

# LABORATORIO 2 - ORDENAMIENTO COMBSORT EN MIPS

ANDRÉS FELIPE GIRALDO YUSTI<sup>1</sup> y JUAN CAMILO VELEZ<sup>2</sup>

<sup>1</sup>Universidad de Antioquia, Facultad de Ingeniería, andres.giraldoy@udea.edu.co

<sup>2</sup>Universidad de Antioquia, Facultad de Ingeniería, camilo.velez4@udea.edu.co

JOHN BYRON BUITRAGO PANIAGUA

## 1. Introducción

En esta práctica se desarrolló un programa en lenguaje ensamblador MIPS que permite leer un archivo de texto con números separados por comas, luego los ordena utilizando el algoritmo CombSort en orden ascendente, y después guardar el resultado ordenado en un nuevo archivo de texto llamado lista-ordenada.txt.

El objetivo principal es aplicar conceptos de bajo nivel como el acceso a memoria y manipulación de registros, evitando utilizar pseudo-instrucciones para trabajar directamente con el conjunto de instrucciones básicas de MIPS.

## 2. Planteamiento del problema y objetivos

### 2.1. Problema

Desarrollar un programa en MIPS que:

- Lea un archivo con números enteros separados por coma.
- Procese los datos y los almacene en memoria.
- Ordene el arreglo utilizando el algoritmo CombSort de forma ascendente.
- Escriba el resultado en un nuevo archivo de salida, separados por coma.

### 2.2. Objetivos

- Implementar entrada/salida de archivos mediante llamadas al sistema.
- Implementar desde cero el algoritmo CombSort en MIPS sin pseudo-instrucciones.
- Escribir el resultado final en un nuevo archivo.

## 3. Fundamentos teóricos

CombSort es un algoritmo de ordenamiento basado en la idea de comparar elementos de un arreglo que están separados por una distancia determinada, llamada "gap". En cada iteración, se comparan los elementos ubicados a esa distancia y se intercambian si están desordenados. Luego, el "gap" se reduce progresivamente hasta llegar a uno, momento en el que se realizan comparaciones entre elementos contiguos.

El algoritmo permite recorrer y ordenar los datos de forma eficiente, siguiendo una lógica estructurada de comparaciones repetidas mientras se reduce el "gap". Esta mecánica se puede implementar mediante ciclos anidados y condiciones de intercambio.

En esta práctica, se implementó CombSort en lenguaje ensamblador MIPS, utilizando el entorno MARS. Se manejaron operaciones como la lectura y escritura de archivos mediante llamadas al sistema (syscalls), y se utilizaron registros para almacenar temporalmente datos.

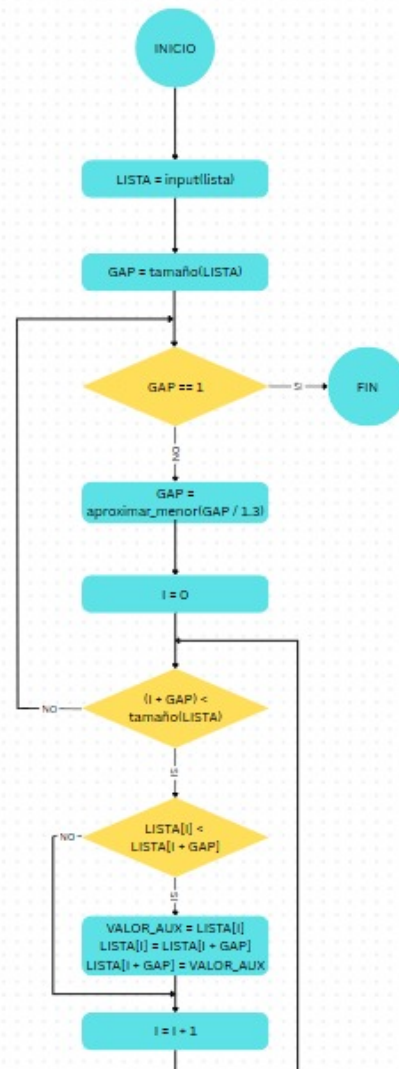


Figura 1. Diagrama de flujo del algoritmo de ordenamiento CombSort

## 4. Decisiones de diseño

Durante el desarrollo del programa se tomaron decisiones clave para garantizar una estructura modular, mantenible y funcional en el lenguaje ensamblador MIPS, sin el uso de pseudo-instrucciones. A continuación, se describen y justifican estas decisiones:

### Separación en funciones

El programa fue dividido en funciones o bloques de código que realizan tareas específicas. Esto permite una mejor organización del código, facilita la depuración por partes y hace que el flujo general del programa sea más claro. Las funciones implementadas fueron:

- **main:** función principal que coordina todo el flujo de ejecución. Se encarga de llamar en orden a los módulos de lectura, parseo, ordenamiento y escritura. Esta función no tiene entradas, ya que es el punto inicial, y organiza toda la lógica del programa.

- **parse\_numbers:** esta función convierte el contenido del archivo leído, que está en formato ASCII y separado por comas, en números enteros. Se decidió crearla como módulo separado porque el parseo requiere control de caracteres, acumulación numérica y validación, procesos que son independientes del resto del algoritmo.
- **combsort:** se implementó como una función dedicada al ordenamiento, para separar la lógica de procesamiento de datos de la lógica de control del programa. Esto permite verificar y reutilizar este módulo sin depender del resto del código.
- **escribir\_archivo:** esta función escribe el resultado ordenado en un nuevo archivo. Fue útil aislarla para manejar únicamente la conversión de enteros a ASCII y el uso de las syscalls de escritura, simplificando así el diseño del programa principal.

### Uso de memoria alineada

Se tomó la decisión de almacenar los números en memoria alineada a 4 bytes debido a que las instrucciones de lectura y escritura en MIPS requieren acceso a direcciones alineadas para evitar excepciones de ejecución. Por ello, se reservó una región de memoria y se manipuló con direcciones ajustadas mediante saltos de 4 bytes.

### Control de flujo

Se utilizaron ciclos y condiciones con registros temporales para iterar sobre los datos y comparar valores en el algoritmo CombSort. Se decidió usar 'mult', 'div', 'mflo', y comparaciones explícitas en lugar de operaciones más abstractas, para cumplir con la restricción de no usar pseudo-instrucciones.

### Syscalls

Se emplearon las llamadas al sistema disponibles en el entorno MARS: syscall 13 para abrir archivos, syscall 14 para leer, syscall 15 para escribir, y syscall 16 para cerrar archivos. Estas llamadas permiten simular entrada/salida de archivos en un entorno que normalmente no tiene acceso directo a sistemas de archivos.

### Elección del entorno MARS

Se escogió el simulador MARS para el desarrollo debido a su compatibilidad con syscalls, su facilidad para inspeccionar registros y memoria, y la posibilidad de realizar pruebas paso a paso con mensajes de depuración. Esto fue crucial para identificar errores de alineación y verificar el correcto funcionamiento del algoritmo.

### Gestión de errores

Durante el diseño se validó cada bloque por separado antes de integrarlo, lo cual permitió detectar errores como desalineación de memoria, bucles infinitos o fallos en el parseo. La modularidad del código facilitó estas validaciones y permitió mejorar cada parte del programa de manera progresiva.

### Formato de salida

La decisión de escribir el archivo de salida en el mismo formato que el archivo de entrada —números decimales separados por comas— se tomó para facilitar la validación visual del resultado y asegurar la compatibilidad con otras herramientas que usen el mismo formato de datos.

## 5. Metodología

El desarrollo de esta práctica se realizó construyendo y probando cada parte del programa en MIPS paso a paso, con el objetivo de leer un archivo, procesar sus datos numéricos, ordenarlos y escribir el resultado en otro archivo.

La Figura 2 muestra el flujo lógico general que se implementó para lograr el objetivo del laboratorio. Las fases principales de la metodología aplicada fueron:



Figura 2. Diagrama de flujo del algoritmo

- **Lectura del archivo:** Se abrió el archivo que contiene los números y se leyó su contenido.
- **Parseo de los datos:** se recorrió el contenido leído, identificando los números según el separador definido y convirtiéndolos a enteros que fueron almacenados en un arreglo en memoria.
- **Ordenamiento:** se aplicó el algoritmo CombSort directamente sobre el arreglo cargado, comparando y reordenando los elementos mediante ciclos y saltos condicionales en MIPS.
- **Escritura del archivo de salida:** tras ordenar los datos, se generó un nuevo archivo, escribiendo allí los números ordenados como texto decimal.

Cada etapa fue probada individualmente para asegurar su correcto funcionamiento antes de integrarse al flujo completo. Se puso especial atención en el manejo de memoria alineada y en el uso correcto de los syscalls para interactuar con archivos en MIPS bajo el entorno MARS.

## 6. Diseño del sistema

El sistema desarrollado en MIPS Assembly permite leer una lista de números desde un archivo de texto, almacenarlos en un arreglo en memoria, aplicar el algoritmo de ordenamiento **CombSort**, y finalmente guardar el resultado en un nuevo archivo. A continuación, se describen los bloques funcionales y su interacción general.

### 6.1. Lectura del archivo

El programa comienza solicitando la apertura del archivo `lista.txt`, ubicado en el mismo directorio del ejecutable. Para esto se utiliza la syscall correspondiente a `open` en modo lectura. El contenido se carga en un `buffer` usando la syscall `read`, el cual posteriormente es mostrado en consola para verificarlo.

### 6.2. Parseo de datos

Una vez leído el contenido, se realiza un recorrido carácter por carácter del `buffer`. Se construyen números decimales detectando separadores (comas) y se almacenan en una estructura de datos en memoria utilizando direcciones alineadas. Este proceso convierte cada secuencia de caracteres ASCII en su equivalente numérico, utilizando multiplicaciones sucesivas por 10 y sumas de dígitos.

### 6.3. Ordenamiento CombSort

Sobre el arreglo de enteros se aplica el algoritmo CombSort. Este algoritmo mantiene una variable `gap` inicializada como el número de elementos ( $n$ ) y en cada iteración la reduce según la fórmula:

$$gap = \left\lceil \frac{gap}{1.3} \right\rceil$$

El algoritmo compara elementos con un espacio de `gap` entre ellos, intercambiándolos si es necesario. Se repite el proceso hasta que `gap = 1` y no se hayan hecho más intercambios, garantizando que el arreglo está ordenado.

### 6.4. Creación del archivo de salida

Una vez ordenado el arreglo, se abre un nuevo archivo con nombre `lista_ordenada.txt` utilizando la syscall `open` en modo escritura

con permisos de creación. Cada número del arreglo se convierte nuevamente a ASCII, carácter por carácter, y se escribe en el archivo seguido de una coma.

### 6.5. Visualización y validación

El sistema imprime el contenido original del archivo y el número total de elementos parseados. También puede imprimirse el arreglo ordenado por consola si se habilita dicha sección en el código. Esto permite validar la correcta ejecución del proceso completo sin necesidad de abrir los archivos manualmente.

## 7. Resultados y análisis

El sistema logró leer correctamente archivos de entrada como 'lista.txt' con contenido tipo: '12,8,15,4,21', ordenarlos y generar el archivo 'lista\_ordenada.txt' con los valores: '4,8,12,15,21'.

Se detectaron y corrigieron errores comunes mal uso de syscall y conversión incorrecta de caracteres. El sistema fue validado progresivamente por partes para garantizar estabilidad y comprensión del código.

## 8. Conclusiones

Este laboratorio permitió poner en práctica conceptos esenciales de programación en ensamblador, manejo de memoria, archivos y estructuras algorítmicas. La implementación del algoritmo CombSort sin pseudo-instrucciones representó un reto significativo que consolidó el entendimiento de bajo nivel en la arquitectura MIPS.

Además, se reforzó la importancia de las conversiones entre tipos de datos y el uso correcto de las llamadas al sistema, aspectos clave en la programación de sistemas embebidos o de bajo nivel.