Universidad de La Serena
Informe Algoritmos de Ordenamientos

Diseño y Análisis de Algoritmos
Profesor: Erick Luciano Castillo Bastias
Fecha : Miércoles 29 de mayo
Equipo 2

Integrantes :
Fernando López
Rodrigo Mamani
Bryan Townsend

Índice

1. Introducción

En el siguiente informe se detallara la actividad realizada durante el desarrollo de este. Como sabemos los métodos de ordenamiento cumplen un factor relevante en la computación, ya que el propósito principal por lo general es ordenar una secuencia para luego facilitar las búsquedas de los miembros del conjunto ordenado. Es tanto así que existen diferentes tipos de algoritmos de ordenamiento en los cuales una buena practica sería escoger cuidadosamente que algortimo nos será mas util al momento de tener una secuencia de datos por ejemplo. Un concepto importante a mencionar es la estabilidad del algortimo que escojamos, ya que será fundamental en el proceso de ordenamiento, al igual que la memoria que necesitará cada uno de estos algoritmos. A continuación se muestra distintos tipos de Algoritmos de Ordenamientos con sus cualidades y caracaterísticas, en el cual se podrá diferenciar las similitudes y diferencias entre uno y otro.

Figura 1: Tipos de Algortimos de Ordenamiento(Estables e Inestables)

Estables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento de burbuja	Bubblesort	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento de burbuja bidireccional	Cocktail sort	$O(n^2)$	$O(1)$	Intercambio
Ordenamiento por inserción	Insertion sort	$O(n^2)$("(en el peor de los casos)")	$O(1)$	Inserción
Ordenamiento por casilleros	Bucket sort	$O(n)$	$O(n)$	No comparativo
Ordenamiento por cuentas	Counting sort	$O(n+k)$	$O(n+k)$	No comparativo
Ordenamiento por mezcla	Merge sort	$O(n \log n)$	$O(n)$	Mezcla
Ordenamiento con árbol binario	Binary tree sort	$O(n \log n)$	$O(n)$	Inserción
	Pigeonhole sort	$O(n+k)$	$O(k)$	
Ordenamiento Radix	Radix sort	$O(nk)$	$O(n)$	No comparativo
	Distribution sort	$O(n^3)$ versión recursiva	$O(n^2)$	
	Gnome sort	$O(n^2)$	$O(1)$	
Inestables				
Nombre traducido	Nombre original	Complejidad	Memoria	Método
Ordenamiento Shell	Shell sort	$O(n^{1.25})$	$O(1)$	Inserción
	Comb sort	$O(n \log n)$	$O(1)$	Intercambio
Ordenamiento por selección	Selection sort	$O(n^2)$	$O(1)$	Selección
Ordenamiento por montículos	Heapsort	$O(n \log n)$	$O(1)$	Selección
	Smoothsort	$O(n \log n)$	$O(1)$	Selección
Ordenamiento rápido	Quicksort	Promedio: $O(n \log n)$, peor caso: $O(n^2)$	$O(\log n)$	Partición
	Several Unique Sort	Promedio: $O(n \mathrm{u})$, peor caso: $O(n^2)$; u=n; u = número único de registros		

es por eso que a partir de la Figura 1,surgen diferentes preguntas considerando a cada uno de los tipos de ordenamientos,una de ellas es:
¿Cuándo conviene usar uno o otro método de ordenamiento?.

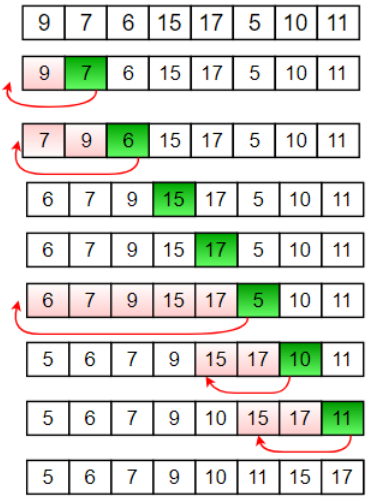
y la respuesta más rapida es considerar cuando se requiere hacer una cantidad considerable de búsquedas y es importante el factor tiempo.

Una vez contextualizado el tema que estaremos constantemente abordando, y aterrrizando con la actividad planteada propongo dar una breve definicón muy por encima de los algoritmos que estaremos estudiando y poniendo a prueba durante el desarrollo de este informe, para que posteriormente en la siguiente sección(Desarrollo de Software) se detalle con más profundidad con ayuda de los desarrolladores.

1.1. Orden por Inserción (InsertionSort)

Dado un array A de n-elementos, la estrategia consiste en realizar n-recorridos por el array. En el recorrido i-ésimo, el elemento almacenado en A[i-1] es trasladado a su lugar apropiado en A[0..i - 1]. De este modo, después del recorrido i-ésimo, los elementos en A[0...i -1] están en orden, lo que significa que después de n-recorridos, el array A completo estará ordenado.

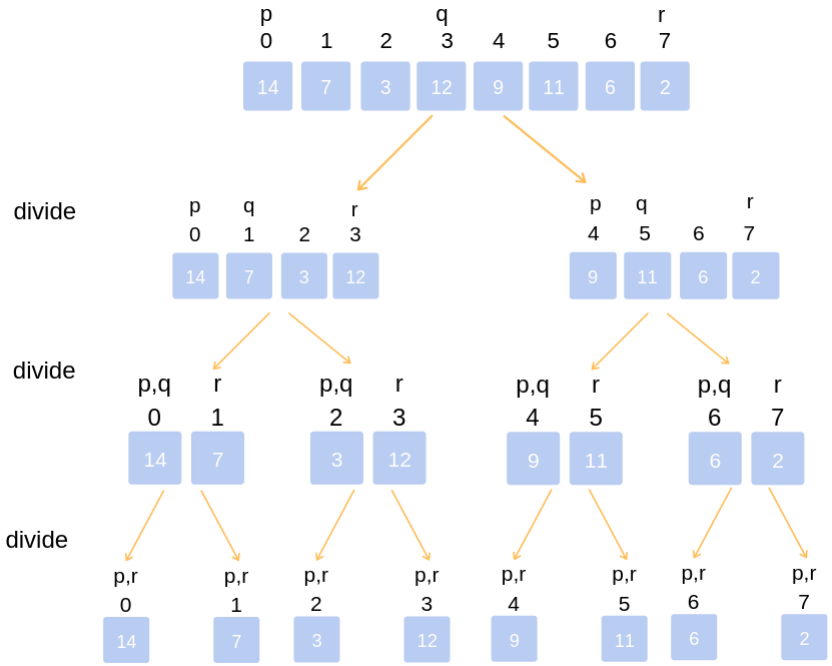
Figura 2: Ejemplo de Ordenamiento por InsertionSort.



1.2. Ordenación por Mezcla (MergeSort)

Dado un array A de n-elementos, la estrategia esta basada en la metodología Dividir para Conquistar, en donde el array es particionado en 2 subarrays, y así sucesivamente hasta Finalmente lograr los subarrays ya ordenados para luego combinarlos para producir el array ordenado final.

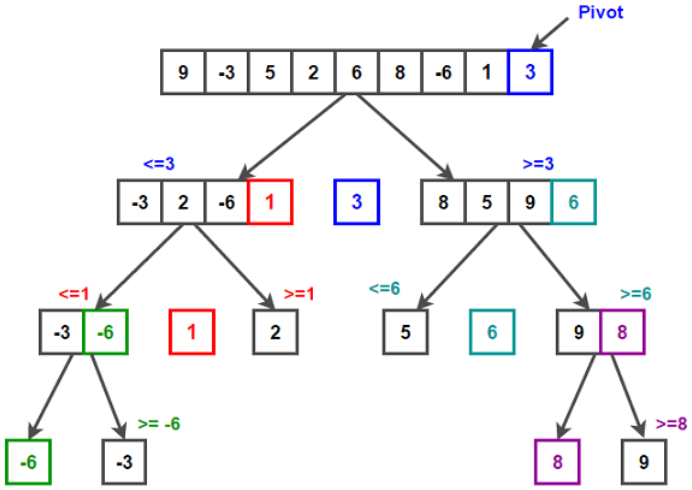
Figura 3: Ejemplo de Ordenamiento por MergeSort.



1.3. Orden Rápido (QuickSort)

Dado un array A de n-elementos, la estrategia también esta basada en la metodología Dividir para Conquistar. Sin embargo, a diferencia del algoritmo de ordenación por mezcla, la parte no recursiva del algoritmo de ordenación rápida con lleva la construcción de subejemplares por medio de una técnica de partición, Se selecciona un elemento especifico del array A[p] llamado pivote, seguidamente se recorre el array haciendo una serie de operaciones considerando en todo momento este elemento que escogeremos, hasta lograr así el ordenamiento.

Figura 4: Ejemplo de Ordenamiento por QuickSort.



2. Desarollo de Software
Algoritmos desarrollados y métodos utilizados:
Bryan Townsend 3-6, Rodrigo Mamani 4-5.

En la siguiente sección se describirá el progreso que realizamos en conjunto con los desarrolladores durante el progreso de la implementación de los metodos de ordenamiento, para luego poder análizar y diferenciar cada uno de los algoritmos como así tambien sus métodos utilizados.

3. InsertionSort

Insertion Sort es un algoritmo eficiente para ordenar un pequeño número de elementos. Esto se realiza **in place** lo que implica que el costo en memoria es $O(1)$.

El algoritmo realiza el ordenamiento comparando cada elemento i-ésimo del arreglo con los elementos del arreglo en las posiciones entre A[0..i-1](arreglo ordenado) ubicándolo en el lugar correspondiente tal que A[0..i] quede ordenado. Esto se puede realizar utilizando un método **for** que recorra el arreglo desde el elemento A[1] hasta A[n-1] y un método **while** que recorra el arreglo ya ordenado buscando la posición correcta del elemento A[i]. Como esto se realiza en dos ciclos de costo cada uno $O(n)$, tendrémos un costo de $O(n^2)$ en el caso promedio y en el peor caso. En caso de que el arreglo ya esté ordenado el costo será $O(n)$, porque el algoritmo no recorrerá el método while (A[i] siempre será mayor en este caso que A[i-1]).

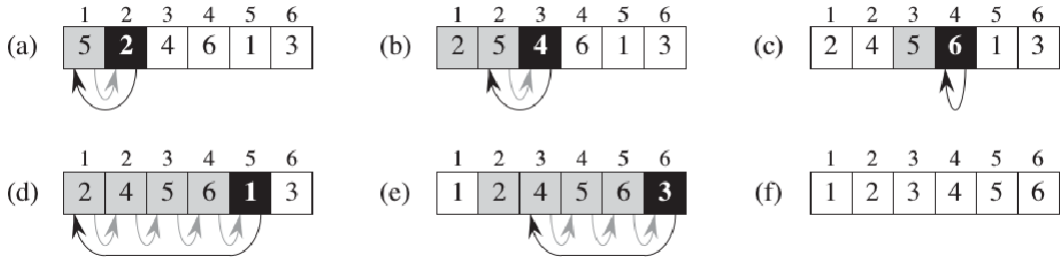


Figura 5: Pasos sucesivos en como se realiza insertionSort. Color negro: Elemento que se va a comparar. Color gris: Elementos ya ordenado, Color blanco: Los elementos que faltan por ordenar

3.1. Descripción

El algortimo trabaja de la siguiente forma:
Para realizar este algoritmo, se resuelve dos casos:

- Ordenar un arreglo de enteros.
- Ordenar una lista de enteros.

El método recibe un arreglo (o lista) siendo parámetros de referencia. Se debe recorrer el arreglo desde el segundo elemento del arreglo hasta el último, ya que el primer elemento ya está ordenado.

- for j = 2 hasta el largo del arreglo

Luego se debe almacenar en una variable auxiliar el elemento que se debe insertar al sub-arreglo ya ordenado, este es A[i]

- aux = A[i];

Como debemos insertar A[i] en el subarreglo A[1..i-1], debemos recorrer este arreglo a lo más i-1 veces, lo que corresponde a j-1 veces.

- x = j - 1 (x será el índice del elemento anterior a A[i];

Este recorrido se realiza en un ciclo **while** hasta que haya recorrido todo el subarreglo o haya encontrado un valor A[x] <= aux

- while (i > 0 y A[x] > aux)

Durante el ciclo debe ir intercambiando los valores de A[x] con A[x-1] sucesivamente hasta salir del ciclo, esto mediante intercambio con variables auxiliares (swap) costo $O(1)$.

- A[x+1] = A[x] y al salir del ciclo tenemos que A[x] = aux.

3.2. Implementación

3.2.1. Sort

La figura 2 muestra el código escrito en Java. Los métodos utilizados para insertionSort. Con la diferencia en que uno se implementa para una lista y otro para un arreglo.

```
// Primer método utilizando un arreglo de enteros
public static void insertionSortArray(int A[]) {

    for(int j=1; j < A.length; j++) {
        int clave = A[j];
        int i = j - 1;
        // Inserto el elemento A[i] en el arreglo ordenado A[0..i-1]
        while(i >= 0 && A[i] > clave) {
            A[i+1] = A[i];
            i--;
        }
        A[i+1] = clave ;
    }

}
// Segundo método utilizando una lista de enteros
public static void insertionSortList(ArrayList <Integer> list) {
    for(int j=1; j < list.size(); j++) {
        int clave = list.get(j);
        int i = j - 1;
        // Inserto el elemento A[i] en la lista ordenada A[0..i-1]
        while(i >= 0 && list.get(i) > clave) {
            list.set(i+1, list.get(i));
            i--;
        }
        list.set(i+1, clave);
    }

}
```

Figura 6: Metodo Sort para una Lista y para un Arreglo

4. QuickSort

El ordenamiento rápido(quicksort en ingles) es un algoritmo basado en comparaciones que utiliza la técnica divide y vencerás que permite ordenar N con costo $O(nlog(n))$. Es una de las técnicas más rápidas para ordenar elementos (basado en comparaciones). Fue desarollada por el científico brítanico en computacion Charles Antony Richard en 1960.

4.1. Descripción

El algortimo trabaja de la siguiente forma:

1. Elegir un elemento del arreglo a ordenar, que llamaremos pivote.
2. Resituar los demas elementos del arreglo a acada lado del pivote, de manera que al lado izquiero solo queden los menores a el pivote y al lado derecho solo queden lo mayores a el pivote.
3. El arreglo queda "dividido.[en] dos los menores al pivote y los mayores a el.
4. Repetir de forma recursiva para cada división hasta que el tamaño de las diviciones sea de 1 elemento.
5. luego de realixar todas las diviciones el arreglo queda ordenado.

4.2. Implementación

4.2.1. Pivote

El pivote sera el primer elemento del arreglo a ordenar

4.2.2. Sort

El método *Sort*() sera el encargado de dividir el arreglo de forma recursiva hasta que solo contengan un elemento.

```
private void Sort(int[] ordenar, int bot, int top) {
    int j = 0;
    if (bot < top) {
        j = particion(ordenar, bot, top);
        Sort(ordenar, bot, j - 1);
        Sort(ordenar, j + 1, top);
    }
}
```

Figura 7: Metodo Sort

4.2.3. Particion

EL método *paricion*() es el encargado de mover los elementos del arreglo a los lados del pivote y retorna un entero que sera la pocición final del pivote donde se dividira el arreglo.

```
private int particion(int[] ordenar, int bot, int top) {
    int piv = ordenar[bot];
    int i = bot;
    int j = top + 1;
    while (true) {
        // encuentra mayor
        while (ordenar[++i] <= piv) {
            if (i == top) {
                break;
            }
        }
        // encuantra menor
        while (ordenar[--j] >= piv) {
            if (j == bot) {
                break;
            }
        }
        if (i >= j) {
            break;
        }
        Swap(ordenar, i, j);
    }
    Swap(ordenar, bot, j);

    return j;
}
```

Figura 8: Metodo particion

5. MergeSort

El ordenamiento por mescla (mergesort en ingles) es un algortimo basado en comparaciones que al igual que en el Quicksort utiliza la tecnica de divir y vencerás, este algortimo permite ordenar N elementos de un arreglo con costo $O(nLog(n))$ ademas de ser estable.

5.1. Descripción

El algortimo trabaja de la siguiente forma:

1. Si el arreglo es de longitud 0 o 1 entonces ya se encuentra oredenado paso 3. En caulquier otro caso paso 2.
2. Divivir el arreglo en dos mitades de aproximadamente la mitad del tamaño, con cada arreglo repetir paso 1.
3. Reordenar cada sub-array de forma recursiva aplicandoe el ordenamiento por mescla.
4. Mesclar los dos sub-array resultantes para generar un rreglo ordenado.

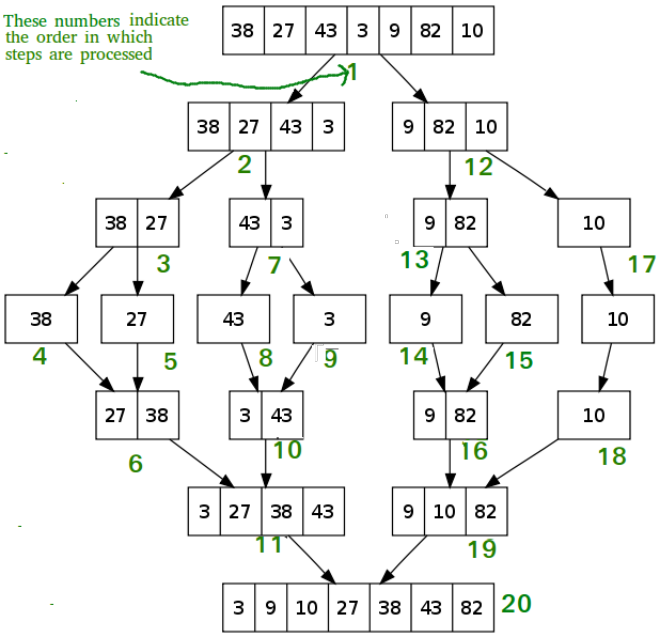


Figura 9: MergeSort

5.2. Implementación

5.2.1. Sort

Primero debemos crear un arreglo auxiliar que nos ayudara a ordenar el arreglo original.

```
public void Sort(int[] ordenar) {
    int[] aux = new int[ordenar.length];
    Sort(ordenar, aux, 0, ordenar.length);
}
```

Figura 10: Sort

5.2.2. Merge

Método que ordenara los elementos de los sub arreglos mediante comparaciones en el arreglo auxiliar, luego de tener el arreglo auxiliar ordenado, se pasan los elementos a el arreglo original.

```
private void Merge(int[] ordenar, int[] aux, int bot, int mid, int top) {
    int i = bot;
    int j = mid;

    for (int k = bot; k < top; k++) {
        if (i == mid) {
            aux[k] = ordenar[j++];
        } else if (j == top) {
            aux[k] = ordenar[i++];
        } else if (ordenar[j] < ordenar[i]) {
            aux[k] = ordenar[j++];
        } else {
            aux[k] = ordenar[i++];
        }
    }

    for (int k = bot; k < top; k++) {
        ordenar[k] = aux[k];
    }
}
```

Figura 11: Merge

6. Array.sort y Collections.sort

6.1. Descripción

El ordenamiento en una estructura de dato del tipo arreglo en Java es realizado mediante el algoritmo doble pivote QuickSort. El costo de este algoritmo es $O(nlogn)$ en el caso promedio, en el peor caso (los elementos ordenados) este tiene un costo de $O(n^2)$, el algoritmo a pesar de tener un costo asintótico óptimo para los algoritmos basados en comparaciones tiene un tiempo menor que otro algoritmos.
El problema es que QuickSort no es un algoritmo estable, es por ello que este es el método que utilza Java solo cuando trabaja con datos primitivos (como es el caso de esta tarea).

La idea de usar doble pivote es elegir uno a la izquierda y otro a la derecha, realizando 3 particiones. Elementos menores del pivote izquierda se colocan en la izquierda, elementos mayores a pivote izquierda y menores a pivote derecha en la partición en el medio y elementos mayores a pivote derecha a la derecha.

Para el ordenamiento de objetos en el arreglo, necesitamos un algoritmo estable, para ello Java utiliza timSort (nombre debido a quién lo creó). Este algoritmo consiste en utilizar insertionSort para un subarreglo del arreglo original, ya que insertionSort es eficiente para pocos datos y luego utilizar la función merge de MergeSort que permita unir los arreglos ordenandolos con bloques potencia de 2, ya que es donde merge tiene menor costo. Este algoritmo es de costo $O(nlogn)$ Para cualquier caso.

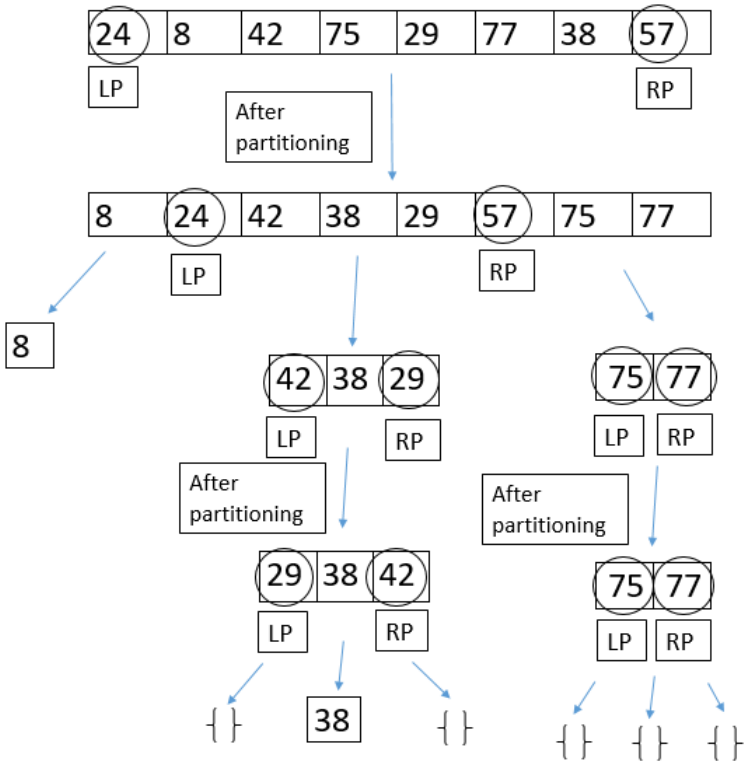


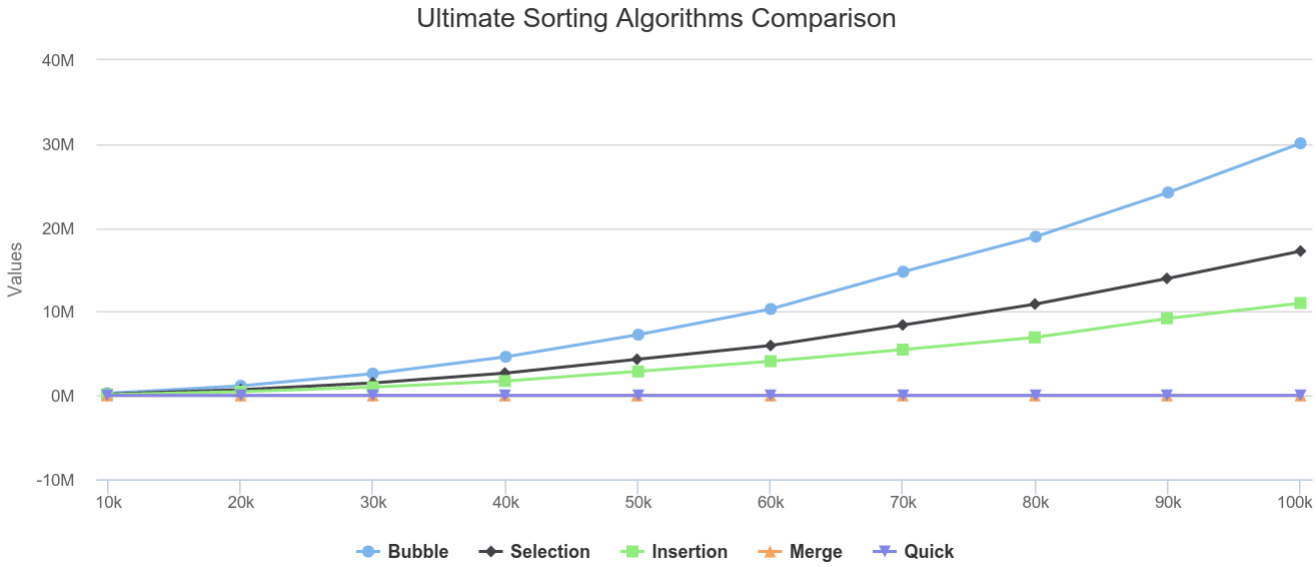
Figura 12: Ejemplo doble pivote QuickSort

En Java es posible ordenar un objeto del tipo Collection (ArrayList, Linklist) utilizando el método Collections.sort que consiste en utilizar el mismo método mencionado para array.sort, pero dedicado principalmente a listas.

7. Diferencias entre algoritmos y métodos

Para encontrar diferencias entre los algoritmos estudiados es fundamental considerar los tiempos de ordenamiento para una secuencia dada, es decir, su complejidad, por ende es inverosimil escoger a priori y rapidamente que algoritmo es mejor que otro, por lo que debemos fijarnos cual de estos nos podra ser más util dependiendo de la problemática y la secuencia de la que queremos ordenar. Es por esto que la diferencia más considerable respecto a estos algortimos es el tiempo de ejecución.
Por lo que como sabemos en el peor de los casos QuickSort se tiene $O(n^2)$ al igual que el InsertionSort, y donde el ordenamiento MergeSort será de $0(nlogn)$

Figura 13: Gráfica y Análisis Algoritmos de Ordenamientos.



Como se puede observar en esta gráfica comparativa se muestran cinco algoritmos de ordenamiento, en los cuales nos enfocaremos en los tres que estudiamos durante el desarollo de esta actividad (InsertionSort, MergeSort,QuickSort). Podemos conocer a ciencia cierta quien fue el ganador, y también se puede notar con total seguridad quien fue el claro perdedor, el cual fue el algoritmo Insertion con una complejidad algoritmica de $O(n^2)$, pero sus implementaciones son un poco más eficientes dado que se hacen menos comparaciones, pero no dejan de ser algoritmos poco competentes.

8. Software Puesto a Prueba

A Continuación se visualizará y se pondrá a prueba el Software creado para esta actividad, en el cuál le ingresaremos una cierta cantidad de números y así analizaremos sus tiempos de ejecución respectivos, para así poder generar sus gráficas y analizar los datos obtenidos de mejor forma. Cabe mencionar que la implementación se realizó en Java 8.0.

8.1. Orden de magnitud 1

Orden de magnitud 1: 10, 20, 30, 40, 50, 60, 70, 80, 90.

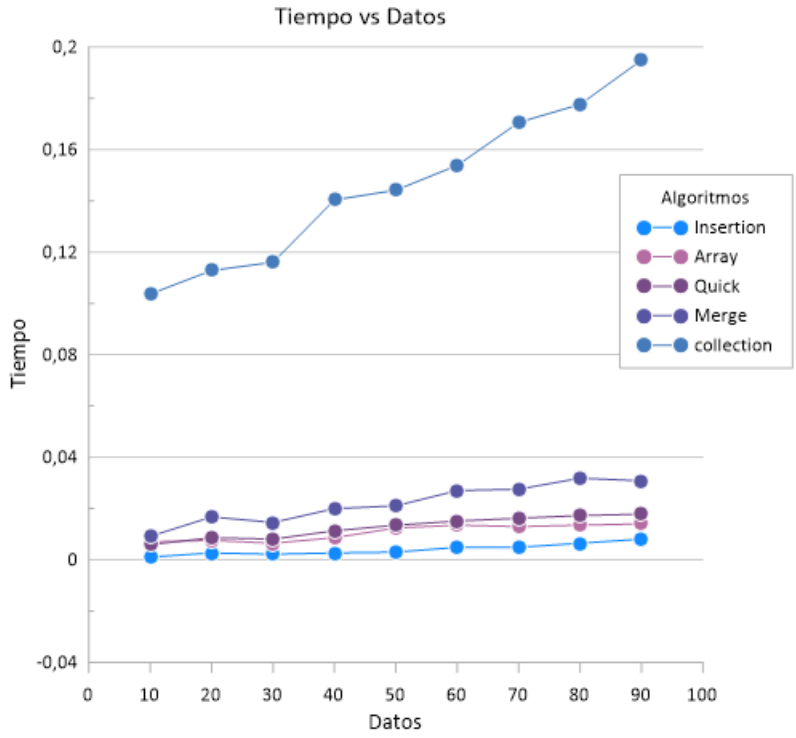


Figura 14: Gráfica Tiempo Vs Datos

8.2. Orden de magnitud 2

Orden de magnitud 2: 100, 200, 300, 400, 500, 600, 700, 800, 900.

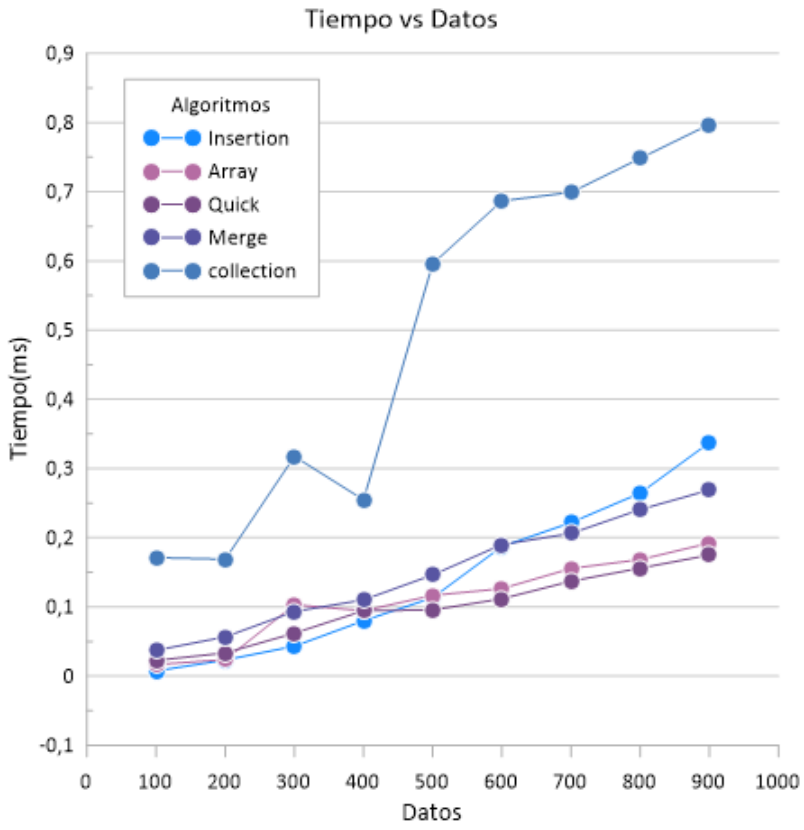


Figura 15: Gráfica Tiempo Vs Datos

8.3. Orden de magnitud 3

Orden de magnitud 3: 1.000, 2.000, 3.000, 4.000, 5.000, 6.000, 7.000, 8.000, 9.000.

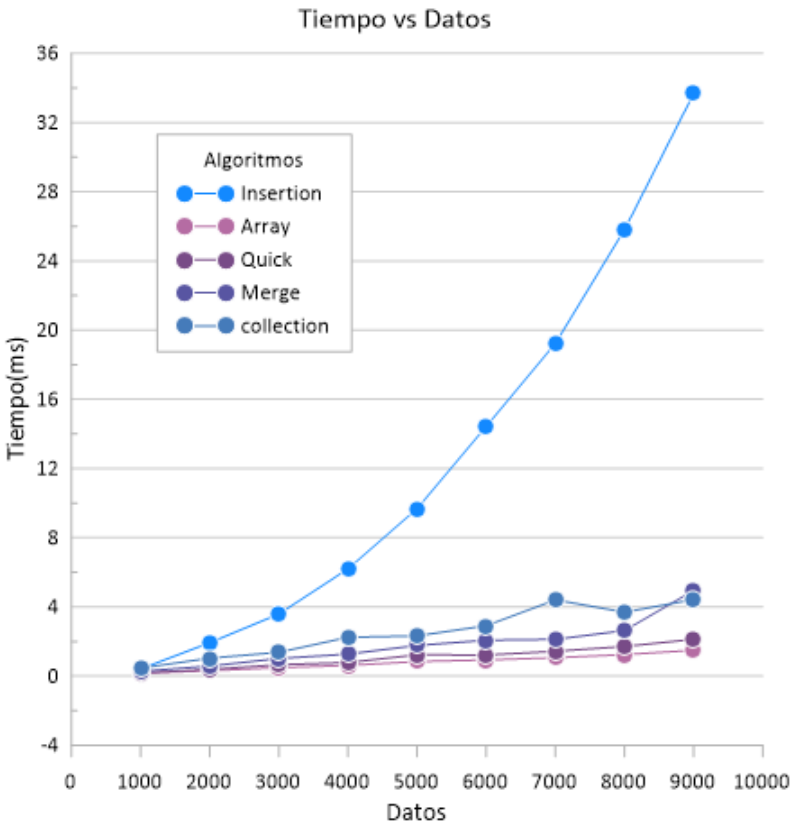


Figura 16: Gráfica Tiempo Vs Datos

8.4. Orden de magnitud 4

Orden de magnitud 4: 10.000, 20.000, 30.000, 40.000, 50.000, 60.000, 70.000, 80.000, 90.000.

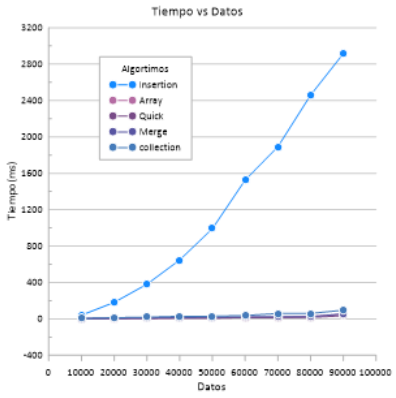


Figura 17: Gráfica Tiempo Vs Datos

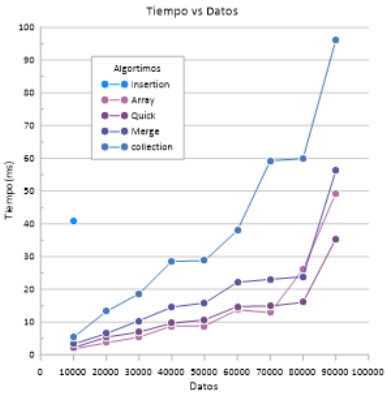


Figura 18: Gráfica Tiempo Vs Datos

8.5. Orden de magnitud 5

Orden de magnitud 5: 100.000, 200.000, 300.000, 400.000, 500.000, 600.000, 700.000, 800.000, 900.000.

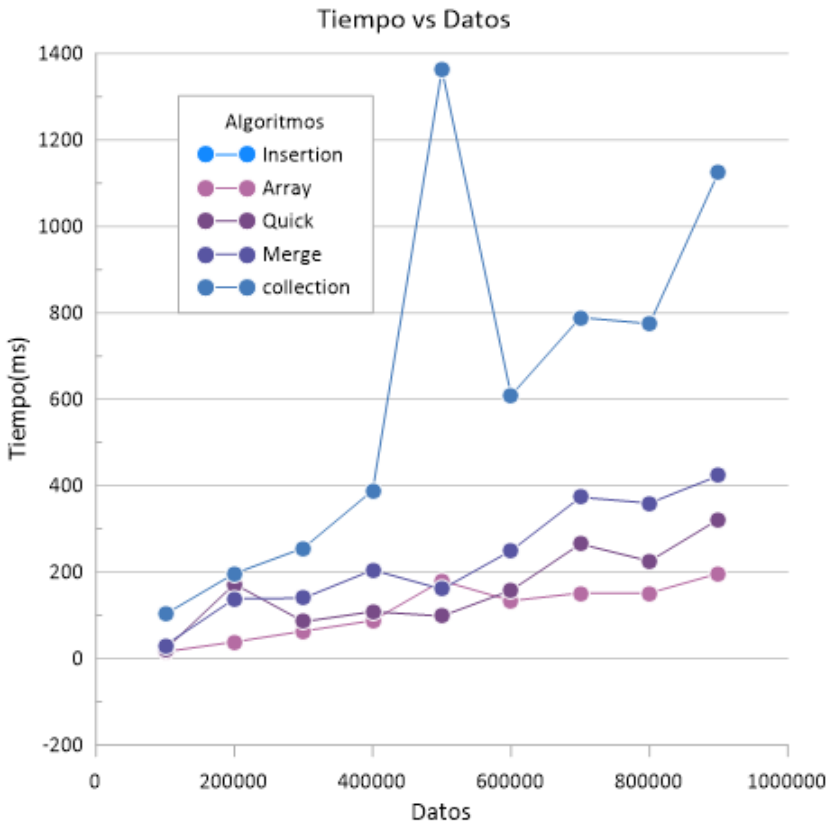


Figura 19: Gráfica Tiempo Vs Datos

8.6. Orden de magnitud 6

Orden de magnitud 6: 1.000.000, 2.000.000, 3.000.000, 4.000.000, 5.000.000, 6.000.000, 7.000.000, 8.000.000, 9.000.000.

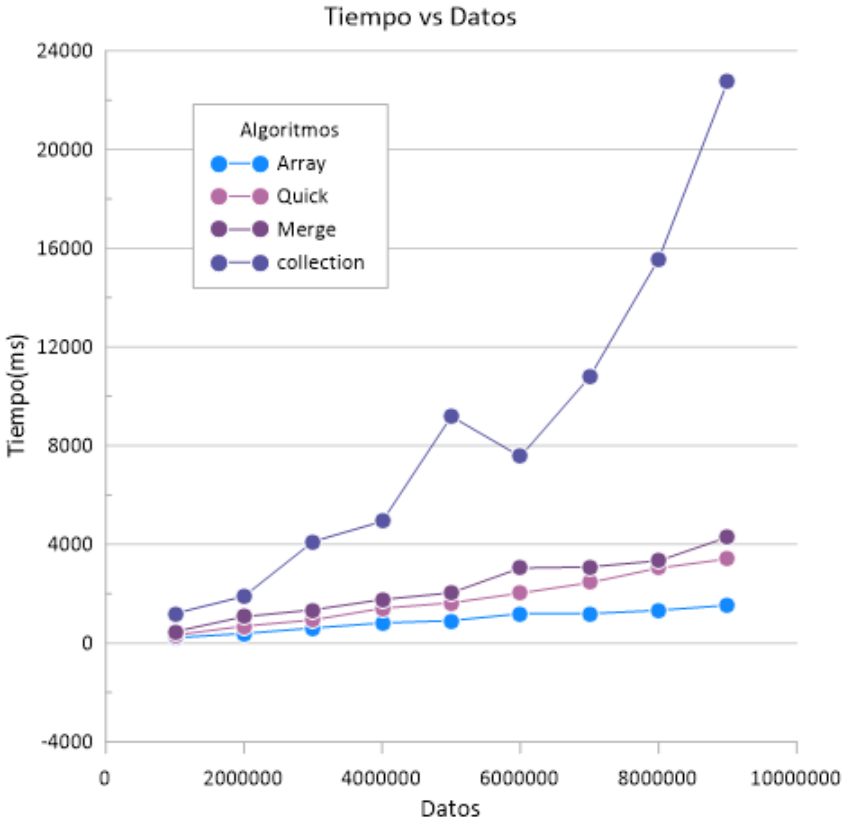


Figura 20: Gráfica Tiempo Vs Datos

9. Conclusión

Luego de haber obtenidos las gráficas que anteriormente se visualizó, en donde se puede observar claramente el comportamiento de los Algoritmos de Ordenamientos, se puede llegar a una conclusión en el cuál el ordenamiento de datos estará constatemente ligado a las magnitudes que queramos ordenar respectivamente.

Es por esto que como pudimos observar durante la actividad, existen una gran variedad de Algoritmos de Ordenamientos por lo cuál la verdadera problemática que ocurre es el escoger de buena forma cual de todos escogeremos y usaremos, para así cumplir con nuestro proposito de ordenar la secuencia que tengamos.

Las consideraciones anteriores son un claro ejemplo de los retos que se enfrentan día a día quienes trabajan en ciencias de la computación, ya que independiente del lenguajes de programación que escogamos, o herramientas, casi siempre habra un caso "Óptimo"que será el mejor si consideramos el factor tiempo de ejecución, ya que por ejemplo ordenar con Insertion Sort una cantidad muy grande de datos no será muy efectivo en comparación a otros tipos de Algoritmos de Ordenamiento que vimos durante el desarrollo del Software.

```
                                numeros ordenados   numeros random
Orden de magnitud 4: 10.000---------  0.052858          33.308717      milisegundos
                     20.000---------  0.067227          77.308596      milisegundos
                     30.000---------  0.09032           179.443493     milisegundos
                     40.000---------  0.125217          307.903186     milisegundos
                     50.000---------  0.156008          481.442499     milisegundos
                     60.000---------  0.181154          685.809284     milisegundos
                     70.000---------  0.210919          991.168617     milisegundos
                     80.000---------  0.252486          1214.962494    milisegundos
                     90.000---------  0.271475          1544.599534    milisegundos

                                numeros ordenados   numeros random
Orden de magnitud 5: 100.000--------- 0.514211           1922.660613   milisegundos
                     200.000--------- 0.858557           7677.992465   milisegundos
                     300.000--------- 0.90423            17234.067306  milisegundos
                     400.000--------- 1.248064           30758.824069  milisegundos
                     500.000--------- 1.549302           48099.322649  milisegundos
                     600.000--------- 1.910071           no óptimo     milisegundos
                     700.000--------- 2.197967           no óptimo     milisegundos
                     800.000--------- 2.412991           no óptimo     milisegundos
                     900.000--------- 2.740916           186983.622761 milisegundos

                                 numeros ordenados   numeros random
Orden de magnitud 6: 1.000.000--------- 3.108355         no óptimo     milisegundos
                     2.000.000--------- 6.41326          no óptimo     milisegundos
                     3.000.000--------- 9.101317         no óptimo     milisegundos
                     4.000.000--------- 12.231226        no óptimo     milisegundos
                     5.000.000--------- 15.422205        no óptimo     milisegundos
                     6.000.000--------- 18.822047        no óptimo     milisegundos
                     7.000.000--------- 21.321253        no óptimo     milisegundos
                     8.000.000--------- 24.372646        no óptimo     milisegundos
                     9.000.000--------- 27.559518        no óptimo     milisegundos
```

Figura 21: Ejemplo, Ordenamiento por Insertion Sort no Óptimo

10. Referencias Web

Referencias

[1] Diseño y Análisis de Algoritmos. (DAA-2009)–Dr. Eric Jeltsch F. *PDF Diseño y Análisis de Algoritmos*, Segunda Clase, DAA 2019.

[2] E78. INGENIERÍA DEL SOFTWARE 5º CURSO DE INGENIERÍA INFORMÁTICA 2000-2001, *Especificación de Requisitos Software según el estándar de IEEE 830*, Departament de Informática Universitario Jaume I.

[3] Introduction to algorithms y H.D. Sherali, *Thomas H. Cormen [et al.].—2nd ed*

[4] Diseño y Análisis de Algoritmos. *https://epubs.siam.org/doi/abs/10.1137/1.9781611972931.5*

[5] Diseño y Análisis de Algoritmos. *https://cafe.elharo.com/programming/java-programming/why-java-util-arrays-uses-two-sorting-algorithms/*

[6] Diseño y Análisis de Algoritmos. *https://www.geeksforgeeks.org/dual-pivot-quicksort/*

[7] Diseño y Análisis de Algoritmos. *https://www.goldensoftware.com/products/grapher/*

Appendices

Anexo I: Diagramas UML (Casos de Uso y de Clases)

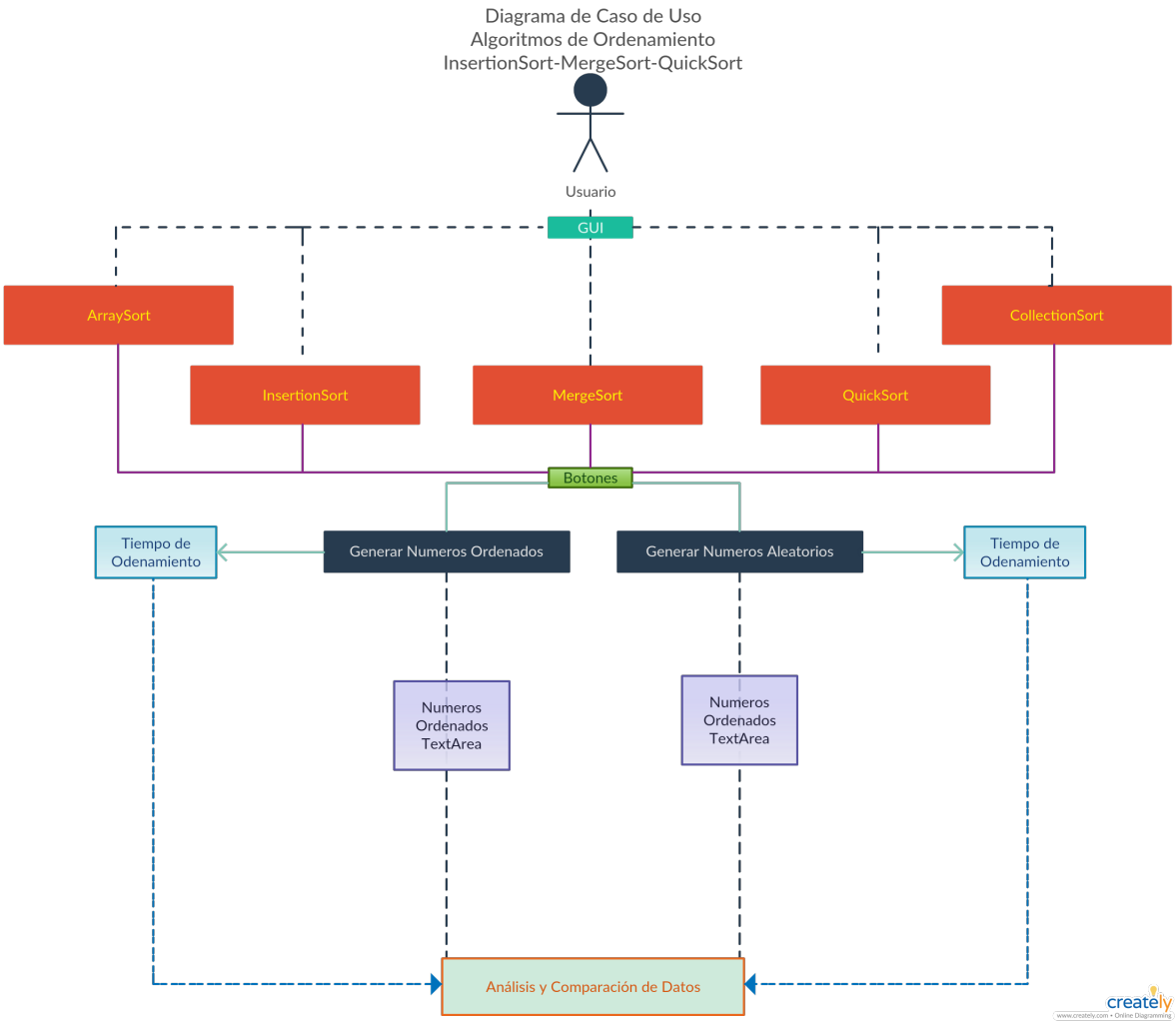


Figura 22: Diagrama Caso de Uso de Algoritmos de Ordenamiento

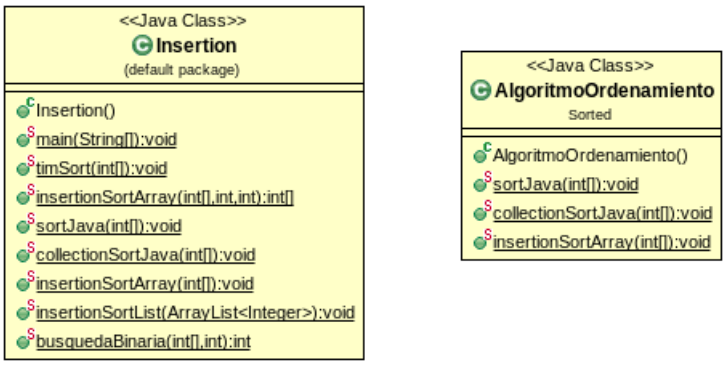


Figura 23: Diagrama Caso de Uso Clase IsertionSort-ArraySort-CollectionSort

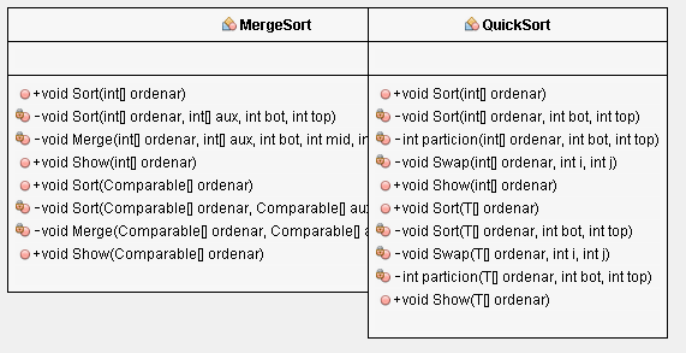


Figura 24: Diagrama Caso de Uso Clase MergeSort-QuickSort

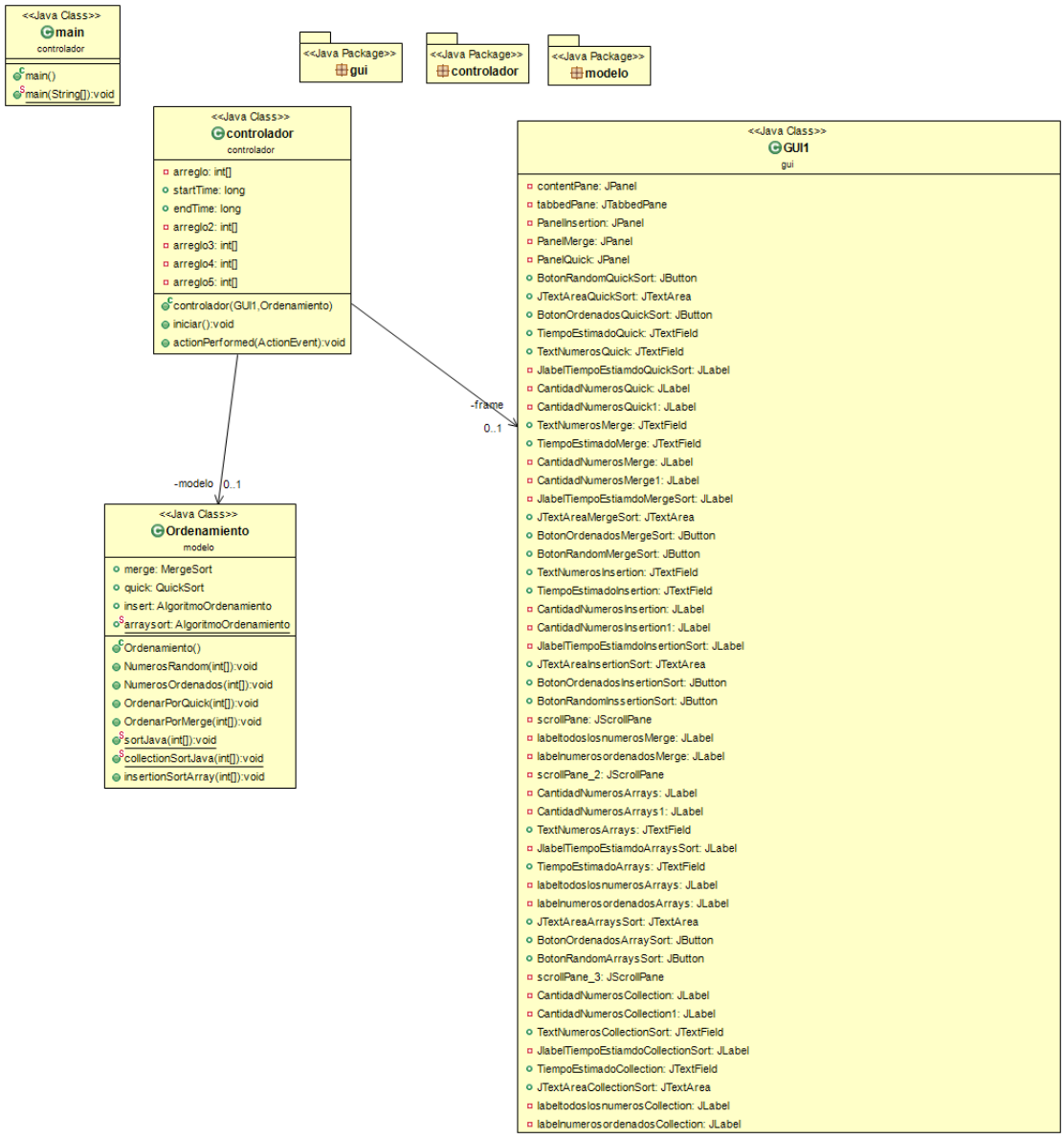
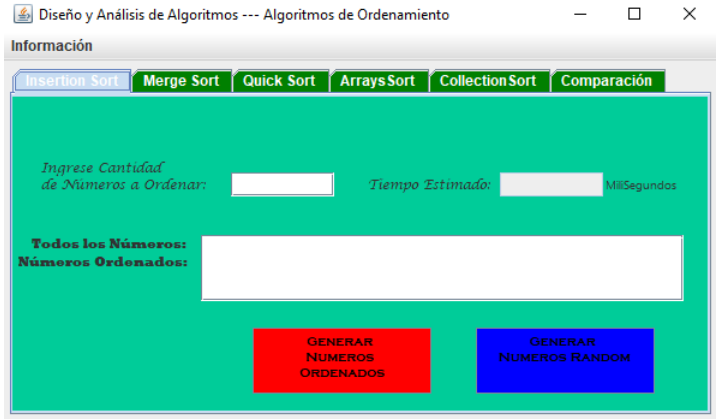


Figura 25: Diagrama Caso de Uso de Algoritmos de Ordenamiento

Anexo II: Manual de Usuario

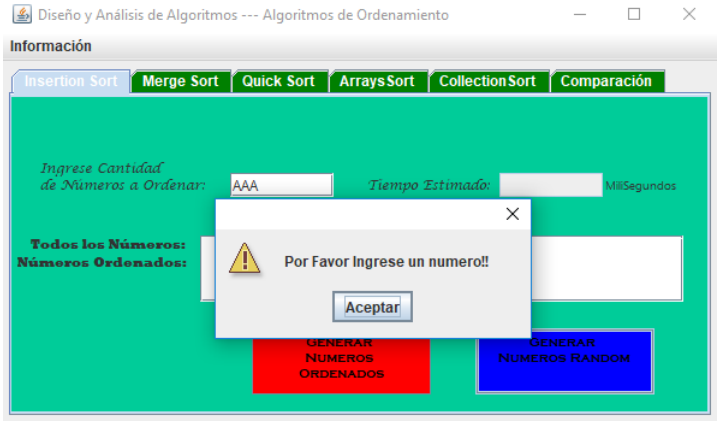
En la siguiente sección se dará a conocer el manual de usuario del software en donde trabajamos, para esto se visualizará a través de capturas de pantalla. Con esto lograremos ir adentrandonos al proyecto, ya que constantemente mencionaremos los componentes que integran la GUI.

Figura 26: Interfaz Principal Algoritmos de Ordenamiento



La Interfaz Gráfica contiene 6 pestañas en cuales gracias a un JTabbedPane vamos escogiendo cada panel. Debemos ingresar un numero(n) el cual se utilizara como la cantidad de números del arreglo que queremos ordenar.

Figura 27: Si ingresa otro caracter, este lo detendrá(Depurado)



El primer ordenamiento es el Insertion Sort, que ordenará una vez ingresemos el n y posteriormente seleccionemos un tipo de arreglo en él que queramos ordenar.

Figura 28: Insertion Sort para arreglo de numeros ordenados

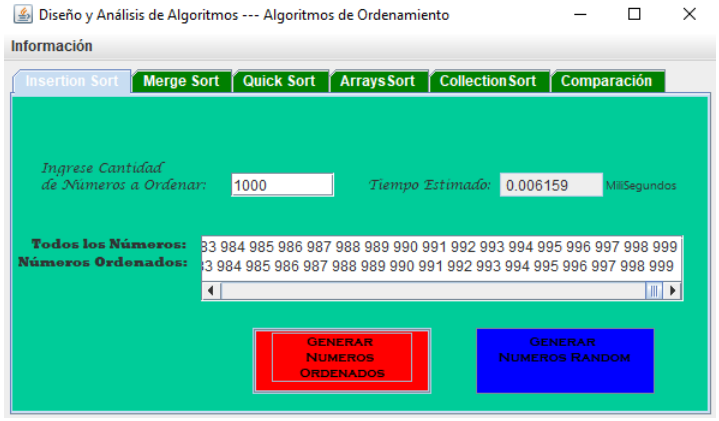
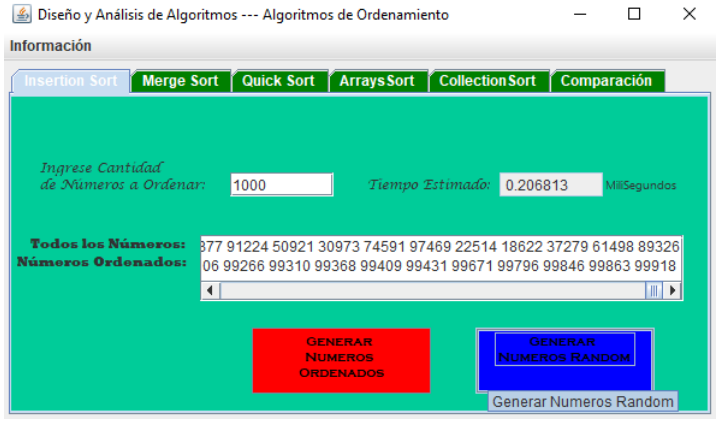


Figura 29: Insertion Sort para arreglo de numeros random



El segundo ordenamiento es el Merge Sort, que ordenará una vez ingresemos el n y posteriormente seleccionemos un tipo de arreglo en él que queramos ordenar.

Figura 30: Merge Sort para arreglo de números ordenados

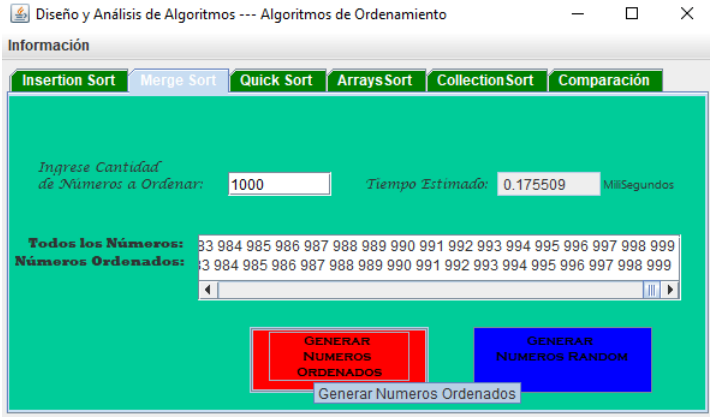
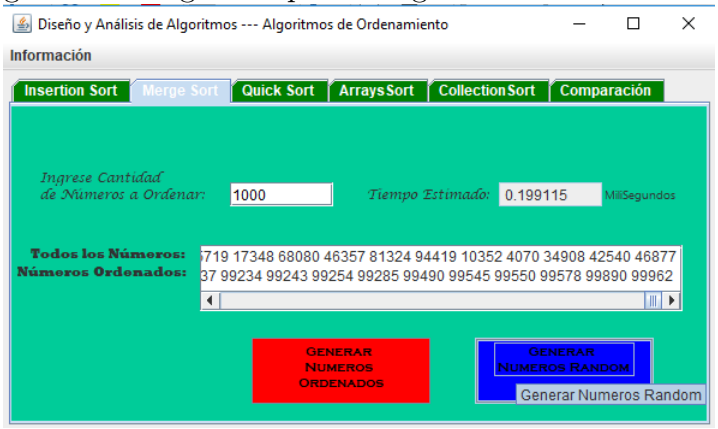


Figura 31: Merge Sort para arreglo de números random



El tercer ordenamiento es el Quick Sort, que ordenará una vez ingresemos el n y posteriormente seleccionemos un tipo de arreglo en él que queramos ordenar.

Figura 32: Quick Sort para arreglo de números ordenados

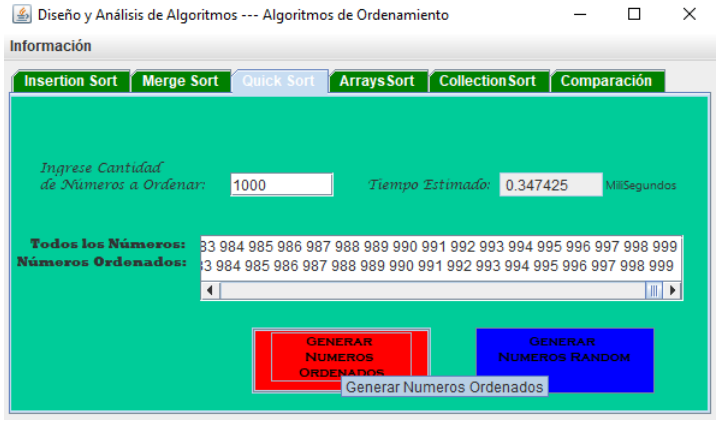


Figura 33: Quick Sort para arreglo de números random

El cuarto ordenamiento es el Arrays Sort, que ordenará una vez ingresemos el n y posteriormente seleccionemos un tipo de arreglo en él que queramos ordenar.

Figura 34: Array Sort para arreglo de números ordenados

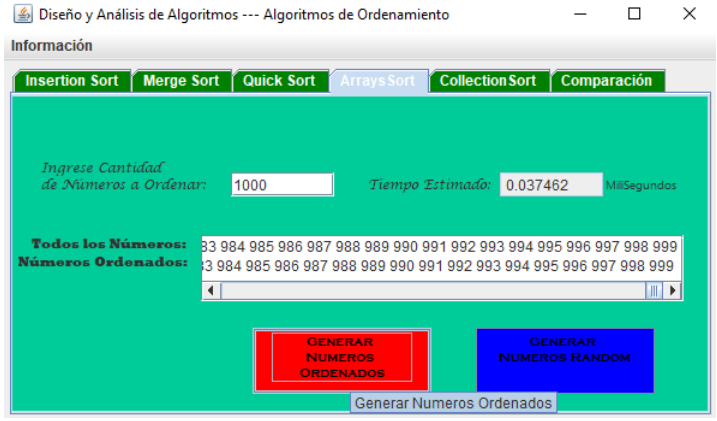
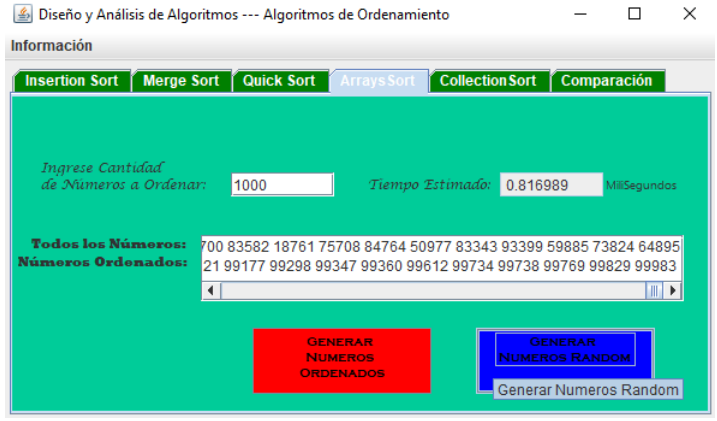


Figura 35: Array Sort para arreglo de números random



El quinto ordenamiento es el Collection Sort, que ordenará una vez ingresemos el n y posteriormente seleccionemos un tipo de arreglo en él que queramos ordenar.

Figura 36: Collection Sort para arreglo de números ordenados

Figura 37: Collection Sort para arreglo de números random

El último panel será el llamado Çomparación", en el cuál realiza el cálculo del tiempo de ejecución de todos los ordenamientos a la vez, para que así podamos analizar cada uno de ellos de manera inmediata, cabe mencionar que el n que ingresemos será la cantidad de numeros del arrreglo de números random.

Figura 38: Comparación de Algoritmos

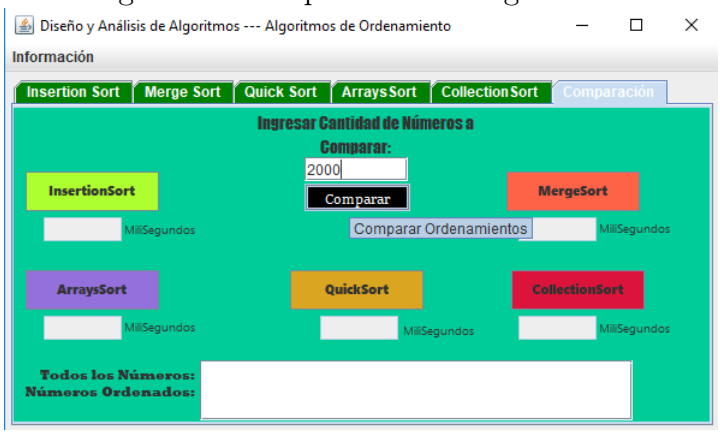


Figura 39: Comparación de Algoritmos

Figura 40: Comparación de Algoritmos (Consola)

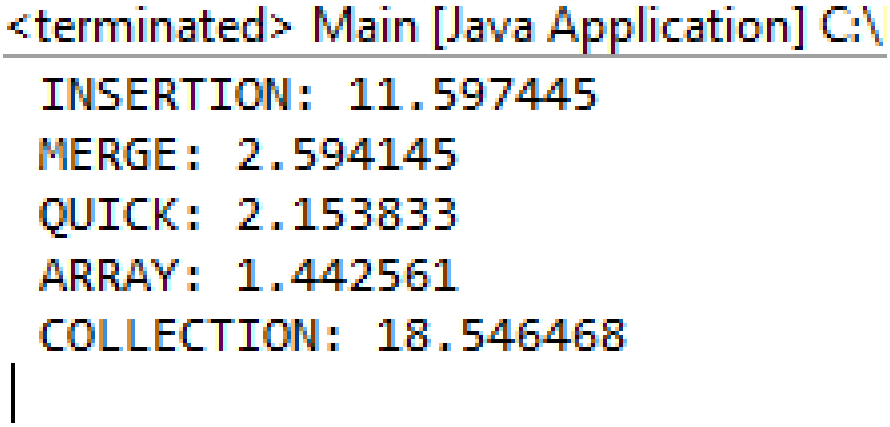


Figura 41: Información sobre los creadores del Software