Universidad Nacional Autónoma de México

Facultad de Ingeniería

(1317) Estructura de Datos y Algoritmos II

Profesor: M.I. Edgar Tista Garcia

Grupo: 05

Semestre 2024-2

Proyecto 1 - Complejidad Computacional en los Algoritmos de Ordenamiento.

Trabajo Escrito

Equipo 02

Integrantes:

Cabrera Rojas Oscar

Chavez Marquez Sergio Antonio

Noyola Torres Pablo Sebastian

Lunes 25 de marzo del 2024

1. Objetivo

Que el alumno observe la complejidad computacional de los algoritmos de ordenamiento para comparar su eficiencia de ejecución en grandes volúmenes de información.

2. Introducción

El ordenamiento desempeña un papel fundamental tanto en el ámbito de las aplicaciones computacionales como en diversos campos, ya que nos proporciona la capacidad de gestionar grandes volúmenes de información de manera eficiente, priorizando la rapidez del proceso. La operación de ordenamiento, esencial en este contexto, implica el reacomodo de una colección de elementos siguiendo una lógica o criterio predefinido, ya sea ascendente o descendente.

Sin embargo, la selección del algoritmo de ordenamiento adecuado va más allá de simplemente seguir una regla establecida. Para determinar el "mejor.ªlgoritmo en una situación dada, es imperativo considerar dos aspectos fundamentales: los datos que se van a reorganizar y los recursos de cómputo disponibles, tales como la memoria y el procesador.

En este informe presentamos los resultados de una exhaustiva comparación entre nueve distintos algoritmos de ordenamiento: Insertion, Selection, Bubble, Heap, Quick, Merge, Counting, Radix y Pancake Sort. A través de este análisis práctico de los procesos que ejecuta cada algoritmo, se puede evaluar de manera tangible su complejidad y eficiencia. De este modo, podemos determinar cuál de ellos se destaca como la mejor opción en un conjunto específico de circunstancias, contribuyendo así a una toma de decisiones informada en el diseño de sistemas computacionales y la gestión de datos.

3. Algoritmos de ordenamiento

Definimos a la operación de ordenamiento como el proceso de reacomodo de elementos de una colección para que sigan una lógica y criterio (ascentente y desdecende).

Las estretegias de construcción de algoritmos que más se utilizan en algoritmos de ordenamiento, son:

- Fuerza bruta.
- Divide y venceras.

Además, de que tienen 4 tipos de operaciones principales en estos algoritos:

1. Comparación.
2. Intercambio.
3. Insercción.
4. Intercalación.

Sabemos que para cada algoritmo de ordenamiento que se analiza se debe considerar las siguientes cuestiones:

1. **Verificación:** Sin importar la distribución del conjunto de entrada, el algoritmo siempre logra ordenarlo.
2. Tiempo de ejecución.
3. Memoria.

Estas cuestiones son en las que en este reporte de proyesto se hace más enfasis, porque durante todas las pruebas relaizadas con diferentes tamañps de valores, logramos entender la eficiencia de cada uno por el tiempo de ejecicuón que les tomaba.

3.1. InsertionSort

El algoritmo de ordenamiento interno, *Insertion-Sort*, funciona recorriendo una determinada colección elemento por elemento y conforme se avanza en la lista, cada dato se compara con aquellos que se encuentren a su izquierda.[1] En cada caso, se debe considerar que aquellos a la izquierda del elemento en cuestión ya se encuentran ordenados y lo que se pretende es poder encontrar su posición final.

Para el mejor caso, que es cuando la lista ya se encuentra ordenada y no se entre al while del código, se tiene una complejidad $O(n)$ y para el caso promedio y el peor de los casos, su complejidad es de $O(n^2)$, esto se puede ver en el comportamiento de su gráfica generada para diferentes tamaños de listas. El hecho de que su complejidad sea esa, nos indica que es un algoritmo lento para listas muy grandes y también se consideraría ineficiente.

En este caso, para realizar el conteo de operaciones de este algoritmo con diferentes tamaños de lista, con nuestra variable contadora de operaciones fuimos incrementando su valor dentro del ciclo while que se muestra en la Figura 1, ya que cada vez que se ejecuta se realiza una operación de comparación y así se obtienen el valor total de todas las operaciones realizadas en cada ordenamiento.

```
while (j>0 && aux < array[j-1]){
    ++conteo_operaciones;
    array[j] = array[j-1];
    j--;
}
```

Figura 1: Segmento de código colaborativo para la función de *InsertionSort*.

[1] Considerando que la colección se está ordenando de izquierda a derecha.

3.2. SelectionSort

El ordenamiento por selección se considera una forma "natural" de ordenar valores. Consiste en revisar todos los elementos y "seleccionar" el valor más pequeño de la colección para intercambiarlo con el primero. En cada iteración se repite el procedimiento de buscar el menor elemento y colocarlo en la posición correspondiente.

La complejidad temporal para este algoritmo es de $O(n^2)$ para todos los casos, ya que incluso para el mejor caso, que es cuando la lista ya se encuentra ordenada, en el ordenamiento siempre se tiene que buscar el elemento mínimo en cada iteración, por lo que hará el mismo número de comparaciones que cuando se encuentra en el peor, por lo que este algoritmo es demasiado ineficiente, quizás funciones para listas pequeñas, ya que de lo contrario, su proceso de ordenamiento es demasiado lento.

Para este algoritmo hicimos uso de dos variables para contar las operaciones realizadas por el ordenamiento, las cuales fueron conteo comparaciones al inicio del segundo ciclo *for* y conteo intercambios dentro del *if*, este funcionamiento se puede ver en la Figura 2. Al final se sumaron los dos valores obtenidos de las variables de conteo y gracias a esto determinamos el numero de comparaciones para los diferentes tamaños de listas con respecto a este algoritmo. El segmento de código se muestra a continuación.

```
for(int i = 0; i < size-1; i++){
  indice_menor = i;
  for(int j = i+1; j < size; j++){
  ++conteo_comparaciones;
  if(array[j] < array[indice_menor])
    indice_menor = j;
  }
  if(i != indice_menor){
    Utilerias.swap(array, i, indice_menor);
    ++conteo_intercambios;
  }
}
```

Figura 2: Segmento de código del algoritmo *SelectionSort*.

3.3. BubbleSort

En *bubblesort*, de la lista a ordenar se verifica cada dato en relación con el elemento inmediato siguiente, intercambiándose de posición si se encuentra en un orden inverso al requerido.

En este algoritmo se implementó una modificación con la variable k para que el algoritmo sea sensible a cuando el arreglo ya se encuentra ordenado y no tenga que hacer por consiguinte todas las iteraciones, ésto con el fin de mejorar su complejidad, teniendo que para el mejor de los casos, sea de $O(n)$, y para el caso promedio y el peor de los casos es $O(n^2)$, por lo que también es muy ineficiente para ordenar colecciones de muchos elementos.

En est algoritmo, para poder observar su comportamiento en la gráfica generada, contamos con la variable *conteo_comparaciones* en el segundo *for* las comparaciones realizadas por los elementos de la lista y con la variable conteo intercambios, los correspondientes intercambios realizados en total por el algoritmo después se sumarón los valores de estos contadores. Lo anterior se puede apreciar a continuación, en la Figura 3.

```
int k = 1;
for(int i = size-1; i > 0; i--){
  if(k == 0)
    break;
  k = 0;
  for(int j = 0; j < i; j++) {
    ++conteo_comparaciones;
    if(array[j] > array[j+1]){
      Utilerias.swap(array, j, j+1);
      ++conteo_intercambios;
      k = 1;
    }
  }
}
conteo_operaciones = conteo_comparaciones + conteo_intercambios;
```

Figura 3: Segmento del código de *BubbleSort*.

3.4. HeapSort

HeapSort entra en la lista de los *Algoritmos de Selección*, teniendo la siguiente funcionalidad:

1. Convertir colección a ordenar en un *heap*.
2. En cada iteración, eliminar la raíz (tenemos en cuenta que el proceso de eliminación conlleva la reconstrucción del *Heap*, porque este siempre debe mantener su integridad).

En otras palabras, el algorimo consiste en la construcción de un árbol binario llamado *Heap*, en donde se harán las comparaciones/intercambios con una parte de los elementos de la colección, y finalmente se recorre linealmente todos los elementos al intercambiar con la última posición.

Teniendo en cuenta el funcionamiento del algoritmo y entendiendo los procedimientos que realiza, es como logramos determinar que deberían incluirse dos contadores

en nuestro código, el primero ubicándolo cuando se construye el *Heap* en el ciclo *for* dentro del método *buildHeap*, y el segundo contador cuando se extraen los elementos y se reconstruye el *heap* hasta que ya no haya más elementos, esto en el método *heapify*.

Cabe resaltar que la complejidad de este algoritmo para todos sus casos es de $O(n \log n)$.

3.5. QuickSort

El algoritmo *QuickSort* utiliza el método de intercambio para la realización del reacomodo de los valores y, tomando en cuenta el tiempo y la memoria de una computadora, se termina considerando como uno de los mejores algoritmos de ordenamiento; teniendo tanto para el mejor caso, como para el promedio una complejidad de $O(n \log(n))$ y para su peor caso una complejidad de $O(n^2)$.

Este algoritmo emplea como estrategia general:

1. Seleccionar un elemento de la colección [pivote].
2. Verificar este pivote en relación con el resto de los elementos y realizar los intercambios, de tal manera que, aquellos que se encuentren a su izquierda sean menores o igual, y aquellos a la derecha mayores.
3. Repetir lo anterior en los conjuntos de datos que se generan a la izquierda y a la derecha.

Para la implementación del código en este proyecto, utilizamos el repositorio anteriormente entregado por el profesor durante las prácticas, y las únicas modificaciones que realizamos fueron al momento de inicializar el algoritmo:

Implementamos el método: *iniciarQuickSort(int[] array, int low, int high)*, que declara el contador en 0, y de esta manera cuando el algoritmo aplique la recursividad, no se vea afectado el conteo de iteraciones que este realiza, como se puede ver en la Figura 4.

```
public static int iniciarQuickSort(int[] array, int low, int high){
  conteo_operaciones = 0;
  quickSort(array, low, high);
  return conteo_operaciones;
}
```

Figura 4: Código del método: *iniciarQuickSort* en la clase *QuickSort*.

El conteo de operaciones se realiza en el método *partition*, dentro de la estructura de selección *if*, y también hace el conteo en la función *quickSort*, para cada llamada recursiva.

3.6. MergeSort

El algoritmo de ordenamiento *Merge-sort* sigue la lógica *divide y vencerás* y su funcionamiento radica principalmente en combinar dos listas que ya han sido ordenadas para obtener una respectiva lista ordenada más grande.

La estrategia general para este algoritmo es la siguiente:

1. Dividir la sublista de n elementos en dos sublistas.
2. Ordenar las dos sublistas utilizando *merge-sort*.
3. Combinar(intercalar) las dos sublistas ordenadas para obtener la lista ordenada final.

La complejidad de este algoritmo es de $O(n \log(n))$ para todos los casos, lo que lo hace ser rápido con la desventaja de que este algoritmo hace uso de memoria adicional, para tener un arreglo o lista auxiliar para hacer el acomodo de las sublistas que se van generando.

Para contar las operaciones realiadas por este algoritmo, se agrego la variable *conteo_ operaciones* dentro de los ciclos *while* por la comparaciones realizadas y dentro del método *mergeSort* por cada vez que se llame a dicho metodo y contar estas operaciones, ésto se puede ver en la Figura 5 y 6.

```java
public static void mergeSort(int[] array, int first, int last){
    conteo_operaciones++;
    int mid;
    if(first < last){
        mid = (first+last)/2;
        mergeSort(array, first, mid);
        mergeSort(array, mid+1, last);
        merge(array, first, mid, last);
    }
}
```

Figura 5: Segmento de la clase *MergeSort*.

```java
public static void merge(int[] array, int first, int mid, int last){
    int[] array2 = new int[last-first+1];
    int i=first, j=mid+1, k=0;
    while(i <= mid && j <= last){
        conteo_operaciones++;
        if(array[i] < array[j])
            array2[k++] = array[i++];
        else
            array2[k++] = array[j++];
    }
    while(i <= mid){
        array2[k++] = array[i++];
        conteo_operaciones++;
    }
    while(j <= last){
        array2[k++] = array[j++];
        conteo_operaciones++;
    }
    for(i = last; i >= first; i--)
        array[i] = array2[--k];
}
```

Figura 6: Segmento del método *merge* de *MergeSort*.

3.7. CountingSort

El algoritmo de ordenamiento *CountingSort* no partió de un código brindado, sino que se propuso su implementación en su totalidad.

Este ordenamiento destaca en su funcionamiento puesto que requiere una gran cantidad de memoria adicional para ser ejecutado, si bien como se verá más adelante funciona de una menra sumamente rápida, todo este almacenamiento adicional a la colección original lo vuelven una opción no preferida y en algunos casos inviable.

La primera colección adicionarl debe ser del mismo tamaño que la original, en la implementación esta colección se llama *sorted*, la segunda es una colección que debe tener por tamaño el número o rango de elementos que pueden aparecer a ordenar. Por ejemplo, si se quieren ordenar número del uno al cinco, el tamaño de esta colecccón será de cinco, pues los posibles elementos pueden ser 1, 2, 3, 4 ó 5; pero si se quieren ordenar números del 10 al 13, el tamaño de la colección será de 4, pues los elementos se pueden encontrar entre el 10, 11, 12 y 13. En la implementación, dicha colección se llama *counting* y su tamaño está dado por la constante *RANGO* definida en el archivo *Ordenamientos.java* con el valor fijo de 99999.

El ordenamiento original divide su funcionamiento en tres etapas muy claras:

1. Recorrer toda la lista (arreglo) de atrás para adelante[2] haciendo una primera pasada a todos los elementos de la lista, e incrementando en una unidad su correspondiente espacio en el arreglo *grande* (*counting* para el caso) por cada aparición que tenga dicho elemento.

2. Recorrer ordenadamente el arreglo contador (*counting* para el caso), reescribiendo cada elemento por su suma con todos los elementos anteriores en el arreglo. Ésto permite obtener al instante la posición final de dicho elemento al momento de escribir de nuevo la colección esta vez ordenada.

3. Recorrer una segunda vez la colección original, esta vez de adelante para atrás, y colocar cada elemento en su posición correcta en la colección del mismo tamaño a la original (*sorted* para el caso). Para conocer la posición final se accede al número que tiene relacionado en el arreglo contador (obtenido en el paso dos) decrementado en uno, si la posición no está vacía significa que otro mismo elemento como él ocupa el espacio y decrementa otra unidad la posición hasta encontrar finalmente un espacio vacío. Finalmente copia el arreglo ordenado en el original.

Estos tres puntos del funcionamiento, tal como se describieron son implementados cada uno en un método específico:

1. count(int[] array, int[] counting, int size)

2. suma(int[] counting, int numElements)

3. sort(int[] array, int[] sorted, int[] counting, int size)

[2]Siguiendo la suposición general de que todos los arreglos se quieren ordenar de manera ascendente.

3.8. RadixSort

El funcionamiento de *RadixSort*, al igual que *CountingSort*, es un tanto alternativo al resto de los presentados, pues no se basan en las operaciones de comparación e intercambio sino de una manera más *innovadora*.

La lógica original de este ordenamiento consiste en tomar cada dígito de cada elemento en la colección y, dependiendo de su valor, agregarlo a una cola (*queue*) con el valor asignado, es decir, que existe una cola para cada número natural (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Una vez encolados todos los elementos, se deben desencolar todas las colas de manera ordenada, es decir, primero la del uno, luego la del dos, luego la del tres... Esta operación se repetirá para todos los dígitos de todos los elementos en la colección.

De nuevo, este ordenamiento aunque sí destaca por la velocidad temporal al momento de trabajar con grandes volúmenes de información, tiene también un alto costo en materia de almacenamiento adicional equivalente al doble de la colección original, pues en cada iteración el contenido total del arreglo original será repartido en todas las colas, que a su vez reescribirán la colección original.

La implementación para este proyecto inicia con la declaración de las diez colas, y obtiene el número de digítos del elemento más grande en la lista mediante la invocación a las funciones *getMax* y *getIterations*[3].

El algoritmo funciona en un ciclo *for* para cada dígito, en el que recorre cada vez la lista y obtiene el dígito correspondiente de los números (haciendo división entre 10, 100, 1000 y así sucesivamente) y añade el dígito a la cola que le corresponde. Finalmente reescribe el arreglo original vaciando el contenido de las colas en orden. El contador se incrementa en el segundo ciclo *for*, cuando recorre *iteraciones* veces los elementos del arreglo. Para este caso el número más grande es 99999, por lo que el número máximo de dígitos será de cinco, y posteriormente en la gráfica se debería de ver su gráfica como $5n$, siendo n el número del tamaño del arreglo.

3.9. PancakeSort

Para este proyecto, se decidió agregar el algoritmo de ordenamiento PancakeSort.

PancakeSort es un algortimo de ordenamiento, en el cual su funcionamiento se asemeja a la actividad de voltear hotcakes o panqueques, pues el procedimiento de este algoritmo consiste en voltear segmentos de lista, tal y como se haria con una porción de panqueques.

Este algoritmo, encuentra al elemento de mayor valor en la lista y posteriormente lo coloca en su posición correspondiente mediante operaciones de volteo. El proceso anterior se repite con los elementos mayores en la lista, hasta que se ordene en su totalidad dicha colección.

[3]Ésto permite que el programa funcione correctamente ante cualquier arreglo de entrada, siempre que sea del mismo tipo, sin importar el tamaño de los números contenidos en el arreglo.

Es importante mencionar, que para este algoritmo, a diferencia de otros donde se busca hacer el menor número de comparaciones, en este la operación principal es el volteo de los segmentos de lista.[3]

Para llevar a cabo la implementación de este algoritmo, con ayuda del material de apoyo obtenido de fuentes de internet[1], encontramos un código del algoritmo en el lenguaje de programación C++ y analizando su estructura, lo adaptamos a nuestro proyecto realizado en el lenguaje de programación Java, satisfaciendo nuestras necesidades, en referencia al ordenamiento de listas de diferentes tamaños.

Respecto a la complejidad espacial de este algoritmo, esta se denota por $O(1)$ ya que no ocupa memoria adicional, sin embargo, en lo que refiere a su complejidad temporal, para el mejor de los casos es $O(n)$, pero para el caso promedio y el peor de los casos, la complejidad es de $O(n^2)$, pues el algoritmo en un proceso de ordenación de una determinada lista, puede llegar a requerir realizar hasta dos operaciones de volteo por elemento, lo que lo hace ser muy lento para ordenar listas de tamaños de elementos muy grandes y en consecuencia se vuelve ineficiente su uso.[1]

Una manera fácil de entender el funcionamiento general de este algoritmo la ofrece el video *Pancake sort algorithm, visualization with VTK* (Mario Storti, 2009)[4].

Más adelante, gracias a la correspondiente grafica generada entre los algoritmos de ordenamiento de complejidad cuadrada, pudimos apreciar su correspondiente comportamiento para listas de tamaños muy grandes, así como ver que algoritmos son mejores o peores en relación a PancakeSort.

Para este algoritmo, incrementando la variable *conteo_ operaciones* se contaron sus correspondientes operaciones realizadas como se muesta en la Figura 7, que es donde se realizan los respectivos volteos del algoritmo.

```
while (start < i) {
    conteo_operaciones++;
    temp = arr[start];
    arr[start] = arr[i];
    arr[i] = temp;
    start++;
    i--;
}
```

Figura 7: Segmento de la clase *PancakeSort*.

4. Análisis de resultados

Al realizar el análisis de complejidad en la reiterada ejecución de estos algoritmos de ordenamiento, se buscan ciertas formas específicas en los trazos que deberían tener. Al momento de decidir qué algoritmos graficar juntos se tomó la decisión de poner en el mismo cuadro aquellas que tengan un comportamiento téorico similar, de manera que:

- $\mathbf{O(n^2)}$
  InsertionSort, *SelectionSort*, *BubbleSort*, *PancakeSort*.
- $\mathbf{O(n\log(n))}$
  QuickSort, *HeapSort*, *MergeSort*.
- $\mathbf{O(n+k)}$ & $\mathbf{O(n+k)}$
  CountingSort y *RadixSort*.

Se decidió añadir, junto a los trazos de cada ordenamiento, en su respectivo cuadro un trazo extra de control, en el cual se muestran la gráfica cuya forma debe ser esperada: $y = x^2$, $y = x\log(x)$ y $y = x$ respectivamente. De manera adicional, hizo uso de la aplicación *GeoGebra* para tener también a la mano las gráficas correspondientes y la forma en la que idealmente se deberían de ver, la primera y tercera se muestran en la Figura 8 y Figura 9, respectivamente.

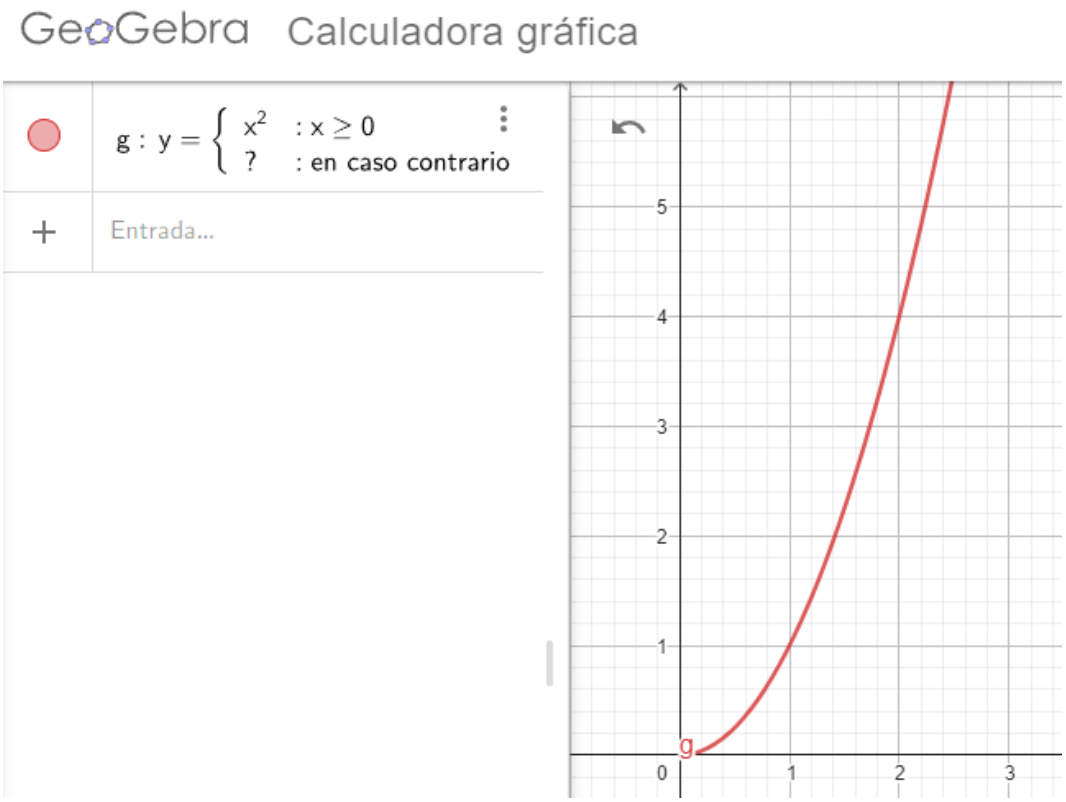

Figura 8: Gráfica teórica para $y = x^2$ obtenida en *GeoGebra*.[2]

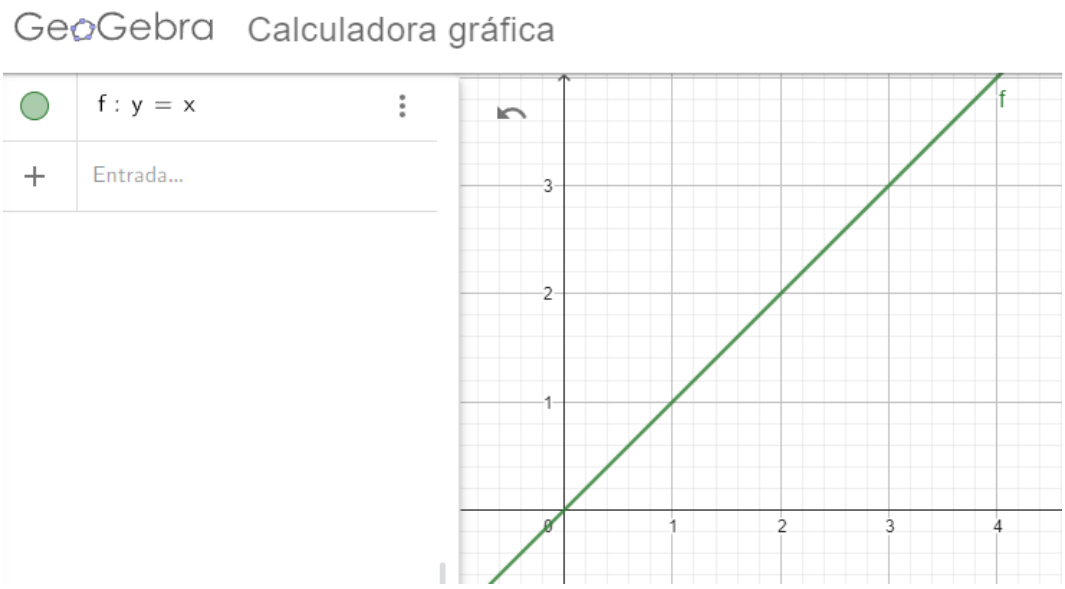

Figura 9: Gráfica teórica para $y = x$ obtenida en *GeoGebra*.[2]

Ante la incertidumbre de cómo se debe de generar la gráfica de $n\log(n)$, del mismo modo que en las gráficas anteriores, se hizo en el software de *GeoGebra* la gráfica buscada para la referencia precisa de cómo se debería de ver, dicha imagen se encuentra en la Figura 10.

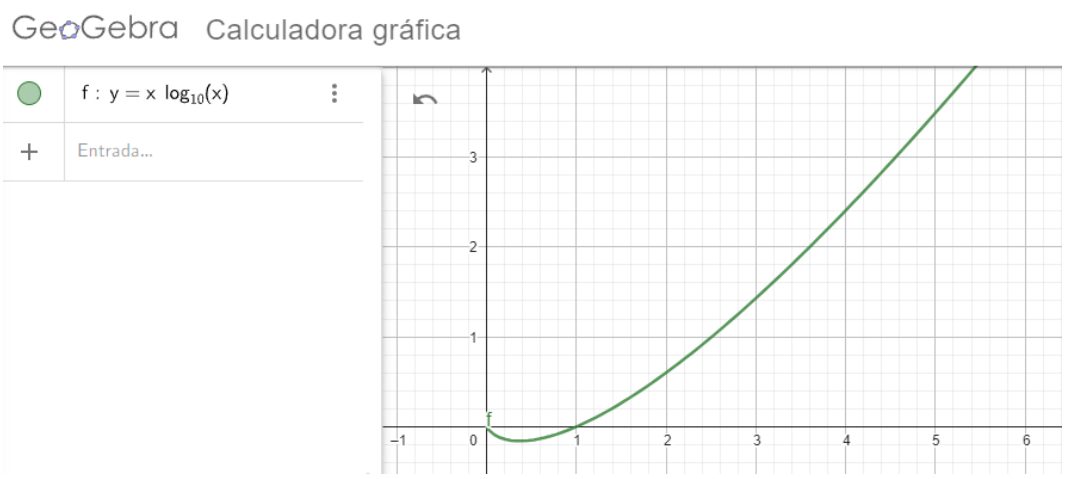

Figura 10: Gráfica teórica para $y = xlog(x)$ obtenida en *GeoGebra*.[2]

Y teniendo en mente ahora la idea de cómo se debería de ver, se ejecutó en el archivo *GraficarLogaritmo.py* la misma gráfica desde a partir de un centésimo y hasta cinco mil, como se puede ver en la Figura 11.

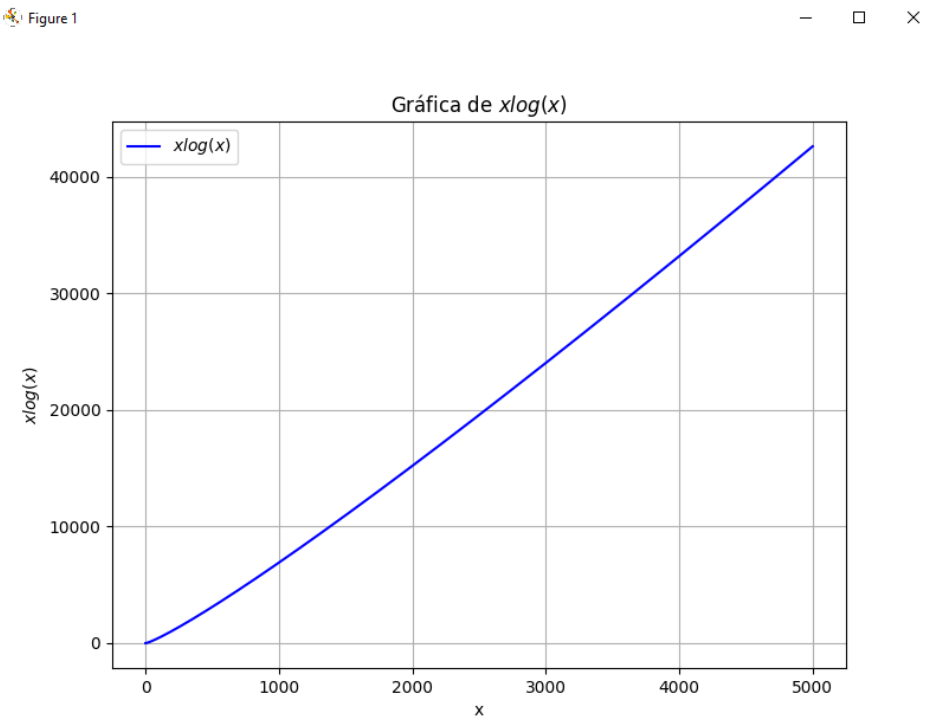

Figura 11: Gráfica comparativa para $y = xlog(x)$ obtenida en *GraficarLogaritmo.py* desde un valor mínimo de 0.01.

Y teniendo la muestra de lo que se debería esperar ver en las gráficas capaces de realizar de primera mano, se hizo una ejecución donde el tamaño de entrada fuera cinco mil, ésto con el propósito de que la entrada (y por tanto el eje x) de ambas gráficas pudiera coincidir. El resultado se muestra en la Figura 12.

Dándose cuenta de que, aún así, la gráfica obtenida al realizar los ordenamientos difiere levente de lo esperado, aunque este hecho pudo ser fácil de explicar al contrastar el tamaño mínimo de entrada, en la gráfica de muestra se tiene como valor mínimo en el eje x uno muy pequeño, de 0.01, mientras que la entrada mínima del programa es de 200, intervalo decidido así por conveniencia en los distintos tamaños. Al darle una entrada a *GraficarLogaritmo.py* igual a la del programa principal, de 200, se obtuvo la gráfica que se ve en la Figura 13.

Es así como se logró ganar la fiabilidad de la correcta graficación el programa desarrollado en Python para mostrar las gráficas, siendo que, bajo las mismas condiciones, ambas gráficas se asemejan de una manera en la que se ven casi exactamente iguales, indicando que el trazo en el gráfico de ordenamientos de control ($y = xlog(x)$) es correcto en su comportamiento y capaz de ser un punto de control para el trazo obtenido en la imagen de las gráficas con los volúmenes de información más grandes realizados.

En la siguiente gráfica (*Figura 14*) se muestran los Algoritmos de Ordenamiento de *Insertion*, *Selection*, *Bubble* y *Pancake*. Estos algortimos se recopilaron en una misma gráfica porque en el caso promedio todos tienen una complejidad $O(n^2)$.

Para la construcción de esta gráfica consideramos que se ordenaron 20,000 elementos en cada uno de los algoritmos, y podemos apreciar que con esta cantidad de elementos se llegó a realizar en el peor algoritmo (BubbleSort) hasta 3×10^8 operaciones, mientras

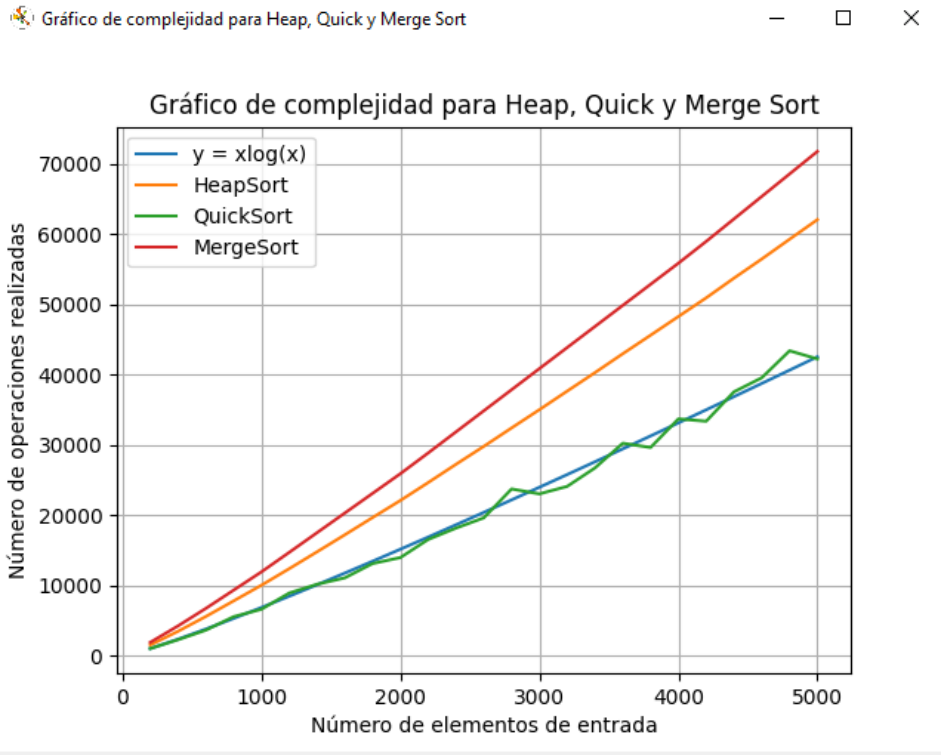

Figura 12: Gráfica obtenida experimentalmente para $y = xlog(x)$ obtenida en *Graficar.py* con una entrada de cinco mil.

que el mejor algoritmo (InsertionSort) realizó 1×10^8 operaciones.

Cabe destacar que en la gráfica se colocó el valor teorico $y = x^2$ para tenerlo como referencia y así poder hacer la comparación con los resultados de los algoritmos. De esta manera, al tener una similitud en la graficicación entre el valor teorico y los resultados de los algoritmos, podemos afirmar que todos cumplen con la complejidad $O(n^2)$.

En esta gráfica al igual que en la anterior se muestra el comportamiento de los algoritmos Insertion, Selection, Buble y Pancaker sort, pero en este caso, ordenando listas de hasta un tamaño 100,000 elementos, ésto con la finalidad de poder corroborar para listas de tamaños mucho más grandes, qué algoritmo es el que termina siendo más eficiente o ineficiente en su defecto, visualizando cuál tarda más en ordenar la lista según el tamaño.

Como se puede apreciar, de todos los algoritmos, según sus correspondientes comportamientos, Bubble es el ordenamiento que más se aproxima al comportamiento teórico de $O(n^2)$ y la que menos se parece a dicho comportamiento teórico es la de insertion, por lo que comparando a estos cuatro algoritmos, posiblemente el mejor de ellos es InsertionSort. Figura 15.

Para la siguiente gráfica, se presentan los respectivos comportamientos de los algoritmos Heap, Quick y Merge sort, para listas que van hasta un tamaño de 20,000 elementos. Como se puede observar, estos algorimos tenen un comportamiento con complejidad $O(n \log(n))$, siendo quick, la que se comporta mejor de las tres y, en consecuencia, es más rápida a la hora de realizar los ordenamientos.

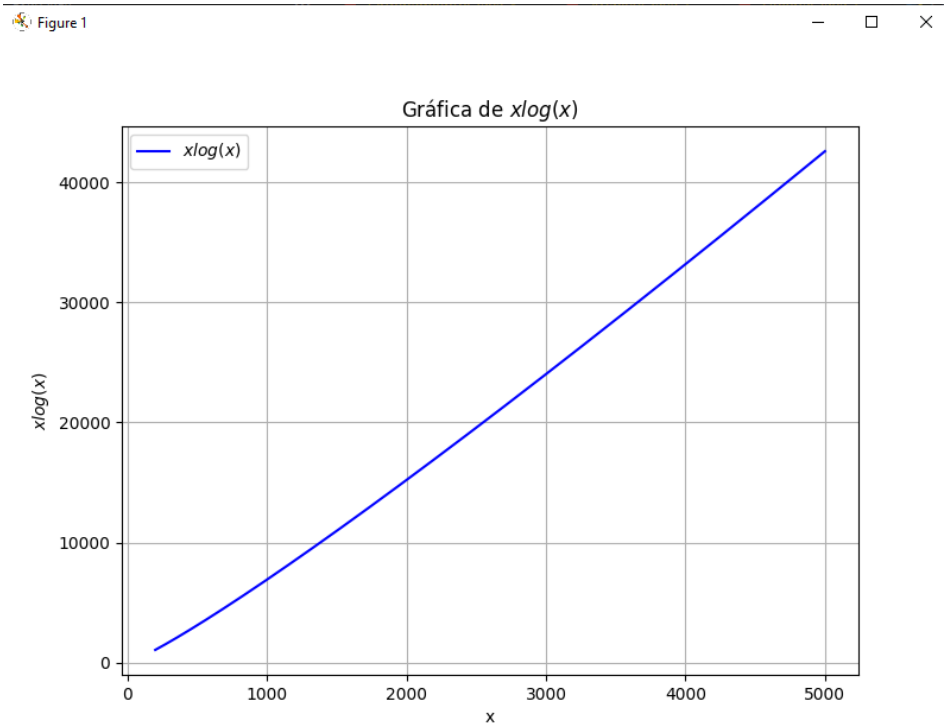

Figura 13: Gráfica comparativa para $y = xlog(x)$ obtenida en *GraficarLogaritmo.py* desde un valor mínimo de 200.

Por otro lado, primero se graficó probando con esta cantidad de elementos, para posteriormente visualizar qué tanto difiere o no con listas de tamaños aun más grandes y así verificar qué algoritmo es el mejor.

Continuamos con la gráfica que compara los algoritmos: Heap, Quick y Merge (*Figura 17*). En este caso, se realiza un ordenamiento de 100,000 elementos, siendo el mejor algoritmo QuickSort, que realiza un aproximado de 1.2×10^6 operaciones, seguido de HeapSort que realiza un aproximado de 1.6×10^6 operaciones, y finalmente como peor algoritmo de esta comparación, tenemos a MergeSort que realiza aproximadamente 1.8×10^6 operaciones.

Recordamos que para el caso promedio de estos algoritmos, su complejidad es de $O(n \log(n))$.

Para esta gráfica, se presentan los algoritmos Couting y Radix sort, ordenando listas de elementos de hasta tamaño 20,000. Como se puede visualizar, se cumple con su compleidad que es de $O(n + k)$ y $O(nk)$ respectivamente.

Se puede visualizar que ambas siguen su correspondiente comportamiento esperado y se parecen a la gráfica teórica. Otro punto importante a destacar es que se puede observa cómo es que Radix es peor que Couting, pues el número de operaciones que realiza el primero es mucho mayor que el de couting, lo que nos puede ayudar a entender que en este caso, termina siendo mejor CoutingSort.

Se optó por realizar esta gráfica para primero ver su comportamiento con una lista de elementos de tamaño menor, para posteriormente apreciar como se comportan los

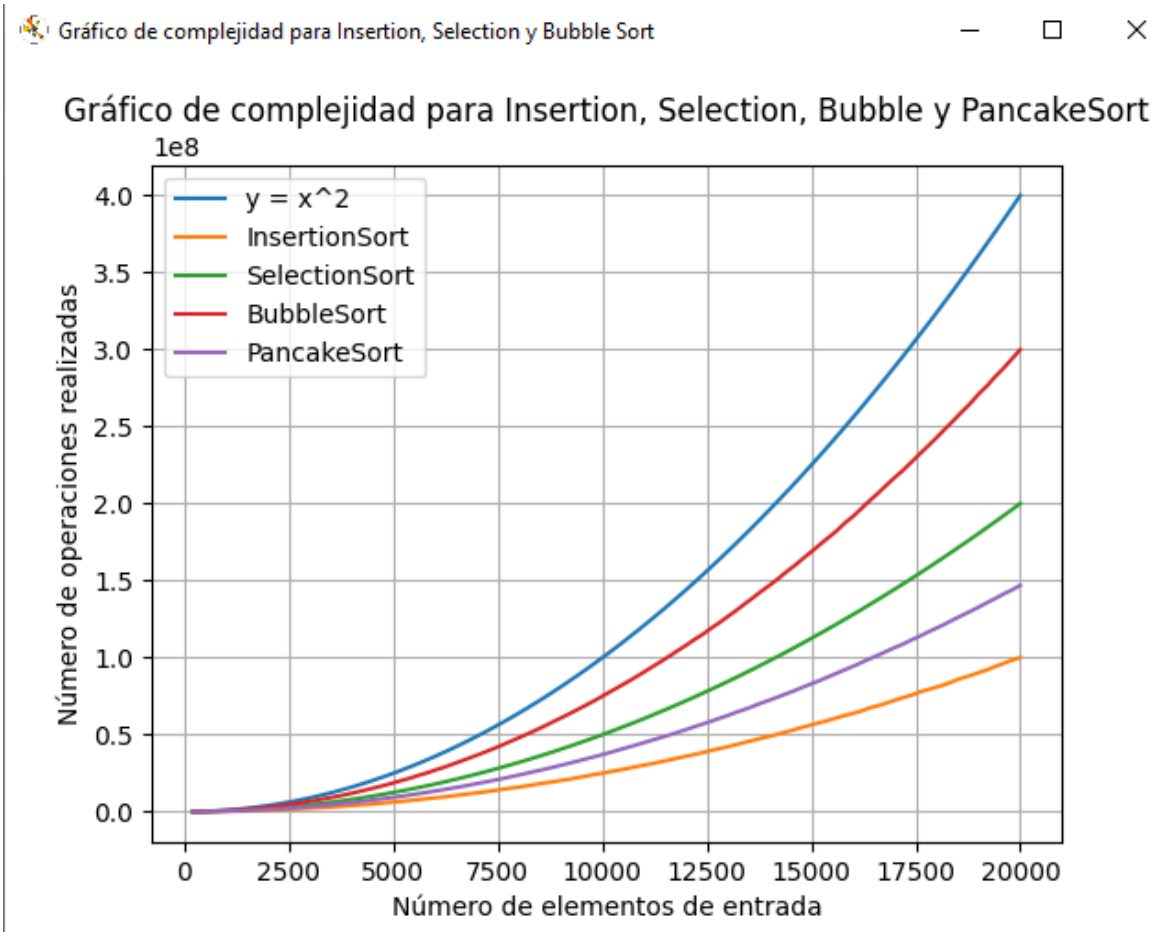

Figura 14: Gráfica de ordenamientos con complejidad $O(n^2)$ para una entrada de veinte mil elementos.

algoritmos con listas mas grandes. Figura 18.

Ahora la gráfica (*Figura 19*) que de igual manera compara los algoritmos de Countig y Radix Sort, ordenan 100,000 elementos. Y el resultado que se muestra nos hace corroborar que por más elementos que ordenen siempre será CountingSort el algoritmo más rápido, ya que este realizó 200,000 operaciones, mientras que RadixSort realizó 500,000 operaciones.

Cabe recodar que estos algoritmos tienen una complejidad en su caso promedio de $O(n+k)$, que de igual manera se muestra como referencia en la gráfica de color azul.

En la gráfica actual, se junto a todos los algoritmos estudiados en el proyecto, realizandoles pruebas de ordenamiento para listas de hasta 20,000 elementos. Lo anterior fue con motivo a poder observar en general a todas las gráficas y asi que algoritmo es más eficiente.

Como se puede apreciar, al variar sus complejidades, todos los que no sean O(n^2) prácticamente no es posible verlos y únicamente se aprecian los que sí pertenezcan a dicha complejidad, por lo que podemos ver que, comparando a todos los algoritmos, aquellos que se pueden visualizar su complejidad cuadratica son los peores, tal es el caso de bubble, selection, pancake e insertion sort. Figura 20.

En esta gráfica (*Figura 21*), también tenemos todos los algoritmos analizados en

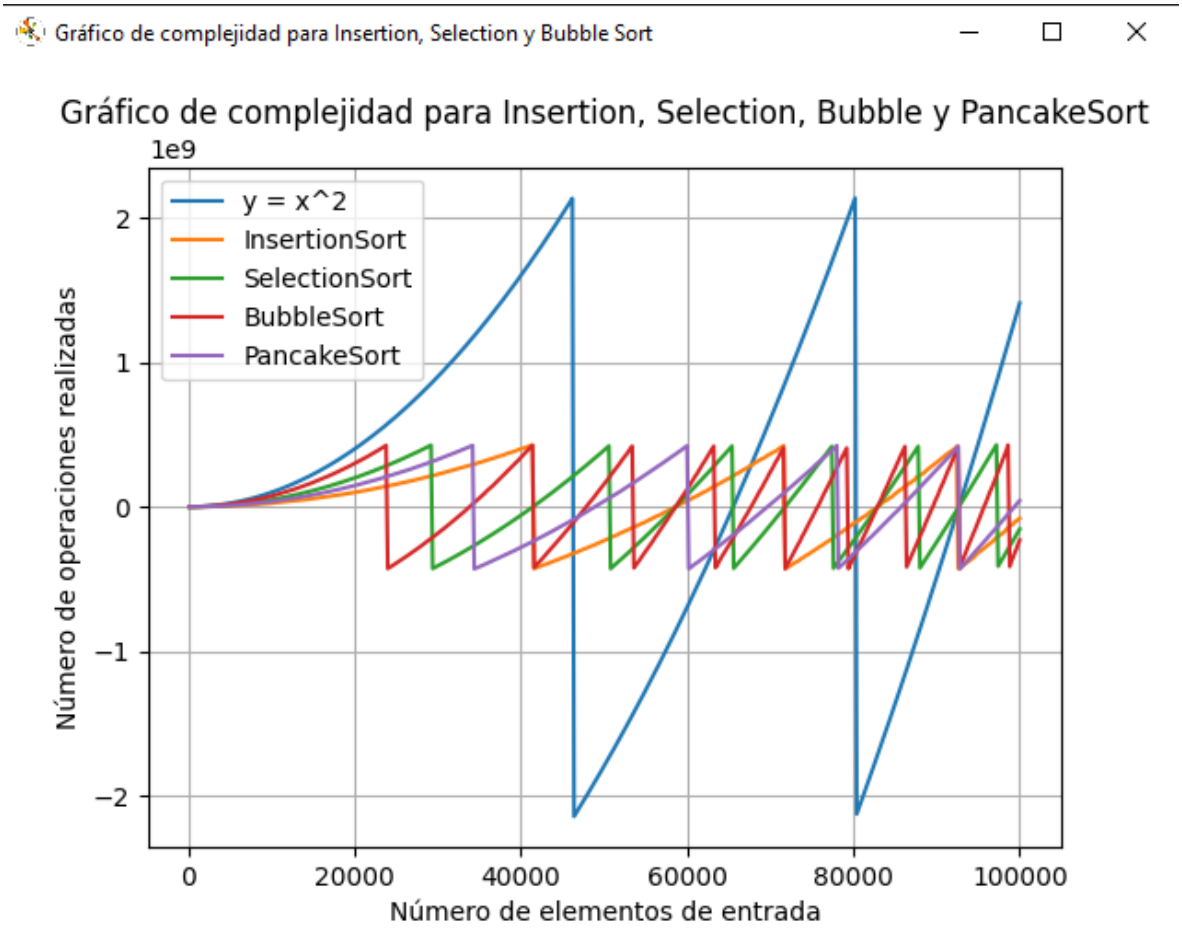

Figura 15: Gráfica de ordenamientos con complejidad $O(n^2)$ para una entrada de cin mil elementos.

este proyecto, solo que en esta ocasión, el número de elementos a ordenar es de 100,000.

Obvervando lo que sucede con cada algoritmo, podemos notar que los algoritmos Counting, Merge, Quick, y Heap se encuentran ocultos, ya que están por debajo de la graficación de RadixSort. Esto nos muestra la gran diferencia que hay entre los algoritmos de complejidad $O(n^2)$ y, los de complejidad $O(n\log(n))$ y $O(n+k)$. Siendo mejor los algoritmos: Radix, Counting, Merge, Quick, y Heap. Para el ordenamiento de elementos muy grandes. Aunque, hay que recordar que los algoritmos de Counting y Radix por más eficientes que sean en el tiempo para el ordenamiento de elementos, utilizan mucha memoria.

En la presente gráfica, se comparan los algoritmos de complejidad $O(nlog(n))$ Quick, Merge y Heap sort, junto con los algoritmos Couting y Radix sort, de complejidad $O(n+k)$ y $O(nk)$ respectivamente. En este caso, se les encomendó ordenar listas de hasta tamaño 20,000. La finalidad de esta gráfica es observar inicialmente cómo se comportan los algoritmos al ordenar primero hasta la cantidad mencionada de elementos.

Lo que se puede observar tras realizar el correspondiente análisis de los algoritmos, es que quick es mejor que sus semejantes en comlejidad pero es superado por couting y radix, pues estos dos últimos realizan menos operaciones, pero no hay que olvidar que requieren del uso de memoria adicional para ser utilizados. Figura 22.

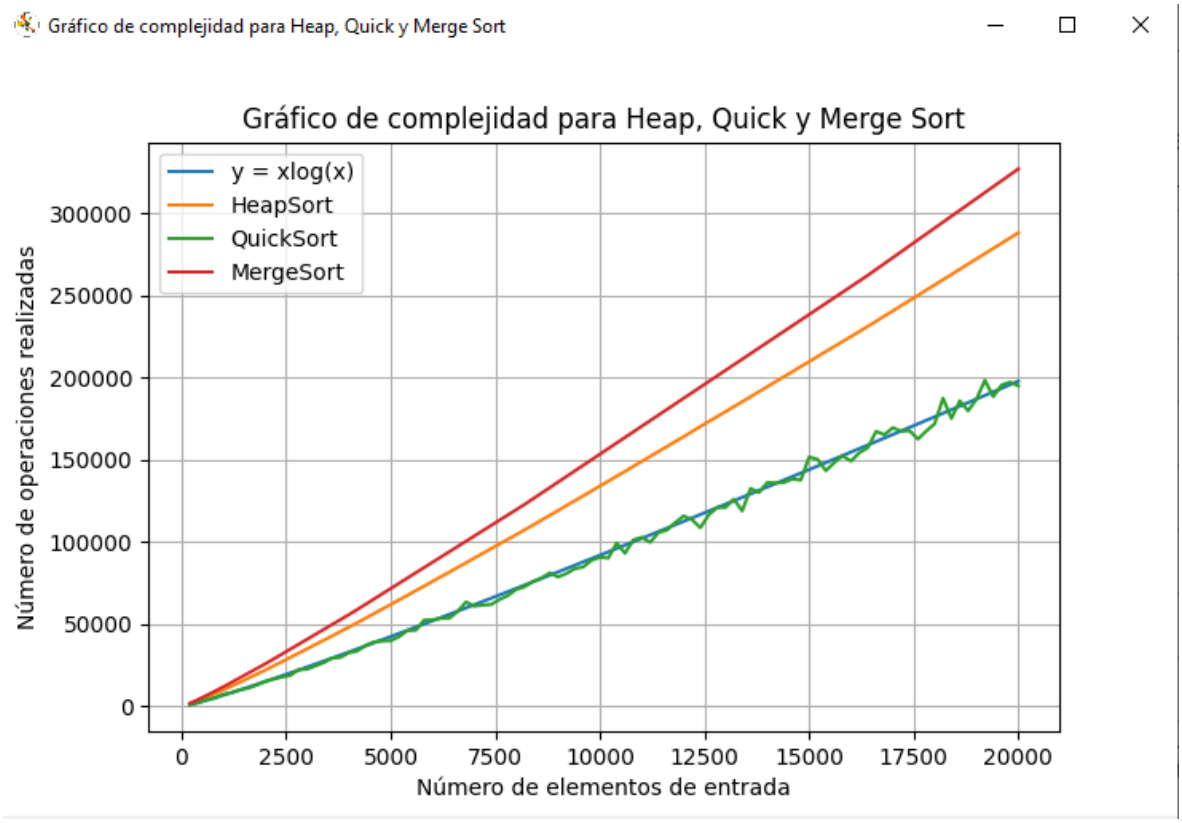

Figura 16: Gráfica de ordenamientos con complejidad $O(n\log(n))$ para una entrada de veinte mil elementos.

Finalmete, la siguiente gráfica (*Figura 23*), nos muestra la comparación de los algoritmos: Heap, Quick, Merge, Counting y Radix. Que corresponden a una complejidad en sus casos promedios de $O(n\log(n))$ y $O(n+k)$, respectivamente.

Podemos notar que el mejor de estos algoritmos es CountingSort, realizando un aproximado de 0.3×10^6 operaciones, después tenemos a RadixSort que realiza un aproximado de 0.5×10^6 operaciones. Estos dos algoritmos cumplen con una complejidad de $O(n+k)$.

Posteriomente, tenemos los algoritmos de complejidad $O(n\log(n))$, siendo el 3er mejor algortmo QuickSort, después le sigue HeapSort, y al final tenemos a MergeSort que sería el peor algoritmo de estas comparaciones realizando un aproximado de 1.80×10^6 operaciones.

De esta manera concluimos que los algoritmos de complejidad $O(n+k)$ son los mejores ordenando elementos en cuestión de tiempo, auqnue su desventaja es el uso excesivo de memoria.

Después de haber probado todos los algoritmos de ordenamiento del proyecto, así como haber analizado el comportamiento de sus graficas generadas, pudimos observar que los algoritmos *insertion*, *selection*, *bubble* y *pancake*, al ser algoritmos de complejidad $O(n^2)$, son muy lentos al realizar los ordenamientos. Por su parte, *couting* y *radix*, de complejidad $O(n+k)$ y $O(nk)$ que por el factor espacial su complejidad temporal

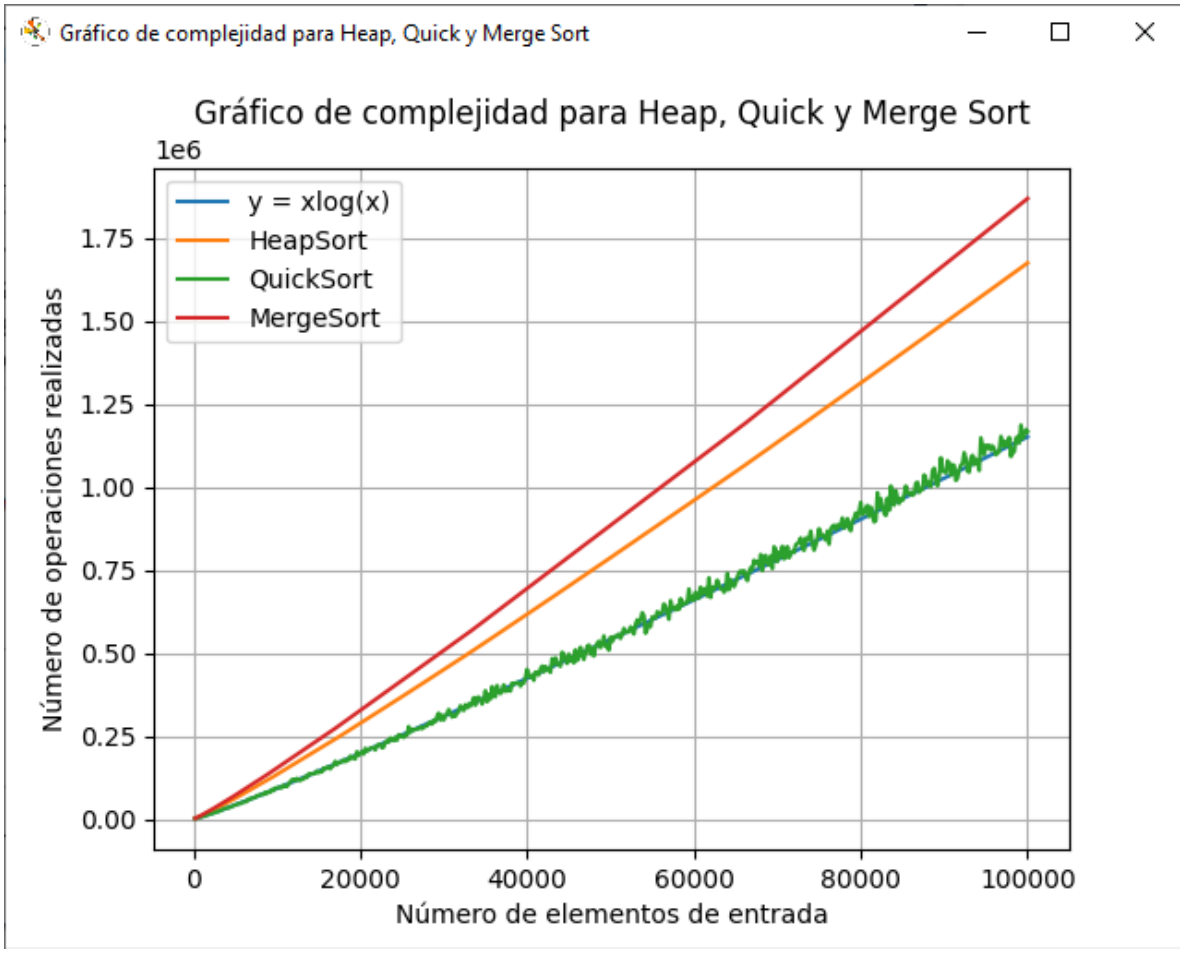

Figura 17: Gráfica de ordenamientos con complejidad $O(n\log(n))$ para una entrada de cien mil elementos.

termina siendo lineal, por ese factor del espacio terminan siendo menos eficientes y más costosos computacionalmente.

Finalmente los algoritmos de mayor eficiencia fuerón *heapsort*, *quicksort* y *mergesort*, por su complejidad $O(n\log(n))$, son los más veloces y entre estos, según la gráfica obtenida, el mejor fue *quicksort*, por lo que de todos los algoritmos analizados este fue el más eficiente en cuestión de tiempo, sin embargo, a pesar de que consideramos que pusimos a contar las operaciones de este algoritmo en el lugar correcto del código, por alguna razón el comportamiento de la gráfica tiene unas ciertas variaciones y sinceramente desconocemos porque se dio este comportamiento. Aunque podemos suponer que tal vez se dio debido a algún ligero error en el código de la implementación de este algoritmo o tal vez no logramos identificar de manera correcta en su totalidad las partes del código donde se debían contar las operaciones, pues el comportamiento de la gráfica es demasiado similar al esperado.

Tras lo anterior y todo el análisis realizado de los algoritmos comparados y estudiados en el proyecto, consideramos que con respecto al cuestionamiento de cuál es mejor, *quicksort* o *heapsort*, *quicksort* termina siendo más rapido que heap, pues en la grafica, este algoritmo tiene un mejor comportamiento y esto posiblemente se debe a que quick

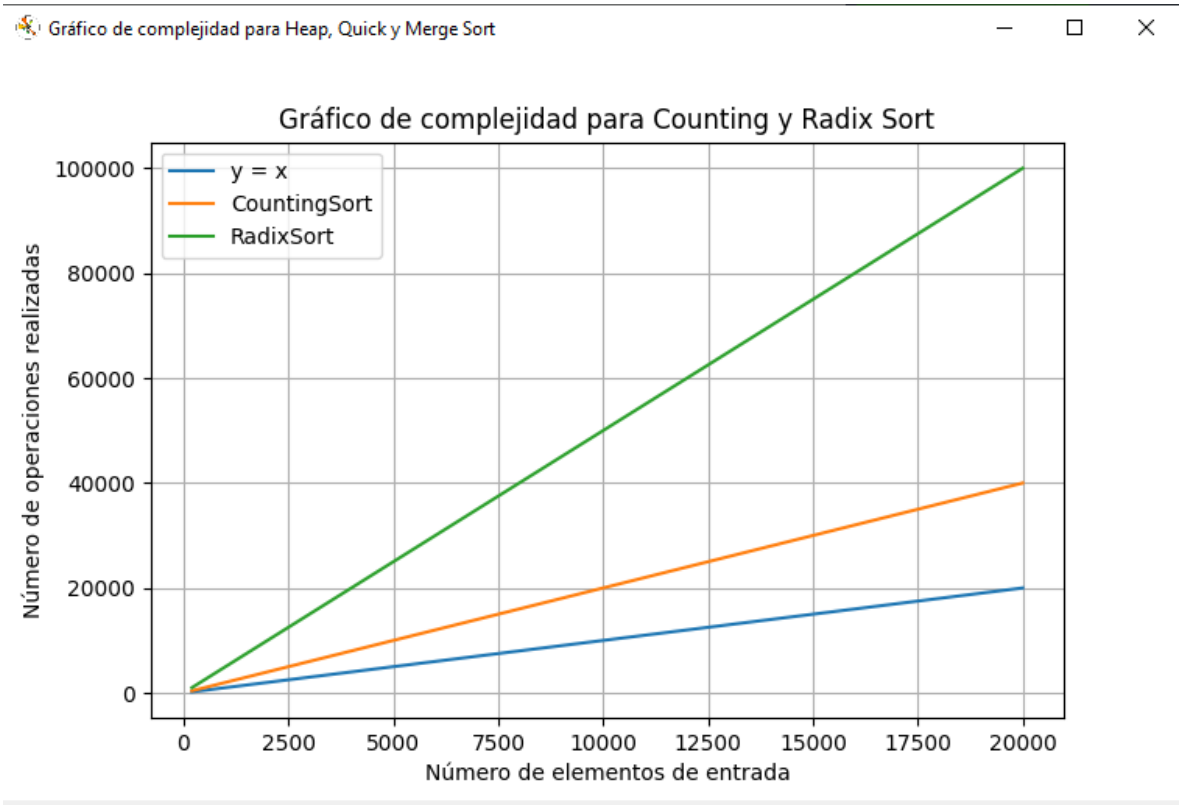

Figura 18: Gráfica de ordenamientos con complejidad $O(n + k)$ y $O(nk)$ para una entrada de veinte mil elementos.

realiza intercambios mas eficientes y *heap* termina haciendo un numero mayor de comparaciones e intercambios.

Recopilando los resultados obtenidos por las gráficas de las Figuras 20 y 23, y basándonos en los análisis de complejidad de ellas, podemos enumerar nuevamente todos los ordenamientos trabajados en el orden de eficacia, siendo el número uno el más eficiente de los nueve, y el número nueve el menos eficiente de todos.

1. CountingSort.
2. RadixSort.
3. QuickSort.
4. HeapSort.
5. MergeSort.
6. InsertionSort.
7. PancakeSort.
8. SelectionSort.
9. BubbleSort.

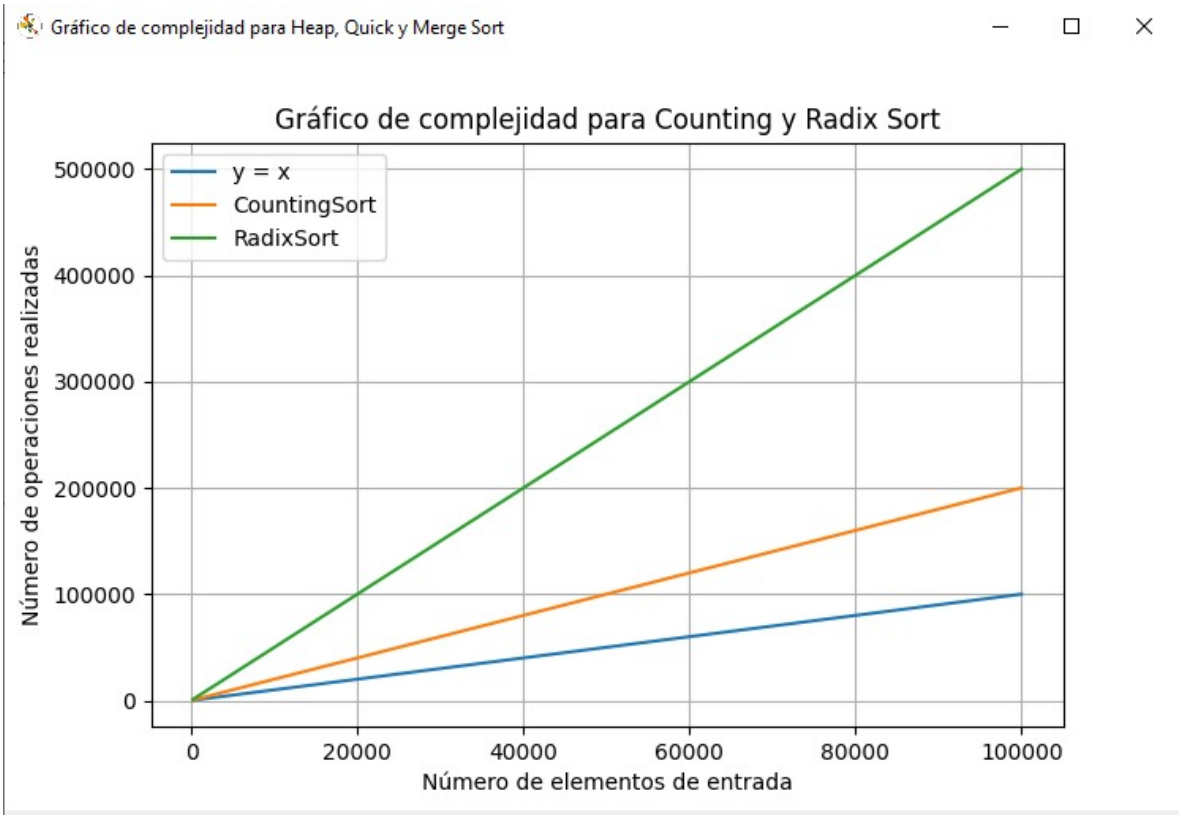

Figura 19: Gráfica de ordenamientos con complejidad $O(n + k)$ y $O(nk)$ para una entrada de cien mil elementos.

Sin embargo no debemos de perder de vista el hecho de que todo el análisis hecho para estos ordenamientos presenta únicamente el análisis **temporal**, y para el caso de los ordenamientos *RadixSort* y *CountingSort* se debe de considerar seriamente también el análisis **espacial** que no aborda esta investigación por lo que, enumerando nuevamente los ordenamientos bajo el mismo criterio, pero sin considerar a estos dos algormitmos por las razones dichas, tenemos que:

1. QuickSort.
2. HeapSort.
3. MergeSort.
4. InsertionSort.
5. PancakeSort.
6. SelectionSort.
7. BubbleSort.

De nuevo, si limitamos esta lista a únicamente los ordenamientos abordados a lo largo de la impartición del tema en la asignatura, la lista queda de la siguiente forma:

1. QuickSort.

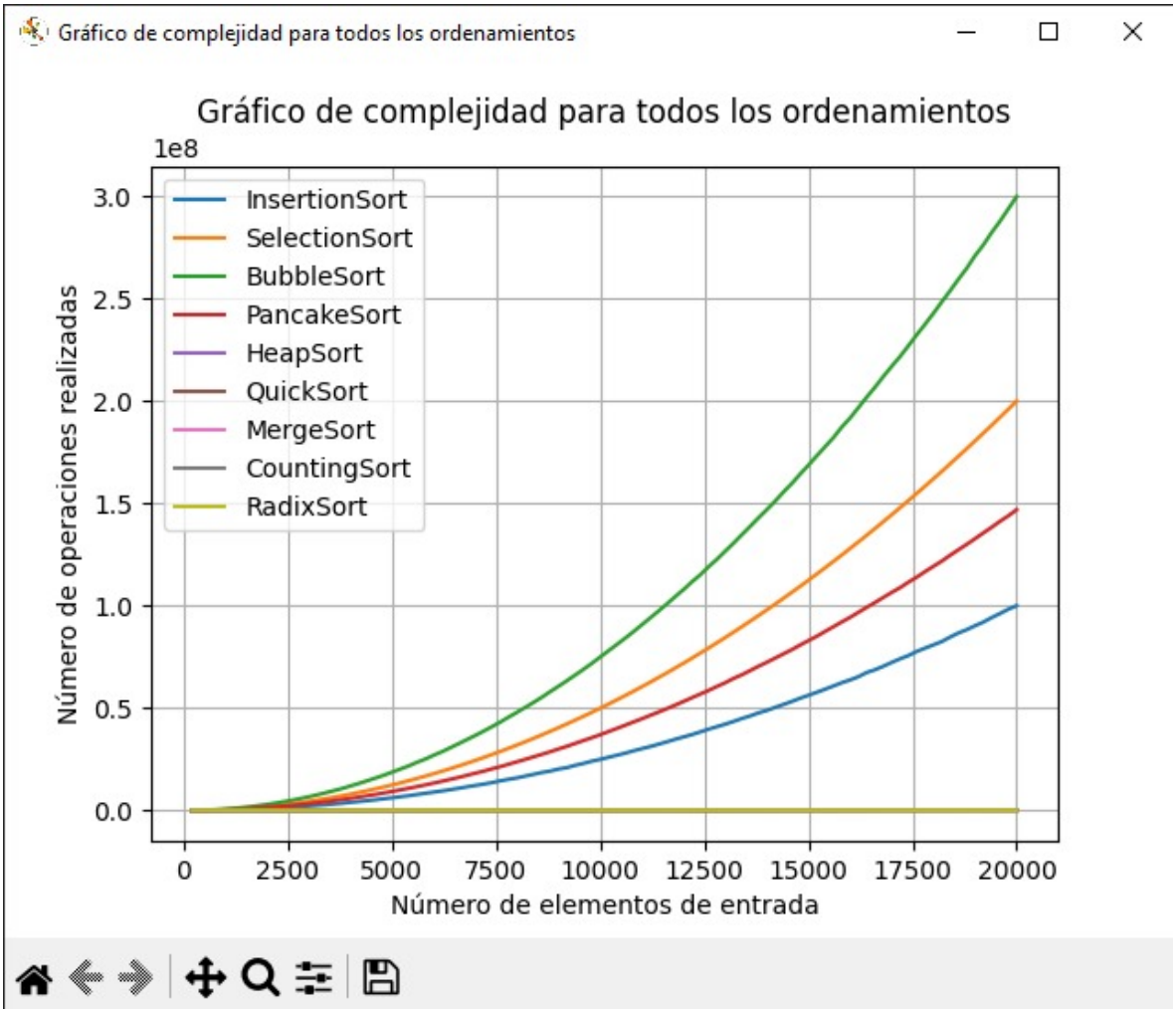

Figura 20: Gráfica de todos los ordenamientos para una entrada de veinte mil elementos.

2. HeapSort.
3. MergeSort.
4. InsertionSort.
5. SelectionSort.
6. BubbleSort.

Es así como se puede ver claramente que el ordenamiento de *QuickSort* es el más veloz de todos, seguido de *HeapSort*, y en tercer lugar el último de aquellos de complejidad $O(n\log(n))$ *MergeSort*. Siendo que, como se vió en la Figura 15, cuando se escala el tamaño de la entrada, los algoritmos de complejidad $O(n^2)$ se vuelven inviables de utilizar. Aún así, de entra éstos últimos el más eficiente resulta ser *InsertionSort*, seguido de *PancakeSort*, *SelectionSort* y, como el peor de todos los ordenamientos presentados, *BubbleSort*.

Un comentario extra a lo dicho sobre el análisis es que, la ejecución del programa con la entrada de diez mil tardó exactamente diecisiete horas y veinte minutos. A veces es difícil que podamos interpretar por nuestra cuenta los datos obtenidos experimentalmente, en especial cuando se trabajan con tantos datos. De las diecisiete horas y

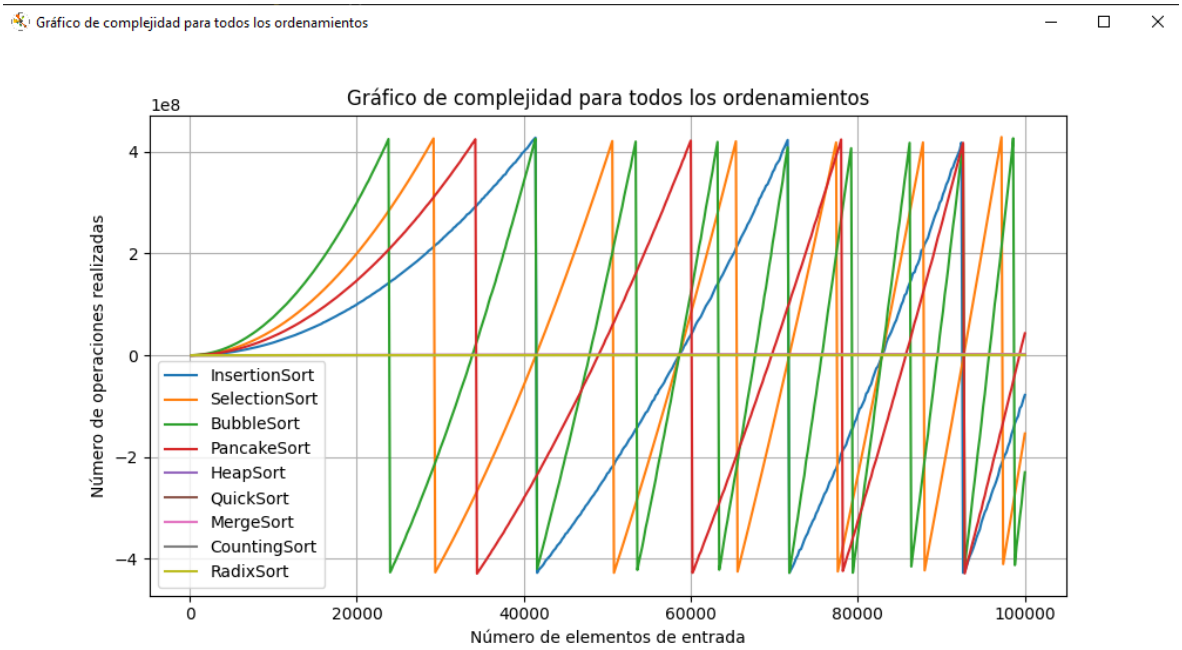

Figura 21: Gráfica de todos los ordenamientos para una entrada de cien mil elementos.

veinte minutos, los primeros cuatro algoritmos en realizarse (los de complejidad $O(n^2)$) tardaron diecisiete horas exactas en ejecutarse, los veinte minutos restantes fue lo que tardaron el resto de algoritmos en ejecutarse. Este comportamiento refuerza lo visto en al Figura 20.

5. Conclusiones

5.1. Cabrera Rojas Oscar

Los objetivos se cumplieron satisfactoriamente, al alumno implementó, recopiló e investigó de manera correcta distintos algoritmos de ordenamiento para el lenguaje de programación Java, e identificó correctamente los lugares del código en los que se realizan operaciones de inserción, comparación, intercambio e intercalación mediante el uso variables contadoras cuyo fin es contar el número de operaciones que necesitó, en cada ejecución, para cumplir su cometido; siendo capaz de desarrollar un programa principal cuyo funcionamiento ejecuta el número de veces deseado por el usuario cada uno de los ordenamientos cinco veces para obtener un valor promedio y automatizó el proceso de construcción de tablas y graficación de resultados para analizar satisfactoriamente lo sucedido en cada caso, permitiéndole manejar grandes volúmenes de información de manera fácil y eficaz.

Con todo ésto creo el entorno necesario para analizar y observar la complejidad temporal computacional de cada algoritmo de ordenamiento, evidenciando las principales diferencias entre cada uno de ellos y obteniendo una concordancia directa con la complejidad teórica en cada caso.

Finalmente concluyó que el algoritmo más eficiente es *Countingsort* a costa de una cantidad considerable de memoria adicional por colecciones adicionales a la original, y

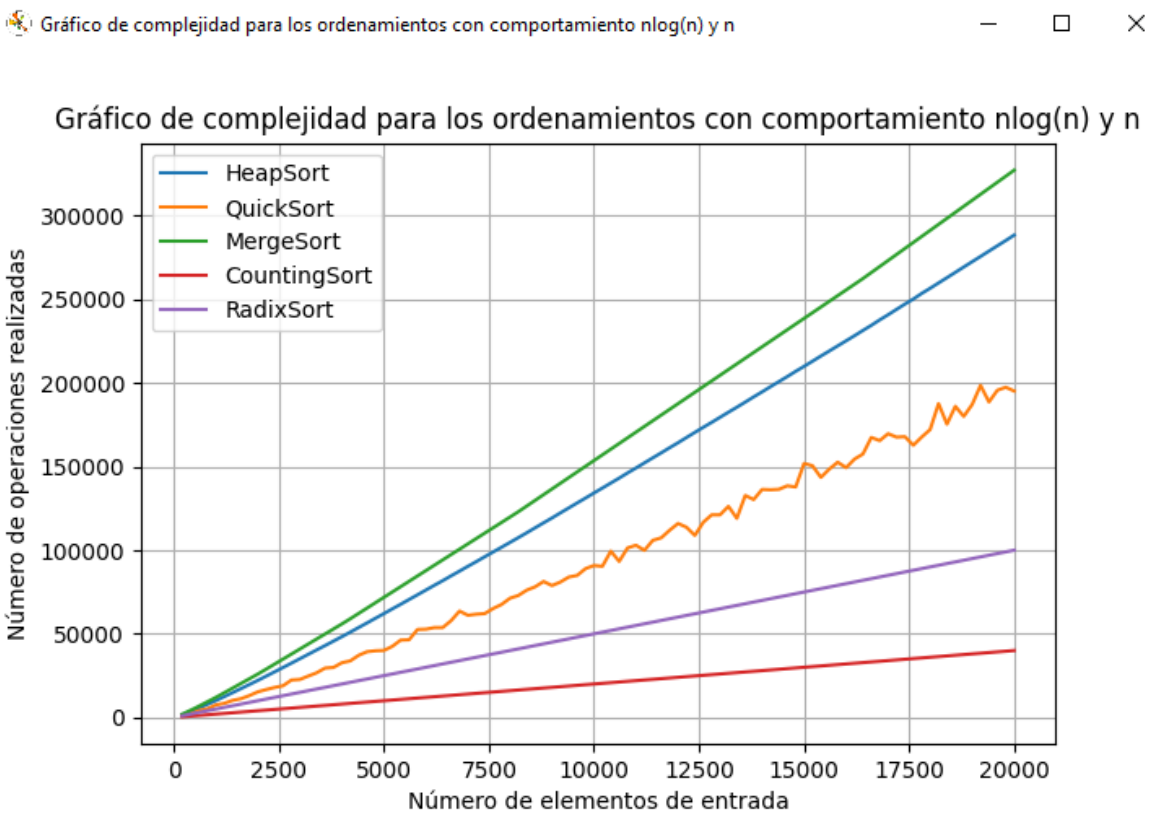

Figura 22: Gráfica de todos los ordenamientos que no tienen una complejidad $O(n^2)$ para una entrada de veinte mil elementos.

enfocado meramente en el análisis temporal, que fue el abordado en esta investigación, el algoritmo de ordenamiento más eficiente es *QuickSort*.

Colaboró y se organizó con los integrantes de su equipo para la delegación de tareas, creando un flujo de trabajo acorde, y en el que la delegación de tareas fue adecuada y afin a las aptitudes de cada integrante, logrando una participación activa en el tanscurso del desarrollo del proyecto; la resolución de problemas fue abordada de manera democrática y adecuada para todos los problemas. Las reuniones fueron programadas y organizadas enfocadas en sus objetivos para maximizar el tiempo invertido.

El alumno requirió de conocimientos intermedios sobre el lenguaje de programación Java como medio y herramienta para un fin más que como un lenguaje orientado a objetos; conocimientos en profundidad sobre los distintos algoritmos de ordenamiento y en la teoría de lo que son las operaciones de ordenamiento, las necesidades y limitaciones, así como del compromiso espacio-temporal. Adicionalmente también requirió de conocimientos superficiales en el uso general de archivos para su escritura / lectura, y del lenguaje de programación Python, enfocado a la creación de gráficos.

La creación de este proyecto y la conjunción de esfuerzos como actividades realizadas tal como se planteó fue el cierre perfecto para el tema de ordenamientos, obligando al alumno a entenderlos en su totalidad para ser capaz de implementarlos, y reforzando el entendimiento que tiene sobre ellos al ver la consecuecia gráfica de lo realizado. Sin

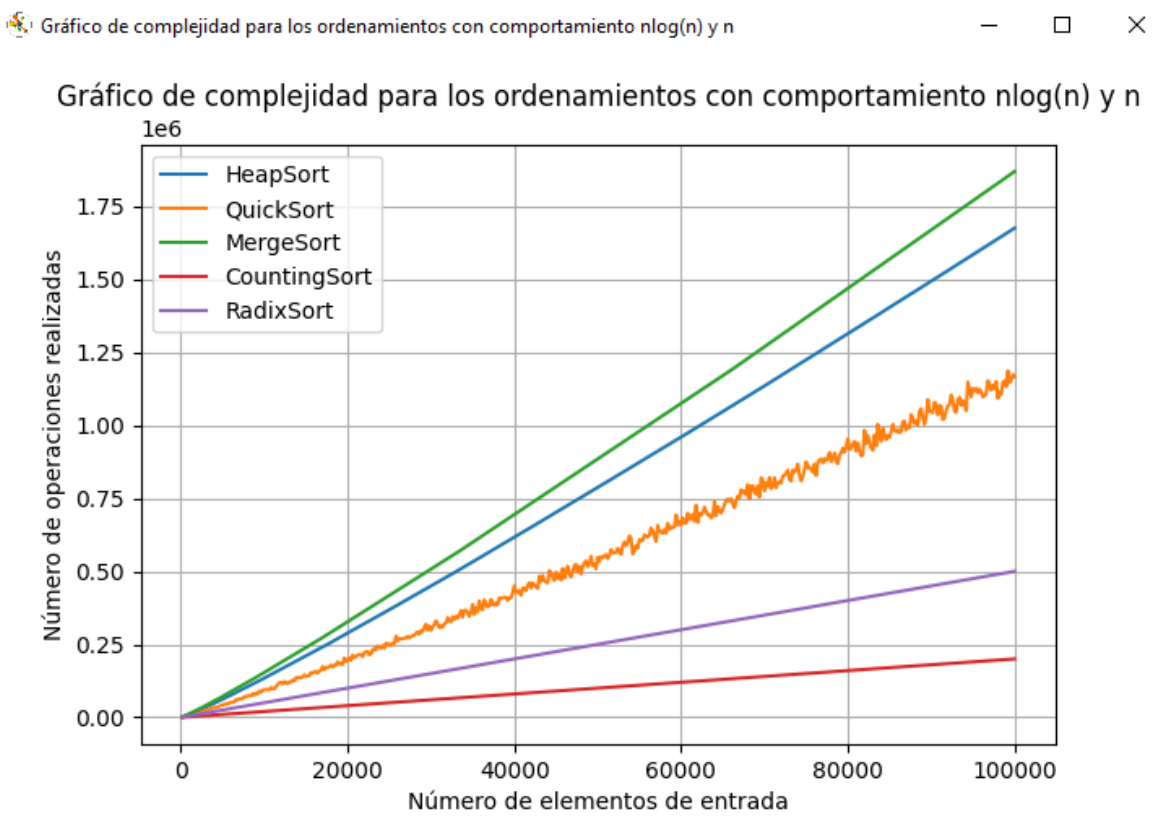

Figura 23: Gráfica de todos los ordenamientos que no tienen una complejidad $O(n^2)$ para una entrada de cien mil elementos.

duda alguna ayudó al estudiante a comprender mejor el tema.

5.2. Chavez Marquez Sergio Antonio

Al termino de este proyecto, se cumple con todos los requisitos obligatorios estipulados en el archivo *"Proyecto 1. Ordenamiento"*: Códigos, video explicativo, documentación y el trabajo escrito, que es el presente documento.

Respecto al objetivo, considero que se cumple con éxito, ya que este proyecto además de cumplir con la contabilización de datos en cada algoritmo, se muestran las tablas de los valores promedio comparados y el tiempo que le tomo a cada algoritmo ejecutar los ordenamientos (tomando en cuenta que el tiempo fue medido con el conteo de las iteraciones en cada algoritmo). Estos datos finales que nos proyecta el programa nos permitieron hacer la construcción de las gráficas para la posterior comparación de complejidad entre los algoritmos.

Cabe resaltar que el proyecto viene acompañado de unos archivos Python que son auxiliares en la construcción de gráficas, y que nos permitieron realizar un mayor número de pruebas para la comparación del funcionamiento del programa en distintos tamaños de la lista (arreglo). Es decir, que el usuario no tiene la necesidad de estar registrado las iteraciones que realiza cada algoritmo, automáticamente el programa crea archivos ".txt" que almacenan esta información, y que son utilizados al ejecutar el ar-

chivo “Graficar.py”.

En relación a los resultados obtenidos, considero que todos los algoritmos demostraron la complejidad de la que ya teniamos noción en las clases teoricas (para el caso promedio), aunque se presentó una única “anomalia” en la graficación de QuickSort, pero al final lo consideramos como aceptable. De esta manera, logramos ver que este algoritmo termina siendo el mejor de todos, “ganando” la comparación en eficiencia que se tenía con HeapSort.

Respecto al trabajo colaborativo con los integrantes del equipo. Considero que se realizó el mejor de los trabajos posibles, teniendo comunicación entre nosotros, aportación de ideas y soluciones cuando existían problemas, y en general todo el trabajo se realizó de la mejor manera.

5.3. Noyola Torres Pablo Sebastian

Al finalizar la realización del proyecto número 1 de la asignatura Estructura de datos y algoritmos 2, personalmente, considero que haberlo realizado fue de gran utilidad para estudiar, analizar y comprender de una mejor forma en general los diferentes conceptos contenidos en el tema 1 de ordenamientos, pues el hecho de tener que utilizarlos nuevamente y graficar su comportamiento para diferentes tamaños de listas, me ha servido para comprender su funcionamiento y también su eficiencia, al visualizar la complejidad que dichos algoritmos presentan.

Tras observar el comportamiento de todos los algoritmos, me pareció que Quicksort es el más eficiente, debido a que fue el que mostró un mejor comportamiento en la gráfica correspondiente. Aunque también me gustaría mencionar que a pesar de que este es el más eficiente y en consecuencia el posiblemente mejor de todos los vistos, en mi opinión quick no es tan sencillo de implementar y hay otros como bubble o pancake, que son mucho más fáciles de realizar en un programa, pero lamentablemente, no son eficientes y no nos sirven para listas de tamaños muy grandes, por lo que no es la idea más optima utilizar este tipo de algoritmos de complejidad cuadrática. Por su parte igual están otros como radix o couting, pero en el caso de estos algoritmos, se requiere utilizar memoria extra, lo que los hace más costoso hablando computacionalmente.

Por otra parte, en algunos algoritmos como fue en el caso del mismo quick o merge, nos costo trabajo encontrar las zonas del código donde necesitábamos contabilizar las operaciones de dichos algoritmos, pues al realizar sus correspondientes gráficas, sus comportamientos no correspondían con los esperados, como con quick que al final, aunque su grafica es muy parecida a como debería mostrarse, tiene un comportamiento peculiar con algunas modificaciones en su estructura.

En consecuencia, tras el desarrollo de la práctica, considero que se cumplió con el objetivo principal del proyecto, pues si se pudo visualizar el comportamiento de los algoritmos de ordenamientos para volúmenes muy grandes de información y por ende se comprendió cual fue el mejor, aunque como se mencionó anteriormente, ciertos algoritmos son más sencillos de implementar y tal vez sean de utilidad para ordenar listas pequeñas o medianas y para casos donde no sea importante la eficacia en dichos

ordenamientos.

Por otro lado, el trabajo en equipo fue fundamental para llevar a cabo este proyecto, ya que gracias a eso creo que el proyecto pudo avanzar y realizarse de una buena manera, pues con una adecuada organización y comunicación, pudimos trabajar en la realización de este proyecto. Por supuesto que por cuestiones de tiempo no fue posible mejorar algunos aspectos y quizás algunas cosas pudimos haberlas hecho de otra forma o mejor, pero sinceramente creo que hicimos lo que estuvo en nuestras manos y siempre el factor del tiempo juega un papel importante.

Finalmente, considero que fue un buen proyecto donde se comprendieron mejor muchos conceptos sobre los ordenamientos y considero que aun tengo que mejorar en varios aspectos de mis habilidades de programación y también en mi manera de organizarme para poder realizar mejor las cosas y ser más determinante en mis contribuciones con el fin de que este tipo de proyectos salgan de la mejor forma posible. La verdad es que fue un proyecto bastante entretenido y algo pesado, pero creo que este tipo de actividades son útiles para nuestro aprendizaje.

Referencias

[1] GeeksForGeeks. Pancake sorting, Abril 2023. Accessed on March 22, 2024.

[2] GeoGebra. Calculadora gráfica - GeoGebra.

[3] Jindal, H. Ordenamiento de panqueques, Octubre 2023. Accessed on March 21, 2024.

[4] Storti, M. Pancake sort algorithm, visualization with VTK, Noviembre 2009. Accessed on March 22, 2024.