

# Método de ordenamiento de la burbuja en lenguaje ensamblador 8086

Antonio Muñoz Barrientos, Alan Alberto Montes Pérez  
7° Semestre Ing. Computación inteligente, Centro de ciencias básicas, Benemérita  
Universidad Autónoma de Aguascalientes  
20100  
al{229279,228063}@edu.uaa.mx

## Resumen

La aplicación del algoritmo de ordenamiento de la burbuja implementado en lenguaje ensamblador haciendo uso de registros y variables posicionales. Tratar los problemas de conversión ASCII – número para una y varias decenas de un número. Criterios de paro para la optimización de código. Reporte de complejidad y eficacia del algoritmo para tratar distintos tamaños de arreglos.

**Key words:** Ordenamiento de la burbuja, Conversión ASCII – número, Optimización de código, Complejidad.

## 1. Introducción

Durante una amplia cantidad de situaciones se debe trabajar con arreglos de números, y en muchos campos como estadística y análisis de datos debemos trabajar con arreglos ordenados de números. Existen muchos algoritmos enfocados en el ordenamiento de números, en este caso trataremos el método conocido como ordenamiento burbuja. Este algoritmo simula un comportamiento físico descubierto en la era de la antigua Grecia. Su principal ventaja es su fácil implementación, pero también recae en mucha complejidad a medida que crece la cantidad de números.

Se dará un enfoque en la aplicación del algoritmo en el lenguaje de bajo nivel ensamblador simulado en el programa EMU 8086. Se reportará el funcionamiento del algoritmo y se concluye con un análisis del comportamiento de este.

## 2. Marco teórico

### 2.1. Método de la burbuja

Dentro de los métodos de ordenamiento encontramos el método de la burbuja como uno de los más básicos y fáciles de implementar. Se considera un método comparativo e iterativo que hace referencia al concepto de física propuesto por Arquímedes, *la densidad* [1].

La densidad dice que los objetos más densos se colocan debajo de los objetos menos densos. En ordenamiento, esta densidad la consideramos como el valor propio que puede tener un valor. Es decir, si el valor es más grande se considera que tiene una mayor “densidad”, por lo tanto, los valores con mayor densidad se colocarán hasta abajo (al final del arreglo) y los valores con menor densidad hasta arriba (al principio del arreglo).

En complejidad computacional, el algoritmo de ordenamiento de la burbuja se considera con una complejidad de  $O(n^2)$ , esto quiere decir que entre más datos se ingresen como entrada más tardará el algoritmo en completar el ordenamiento. No se recomienda usar este método para un conjunto de datos muy grande por la poca optimización que tiene [2].

La estructura del algoritmo es simple, son dos ciclos *for* anidados donde el primero va recorriendo todo el arreglo posición por posición; el segundo recorre todo el arreglo desde la posición donde está el elemento actual del primero *for* y hasta el final del arreglo. Con esto logramos que cada posición del arreglo se compare con todos los demás. Dentro de los *for* tenemos una condición *if* para verificar la densidad de los valores que tenemos actualmente. Si el valor que tenemos en el primer *for* es mayor que el valor que tenemos en el segundo, debemos de intercambiar de posición estos valores. Debemos hacer uso de una variable temporal para poder realizar el intercambio de valores para no perder ningún valor.

```
i = N;
sorted = false;
while((i > 1) && (!sorted))
{
    sorted = true;
    for(int j=1; j<i; j++){
        if (a[j-1] > a[j]) {
            temp = a[j-1];
            a[j-1] = a[j];
            a[j] = temp;
            sorted = false;
        }
    }
    i--;
}
```

### 3. Desarrollo

Para poder realizar el ordenamiento de un arreglo primero necesitamos obtener los valores que se van a ordenar. Se utilizó la interrupción 0Ah para poder leer los números. Los parámetros establecidos fueron un vector de hasta 20 posiciones que contenga números positivos entre 0-99. Con esta información se estimó el tamaño de memoria necesario para el vector, en este caso se utilizaron 80 posiciones para evitar inconvenientes. Por cada valor individual ingresado se realizaron varias validaciones. Primero se detectó si el valor ingresado es el valor llave para identificar dos valores diferentes en el arreglo, en este caso se utilizó la coma como ese valor llave. La segunda validación revisa si el valor ingresado es un número, y la tercera validación comprueba si el valor ingresado es de un carácter (0-9) o de dos caracteres (10-99).

Ilustración 1: Pseudocódigo bubble sort

Tras la verificación del arreglo se comenzó a transformar los valores ASCII en valores numéricos. Para eso comprobamos la cantidad de caracteres que tiene un valor entre dos comas, dependiendo de esa cantidad se realizaron tres posibles acciones.

Ilustración 2: Caso 1 y 2 de la cantidad de valores

1. El valor contiene un solo carácter: Se transforma a número y se asigna a la posición del arreglo.
2. El valor contiene dos caracteres: Transformamos cada carácter de forma independiente, lo multiplicamos por 10 el primer carácter, le sumamos el segundo y lo asignamos a la posición del arreglo.
3. El valor contiene más de dos caracteres: Se considera un error y regresamos el programa al estado para pedir los números.

```
TwoDigits:
MOV BX, DI
MOV DI, CX
MOV AL, STRINGNUMS[DI]
MUL X10
MOV NUMS[SI], AL
INC DI
MOV AL, STRINGNUMS[DI]
ADD NUMS[SI], AL
JMP Reset

OneDigit:
MOV BX, DI
MOV DI, CX
MOV AL, STRINGNUMS[DI]
MOV NUMS[SI], AL
JMP Reset
```

Para el ordenamiento de los valores utilizamos dos variables contadores y dos registros para realizar las comparaciones. Para poder recorrer el arreglo con los dos ciclos necesarios utilizamos dos variables posicionales *DI*, *SI* y utilizamos los registros *AH* y *AL* para comparar los valores.

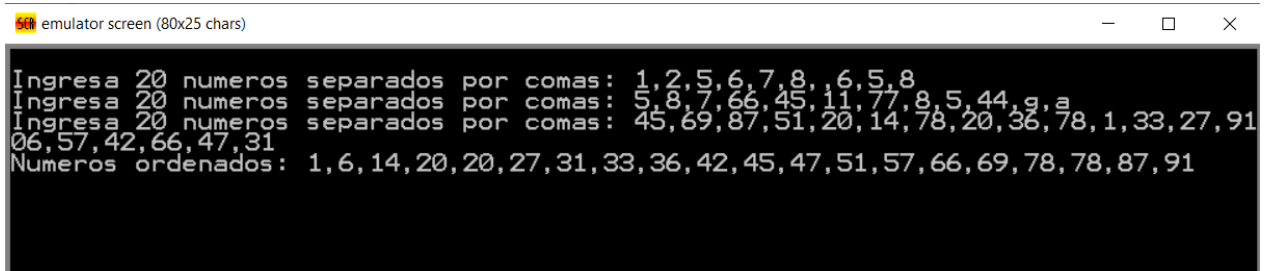
Se agregó una variante al algoritmo para optimizar el recorrido, para evitar que recorra todo el arreglo cuando ya está ordenado y puede que aún le falten posiciones por recorrer, se utilizó una variable llamada *changes* que cuenta cuantos cambios se realizaron en una iteración del arreglo. Si la cantidad es de 0 quiere decir que el arreglo ya está ordenado, por lo tanto, se puede terminar el ciclo antes de recorrer todo el algoritmo. Este caso de optimización tiene un problema. En el caso de que el arreglo esté totalmente desordenado y todas las posiciones requieran cambiar; al finalizar de ordenar se realiza una última iteración que valide la cantidad de cambios, como ya está ordenado la cantidad es de 0 y finaliza el algoritmo.

```
While:
CMP Changes, 0h
JE Reset3
JMP Reset2
```

Ilustración 3: Aplicación de variable *changes*

## 4. Resultados

A continuación, se presentan los resultados obtenidos en la ejecución del código. Para el input se presentaron diversos casos como una doble coma, una letra y números que empiecen con el 0.



```
emulator screen (80x25 chars)
Ingresa 20 numeros separados por comas: 1,2,5,6,7,8,,6,5,8
Ingresa 20 numeros separados por comas: 5,8,7,66,45,11,77,8,5,44,g,a
Ingresa 20 numeros separados por comas: 45,69,87,51,20,14,78,20,36,78,1,33,27,91
06,57,42,66,47,31
Numeros ordenados: 1,6,14,20,20,27,31,33,36,42,45,47,51,57,66,69,78,78,87,91
```

Se observamos que en los casos donde existen errores como dobles comas y letras el programa repite la instrucción de pedir los valores. Después se ingresan los valores de manera adecuada y se espera a que el programa arroje el arreglo ordenado. Se considera también que se puedan repetir números y que estos se sitúen en su posición.

## 5. Discusión

Tras analizar el comportamiento del algoritmo, su implementación y su codificación se llegó a la conclusión de que el algoritmo no es eficiente de aplicar para un conjunto de números muy grande. En la simulación en 8086 el algoritmo tomó 2 minutos en ordenar 20 números desordenados completamente.

Considerando la complejidad cuadrática del algoritmo, utilizar más números incrementaría mucho el tiempo necesario de respuesta.

## 6. Conclusión

El uso e implementación del algoritmo del método de la burbuja es sencillo de entender. Es un método poderoso para asegurarse el ordenamiento de arreglos de tamaño pequeño. Su principal desventaja es su poca optimización a mayor cantidad de números ingresados. Para su aplicación en lenguaje ensamblador, el principal reto fue tener orden las variables posicionales, ya que son las que van marcando la pauta del algoritmo y son las bases de las comparaciones e intercambios.

## 7. Referencias

1. Arquímedes: *Principio arquímedes*
2. Reem Saadeh; Mohammad Qatawneh: *PERFORMANCE EVALUATION OF PARALLEL BUBBLE SORT ALGORITHM ON SUPER COMPUTER IMAN1*. Jordania: Department of Computer Science, King Abdullah II School for Information Technology, The University of Jordan, Amman, Jordan.