Integrador TP N° 1

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

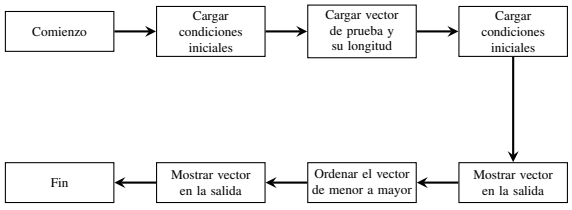
I. Algoritmo elegido

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. 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 I-B en sí se realizó en la subrutina I, 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 tanto.

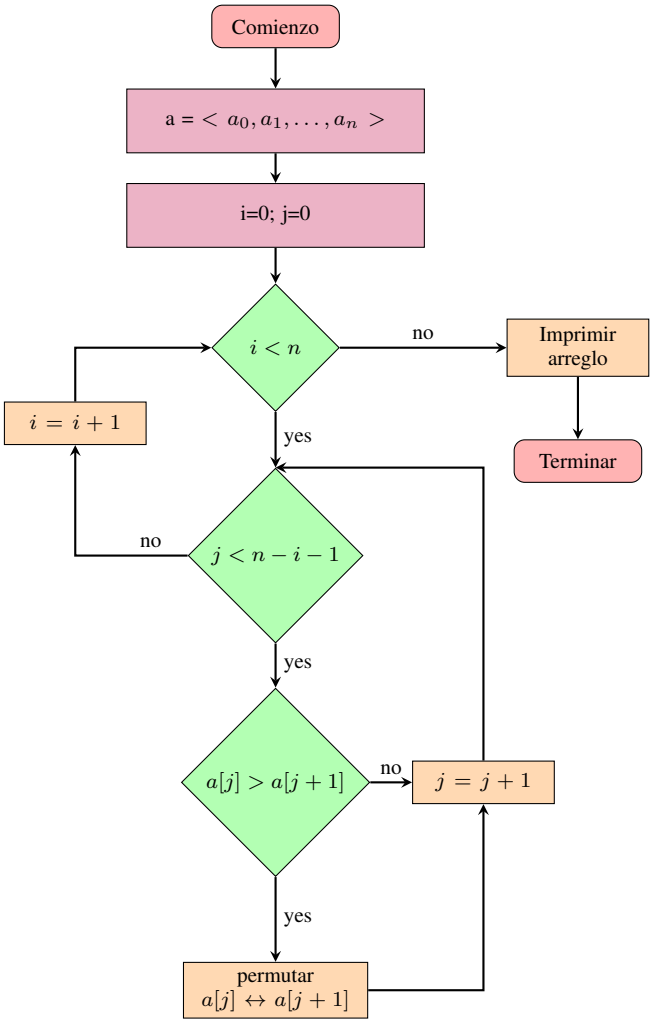
A continuación se presenta un diagrama que explica mejor el flujo general del programa.



I-B. Algoritmo implementado

El algoritmo de Bubble Sort implementado se hizo a partir del diagrama de flujo I-B, 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 al que se indican condiciones inciales.

Fig. 1. Diagrama de flujo de Bubble Sort



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

```
/ ------------------------------------------------------------
/ Entry Point del algoritmo de ordenamiento
/ 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


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