Análisis de complejidad y rendimiento algorítmico de estructuras programáticas orientadas a la optimización del data mining

Complexity and algorithmic performance analysis of programmatic structures aimed at data mining optimization

David Galarza G.[1] Carlos Tamayo-Ruiz [1,2], Bryan Ortiz [1], Cristian Sánchez[1].

[1] Instituto Tecnológico Superior Quito Metropolitano. Carán N3-195 y Calle B (Nueva Tola 2) Quito, Ecuador., dgalarza@itsqmet.edu.ec, ctamayo@itsqmet.edu.ec, bortizs@itsqmet.edu.ec, cisanchez@itsqmet.edu.ec

[2] Universidad Politécnica Salesiana del Ecuador, C. Vieja 12-30 y, Cuenca 010105, ctamayo@est.ups.edu.ec

ÉLITE 2020, VOL. (2). NÚM. (2)
ISSN: 2600-5875

Recibido: 26/05/2020
Revisado: 08/07/2020
Aceptado: 09/08/2020
Publicado: 10/09/2020

RESUMEN:

El presente articulo tiene el objetivo de realizar un análisis de complejidad y rendimiento a tres algoritmos de búsqueda: Burbuja, Burbuja mejorada y un algoritmo propuesto. Las variables medibles que servirán de indicadores serán:

- Para complejidad algorítmica: número de líneas de código, instancia y declaración de variables, implementación de estructuras de control y varianzas de declaración de variables dentro de las comparaciones en estructuras de control
- Para rendimiento algorítmico: tiempo que tarda el método programático en ordenar un arreglo de posiciones similares, utilizando el mismo compilador, así como el mismo IDE.

Los resultados que entregue el análisis de complejidad y rendimiento algorítmico servirán para esquematizar la aplicabilidad y usabilidad de los algoritmos en situaciones puntuales con el fin de asegurar la transacción eficiente de la información a niveles arquitectónicos como en capas de controladores y secciones hibridas.

Palabras clave: rendimiento, funcionalidad, algoritmo, ordenamiento, búsqueda.

ABSTRACT:

This article aims to perform a complexity and performance analysis on three search algorithms: Bubble, Enhanced Bubble, and a proposed algorithm. The measurable variables that will serve as indicators will be:

- For algorithmic complexity: number of lines of code, instance and declaration of variables, implementation of control structures and variance of declaration of variables within comparisons in control structures
- For algorithmic performance: time it takes for the programmatic method to sort an array of similar positions, using the same compiler, as well as the same IDE.

The results delivered by the algorithmic performance and complexity analysis will serve to outline the applicability and usability of the algorithms in specific situations to ensure the efficient transaction of information at architectural levels such as controller layers and hybrid sections.

Keywords: performance, functionality, algorithm, ordering, search.

INTRODUCCIÓN:

Un algoritmo está estructuralmente diseñado para cumplir con una función en específico, teniendo entras, procesamiento y salidas como elementos necesarios y fundamentales para su ejecución. Para el presente estudio se procederá a evaluar el rendimiento y complejidad de tres algoritmos de ordenamiento, los cuales son:

- Burbuja
- Burbuja mejorada
- Algoritmo propuesto

Para los algoritmos mencionados se implementará el seudocódigo genérico y base que es de conocimiento de la comunidad de desarrolladores, por lo que, únicamente ordenara números enteros.

Este estudio de rendimiento y complejidad algorítmica se lo realizará a cada algoritmo versus los restantes. Se utilizará el mismo IDE de desarrollo y el mismo compilador para garantizar la transparencia del estudio.

MARCO CONCEPTUAL:

Conceptos:

Arrancamos esta sección de conceptos definiendo los algoritmos que están presentes en este trabajo de investigación, los cuales son:

Ordenación

Es un procedimiento donde el algoritmo busca recopilar los datos y procede a ordenarlos siguiendo un patrón específico -indicado al inicio-.

La ordenación, también conocida como clasificación, es el proceso de organizar un conjunto de datos en algún orden y/o secuencia específica, tal como sucede en el creciente- decreciente para datos numéricos o el orden alfabético para datos

numéricos o el orden alfabético para datos compuestos por caracteres. "Los algoritmos de ordenación permutan los elementos del conjunto de datos hasta conseguir dicho orden. Para ello se basan en dos operaciones básicas: la comparación y el intercambio" (Universidad CEU San Pablo, 2012, p. 15).

Búsqueda

Este procedimiento busca recorrer todo el arreglo, generalmente del número mayor y menor o el índice de un determinado elemento.

La búsqueda del elemento dentro de un arreglo es una de las operaciones más importantes en el procesamiento de la información, y a su vez, permite la recuperación de datos previamente almacenados. "El tipo de búsqueda se puede clasificar como interna o externa según el lugar en el que esté almacenada la información, tanto en memoria como dispositivos externos" (Díaz, 2006, párr. 2).

Antecedentes

La sociedad como tal tiene entre sus funciones diarias muchas actividades que involucran la manipulación de datos, sin que se esté consiente o no de que dicho procesamiento -de datos- es parte de un algoritmo, el mismo que busca ordenar de manera específica los requerimientos de información. Las empresas en su día a día manejan códigos con el objetivo de conseguir una entrega de información eficiente hacia el usuario final; las facturas de consumo diario se ordenan por fecha para la correspondiente verificación de venta, las bases de datos muchas veces se ordenan de forma alfabética con el fin de encontrar fácilmente la información deseada. Por estas y más razones, una de las tareas más comunes de un computador es la ordenación en el procesamiento de datos.

Los diferentes métodos tanto de ordenación como de búsqueda son desde un punto de vista teórico muy interesantes, pero para el patrón de vida diario es mucho más práctico. Para esto, a continuación, veremos los diferentes algoritmos que existen para el ordenamiento y la búsqueda, los mismos reciben como argumento "un arreglo", el cual pasa elementos y a su vez un entero, (este último representa la cantidad de elementos a buscar y ordenar).

Algoritmos de Ordenamiento

Existen diferentes algoritmos de ordenamiento elementales o básicos cuyos detalles de implementación se pueden encontrar en diferentes libros algorítmicos. Los algoritmos presentan diferencias entre ellos que los convierten en más o menos eficientes y prácticos según sea la rapidez y eficiencia demostrada por cada uno de ellos.

Los algoritmos básicos de ordenamiento más simples y clásicos son:

- **Ordenamiento por burbuja.**
- **Ordenamiento por inserción.**
- **Ordenamiento por selección.**

Los métodos de ordenamiento más recomendados son el de selección y el de inserción. Aunque se estudia el método de burbuja por ser el más sencillo al inicio de ciclo de formación para un programador, este es considerado como el más ineficiente; por esta causa no se recomienda su uso, pero si conocer su técnica.

Algoritmo de Burbuja

El algoritmo de burbuja es uno de los métodos de ordenación más conocidos y uno de los primeros que aprenden los programadores.

Su ejecución comprende en comparar pares de elementos adyacentes en un arreglo y si están desordenados intercambiarlos hasta que estén todos ordenados. Esto funciona al revisar cada elemento del arreglo con el siguiente, para esto es necesario revisar varias veces todo el arreglo hasta que no existan más intercambios o lo que es lo mismo que el arreglo ya esté ordenado.

El algoritmo de burbuja recibe su nombre por la forma en cómo van emergiendo los elementos del arreglo al realizar los intercambios, como si fueran pequeñas "burbujas".

Consideremos entonces:

Si A es el arreglo a ordenar, se realizan A.length-1 pasadas. Si la variable (i) es la que cuenta el número de pasadas, en cada pasada (i) se comprueban los elementos adyacentes desde el primero hasta A.length-i-1 ya que el resto hasta el final del "array" están ya ordenados. Si los elementos adyacentes están desordenados se intercambian.

El método de ordenación de la burbuja en Java es el siguiente:

```
1  static void burbuja(int arreglo[])
2      {
3          for(int i = 0; i < arreglo.length - 1; i++)
4          {
5              for(int j = 0; j < arreglo.length - 1; j++)
6              {
7                  if (arreglo[j] < arreglo[j + 1])
8                  {
9                      int tmp = arreglo[j+1];
10                     arreglo[j+1] = arreglo[j];
11                     arreglo[j] = tmp;
12                 }
13             }
14         }
15         for(int i = 0;i < arreglo.length; i++)
16         {
17             System.out.print(arreglo[i]+"\n");
18         }
19     }
```

Figura 1. Código de Ordenamiento Burbuja para Java.

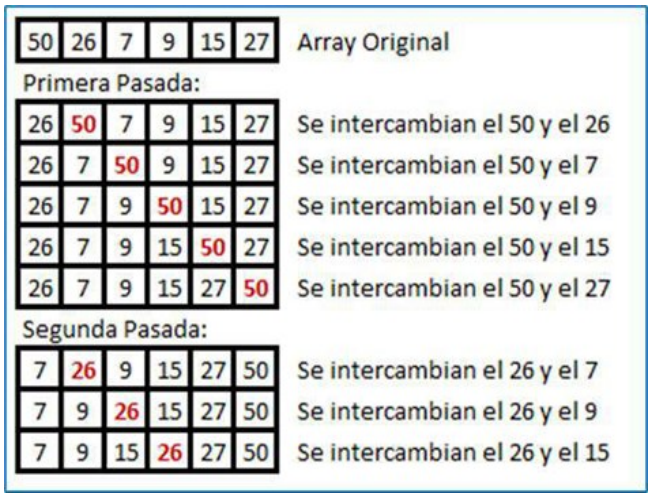

Figura 2. Ejemplo de Ejecución de Ordenamiento por Burbuja.

Algoritmo de Inserción

El algoritmo de ordenamiento por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético. Consiste en insertar un nombre en su posición correcta dentro de una lista que ya está ordenada. Requiere O(n^2) operaciones para ordenar una lista de n elementos.

Consideremos entonces:

Consiste en N − 1 pasadas. En las pasadas 2 a N se cumplirá que los elementos de las posiciones 1 a P

están ordenados. En la pasada P movemos el elemento P a su lugar correcto, este lugar es encontrado en las posiciones de los elementos 1 a P.

Los pasos para entenderlo mejor se resumen así:

1. El primer elemento a[0] se considera ordenado; es decir, la lista inicial consta de un elemento.
2. Se inserta a[1] en la posición correcta; delante o detrás de a[0], dependiendo de si es menor o mayor.
3. Por cada bucle o iteración i (desde i=1 hasta n-1) se explora la sublista a[i-1] a [0] buscando la posición correcta de inserción de a[i]; a la vez, se mueven hacia abajo (a la derecha en la sublista) desde una posición todos los elementos mayores que el elemento a insertar a[i], para dejar vacía esa posición.
4. Insertar el elemento a[i] a la posición correcta.

El método de ordenación de inserción en Java para ordenar un arreglo es el siguiente:

```
1  public static void insercionDirecta(int A[]){
2      int p, j;
3      int aux;
4      for (p = 1; p < A.length; p++){ // desde el segundo elemento hasta
5          aux = A[p];                 // el final, guardamos el elemento y
6          j = p - 1;                  // empezamos a comprobar con el anterior
7          while ((j >= 0) && (aux < A[j])){ // mientras queden posiciones y el
8              // valor de aux sea menor que los
9              A[j + 1] = A[j];    // de la izquierda, se desplaza a
10             j--;                // la derecha
11         }
12         A[j + 1] = aux;         // colocamos aux en su sitio
13     }
14 }
```

Figura 3. Código de Ordenamiento Inserción para Java.

Figura 4. Ejemplo de Ejecución de Ordenamiento por Inserción.

Algoritmo de Selección

El algoritmo de ordenamiento por selección mejora ligeramente al algoritmo -de ordenamiento- por burbuja, ya que este realiza muchas menos operaciones de intercambio, pero cuando se trata de ordenar un arreglo de datos más complejo, la operación de intercambiar es más costosa, en este caso, refiriéndonos a términos de rendimiento en el computador.

Consideremos entonces:

Ordenar un arreglo a[] de enteros en orden ascendente, es decir, si el -arreglo a[] tiene n elementos, se tratan de ordenar los valores del arreglo de modo que a[0] sea el valor más pequeño, el valor almacenado en a[1] sea el más pequeño y así hasta a[n-1] que ha de contener el elemento de mayor valor. El algoritmo se apoya en las pasadas que intercambian el elemento más pequeño, luego van en sucesión con el elemento de la lista a[], el cual ocupa la posición igual al orden de pasada (hay que considerar el índice 0). La pasada inicial busca el elemento más pequeño de la lista y se intercambia con a[0], considerado como el primer elemento de la lista.

Después de terminar esta primera pasada, el frente de la lista está ordenado y el resto de la lista a[1], a[2]...a[n-1] permanece desordenado. La siguiente pasada busca en esta lista desordenada y selecciona el elemento más pequeño, el cual se almacena en la posición a[1]. De este modo, los elementos a[0] y a[1] están ordenados y la sublista a[2], a[3]... a[n-1] desordenados; entonces, se selecciona el elemento más pequeño y se intercambia con a[2].

El proceso continúa hasta realizar n-1 pasadas, en ese momento la lista desordenada se reduce a un elemento (el mayor de la lista) y el "array" completo queda ordenado. Un ejemplo práctico ayudará a la comprensión del algoritmo. El método de ordenación por selección en

Java es el siguiente:

```
1  public static void seleccion (int A[]) {
2      int i, j, menor, pos, tmp;
3      for (i = 0; i < A.length - 1; i++) {      // tomamos como menor el primero
4          menor = A[i];                         // de los elementos que quedan por ordenar
5          pos = i;                              // y guardamos su posición
6          for (j = i + 1; j < A.length; j++){ // buscamos en el resto
7              if (A[j] < menor) {              // del array algún elemento
8                  menor = A[j];                // menor que el actual
9                  pos = j;
10             }
11         }
12         if (pos != i){                        // si hay alguno menor se intercambia
13             tmp = A[i];
14             A[i] = A[pos];
15             A[pos] = tmp;
16         }
17     }
18 }
```

Figura 5. Código de Ordenamiento Selección para Java.

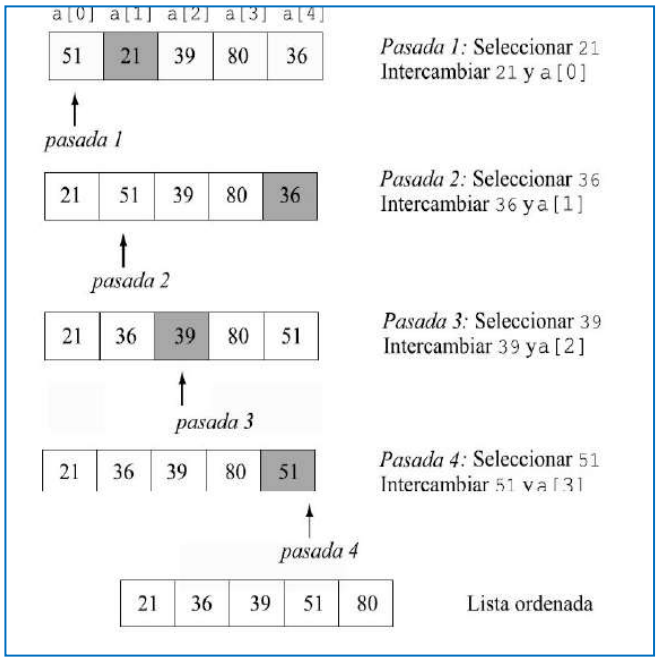

Figura 6. Ejemplo de Ejecución de Ordenamiento por Selección.

Algoritmos de Búsqueda

Con el avance de la tecnología y en sí de las computadoras junto con todo su poderoso proceso de cómputo, cada vez es más útil, sencillo y seguro realizar búsqueda de información a través de nuestras computadoras de escritorio, portátiles o incluso nuestros smartphones.

La búsqueda de un elemento dentro de una colección de datos es sin duda algo común en nuestras operaciones diarias, y tal como se menciona al inicio de la presente investigación, este es una de las operaciones más importantes en el procesamiento de información.

Los algoritmos de búsqueda que encontramos hoy en día son:

- **Búsqueda Lineal.**
- **Búsqueda Binaria.**
- **Búsqueda de Hash o de Índice.**

Los algoritmos de búsqueda tienen dos finalidades:

Algoritmos de Búsqueda

Con el avance de la tecnología y en sí de las computadoras junto con todo su poderoso proceso de cómputo, cada vez es más útil, sencillo y seguro realizar búsqueda de información a través de nuestras computadoras de escritorio, portátiles o incluso nuestros smartphones.

La búsqueda de un elemento dentro de una colección de datos es sin duda algo común en nuestras operaciones diarias, y tal como se menciona al inicio de la presente investigación, este es una de las operaciones más importantes en el procesamiento de información.

Los algoritmos de búsqueda que encontramos hoy en día son:

- **Búsqueda Lineal.**
- **Búsqueda Binaria.**
- **Búsqueda de Hash o de Índice.**

Los algoritmos de búsqueda tienen dos finalidades:

Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.

Si el elemento está en el conjunto, hallar la posición en la que se encuentra.

Búsqueda Lineal

Este es el método de búsqueda más lento, pero si la información se encuentra completamente desordenada es el único que podrá ayudar a encontrar el dato que se busca. Este algoritmo compara uno a uno los elementos del arreglo hasta recorrerlo por completo indicando si el número buscado existe. En este caso no se ordenará la lista de elementos a diferencia de otros algoritmos.

Su implementación es la siguiente:

```
for (int i = 0; i < texto_array.length; i++) {
    if(texto_array[i].equalsIgnoreCase(clave)){
        posicion = posicion + "[" + i + "]";
        texto_respuesta = texto_array[i];
    }

}
```

Figura 7. Algoritmo de Búsqueda Lineal.

Búsqueda Binaria (dicotómica)

Este algoritmo permite buscar de una manera más eficiente un dato dentro de un arreglo. Para hacer esto se determina el elemento central del arreglo y se compara con el valor que se está buscando; si coincide termina la búsqueda y en caso de no ser así se determina si el dato es mayor o menor que el elemento central, de esta forma se elimina una mitad del arreglo junto con el elemento central para repetir el proceso hasta encontrar o tener sólo un elemento en el arreglo.

En base a esto determinamos lo siguiente:

- Si el elemento buscado es menor que el elemento medio, entonces sabemos que el elemento está en la mitad inferior de la tabla.
- Si es mayor es porque el elemento está en la mitad superior.
- Si es igual se finaliza con éxito la búsqueda ya que se ha encontrado el elemento.

Se puede aplicar tanto a datos en listas lineales (Vectores, Matrices, etc.) como en árboles binarios de búsqueda. Los prerrequisitos principales para la búsqueda binaria son:

- La lista debe estar ordenada en un orden específico de acuerdo al valor de la llave.
- Debe conocerse el número de registros.

Su implementación es la siguiente:

```
int bajo = 0;
int alto = texto_array.length-1;
boolean buscado = false;
while (buscado == false && bajo <= alto) {
    int posicion_central = (bajo + alto)/2;
    int valorCentral = Integer.parseInt(texto_array[posicion_central]);

    if (valorCentral == clave){
        buscado = true;
        posicion_general = posicion_general + "[" + posicion_central + "]";
        }else if (valorCentral > clave) {
        alto = posicion_central -1;
        buscado = false;
        }else if (valorCentral < clave){
        bajo = posicion_central +1;
        buscado = false;
    }
}
texto_respuesta += "[" + clave + "]";
System.out.println ("Elemento Encontrado: " +texto_respuesta);
```

Figura 8. Algoritmo de Búsqueda Binaria.

Búsqueda de Hash o de Índice

registros "vistas", emplean un proceso de búsqueda que implica cierto tiempo y esfuerzo. El método de transformación de claves nos permite encontrar directamente el registro buscado en tablas o archivos que no se encuentran necesariamente ordenados -en un tiempo independiente de la cantidad de datos-.

Mediante cada elemento del arreglo índice se asocian grupos de elementos del arreglo inicial. Los elementos en el índice y en el arreglo deben estar ordenados.

El método consta de dos pasos: Primero, buscar en el arreglo índice el intervalo correspondiente al elemento buscado y restringir la búsqueda a los elementos del intervalo que se localizó previamente. La ventaja es que la búsqueda se realiza en el arreglo de índices y no en el arreglo de elementos. Cuando se ha encontrado el intervalo correcto se hace una segunda búsqueda en una parte reducida del arreglo. Estas dos búsquedas pueden ser secuenciales o binarias y el tiempo de ejecución dependerá del tipo de búsqueda utilizado en cada uno de los arreglos.

Algoritmo que genera un índice basado en un archivo de texto y que busca los registros que pide el usuario en él.

```
1   T: entero
2   fDT: archivo
3   a[100000],temp,i,j,n,k: entero
4   x: binario
5   c: caracter
6   Inicio
7   cargararchivo "datos.txt" en fDAT
8   i=0
9   mientras no fDAT.eof hacer
10  a[i+1]=leerarchivo(fDAT, posición 1+i*100)
11  T=i
12  i=i+1
13  fin mientras
14  c="s"
15
16  hacer
17    escribir "Número identidad del registro a desglosar: "
18    leer n
19    x=falso
20    i=0
21    hacer
22    i=i+1
23    si a[i] = n entonces
24    j=i
25    x=verdadero
26    fin si
27    mientras i<T && x=falso
28
29    si x=falso entonces
30    escribir "No se encontró el número"
31    si no
32     desde i=0 hasta 98 incremento 1 hacer
33        escribircaracter leerarchivo(fDAT, posición (1+i)+(100*j))
34     fin desde
35    fin si
```

Figura 9. Algoritmo de Búsqueda Indexada.

METODOLOGÍA:

Alcance del algoritmo propuesto

Dentro del criterio o performance algorítmicos, la propuesta funciona con n elementos dentro del arreglo que se desea ordenar entre números positivos, negativos -incluido el cero-; sin embargo, su alcance, en vista justamente de su proceso, busca ordenar la mitad del arreglo en mención, ya que este algoritmo divide en la mitad, desde un inicio, la longitud del arreglo para poder iniciar el ordenamiento.

Esta propuesta está pensada para arreglos que abarcan muchos números y/o posiciones, dentro de los cuales se busca filtrar el orden entre números mayores o menores sin la necesidad de ordenar todo el arreglo, sea esta por razones de tiempo o de recursos disponibles para tal efecto. La propuesta algorítmica puede ser aplicada para escenarios donde solo se necesiten filtrar los primeros 20 datos de un lote de números por -poner un ejemplo-.

Burbuja vs Burbuja mejorada.

Complejidad.

Este apartado de complejidad es básicamente el mismo, pues ambos algoritmos de ordenamiento se escriben en la misma cantidad de líneas, salvo que el método de ordenamiento de burbuja mejorada considera dentro del primer bucle “for” restar la variante del bucle “for” al que pertenece, esto se evidencia en la siguiente imagen:

```
1  for (int j = 0; j < (arreglo_arr.length); j++) {
2      for (int k = 0; k < arreglo_arr.length - j - 1; k++) {
3          if (tipoOrdenamiento.equalsIgnoreCase("Ascendente") && Integer.parseInt(arreglo_arr[k]) > Integer.parseInt(
           arreglo_arr[k + 1])) {
4              int temp = Integer.parseInt(arreglo_arr[k]);
5              arreglo_arr[k] = arreglo_arr[k + 1];
6              arreglo_arr[k + 1] = temp + "";
7              } else if (tipoOrdenamiento.equalsIgnoreCase("Descendente") && Integer.parseInt(arreglo_arr[k]) < Integer.
               parseInt(arreglo_arr[k + 1])) {
8              int temp = Integer.parseInt(arreglo_arr[k]);
9              arreglo_arr[k] = arreglo_arr[k + 1];
10             arreglo_arr[k + 1] = temp + "";
11         }
12
13     }
14 }
15
16
```

Figura 10. Algoritmo de Ordenamiento: Burbuja Mejorado.

Entendiendo el algoritmo se expone lo siguiente:

El ordenamiento burbuja consiste en un algoritmo que permite ordenar de manera ascendente o descendente una serie de números, de tal manera que realiza múltiples pasadas comparando los ítems y va intercambiando los que no están en orden.

Estos ítems pueden ser almacenados en un arreglo; en este caso, burbuja iniciará en la posición 0 e irá incrementando hasta llegar a la longitud total del arreglo. Lo que realizará en su proceso es comparar el ítem en la posición n con n+1 siendo n=0 e ir ordenando de acuerdo con lo que se especifique, así este algoritmo compara los ítems seleccionados y lo que hace es intercambiar esos números en su posición según la especificación que se le haya emitido, sea de mayor a menor o viceversa.

Si hay n ítems en la lista, entonces hay n−1 parejas de ítems que deben compararse en la primera pasada. Al comienzo de la segunda pasada el valor más grande ya está en su lugar, aquí quedan n−1 ítems por ordenar, lo que significa que habrá n−2 parejas.

RENDIMIENTO:

En cuanto al rendimiento “burbuja”, se estima un tiempo de 0.30 milisegundos para ejecutar y resolver el arreglo. A continuación, se lo representa en el siguiente vector -como ejemplo-:

array_ejemplo = [1,7,1,6,1,3,6,6,1,7]

Burbuja --> arreglo de 10 posiciones, corre 90 veces hasta terminar de ordenar el vector, mientras que el algoritmo propuesto corre 35

Burbuja mejorada vs Algoritmo propuesto.

Complejidad.

Según las consideraciones realizadas, la complejidad del algoritmo propuesto se basa íntegramente en el algoritmo de “burbuja mejorada”; lo que nos indica que en teoría el proceso que realiza el algoritmo es similar al de ordenamiento, considerando que al inicio del algoritmo propuesto en el primer “for”, existe la división de la longitud del arreglo para luego continuar con el proceso de ordenamiento, sea este en forma descendente o ascendente, según se requiera.

La hipótesis inicial analizada en este algoritmo propuesto señala que el vector que entregue el usuario lo va a ordenar. Hay que considerar que la mitad de este va a estar ordenada de forma íntegra y la otra mitad va a depender de si han estado o no en orden las diferentes posiciones del arreglo original, para que de forma gráfica se evidencie el ordenamiento parcial de esta segunda mitad en el arreglo final (ordenado). Todo este proceso mediante el algoritmo propuesto.

Rendimiento.

Este apartado nos muestra según las pruebas efectuadas y que se verá evidenciado más adelante en el apartado de los resultados, cuya comparación con el algoritmo “burbuja mejorada” nos indica que este tiene un porcentaje del 22.22 -más rápido- en ordenar el vector dado.

Tomando en cuenta que la propuesta algorítmica que estamos efectuando en este estudio

contempla el ordenar únicamente la mitad del arreglo solicitado por el usuario final.

Tomando el ejemplo del vector de pruebas:

array_ejemplo = [1,7,1,6,1,3,6,6,1,7]

Burbuja mejorada --> arreglo de 10 posiciones, corre 45 veces hasta ordenar el vector, mientras que el algoritmo propuesto corre 35 veces en arreglar el mismo vector. Cabe indicar que según las pruebas que serán evidenciadas en el apartado de los resultados, habrá arreglos que se ordenen de forma íntegra todas sus posiciones. Esto, como se ha mencionado anteriormente, va a depender estrictamente de que tan ordenadas hayan estado dichas posiciones en el arreglo original antes del proceso de ordenamiento algorítmico, ya que de forma gráfica se verán que hay arreglos que no se ordenan en su totalidad. Esto, justamente por el hecho de que esta propuesta algorítmica busca ordenar –únicamente- el 100% de la mitad del arreglo solicitado.

DISCUSIÓN DE RESULTADOS:

Los resultados obtenidos después de un análisis orientado al rendimiento y complejidad algorítmica entre los algoritmos de burbuja, burbuja mejorada y el algoritmo propuesto se presentan a continuación:

Algoritmo	Complejidad
Burbuja	Líneas de código predefinidas.
Burbuja mejorada	Mismas líneas de código. Ajuste de pasadas en n/2 respecto a Burbuja
Algoritmo propuesto	Mismas líneas de código. Ajuste de pasadas en n/2 y de comparaciones en n/2 respecto a Burbuja.

Tabla 5: Complejidad vs algoritmos. (Elaboración propia)

De acuerdo con lo expuesto en la tabla anterior, se demuestra que burbuja mantiene una complejidad algorítmica adecuada referente a burbuja mejorada, así como también al algoritmo propuesto.

Ahora, se presenta la tabla de resultados resumida para el análisis de los algoritmos versus el rendimiento.

Algoritmo	Rendimiento
Burbuja	• En un arreglo de 10 posiciones, **corre 90 veces** antes de terminar de ordenar • Se estima un **0.30 seg**. en ejecutar el algoritmo y ordenar el arreglo
Burbuja mejorada	• En un arreglo de 10 posiciones, **corre 45 veces** hasta terminar de ordenar el mismo vector. • Se estima un **0.21 seg**. en ejecutar el algoritmo y ordenar el arreglo
Algoritmo propuesto	• Corre **35 veces** en arreglar el mismo vector. • Se estima un **%61.11** más rápido que burbuja mejorada para ejecutar el algoritmo y ordenar el arreglo

Tabla 6: Rendimiento vs algoritmos. (Elaboración propia)

De acuerdo con lo presentado en la tabla anterior, el algoritmo propuesto tiene un rendimiento superior a los algoritmos de ordenamiento: burbuja y burbuja mejorada.

CONCLUSIONES:

La creación, ejecución y estudio de los algoritmos descritos, a nivel de complejidad y rendimiento, dio como resultado las siguientes conclusiones:

Si bien el algoritmo propuesto tiene una gran ventaja en rendimiento, únicamente funciona al 100% con una cantidad controlada de elementos en el arreglo que será ordenado, si es que esa cantidad sobrepasa del control, el ordenamiento utilizando este algoritmo seria erróneo.

La complejidad algorítmica no genera una ventaja significativa desde su interpretación semántica, sin embargo, esta complejidad se ve reflejada en el rendimiento.

Se puede concluir que la complejidad algorítmica sirve de insumo positivo para que el rendimiento sea mucho más eficiente.

El rendimiento de la ejecución de los algoritmos de ordenamiento será más significativo con arreglos de tamaños mucho mas grandes o, estos mismos ejecutados en arreglos denominados multidimensionales.

REFERENCIAS BIBLIOGRAFÍCAS:

- Baase, S., & Gelder, A. (2000). "Computer algorithms." Addison Wesley Longman.
- CEU San Pablo, U. (2012). "Algoritmos de ordenación y búsqueda". CEU Universidad San Pablo. Retrieved from http://biolab.uspceu.com/aotero/ recursos/doc encia/TEMA%208.pdf
- Cormen, T., Leiserson, C., Rivest, R., & Stein, C.(2007). "*Introduction to algorithms.*" MITPress.
- De Giusti, A. (2001). "*Algoritmos, datos y programas.*" Pearson Educación, S.A
- Díaz, N. (2006). "*BÚSQUEDA DE VECTORES*".
- FIET - UNICAUCA. Retrieved from http://artemisa.unicauca.edu.co/~nediaz/ ED DI/cap02.htm
- Joyanes Aguilar, L. (2003). "Programación en C++." McGraw-Hill Interamericana de España.
- Knuth, D. (2014). "The art of computer programming." Addison-Wesley.
- Lage, F., Cataldi, Z., & Salgueiro, F. (2008). "Fundamentos de algoritmos y programación." Nueva librería.
- Pariser, E. (2017). "El filtro burbuja". Taurus.
- Pharr, M., Wenzel, J., & Humphreys, G. (2017).
- "Physically based rendering." Morgan Kaufmann.
- Weiss, M. (1995). "Estructuras de datos y algoritmos." Addison-Wesley Iberoamericana.