Introducción

Para el desarrollo de esta práctica se implementaron, en *Python*, cinco métodos de ordenamiento: ordenamiento por selección directa, inserción directa, *merge sort, quick sort* y *bubble sort*. En el desarrollo, estos métodos fueron probados utilizando tres tipos de datos: *integers*, *floats* y *strings*. Se generaron arreglos ordenados, inversamente ordenado, y con orden aleatorio de tamaños 3^i para $i \in \{1, ..., 9\}$. En cada caso, estos arreglos fueron ordenados con cada metodo de ordenamiento, midiendo su desempeño en tiempo y número de comparaciones que realizaba cada método. Estás mediciones se reportan en distintas gráficas generadas con la libreria *matplotlib*. Comenzamos con el desempeño individual de cada método.

Comportamiento Individual

Selección Directa

El desempeño de *selection sort* no depende de la calidad del arreglo de entrada, si está ordenado o inversamente ordenado, sino solo del tamaño de este. Esto se debe a que en el primer paso, el método debe encontrar el mínimo de los elementos, para colocarlo en la primera posición, así en cada paso debe encontrar el mínimo de los elementos restantes y colocarlo en la posición adecuada. Dado que el método no comprueba a cada paso si esta ordenado, en cada iteración debe recorrer la totalidad de elementos a la derecha, sin importar el orden en el que estén. Este comportamiento se ve reflejado en las gráficas que se presentan a continuación, dado que, sin importar si se trata del experimento con arreglos ordenados, inversamente ordenados, o con orden aleatorio, el tiempo de ejecución permanece constante para cada tipo de dato.

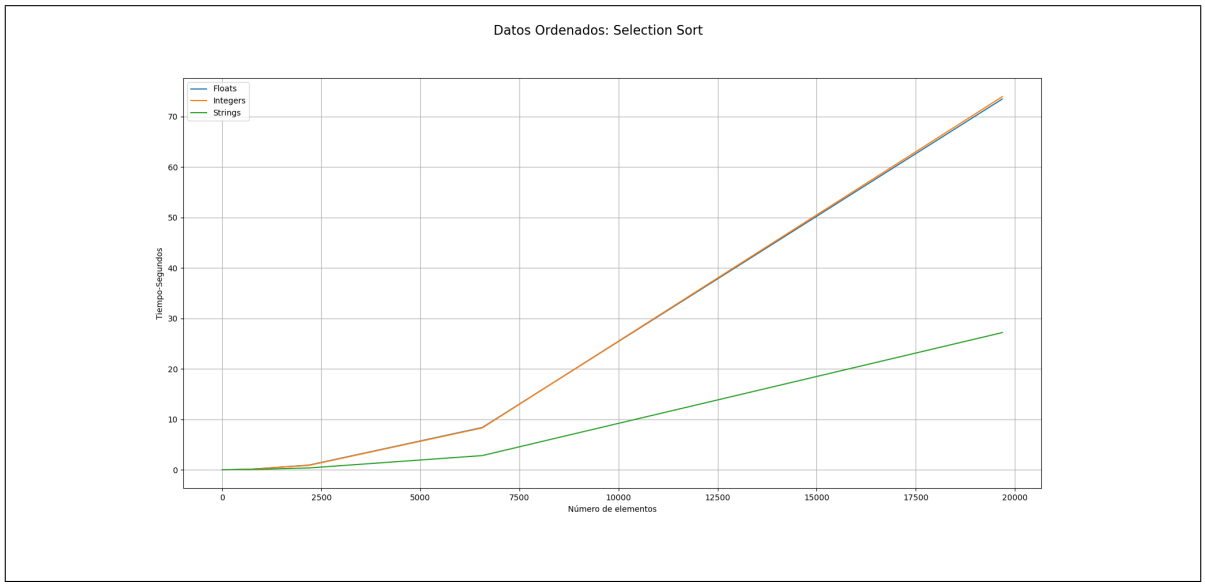

Figura 1: Tiempo de Ejecución-Datos Ordendos

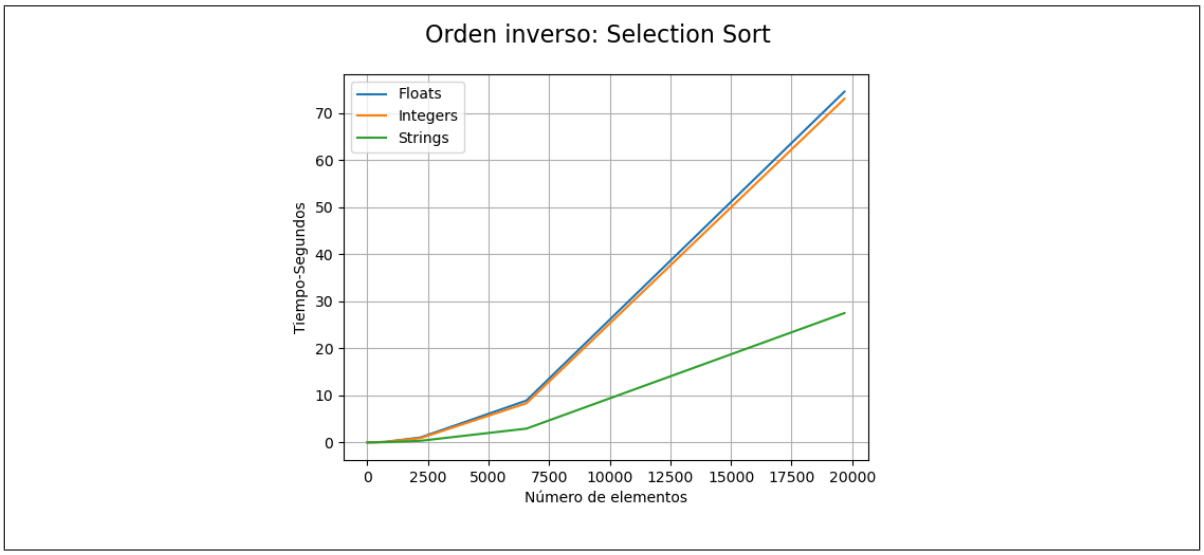

Figura 2: Tiempo de Ejecución-Datos en Orden Inverso

Algo que podemos notar es que el método parece tardar lo mismo para ordenar datos de tipo *integer* y tipo *float*, o al menos no hay una diferencia que sea significativa. Sin embargo, para los *strings* sí parece haber una diferencia significativa en el tiempo de ejecución del método, pues a partir de $10,000$ elementos, el tiempo requerido para ordenar un arreglo de *integers* o *float* es al menos el doble que el tiempo requerido para ordenar un arreglo de *strings* del mismo tamaño. Esto no significa que la complejidad del algoritmo cambie, simplemente significa que el tiempo constante requerido para cada operacion es menor en el caso de los *strings*, y por tanto los coeficientes que caracterizan a su cota asintótica, son distintos. A continuación se presentan las mediciones para el ordenamiento por selección directa cuando los datos ya se encuentran ordenados. Como podemos observar, el comportamiento del método no es distinto a los dos casos anteriores.

Figura 3: Tiempo de Ejecución-Orden Aleatorio

Inserción Directa

Por otro lado, el desempeño del algoritmo por inserción directa sí depende de la calidad del arreglo de datos. Supongamos que en cierto paso hemos ordenado ya k elementos del arreglo, procedemos entonces a insertar el siguiente elemento $k+1$. Para esto comparamos al elemento en la posición $k+1$ con el primer elemento a su izquierda, el elemento en la posición k, si este elemento izquierdo es mayor que nuestro elemento a insertar, realizamos un cambio de posiciones y volvemos a comparar el elemento a insertar con el siguiente elemento a la izquierda, el elemento en la posición $k-1$. Procedemos de esta forma hasta llegar a la primera posición del arreglo o encontrar un elemento que sea menor o igual al elemento a insertar.

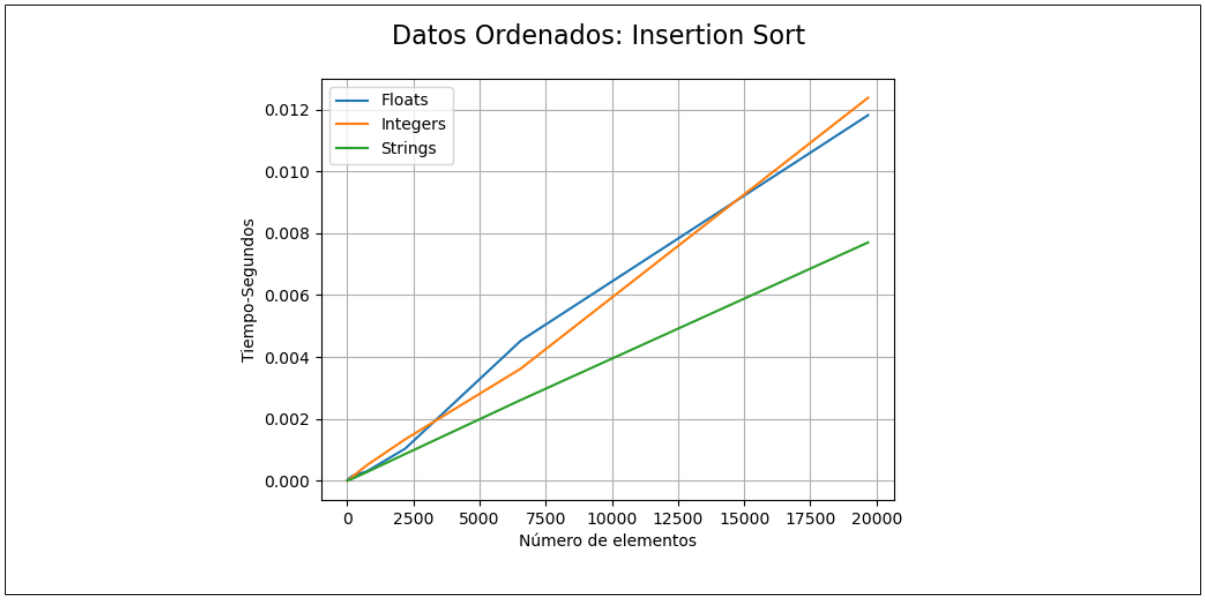

Figura 4: Tiempo de Ejecución-Datos Ordenados

Cuando el arreglo se encuentra completamente ordenado, se compara el primer elemento con el segundo y no hay intercambio, el tercero con el segundo y tampoco hay intercambio. Procediendo de esta manera hasta comparar el elemento n con el $n-1$, sin haber intercambio. Es decir, a cada paso el ciclo *while* interno no se ejecuta realizando únicamente el ciclo *for* externo. El comportamiento explicado anteriormente coincide con los resultados prácticos, teniendo la gráfica del ordenamiento por inserción, una forma de línea recta. Como en el caso anterior, realizar el ordenamiento de los arreglos tipo *string* resulta menos tardado que para los otros dos tipos de datos.

A continuación se presenta la gráfica correspondiente a este método de ordenamiento, cuando los datos se encuentran en orden inverso.

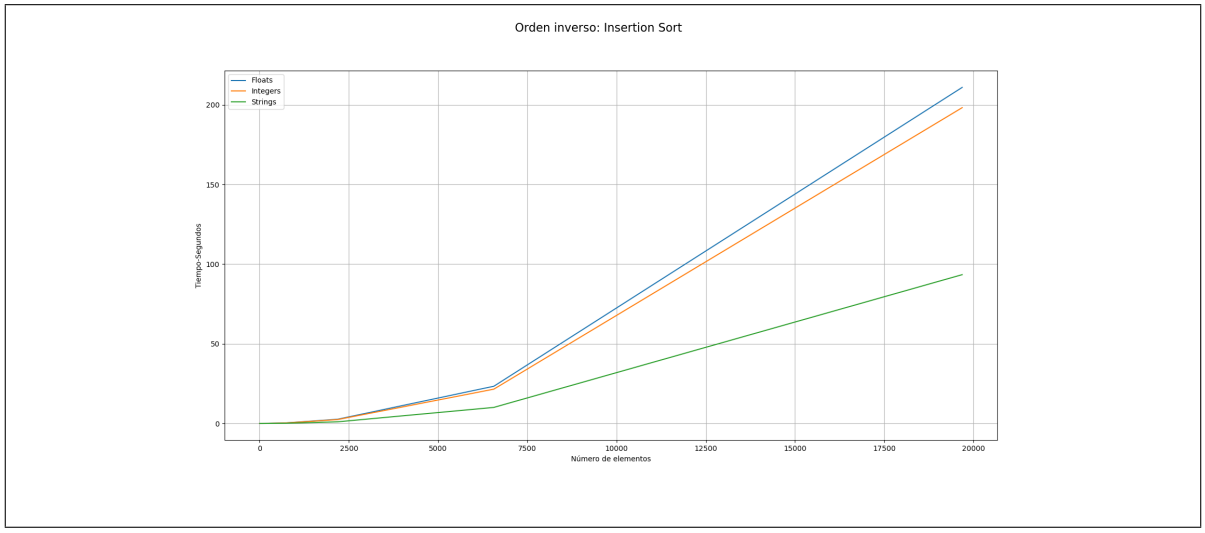

Figura 5: Tiempo de Ejecución-Datos en Orden Inverso

Algo distinto sucede cuando el arreglo se encuentra ordenado de forma inversa, pues a cada paso debemos colocar el elemento a insertar hasta la izquierda del arreglo, convirtiendo el ciclo interno *while* en un *for* realizando el mismo número de comparaciones que realiza el ordenamiento por selección directa. Este comportamiento también se ve reflejado en el cambio de tendencía de la gráfica, pasando de tener una forma de una línea recta a una de una función convexa, una función cuadrática.

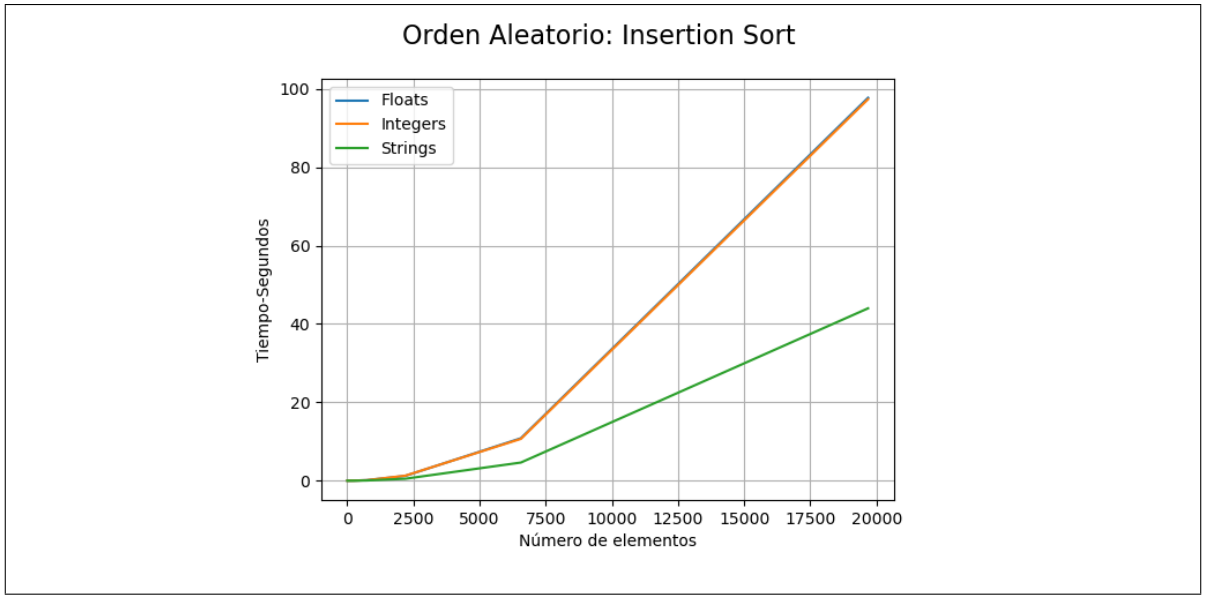

Figura 6: Tiempo de Ejecución-Orden Aleatorio

En el caso del arreglo de datos con un orden aleatorio podemos observar un comportamiento similar al que tiene este mismo método con los datos en orden inverso, es decir, tiene un desempeño similar al ordenamiento por selección directa, solo que la tendencía hacía el es un poco más lenta que en el caso anterior, para los arreglos más grandes, necesita el doble de tiempo para ordenar los arreglos en orden inverso que para ordenar aquellos en orden aleatorio. Al igual que en el método de selección directa, este método es más tardado ordenando arreglos tipo *integer* y *float* que arreglos formados por *string*, sin importar el orden de los arreglos.

Merge Sort

El desempeño de este método parece no depender del orden del arreglo de entrada. En cada uno de los experimentos, si bien los tiempos de ejecución varían un poco, la tendencia parece no cambiar en comparación con en el caso del ordenamiento por inserción directa. Pero el *merge sort* no solo es constante en su desempeño, sino que sus tiempos registrados, para cualquiera de los tres experimentos, resultan mucho menores a los tiempos promedios registrados para los dos métodos anteriores. Al igual que los dos métodos anteriores, este requiere de más tiempo para ordenar datos númericos que cadenas de caracteres.

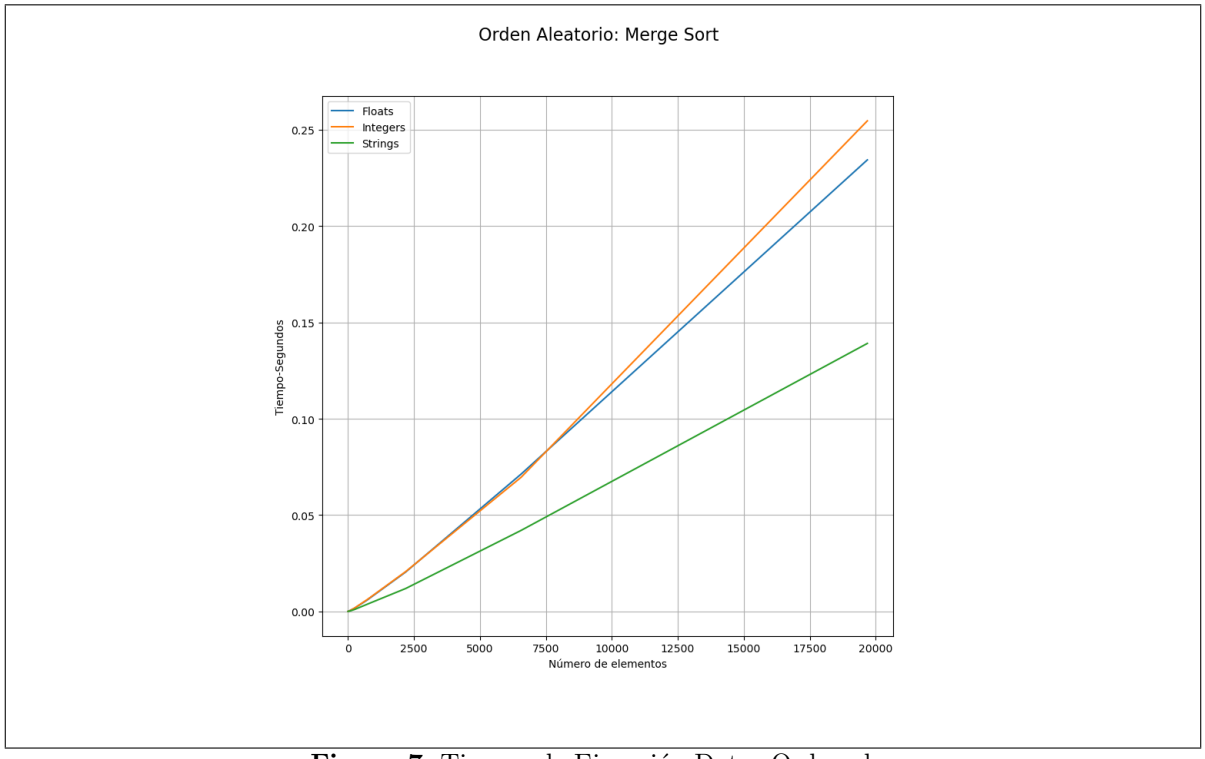

Figura 7: Tiempo de Ejecución-Datos Ordenados

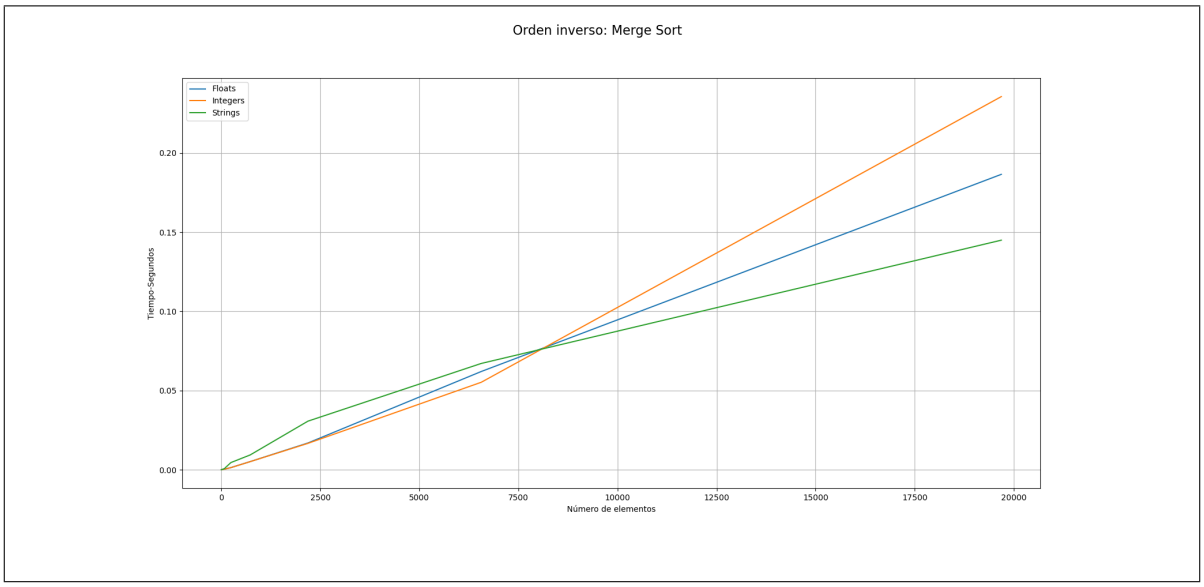

Figura 8: Tiempo de Ejecución-Datos en Orden Inverso

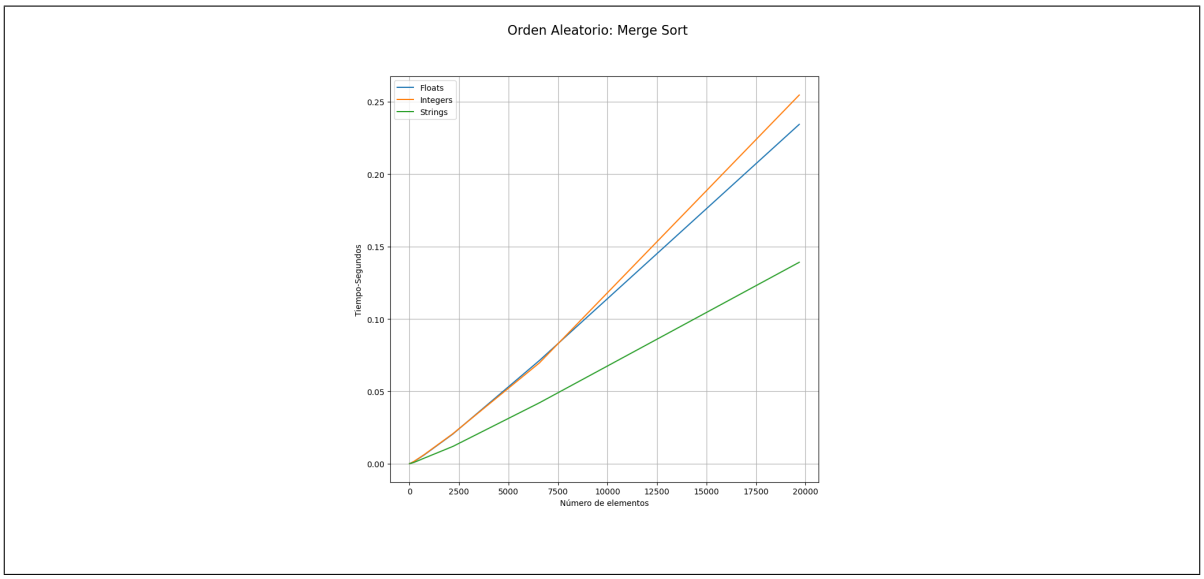

Figura 9: Tiempo de Ejecución-Orden Aleatorio

Comparación

Debido a la disparidad en el crecimiento, tanto en el número de comparaciones como en el tiempo de ejecución, en el eje y no se reporta el número de comparaciones ni el tiempo, si no $\log_{10}$ de estas mediciones, es decir si para ordenar cierto arreglo se requiere un tiempo $t = 10^p$, $p \in \mathbb{R}$, entonces $\log_{10} t = \log_{10} 10^p = p$, que es el orden de magnitud de la medición.

Datos Ordenados

A continuación se presenta el desempeño de los cinco métodos de ordenamiento juntos para arreglos de tamaño hasta 3^9.

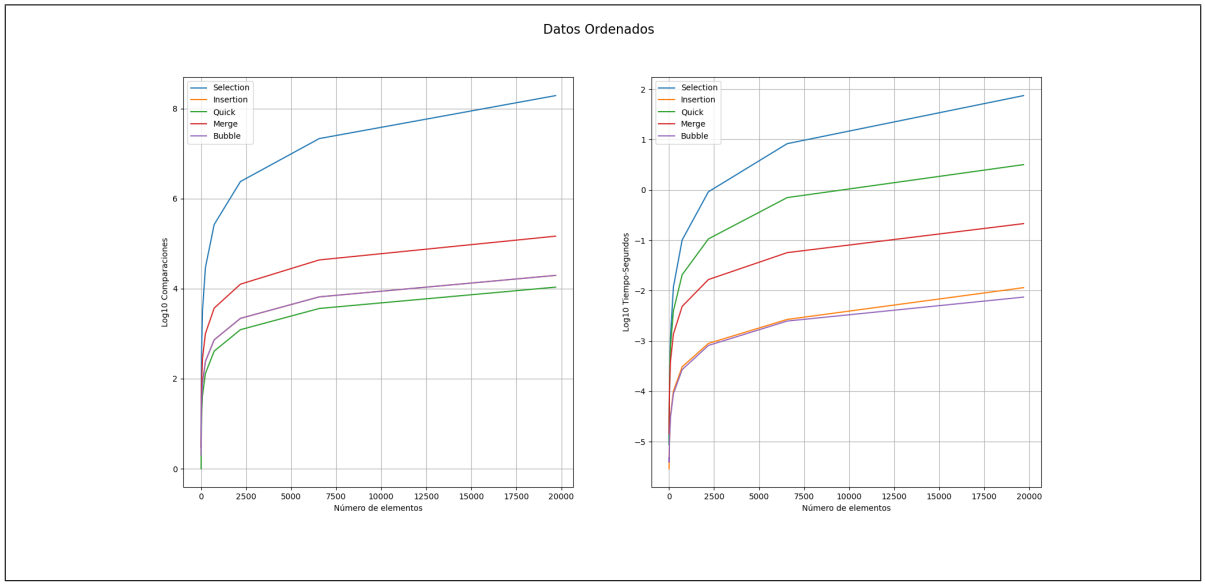

Figura 10: Número de comparaciones y Tiempo de Ejecución

De las gráficas anteriores podemos concluir algunas cosas. Lo primero a notar es que, cuando el arreglo de datos ya se encuentra ordenado, el algoritmo *bubble sort* es muy rápido. Esto se debe a que en la primera vuelta, al no haber intercambio de elementos, es capaz de darse cuenta que el arreglo ya está ordenado, por lo que si el arreglo es de tamaño n, entonces solo requiere $n - 1$ comparaciones, a saber, el primero con el segundo, el segundo con el tercero, así hasta el $n - 1$-ésimo con el n-ésimo. El bajo número de operaciones que realiza, $\Omega(n)$, es consecuencia de que solo necesita dar una vuelta al arreglo para detenerse .

Lo siguiente que podemos notar es que el número de comparaciones que realiza el algoritmo por selección directa se aleja entre tres y cuatro ordenes del número de operaciones que realizan el resto de métodos para los arreglos más grandes, es decir, el número de comparaciones que realiza este método es al menos 1000 veces más grande que el número de operaciones del resto: *merge sort* para ordenar casi $20,000$ elementos requiere aproximadamente 10^5 comparaciones; el método por selección directa

requiere 10^8 y la tendencia sugiere una disparidad cada vez mayor a medida que aumenta el tamaño del arreglo. La razón de porqué *selection sort* requiere tantas operaciones es, como se mencionó antes, a que a cada paso el algoritmo debe encontrar el mínimo de los elementos restantes, dando una vuelta completa, no hay forma que pueda darse cuenta que el arreglo está ya ordenado y termine antes, por lo que siempre realiza lo proporcional a n^2 operaciones.

Por otro lado, el algoritmo por inserción directa tiene un comportamiento similar al *bubble sort*, esto se debe a que su cilo interno *while* nunca se ejecuta y, de hecho, realiza el mismo número de comparaciones que *bubble sort*, compara el segundo con el primer elemento, el tercero con el segundo, y así, hasta el n-ésimo con el $n-1$-ésimo elemento, realizando $n-1$ comparaciones. Al igual que *bubble sort*, en la primera vuelta es capaz de darse cuenta que el arreglo está ordenado. Es por esto que, en la gráfica izquierda, las curvas correspondientes al número se traslapan, y en la de la derecha, las curvas correspondientes al tiempo de ejecución son casi iguales.

Con *quick sort* algo curioso sucede, este método no parece beneficiarse de que los datos ya estén ordenados, lo cual después de pensarlo tiene sentido, pues el mejor de los casos ocurriría si el pivote siempre quedará a la mitad del arreglo, reduciendo el problema de ordenar a la mitad a cada paso (como en el caso de *merge sort*). No solo parece no beneficiarse, sino que, como veremos más adelante, su comportamiento es similar al caso donde los elementos se dan en un orden aleatorio, la razón no es evidente, pero revisando a cada paso la forma en que se crea la partición lo podemos entender. Supongamos que queremos ordenar la lista $7, 8, 9, 10$. En el primer paso, tomamos al pivote, el primer elemento, y lo comparamos con el siguiente elemento, como este es estrictamente mayor, lo intercambiamos con el 10:

$$7, \mathbf{8}, 9, \mathbf{10} \longrightarrow 7, \mathbf{10}, 9, \mathbf{8}$$

El 10 sigue siendo estrictamente mayor a 7, entonces procedemos a realizar un cambio entre él y el 9.

$$7, \mathbf{10}, \mathbf{9}, 8 \longrightarrow 7, \mathbf{9}, \mathbf{10}, 8$$

De nuevo, notamos que el 9 sigue siendo mayor al 7, pero hemos agotado las posibilidades de cambio, entonces en ese momento paramos. A la izquierda no quedan elementos, a la derecha quedan el resto de ellos. En el siguiente paso hay que repetir el proceso con $9, 8, 10$ donde este arreglo ya **no** está ordenado y por lo tanto tendremos un arreglo a cada lado; esto solo en el primer paso para un arreglo de cuatro elementos, para arreglos más grandes, mientras se avanzan más pasos, la condición inicial de orden parece importar poco pues esta se comienza a perder desde la primera partición.
Por su parte, de *merge sort* aun no podemos decir nada destacable, si bien no parece tener un mal desempeño, este es vencido por el ordenamiento por inserción directa y el *bubble sort*.

Orden Inverso

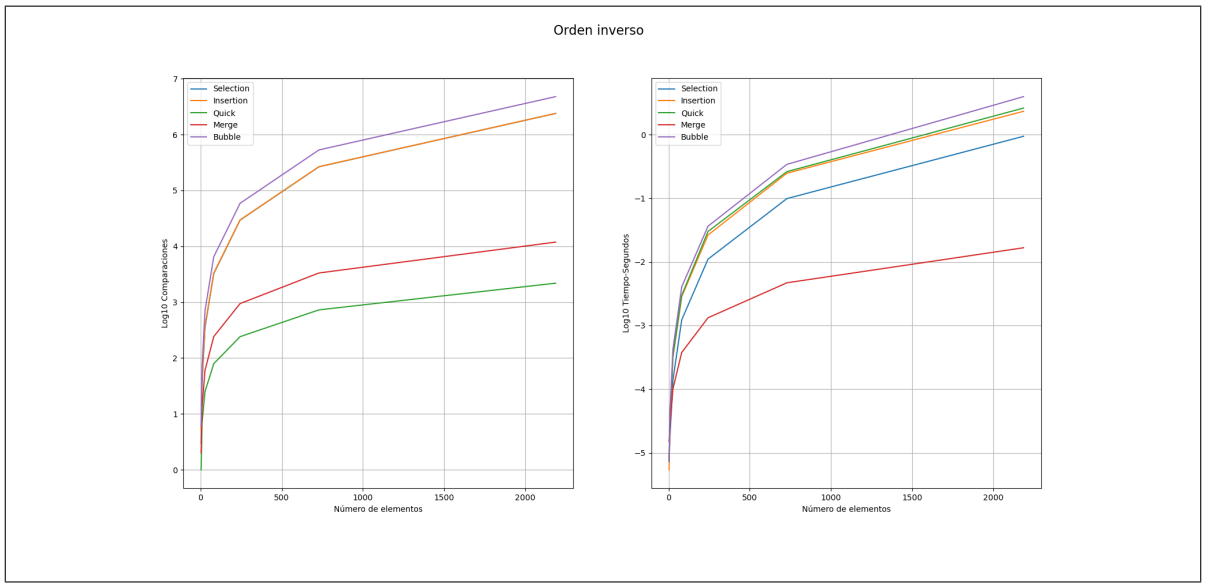

Figura 11: Número de comparaciones y Tiempo de Ejecución

Como lo vimos anteriormente, el ordenamiento por selección directa es, por decirlo de alguna manera, igual de bueno cuando el arreglo se encuentra en orden inverso que cuando el arreglo se encuentra ordenado, pero para este punto eso no es una sorpresa. Por otro lado, es remarcable como el ordenamiento por inserción y *bubble sort* presentan ambos un comportamiento muy distinto al observado cuando los datos se encuentran ya ordenados, esto es de esperarse. En el caso del ordenamiento por inserción, en cada paso, el elemento de la derecha debe pasar hasta la primera posición a la izquierda (pues este es menor a todos los elementos izquierdos), es decir, si ya hay k elementos ordenados, el siguiente elemento, $k+1$ debe compararse k veces en cada paso, por ejemplo:

$$10, 9, 8, 7$$

Comparamos 9 con 10 y hay un cambio.

$$9, 10, 8, 7$$

Ahora hay dos elementos ordenados, el 8 debe compararse primero con el 10 y luego con el 9, dos veces para llegar a la posición correcta.

$$8, 9, 10, 7$$

Tenemos en este paso tres elementos ordenados, el 7 debe compararse con el $10, 9$, y 8 en ese orden para llegar la posición correcta.

$$7, 8, 9, 10$$

Entonces, en el primer paso hay 1 elemento ordenado, se realiza una comparación, en el segundo paso hay 2 elementos ordenados, se realizan dos comparaciones hasta llegar a la posición correcta, así hasta que en el paso último paso hay $n-1$ elementos ordenados y se realizan $n-1$ comparaciones para llegar a la posición correcta, dando un total de $\sum_{k=1}^{n-1} k = 1+2+3+...+(n-1) = \frac{(n-1)(n)}{2}$ operaciones , en comparación con las $n-1$ comparaciones cuando los datos están ordenados. La causa del comportamiento de *bubble sort* es similar: a cada paso el mayor de los elementos se encuentra hasta la izquierda, para llegar al final debe pasar por el resto de los elementos: en el primer paso el primer elemento debe pasar por $n-1$ posiciones hasta llegar a la última posición, quedando como primer elemento el siguiente mayor. Este segundo mayor elemento debe pasar por $n-2$ posiciones hasta llegar a la penúltima posición, así hasta solo tener que recorrer una posición (nunca sale antes, debido al orden inverso de los datos), lo que resulta en un número de comparaciones similar a las realizadas por el *selection sort*.

Algo que ahora, con dos experimentos en condiciones distintas, es notable, es que si bien el *merge sort* no es el más rápido cuandos se trata de una lista ya ordenada, dado el cambio tan abrupto de condiciones, su comportamiento ha cambiado realmente poco, como en el caso del ordenamiento por selección, pero con número de tiempo y comparaciones considerablemente menor. Pero no solo su comportamiento permanece estable, sino que es mucho más rápido que el resto de métodos, llegando a sacar casi dos órdenes de magnitud al segundo mejor método desde arreglos de tamaño 1000. Es decir, cuando los datos se encuentran en orden inverso, a partir de arreglos de tamaño 1000, el segundo mejor método es 100 veces más tardado que *merge sort*.

También podemos ver que cuando los datos están en orden inverso el método *quick sort* desmejora bastante al punto de tener un comportamiento similar a *selection sort, insertion sort, bubble sort* que realizan n^2 operaciones. La razón para que esto sea así es la siguiente: esta implementación de *quick sort* utiliza como pivote siempre el primer elemento, que dado el orden inverso de los datos será siempre el mayor de los elementos restantes a ordenar.

$$\mathbf{10}, 9, 8, 7 \longrightarrow 9, 8, 7, \mathbf{10}$$

Si nos encontramos en el primer paso, con un arreglo de n elementos, llevar el primer elemento a su posición correcta nos toma $n-1$ comparaciones, es importante notar que para el siguiente paso solo hemos generado un arreglo de tamaño $n-1$ (solo se redujo el problema en uno), pues por el orden de los datos, no hay elementos mayores al pivote. Además, es importante notar que el orden del arreglo $n-1$ no se ha alterado, es decir, este arreglo sigue en orden inverso. En este segundo paso toca colocar el primer elemento en su posición correcta, esto nos toma $(n-1)-1 = n-2$ comparaciones, y al igual que en el paso anterior generamos un arreglo de tamaño $n-2$, de nuevo, en orden inverso. Procediendo de esta forma, reduciendo en solo uno el tamaño del problema, para ordenar la totalidad de los elementos del arreglo requerimos $(n-1)+(n-2)+...+1 = \sum_{k=1}^{n-1} k = \frac{(n-1)(n)}{2}$ comparaciones. Un detalle importante durante la ejecución del programa es que para arreglos grandes, checar los tamaños a partir de 3^8 elementos, el programa forzaba la detención del método, arrojando el error *maximum recursion depth exceeded*, lo cual quiere decir que no solo resulta ineficiente en este caso, sino que nisiquiera es capaz de completar el ordenamiento ordenamientos.

Orden Aleatorio

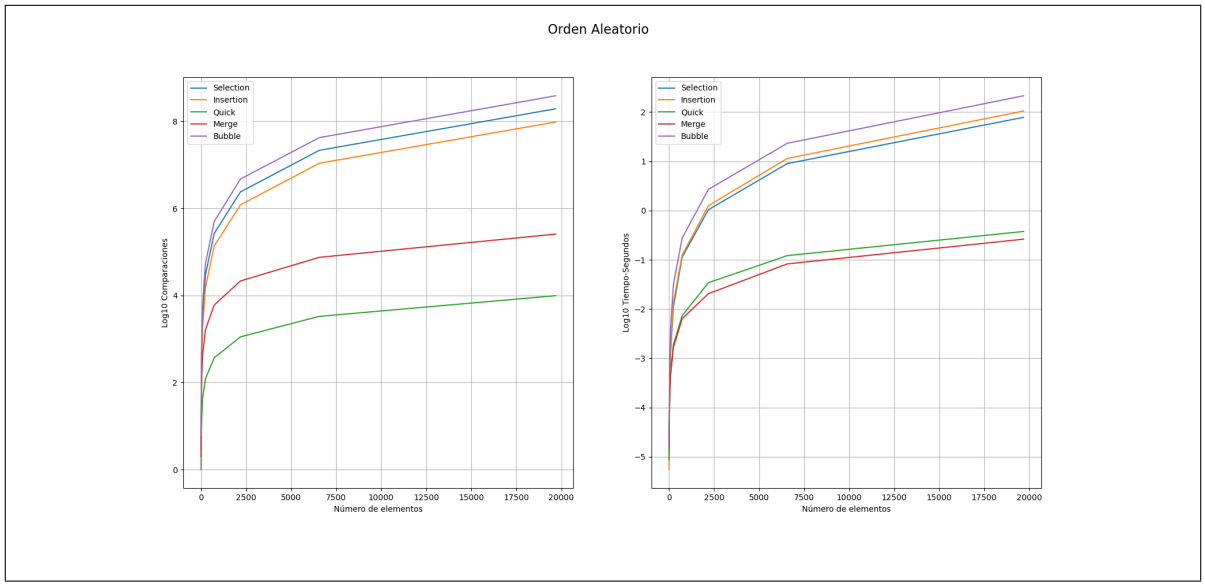

Figura 12: Número de comparaciones y Tiempo de Ejecución

El experimento con los datos en orden aleatorio es interesante, pues nos muestra el comportamiento promedio que podemos esperar de cada método. Lo primero que podemos observar es una división en dos grupos, el primero, conformado por el ordenamiento por selección directa, inserción directa y *bubble sort*, y el segundo, conformado por *merge sort y quick sort*. La razón de esta división es clara, sabemos ya que sin importar la *calidad* de los datos, el ordenamiento por selección directa realiza un número de operaciones proporcional a n^2, tanto en el mejor como en el peor de los casos. Por su parte, el ordenamiento por inserción directa y *bubble sort* tambien constan de un ciclo anidado dentro de otro, que recorerán más o menos veces según el ordenamiento previo de los datos, sin embargo, como los datos no siguen un orden específico como en los experimentos anteriores, esperamos que normalmente alcancen el ciclo anidado, realizando así lo proporcional a n^2 operaciones, de aquí que su comportamiento asintótico sea similar, pero la tendencía hacia él sea más lenta.

Por otro lado, tenemos el grupo conformado por *merge sort* y *quick sort*. De nuevo, el comportamiento de *merge sort* permanece constante dado el orden aleatorio, para ordenar los mismos $10,000$ elementos ha requerido unicamente 10^{-1} segundos, al igual que cuando los datos se encontraban ya ordenados. No solo eso, sino que como en el caso que cuando los datos se encontraban en orden inverso, es el método que mejor lo hace, sacando hasta dos órdenes de magnitud a los algoritmos del primer grupo, es decir, estos ultimos son hasta cien veces más tardados para arreglos de tamaño superior $15,000$. Por otro lado, parece que de los tres experimentos este ha sido en el que mejor se ha desempeñado *quick sort*, teniendo un comportamiento similar al de *merge sort*. Lo anterior se debe a que, como los datos están tomados en un orden aleatorio, se espera que a cada paso la distribución de elementos menores o iguales que el pivote y de elementos mayores a él sea más o menos la misma, haciendo que el arreglo se parta más o menos a la mitad a cada paso, reduciendo el problema a la mitad.

Terminados los experimentos, podemos rescatar algunas cosas importantes.

De los cinco métodos de ordenamiento, el que mejor se ha desempeñado ha sido *merge sort*: su comportamiento no solo ha sido bueno en los tres casos, sino que además ha sido constante, mostrando ser resistente ante cambios tan abruptos de las condiciones del problema.

En promedio, los algoritmos $\mathcal{O}(n^2)$ son peores a *merge sort*, ya que desde los pocos miles de elementos podemos notar que tanto la cantidad de comparaciones como el tiempo de ejecución crecen de mucho más rápido que el método ya mencionado. Sin embargo, en algo se parecen *merge sort* y el ordenamiento por selección, a saber, en la constancia de su comportamiento sin importar las condiciones del arreglo de datos, esto para *merge sort* significa no solo eficiencia, sino certidumbre, lo cual en el desarrollo de un proyecto resulta tan importante como lo primero. No así el ordenamiento por inserción y *bubble sort*, los cuales pueden son los más rápidos en darse cuenta si el arreglo está ordenado, pero también son los más lentos cuando el arreglo está en orden inverso.

Sobre *quick sort* podemos decir que es algo impredecible, debido a que este método altera a cada paso la parte del arreglo que le queda por ordenar, a tal punto que el hecho de que el arreglo se encuentre originalmente ordenado no le ayuda en nada. Además, en primera instancia resulta curioso que su mejor desempeño haya sido cuando los dartos se encontraban en orden aleatorio. También podemos ver que, en el orden adecuado, este método puede ser, por decirlo de alguna forma, tan bueno como *bubble sort*. Sin embargo, además de que en promedio su desempeño no sea malo, su implementación me parece una ingeniosa aproximación a lo que nos gustaría que hiciera un método como *merge sort*.

En síntesis, podemos decir que los resultados prácticos de los experimentos son los esperados: *merge sort* y el ordenamiento por selección no se ven afectados por el orden previo que pudiera llevar el arreglo, siendo el primero superior al segundo. Por otro lado el ordenamiento por inserción, *bubble y quick sort* son muy dependientes del orden previo del arreglo de entrada, dando pie a que sean extremadamente tardados si el arreglo está en orden inverso.

Referencias

[1] Zeit-100. (2016). Generador de strings aleatorios. 25 de febrero de 2021, de hackxcrack Sitio web: https://hackxcrack.net/foro/python/generador-de-strings-aleatorios/

[2] Python Software Foundation.. (2021). time — Time access and conversions. 25 de febrero de 2020, de Python Software Foundation. Sitio web: https://docs.python.org/3/library/time.html