TP1 - Arquitectura

Kevin Giribuela

Octubre 2021

1. Introducción

En el presente informe se pretende realizar en MARIE un programa que ordene un vector de datos. La cantidad de elementos que contendrá el vector se va a encontrar en la posición de memoria $0x001$. A partir de allí en adelante, se hallan los elementos del vector, es decir, en la posición $0x002$ encontraremos el primer elemento del vector.

Para ordenar el vector, se seleccionó uno de los muchos algoritmos disponibles. El algoritmo utilizado fue el *algoritmo de inserción*.

2. Objetivo

La meta del programa es realizar la tarea de ordenamiento en la menor cantidad de instrucciones posibles sin perder la prolijidad y legibilidad del código.

3. Desarrollo

3.1. ¿Por qué este algoritmo?

Si bien la elección de el algoritmo de ordenamiento parece ser una tarea trivial, no lo es en absoluto. A la hora de elegir uno de ellos, se deben tener muchos factores en mente.

En este caso, se cuenta con una máquina de escaso repertorio de instrucciones. Este hecho, dificulta de manera significativa la implementación y la prolijidad del código. MARIE se caracteriza por no contar con una pila en sus registros para poder manipular de una manera más eficiente los punteros, consecuentemente, cuando se necesita utilizar funciones recursivas, la tarea de ordenamiento se dificulta ya que no se cuenta con un registro específico que almacene la dirección de memoria en donde se encontraba el programa antes de entrar nuevamente a la misma subrutina.

Lo detallado anteriormente, ya da un indicio de que cualquier algoritmo que necesite el uso de subrutinas recursivas debe ser implementado con total cuidado y prolijidad para poder lograr un código eficiente.

Debido a que la gran mayoría de los algoritmos de ordenamiento requieren de alguna manera u otra, algún tipo de recursividad, la elección se enfocó en qué algoritmo se adapta mejor a los requerimientos propuestos. Esto es, dado un vector de la forma

$$\{-3, -2, -1, 0, 1, 2, 3\} \tag{1}$$

hallar un algoritmo que ordene los elementos de la manera más eficiente posible. Por supuesto, es evidente que el vector de la Ec. 1 se encuentra ordenado y no es necesario usar un algoritmo, sin embargo, tal vector será de utilidad para analizar el rendimiento del algoritmo mas adelante.

Dado que el largo del vector que se desea ordenar no es grande, un algoritmo que ordene de manera eficiente los elementos, es el de inserción. Sin embargo no se debe perder de vista que tal algoritmo, es *altamente ineficiente* cuando se lo utiliza para ordenar vectores gran tamaño.

3.2. Implementación

3.2.1. Diagrama de flujo

en la Figura 1 se puede observar el diagrama de flujo utilizado para lograr la implementación del algoritmo. Por supuesto, el indice j y $j-1$ son puramente a modo ilustrativo ya que en MARIE, los punteros harán las veces de índices.

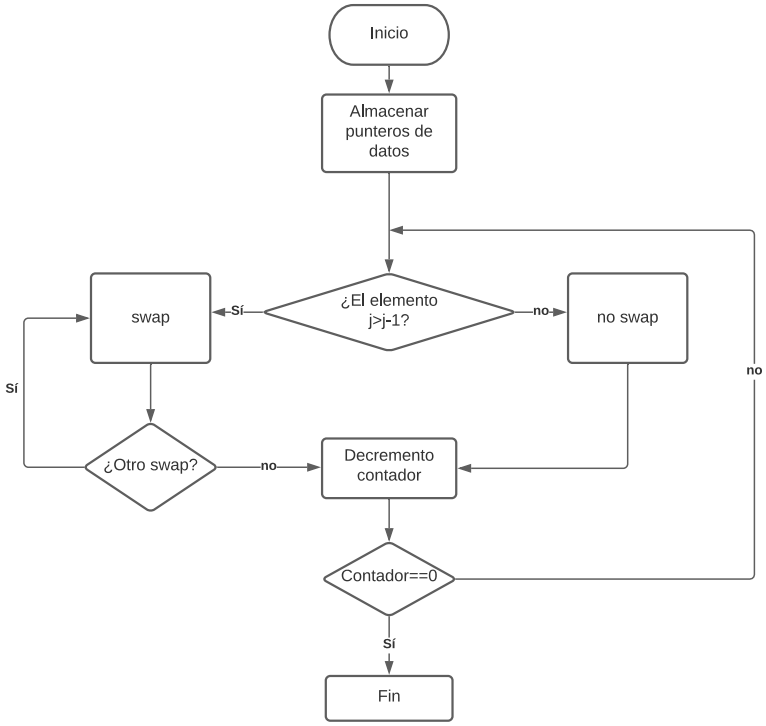

Figura 1: Diagrama de flujo

3.2.2. Código

Una vez analizado el diagrama de flujo del algoritmo, se examinará el código implementado. Nos detendremos en secciones del mismo y analizaremos la funcionalidad en bloques.

```
/****************************************** VARIABLES ******************************************/
LENGTH, HEX 001     / Posición de memoria donde se encuentra la longitud del vector
START,  HEX 002     / Posición de memoria donde se encuentra el inicio del vector

P1,     DEC 0       / Puntero
P2,     DEC 0       / Puntero
P3,     DEC 0       / Puntero
P4,     DEC 0       / Puntero

ONE,    DEC 1       / Variable utilizada para sumar o restar 1

AUX1,   DEC 0       / Variable auxiliar para realizar el "swap" entre elementos del vector
AUX2,   DEC 0       / Variable auxiliar para realizar el "swap" entre elementos del vector

CONT,   DEC 0       / Contador
```

Figura 2: Variables utilizadas

En la Figura 2 no hay mucho de que hablar, en ella se ilustran la totalidad de variables utilizadas y la utilidad que tiene cada una. Nuevamente, recordando lo mencionado en secciones previas, al no contar con una pila en MARIE, se deben utilizar una mayor

cantidad de punteros *auxiliares* para no perder la posición de memoria en la que se encontraban los punteros originalmente.

Siguiendo con el orden del diagrama de flujos, presentamos la inicialización de variables (ver Figura 3). Como bien dice la figura, esta sección del código se encarga de cargar en el puntero $P1$ la posición de memoria del primer elemento del arreglo que va a ser ordenado. Seguido esto, se toma *el contenido* en memoria de la variable $LENGTH$, se le resta 1, se lo almacena en la variable $CONT$. Esta última, llevará la cuenta de cuántas veces se ha ordenado el arreglo, de esta forma, el algoritmo sabrá cuando finalizar el ordenamiento.

```
/****************************************** INICIALIZACIÓN ******************************************/

ordenar,        Load    START   / Almaceno en P1 la dirección de memoria donde comienzan los datos
                Store   P1

                LoadI   LENGTH  / Almaceno en la variable CONT = LENGTH - 1
                Subt    ONE
                Store   CONT
```

Figura 3: Inicialización de variables

Una vez inicializadas las variables principales, se entra al *loop* principal del programa (Figura 4). En él se lleva a cabo la tarea de comparación de elementos, luego el resultado de tal comparación decidirá si es necesario realizar un *swap* o no.

```
/****************************************** LOOP PRINCIPAL ******************************************/
loop,   Load    P1      / Cargo el puntero P1 y lo almaceno en P4 para utilizar a P1 sin perder la posición original
        Store   P4

        Load    P1      / Cargo P1 en el AC, le sumo 1 y lo almaceno en P2
        Add     ONE
        Store   P2

        LoadI   P2      / Cargo el contenido del puntero P2 y lo almaceno en AUX2
        Store   AUX2

        LoadI   P1      / Cargo el contenido de lo que apunta P1, le resto el contenido de lo que apunta P2 y me fijo si es mayor a 0
        Subt    AUX2    / Si el AC > 0, salto a swap, sino, salto a nswap.
        Skipcond 800
        Jump    nswap
        Jump    swap
```

Figura 4: Loop principal del programa

Se puede apreciar cómo en la primer línea del loop se carga en el AC al puntero $P1$ y luego se lo almacena en $P4$. Esto da un indicio de que el puntero $P1$ es utilizado por las subrutinas para realizar las tareas de *swap* y por lo tanto, su valor se verá alterado en el trayecto.

Mirando con detenimiento el loop, se logra ver que una vez cargado en memoria el puntero $P1$ y $P4$, se carga $P2$ y luego se carga su contenido en memoria para hacer una comparación aritmética. La Figura 5 ilustra esta idea.

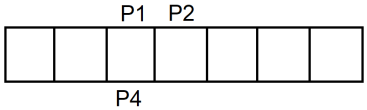

Figura 5: Representación gráfica del funcionamiento de los punteros $P1$ y $P2$

Una vez hecha la comparación aritmética, se decide si es necesario realizar un *swap* o no. Tomemos el caso más sencillo para comenzar el estudio de las subrutinas implementadas, esto es, la subrutina *nswap*.

```
nswap,  Load    CONT    / Cargo el contador y le resto uno
        Subt    ONE
        Store   CONT

        Load    P1      / Cargo el puntero P1 y le sumo 1
        Add     ONE
        Store   P1

        Load CONT       / Chequeo si el contador es 0
        Skipcond 400    / Si el AC==0, salto a halt, sino, salto a loop
        Jump    loop
        Jump    halt
```

Figura 6: Subrutina - nswap

Como se observa, se trata de una subrutina bastante modesta. En ella se decrementa el contador indicando que ya se analizó un elemento del vector, luego se incrementa en 1 la posición del puntero $P1$ y finalmente se corrobora que el contador no haya llegado a cero.

La situación cambia rotundamente cuando la tarea que hay que realizar es un *swap* ya que además de realizar ese intercambio entre elementos del vector hay que corroborar si es necesario o no realizar otro *swap*. En esta sección de código es donde entra en juego el problema de recursividad ya mencionado. En la Figura 7 se observa dicha subrutina.

```
swap,   LoadI   P2      / Cargo el contenido en memoria de lo que apunta P2 y lo almaceno en AUX2 para hacer el swap
        Store   AUX2

        LoadI   P1      / Cargo el contenido en memoria de lo que apunta P1 y lo almaceno en AUX1 para hacer el swap
        Store   AUX1

        /***** Ejecuto el swap *****/
        Load    AUX1    / Cargo el valor de AUX1 y lo almaceno en la dirección que apunta P2
        StoreI  P2

        Load    AUX2    / Cargo el valor de AUX1 y lo almaceno en la dirección que apunta P2
        StoreI  P1
```

Figura 7: Subrutina - swap

Para solucionar el inconveniente de la recursividad, dentro de la subrutina *swap*, se implemento la operación IF, $ELSE$.

En pocas palabras, si $P1$ **no apunta** a la primer posición del vector, se define un puntero $P3$ tal que apunte a $P1 - 1$. Luego, se realiza la misma comparación aritmética que se llevó a cabo en el loop principal. La Figura 8 da una idea gráfica de la situación.

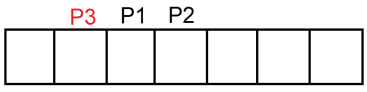

Figura 8: Representación gráfica de la necesidad de otro *swap*

- **En caso de ser necesario realizar otro** *swap*, se salta a otra subrutina definida como *pila*. Nuevamente queda en evidencia el inconveniente de la recursividad imponiendo la necesidad de crear otra subrutina que se encargue del manejo de los punteros para poder volver a realizar la misma operación (ver Figura 9).

```
/*****************************************************************************************************/
pila,  Load    P1      / Cargo el puntero P1 y lo alcameno en P2
       Store   P2

       Load    P3      / Cargo el puntero P3 y lo almaceno en P1
       Store   P1

       Jump    swap    / Salto a la subrutina que hace el swap. Pero esta vez, los punteros redefinidos
```

Figura 9: Subrutina - pila

En esta subrutina, se administran los punteros para poder volver a entrar a la subrutina *swap*. La idea gráfica se observa en la Figura 10.

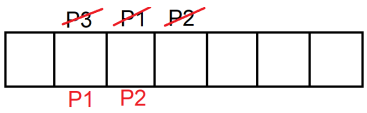

Figura 10: Tarea de la subrutina *pila*

- **En caso de NO ser necesario implementar otro** *swap*, el programa salta a la subrutina *go*. Esta subrutina se encarga de decrementar el contador, volver a darle al puntero $P1$ su valor original que tenía en el momento de entrar al loop principal y finalizar el programa en caso de que el contador halla llegado a 0.

```
go,    Load    CONT    / Cargo el contador y lo decremento
       Subt    ONE
       Store   CONT

       Load    P4      / Cargo P4 (donde originalmente se encontraba P1), y lo almaceno en P1
       Add     ONE
       Store   P1

       Load CONT       / Cargo el contador y corroboro que no sea 0.
       Skipcond 400    / Si el AC==0, simplemente salto a halt, sino es cero, salto a loop
       Jump    loop
       Jump    halt
```

Figura 11: Subrutina - go

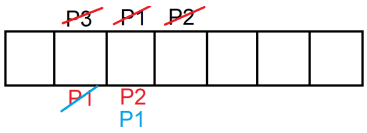

Figura 12: Representación gráfica de subrutina go

4. Performance

4.1. Velocidad

Una vez discutido el código del algoritmo de ordenamiento, se procede a realizar un análisis del performance del código cuando el vector que tenemos que ordenar toma los

siguientes valores:

$$V_1 = \{3, 2, 1, 0, -1, -2, -3\}$$
$$V_2 = \{-3, -2, -1, 0, 1, 2, 3\}$$
$$V_3 = \{0, -2, 1, 3, -1, 2, -3\}$$

Con el objetivo de tener un contraste o tomar dimensión de la rapidez con la que ordena el algoritmo, se comparan los resultados obtenidos con el algoritmo de la burbuja implementado por tres compañeros distintos. Para ello, se han ejecutado los mismos vectores de datos con los dos algoritmos y se relevó la cantidad de instrucciones que debió realizar cada uno de ellos para obtener el resultado final.

En la Tabla 1 se pueden observar los resultados obtenidos.

Cuadro 1: Resultados obtenidos.

Vector	**Algoritmo**	**Instrucciones**
$\{-3, -2, -1, 0, 1, 2, 3\}$	**Inserción**	**127**
	Burbuja1	156
	Burbuja2	160
	Burbuja3	117
$\{3, 2, 1, 0, -1, -2, -3\}$	**Inserción**	**589**
	Burbuja1	988
	Burbuja2	787
	Burbuja3	930
$\{0, -2, 1, 3, -1, 2, -3\}$	**Inserción**	**375**
	Burbuja1	918
	Burbuja2	677
	Burbuja3	880

4.1.1. ¿Qué sucede con vectores de gran longitud?

Se ha visto en la introducción que el algoritmo utilizado era ineficiente frente a una gran volumen de datos que se deseen ordenar. Para ver qué tan ineficiente es, nuevamente lo comparamos contra el algoritmo de la burbuja pero esta vez utilizando un vector de 20 datos ordenado de manera decreciente (peor condición para ambos algoritmos). Los resultados obtenidos se ven a continuación

Cuadro 2: Resultados obtenidos con gran volumen de datos.

Vector	**Algoritmo**	**Instrucciones**
V_{20}	**Inserción**	**5061**
	Burbuja1	8671
	Burbuja2	6286
	Burbuja3	7729

4.2. Espacio en memoria

Si bien la velocidad de ordenamiento importa, no significa que el espacio en memoria consumido por el programa no. Este recurso es vital cuando se está trabajando con limitados recursos, es por eso que también entra en el análisis de performance.

Al igual que para el caso de la velocidad, el algoritmo contra el que se comparó el espacio en memoria es el de la burbuja. La Tabla 3 muestra los resultados

Cuadro 3: Resultados obtendos.

Algoritmo	**Espacio en memoria [kB]**
Inserción	**168kB**
Burbuja1	122kB
Burbuja2	156kB
Burbuja3	148kB

5. Conclusiones

Como se ha visto, los resultados obtenidos a través de diferentes pruebas han sido favorables para el algoritmo de inserción.

En cuanto a velocidad, tanto en pequeña como en gran cantidad de datos, el que mejor resultado obtuvo fue el algoritmo seleccionado.Sin embargo, cuando se inspecciona el espacio en memoria consumido por el algoritmo de inserción y los otros 3 de la burbuja, el algoritmo de inserción no parece ser el adecuado.

Mas allá de la velocidad de ordenamiento y espacio en memoria, no se puede no hablar de la complejidad que presenta implementar el algoritmo de inserción con un repertorio de instrucciones tan escueto. Cuando se pone esto en la balanza, el algoritmo de la burbuja pareciera ser una muy buena elección. Esto es así ya que el algoritmo de inserción luce todo su potencial con el uso de la recursividad, característica que no está disponible en la arquitectura MARIE. Sin embargo, cuando miramos la simpleza del algoritmo de la burbuja, uno se ve tentado a encarar el problema con ese algoritmo.

En conclusión, si la arquitectura que debe utilizarse es MARIE, y se desea la velocidad máxima en el ordenamiento sin importar al complejidad de código obtenida, una buena elección puede ser el algoritmo de inserción. Sin embargo, si la velocidad no es un parámetro tan exigente en nuestro diseño, el código puede verse altamente simplificado y elegante utilizando el algoritmo de la burbuja.