



Trabajo Práctico Integrador

## **Algoritmos de ordenamiento**

**Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.**

### **Programación 1**

Alumnos:

Ignacio Salazar – ignaciosalazarg86@gmail.com

Lautaro Zullo – zullolau@gmail.com

Materia: Programación 1

Profesor: AUS Bruselario, Sebastián

Fecha de Entrega: 09 de junio de 2025

# Índice

<b>Índice.....</b>	<b>1</b>
<b>Introducción.....</b>	<b>2</b>
<b>Marco Teórico.....</b>	<b>3</b>
Bubble sort.....	3
Insertion sort.....	3
Selection sort.....	3
Quicksort.....	3
Tiempos de los algoritmos.....	3
<b>Caso Práctico.....</b>	<b>4</b>
Función del algoritmo Bubble Sort:.....	4
Función del algoritmo QuickSort:.....	5
<b>Metodología Utilizada.....</b>	<b>6</b>
<b>Resultados Obtenidos.....</b>	<b>6</b>
<b>Conclusiones.....</b>	<b>6</b>
<b>Bibliografía.....</b>	<b>7</b>
Anexos.....	7

# Introducción

En el presente trabajo práctico integrador abordamos el estudio comparativo de dos algoritmos de ordenamiento: **Bubble Sort** y **Quicksort**. El objetivo principal fue analizar el rendimiento de ambos métodos mediante la medición de los tiempos de ejecución al ordenar listas de números aleatorios de diferentes tamaños.

Para ello, desarrollamos un algoritmo en Python de forma modular, que genera listas aleatorias y aplica ambos algoritmos de ordenamiento, registrando el tiempo que tarda cada uno en completarse. El experimento se repite múltiples veces, incrementando progresivamente el tamaño de las listas, hasta detectar en qué punto **Quicksort** comienza a superar en eficiencia a Bubble Sort.

Este trabajo busca no solo comparar el comportamiento práctico de estos algoritmos, sino también reforzar los conceptos teóricos de eficiencia y su aplicación en problemas concretos de programación.

# Marco Teórico

En computación y matemáticas un algoritmo de ordenamiento nos permite poner elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación o reordenamiento de la entrada que satisfaga la relación de orden dada.

Algunos de los algoritmos de ordenamiento más conocidos son:

## Bubble sort

Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

## Insertion sort

El Insertion Sort o Ordenamiento por Inserción, divide el conjunto de elementos en una parte ordenada y otra desordenada. Toma un elemento de la parte desordenada y lo inserta en la posición correcta en la parte ordenada. Repite este proceso hasta que todos los elementos estén ordenados.

## Selection sort

Selection Sort o Ordenamiento por Selección, busca el elemento más pequeño en el conjunto de elementos y lo coloca en la posición correcta. Luego, busca el siguiente elemento más pequeño y lo coloca en la siguiente posición correcta. Repite este proceso hasta que todos los elementos estén ordenados.

## Quicksort

Quicksort o Ordenamiento Rápido, elige un elemento llamado "pivote" y divide el conjunto en dos subconjuntos, uno con elementos menores que el pivote y otro con elementos mayores. Luego, aplica el mismo proceso de forma recursiva en cada uno de los subconjuntos. Este algoritmo también utiliza la estrategia de divide y vencerás.

## Tiempos de los algoritmos

Algoritmo	Mejor caso	Peor caso	Caso promedio	Eficiencia	Uso Recomendado
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	Baja	Listas pequeñas o casi ordenadas
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	Baja	Listas pequeñas
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	Media	Listas pequeñas o casi ordenadas
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	Alta	Listas grandes (evitar peor caso)

## Caso Práctico

Con el objetivo de observar en la práctica el comportamiento de los algoritmos de ordenamiento, se implementó un programa en Python que compara el tiempo de ejecución de Bubble Sort y Quicksort sobre listas de números aleatorios. La lógica consiste en generar listas de tamaño creciente y aplicar ambos algoritmos para medir cuánto tiempo tardan en ordenar la misma lista. El proceso se repite hasta encontrar el punto en el que Quicksort supera consistentemente a Bubble Sort en eficiencia. A continuación, se presenta el código de las dos funciones utilizadas en la comparación de eficiencia para llevar a cabo esta experimentación:

### Función del algoritmo Bubble Sort:

```
def bubble_sort(arr):  
    for n in range(len(arr) - 1, 0, -1):  
        swapped = False  
        for i in range(n):  
            if arr[i] > arr[i + 1]:  
                arr[i], arr[i + 1] = arr[i + 1], arr[i]  
                swapped = True  
        if not swapped:  
            break
```

## Función del algoritmo QuickSort:

```
def quicksort(arr):  
    if len(arr) <= 1:  
        return arr  
    else:  
        pivot = arr[0]  
        less = [x for x in arr[1:] if x <= pivot]  
        greater = [x for x in arr[1:] if x > pivot]  
        return quicksort(less) + [pivot] + quicksort(greater)
```

## Metodología Utilizada

La elaboración del trabajo se realizó en las siguientes etapas:

- Metodología Utilizada
- Recolección de información teórica en documentación confiable.
- Implementación en Python de los algoritmos estudiados.
- Pruebas con diferentes conjuntos de datos.
- Registro de resultados y validación de funcionalidad.
- Elaboración de este informe y preparación de anexos.

## Resultados Obtenidos

- El programa ordena correctamente las listas dadas.
- El promedio de elementos en una lista donde quicksort supera a bubble sort es de entre 23 y 27 elementos generalmente.
- Se comprendió el uso y diferencia de usos entre los algoritmos vistos.

## Conclusiones

A través de este trabajo práctico pudimos comprobar cómo varía el rendimiento entre dos algoritmos de ordenamiento: Bubble Sort y Quicksort. Si bien Bubble Sort resulta más sencillo de implementar y entender, su eficiencia se ve rápidamente superada por Quicksort a medida que aumenta la cantidad de datos.

Los resultados obtenidos confirman lo que se deduce del análisis teórico: **Quicksort es significativamente más eficiente en listas grandes**, mientras que Bubble Sort puede resultar útil únicamente en contextos muy acotados.

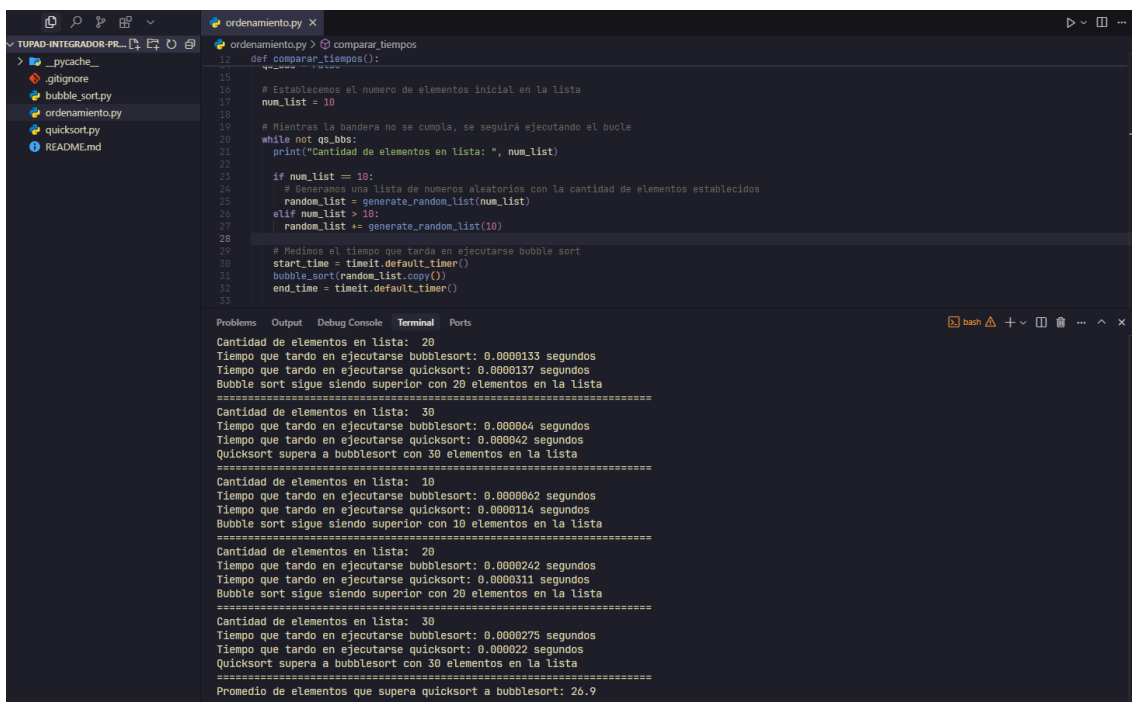
La experiencia también permitió afianzar conocimientos sobre el análisis de algoritmos, la medición de tiempos de ejecución, y la importancia de elegir correctamente el algoritmo según la necesidad del problema a resolver.

# Bibliografía

- [https://es.wikipedia.org/wiki/Algoritmo\\_de\\_ordenamiento](https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento)
- <https://www.geeksforgeeks.org/python-program-for-bubble-sort/>
- <https://www.geeksforgeeks.org/python-program-for-quicksort/>
- <https://docs.python.org/3/library/>
- [https://es.wikipedia.org/wiki/Ordenamiento\\_de\\_burbuja](https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja)
- <https://www.swhosting.com/es/comunidad/manual/algoritmos-de-ordenacion-con-ejemplos-en-c#:~:text=El%20Insertion%20Sort%20o%20Ordenamiento.todos%20los%20elementos%20est%C3%A9n%20ordenados.>
- Apuntes en formato pdf utilizados a lo largo de la materia.

## Anexos

- Link a repositorio del proyecto:  
<https://github.com/LautiZ/TUPaD-Integrador-Prg1>
- <https://youtu.be/TnTu1QWaWco?si=r4tQtFzosK87NxNk>
- Video explicativo: <https://youtu.be/22iWswgVnIU>



```
ordenamiento.py
ordenamiento.py > comparar_tiempos
def comparar_tiempos():
    # Establecemos el numero de elementos inicial en la lista
    num_list = 10
    # Mientras la bandera no se cumpla, se seguirá ejecutando el bucle
    while not qs_bbs:
        print("Cantidad de elementos en lista: ", num_list)
        if num_list == 10:
            # Generamos una lista de numeros aleatorios con la cantidad de elementos establecidos
            random_list = generate_random_list(num_list)
            elif num_list > 10:
                random_list += generate_random_list(10)
        # Medimos el tiempo que tarda en ejecutarse bubble sort
        start_time = timeit.default_timer()
        bubble_sort(random_list.copy())
        end_time = timeit.default_timer()

Problems Output Debug Console Terminal Ports
Cantidad de elementos en lista: 20
Tiempo que tardo en ejecutarse bubblesort: 0.0000133 segundos
Tiempo que tardo en ejecutarse quicksort: 0.0000137 segundos
Bubble sort sigue siendo superior con 20 elementos en la lista
=====
Cantidad de elementos en lista: 30
Tiempo que tardo en ejecutarse bubblesort: 0.000064 segundos
Tiempo que tardo en ejecutarse quicksort: 0.000042 segundos
Quicksort supera a bubblesort con 30 elementos en la lista
=====
Cantidad de elementos en lista: 10
Tiempo que tardo en ejecutarse bubblesort: 0.000062 segundos
Tiempo que tardo en ejecutarse quicksort: 0.0000114 segundos
Bubble sort sigue siendo superior con 10 elementos en la lista
=====
Cantidad de elementos en lista: 20
Tiempo que tardo en ejecutarse bubblesort: 0.0000242 segundos
Tiempo que tardo en ejecutarse quicksort: 0.0000311 segundos
Bubble sort sigue siendo superior con 20 elementos en la lista
=====
Cantidad de elementos en lista: 30
Tiempo que tardo en ejecutarse bubblesort: 0.0000275 segundos
Tiempo que tardo en ejecutarse quicksort: 0.000022 segundos
Quicksort supera a bubblesort con 30 elementos en la lista
=====
Promedio de elementos que supera quicksort a bubblesort: 26.9
```