Martín Aburto

Sandy Pérez

TALLER 2 ACI

Ejercicio 1:

El proceso para crear el algoritmo para encontrar el valor máximo en un arreglo de 20 enteros no fue tan complicado, algo que nos ayudó bastante fue pensar en cómo se vería el mismo código en otro lenguaje de programación, como C, a partir de ahí era mucho mas fácil convertirlo a MIPS. Quizás la parte más complicada de este punto era el hecho de que los resultados debían guardarse en las posiciones anteriores al primer elemento del arreglo, esto dio muchos errores antes de finalmente encontrar la solución correcta.

El punto B se trataba de ordenar un arreglo mediante una especie de insertionSort en donde buscábamos el valor máximo del arreglo, lo copiábamos en un arreglo nuevo y borrábamos el valor del arreglo original. Este algoritmo resultó sencillo porque reutilizamos el código que ya estaba hecho del punto anterior, aún así tuvimos algunos problemas a la hora de mostrar el arreglo por consola.

El punto C sin duda fue el ejercicio mas difícil de todos desde nuestro punto de vista, ya que el BubbleSort requiere de la utilización de bucles anidados y por consecuencia mucho mayor control de los índices en el arreglo, ya que es muy fácil pasarse o hacer bucles infinitos. Además, como sigue una estructura un poco más compleja, es más difícil identificar los puntos donde el programa falla y encontrarle una solución.

ANÁLISIS DE COMPLEJIDAD:

Complejidad del algoritmo del punto b:

- La inicialización de los recursos consume 6 instrucciones en total.

- Para ordenar el arreglo se necesitan = 6 + (29 * 20) * 20 – 7, puesto que se recorren las 20 posiciones del arreglo 20 veces para encontrar todos los valores.

- La función guardar toma en total 14 instrucciones en ejecutarse.

- Luego, para mostrar el arreglo ordenado se utilizan = (14 * 20) + 4 instrucciones.

Por lo tanto, para ejecutar el programa con el arreglo dado, se requieren un total

de = *11,897 instrucciones*.

Para ordenar un vector que esté completamente ordenado como el siguiente: [20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

El programa debería hacer algunas iteraciones extra, en total las instrucciones requeridas para ordenar y mostrar el arreglo ascenderían a = 6 + (37 * 20) – 1.

O sea que, sumando el resto de las instrucciones, quedarían un total de = 817 instrucciones necesarias para ordenar un arreglo de 20 elementos de mayor a menor en el peor caso.

Para ordenar un vector de 100 elementos (en el peor caso) se necesitarían aproximadamente = *4,085 instrucciones*.

Para un vector de 1000 elementos = *40, 850 instrucciones*.

Para un vector de 1,000,000 elementos = *40,850,000 instrucciones*.

Complejidad del algoritmo del punto b:

La inicialización de todas las variables en el Main requiere un total de *5* instrucciones.

El bucle interno se ejecuta como máximo 20 veces, pero no hay que olvidar que este bucle esta anidado dentro de un bucle externo que también se ejecuta 20 veces y contiene 4 instrucciones. Por todo lo antes mencionado, la cantidad de instrucciones que se requieren es de: (20 * 4) * (20 * 19) = *30,400 instrucciones*.

Luego, para mostrar el arreglo por consola se necesita la cantidad de:

4 + (14 * 20) = *284 instrucciones*.

Lo que da un total de 30,691 instrucciones en total.

Al igual que en el anterior ejercicio, la complejidad que tiene el algoritmo depende de la longitud del arreglo, por ejemplo:

Si el arreglo tuviera una longitud de 100 elementos, la complejidad sería de:

153,455 instrucciones.

Si tuviera 1000 elementos, la complejidad sería de: 1,534,550 instrucciones.

Si tuviera 1,000,000 de elementos: 1,534,550,000.

Pero todos esos cálculos serían tomando en cuenta el peor caso del BubbleSort, que es cuando todos los elementos están desordenados (que de hecho es el caso más frecuente). En caso de que estemos ante el mejor caso, que es cuando todos los elementos del arreglo están ordenados se acortaría enormemente la cantidad de instrucciones requeridas para el programa.

Cuando el arreglo está totalmente ordenado, el cálculo para la complejidad del bucle anidado pasa de (20 * 4) * (20 * 19) a (20 * 4) * (20 * 11) lo que da un

total de = *17,600 instrucciones*.

Entonces, en el mejor de los casos, este algoritmo tiene una complejidad

de = 17,889 instrucciones.

En conclusión, el primer algoritmo de ordenamiento resultó ser mas rápido que el bubbleSort en términos de tiempo, pero también cabe recalcar que la utilización de memoria del primer algoritmo es mucho mas alta, ya que implica la creación de un nuevo arreglo con la misma longitud del arreglo original.

Recursividad y stack

Esta parte del trabajo fue bastante confusa, pero en comparación a las demás, no resultó tan complicada. Utilizamos el stack para hacer llamadas recursivas y poder pasarle “argumentos” a la función.

Debido a que el tamaño del stack en MIPS es limitado, no se pueden hacer demasiadas llamadas recursivas y teniendo en cuenta que sacar la factorial requiere un uso exhaustivo de la recursión, el resultado es que es imposible sacar la factorial de un número pequeño como el 20. De hecho, el máximo numero al cuál pudimos calcularle la factorial fue el 16, a partir de ahí los resultados eran muy erróneos.

CONCLUSIÓN:

La experiencia de programar en MIPS fue bastante interesante, ya que estas obligado a aprender como funciona la arquitectura para poder programar correctamente , además al ser un lenguaje de muy bajo nivel, las instrucciones son muy extensas y se requiere de mucha escritura para poder completar algún algoritmo, esto también tiene su parte positiva, ya que la eficiencia de los programas creados en este lenguaje es realmente muy buena, ya que no hay un compilador tan automático como en lenguajes de más alto nivel, eso sí, a costa de que programar en MIPS sea mucho más complicado.