Integrador TP N° 1

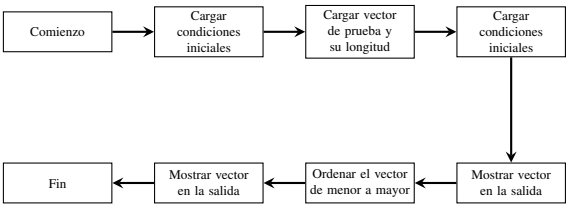
Tomás Vidal
Arquitectura de Computadoras
Facultad de Ingeniería, UNLP, La Plata, Argentina.
26 de Abril, 2024.

I. Algoritmo elegido

Para resolver el problema se empleó el algoritmo de **Bubble Sort**, debido a que cumple con los requerimientos, es simple, facil de implementar y, solo necesita un espacio extra de memoria (pero en MARIE hay que conservar la dirección también, por lo que se hacen dos espacios). Aunque no todo es perfecto, este algoritmo implica un coste de media de $\mathbf{O}(n^2)$, en el mejor de los casos es $\mathbf{O}(n)$, y en el peor nuevamente es $\mathbf{O}(n^2)$; esto quiere decir que si se tiene un vector con n elementos, se tienen que realizar de media n^2 operaciones para ordenar este vector con el algoritmo.

I-A. Lógica del programa

La implmentación del algoritmo de la figura 1 en sí se realizó en la subrutina de la figura 2, el resto del código se enfoca en cargar los datos de prueba y adecuar las posiciones de memoria especificadas correctamente. La extensión del código se debe a su *alta organización*, *documentación* y *buenas prácticas generales*, porque de otra manera el algoritmo por sí solo no hubiera llevado tantas líneas. El siguiente diagrama explica mejor el flujo general del programa.



Como se puede observar se cargan dos veces las condiciones iniciales (es una forma de decirle a los valores que toman las variables inicialmente), esto se debe a que el primer *seteo* es por si se quiere reiniciar el programa sin tener que reensamblar el código, y la segunda condición es debido a que al principio se cargan los datos del vector de prueba en la posición de memoria correcta, y para esto se reutilizan variables empleadas posteriormente en el algoritmo de organización del vector de datos, por lo que se requiere un reestablecimiento de los valores inciales de las mismas.

I-B. Algoritmo implementado

El algoritmo de Bubble Sort implementado se hizo a partir del diagrama de flujo 1, en el código de se puede apreciar que se documenta cada parte acorde a este diagrama. De todas formas se hará una breve explicación del mismo. La subrutina consiste básicamente en dos bucles (*iIndex* y *jIndex*) que basados en ciertas condiciones hacen cambiar el dato contenido en la posición de memoria: **0x0002 + j**, es decir $a[j]$, con el valor de $a[j+1]$. La condición justamente sería que si $a[j]$ es mayor que $a[j+1]$ (el valor anterior es mayor que el posterior) hay que hacer la permutación de los datos, para lo cual se debe almacenar temporalmente uno de los datos mientras se hace dicha permutación (en el código: *aJAddr* y *aJData*), una vez que se concreta el recorrido del bucle de *iIndex* (que es el que contiene a *jIndex*) se da por finalizada la organización del vector y se sale de esta subrutina. También cabe aclarar que esta subrutina tiene al comienzo una sección en la que se indican condiciones inciales.

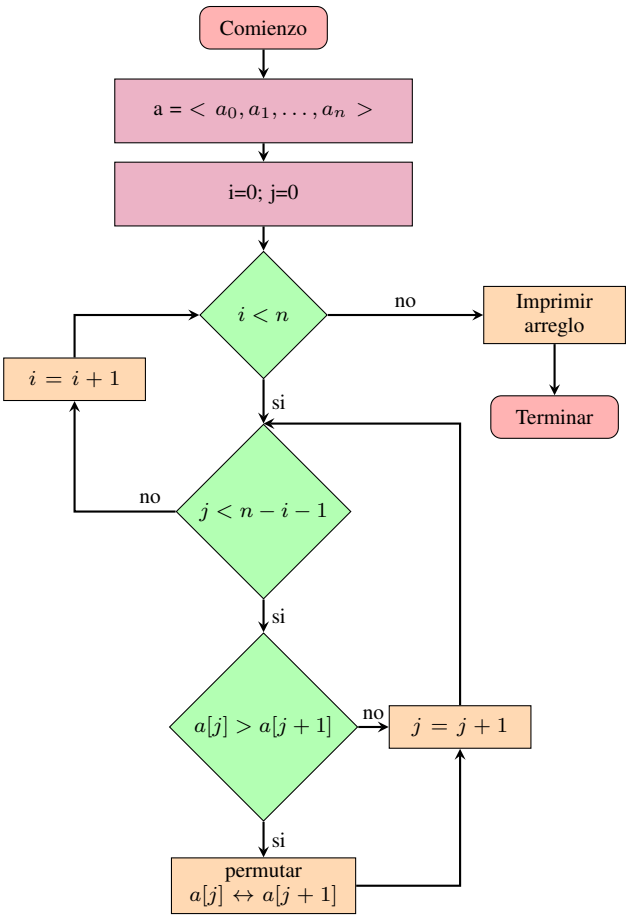


Fig. 1. Diagrama de flujo de Bubble Sort

I-C. Memoria ocupada

Para poder funcionar, el programa ocupa 122 instrucciones en total, contenidas en las direcciones **0x0100** hasta **0x0179** (en hexadecimal); de las cuales 9 son empleadas para variables (memoria de datos).

II. Resultados de instrucciones

Para analizar el rendimiento o eficacia del programa una metrica que se puede emplear es la cantidad de instrucciones que le lleva al programa efectuar la organización de los datos, y esta misma es la forma en que se midió el rendimiento. Como se explicó previamente en la sección I se espera que la cantidad de instrucciones para cada vector sea aproximadamente la misma, independientemente de los datos de los vectores *(suponiendo vectores de mismas dimensiones)*, aunque primero se analizará como son los resultados relativos. Imaginemos que tenemos un vector ya **ordenado**, entonces nunca se entraría en el bucle de permutar los datos, por lo cual este sería el caso con menos instrucciones. Ahora si imaginamos un vector con todos los datos desordenados (vector **invertido**), es decir que habría que permutarlos a todos, en este caso se ejecutarían la máxima cantidad de instrucciones, por lo que sería el caso con más instrucciones. Y por último (caso más típico) si tenemos un vector con sólo algunos datos ordenados (vector **desordenado**), llevaría más instrucciones que el primer caso, pero menos que el segundo. Y esto mismo ocurrió en el resultado de la tabla I.

	Ordenado	Desordenado	Invertido
Solo ordenamiento	534	567	597
Programa entero	853	886	916

TABLA I. Instrucciones para organizar cada vector

Los vectores empleados para realizar la tabla I son los siguientes:

Ordenado	Desordenado	Invertido
<-3,-2,-1,0,1,2,3>	<0,-2,1,3,-1,2,-3>	<3,2,1,0,-1,-2,-3>

TABLA II. Vectores empleados para en análisis

III. Conclusiones

Se concluye que el algoritmo funciona como se esperaba, y que la ventaja del mismo es que no depende tanto de los datos, sino que lo hace más de su dimensión. Además se debe tener en cuenta que si bien la dimensión de los vectores es 7 y que las instrucciones medidas fueron de media 566 (es decir que uno esperaría que fueran 7^2) no significa que esté mal, sino que es el conjunto de instrucciones que permutan los datos, los que son ejecutados 49 veces, y que en la arquitectura MARIE se está limitado a hacer ciertas tareas básicas, como alojar datos en una variable o redirijir el flujo del programa, con varias instrucciones, por esto mismo es que le lleva tantos ciclos de reloj a la arquitectura completar la tarea.

Listing 1. Fragmento del código *"69854_1_0.asm"*

```
/ es un bucle 'while' que itera cada posicion del vector de datos
/ y los va ordenando en la misma posicion de memoria
/ -> Esto replica tal cual el diagrama de flujo provisto
BubbleSortInit, Load  Zero
                Store iIndex / i = 0
                Store jIndex / i = 0

                / DisplayRtrnPath++
                Load    DisplayRtrnPath
                Add     One
                Store   DisplayRtrnPath

                / i < n ?
                / si -> ir a (j<n-i-1?)
                / no -> terminar bucle
StartILoop,     LoadI     DataLengthPtr
                Subt      iIndex
                Skipcond  800
                Jump      SortEnded

                / j = 0
                Load      Zero
                Store     jIndex

                / j < n-i-1 ?
                / si -> ver si permutar datos
                / no -> i++
AfterStartILoop,LoadI     DataLengthPtr
                Subt      iIndex
                Subt      One
                Subt      jIndex
                Skipcond  800
                Jump      IncrementI

                / Compruebo si se tienen que permutar los datos
                / para eso debo guardo temporalemente el dato a[j]
                / y la direccion de a[j] (&a[j])
                Load      DataPtr
                Add       jIndex
                Store     aJAddr

                / incremento la direccion &a[j+1]
                Add       One
                Store     aJ1Addr

                / temp = a[j]
                LoadI     aJAddr
                Store     aJData

                / a[j] > a[j+1] ?
                / si -> permutar
                / no -> j++
                LoadI     aJ1Addr     / a[j+1]
                Subt      aJData      / a[j]
                Skipcond  000
                Jump      IncrementJ

                / Se permutan los datos

                / almaceno en &a[j] el dato de a[j+1]
                LoadI     aJ1Addr
                StoreI    aJAddr

                / almaceno en &a[j+1] el dato temporal (antiguo a[j])
                Load      aJData
                StoreI    aJ1Addr

                / j++
IncrementJ,     Load      jIndex
                Add       One
                Store     jIndex

                Jump      AfterStartILoop


                / i++
IncrementI,     Load  iIndex
                Add   One
                Store iIndex
                Jump  StartILoop


/ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Fig. 2. Segmento del código que contiene la subrutina de ordenamiento