Universidad Icesi – Facultad de Ingeniería
Algoritmos y estructuras de datos

Juan Camilo Olano Rosas – A00137588

Nicolás Javier Salazar Echeverry – A00348466

Introducción

El problema de ordenar datos (de cualquier tipo; int, String, long, Object , etc), es un problema que aparentemente puede ser sencillo pero para el cual se han desarrollado muchas estrategias para la solución del mismo; algunas de estas estrategias más eficientes que otras, pero , ¿cómo determinar que estrategias de ordenamiento son mejores que otras? En ciencias de la computación se le llama a estas estrategias, algoritmos de ordenamiento y el problema antes enunciado es con *El análisis de complejidad*

1. Análisis del problema:

El problema al cual nos enfrentamos es el probar diferentes algoritmos de ordenamiento los cuales deberán de ser seleccionados a través de diferentes criterios (los cuales serán especificados más adelante en esté informe), esto con el fin de saber cuales son los algoritmos más eficientes para implementarlos en los coprocesadores matemáticos de la empresa fabricante de microprocesadores.

1.1. Requerimientos Funcionales

Nombre	**R1. Ingresar datos numéricos**
Resumen	El programa permite ingresar datos para ser ordenados
Entradas	
Los datos numéricos	
Resultados	
Los valores están ordenados por in criterio seleccionado	

Nombre	**R2. Generar datos aleatorios**
Resumen	Genera datos completamente aleatorios para ser ordenados (especificando cantidad de datos)
Entradas	
Resultados	
Se generan nuevos datos para ser ordenados	

Nombre	**R3. Seleccionar cantidad de datos**
Resumen	El usuario selecciona la cantidad de datos que desea generar
Entradas	
Resultados	
Se genera la cantidad de datos seleccionada	

Nombre	**R4. Seleccionar intervalo de valores para el auto generado**
Resumen	Selecciona un intervalo de valores en los cuales se generaran los valores aleatorios
Entradas	
El intervalo de valores en el que se desean general los datos	
Resultados	
Se han generado los datos	

Nombre	**R5. Generar números repetidos**
Resumen	Selecciona si se generaran o no números repetidos
Entradas	
Resultados	
Generan los datos repetidos o no	

Nombre	**R5. Generar números repetidos**
Resumen	Selecciona si se generaran o no números repetidos
Entradas	
Resultados	
Generan los datos repetidos o no	

Nombre	R6. Ordenar de forma ascendente
Resumen	Ordena los datos numéricos de menor a mayor
Entradas	
Los datos por ordenar	
Resultados	
Los datos han sido ordenados	

Nombre	R7. Ordenar los datos de forma descendente
Resumen	Ordena los datos numéricos de mayor a menor
Entradas	
Los números que serán ordenados	
Resultados	
Los números han sido ordenados	

Nombre	R8. Ordenar de forma aleatoria
Resumen	Ordena los números de forma desordenada
Entradas	
Los números ha desordenar	
Resultados	
Los numeros se han ordenado de forma completamente aleatoria	

Nombre	R9. Desordenar en un porcentaje
Resumen	Desordena los datos en un cierto porcentaje
Entradas	
Porcentaje de desorden	
Resultados	
Los datos han sido desordenados	

Nombre	R10. Mostrar tiempo de ejecución del ordenamiento
Resumen	Muestra el tiempo que se a demorado un algoritmo para ordenar
Entradas	
Resultados	
Los datos han sido desordenados	

Nombre	R9. Desordenar en un porcentaje
Resumen	Desordena los datos en un cierto porcentaje
Entradas	
Porcentaje de desorden	
Resultados	
Los datos han sido desordenados	

1.2. Requerimientos no funcionales

Nombre	RNF1. Restringir tipo de datos
Resumen	Restringe la opción de algún algoritmo si este no puede ordenar los de cierto tipo
Entradas	
Resultados	

Nombre	RNF2. Leer los datos ingresados por el usuario
Resumen	Leer los datos ingresados por el usuario para determinar si todos los algoritmos pueden implementarse en estas entradas
Entradas	
Los datos ingresados por el usuario	
Resultados	

2. Recopilación de información

2.1. Algoritmos de ordenamiento

Los algoritmos de ordenamiento son aquellos procesos Que, como su nombre lo indica, se encargan de ordenar una serie de datos, los cuales pueden estar contenidos en vectores o matrices.

2.1.1. Algoritmo de inserción (insertion sort)

Este método toma cada elemento del arreglo para ser ordenado y lo compara con los que se encuentran en posiciones anteriores a la de él dentro del arreglo. Si resulta que el elemento con el que se está comparando es mayor que el elemento

que ordenar, se recorre hacia la siguiente posición superior. Si por el contrario, resulta que el elemento con el que se está comparando es menor que el elemento que ordenar, se detiene el proceso de comparación, pues se encontró que el elemento ya está ordenado y se coloca en su posición (que es la siguiente a la del último número con el que se comparó).

2.1.2. Algoritmo de selección (selection sort)

Consiste en encontrar el menor de todos los elementos del arreglo e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenar todo el arreglo.

2.1.3. Algoritmo burbuja (buble sort)

Se recorre el arreglo intercambiando los elementos adyacentes que estén desordenados. Se recorre el arreglo tantas veces hasta que ya no haya cambios. Prácticamente lo que hace es tomar el elemento mayor y lo va corriendo de posición en posición hasta ponerlo en su lugar

2.1.4. Algoritmo de Shell (Shellsort)

Este método utiliza una segmentación entre los datos; funciona comparando elementos que están distantes. La distancia entre comparaciones decrece conforme el algoritmo se ejecuta hasta la última fase, en la cual se comparan los elementos adyacentes, por esta razón se le llama también ordenación por disminución de incrementos.

2.1.5. Algoritmo de mezcla (merge sort)

La ordenación por mezcla o combinación de capas se basa en la estrategia de dividir para conquistar (divide y vencerás). Primero divide el arreglo en dos, ordena recursivamente cada una de las partes y luego las mezcla. El procedimiento debe entonces recibir el arreglo y un par de índices que delimitan los bordes de la ordenación.

2.1.6. Algoritmo de montículos (Heapsort)

Este algoritmo consiste en almacenar todos los elementos del vector que se desea ordenar en un montículo (heap), para luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado.

2.1.7. Algoritmo de ordenamiento rápido (Quicksort)

Consiste en elegir un elemento de la lista para ordenar, al que llamaremos pivote. Resituamos los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada.

2.1.8. BogoSort

Consiste en verificar si la lista o arreglo esta ordenado, si no lo esta los re ordena de forma aleatoria, luego se vuelve a verificar si esta ordenado y así sucesivamente hasta que con una combinación aleatoria el arreglo este ordenado

2.1.9. Radix sort

Este algoritmo (también llamado ordenamiento de raíz), es un algoritmo de ordenamiento de enteros el cual consiste en avaluar de forma individual cada uno de los dígitos de los números a ordenar y ordenarlos por sus cifras significativa, esto es: ordenando los datos del digito menos significativo al más significativo (a pesar de que está pensado para enteros se puede llegar a implementar para números de punto flotante o para cadenas de texto).

2.1.10. ordenamiento por cuentas (Counting sort)

este algoritmo solo es aplicable a números de tipo entero, no puede ser aplicado a otro tipo de datos (double, String, float, chart,etc) consiste en:

> El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya en el intervalo [*mínimo*, *máximo*], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados. (Wikipedia, 2017).

2.1.11. Block sort

consiste en dividir el arreglo en dos partes ordenarlas y luego insertar una en la otra (es una combinación del merge sort y el insertion sort)

2.1.12. ordenamiento por casilleros (Bucket sort)

consiste en crear “casilleros” o sub-arreglos los cuales son llenados con fragmentos del arreglo original para ser ordenados, luego se juntan cada uno de ellos y se asignan al arreglo original

2.1.13. Gnome sort

Es un algoritmo basado en la estrategia de burbuja que lo que hace es compara secciones contiguas del arreglo el menor lo pasa hasta la izquierda y el mayor se va corriendo poco a poco a la derecha

2.1.14. Comb sort

Consiste en usar una variable dentro del método (variable temporal) la cual por lo general tiene el valor de 1.3 (este valor también es conocido como el factor de encogimiento), esta variable es dividida entre el tamaño del arreglo, el resultado es la posición j la cual es comparada con la posición i y son intercambiadas si $j < i$ de tal forma que el arreglo quede ordenado así $i_n < j_n$ donde n representa las posiciones del arreglo desde la primera hasta la ultima

2.1.15. Algoritmo de burbuja bidireccional

Esta es otra variación del algoritmo de burbuja el cual consiste en ordenar desde los extremos del arreglo, es decir en la primera iteración ordena la primera y la última posición, luego la segunda y la n-1 posiciones y así hasta tener el arreglo ordenado.

3. búsqueda de soluciones creativas

Para esta etapa del método de ingeniería se decidió utilizar un proceso de SCAMPER el cual consiste en responder 7 preguntas (o la mayor parte de ellas) hechas en el contexto del problema de esta forma llegamos a estas ideas

A. Modificar el bogo sort para que agrupe los elementos parcialmente arreglados en arreglos temporales para luego juntarlos en un solo arreglo ordenado

B. Ejecutar el programa en computadoras con una velocidad de procesamiento de 3,5 GHz y una memoria RAM de 16gb
C. Combinar el merge sort con el selection sort, es decir separa los datos en grupos y luego ordenar estas partes por selección para después juntar todo
D. Utilizar el bogo sort una sola vez para generar los datos de forma aleatoria, es decir generar datos de forma ordenada y luego utilizar el bogo sort para desordenarlos
E. Utilizar alguno de los algoritmos de ordenamiento ineficientes (como el bublue sort) para ordenar solamente datos ordenados es decir restringir solamente esta opción para cuando la entrada sean datos ya ordenados
F. Usar bucket sort de forma "recursiva" es decir dividir los casilleros en sud-casilleros mas pequeños para ser ordenados de forma interna y luego ordenarlos hasta llagar a los casilleros mas grandes y al final tener el arreglo ordenado
G. Usar solo las tres variaciones del algoritmo de burbuja, ya que los tres son algoritmos estables.
H. Usar un simulador online para visualizar como funciona cada uno de los algoritmos de ordenamiento y en base a esto tomar una decisión de que algoritmos utilizar en la solución del problema
I. Cambiar cualquier algoritmo recursivo a una versión de este que sea iterativo
J. Diseñar una forma de especificar que tipo de dato se quiere tratar.

4. Diseño preliminar:

4.1. Descartar ideas:

A. Modificar el bogo sort: esta idea es descartada puesto que modificarlo de esta forma, es más complicado que simplemente usar otro algoritmo
B. Ejecutar el programa en equipos específicos es una idea también descartada puesto que limitaría el uso del programa además de que no poseemos un equipo con estas especificaciones
C. La combinación del merge con el selection podría funcionar, pero nos arriesgamos a aumentar la complejidad del caso promedio del algoritmo de merge
D. Dividir el buket sort para tener "casilleros dentro de casilleros "es una idea que aparentemente aumentaría la complejidad del algoritmo, y lo volvería un poco mas engorroso de lo que ya es.
E. Usar las tres variaciones del algoritmo de burbuja es un atentado contra el tiempo de ejecución del programa ya que si bien los tres son estables los tres son n^2

4.2. Ideas aprobadas:

A. Utilizar el bogo sort no para ordenar si no para desordenar los datos que se auto generen, en el programa, si bien este algoritmo es de complejidad O(n x n!), creemos que se puede usar una sola vez para desordenar los datos.
B. Usar algún algoritmo de ordenamiento de complejidad O(n^2), pero que sea estable (como el buble sort , el selection sort o el insertion sort), esto con el fin de tener una opción para ordenar en el mejor de los casos (cuando los datos estén completamente ordenados)
C. Cualquier algoritmo que sea escogido y tenga una estructura recursiva deberá ser cambiado a una de forma iterativa esto con el fin de, primero minimizar la complejidad espacial del algoritmo y la de facilitar el calculo de la complejidad temporal del algoritmo.
D. Usar el simulador online para visualizar el funcionamiento de los algoritmos de ordenamiento para esto hemos decidido utilizar este: http://www.algostructure.com/sorting/gnomesort.php, en base a este ultimo aspecto se decidió hacer un filtro y escoger los siguientes algoritmos como candidatos a hacer parte de la solución del problema: insertion sort, Shell sort, Quicksort, Radix sort, Block sort , Gnome sort , merge sort
E. Especificar el tipo de dato que se desea tratar bien sea para generar datos de ese tipo o para recibirlos el usuario deberá seleccionar si desea trabajar con datos de tipo entero o real

5. Evaluación de las ideas preliminares

5.1. Establecer criterios:

Los criterios que estableceremos son:

A. Complejidad temporal (1-8): este criterio se basa en darle a cada complejidad un valor así por ejemplo la complejidad O(1) tendrá el valor de 8 y la complejidad O(n!) tendrá una calificación de 1
B. Complejidad espacial (1-4) : este criterio se evalúa de forma similar al anterior, solo que en una escala mas pequeña ya que la complejidad temporal de estos algoritmos solo llega en el peor de los casos a O(n^2) la cual tiene una calificación de 1 , la de O(1) tiene una calificación de 1
C. Intuitivo (1-3): en este caso es un criterio que decidimos colocar, para caracterizar que algunos de estos algoritmos son un poco más compactos,

más fáciles de entender y/o más fáciles de comprender a simple vista; según nuestra percepción esta escala se califica del 1 al 3 según nos parezca de menos intuitivo a mas intuitivo respectivamente

D. Estabilidad del algoritmo (1-2): este criterio se basa en una calificación simple si es estable tiene 2 y si no tiene 1
E. Favoritismo personal (1-5): esta medida es una subjetiva y se basa únicamente en cuanto nos ha llamado la atención alguno de estos algoritmos.

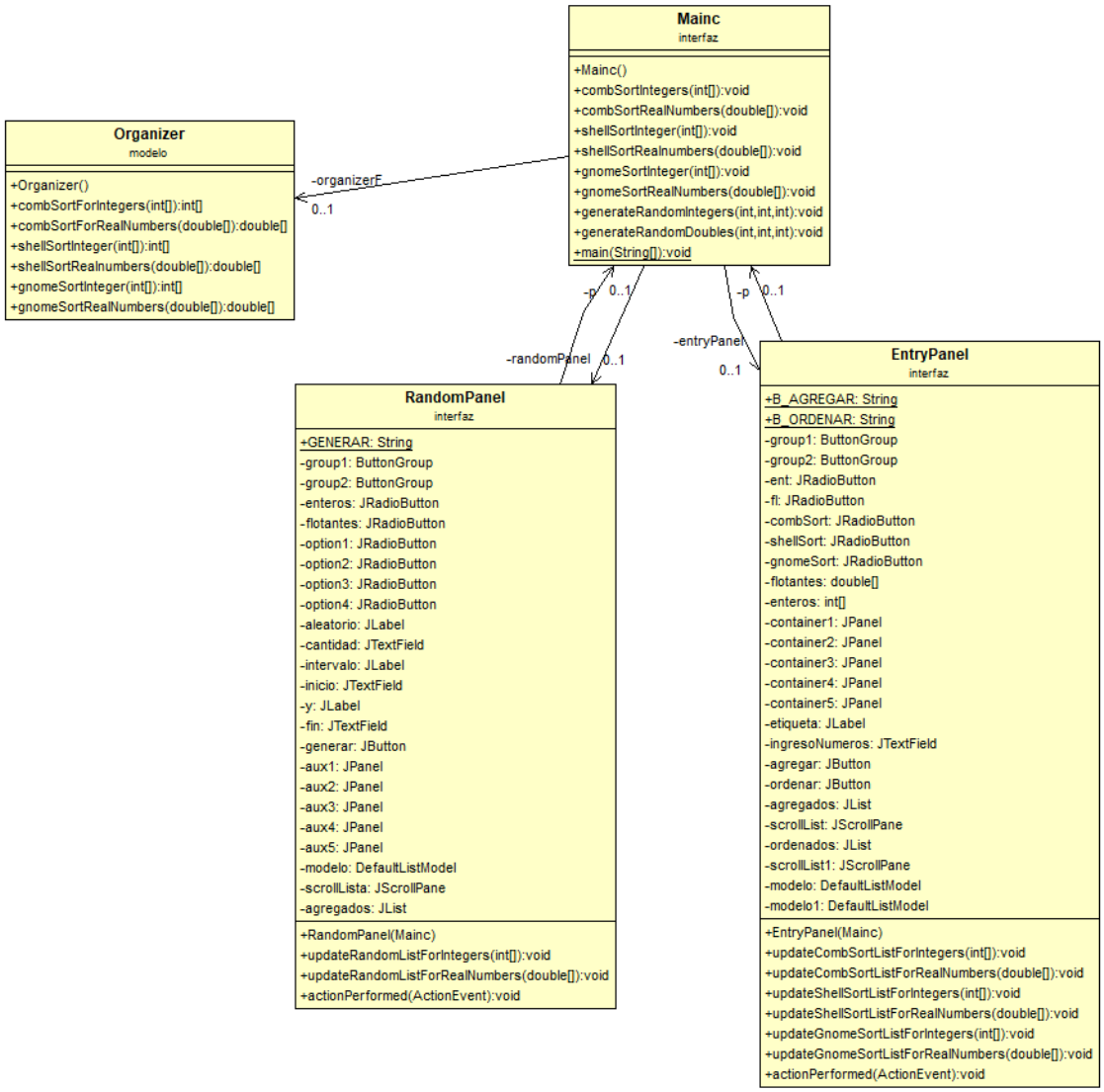
Estos criterios han sido especificados en esta tabla, cada criterio tiene un intervalo entre paréntesis esta será la calificación individual de cada criterio de menos viable (1) a más viable (número mayor que se encuentre en el intervalo)

Algoritmo	C. temporal	C. espacial	Intuitivo	estabilidad	Favoritismo personal	Total
Insertion sort	3	4	3	2	1	13
Shell sort	4	4	2	1	3	14
Quicksort	5	2	1	1	3	12
Radix sort	6	3	1	2	1	13
comb sort	5	4	3	1	4	17
Gnome sort	3	4	3	2	4	16
Merge sort	5	3	1	2	2	13

Bajo estos criterios se ha decidido que los tres algoritmos que se utilizaran en el desarrollo del problema son: comb sort, gnome sort y el Shell sort.

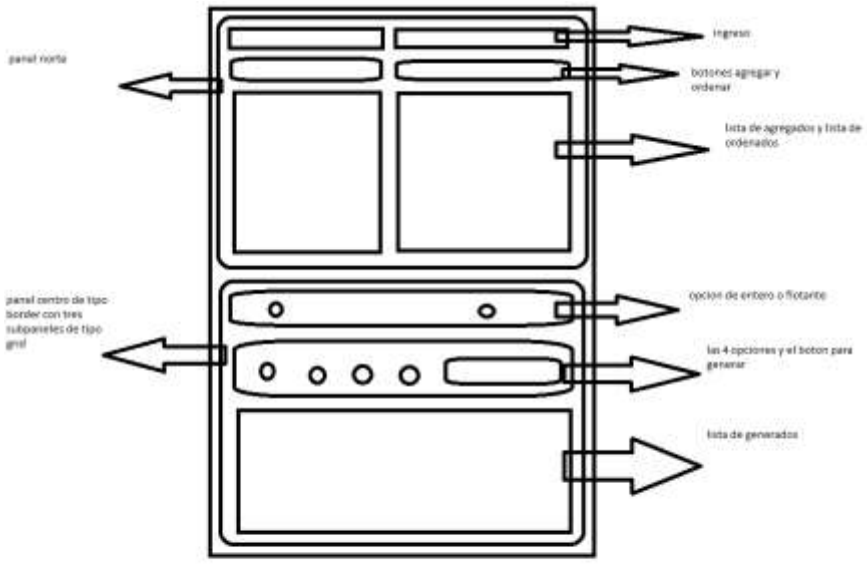
6. preparación de informes y especificaciones.

6.1. Diagrama de clases de la solución.



6.2. Diseño de la interfaz de usuario.

Para la interfaz de usuario decidimos hacer este diseño preliminar a como se verá.



6.3. Pseudcódigo de los algoritmos más relevantes.

6.3.1. **Comb sort**:

```
1. function combsort(array input)
2.     gap := input.size //initialize gap size
3.     loop until gap = 1 and swaps = 0
4.         //update the gap value for a next comb. Below is
an example
5.         gap := int(gap / 1.25)
6.         if gap < 1
7.           //minimum gap is 1
8.           gap := 1
9.         end if
10.        i := 0
11.        swaps := 0 //see Bubble Sort for an explanation
       //a single "comb" over the input list
12.        loop until i + gap >= input.size //see Shell sort
for similar idea
13.            if input[i] > input[i+gap]
14.                swap(input[i], input[i+gap])
15.                swaps := 1 // Flag a swap has occurred, so
the
                           // list is not guaranteed sorted
```

```
16.             end if
17.             i := i + 1
18.         end loop
19.     end loop
20. end function
```

(tomado de: https://rosettacode.org/wiki/Sorting_algorithms/Comb_sort):

6.3.2. **Gnome sort**

```
1.  function gnomeSort(a[0..size-1])
2.    i := 1
3.    j := 2
4.    while i < size do
5.        if a[i-1] <= a[i] then
6.           // for descending sort, use >= for comparison
7.            i := j
8.          j := j + 1
9.     else
10.          swap a[i-1] and a[i]
11.        i := i - 1
12.        if i = 0 then
13.            i := j
14.            j := j + 1
15.        endif
16.    endif
17.  Done
```

(tomado de: https://rosettacode.org/wiki/Sorting_algorithms/Gnome_sort):

6.3.3. **Shell sort**

```
1. PROCEDURE tri_Insertion ( Tableau a[1:n],gap,debut)
2.     POUR i VARIANT DE debut A n AVEC UN PAS gap FAIRE
3.         INSERER a[i] à sa place dans a[1:i-1];
4. FIN PROCEDURE;
5.
6. PROCEDURE tri_shell ( Tableau a[1:n])
7.     POUR gap DANS (6,4,3,2,1) FAIRE
8.        POUR debut VARIANT DE 0 A gap - 1 FAIRE
9.           tri_Insertion(Tableau,gap,debut);
10.            FIN POUR;
11.         FIN POUR;
```

```
12.     FIN PROCEDURE;
```

(tomado de: http://lwh.free.fr/pages/algo/tri/tri_shell_es.html)

6.4. Diseño de pruebas unitarias:

Prueba No:	1
Tipo de Prueba:	Unitaria Automática
Método a Probar:	combSortForIntegers(long[] numbers)
Entradas:	El arreglo que será ordenado

Prueba No:	2
Tipo de Prueba:	Unitaria Automática
Método a Probar:	combSortForRealNumbers(duoble[] numbers)
Entradas:	Arreglo que será ordenado

Prueba No:	3
Tipo de Prueba:	Unitaria Automática
Método a Probar:	shellSortInteger(long arr[])
Entradas:	Arreglo que será ordenado

Prueba No:	4
Tipo de Prueba:	Unitaria Automática
Método a Probar:	shellSortRealnumbers(double arr[])
Entradas:	Arreglo que será ordenado

Prueba No:	5
Tipo de Prueba:	Unitaria Automática
Método a Probar:	gnomeSortInteger(long[] a)
Entradas:	Arreglo que será ordenado

Prueba No:	6
Tipo de Prueba:	Unitaria Automática
Método a Probar:	gnomeSortRealNumbers(double[] a)
Entradas:	Arreglo que será ordenado

6.5. Análisis de complejidad temporal:

6.6. Análisis de complejidad espacial:

7. Implementación

en el siguiente enlace a github esta el control de versiones del laboratorio:

https://github.com/camdnd/laboratorio1.git

Webgrafía

- Rosetta Code(2016) *comb sort* recuperado de http://rosettacode.org/wiki/Comb_sort
- Rosetta Code(2016) *Shell sort* recuperado de rosettacode.org/wiki/Shell_sort
- Rosetta Code(2016) *Gnome sort* recuperado de http://rosettacode.org/wiki/Gnome_sort
- Algostructure(2018) *merge sort* recuperado de http://www.algostructure.com/sorting/mergesort.php
- GeeksforGeeks *Counting Sort* recuperado de https://www.geeksforgeeks.org/counting-sort/