Trabajo Práctico – Búsqueda y Ordenamiento

Alumnos:

Fulladoza, Pablo – fulladoza-1996@hotmail.com

Materia: Programación I

Profesor: Ariel Enferrel

Tutor: Franco Gonzalez

Fecha de entrega: 09 de junio del 2025

Índice:

Introducción:

En el ámbito de la programación, los algoritmos de búsqueda y ordenamiento desempeñan un papel fundamental para el procesamiento y manipulación eficiente de grandes volúmenes de datos. Es una tarea común en muchas aplicaciones como bases de datos, sistemas de archivos y algoritmos de inteligencia artificial.

Mantener los datos correctamente ordenados permite usar algoritmos de búsqueda más eficientes. Pero esta eficiencia es debatible en grupos de datos más pequeños.

En el presente trabajo se buscará demostrar cuales son los algoritmos de ordenamiento más eficientes, utilizando un análisis empírico de los algoritmos. Asimismo, se buscará determinar cuando este costo de procesamiento resulta más costoso que una simple búsqueda lineal sobre los datos desordenados.

Marco teórico:

Búsqueda lineal: es un algoritmo simple que consiste en iterar secuencialmente por todos los elementos de una lista hasta encontrar el buscado.

Búsqueda binaria: es un algoritmo eficiente que solo puede ser utilizado con un conjunto de datos ordenados. Divide el conjunto de datos a la mitad, toma el valor del medio y lo compara con el buscado, si este no es el correcto descarta la mitad superior o inferior según corresponda y continúa dividiendo las mitades hasta encontrarlo o determinar que el mismo no existe.

Ordenamiento burbuja (BubbleSort): es un algoritmo simple y facil de implementar, pero ineficiente para grandes volúmenes de datos. El mismo compara cada elemento con el siguiente y los intercambia si se encuentran mal ordenados.

Ordenamiento rápido (QuickSort): es un algoritmo eficiente que divide la lista en dos partes y las ordena de forma recursiva.

Ordenamiento por selección (SelectionSort): es un algoritmo que consiste en buscar el elemento más pequeño de la lista y lo intercambia por el primer elemento. Esto continúa hasta tener el conjunto completamente ordenado.

Ordenamiento por inserción (InsertionSort): es un algoritmo que consiste en insertar cada elemento en su posición correcta en una lista ordenada.

Análisis Empírico: el mismo consiste en realizar una observación directa del tiempo de ejecución de distintos algoritmos.

Caso práctico:

Se desarrollo un programa en Python incluyendo los algoritmos de ordenamiento y búsqueda mencionados.

Se incluyo la librería *timeit* para poder medir los tiempos de ejecución sobre listas aleatorias.

Se realizaron comparaciones entre los distintos algoritmos.

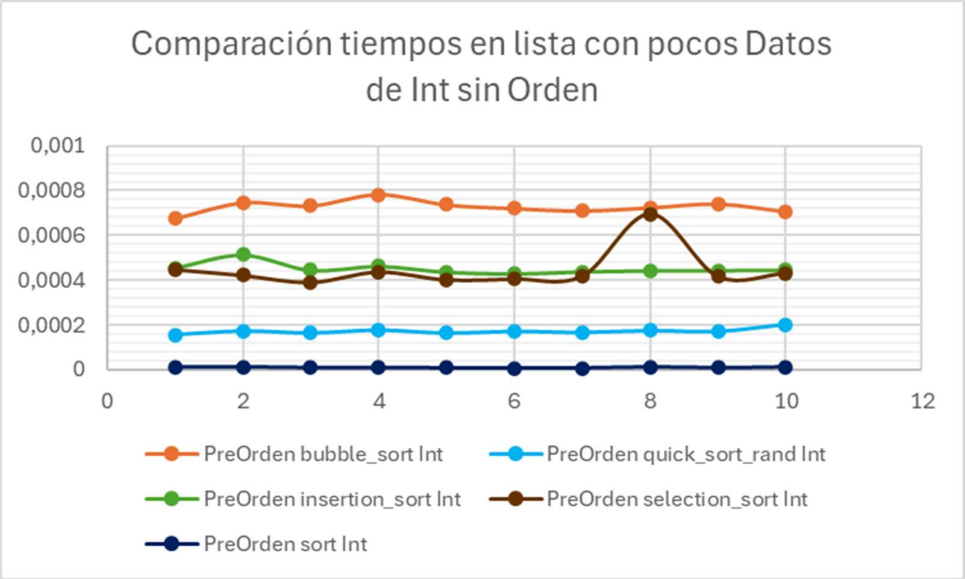
Metodología utilizada:

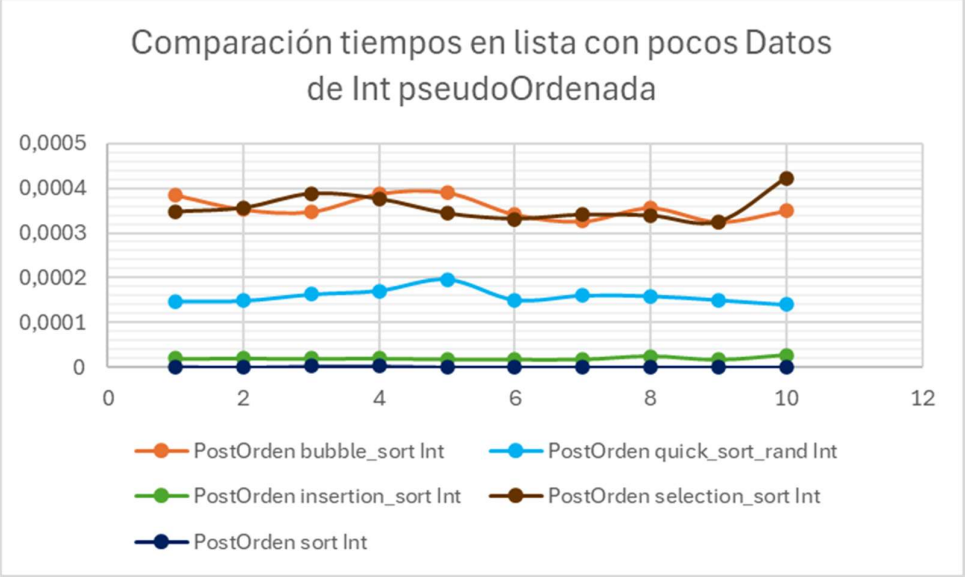
La elaboración del trabajo se realizó en las siguientes etapas:

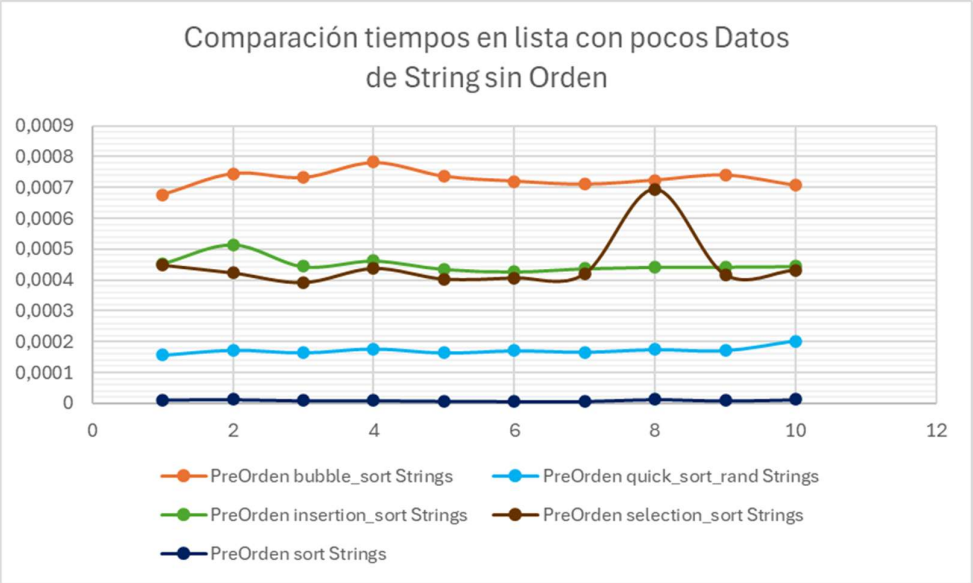
- Recolección de información dentro del material de la cátedra.
- Implementación en Python de los algoritmos estudiados, se sumo a los mismos el algoritmo standard *sort* de los elementos *list*.
- Pruebas de los distintos algoritmos con pocos datos -200-, separando pruebas entre ints y strings.
- Medición de los tiempos de las pruebas.
- Ordenamiento del listado e ingreso de -20- nuevos datos a listas.
- Medición de nuevos tiempos de ejecución sobre listado parcialmente ordenado.
- Pruebas de los distintos algoritmos con un gran volumen de datos -20000-, separando pruebas entre ints y strings.
- Medición de los tiempos de las pruebas.
- Ingreso de -200- nuevos datos a las listas.
- Pruebas de los algoritmos frente a listas parcialmente ordenadas.
- Registro de los nuevos tiempos y confección de un informe.
- Prueba de algoritmo de búsqueda incrementando progresivamente el tamaño de los datos hasta encontrar el punto donde la búsqueda lineal es más lenta que el método de ordenamiento más rápido (según los resultados anteriores) y usar búsqueda la binaria.
- Incorporación de este resultado al informe.

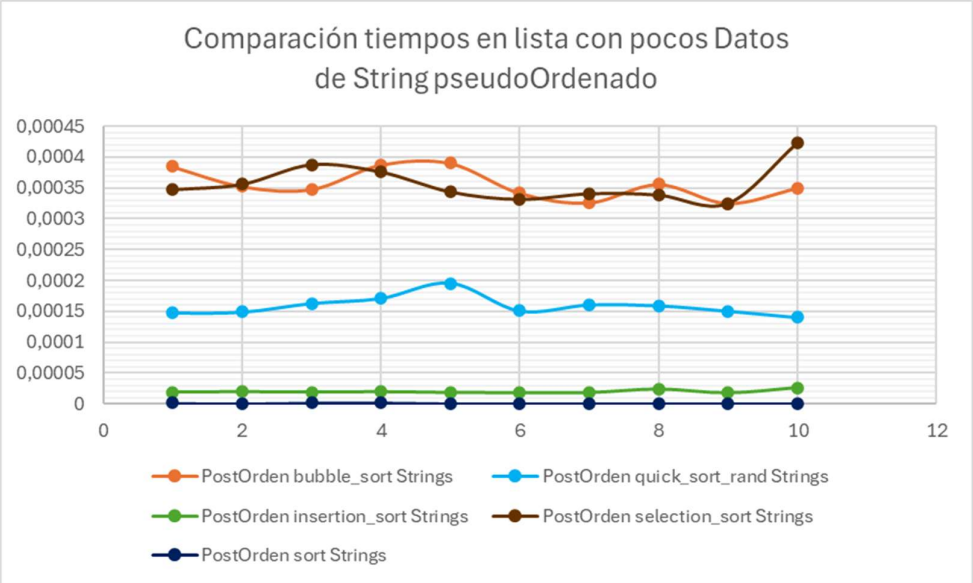
Resultados obtenidos:

- Se implemento correctamente los algoritmos sobre un bajo volumen de datos. Arrojando los siguientes resultados:









Cuando	Algoritmo de ordenamiento	Promedio Int	Promedio Strings	Variación
PreOrden	bubble_sort	0,00072755	0,00087495	120%
PreOrden	quick_sort_rand	0,00017177	0,00019849	116%
PreOrden	insertion_sort	0,00045029	0,00048505	108%
PreOrden	selection_sort	0,00044744	0,00057311	128%
PreOrden	sort	9,36999E-06	1,171E-05	125%
PostOrden	bubble_sort	0,00035614	0,00058231	164%
PostOrden	quick_sort_rand	0,00015851	0,00019345	122%
PostOrden	insertion_sort	1,98E-05	2,818E-05	142%
PostOrden	selection_sort	0,00035696	0,00058651	164%
PostOrden	sort	1,21001E-06	1,96999E-06	163%

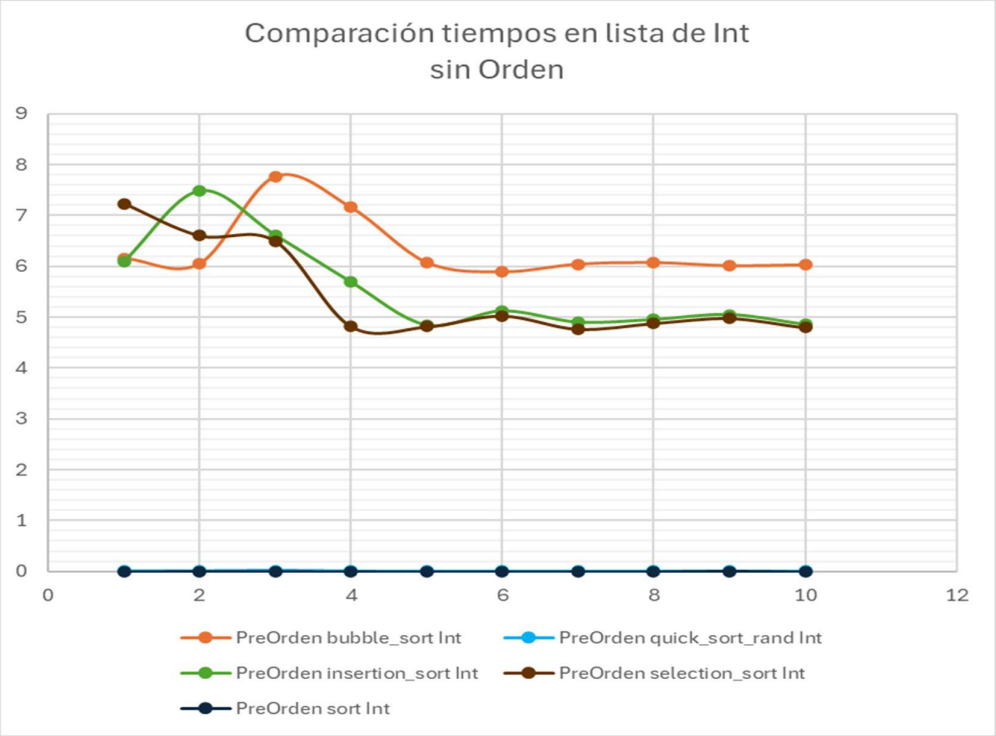
- Se produjo un error de recursión al intentar realizar un quicksort sobre un gran volumen de datos previamente ordenados.

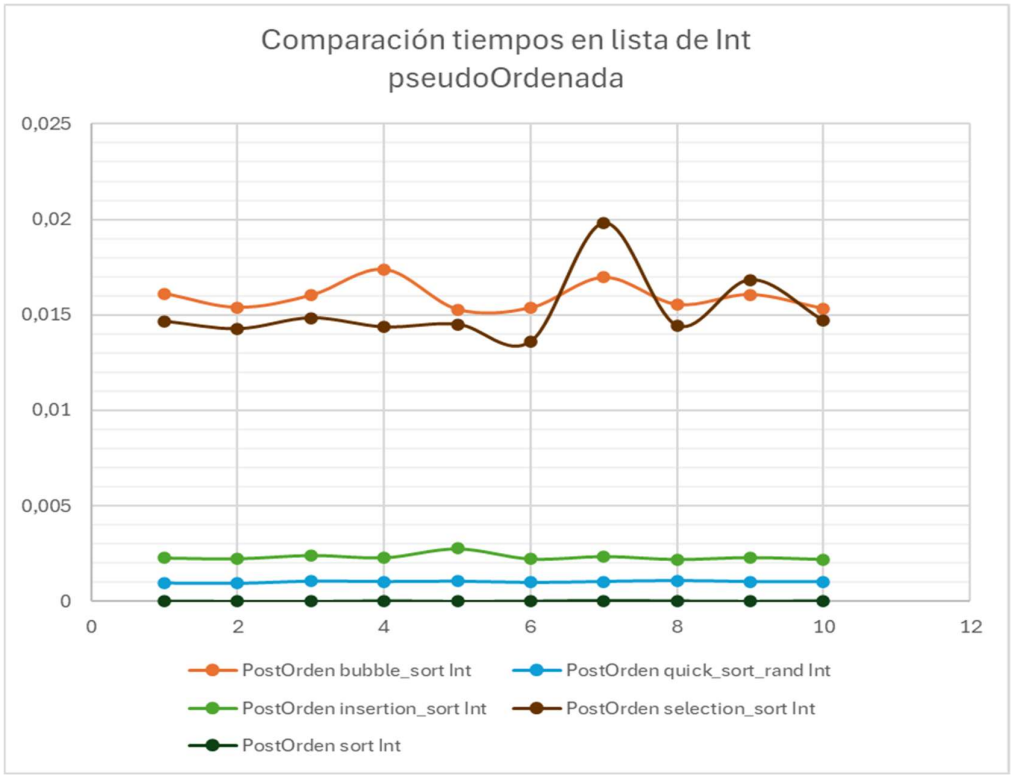
```
  [Previous line repeated 994 more times]
  File "d:\Documentos\Programacion\UTN distancia 2025\Programacion I\TP Integrador\UTN-TUP-Programacion-TP-Integrador\sort.py", line 36, in quick_sort
    less = [x for x in arr[1:] if x <= pivot]
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
RecursionError: maximum recursion depth exceeded
```

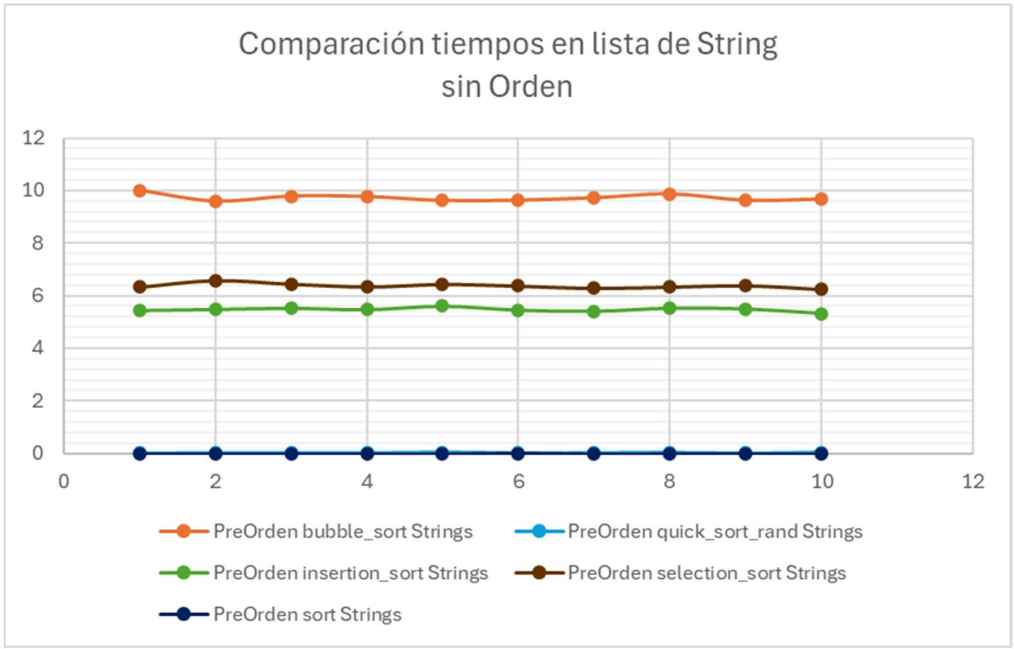
Luego de revisarlo con la inteligencia artificial esta nos sugirió modificar el pivot utilizado por el quicksort del primer elemento (*pivot = arr[0]*) a un pivot aleatorio lo cual lo hace más eficiente para grupos voluminosos de datos ordenados, con solo una leve perdida de rendimiento frente a datos desordenados.

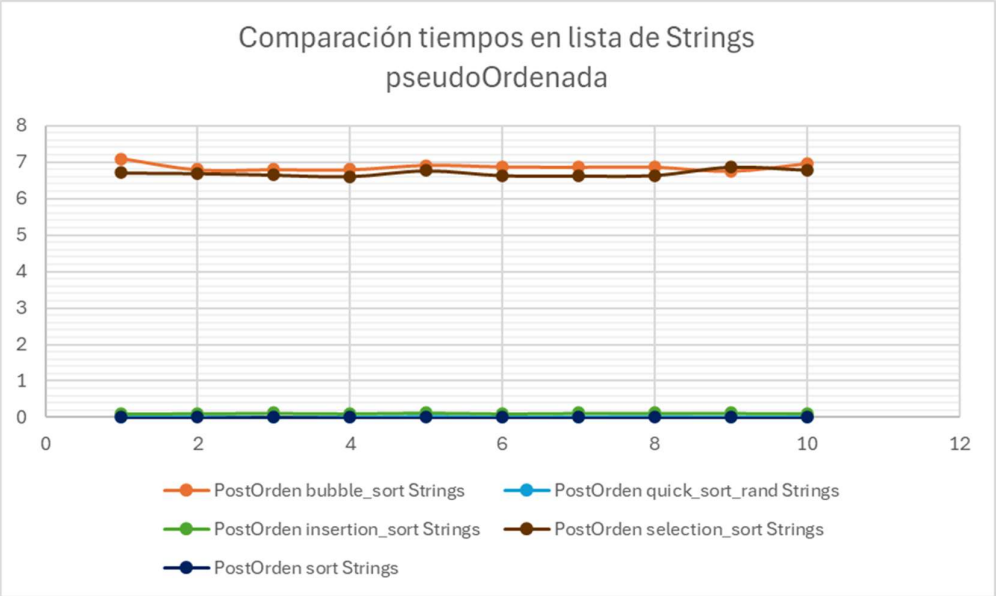
```
Tiempo de ejecución de quick_sort, 0.04193109995685518, segundos
Tiempo de ejecución de quick_sort_rand, 0.043142400099895895, segundos
```

- Se implemento correctamente los algoritmos en Python para grandes volúmenes de datos. Arrojando los siguientes resultados:




Comparación tiempos en lista de Int
pseudoOrdenada
0,025
0,02
0,015
0,01
0,005
0
0
2
4
6
8
10
12
PostOrden bubble_sort Int
PostOrden quick_sort_rand Int
PostOrden insertion_sort Int
PostOrden selection_sort Int
PostOrden sort Int


Comparación tiempos en lista de String
sin Orden
12
10
8
6
4
2
0
0
2
4
6
8
10
12
PreOrden bubble_sort Strings
PreOrden quick_sort_rand Strings
PreOrden insertion_sort Strings
PreOrden selection_sort Strings
PreOrden sort Strings



Cuando	Algoritmo de ordenamiento	Promedio Int	Promedio Strings	Variación
PreOrden	bubble_sort	6,32655363	9,72766129	154%
PreOrden	quick_sort_rand	0,01165941	0,03035653	260%
PreOrden	insertion_sort	5,5580832	5,47674213	99%
PreOrden	selection_sort	5,44064363	6,37101803	117%
PreOrden	sort	0,00205773	0,00330548	161%
PostOrden	bubble_sort	0,01593747	6,86504683	43075%
PostOrden	quick_sort_rand	0,00101265	0,02905243	2869%
PostOrden	insertion_sort	0,00231668	0,11949614	5158%
PostOrden	selection_sort	0,01519257	6,68926136	44030%
PostOrden	sort	2,066E-05	0,00020933	1013%

- Se logro observar que, para el caso de listas de datos, el sort incluido en el método de listas es el más eficiente en cualquier situación probada con datos básicos.
- Se logro observar que el algoritmo de quick_sort bien implementado es la solución más rápida ya sea para pocos o muchos datos si dejamos de lado el método integrado en listas.
- Se observó que el algoritmo de burbuja es de los más ineficientes en todos los casos junto con el algoritmo de selección.
- Se observó que el algoritmo de inserción presenta una mejora significativa ante listas parcialmente ordenadas.
- Se observó que según el tipo de dato cambia drásticamente la velocidad de procesamiento de los algoritmos.
- Implementando un algoritmo para incrementar el tamaño de los datos progresivamente para comparar el tiempo de una *búsqueda lineal* contra el

tiempo de *ordenamiento + búsqueda* se determinó que los algoritmos de Ordenamiento son sumamente costosos, por los que, aunque una búsqueda binaria es mucho más rápida en cualquier caso que una lineal, esto no amerita ordenar los datos para realizar una sola búsqueda.

- Implementando un algoritmo para incrementar el tamaño de los datos progresivamente para comparar el tiempo de una *búsqueda lineal* contra el tiempo de *ordenamiento + búsqueda* en una lista parcialmente ordenada, se determinó que la elección correcta de un método de ordenamiento tiene gran importancia para el conjunto de datos, ya que un algoritmo ineficiente va a representar una perdida de tiempo en comparación de una simple búsqueda lineal.

Conclusiones:

Con el desarrollo del presente trabajo se logró probar empíricamente cuál de los algoritmos es más veloz para su uso. Y la importancia de seleccionar correctamente el método de ordenamiento ya que este puede tener diferencias sustanciales en tiempo de ejecución.

Se logro comprobar errores posibles en una recursión mal implementada. Y en consecuencia gracias a esto la importancia de la continua investigación sobre algoritmos de ordenamiento, ya que un pequeño cambio puede aumentar o disminuir significativamente su rendimiento.

Se concluyo que los algoritmos de ordenamiento son muy costosos computacionalmente. Si se busca ordenar los datos cada vez que se ingresan nuevos, uno debe asegurarse de estar usando un algoritmo extremadamente eficiente, para no perder tiempo. Por lo tanto, es recomendable evaluar el número de búsquedas que se van a realizar sobre los datos antes de decidir ordenar. Con el fin maximizar la ganancia de tiempo utilizando las búsquedas binarias y este sea suficiente para compensar el costo de ordenamiento.

Se concluyo que utilizar un algoritmo de ordenamiento para una sola búsqueda resulta en una mayor perdida que ganancia en tiempo de proceso.

A futuro para profundizar este análisis se pueden realizar pruebas sobre datos más complejos, como objetos grandes. También se podría realizar la prueba con más algoritmos de ordenamiento o búsqueda no implementados.

Bibliografía:

- Material y ejercicios propuestos por la materia.
- Generar lista de Strings Random - https://www.geeksforgeeks.org/python-generate-random-string-of-given-length/
- Inicializar dict para poder agrupar los resultados en listas - https://docs.python.org/3/library/collections.html#collections.defaultdict

Anexo:

Github: https://github.com/Fulla1996/UTN-TUP-Programacion-TP-Integrador

Video: https://youtu.be/rXw8BOX_U-k

Tablas de comparaciones en Excel: Tabla para graficos.xlsx (se requiere cuenta @tupad.utn.edu.ar)