**Laboratorio 1**

**Juan Felipe Castillo Rincón**

**Método de la ingeniería**

**Contexto problemático:**

Una empresa de fabricación de microprocesadores requiere de algoritmos de ordenamiento, para evaluar la posibilidad de implementarlos como instrucciones básicas de su próximo coprocesador matemático.

**Fase 1: Identificación del problema**

Se identifican concretamente las necesidades de la problemática, así como sus causas y síntomas.

*Identificación de necesidades y síntomas*

* La empresa desea evaluar la implementación de algoritmos de ordenamiento en su próximo coprocesador.
* Se desea acelerar el rendimiento del procesador principal, evitando que realice tareas de cómputo intensivo.
* La solución al problema debe ser la más eficiente de acuerdo con el tipo de datos usados.
* Los algoritmos de ordenamiento deben adecuarse a los datos a ordenar.

**Fase 2: Recopilación de información**

Con el objetivo de reconocer las diferentes soluciones para implementar en el problema, se realiza una búsqueda de conceptos y algoritmos de ordenamientos para tener clara la solución del problema.

Fuentes:

<https://es.wikipedia.org/wiki/Coma_flotante>

<https://blog.zerial.org/ficheros/Informe_Ordenamiento.pdf>

<https://www.cs.buap.mx/~iolmos/ada/AlgoritmosOrdenamientoP1.pdf>

<http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap03.htm>

<https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento>

<https://es.slideshare.net/rolfpinto/algortimos-de-ordenamiento>

<http://www.algorithmist.com/index.php/Counting_sort>

*Números en formato de coma flotante*

Es una forma de notación científica usada en los microprocesadores con la cual se pueden representar números racionales extremadamente grandes y pequeños de una manera muy eficiente y compacta, y con la que se pueden realizar operaciones aritméticas.

*Algoritmos de ordenamiento*

Existen diferentes algoritmos de ordenamientos, puede ser simples y otros que son mas óptimos.

*Métodos iterativos*

Los algoritmos que se encuentran en estos métodos son:

* Burbuja
* Selección
* Inserción
* Shellsort: Este algoritmo es más eficiente cuando la lista está semi-ordenada, por lo tanto, se tomará en cuenta cuando el usuario genere números de los cuales una parte se encuentre ordenada.

*Métodos recursivos*

Estos métodos son más complejos de entender, pero más rápidos y efectivos, dentro de los algoritmos recursivos se encuentran.

* Mergesort (Ordenamiento por mezcla):
* Quicksort (Ordenamiento rápido): Este será uno de los algoritmos a implementar debido a su rapidez gracias a las llamadas recursivas.

*HeapSort*

Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él.

*BucketSort*

Es un algoritmo de ordenamiento que distribuye todos los elementos a ordenar entre un número finito de casilleros. Cada casillero sólo puede contener los elementos que cumplan unas determinadas condiciones. En el ejemplo esas condiciones son intervalos de números. Las condiciones deben ser excluyentes entre sí, para evitar que un elemento pueda ser clasificado en dos casilleros distintos.

*CountingSort*

Es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).

*ShellSort*

Denominado así por su desarrollador Donald Shell (1959), ordena una estructura de una manera similar a la del Bubble Sort, sin embargo, no ordena elementos adyacentes, sino que utiliza una segmentación entre los datos. Esta segmentación puede ser de cualquier tamaño de acuerdo con una secuencia de valores que empiezan con un valor grande (pero menor al tamaño total de la estructura) y van disminuyendo hasta llegar al '1'.

**Fase 3: Búsqueda de soluciones creativas**

Mediante una lluvia de ideas se pensó en las formas mas pertinente de solucionar el problema. Esta técnica de generación de ideas consiste en que se generan ideas espontaneas para resolver el problema en específico:

1. Implementar los algoritmos más fáciles de entender y de programar con una interfaz sencilla pero útil para el usuario, para así solucionar el problema de forma rápida.
2. Implementar los 3 algoritmos de ordenamiento más rápidos recomendados por la internet.
3. Implementar los siguientes algoritmos de ordenamiento: ShellSort, CountingSort,BucketSort.
4. Entre los 3 algoritmos más fáciles de desarrollar, implementar cada uno de acuerdo a los valores generados por el usuario o el software.
5. Realizar una interfaz amigable con el usuario mediante el uso de JavaFX.
6. Generar todos los valores en desorden e implementar cualquier algoritmo de ordenamiento.
7. Implementar sólo métodos recursivos para hacer el ordenamiento más rápido.

**Fase 4: Transición de las ideas a los diseños preliminares**

Primero que todo se descartan las opciones que son factibles para la solución.

La alternativa 1 y 4 se descarta debido a la complejidad temporal de los algoritmos, ya que al ser los más sencillos de implementar y de entender, toma una gran cantidad de pasos y tiempo para hacer el ordenamiento.

La alternativa 2 y 6 se descarta por el hecho que no cumplirían con la solución al problema. La alternativa 7 no se toma en cuenta ya que aun no sabemos calcular la complejidad de algoritmos recursivos.

Por lo tanto, opciones que quedan son la 3 y 5, estas son las alternativas a implementar.

Alternativa 2: La implementación del algoritmo ShellSort en este problema es útil porque el usuario puede seleccionar la opción de generar una parte de los valores en orden, en estos casos el algoritmo es más útil. Los algoritmos CountingSort y BucketSort también se implementar ya que su complejidad (vista mas adelante) en rápida y se pueden aplicar a este problema.

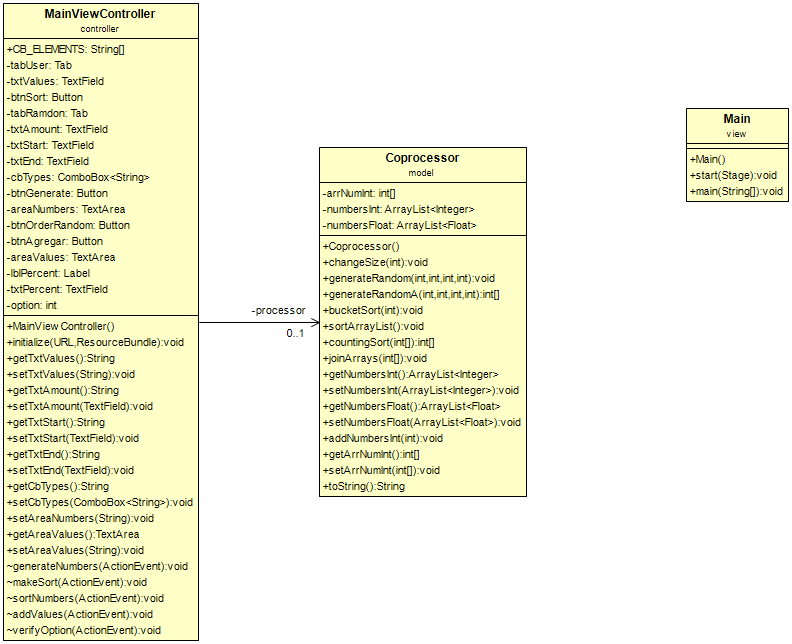
Alternativa 5: Esta opción también se implementar por el hecho de que una interfaz realizada en JavaFX es más estética, más rápida de realizar y mejor para el usuario.

**Paso 5. Evaluación y Selección de la Mejor Solución**

Anteriormente se mencionaron las razones por las cuales estas dos alternativas son las mejores y se pueden realizar juntas, por lo tanto éstas serian las mejores opciones a implementar.

**Paso 6. Preparación de Informes y Especificaciones**

*Diagrama de clases*



*Pseudocódigo de algoritmos más relevantes*

***ShellSort***

*INICIO*

*ENTERO INTERVALO, K, J, I, AUX*

*INTERVALO <- N DIV 2*

*MIENTRAS(INTERVALO > 0){*

*PARA(I <- INTERVALO - 1, HASTA (- N, I (- I + 1){*

*J <- I - INTERVALO*

*MIENTRAS(J >= 0){*

*K <- J + INTERVALO*

*SI(VECTOR[K] <= VECTORlJ] ){*

*AUX (- VECTORlJ]*

*VECTORlJ] (- VECTOR[K]*

*VECTOR[K] (- AUX*

*}*

*SI NO{*

*J <- 0*

*}*

*J <- J - INTERVALO*

*}*

*}*

*INTERVALO <- INTERVALO DIV 2*

***BucketSort***

función bucket-sort(elementos, n)

casilleros ← colección de n listas

para i = 1 hasta longitud(elementos) hacer

c ← buscar el casillero adecuado

insertar elementos[i] en casillero[c]

fin para

para i = 1 hasta n hacer

ordenar(casilleros[i])

fin para

devolver la concatenación de casilleros[1],..., casilleros[n]

***CountingSort***

for i in 0 to (K - 1)

counts[i] = 0

for each input number n

counts[n] = counts[n] + 1

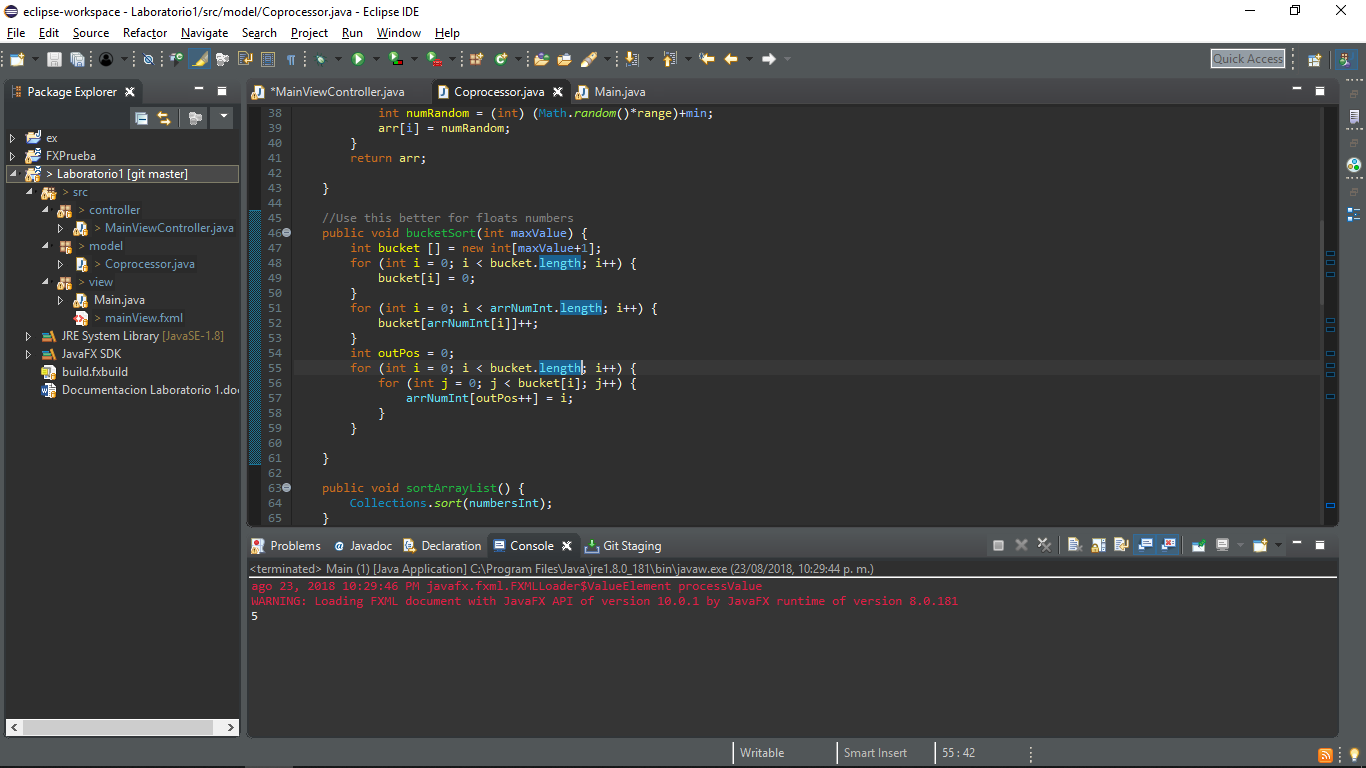
for i in 0 to (K - 1)

assert( 0 <= counts[i] )

for j in 0 to counts[i]

output i

|  |  |
| --- | --- |
|  |  |
|  | 1 |
|  | n |
|  | n-1 |
|  | n |
|  | n-1 |
|  | 1 |
|  | n |
| n | n-1 |
| n | (n-2) |



|  |  |
| --- | --- |
|  |  |
|  | 1 |
|  | 2 |
|  | n |
|  | 4(n-1) |
|  |  |
|  |  |
|  | 1 |
|  | n |
|  | n-1 |
|  | 1 |
|  | n |
|  | n-1 |
|  | n |
|  | n-1 |
|  | 1 |

