Reporte de Resultados

- **Nombre**: Murillo García Sebastián

- **Práctica #1**: Viernes 5 de marzo, 2021

- **Aspectos prácticos adicionales**.

Se investigó un poco sobre la importación de la librería time para utilizar el método adecuado que permitiera la medición del tiempo de ejecución. Al final, debido a la naturaleza del problema, se ocupó del método **time.time_ns()** debido a que en las pruebas regresó números enteros como valores (con ello se pudieron hacer los promedios adecuados) y que los nanosegundos ser la unidad medida más ad hoc para la naturaleza del problema.

Además, se buscó rápidamente sobre la función de logaritmo **math.log(**valor**)** de la librería math para obtener el logaritmo de los valores de tiempo de ejecución ya promediados. Esta decisión se tomó debido a que cuando se hicieron las primeras pruebas de medición de tiempos y se graficaron los resultados, las gráficas no mostraban de la mejor manera la disposición de los datos resultantes. Al calcular el logaritmo, si bien se "perdió" el valor del tiempo como tal, nos otorgó una mejor visualización de los datos en la gráfica para hacer su comparación.

Para la creación de las gráficas se utilizó de la librería matplotlib, específicamente de pyplot y sus funcionalidades más sencillas como **pyplot.plot(**componente x**,** componente y**)** para asignar los valores de la función con los datos de entrada y tiempos de ejecución, **pyplot.title(**título**)** para poner el título a la gráfica, **pyplot.xlabel(**nombre**)** para asignar el nombre de los ejes correspondientes, **pyplot.legend(**[valores_1, valores_2, valores_3]**)** para indicar la leyenda de cuál función corresponde con el algoritmo de ordenamiento, y **pyplot.show()** para mostrar la gráfica en la consola.

Con respecto a la lectura de archivos se investigó rápidamente sobre las funciones de **open(**directorio del texto**)** y **close()**. Es importante recalcar que se debe de poner el directorio exacto donde se encuentre el texto para la lectura del archivo movie_titles.txt. Fue debido a este texto que se implementó la clase Película para crear instancias que se pudieran comparar entre sí, pues el archivo tiene una cantidad considerable de datos que se pudieron ocupar para el problema en cuestión: ordenamiento de datos en una colección.

Por último, se ocuparon dos diferentes funciones de la librería random, a saber: **random.randint(**min, max**)** para obtener valores numéricos enteros entre los límites (ambos incluyentes) propuestos de entrada. Esta función se ocupó para la creación de listas aleatorias de números y de caracteres. Además se utilizó la función **random.shuffle(**lista**)** para reordenar una lista de manera aleatoria y poder medir los tiempos de ejecución en el *caso promedio*.

- **Aspectos conceptuales adicionales**.

Se investigó un poco sobre la función **chr(**valor**)** para la creación de listas aleatorias de caracteres. Esta función solamente "convierte" los valores enteros ASCII en caracteres específicos. Por lo tanto, esta función se utilizó en conjunto con **random.randint(**min, max**)**, donde min = 32 y max = 127, pues si observamos la tabla ASCII se puede notar que esos son los valores mínimos y máximos que se pueden convertir a caracteres.

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

- **Retos**.

  Uno de los retos al trabajar por subtareas fue que en el momento de las reuniones para compartir lo programado, se tuvieron códigos funcionales que no estaban articulados entre sí. Por un lado, uno tenía las creaciones de tres listas: numéricas, de caracteres y de películas con las funciones de manejo de las mismas, pero estaban desarticuladas dentro del código de pruebas que fue hecho por otra persona, pues no se contemplaba en su totalidad cómo es que estas iban a implementarse. Por otro lado, se tenía el código de la lectura del archivo de películas investigado y programado por uno de los integrantes, pero no contemplaba la instanciación de las Película y tampoco la creación de una lista de estos objetos que otro miembro del equipo estaba programando. Por último, una persona investigó sobre la librería de matplotlib para implementar las gráficas necesarias que se iban a analizar y otra hizo los algoritmos de ordenamiento con sus pruebas.

  Debido a estas complicaciones se adaptaron diferentes funciones antes mencionadas y se programó un código que implementa cada uno de ellos: **prueba(**tipoLista, tipoOrden**)**. Esta función crea el tipo de lista que se coloque como entrada (ya sea la de números, caracteres o películas), ordena las listas del modo que se indique en la entrada (en orden, en orden inverso o de manera aleatoria), mide los tiempos promedio de ejecución y regresa estos tiempos medidos en tres listas que contienen el valor de la medición por tamaño de entrada de los tres algoritmos de ordenamiento. De esta manera, con solo un llamado a la función obtenemos los resultados deseados.

  Se debe mencionar que este método tarda en completar su ejecución, pero facilita bastante el trabajo.

- **Detalles experimentos**.

  Implementamos tres tipos de ordenamientos de datos: ordenados [el *mejor de los casos*], inversos [el *peor de los casos*] y aleatorios [el *caso promedio*] para tener una mejor comparación de la complejidad de los algoritmos.

  Se pensó en hacer una lista con números y caracteres, pero se terminaron ocupando solo caracteres que son seleccionados de manera aleatoria, obtenidos por la función chr( ). El método usa los caracteres del código ASCII

Listas num sin repetirse. Optamos por permitir que se repitan los números, pues esto agrega complejidad al ordenamiento, sin embargo, antes de tomar esta decisión sí hicimos el código para evitar repeticiones, fue eliminado una vez tomada la decisión.

Lectura de archivos con num aleatorios. Esto se resolvió con la implementación de shuffle ya que, ingresando los datos manualmente, no se podía llevar a cabo.

Entre más tamaño de entrada, mayor "resolución" va a tener la gráfica, pero el método {prueba} se iba a tardar mucho en ejecutarse, entonces se decidió colocar menos cantidad de entrada.

Hicimos pruebas con un máximo de 12000 datos, pues con cantidades mayores era mayor el tiempo de ejecución y los resultados obtenidos eran los mismos.

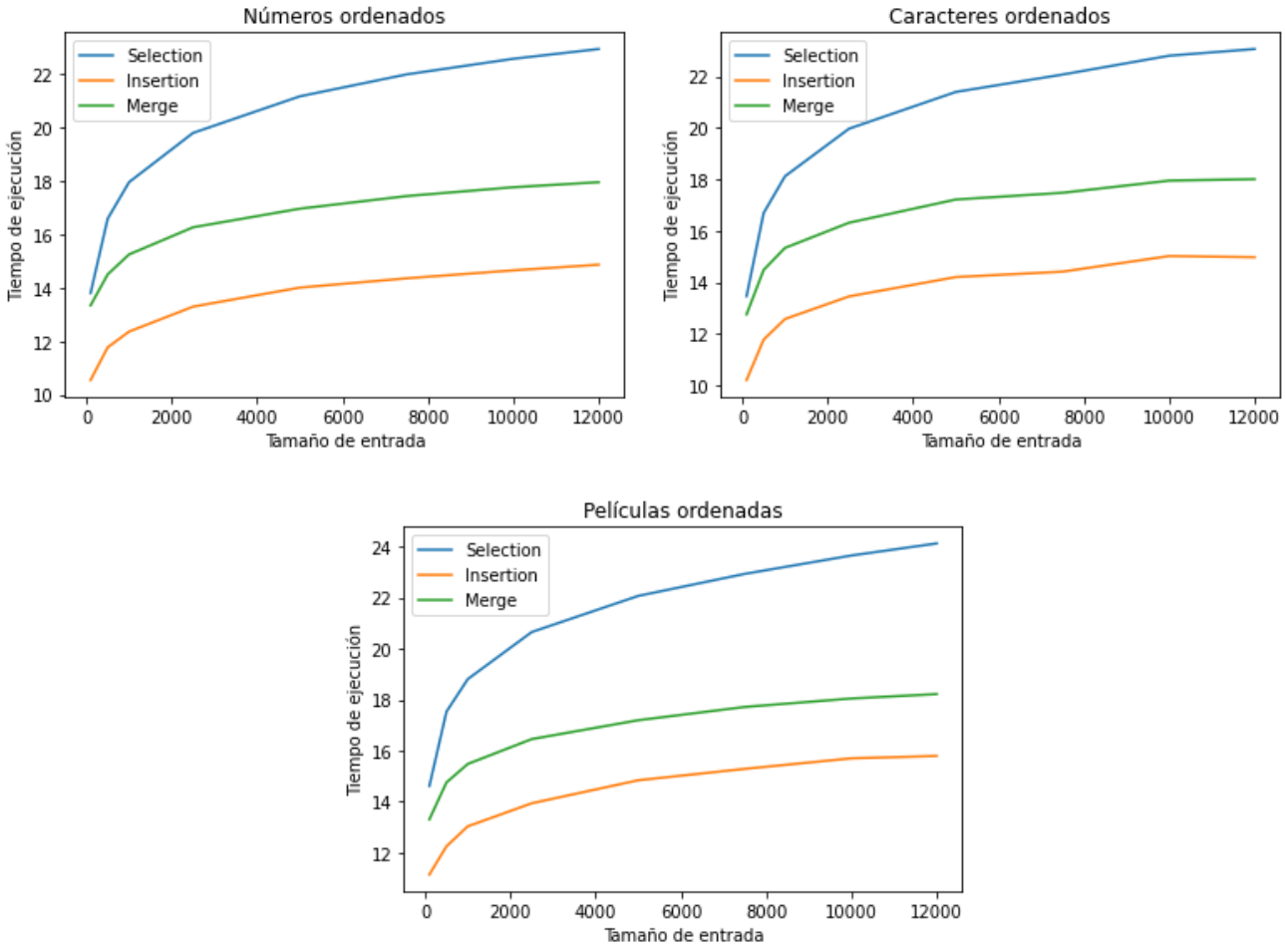
Debido a que nuestras primeras gráficas nos representaban los resultados obtenidos de una forma confusa, se tomó la decisión de aplicarles el log( ) a las curvas de los datos obtenidos para que la visualización de los mismos fuese mejor. Fue conveniente, pues permitió un mejor análisis e interpretación.

- **Gráficas e interpretación de los resultados**.

Para interpretar las gráficas es necesario conocer el tipo de complejidad que tienen los algoritmos de ordenamiento en los tres tipos de casos a considerar: el *mejor de los casos*, el *peor de los casos* y el *caso promedio*. De esta manera se podrá entender un poco más a fondo el funcionamiento de los distintos métodos y se podrá tomar una decisión apropiada para la implementación de los mismos.

Datos completamente ordenados. [El *mejor de los casos*]

I. *Figura A*: Tiempos de ejecución para datos completamente ordenados.



Interpretación.

Desde una primera vista se puede observar claramente que en los tres tipos de datos considerados, el algoritmo Selection Sort fue el más tardado para efectuar su trabajo y que Insertion Sort fue el más rápido. Esto se debe completamente a la complejidad de los algoritmos de ordenamiento. Por un lado, tenemos que Selection Sort tiene una complejidad de **$O(n^2)$** en su *mejor caso*. ¿A qué se debe esto? Se debe a que el método ocupa un for como estructura algorítmica cíclica en su interior; esto provoca que siempre se recorra una cantidad determinada de datos de la colección que se quiere ordenar. De esta manera se tiene que el número de comparaciones que tiene que hacer el algoritmo siempre es de $(n^2+n)/2$ y esto influye completamente en el tiempo de ejecución. Aquí tenemos un resultado importante: el tiempo en que se tarda el algoritmo (y el número de comparaciones que hace) depende enteramente del tamaño de entrada y no del tipo de datos que se está ocupando o el ordenamiento de los mismos.

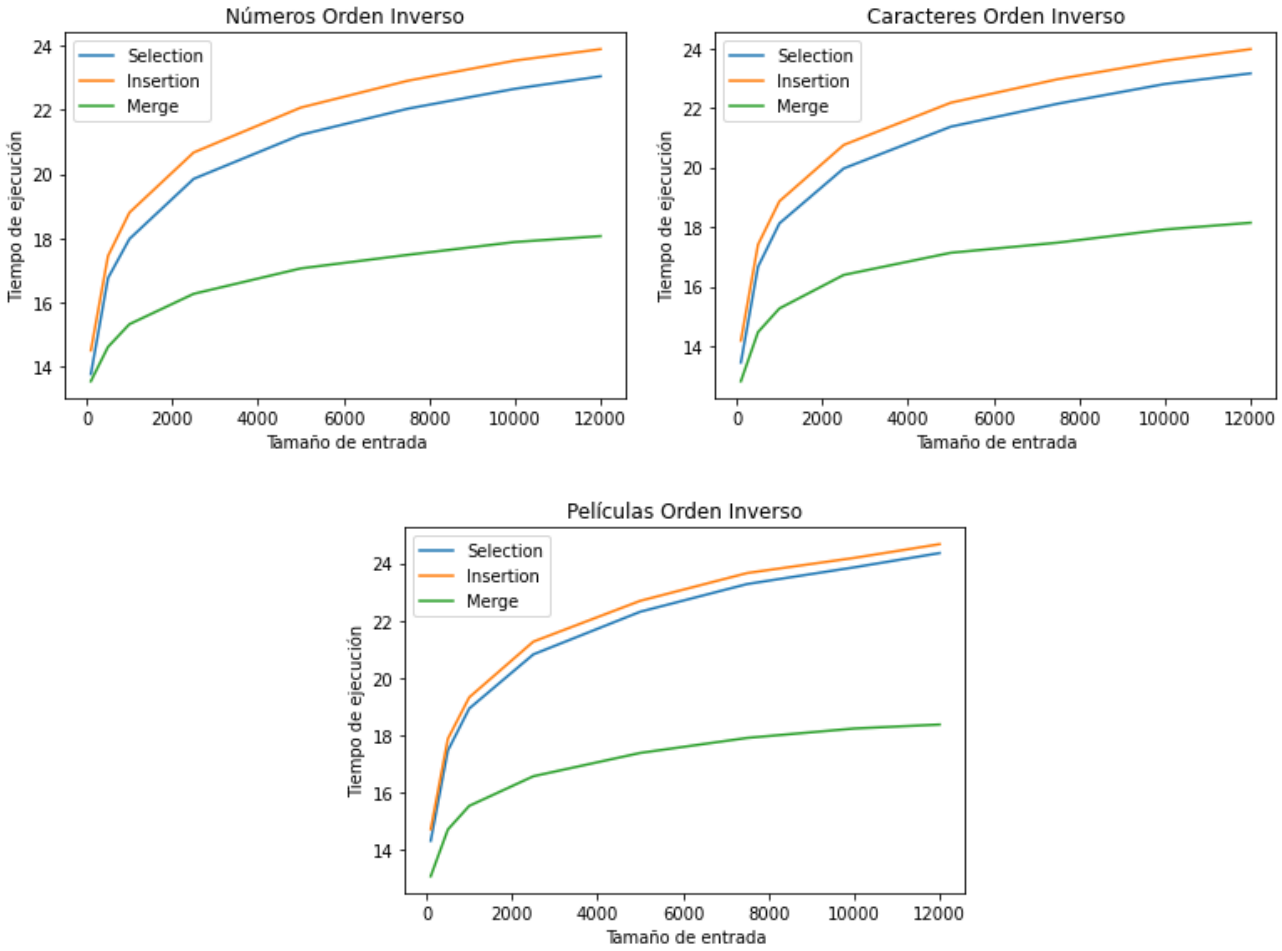
Por otro lado, se tiene que la complejidad de Insertion Sort es de **O(n)** en su *mejor caso*. ¿Esto en qué influye? Si nos ponemos a analizar el código podemos ver algo interesante: ocupa de un while como estructura algorítmica cíclica en su interior. Esta estructura hace que no siempre se tenga que hacer el recorrido en la colección a ordenar. En este caso, como la lista ya se encuentra completamente ordenada, una condición del while no se cumple: que el último dato sea menor que su predecesor, es decir, ya está en su posición adecuada. Entonces en ninguna vuelta del for externo que recorre de manera general a la colección se entra en el while y solo se hacen una cantidad específica de comparaciones: n-1 (siendo 'n' la cantidad de datos a ordenar).

Ahora bien, ¿qué sucede con Merge Sort? En el *mejor de los casos*, este algoritmo de ordenamiento tiene complejidad de **Ω(nlog n)**. Esto se debe a que el método divide la colección en subcolecciones (seguro por ello es el nombre) para ir ordenando poco a poco cada una de ellas; en otras palabras y en comparación con los dos algoritmos antes mencionados, este "divide" el problema general (ordenar) en sub-problemas. No resulta ser tan rápido como Insertion, y esto se puede deber a la jerarquía misma de complejidad: $O(n) \subset O(n \log n) \subset O(n^2)$.

Por lo tanto, el orden de menor a mayor en tiempos de ejecución queda de la siguiente manera: Insertion, Merge y Selection Sort. Esto nos podría decir que los mejores algoritmos para ordenar una colección que esté ordenada (o casi ordenada) son Insertion primeramente, y Merge como segundo lugar.

Datos en un orden inverso. [El *peor de los casos*]

II. *Figura B*: Tiempos de ejecución para datos ordenados de manera inversa.



Interpretación.

Si consideramos el *peor de los casos* tenemos resultados completamente diferentes. Podemos ver instantáneamente que mientras el método de ordenamiento Merge es más rápido por una cantidad considerable de tiempo, Insertion resulta ser el más tardado (Selection solo se acerca al tiempo de Insertion). ¿A qué se deben estos resultados? Sabemos que la complejidad del algoritmo de ordenamiento Merge Sort en su *peor caso* es de **O(nlog n)**. Aquí radica la importancia también del hecho de que ocupa el mecanismo de "divide conquer". Particularmente lo que hace el algoritmo es que divide la colección en en 'n' subcolecciones. A pesar de estar en el caso en el que los datos están ordenados de manera inversa, el algoritmo sigue cumpliendo su función de "reducir" el problema (de nuevo, el ordenar los datos.

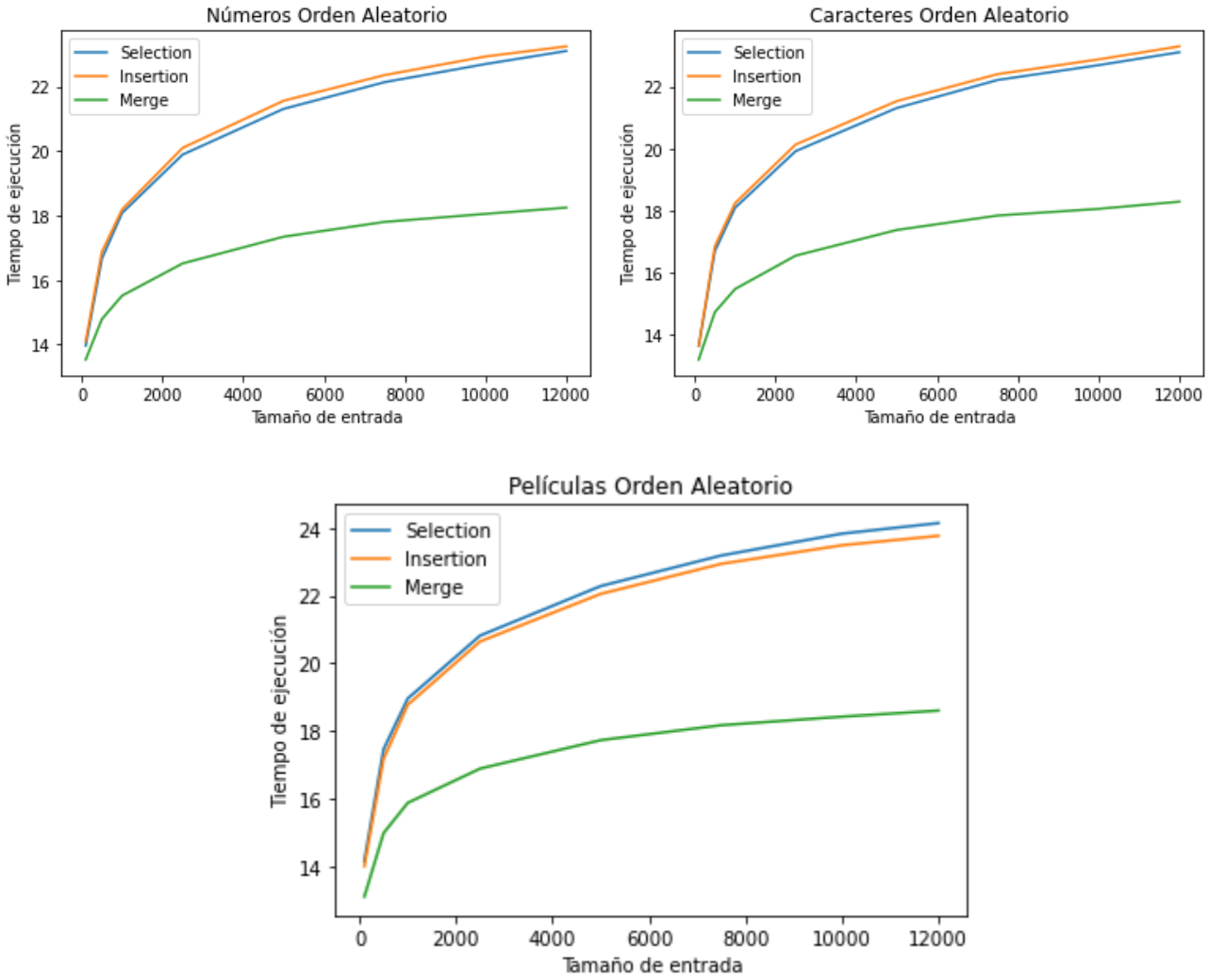
Si consideramos el caso de Insertion tenemos que su complejidad en este caso es de **$O(n^2)$**. Y en comparación con su *mejor caso*, podemos observar que cuando hace la entrada al while interior, la condición que siempre se va a cumplir es que el último elemento es mayor a su predecesor; además, podemos también analizar el hecho de que este último elemento va a tener que recorrer hasta su posición adecuada, que resulta ser al principio de la subcolección que trata en el while. Es por ello que el algoritmo va a hacer $(n^2+n)/2$ comparaciones y esto, como se dijo anteriormente, influye completamente en el tiempo de ejecución.

Por último, ¿qué pasa con Selection Sort? En su *peor de los casos* este también tiene complejidad de **$O(n^2)$**. Entonces, ¿por qué se tarda menos que Insertion y no lo mismo? Si bien el algoritmo también hace $(n^2+n)/2$ comparaciones, este resultado se puede deber enteramente a la estructura misma del algoritmo de ordenamiento. Mientras que Insertion hace los llamados swap dentro de su estructura cíclica interna, Selection va guardando el índice en el que se encuentra el elemento más chico dentro de su estructura cíclica interna y hace el swap en la externa. Tal vez es por ello que se tarda una cantidad menor de tiempo en comparación del Selection Sort. ¿Esto quiere decir que el tiempo de ejecución de estos dos algoritmos pueden parecerse más o menos dependiendo del tipo de ordenamiento que tengan los datos en la colección? Puede que sí.

Por lo tanto, el orden de menor a mayor en tiempos de ejecución queda de la siguiente manera: Merge, Selection e Insertion Sort. Esto nos da el claro resultado de que Merge Sort es el algoritmo de ordenamiento más adecuado cuando se trata de una colección que se encuentra completa o mayormente invertida (además se ve claramente en las gráficas que “le gana” por una cantidad considerable a sus combatientes).

Datos ordenados aleatoriamente. [El *caso promedio*]

III. *Figura C*: Tiempos de ejecución para datos ordenados de manera inversa.



Interpretación.

En el *caso promedio* obtenemos resultados interesantes que podemos analizar. Primeramente hay que mencionar el caso de la complejidad de Merge Sort, ya que este resulta ser el menos tardado cuando se trata de una colección con un ordenamiento aleatorio. Su complejidad en el *caso promedio* es de **Θ(nlog n)**. Podemos recalcar mil veces la ventaja de que este método de ordenamiento divida en sub-colecciones para este tipo de problema, pero sería caer en la repetición.

Lo interesante tal vez radica en el hecho de que tanto Insertion como Selection Sort tienen en su *caso promedio* complejidad de $\mathbf{O(n^2)}$. Si observamos bien, ambos valores en tiempo de ejecución por tamaño de entrada parecen ser muy cercanos. Además, en los casos en que se ocuparon las listas

numéricas y de caracteres Insertion Sort es el más tardado, y cuando se ordenaron instancias de la clase Película el más tardado fue Selection Sort. ¿A qué se debe esto? Podemos tener distintas suposiciones para contestar a esta pregunta. Si volvemos analizar las gráficas anteriores podemos notar que en el *mejor de los casos* Insertion se tarda menos que Selection, y en el *peor de los casos* es al revés. Esto nos puede dar un primer indicador que podemos tomar en cuenta: puede ser que en el caso del ordenamiento de las películas que están dispersas en la lista de manera aleatoria este orden fuera tal que generalmente estuviera más ordenado que desordenado (y claro, en el caso de las listas de números y caracteres fuera lo contrario). Y tiene sentido en algún punto, pues si es cierto que ambos tienen la misma complejidad en el *caso promedio* y se desconoce si los datos están más ordenados o lo contrario, se puede observar en el hecho de que ambos datos de algoritmos en cuanto a tiempo de ejecución estuvieran tan cerca en las pruebas realizadas. Aunque también podemos considerar el caso de las estructuras algorítmicas de repetición y el manejo de los swap en ellas.

Lo que podemos afirmar sin titubear es que Merge Sort resulta ser el menos tardado para ordenar una lista en orden aleatorio debido a la ventaja de su complejidad en el *caso promedio*. En el caso de Insertion y Selection Sort puede variar dependiendo del ordenamiento de los datos que se encuentran en la colección de entrada.

- **Conclusiones**.

Se concluye que, aunque es conveniente usar los algoritmos de ordenamiento al momento de procesar datos, es importante saber distinguir cuál es el más conveniente según el tipo de dato que queramos tratar. También es indispensable tener presente que los datos pueden venir acomodados de diferentes maneras y que su contenido puede variar, por eso en este proyecto se cubrieron todos los escenarios posibles que se nos ocurrieron. Gracias a los análisis hechos anteriormente en este documento, podemos concluir que el mejor algoritmo para procesar datos que están mayormente ordenados es Insertion Sort, pero cuando se trata de datos que están ordenados de manera aleatoria o de manera inversa (en su mayoría), Merge Sort es el indicado para hacer el trabajo.

En general, el proyecto nos hizo darnos cuenta de la importancia de saber programar un mismo algoritmo en diferentes lenguajes de programación ya que el primer acercamiento con los algoritmos de ordenamiento fue en Java. Además, en esta primer práctica se pusieron en práctica muchos de los conocimientos nuevos adquiridos sobre la programación en Python, por lo que se adquirió cierta habilidad que poco a poco se irá puliendo.

- **Bibliografía**.

(S/A). (S/A). *9.8. Clases de tipos integrados.* Recuperado el 2 de marzo, 2021, de COVANTEC. Sitio web: https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion9/clases_integradas.html#python-mtd-readlines

(S/A). (2020) *Cómo leer líneas específicas de un archivo en Python.* Recuperado el 2 de marzo, 2021, de DelftStack. Sitio web: https://www.delftstack.com/es/howto/python/how-to-read-specific-lines-from-a-file-in-python/

Alonso, J. (N/A). *Análisis de la complejidad de los algoritmos.* Recuperado en Marzo, 2021, de la Universidad de Sevilla. Sitio web: http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/tema-28.html

(S/A). (S/A). Analysis of selection sort. Marzo, 2021, de Khan Academy. Sitio web: https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/analysis-of-selection-sort

(S/A). (S/A). *Analysis of insertion sort.* Marzo, 2021, de Khan Academy. Sitio web: https://www.khanacademy.org/computing/computer-science/algorithms/insertion-sort/a/analysis-of-insertion-sort

(S/A). (S/A). *Analysis of merge sort.* Marzo, 2021, de Khan Academy. Sitio web: https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort

CS Dojo. (11 de junio, 2018) *Intro to Data Analysis / Visualization with Python, Matplotlib and Pandas | Matplotlib Tutorial.* [Archivo de video]. Youtube. https://www.youtube.com/watch?v=a9UrKTVEeZAhttps://www.youtube.com/watch?v=KzqSDvzOFNA

Real Python. (27 de junio, 2019) *How to Split Strings in Python With the split() Method.* [Archivo de video]. Youtube. https://www.youtube.com/watch?v=-yzfxeMBe1s

Chinmoy Lenka. (2017). *chr() in Python.* Recuperado en Marzo, 2021, de GeeksforGeeks. Sitio web:https://www.geeksforgeeks.org/chr-in-python/#:~:text=The%20chr()%20method%20returns,code%20point%20is%20an%20integer.&text=The%20chr()%20method%20takes,(0x10FFFF%20in%20base%2016).