Proyecto Integrador Nº1

Ignacio Nahuel Chantiri 69869/1

Facultad de Ingeniería de la Universidad Nacional de La Plata

1. Introducción

El siguiente informe documenta la selección e implementación de un algoritmo de ordenamiento (en orden creciente) de vectores numéricos, a ser desarrollado en arquitectura MARIE; acompañado de un estudio de performance y una comparación de los distintos tipos de soluciones para el problema de ordenamiento.

2. Consideraciones para la selección adecuada del Algoritmo

Se tuvieron en cuenta distintos parámetros para la selección del algoritmo:

- **Complejidad de implementación en MARIE:** El desarrollo de cualquier algoritmo, en general, es más complejo cuando se realiza en lenguaje ensamblador. A esto se suma que MARIE cuenta con un repertorio de instrucciones más limitado que otras arquitecturas, haciendo aún más compleja la implementación de algunos algoritmos, por ejemplo, los que hacen uso de una pila, puesto que no se dispone de instrucciones nativas para su manejo.
- **Memoria utilizada:** MARIE cuenta con una memoria de instrucciones de 4096 celdas. Aunque pueda parecer una cantidad considerable, planteese la siguiente situación:
  Los algoritmos más complejos, como el Quick Sort, son superadores en eficiencia mientras más largo sea el vector a ordenar.
  Al mismo tiempo, mientras más largo es el vector a ordenar, más espacio de memoria se ocupa no solo por el vector mismo, sino por el agrandamiento de la pila (que en cierto momento será igual de grande que el vector).
  Se forma entonces un dilema: *Mientras más grande el vector, más eficiente el algoritmo, pero si el vector es muy grande, quizá no se pueda implementar el mismo algoritmo debido al gran espacio que ocuparía*.
- **Eficiencia:** MARIE cuenta con un solo registro (acumulador) para realizar comparaciones o aritmética. Esto significa que se requiere buscar uno de los argumentos en memoria, lo cual es ineficiente en el sentido de la cantidad de ciclos de reloj que requiere.
  En algunos algoritmos simples, es predecible la cantidad de accesos de memoria que se ejecutarán, permitiendo optimizarlos en ese sentido.

A continuación se desarrollan varios puntos a favor y en contra de los principales algoritmos, que llevaron a la decisión final, junto con una breve descripción de los mismos:

2.1 Bubble Sort

Compara dos elementos consecutivos del vector y los ordena. Repite iterativamente con los dos elementos siguientes hasta que el vector esté completamente ordenado.
La mayor ventaja es su facilidad de implementación. Al hacer uso de solo dos punteros, no requiere prácticamente espacio de memoria extra para una pila. Además, para ordenar vectores cortos, se desempeña de manera similar que otros algoritmos considerados más eficientes.

2.2 Selection Sort

Este algoritmo busca el menor elemento y lo ubica al comienzo, iterativamente.
Otro algoritmo de fácil implementación y de velocidad (su desventaja) similar al Bubble Sort. Tampoco ocupa una cantidad considerable de memoria.

2.3 Insertion Sort

Inserta cada nuevo elemento en la posición correcta dentro de una parte del vector ya ordenada.
Su implementación es relativamente más dificultosa que los algoritmos anteriores, aunque no muy compleja. No ocupa una cantidad considerable de memoria. Se descartó por no ofrecer una velocidad mayor que otras opciones más simples.

2.4 Merge Sort

Divide el vector a la mitad, recursivamente (problema). Requiere un espacio extra igual al vector original, haciéndolo mala opción para vectores grandes.
Utiliza recursividad, lo que torna dificultosa su implementación. Las ventajas podrían compensar estas desventajas, pues es muy eficiente (en velocidad) aún con vectores muy grandes.

2.5 Quick Sort

Toma un pivote, y divide entre números mayores o menores a él, repitiendo de manera recursiva para cada sub-vector resultante.
Otro algoritmo recursivo muy eficiente para vectores largos, parecido al Merge Sort, pero que también requiere mucho espacio y utilización de pila.

3. Algoritmo elegido: Bubble Sort

Se decidió implementar el algoritmo **Bubble Sort**.

Como el objetivo principal es ordenar un vector de 7 elementos, no se aprovecha la capacidad de los algoritmos más complejos de ordenar vectores largos eficientemente. En la siguiente tabla se compara el algoritmo elegido junto con uno más complejo:

Caso	**Bubble Sort**	**Quick Sort**
Mejor	$O(n)$	$O(n \log_2 n)$
Promedio	$O(n^2)$	$O(n \log_2 n)$
Peor	$O(n^2)$	$O(n^2)$
Operaciones estimadas para n=7)		
Caso	**Bubble Sort**	**Quick Sort**
Mejor	7	$7 \cdot \log_2 7 \approx 19{,}65$
Promedio	49	$7 \cdot \log_2 7 \approx 19{,}65$
Peor	49	49

En la última sección de la tabla se observan los resultados para el vector de 7 elementos que se requiere ordenar.
Para el mejor caso de cada uno, el Bubble Sort realiza menos de la mitad de operaciones que el Quick Sort.
Para el caso promedio, el Quick Sort realiza menos de la mitad de operaciones que el Bubble Sort.
Para el peor caso, realizan la mismas operaciones.

Es importante tener en cuenta que el peor caso para cada algoritmo se da bajo distintas condiciones: para el Quick Sort, el peor caso depende de la correcta elección del pivote, y no necesariamente de que tan ordenado está el vector orignalmente, mientras que para el Bubble Sort el vector invertido es el peor caso; por lo que estas conclusiones son estimadas.

En resumen, existe un comportamiento muy parecido entre ambos algoritmos para el caso n = 7, por lo que no vale la pena implementar un algoritmo más complejo para obtener casi el mismo resultado.
Si nos valiésemos más por lo que sucede en el caso promedio, quizá Quick Sort sea la mejor implementación. Pero para el caso requerido, en el que hay que ordenar un vector ordenado, uno invertido, y uno random, el resultado promedio es el similar independientemente del algoritmo.

Por último, el desarrollo en lenguaje ensamblador fué muy rápido para el Bubble Sort en comparación con el del Quick Sort.

4. Desarrolo del código en detalle

4.1 Esquema general

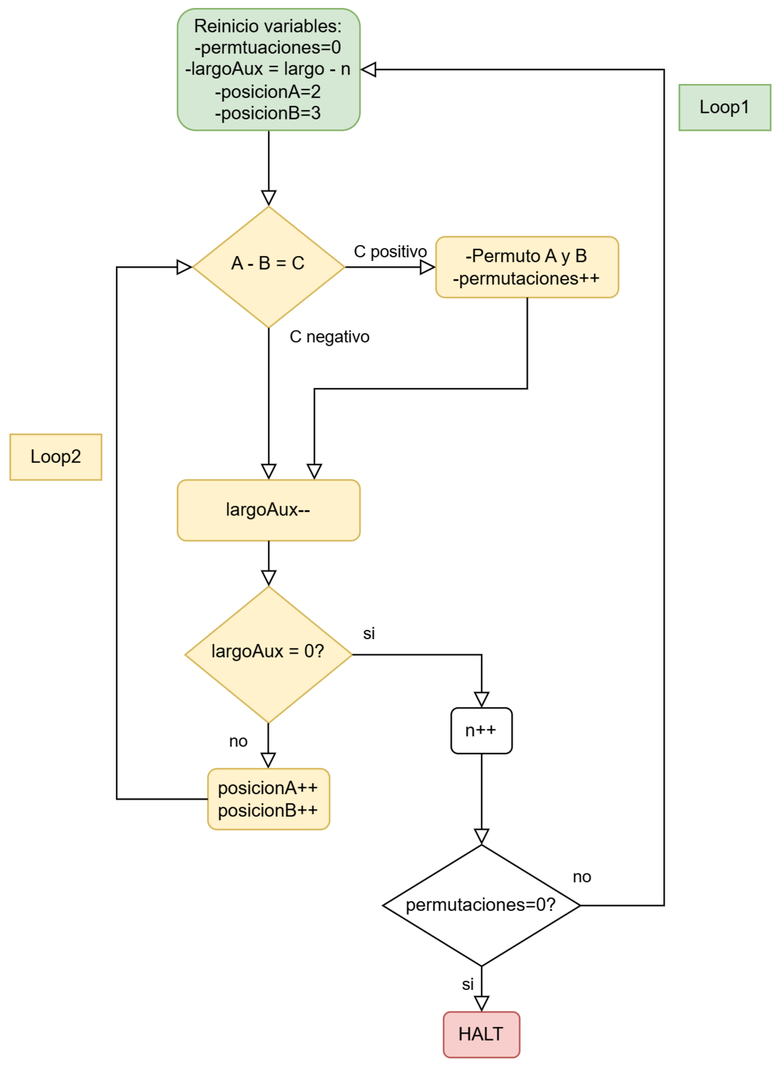

Figura 1: *Esquema general del código implementado para el Bubble Sort*

4.2 Pseudo-código

La implementación se reduce al siguiente pseudo-código:

```
largo = largo del vector
n = 1

Loop1{
     permutaciones = 0
     largoAux = largo - n
     punteroA = 0
     punteroB = 0

     Loop2{
          Comparo A con B
          Permuto si es necesario
          permutaciones ++
          punteroA ++
          punteroB ++
          largoAux --
          if largoAux = 0, salgo de Loop2
          }
    n ++
    if permutaciones = 0, salgo de Loop1
     }
Halt
```

4.3 Explicación general

El código es relativamente simple: Los elementos en **negrita** (A y B respectivamente) son comparados, y si es necesario, permutados. La comparación se explica en detalle en la **sección 4.2**.

3, **2**, 1, 0, -1, -2, -3

Luego se aumentan los punteros de A y B, seleccionando los dos elementos siguientes, y se repite la operación sucesivamente hasta recorrer el vector completo.
La variable largoAux hace de flag para detectar si ya se recorrió el vector completo.

2, **3**, **1**, 0, -1, -2, -3

2, 1, **3**, **0**, -1, -2, -3

...

2, 1, 0, -1, -2, -3, **3**

Cuando se detecta que el vector fue recorrido en su totalidad, se verifica si hubo alguna permutación.
Si la hubo, no sabemos con certeza si el vector está ordenado, por lo que recorremos nuevamente (esta vez recorremos hasta el índice *largoAux* - *n*, donde *n* indica la cantidad de veces que se recorrió el vector. Esto se debe a que en cada recorrida, el último elemento ya está en su lugar final, y no hace falta compararlo). Se repite sucesivamente hasta que no haya permutaciones.
Cuando no se detecta ninguna permutación, el algoritmo finaliza y el vector se halla en su posición orginal, ordenado.

4.4 Comparación

La comparación se realiza mediante la resta A - B = C.
Observando el signo de C, se puede determinar si A o B es mayor, sin importar el signo de los elementos, tal que:

Si $C \geq 0$, **NO** se permuta A con B

Si $C < 0$, se permuta A con B

La ventaja de este método es su extrema simplicidad, ya que con una misma operación es posible comparar elementos de cualquier signo, aún si los elementos son iguales, o si alguno (o ambos) son cero.
Sin embargo, se debe tener en cuenta que MARIE permite representar números de hasta 16 bits en Ca2, por lo que habrá ciertos casos en el que se genere overflow.
Por ejemplo, suponer el caso en que se comparan los siguientes números:

A = 32.767 (máximo numero en Ca2 que representa MARIE)

B = -1

La comparación A - B dará como resultado 32.767, que no es representable con 16 bits en Ca2, provocando un overflow. La misma situación sucede siempre que $|A - B| > 32{,}767$, por lo que ningún par de elementos del vector debería exceder esa condición para asegurar el correcto funcinoamiento del algoritmo.
El código implementado no considera estas situaciones. MARIE tampoco cuenta con un registro de estados para verificar si sucedió un overflow, aunque el simulador web sí.

5. Especificaciones

5.1 Memoria utlizada

El código completo, considerando memoria de programa y datos (sin incluir el espacio ocupado por el vector, la celda que indica su largo, y el jump que llama a la función) ocupa **48** direcciones de memoria (39 de programa y 9 de datos) de las 4096 disponibles, esto es **el 1.17 % del total de la memoria**, o expresado en bytes, **96 bytes**.

La memoria utilizada se midió mediante el recuento manual de las líneas del programa.

5.2 Performance

Se detallan la candidad de instrucciones ejecutadas para tres casos distintos, según el grado de orden del vector:

- **Vector ordenado (mejor caso): 106 instrucciones, 713 ciclos de reloj**
- **Vector invertido (peor caso): 527 instrucciones, 3739 ciclos de reloj**
- **Vector desordenado (caso promedio): 477 instrucciones, 3299 ciclos de reloj**

Para la medición de ciclos e instrucciones se utilizó la versión modificada del simulador web de MARIE que provee la cátedra.

6. Conclusiones

El trabajo de investigación sobre los distintos algoritmos de ordenamiento proporciona no solo un entendimiento sobre las capacidades de la arquitectura utilizada, sino también una apreciación por la complejidad de implementar algoritmos en sistemas con recursos limitados.

Al intentar implementar varios de los algoritmos, se evidenciaron las características de MARIE como arquitectura, que, usualmente, favorecen soluciones más simples.

Las especificaciones del proyecto, que implicaban la ordenación de tres vectores específicos, destacan la importancia de un enfoque con mayor criterio: el algoritmo más eficiente no siempre es el mejor cuando se considera el contexto real del hardware y las restricciones. La eficiencia no solo se mide en términos de tiempo de ejecución, sino también en el uso de los recursos disponibles, como la memoria y el acceso a registros.