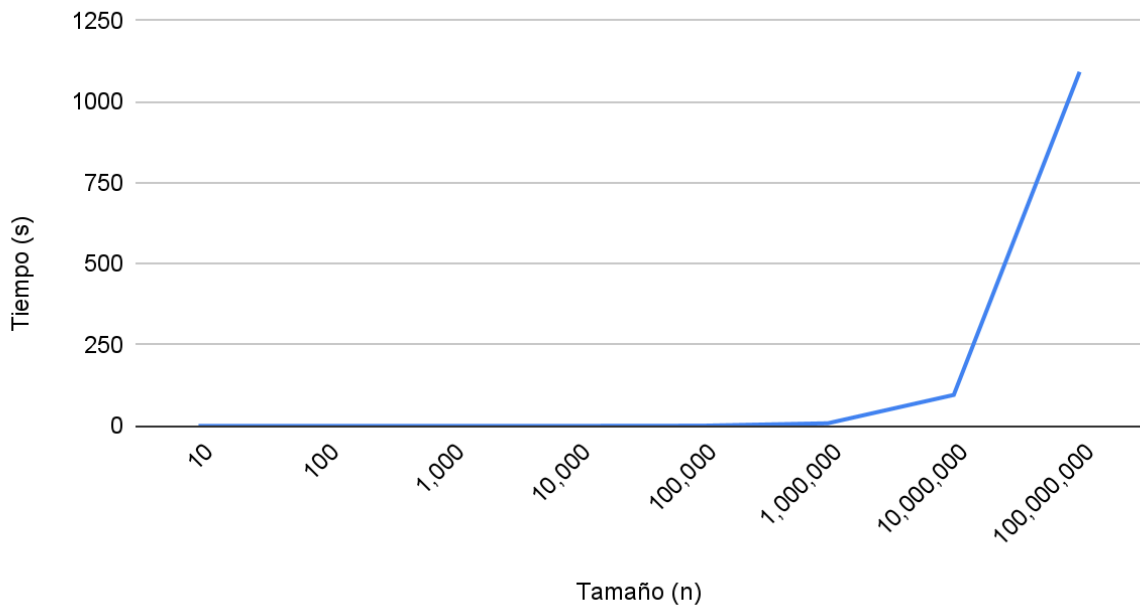
ANÁLISIS EMPÍRICO:

Bubble Sort - Tiempo (s) contra Tamaño (n)



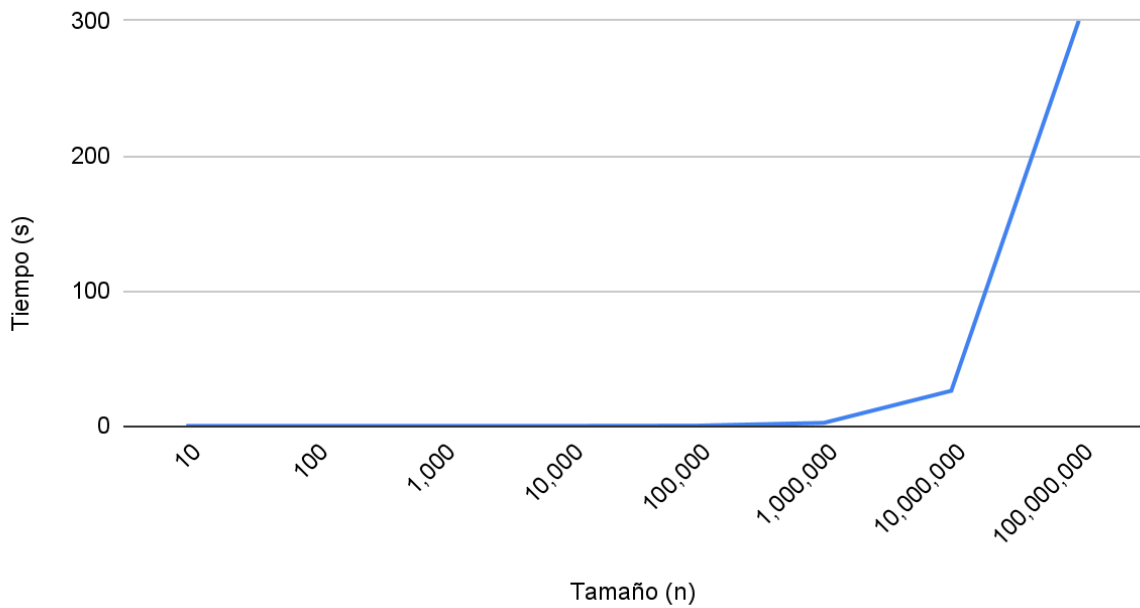
Tamaño (n)	Tiempo (s)
10	0.000012
100	0.000471
1000	0.052448
10,000	5.380653
100,000	538 (9 Minutos)
1,000,000	53800 (14 horas)
10,000,000	5380000 (62 días)
100,000,000	538000000 (17 años)

Merge Sort - Tiempo (s) contra Tamaño (n)



Tamaño (n)	Tiempo (s)
10	0.000026
100	0.000273
1000	0.004026
10,000	0.054564
100,000	0.682
1,000,000	8.19
10,000,000	95.55(1 minuto y medio aprox.)
100,000,000	1092(18 minutos)

QuickSort - Tiempo (s) contra Tamaño (n)



Tamaño (n)	Tiempo (s)
10	0.000012
100	0.000080
1000	0.001175
10,000	0.015049
100,000	0.1879
1,000,000	2.25
10,000,000	26
100,000,000	300 (5 minutos)

CONCLUSIONES:

El análisis teórico de algoritmos permite estimar su eficiencia sin necesidad de ejecutarlos, lo que resulta fundamental para elegir la mejor solución antes de comenzar el desarrollo de un programa. Este enfoque nos brinda una medida independiente del software o hardware utilizado, lo que permite predecir el comportamiento del algoritmo en función del tamaño de los datos de entrada.

Gracias a este análisis, podemos identificar la complejidad temporal y espacial —representada comúnmente con notación Big-O— y anticipar cuál algoritmo será más adecuado para un determinado problema. Aunque el análisis teórico puede resultar más abstracto y complejo, es esencial para comprender el funcionamiento interno del algoritmo y su escalabilidad.

Por otro lado, el análisis empírico complementa al teórico al medir los tiempos de ejecución reales con diferentes volúmenes de datos, permitiendo validar los resultados obtenidos teóricamente. Ambos enfoques, combinados, ofrecen una visión completa que guía el diseño y la implementación de soluciones más eficientes.

En definitiva, el análisis de algoritmos es una herramienta clave para todo desarrollador, ya que permite optimizar el rendimiento del software desde sus primeras etapas.

BIBLIOGRAFÍA:

- Python Software Foundation. (2025). Python 3 Documentation.
<https://docs.python.org/3/>
- Faker Library. (2025). Faker Documentation.
<https://faker.readthedocs.io/en/master/>
- Introducción al análisis de algoritmos. (2025).
<https://www.youtube.com/watch?v=KdJnKVXm1A>
- Introducción al análisis teórico. (2025).
<https://www.youtube.com/watch?v=rdRrKIKP9T8>
- Introducción a Big O. (2025).
<https://www.youtube.com/watch?v=j8PqpS21Gpo>
- Tipos de órdenes de Big O. (2025).
<https://www.youtube.com/watch?v=CooM8LMLp9s>
- Bubble Sort Algorithm. (2025).
<https://www.geeksforgeeks.org/bubble-sort-algorithm/>
- Bubble Sort Algorithm Explained. (2025).
https://www.youtube.com/watch?v=g_xesqdQqvA
- Merge Sort in Python Explained. (2025).
<https://www.youtube.com/watch?v=cVZMah9kEjl>
- Quicksort in Python Explained. (2025).
<https://www.youtube.com/watch?v=9KBwdDEwal8>

ANEXOS:

- Código fuente. <https://github.com/salocin-95/integrador>

VIDEO EXPLICATIVO:

 IntegradorProgramacion