ALGORITMO DE ORDENAMIENTO POR COMPARACIONES HEAPINSERT SORT

The Algorithm Heapinsert Sort

RESUMEN
Se propone una variante del algoritmo de *ordenamiento por inserción*, al cual se le adiciona la función de *construir montón* del algoritmo de *ordenamiento por montones*. Es estratégico primero construir un montón sobre el arreglo que se va a ordenar porque la información del arreglo "tiende" a quedar pseudo ordenada de forma descendente, luego al invertir el arreglo la información "tiende" a quedar pseudo ordenada de forma ascendente, economizando trabajo del algoritmo de ordenamiento por inserción. Las pruebas realizadas sobre la variante del algoritmo muestran una economía de más del 50% del tiempo consumido por el algoritmo original.

PALABRAS CLAVES: Algoritmos de ordenamiento, Ordenamiento por montones, Ordenamiento por Inserción.

ABSTRACT
It proposes a variant of insertion sort algorithm, it's added the function to build a heap from the heap sort algorithm. It's strategic, first, build a heap on the array that is going to sort because the array's information "tends" to be pseudo descending order, then inverting the array's information "tends" to be pseudo ascending order, saving work of the insertion algorithm. The tests show a saving of over 50% of the time consumed by the original algoritm.

KEYWORDS: *Sorting algorihtms, Heap Sort, Insertion Sort.*

HUGO HUMBERTO MORALES PEÑA
Ingeniero de Sistemas.
Profesor Auxiliar
Programa Ingeniería de Sistemas y Computación.
Universidad Tecnológica de Pereira
huhumor@utp.edu.co

ANGEL AUGUSTO AGUDELO
Ingeniero Electricista, Esp.
Profesor Auxiliar
Programa Ingeniería de Sistemas y Computación.
Universidad Tecnológica de Pereira
a3udeloz@utp.edu.co

JORGE IVAN RIOS PATIÑO
Ingeniero Industrial, Msc.
Profesor Titular
Programa Ingeniería de Sistemas y Computación.
Universidad Tecnológica de Pereira
jirios@utp.edu.co

1. INTRODUCCIÓN

Existen diferentes algoritmos de ordenamiento, los cuales se clasifican en algoritmos de ordenamiento por comparaciones y en algoritmos de ordenamiento en tiempo lineal. En el primer grupo, los más conocidos son: Bubble Sort, Insertion Sort, Merge Sort, Heap Sort y Quick Sort, mientras que en el segundo grupo están, Counting Sort, Bucket Sort y Radix Sort.

El algoritmo de ordenamiento por comparaciones Insertion Sort es uno de los más costosos computacionalmente, teniendo en el caso promedio y peor caso una complejidad de $\theta(n^2)$, sin embargo, a partir de este, se han generado nuevas variantes de algoritmos de ordenamiento, tales como *Binary Insertion Sort* y el *Shell Sort*.

En este artículo se propone una nueva variante del algoritmo de ordenamiento por comparaciones Insertion Sort al cruzarlo con la función construir montón del algoritmo Heap Sort. El nuevo algoritmo consiste en lo siguiente: primero se genera un *montón máximo* con la información del arreglo, luego se invierte la información que se encuentra almacenada en el arreglo (de esta forma la información que esta en la posición uno queda en la última posición y la información de la última posición queda almacenada en la primera, y así sucesivamente para el resto de posiciones del arreglo) y por último se hace el llamado al algoritmo Insertion Sort.

¿Qué fue lo que motivo la propuesta de esta variante de algoritmo de ordenamiento?, es sencillo, el algoritmo Insertion Sort, en el mejor de los casos, es de tiempo lineal ($\theta(n)$), la función para construir el montón es de tiempo lineal y el invertir la información del arreglo es de tiempo lineal, adicionalmente, después de construir un *montón máximo* la información en el arreglo "tiende" a quedar pseudo ordenada de forma descendente, esto se presenta porque la información que se encuentra almacenada en la raíz de todo subárbol, es mayor o igual al resto de información que se encuentra almacenado en el subárbol, luego al invertir el arreglo este "tiende" a quedar pseudo ordenado de forma ascendente; de esta forma el Insertion Sort tendría que realizar mucho menos trabajo gracias a recibir un arreglo pseudo ordenado de forma ascendente.

Fecha Recepción: 9 de Septiembre de 2010
Fecha aceptación: 15 de Noviembre de 2010

El contenido del artículo ha sido distribuido en cinco partes. En la sección 2 se presentan los algoritmos de ordenamiento por comparaciones Insertion Sort y Heap Sort. En la sección 3 se presenta la propuesta del algoritmo de ordenamiento Heapinsert Sort, el cual es una variante que se obtiene al combinar los montones con el ordenamiento por inserción. En la sección 4 se hace un análisis de los resultados obtenidos al comparar los tiempos de ejecución de los algoritmos Insertion Sort y Heapinsert Sort. En la sección 5 se presentan las conclusiones del trabajo.

2. ALGORITMOS DE ORDENAMIENTO

Toda esta sección ha sido reescrita a partir de [1], en donde se presentarán únicamente los algoritmos de ordenamiento por comparaciones Insertion Sort (ordenamiento por inserción) y Heap Sort (ordenamiento por montones) los cuales son tradicionalmente referenciados en los programas de Ingeniería de Sistemas en Colombia. Estos algoritmos son el punto de partida para la variante de algoritmo de ordenamiento que se propondrá en la sección 3.

2.1. El Algoritmo de Ordenamiento por Inserción

El algoritmo Insertion Sort es un algoritmo eficiente para ordenar una cantidad pequeña de elementos. El algoritmo Insertion Sort trabaja de la forma como muchas personas ordenan una mano de cartas de póker, donde se comienza teniendo en la mano izquierda la primera de las cartas y el resto de éstas boca abajo sobre la mesa, luego, se toma una a una las cartas de la mesa y se inserta en la posición correcta en las cartas que ya están ordenadas en la mano izquierda. Para encontrar la posición correcta de una carta, esta se compara con cada una de las cartas que ya están ordenadas en la mano izquierda, de derecha a izquierda, la siguiente figura (tomada de [1]) ilustra la situación planteada:

Figura 1. Ordenando una mano de cartas con Insertion Sort.

El pseudo código del algoritmo Insertion Sort es el siguiente:

InsertionSort(A)
1. for $j \leftarrow 2$ to $length[A]$
2. $key \leftarrow A[j]$
3. $i \leftarrow j - 1$
4. while $i > 0$ and $A[i] > key$
5. $A[i+1] \leftarrow A[i]$
6. $i \leftarrow i - 1$
7. $A[i+1] \leftarrow key$

La complejidad del Insertion Sort en el *mejor de los casos*, caso que se presenta cuando recibe un arreglo ordenado de forma ascendente es θ (n). Esto se presenta porque el algoritmo nunca ejecuta, en este caso, el ciclo *while* de las líneas 4-6 porque nunca se cumple la condición de éste ciclo de repetición, de esta forma únicamente se recorren las n posiciones del arreglo para confirmar que la información de cada casilla del arreglo está en la posición donde tiene que quedar.

La complejidad del Insertion Sort en el *peor de los casos*, caso que se presenta cuando recibe un arreglo ordenado de forma descendente es $\theta(n^2)$. Esto se presenta porque el algoritmo ejecuta el ciclo *while* para desplazar a la derecha todos los elementos que se encuentran en posiciones inferiores a la posición que se está analizando, de ésta forma el ciclo *while* se ejecuta una vez para el elemento de la posición dos, se ejecuta dos veces para el elemento de la posición tres, se ejecuta tres veces para el elemento de la posición cuatro, y así sucesivamente hasta llegar a la posición n donde el ciclo se tiene que ejecutar $n-1$ veces. De ésta forma el total de veces que se ejecuta el ciclo *while* es:

$$1+2+3+\ldots+n-1=((n-1).n)/2=(n^2-n)/2=\theta(n^2) \qquad (1)$$

La complejidad del Insertion Sort en el *caso promedio* es la misma que se presenta en el Insertion Sort en el peor de los casos, donde se tiene que el ciclo *while* se ejecuta un orden de $\theta(n^2)$ veces.

2.2. El Algoritmo de Ordenamiento por Montones

El algoritmo de *ordenamiento por montones* presentado en [2], creado por J. W. J. Williams en 1964 hace uso de la estructura de datos montón para almacenar los números a ordenar. La estructura de datos montón, también es utilizada para implementar de forma eficiente colas de prioridades. A continuación se presentan las propiedades de los montones, la función que garantiza la propiedad de montón, la función para construir un montón y por último el algoritmo de ordenamiento por montones.

2.2.1. Los Montones y sus Propiedades

La estructura de datos montón, es un arreglo de objetos muy parecido a un árbol binario completo. Cada nodo del árbol corresponde a un elemento del arreglo que almacena el valor en el nodo. El árbol esta completamente lleno en todos los niveles excepto posiblemente en el nivel más bajo, el cual es llenado de

izquierda a derecha hasta algún punto. La siguiente figura (tomada de [3]) representa la situación anteriormente planteada con respecto a la forma que toma un árbol binario que representa un montón.

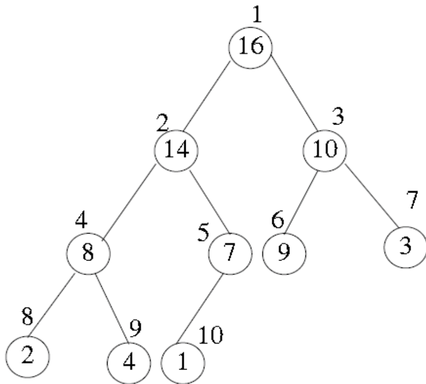

Figura 2. Ejemplo montón.

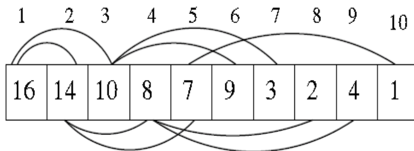

Figura 3. Ejemplo almacenamiento montón en arreglo.

Un arreglo *A* que representa un montón es un objeto con dos atributos: *length*[*A*], el cual es el número de elementos en el arreglo, y *heapSize*[*A*], el cual es el número de elementos del montón almacenados en el arreglo *A*, donde $heapSize[A] \leq length[A]$.
El contenido de la raíz del árbol esta almacenado en $A[1]$, y dado el índice de un nodo, el índice de su padre *Padre*(*i*), el índice del hijo por la izquierda *Izq*(*i*) y el índice del hijo por la derecha *Der*(*i*) pueden ser calculados simplemente con las siguientes funciones:

Padre(*i*) *return* $\lfloor i/2 \rfloor$
Izq(*i*) *return* $2 \cdot i$

Der(*i*) *return* $2 \cdot i + 1$

Hay dos clases de montones binarios: el *montón máximo* y el *montón mínimo*. En ambos casos, los valores de los nodos satisfacen una propiedad de montón, en el caso de un *montón máximo*, la propiedad del montón máximo es que para todo nodo *i* diferente de la raíz se cumple que $A[Padre(i)] \geq A[i]$, esto es que el valor de cada nodo como máximo es el valor de su padre, de esta forma el elemento más grande de un montón máximo está almacenado en la raíz. En el caso de un *montón mínimo*, la propiedad del montón mínimo es que para todo nodo *i* diferente de la raíz se cumple que $A[Padre(i)] \leq A[i]$, esto es que el valor de cada nodo como mínimo es el valor de su padre, de esta forma el elemento más pequeño de un montón mínimo está almacenado en la raíz.

Al mirar un montón como un árbol, se define la *altura* de un nodo en el montón como la cantidad de aristas que hay desde el nodo a alguna de las hojas más lejanas que se alcanzan desde él en una ruta simple descendente, de esta forma la altura del montón es la altura del nodo raíz del árbol. La altura de un montón es $\theta(lg\ n)$.

2.2.2. Garantizar la Propiedad de Montón

Los parámetros de entrada de la función *HeapifyMax* son un arreglo *A* y un subíndice *i* sobre el arreglo. La función *HeapifyMax* asume que tanto el subárbol izquierdo como el subárbol derecho a partir de la posición *i* son montones máximos, pero que *A*[*i*] puede ser más pequeño que alguno de sus dos hijos, con lo que se violaría la propiedad de montón máximo.
La función *HeapifyMax* le permite al valor almacenado en *A*[*i*] "flotar hacia abajo" en el montón máximo hasta lograr que el subárbol con raíz *i* sea un montón máximo.

```
HeapifyMax(A , i)
1. izq ← Izq(i)
2. der ← Der(i)
3. if izq ≤ heapSize[A] and A[izq] > A[i]
4.        then posMax ← izq
5.        else posMax ← i
6. if der ≤ heapSize[A] and A[der] > A[posMax]
7.        then posMax ← der
8. if posMax ≠ i
9.        then A[i] ↔ A[posMax]
10.       HeapifyMax(A , posMax)
```

Si *A*[*i*] es mayor o igual que la información almacenada en la raíz de los subárboles izquierdo y derecho entonces el árbol con raíz en el nodo *i* es un montón máximo y la función termina. De lo contrario, la raíz de alguno de los subárboles tiene información mayor que la que se encuentra en *A*[*i*] y es intercambiada con ésta, con lo cual se garantiza que el nodo *i* y sus hijos cumplen la propiedad de máximo montón, pero, sin embargo el subárbol hijo con el cual se intercambio la información de *A*[*i*] ahora puede no cumplir la propiedad de máximo montón, por lo tanto, se debe llamar de forma recursiva a la función *HeapifyMax* sobre el subárbol hijo con el cual se hizo el intercambio.
La complejidad en el peor de los casos de la función *HeapifyMax* es $O(h)$, donde *h* es la altura del montón, como la altura del montón es $\theta(lg\ n)$ entonces *HeapifyMax* es $O(lg\ n)$.

2.2.3. Construcción de un Montón

Se puede utilizar la función *HeapifyMax* de la manera abajo-arriba, para convertir un arreglo *A* de *n* elementos en un montón máximo. Los elementos del arreglo almacenados en las posiciones $\lfloor n/2 \rfloor+1$, $\lfloor n/2 \rfloor+2, \ldots, n$ son todos hojas del árbol, donde cada hoja es un montón de un solo elemento donde se cumple la propiedad de máximo montón.

BuildHeapMax(*A*)
1. *heapSize*[*A*] ← *length*[*A*]
2. for $i \leftarrow \lfloor \text{length[A]/2} \rfloor$ down to 1
3. do *HeapifyMax*(*A, i*)

En un *análisis sencillo* el peor de los casos de la función *BuildHeapMax* es *O*(*n lg n*), el cual se obtiene por los *O*(*n*) llamados que se realizan de la función *HeapifyMax* que tiene complejidad *O*(*lg n*), donde *O*(*n*) · *O*(*lg n*) = *O*(*n lg n*).
En un *análisis más exacto* de la función si el montón tiene *n* nodos la altura del árbol binario es $h = \lfloor \lg n \rfloor$, como el costo computacional de la función *HeapifyMax* depende es de la altura del nodo sobre el que se esté trabajando, por este motivo en la *Tabla 1* se presentan la cantidad de nodos con respecto a su altura en el árbol.

Altura del nodo	Cantidad de nodos
h	1
h-1	2
h-2	2^2
⋮	⋮
h – (h - 1) = 1	2^{h-1}
0	$\leq 2^h$

Tabla 1. Cantidad de nodos respecto a la altura en el árbol.

Adicionalmente se tiene la fórmula $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ para calcular la cantidad máxima de nodos que se encuentra a una altura *h* en un montón que tiene *n* elementos. Teniendo en cuenta lo anterior, entonces, el análisis de la complejidad del peor de los casos de la función *BuildHeapMax* se puede realizar de la siguiente forma:

$$\sum_{h=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \cdot \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \cdot \sum_{h=1}^{\infty} \frac{h}{2^h} \right) \quad (2)$$

se tiene la siguiente solución para la sumatoria:

$$\sum_{h=1}^{m} \frac{h}{2^h} = 2 - \frac{m+2}{2^m}, m \in Z^+ \quad (3)$$

de donde se obtiene que

$$\sum_{h=1}^{\infty} \frac{h}{2^h} = 2 \quad (4)$$

por lo tanto la complejidad de *BuildHeapMax* en el peor de los casos es:

$$O\left(n \cdot \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \cdot \sum_{h=1}^{\infty} \frac{h}{2^h} \right) = O(n \cdot 2) = O(n) \quad (5)$$

3. EL ALGORITMO DE ORDENAMIENTO HEAPINSERT SORT

El siguiente algoritmo es una variante que se obtiene al combinar la función de *construir montón máximo* (*BuilHeapMax*) del algoritmo de ordenamiento *HeapSort* y el algoritmo *InsertionSort*. En esta variante se busca que el arreglo *A* que es la entrada del algoritmo *InsertionSort* este *pseudo ordenado* de forma ascendente, para que de esta forma el *InsertionSort* no tenga que realizar tanto trabajo. El algoritmo *Heapinsert Sort* se presenta a continuación:

HeapinsertSort(*A*)
1. *BuildHeapMax*(*A*)
2. for $i \leftarrow 1$ to $\left\lfloor \frac{lenght[A]}{2} \right\rfloor$
3. do $A[i] \leftrightarrow A[length[A] - i + 1]$
4. *InsertionSort*(*A*)

3.1. Complejidad Espacial del Heapinsert Sort

El algoritmo *Heapinsert Sort* tiene una complejidad espacial de $\theta(n)$, complejidad lineal en espacio con respecto a la cantidad de elementos a ordenar, esta complejidad se obtiene gracias a que el algoritmo ordena sobre las mismas posiciones del arreglo, es decir *ordena en el lugar* y no necesita estar creando copias de fragmentos del arreglo para poder ordenar. Los algoritmos *Insertion Sort* y *Heap Sort* tienen la misma complejidad en espacio que el *Heapinsert Sort*.

3.2. Complejidad Temporal del Heapinsert Sort

Para el análisis de la complejidad en tiempo de ejecución del *Heapinsert Sort* se considera de forma independiente el análisis para el mejor de los casos, el análisis para el peor de los casos y el análisis para el caso promedio.

3.2.1. Complejidad Temporal del Heapinsert Sort en el Mejor de los Casos

El mejor de los casos para el algoritmo *Heapinsert Sort* es cuando éste recibe un arreglo ordenado de forma descendente, donde la función construir montón de la línea 1 no hace ningún cambio entre los elementos del arreglo porque éste ya cumple la propiedad de montón máximo, luego el ciclo de repetición *for* de las líneas 2-3 invierte el orden como es almacenada la información en el arreglo, de esta forma el arreglo queda ya ordenado de forma ascendente, por último, el arreglo es enviado a ordenar en la línea 4 por el algoritmo *Insertion Sort* pero como ya está ordenado entonces se presenta el mejor de los casos del *Insertion Sort* teniendo que hacer únicamente *O*(*n*) operaciones para determinar que el arreglo ya se encuentra ordenado de forma ascendente.

La complejidad en tiempo de ejecución del *Heapinsert Sort* en el mejor de los casos es $O(n)$ la cual se obtiene de la siguiente forma:

- Complejidad *BuildHeapMax* de la línea 1: $O(n)$
- Complejidad ciclo *for* de las líneas 2-3: $O(n)$
- Complejidad *Insertion Sort* de la línea 4: $O(n)$

sumando las complejidades en notación *O*:

$$O(n) + O(n) + O(n) = 3 \cdot O(n) = O(n) \quad (6)$$

3.2.2. Complejidad Temporal del Heapinsert Sort en el Peor de los Casos

La función *BuildHeapMax* (línea 1 del *Heapinsert Sort*) siempre tiene que hacer un orden $O(n)$ operaciones ya sea en el mejor o peor de los casos para construir un montón máximo; de forma similar el ciclo de repetición *for* de las líneas 2-3 siempre tiene que hacer un orden $O(n)$ operaciones independientemente del arreglo que reciba, de esta forma el peor de los casos del *Heapinsert Sort* viene directamente relacionado con el arreglo de entrada que reciba el *Insertion Sort* de la línea 4 del algoritmo. Es imposible que en esta línea 4 del algoritmo se reciba la peor entrada del *Insertion Sort*, es más, la información en el arreglo de entrada "tiende" a estar pseudo ordenada de forma ascendente, donde ya se cuenta con algunos elementos del arreglo ubicados en la posición que les corresponde en el ordenamiento y otros que no, éste tipo de entradas encaja perfectamente en el caso promedio del *Insertion Sort*, donde éste algoritmo toma una complejidad de $O(n^2)$.

La complejidad en tiempo de ejecución del *Heapinsert Sort* en el peor de los casos es $O(n^2)$ la cual se obtiene de la siguiente forma:

- Complejidad *BuildHeapMax* de la línea 1: $O(n)$
- Complejidad ciclo *for* de las líneas 2-3: $O(n)$
- Complejidad *Insertion Sort* línea 4: $O(n^2)$

sumando las complejidades en notación *O*:

$$O(n) + O(n) + O(n^2) = O(n^2) \quad (7)$$

3.2.3. Complejidad Temporal del Heapinsert Sort en el Caso Promedio

La complejidad en tiempo de ejecución del *Heapinsert Sort* en el caso promedio, es la misma que toma éste algoritmo en el peor caso, esto es debido al algoritmo *Insertion Sort* de la línea 4 que es el que domina en la complejidad del *Heapinsert Sort* donde se presenta ésta situación.

3.3. ¿Propiedad de Estabilidad en el Heapinsert Sort?

La *propiedad de estabilidad* en un algoritmo de ordenamiento es la siguiente: *si existen elementos repetidos en un arreglo a ordenar, entonces estos elementos en el arreglo ya ordenado conservan su orden original entre los elementos repetidos, es decir, entre elementos que tienen el mismo valor queda de primero entre ellos el que se encuentra primero en el arreglo al ser recorrido de izquierda a derecha, queda de segundo entre ellos el que se encuentra de segundo, y así sucesivamente.*

La *propiedad de estabilidad* no se presenta en el algoritmo *Heapinsert Sort*. Una justificación sencilla de esto es cuando el algoritmo recibe un arreglo que contiene en todas sus posiciones el mismo valor, este arreglo cumple la propiedad de montón máximo, motivo por el cual, la línea 1 del algoritmo no genera ningún cambio sobre el arreglo, pero el ciclo de repetición *for* de las líneas 2-3 invierte el mismo, donde el último número repetido ahora está de primero y así sucesivamente hasta que en la última posición del arreglo se encuentra el número repetido que originalmente se encontraba de primero. Por último, el *Insertion Sort* de la línea 4 no genera ningún cambio sobre el arreglo porque éste ya se encuentra ordenado. De esta forma el algoritmo *Heapinsert Sort* no tiene la propiedad de estabilidad porque el orden original de los elementos repetidos en el arreglo no se conserva en el arreglo finalmente ordenado.

4. ANÁLISIS DE RESULTADOS

Los algoritmos fueron implementados en Lenguaje C, utilizando las mismas estructuras de datos para todos los algoritmos; los arreglos a ordenar fueron almacenados en memoria principal. Todas las pruebas fueron realizadas sobre el mismo computador y sistema operativo. Se utilizó un computador con procesador AMD Athlon(tm) de 64 bits a 2.00 GHz, 1 GB en memoria principal y sistema operativo Windows XP versión 2002. Los algoritmo de ordenamiento fueron probados con arreglos de números enteros que se generaron aleatoriamente entre 0 y 99.999, garantizando que todos los algoritmos trabajaran sobre el mismo arreglo de entrada a ordenar, poniendo a competir los diferentes algoritmos con exactamente la misma entrada, lográndose esto con diferentes copias sobre el mismo arreglo para las diferentes longitudes de éste. En la siguiente tabla se presentan los tiempos de ejecución para los dos algoritmos analizados.

Tamaño	Heapinsert	Insertion
100.000	14	29
200.000	50	113
300.000	113	255
400.000	200	459
500.000	313	712
600.000	451	1.026
700.000	616	1.401
800.000	805	1.830
900.000	1.021	2.316
1.000.000	1.265	2.861

Tabla 2. Tiempos de ejecución del Heapinsert Sort versus el Insertion Sort (en segundos).

Con los datos de los resultados experimentales obtenidos para los tiempos de ejecución de los algoritmos *Heapinsert Sort* e *Insertion Sort* se evidencia que el *Heapinsert Sort* consume menos de la mitad del tiempo consumido por el *Insertion Sort* para ordenar exactamente el mismo arreglo.

En las siguientes tablas se calcularán los factores constantes de los algoritmos *Heapinsert Sort* e *Insertion Sort* para ratificar o refutar que la complejidad en el caso promedio del *Heapinsert Sort* es la misma que la del *Insertion Sort*, y donde lo único que cambia es que el *Heapinsert Sort* tiene un factor constante mucho más pequeño que el del *Insertion Sort*. La fórmula para calcular el *factor constante* es la siguiente:

$$Factor = \frac{Tiempo}{Complejidad} \quad (8)$$

Tamaño	Tiempo	Complejidad O(n²)	Factor (E-09)
100.000	14	10.000.000.000	1,400
200.000	50	40.000.000.000	1,250
300.000	113	90.000.000.000	1,256
400.000	200	160.000.000.000	1,250
500.000	313	250.000.000.000	1,252
600.000	451	360.000.000.000	1,253
700.000	616	490.000.000.000	1,257
800.000	805	640.000.000.000	1,258
900.000	1021	810.000.000.000	1,260
1.000.000	1.265	1.000.000.000.000	1,265
		Promedio Factor:	**1,270**

Tabla 3. Cálculo del factor constante del Heapinsert Sort.

Con los datos experimentales obtenidos para el tiempo de ejecución del *Heapinsert Sort* y con los valores calculados para el factor constante se ratifica que la complejidad del algoritmo en el caso promedio es $O(n^2)$, esto es debido a que para los diferentes tamaños de entrada del arreglo el factor constante tiende a estabilizarse tomando valores muy cercanos al promedio de los factores constantes.

Tamaño	Tiempo	Complejidad O(n²)	Factor (E-09)
100.000	29	10.000.000.000	2,900
200.000	113	40.000.000.000	2,825
300.000	255	90.000.000.000	2,833
400.000	459	160.000.000.000	2,869
500.000	712	250.000.000.000	2,848
600.000	1.026	360.000.000.000	2,850
700.000	1.401	490.000.000.000	2,859
800.000	1.830	640.000.000.000	2,859
900.000	2.316	810.000.000.000	2,859
1.000.000	2.861	1.000.000.000.000	2,861
		Promedio Factor:	**2,856**

Tabla 4. Cálculo del factor constante del Insertion Sort.

Como es de esperarse el cálculo del factor constante para el algoritmo *Insertion Sort* ratifica que la complejidad de dicho algoritmo en el caso promedio es $O(n^2)$.

5. CONCLUSIONES Y RECOMENDACIONES

- Se cumplió el objetivo principal de este artículo, dar a conocer un algoritmo de ordenamiento que no es más que una variante o combinación de otros algoritmos tradicionales.
- El análisis experimental de resultados ratificó la hipótesis de este artículo en la cual el llamado al *Insertion Sort* de la línea 4 del *Heapinsert Sort* realizaría menos trabajo que si se ordenara el arreglo utilizando únicamente el *Insertion Sort*.
- El análisis experimental de resultados permitió establecer que el *Heapinsert Sort* consume menos de la mitad del tiempo de ejecución del *Insertion Sort* para ordenar el mismo arreglo de entrada.
- Al estabilizarse el cálculo del factor constante para el algoritmo *Heapinsert Sort* en la *Tabla 3* se garantiza que la complejidad de éste algoritmo es $O(n^2)$.
- Con respecto a los datos trabajados en este artículo, al comparar los factores constantes de los algoritmos *Heapinsert Sort* (1,270E-09) e *Insertion Sort* (2,856E-09) se tiene que el factor constante del *Heapinsert Sort* es el 44.5% del tamaño del factor constante del *Insertion Sort*, esto justifica la economía en tiempo de ejecución en más del 50% en el *Heapinsert Sort* versus el *Insertion Sort*.
- El algoritmo *Heapinsert Sort* no presenta la propiedad de estabilidad, por este motivo no puede utilizarse como subrutina del algoritmo *Radix Sort*.

6. BIBLIOGRAFÍA

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algoritms*, Second Edition, The MIT Press, Cambridge, Massachusetts, Sección 2.1, páginas 15- 27, Capítulo 6, páginas 127-135, 2001.

[2] J. W. J. Williams, “Algorithm 232 (HEAPSORT)”, Communications of the ACM, 7:347-348, 1964.

[3] J. C. López, “Análisis de algoritmos de ordenamiento y selección, Diapositivas de clase del Curso de Fundamentos de Análisis y Diseño de Algoritmos (FADA)”, Programa de Ingeniería de Sistemas, Universidad del Valle, Cali, 2003. [Online].Disponible: http://eisc.univalle.edu.co/materias/FADA/clases/ordenamiento1.pdf