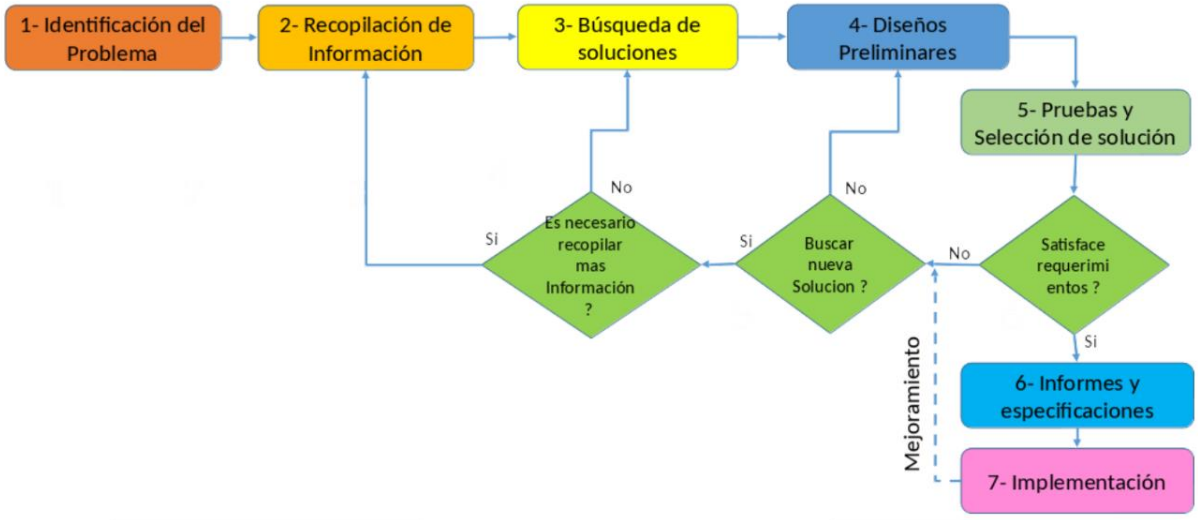
Método de la Ingeniería
Aplicación para la Solución del Laboratorio 1
Realizado por:
Juan Sebastian Cardona Sanchez
Jeferson Lerma Girón
Daniel Gómez

Desarrollo de la Solución

Para resolver la situación anterior se eligió el Método de la Ingeniería para desarrollar la solución siguiendo un enfoque sistemático y acorde con la situación problemática planteada.

Con base en la descripción del Método de la Ingeniería del libro "Introduction to Engineering" de Paul Wright, se definió el siguiente diagrama de flujo, cuyos pasos seguiremos en el desarrollo de la solución.



Fase 1. Identificación del problema

Contexto problemático -identificando causas y síntomas-:

Una empresa de fabricación de microprocesadores está evaluando la posibilidad de implementar varios algoritmos de ordenamiento como instrucciones básicas de su próximo coprocesador matemático, por costos/beneficio la empresa ha decidido implementar tres (3) algoritmos diferentes de ordenamiento que permitan ordenar, muy rápidamente, números enteros de tamaño arbitrariamente grande y números en formato de coma flotante de cualquier tamaño, por

tal motivo la empresa requiere de nosotros para la selección e implementación del prototipo de pruebas en software de los algoritmos que finalmente serán implementados como operaciones nativas en hardware.

Identificación y definición concreta y sin ambigüedad del problema:

Definición del Problema

La empresa de fabricación de microprocesadores requiere que hagamos el respectivo análisis para elegir los algoritmos más apropiado que permitan ordenar, muy rápidamente, números enteros de tamaño arbitrariamente grande y números en formato de coma flotante de cualquier tamaño, y con esto elegir al mejor entre ellos que el cual será implementado para que haga las operaciones nativas en hardware.

Especificación de los requerimientos funcionales asociados con las necesidades planteadas en el enunciado

Nombre	Ingresar valores a ordenar
Resumen	Se va a crear la opción para que el usuario si quiere, puedan ingresar ciertos valores para poder ordenarlos.
Entrada	- Valores a ordenar
Salida	Se han guardado correctamente todos los valores que se quieren ordenar.

Nombre	Generar valores aleatoriamente.
Resumen	Se le va a crear la opción para que el usuario pueda generar valores (tanto enteros como de coma flotante) aleatoriamente y poder configurar la cantidad total de números a generar, además debe permitir indicar si los números a generar deben ser todos diferentes o pueden haber repetidos.
Entrada	- El usuario ha aceptado la opción de generar los valores aleatoriamente. - Intervalo generador. - Cantidad de datos a generar.
Salida	Se han generado correctamente los valores aleatoriamente con el intervalo generador y la cantidad de datos a generar, además se verifica que los datos a generar deben ser todos diferentes o pueden haber repetidos.

Nombre	Configuración de generación aleatoria

Resumen	Se le va a crear la opción para que el usuario tenga la posibilidad de escoger cierta configuración de generación aleatoria, permitiéndole elegir entre: - que los valores estén ya ordenados - que los valores estén ordenados inversamente. - que los valores estén en orden completamente aleatorio - que los valores estén desordenados en un % indicado por el usuario
Entrada	- El usuario ha elegido una de las opciones disponibles.
Salida	Se ha configurado y seleccionado la generación aleatoria de la serie de valores, por lo que la serie de valores queda en ese orden correspondiente.

Nombre	Generar valores ordenados
Resumen	Se va a crear un método capaz de ordenar una serie de valores.
Entrada	-
Salida	Los valores se han ordenados correctamente

Nombre	Generar valores ordenados inversamente
Resumen	Se va a crear un método capaz de ordenar una serie de valores de forma inversa.
Entrada	-
Salida	Los valores se han ordenado de manera inversamente.

Nombre	Generar los valores en orden completamente aleatorio
Resumen	Se va a crear un método capaz de ordenar una serie de valores de forma completamente aleatoria.
Entrada	-
Salida	Los valores se han ordenados de forma completamente aleatoria.

Nombre	Generar los valores en desorden en un % indicado por el usuario
Resumen	Se va crear un método capaz de ordenar una serie de valores con un % de desorden indicada por el usuario.
Entrada	- % de desorden
Salida	Los valores se han desorganizado con él % de desorden indicado por el usuario.

Nombre	Parte 1 para sacar % de desorden
Resumen	Se va a generar ordenadamente la secuencia de datos.
Entrada	-
Salida	La serie de números de han ordenado correctamente.

Nombre	Parte 2 para sacar % de desorden
Resumen	Con base en el tamaño de la secuencia y el % de desorden se obtiene un número k de cuantas posiciones deben estar desordenadas.
Entrada	-
Salida	Se ha calculado el k de cuantas posiciones deben estar desordenadas

Nombre	Parte 3 para sacar % de desorden
Resumen	Se va a generan k/2 pares de posiciones diferentes y se intercambian los valores entre cada par de ellas
Entrada	-
Salida	Se a generan k/2 pares de posiciones diferentes y se intercambian los valores entre cada par de ellas para la serie de valores correspondientes.

Nombre	Analizar, comprobar y escoger un algoritmo apropiado para el ordenamiento de los valores.
Resumen	Se van a tomar varios métodos de ordenamientos y se van a analizar, comprobar y entre ellos escoger los mejores, para quedarse finalmente con uno que es el que se va a usar para todos los algoritmos de ordenamiento del software
Entrada	-
Salida	Se escoge el algoritmos más apropiado para el ordenamiento de todos los valores.

Nombre	Mostrar el tiempo que toma el método de ordenamiento.
Resumen	Se le va a mostrar al usuario el tiempo que toma el ordenamiento realizado (Mostrando el tipo de numero a ordenar y el intervalo de números generados.)
Entrada	-
Salida	Se muestra el tiempo que tomo el método de ordenamiento en cuestión.

Nombre	Restringir/Permitir
Resumen	Se va a dejar poder restringir/permitir los algoritmos para ordenar.
Entrada	-
Salida	Se restringe o permite usar los algoritmos correspondientes para ordenar.

Fase 2. Recopilación de la información necesaria

Con el objetivo de tener total claridad en los conceptos involucrados se hace una búsqueda de las definiciones de los términos matemáticos más estrechamente relacionados con el problema planteado. Es importante realizar esta búsqueda en fuentes reconocidas y confiables para conocer cuáles elementos hacen parte del problema y cuáles no, esto para resolver el problema de la manera más efectiva posible.

Fuente:

http://correo.uan.edu.mx/~iavalos/FP/FP1.html
https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento
https://www.geeksforgeeks.org/counting-sort/
http://www.java2novice.com/java-sorting-algorithms/selection-sort/
https://www.javacodex.com/Sorting/Bucket-Sort
https://www.geeksforgeeks.org/insertion-sort/
https://www.javatpoint.com/bubble-sort-in-java
https://es.wikipedia.org/wiki/Ordenamiento_por_cuentas
https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n
https://es.wikipedia.org/wiki/Ordenamiento_por_casilleros
https://es.wikipedia.org/wiki/Ordenamiento_por_inserci%C3%B3n
https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja
http://articulos.conclase.net/?tema=ordenacion&art=cuenta&pag=000
http://lwh.free.fr/pages/algo/tri/tri_selection_es.html#tabs-algo

Algoritmo:

Un algoritmo es una secuencia de pasos lógicos necesarios para llevar a cabo una tarea específica, como la solución de un problema.

Algoritmo de ordenamiento:

un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada.

Para entender cómo funciona un ordenamiento lo definiremos de forma matemática para su comprensión global, esto es:

Tenemos S tal que:

$$S = \{ a_0,\ a_{1,}\ a_2,\ a_3, \dots, a_n\}$$

Donde $a_0,\ a_{1,}\ a_2,\ a_3, \dots, a_n$ son los datos de entrada y perecen a los números naturales y S es el conjunto de los datos.

Luego de ordenar nuestro conjunto S, obtenemos el conjunto K tal que:

$$K = \{ a_0\ \leq\ a_{1,}\ a_2\ \leq\ \ a_3 \dots, a_n \leq\ a_{n+1}\}$$

El cual está completamente ordenado de forma ascendente.

En esencia nuestro problema radica en utilizar algoritmos de ordenamientos eficientes y eficaces que nos permitan ordenar números extremadamente grandes.

Para poder solucionar el problema buscamos problemas similares y soluciones a esos problemas que a ciencia cierta ya conocemos, como por ejemplo ordenar los números de 1 al 100

Tenemos S tal que:

$$S = \{ a_0, a_1, a_2, a_3, \dots, a_n \}$$

Donde $a_0, a_1, a_2, a_3, \dots, a_n$ donde n = 100 y n pertenece a los números naturales, por lo que.

Luego de ordenar nuestro conjunto S, obtenemos el conjunto K tal que:

$$K = \{ a_0 \leq a_1, a_2 \leq a_3 \dots, a_n \leq a_{n+1} \}$$

El cual está completamente ordenado de forma ascendente.

Y donde K queda:

$$K = \{ 1 \leq 2, 2 \leq 3 \dots, 99 \leq 100 \}$$

También podemos ordenar una serie de números de tal forma que sin importar la entrada de los datos ordenara en otro conjunto de números de forma creciente solo los números pares, estos lo podemos ver más formalmente de tal forma que:

Tenemos S tal que:

$$S = \{ a_0, a_1, a_2, a_3, \dots, a_n \}$$

Donde $a_0, a_1, a_2, a_3, \dots, a_n$ donde a_n = 2m y m pertenece a los números enteros. Por lo que nuestros números ordenados de forma decreciente tal que nuestro conjunto K este de dado por:

$$K = \{ a_n \geq a_{n-1}, a_{n-1} \geq a_{n-2} \dots \}$$

El cual está completamente ordenado de forma decreciente.

Sin duda ordenar una serie de números es de vital importancia para todo tipo de software, desde identificar más rápidamente una serie de números o datos desorganizados, hasta poder con ello tener más claridad de los datos y poder hacer un estudio o análisis más limpio y profundo y con ello deducir una interpretación más veraz y confiable, ayudando también a reducir un tiempo completamente significativo.

Fase 3. Búsqueda de soluciones creativas

Es por ello, que, para nuestra solución puntual, con relación a lo expresado anteriormente, es esencial buscar unos algoritmos capaces de resolver enormes cantidades de datos en unos tiempos óptimos, algunos algoritmos, para encontrar la solución hemos hecho una investigación con los algoritmos más usamos y otros con los que podríamos trabajar debido a que podría ser óptimo para resolver nuestro actual problema, algunos los cuales que conocemos son: Counting Sort, Selection Sort, Bucket Sort, Insertion Sort y Bubble Sort

Counting Sort:

El ordenamiento por cuentas (Counting Sort) 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).

En seudocódigo tenemos:

```
function countingSort(array, min, max):
  count: array of (max - min + 1) elements
  initialize count with 0
  for each number in array do
    count[number - min] := count[number - min] + 1
  done
  z := 0
  for i from min to max do
    while ( count[i - min] > 0 ) do
      array[z] := i
      z := z+1
      count[i - min] := count[i - min] - 1
    done
  done
```

Selection Sort:

Algoritmo de ordenamiento por Selección (Selection Sort en inglés): Consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo más pequeño, y así sucesivamente hasta ordenarlo todo.

En pseudocodigo tenemos:

```
Procedimiento_selection_sort( Vector a[1:n])
  Para i Variando de 1 hasta n - 1 hacer
    Encontrar[j] el elemento más pequeño de [i + 1:n];
    Intercambiar [j] Y [i];
Fin procedimiento;
```

Bucket Sort:

El ordenamiento por casilleros (Bucket Sort o Bin Sort, en inglés) 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 pseudocodigo tenemos:

```
Procedimiento_bucketSort(array, n) is
  buckets ← Nuevo array de n listas vacías
  for i = 0  hasta (length(array)-1) hacer
    insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
    nextSort(buckets[i]);
  return the concatenation of buckets[0], ..., buckets[n-1]
```

Insertion Sort:

El ordenamiento por inserción (Insertion Sort) es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria.

Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

En pseudocodigo tenemos:

```
i ← 1
while i < length(A)
   j ← i
   while j > 0 and A[j-1] > A[j]
      swap A[j] and A[j-1]
      j ← j - 1
   end while
   i ← i + 1
end while
```

Bubble Sort

La Ordenación de burbuja (Bubble Sort en inglés) es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillo de implementar.

En pseudocodigo tenemos:

```
Procedimiento_bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
      newn = 0
      for i = 1 to n-1 inclusive do
         if A[i-1] > A[i] then
            swap(A[i-1], A[i])
            newn = i
         end if
      end for
      n = newn
   until n = 0
end Procedimiento
```

Fase 4. Transición de la formulación de ideas a los diseños preliminares

Se han escogido los 3 algoritmos de ordenamiento, los cuales fueron seleccionados debido a que cumplen con las especificaciones que se encuentran en el enunciado del laboratorio, una de ellas fue que los algoritmos permitan ordenar números enteros. Los siguientes son los algoritmos de ordenamiento que utilizaremos para el análisis de complejidad, y posteriormente serán implementados en el lenguaje de programación java, utilizando el software eclipse:

[1]Counting Sort.

Lo escogimos debido a que su complejidad temporal y espacial es buena(O(n)), lo que brinda a nuestro programa que funcione correctamente y no ocupe grandes cantidades de espacio en memoria al momento de llevar a cabo un ordenamiento utilizando countingSort.

[2]BucketSort.

Este algoritmo de ordenamiento funciona utilizando otros arreglos en donde separa los datos de una forma interesante y los va ordenando, hasta que al final reúne todos los datos de cada arreglo y los une formando el arreglo con todos los datos organizados ascendentemente. Es bastante eficiente y tiene una complejidad de O(n)

[3]SelectionSort.

El algoritmo Selection sort sirve para ordenar diferentes tipos de datos, pero principalmente es usado para ordenar datos de tipo entero. Es un poco peculiar, ya que al ejecutarse ocupa mucho espacio en memoria (según gráficas y vídeos que investigamos), pero es uno de los algoritmos que realiza menos iteraciones mientras dura el tiempo de ejecución, y esto hace que acabe más rápido que otros algoritmos. Lastimosamente tiene una complejidad de O(n^2).

Descartamos los otros algoritmos como el bubble Sort porque su complejidad espacial es de O(n^2), lo que hace que ocupe mucho espacio en memoria y no sea muy eficiente en ciertos casos, por ejemplo, si el arreglo a ordenar esta ordenado inversamente, normalmente este algoritmo realiza muchos accesos a los arreglos que recorre internamente. El algoritmo insertion Sort también tiene una complejidad temporal de O(n^2), sin embargo, al ver sus gráficas de desempeño encontramos que realiza muchas comparaciones y muchos accesos al arreglo que recorre y ordena, lo que hace que se tome un tiempo importante realizando el ordenamiento en comparación con otros algoritmos.

Metodología:

A continuación, se presentan los algoritmos de ordenamiento con su análisis de complejidad espacial y temporal respectivamente, en el siguiente orden: 1) CountingSort, 2) SelectionSort y 3) BucketSort.

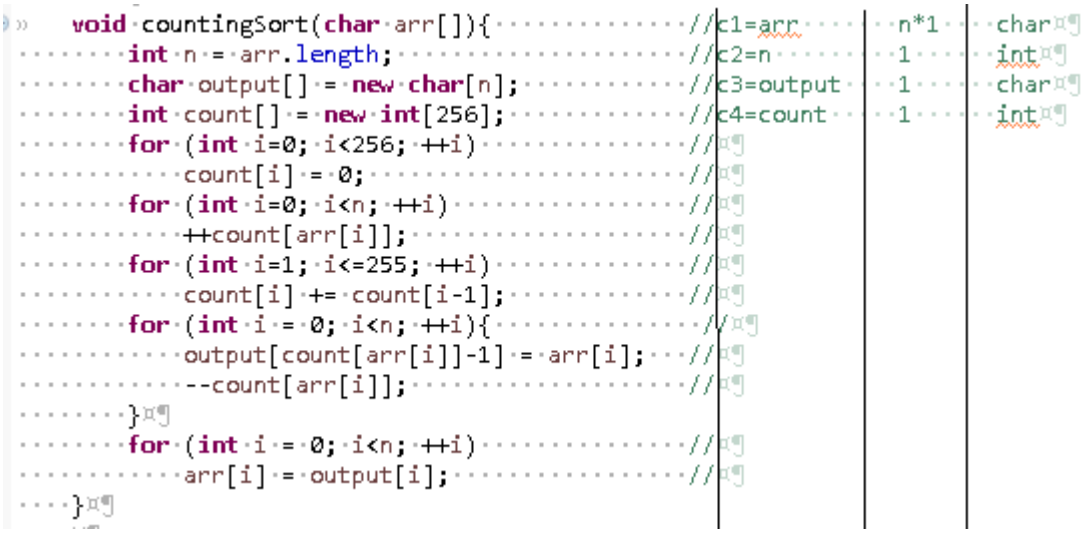
Análisis de complejidad Espacial, Algoritmo de ordenamiento CountingSort

Precondiciones:

arr: es un arreglo de n posiciones

n= arr.length

Algoritmo	Costo	#veces	Tipo de variable



T(n) = n+c1*1+c2*1+c3*1

T(n) = n+ k(c1+c2+c3)

T(n) = n+k

T(n) = n

T(n) = **0(n)**

Análisis de complejidad Temporal, Algoritmo de ordenamiento CountingSort

Caracterización de la entrada:

arr: es un arreglo de n posiciones

n= arr.length

Algoritmo	Costo	#Veces

```
    void countingSort(char arr[]){                    //
        int n = arr.length;                           //c1    1
        char output[] = new char[n];                  //c2    1
        int count[] = new int[256];                   //c3    1
        for (int i=0; i<256; ++i)                     //c4    256+1
            count[i] = 0;                             //c5    255
        for (int i=0; i<n; ++i)                       //c6    n+1
            ++count[arr[i]];                          //c7    n
        for (int i=1; i<=255; ++i)                    //c8    255+1
            count[i] += count[i-1];                   //c9    255
        for (int i = 0; i<n; ++i){                    //c10   n+1
            output[count[arr[i]]-1] = arr[i];         //c11   n
            --count[arr[i]];                          //c12   n
        }
        for (int i = 0; i<n; ++i)                     //c13   n+1
            arr[i] = output[i];                       //c14   n
    }
```

T(n) = c1*1+c2*1+c3*1+c4*(256+1)+c5*255+c6*(n+1)+c7*n+c8*(255+1)+c9*255+c10*(n+1)+c11*n+c12*n+c13*(n+1)+c14*n

T(n) = c1+c2+c3+c4*257+c5*255+c6n+c6+c7n+c8*256+c9*255+c10n +c10+c11n+c12n+c13n+c13+c14n

T(n)= k*(c1+c2+c3+c4*257+c5*255+c6+c7+ c8*256+c9*255+c10+c13)+ n*(c6+c7+c10+c11+c12+c13+c14)

T(n) = n*7+k*1029

T(n) = **O(n)**

Análisis de complejidad Espacial, Algoritmo de ordenamiento SelectionSort

Caracterización de la entrada:

arr: es una arreglo de n posiciones.

n= arr.length

Algoritmo	Costo	#Veces	Tipo de variable

```
void selectionSort(int arr[]){          //c1=arr          n*1     int
    int n = arr.length;                 //c2=n            1       int
    for (int i = 0; i < n-1; i++){
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

T(n) = c1*n +c2*1

T(n) = c1n+c2
T(n) = n+k
T(n) = **O(n)**

Análisis de complejidad Temporal, Algoritmo de ordenamiento SelectionSort

Caracterización de la entrada:

arr: es una arreglo de n posiciones.

n= arr.length

Algoritmo	Costo	#veces

```
void selectionSort(int arr[]){          //      Mejor caso          Peor Caso
    int n = arr.length;                 //c1    |1                  |1
    for (int i = 0; i < n-1; i++){      //c2    |n                  |n
        int min_idx = i;                //c3    |n-1                |n-1
        for (int j = i+1; j < n; j++){  //c4    |((n)^2/2)+(n/2)-1  |((n)^2/2)+(n/2)-1
            if (arr[j] < arr[min_idx]){//c5     |((n)^2/2)+(n/2)    |((n)^2/2)+(n/2)
                min_idx = j;}           //c6    |0                  |((n)^2/2)+(n/2)
        }                               //      |                   |
        int temp = arr[min_idx];        //c7    |n-1                |n-1
        arr[min_idx] = arr[i];          //c8    |n-1                |n-1
        arr[i] = temp;                  //c9    |n-1                |n-1
    }                                   //      |                   |
}                                       //      |                   |
```

En el mejor caso:

T(n) =c1+c2n+c3(n-1)+c4[((n)^2/2)+(n/2)-1]+c5[((n)^2/2)+(n/2)]+c6*0+c7(n-1)+c8(n-1)+c9(n-1)

T(n) =c1-c3-c4-c7-c8-c9 +c2n+c3n+c7n+c8n+c9+c4((n)^2/2)+ c5((n)^2/2)+ c5(n/2)+c4(n/2)

T(n) =-(-c1+c3+c4+c7+c8+c9)+ n[c2+c3+c7+c8+c9+ c5(1/2)+c4(1/2)]+ (n^2)[c4(1/2)+c5(1/2)]

T(n) =-k(4)+n(6)+n^2

T(n) =(n)^2+6n-4k = **O(n^2)**

En el peor caso:

T(n) =c1+c2n+c3(n-1)+c4[((n)^2/2)+(n/2)-1]+c5[((n)^2/2)+(n/2)]+ c6[((n)^2/2)+(n/2)]+c7(n-1)+ c8(n-1)+ c9(n-1)

T(n) =c1-c3-c4-c7-c8-c9+c2n+c3n+c4(n/2)+c5(n/2)+c6(n/2)+c7n+c8n+c9n+ c4((n)^2/2)+ c5((n)^2/2) c6((n)^2/2)

T(n) =-(-c1+c3+c4+c7+c8+c9)+n[c2+c3+c4(1/2)+c5(1/2)+ c6(1/2)+c7+c8+c9]+(n^2)[c4(1/2)+c5(1/2) c6(1/2)]

T(n) =-k(4)+n(13/2)+ 3((n)^2)/2)

T(n) =3((n)^2)/2)+(13n/2)-4k = **O(n^2)**

Análisis de complejidad Espacial, Algoritmo de ordenamiento BucketSort

*Caracterización de la entrada:

a: es un arreglo de n posiciones.

n= a.length.

maxVal: es el tamaño del arreglo a.

m: se inicia con el valor de maxVal.

Algoritmo	Costo	#Veces	Tipos de variables

z

```
public static void BucketSort(int[] a, int maxVal) {  //c1=a[],maxVal          n*1,1     int
                                                      //
    int [] bucket=new int[maxVal+1];                  //c2=bucket[]            n*1       int
    for (int i=0; i<bucket.length; i++) {             //
        bucket[i]=0;                                  //
    }                                                 //
    for (int i=0; i<a.length; i++) {                  //
        bucket[a[i]]++;                               //
    }                                                 //
    int outPos=0;                                     //c3=outPos              1         int
    for (int i=0; i<bucket.length; i++) {             //
        for (int j=0; j<bucket[i]; j++) {             //
            a[outPos++]=i;                            //
        }                                             //
    }                                                 //
}                                                     //
```

T(n) =c1n+c1+c2n+c3

T(n) =n(c1+c2)+ k(c1+c3)

T(n) =n(2)+k(2)

T(n) =**O(n)**

Análisis de complejidad Temporal Algoritmo de ordenamiento BucketSort

*Caracterización de la entrada:

a: es un arreglo de n posiciones.

n= a.length.

maxVal: es el tamaño del arreglo a.

m: se inicia con el valor de maxVal.

Algoritmo	Costo	#Veces

```
public static void BucketSort(int[] a, int maxVal) {
    int [] bucket=new int[maxVal+1];          //c1  |  1
    for (int i=0; i<bucket.length; i++) {     //c2  |  m+1
        bucket[i]=0;                          //c3  |  m
    }                                         //    |
    for (int i=0; i<a.length; i++) {          //c4  |  n+1
        bucket[a[i]]++;                       //c5  |  n
    }                                         //    |
    int outPos=0;                             //c6  |  1
    for (int i=0; i<bucket.length; i++) {     //c7  |  m+1
        for (int j=0; j<bucket[i]; j++) {     //c8  |  m+1
            a[outPos++]=i;                    //c9  |  m
        }                                     //    |
    }                                         //    |
}                                             //    |
```

T(n) = c1*1+c2(m+1)+c3*m+c4(n+1)+c5*n+c6*1+c7(m+1)+c8(m+1)+c9*m

T(n) =c1+c2+c4+c6+c7+c8+c2m+c3m+c7m+c8m+c9m+c4n+c5n

T(n) =k(c1+c2+c4+c6+c7+c8)+m(c2+c3+c7+c8+c9)+n(c4+c5)

T(n) = 6k+5m+2n

T(n) = **O(n)**

Fase 5. Evaluación y selección de la mejor solución

	Criterios		
A	El algoritmo de ordenamiento está clasificado como estable.		
	si=2	no=1	
B	El algoritmo de ordenamiento es fácil de entender		
	si=3	no=1	
C	El algoritmo tiene complejidad lineal		
	si=4	no=2	
D	¿El algoritmo tiene la misma complejidad temporal y espacial?		
	si=3	no=1	
E	El algoritmo tarda mucho tiempo en terminarse de ejecutar con una misma cantidad de datos		
	si=0	no=3	
F	El algoritmo es no comparativo		
	si=3	no=1	

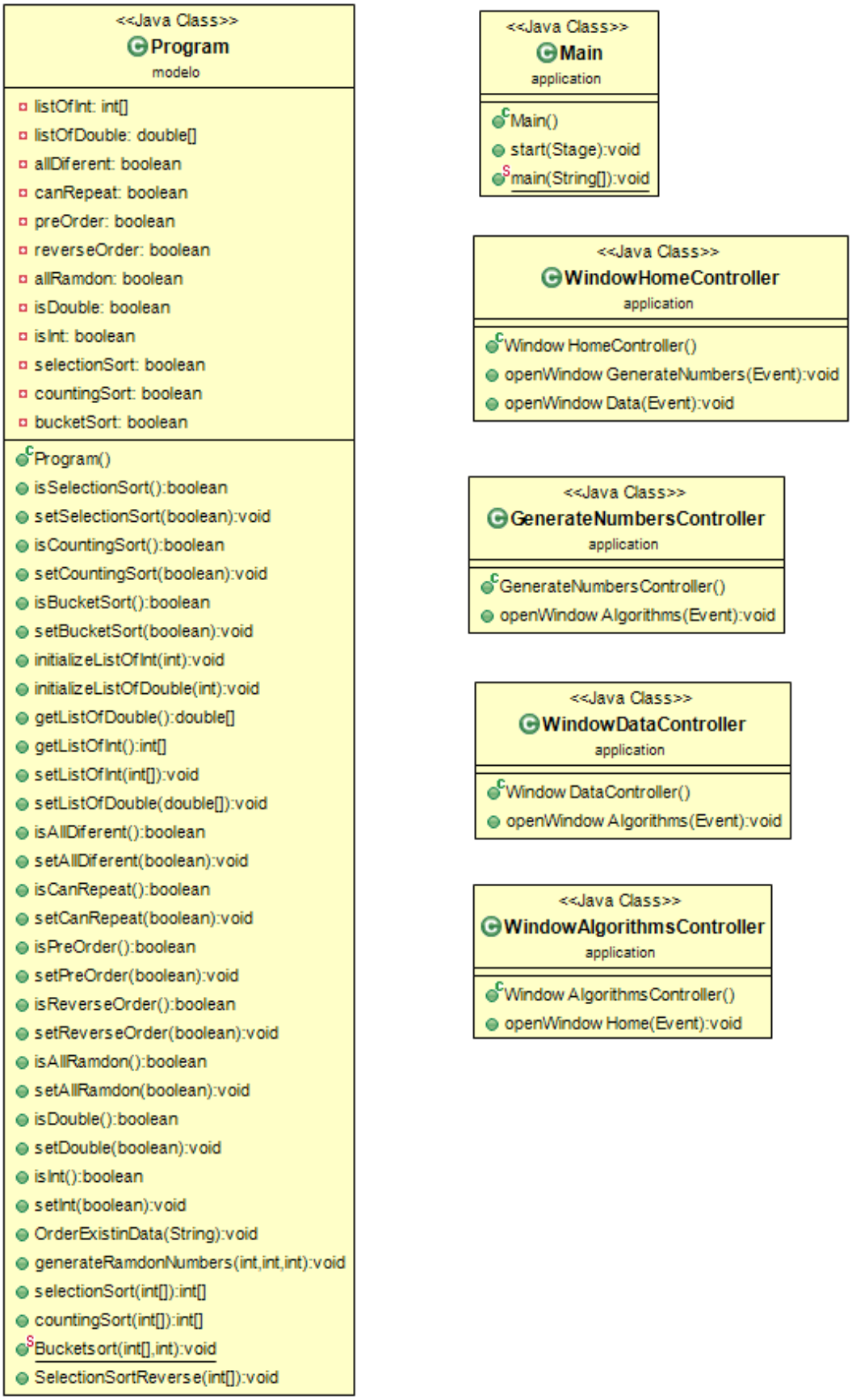
Algoritmo de ordenamiento	Criterio A	Criterio B	Criterio C	Criterio D	Criterio E	Criterio F	Total
countingSort	2	1	4	3	3	3	16
BucketSort	2	1	4	3	3	3	16
SelectionSort	1	3	2	1	3	1	11
InsertionSort	2	3	2	1	0	1	9
BubbleSort	2	3	2	1	0	1	9

Fase 6: Preparación de informes y especificaciones y Fase 7: Implementación

Con lo que con concluimos con este respectivo a este estudio y su respectivo análisis, buscando los mejores algoritmos para ser implementados en el programa especificado y así satisfacer los requerimientos antes planeados, se han seleccionado los algoritmos Counting Sort, Bucket Sort y Selection Sort, estos por varios criterios anteriormente especificados, es por lo anterior que ahora en adelante podemos comenzar a implementar todo lo aquí expresado, siguiente las pautas necesarias y coherentes para un proceso óptimo.

Diagrama de clase:

Diseño del diagrama de clases de la solución.


<<Java Class>>
Program
modelo
listOfInt: int[]
listOfDouble: double[]
allDiferent: boolean
canRepeat: boolean
preOrder: boolean
reverseOrder: boolean
allRamdon: boolean
isDouble: boolean
isInt: boolean
selectionSort: boolean
countingSort: boolean
bucketSort: boolean
Program()
isSelectionSort():boolean
setSelectionSort(boolean):void
isCountingSort():boolean
setCountingSort(boolean):void
isBucketSort():boolean
setBucketSort(boolean):void
initializeListOfInt(int):void
initializeListOfDouble(int):void
getListOfDouble():double[]
getListOfInt():int[]
setListOfInt(int[]):void
setListOfDouble(double[]):void
isAllDiferent():boolean
setAllDiferent(boolean):void
isCanRepeat():boolean
setCanRepeat(boolean):void
isPreOrder():boolean
setPreOrder(boolean):void
isReverseOrder():boolean
setReverseOrder(boolean):void
isAllRamdon():boolean
setAllRamdon(boolean):void
isDouble():boolean
setDouble(boolean):void
isInt():boolean
setInt(boolean):void
OrderExistinData(String):void
generateRamdonNumbers(int,int,int):void
selectionSort(int[]):int[]
countingSort(int[]):int[]
Bucketsort(int[],int):void
SelectionSortReverse(int[]):void
<<Java Class>>
Main
application
Main()
start(Stage):void
main(String[]):void
<<Java Class>>
WindowHomeController
application
Window HomeController()
openWindow GenerateNumbers(Event):void
openWindow Data(Event):void
<<Java Class>>
GenerateNumbersController
application
GenerateNumbersController()
openWindow Algorithms(Event):void
<<Java Class>>
WindowDataController
application
Window DataController()
openWindow Algorithms(Event):void
<<Java Class>>
WindowAlgorithmsController
application
Window AlgorithmsController()
openWindow Home(Event):void