

“Conjunto de instrucciones MIPS”

Federico Colangelo, Padrón Nro. 89.869
federico.colangelo@semperti.com

Facundo Rossi, Padrón Nro. 86.707
frossi85@gmail.com

Federico Martín Rossi, Padrón Nro. 92.086
federicomrossi@gmail.com

1er. Cuatrimestre 2013
66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

Índice

| | |
|---|-----------|
| 1. Introducción | 1 |
| 2. Compilación | 1 |
| 3. Utilización | 1 |
| 3.1. Implementación en C | 1 |
| 3.2. Implementación en Assembly | 1 |
| 4. Implementaciones | 1 |
| 4.1. Implementación en C | 2 |
| 4.1.1. Algoritmo <i>Bubblesort</i> | 2 |
| 4.1.2. Algoritmo <i>Shellsort</i> | 3 |
| 4.2. Implementación en Assembly | 5 |
| 4.2.1. Algoritmo <i>Shellsort</i> | 5 |
| 5. Debugging | 6 |
| 6. Pruebas | 7 |
| 7. Conclusiones | 8 |
| Appendices | 9 |
| A. Implementación completa en lenguaje C | 9 |
| A.1. <i>tp0.c</i> . Implementación del main del programa | 9 |
| A.2. <i>bubblesort.h</i> . Declaración de la librería Bubblesort | 13 |
| A.3. <i>bubblesort.c</i> . Definición de la librería Bubblesort | 14 |
| A.4. <i>shellsort.h</i> . Declaración de la librería Shellsort | 15 |
| A.5. <i>shellsort.c</i> . Definición de la librería Shellsort | 15 |
| A.6. <i>fileloader.h</i> . Declaración de la librería File Loader | 16 |
| A.7. <i>fileloader.c</i> . Definición de la librería File Loader | 17 |
| B. Implementación completa de Shellsort en assembly MIPS | 19 |
| B.1. <i>tp0.c</i> . Implementación del main del programa | 19 |
| B.2. <i>shellsort.S</i> . Implementación del algoritmo Shellsort | 20 |
| B.3. <i>compare.S</i> . Implementación del comparador de palabras | 21 |
| B.4. <i>swap.S</i> . Implementación del comparador de palabras | 23 |

1. Introducción

En este trabajo presentamos la comparación entre dos algoritmos de ordenamiento: el *Bubblesort* y el *Shell-sort*.

Para realizar la comparación entre ambos se realizó la implementación en lenguaje C de cada uno. A su vez, para comparar la performance entre código de alto nivel y código nativo, se hizo la implementación de shellsort en assembler MIPS, con el fin de poder comparar los tiempos de ejecución de ambos programas.

Todo el trabajo se realizó en una plataforma *NetBSD/MIPS-32* mediante el *GXEmul* [1]

Todos los archivos y códigos fuente aquí mencionados, así como también el presente informe, pueden ser descargados de la sección Downloads del repositorio del grupo¹.

2. Compilación

La herramienta para compilar tanto el código assembly como C será el *GCC* [2]. Para tratar de equiparar al máximo el *S* generado por ambas implementaciones, se utilizará el flag de gcc “-O0” para que no realice optimizaciones sobre el código en lenguaje C.

Para automatizar las tareas de compilación se hace uso de la herramienta *GNU Make*. Los Makefiles utilizados para la compilación se incluyen junto al resto de los archivos fuentes del presente trabajo².

3. Utilización

En los siguientes apartados se especifica la forma en la que deben ser ejecutados los programas implementados tanto en C como en assembly.

3.1. Implementación en C

El resultado de compilación con “make” será un programa ejecutable, de nombre *tp0*, que podrá ser invocado con los siguientes parámetros:

- *-h*: Imprime ayuda para la utilización del programa;
- *-V*: Imprimir la versión actual del programa;
- *-b [ARGS]*: El programa recibe nombres de archivos de texto o strings ingresados por *stdin*, ordenandolos utilizando el algoritmo bubblesort. Para utilizar *stdin* deberá omitirse [ARGS] y luego introducir las palabras;
- *-s [ARGS]*: El programa recibe nombres de archivos de texto o strings ingresados por *stdin*, ordenandolos utilizando el algoritmo shellsort. Para utilizar *stdin* deberá omitirse [ARGS] y luego introducir las palabras.

3.2. Implementación en Assembly

El resultado de compilación con “make” será un programa ejecutable, de nombre *tp0*, el cual aceptará un archivo de texto como argumento y lo ordenará con el algoritmo Shellsort.

4. Implementaciones

En lo que sigue de la sección, se presentarán los códigos fuente de las implementaciones de los algoritmos. Aquellos lectores interesados en la implementación completa de los dos programas, pueden dirigirse a los apéndices ubicados al final del presente informe. Recordamos que se han separado las implementaciones en dos de manera de poder mantener un orden entre ambos.

¹URI del Repositorio: <https://code.google.com/p/tp-orga-computadoras/>

²Los archivos se encuentran separados según la implementación a la que pertenecen, por lo que habrán dos Makefiles distintos, uno para la implementación en lenguaje C y otro para la implementación en assembly

4.1. Implementación en C

La implementación del programa fue dividida en los siguientes módulos:

- **tp0**: Programa principal responsable de interpretar los comandos pasados por la terminal de modo que realice las tareas solicitadas por el usuario. Su principal función es encadenar el funcionamiento de los otros módulos y mostrar por pantalla el resultado obtenido;
- **fileloader**: Módulo encargado de levantar un archivo de texto desde el filesystem y convertirlo en un array donde cada posición es una palabra. Permite además dimensionar el tamaño en memoria necesario para cargar todo el archivo como un arreglo de palabras;
- **bubblesort**: Módulo encargado de levantar un archivo de texto desde el filesystem y convertirlo en un array donde cada posición es una palabra. Permite además dimensionar el tamaño en memoria necesario para cargar todo el archivo como un arreglo de palabras;
- **shellsort**: Módulo encargado de implementar el algoritmo de shellsort. Recibe como parámetros un arreglo de palabras desordenado y el tamaño del mismo. Como resultado devuelve dicho arreglo ordenado.

4.1.1. Algoritmo *Bubblesort*

En el *Código 1* se muestra el header de la librería, donde se declara la función Bubblesort, mientras que en el *Código 2* se muestra la definición de la librería.

Código 1: “*bubblesort.h*”

```
1 /* *****
2 * *****
3 *
4 * LIBRERIA BUBBLESORT
5 *
6 * *****
7 * *****/
8
9
10
11 #ifndef BUBBLESORT_H
12 #define BUBBLESORT_H
13
14
15
16 // Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
17 // arreglo de palabras.
18 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
19 // es el tamaño de dicho arreglo.
20 // POST: el arreglo 'words' queda ordenado.
21 void bubblesort(char* words[], int arraysize);
22
23
24
25 #endif
```

Código 2: “*bubblesort.c*”

```
1 /* *****
2 * *****
3 *
4 * LIBRERIA BUBBLESORT
5 *
6 * *****
7 * *****/
8
9
10 #include "bubblesort.h"
11 #include <stdbool.h>
12 #include <string.h>
13
14
```

```

15
16 // Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
17 // arreglo de palabras.
18 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
19 // es el tamaño de dicho arreglo.
20 // POST: el arreglo 'words' queda ordenado.
21 void bubblesort(char* words[], int arraysize)
22 {
23     // Variables de procesamiento
24     bool huboIntercambio;
25     int i;
26     int n = arraysize;
27     char* sAux;
28
29     // Recorremos el arreglo haciendo intercambios hasta que ya no se registre
30     // ningun cambio realizado.
31     do
32     {
33         huboIntercambio = false;
34
35         for(i = 1; i < n; i++)
36         {
37             // Si el de indice menor es mayor que el de indice superior, los
38             // intercambiamos
39             if(strcasecmp(words[i-1], words[i]) > 0)
40             {
41                 sAux = words[i-1];
42                 words[i-1] = words[i];
43                 words[i] = sAux;
44
45                 // Cambiamos el flag para registrar que hubo un cambio
46                 huboIntercambio = true;
47             }
48         }
49
50         // Como el elemento del indice superior se encuentra ya ordenado una
51         // vez finalizada la pasada, se reduce en uno la cantidad de indices
52         // a iterar en la proxima pasada.
53         n -= 1;
54     } while(huboIntercambio);
55 }
56

```

4.1.2. Algoritmo *Shellsort*

En el *Código 3* se muestra el header de la librería, donde se declara la función Shellsort, mientras que en el *Código 4* se muestra la definición de la librería.

Código 3: “shellsort.h”

```

1 /* *****
2  * *****
3  *
4  * LIBRERIA SHELLSORT
5  *
6  * *****
7  * *****/
8
9
10
11 #ifndef SHELLSORT_H
12 #define SHELLSORT_H
13
14
15
16 // Funcion que aplica el algoritmo de ordenamiento Shellsort para ordenar un
17 // arreglo de palabras.
18 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
19 // es el tamaño de dicho arreglo.
20 // POST: el arreglo 'words' queda ordenado.
21 void shellsort(char* words[], int arraysize);

```

```

22
23
24
25 #endif

```

Código 4: “shellsort.c”

```

1  /* *****
2  * *****
3  *
4  * LIBRERIA SHELLSORT
5  *
6  * *****
7  * *****/
8
9
10 #include "shellsort.h"
11 #include <string.h>
12
13
14
15 // Funcion que aplica el algoritmo de ordenamiento Shellsort para ordenar un
16 // arreglo de palabras.
17 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
18 // es el tamaño de dicho arreglo.
19 // POST: el arreglo 'words' queda ordenado.
20 void shellsort(char* words[], int arraysize)
21 {
22     int intervalo, k, j, i;
23     char* sAux;
24
25     // Tomamos como intervalo el modulo de la mitad del arreglo
26     intervalo = arraysize / 2;
27
28     // Iteramos hasta que el intervalo sea nulo
29     while(intervalo > 0)
30     {
31         for(i = intervalo-1; i < arraysize; i++)
32         {
33             j = i - intervalo;
34
35             // Procesamos hasta caer en un indice fuera de rango del arreglo
36             while(j >= 0)
37             {
38                 k = j + intervalo;
39
40                 // Si el de indice menor es mayor que el de indice superior, los
41                 // intercambiamos
42                 if(strcasecmp(words[j], words[k]) > 0)
43                 {
44                     sAux = words[j];
45                     words[j] = words[k];
46                     words[k] = sAux;
47                 }
48                 else
49                     j = 0;
50
51                 j -= intervalo;
52             }
53         }
54
55         // Tomamos como nuevo intervalo el modulo de la mitad del intervalo
56         // anterior
57         intervalo = intervalo / 2;
58     }
59 }

```


4.2. Implementación en Assembly

La implementación del programa fue dividida en los siguientes módulos:

- **tp0**: Solamente recibe un texto como argumento por linea de comandos y lo imprime ordenandolo mediante shellsort. Esta implementado en C;
- **fileloader**: Se utiliza el mismo de la otra implementación;
- **swap**: Función implementada en assembler para intercambiar el valor de dos registros recibidos como parametros;
- **compare**: Función similar a strcmp de C. Recibe dos argumentos de tipo texto y decide cual es el que precede alfabeticamente.
- **shellsort**: Implementación en assembler del algoritmo de ordenamiento.

4.2.1. Algoritmo *Shellsort*

En el *Código 5* se muestra la implementación en assembly del algoritmo Shellsort.

Código 5: “*shellsort.S*”

```
1 #ifndef USE_MIPS_ASSEMBLY
2 #define USE_MIPS_ASSEMBLY
3 #include <mips/regdef.h>
4 #include <sys/syscall.h>
5
6 .text
7 .align 2
8 .globl shellsort
9 .extern swap
10 .extern compare
11
12
13 shellsort:  subu sp,sp,24    # creo stack frame, es de 24 porque necesito guardar ra
14             sw ra,16(sp)
15             sw $fp,12(sp)
16             sw gp,8(sp)
17             move $fp,sp
18
19             #Guardo los parametros recibidos en fp
20             sw a0,24($fp)   # words = char **
21             sw a1,28($fp)   # size
22
23             #Guardo los parametros cargados en fp a variables temporales
24             lw s6, 24($fp)   #en 24 esta el words
25             lw s7, 28($fp)   #en 28 esta el size
26
27             #Iniciaizo variables locales
28             srl s0, s7, 1    # intervalo = size/2
29             li s1, 0        # k
30             li s2, 0        # j
31             li s3, 0        # i
32             li s4, 0        # sAux
33
34
35             #add v0, s0, zero
36             #j fin
37
38             #while(intervalo > 0)
39 loop1:      bgt s0, zero, initloop2 #branch grater than, salta si s0 es mayor a 0
40             j fin
41
42             #for(i = intervalo-1; i < arraysiz; i++)
43 initloop2:  addi s3, s0, -1    # i = interlvalo - 1
44 loop2:      bge s3, s7, endloop2 # si i >= size fin de for, bge=branch greater than or equal
45             sub s2, s3, s0    #j = i - intervalo;
46
47
48             #while(j >= 0)
49 loop3:      bge s2, zero, loop3task
50             addi s3, s3, 1
51             j loop2
```

```

52
53
54
55
56 loop3task:  add s1, s2, s0    #k = j + intervalo;
57
58     #Para comparar cadenas, la forma mas sencilla es iterar sobre cada caracter de cada cadena, y restarlos. ↵
59     Si el resultado es 0, son
#iguales. Si no, a continuacion, si el resultado es > 0, entonces la primera cadena es mayor de que la otra ↵
    cadena, de lo contrario la segunda cadena es inferior y #habria que intercambiarlas. Si te quedas sin ↵
    alguna de las cadenas antes que el otro, y son iguales hasta llegar a ese punto, la cadena mas corta es ↵
    menor que el que #mas.
60     #CARGO LAS CADENAS A COMPARAR
61     #words[j]
62     ##Guardo en variables auxiliares las direcciones de memoria a intercambiar
63     #words = s6
64     #Guardo i y j por q las voy a modificar y no las quiero perder
65     sw s1, 0($fp)
66     sw s2, 4($fp)
67
68
69
70     sll s1, s1, 2
71     sll s2, s2, 2
72     addu s1, s1, s6
73     addu s2, s2, s6
74
75     #Cargo los strings a comparar en ax que luego compare utiliza para trabajar
76     lw a0, 0(s1)    #OPTIMIZAR CODIGO PARA ISAR MENOS ti
77     lw a1, 0(s2)
78     jal compare    #llamo a compare
79
80     lw s1, 0($fp)    #Cargo el valor de k en stack
81     lw s2, 4($fp)    #Cargo el valor de j en stack
82
83     bltz v0, swapping #swap si compare > 0
84     add s2, zero, zero #j=0
85     j endif
86
87     # poner el valor a utilizar en los registros correctos y llamar a swap
swapping: add a0, s6,0
88     add a1, s1, 0
89     add a2, s2, 0
90     jal swap
91
92
93 endif:    sub s2, s2, s0    #j = j - intervalo;
94     j loop3
95
96
97 endloop2: srl s0, s0, 1    #intervalo = intervalo / 2;
98     j loop1
99
100
101
102     ##Comienzo retorno
103 fin:     move sp, $fp
104     lw gp, 8(sp)
105     lw $fp, 12(sp)
106     lw ra, 16(sp)
107     addiu sp, sp, 24
108     jr ra    #ra = return address es la direccion a la q quiero volver luego de llamar la funcion
109
110 #endif

```

5. Debugging

Para analizar el correcto funcionamiento de los programas se crearon programas adhoc que fueran corroborando el correcto funcionamiento de cada uno de los modulos de manera individual. La idea era simular el concepto de test unitario de lenguajes de alto nivel.

Una vez que todos los módulos funcionaban de la manera esperada, se utilizó el programa principal como test de integración. Finalmente se utilizó la herramienta *Valgrind* para realizar un análisis minucioso del uso de memoria, corrigiendo así las pérdidas de memoria que se presentaban en cada módulo.

6. Pruebas

Para realizar las pruebas de cada algoritmo se procesaron cuatro textos provistos por la cátedra. Estos son “*Alicia en el País de las Maravillas*” (173kB), “*Beowulf*” (220kB), una “*Enciclopedia*” (643kB) y “*Don Quijote*” (2147kB).

A cada ejecución se le midió el tiempo de procesamiento mediante el comando *GNU “time”* [5]. A continuación un ejemplo de su utilización:

```
$ time ./tp0 -b alice.txt
```

Esto imprimirá por pantalla el tiempo de ejecución que fue tabulado para cada caso. En el *Cuadro 1* se muestran los valores de tiempo obtenidos.

| Archivo de texto | bubblesort.c [s] | shellsort.c [s] | shellsort.S [s] |
|------------------|------------------|-----------------|-----------------|
| alice.txt | 571,715 | 2,207 | 1,035 |
| beowulf.txt | 958,527 | 3,418 | 1,500 |
| cyclopedia.txt | >3600 | 12,184 | 6,914 |
| elquijote.txt | >3600 | 48,957 | 33,098 |

Cuadro 1: Tiempos en segundos obtenidos en la ejecución de los distintos algoritmos.

En los graficos que siguen se muestran distintas comparaciones hechas con estos datos obtenidos, los cuales nos llevarán a la conclusión final.

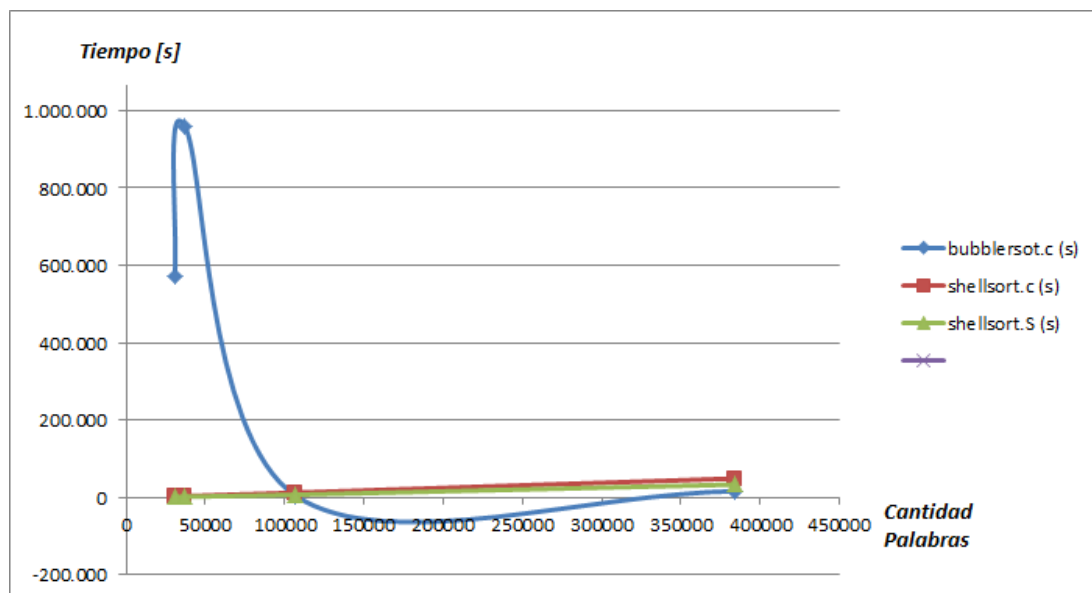


Figura 1: Gráfico de tiempo insumido (en segundos) contra el tamaño de la muestra (cantidad de palabras)

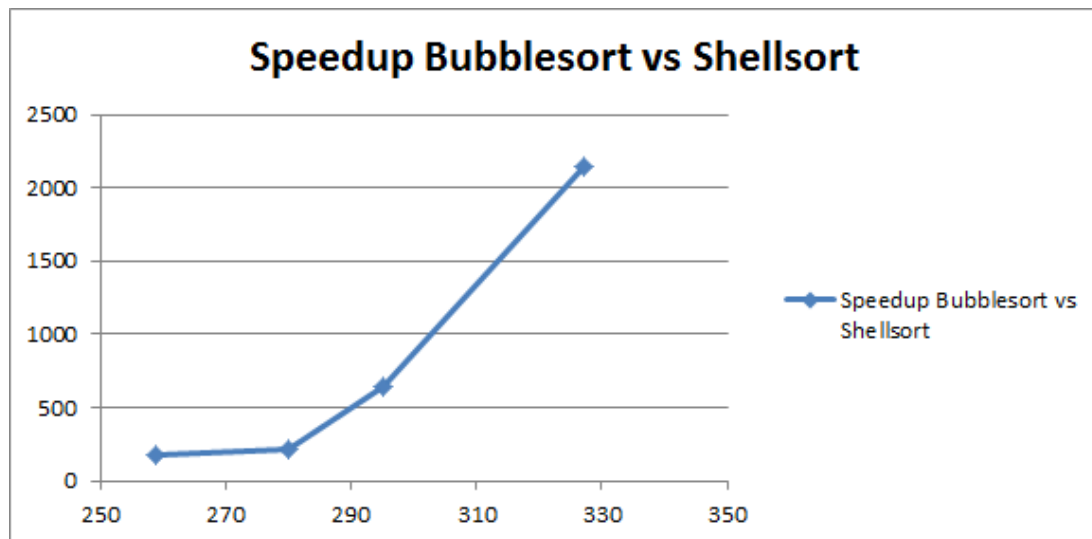


Figura 2: *Gráfico del speedup del Bubblesort contra Shellsort para los diversos tamaños de archivo.*

7. Conclusiones

Al realizar la comparación entre los algoritmos bubblesort y shellsort en C, vemos que la diferencia entre ambos es notable. Estamos hablando de varios ordenes de magnitud.

Esto se debe a que por su código bubblesort tiene un coste de $O(n^2)$ mientras que si bien no podemos determinarlo fácilmente para shellsort, ya que depende mucho del gap elegido, estamos hablando de algo cercano a $O(n \log(n))$. Esto hace que para textos grandes no sea una opción viable la utilización de bubblesort, mientras que shellsort prueba ser muy eficiente y el esfuerzo de codificación no es demasiado grande.

En cuanto a la comparación de assembler y C, notamos una ligera ventaja del primero. Creemos que esto se debe a que muchas de las cosas que el algoritmo en C utiliza como variables en memoria principal, assembler utiliza registros los cuales presentan un tiempo de lecto/escritura notablemente menor.

Referencias

- [1] The NetBSD project, <http://www.netbsd.org/>
- [2] GCC, the GNU Compiler Collection, <http://gcc.gnu.org/>
- [3] Bubblesort, http://en.wikipedia.org/wiki/Bubble_sort
- [4] Shellsort, http://en.wikipedia.org/wiki/Shell_sort
- [5] time man page, <http://unixhelp.ed.ac.uk/CGI/man-cgi?time>
- [6] MIPS ABI, <http://www.sco.com/developers/devspecs/mipsabi.pdf>
- [7] J. L. Hennessy and D. A. Patterson, "Computer Architecture. A Quantitative Approach," 4th Edition, Morgan Kaufmann Publishers, 2000.

Apéndices

A. Implementación completa en lenguaje C

A.1. *tp0.c*. Implementación del main del programa

Código 6: “*tp0.c*”

```
1 /* *****  
2 * PROGRAMA DE ORDENAMIENTO DE PALABRAS  
3 * *****  
4 *  
5 * Facultad de Ingenieria - UBA  
6 * 66.20 Organizacion de computadoras  
7 * Trabajo Practico: Conjunto de instrucciones MIPS  
8 *  
9 * ALUMNOS:  
10 *  
11 * Facundo Rossi (86707) - frossi85@gmail.com  
12 * Federico Colangelo (89869) - federico.colangelo@semperti.com  
13 * Federico Martin Rossi (92086) - federicomrossi@gmail.com  
14 *  
15 * *****  
16 *  
17 * Programa que permite aplicar metodos de ordenamiento a un texto o textos  
18 * especificados por el usuario. Estos textos pueden ser ingresados a traves de  
19 * la entrada estandar o bien a traves de la especificacion de los nombres de  
20 * archivo que se desean procesar. Cabe resaltar que al especificar varios  
21 * nombres de archivos, los metodos seran aplicados al conjunto total de los  
22 * textos contenidos en estos. Es decir, el resultado del ordenamiento sera  
23 * unico ya que contendra la mezcla de palabras de todos los archivos.  
24 * Los metodos de ordenamiento soportados por la presente version del programa  
25 * son:  
26 *  
27 * - Bubblesort  
28 * - Shellsort  
29 *  
30 *  
31 *  
32 * FORMA DE USO  
33 * =====  
34 *  
35 * El programa debera ser ejecutado por consola mediante el siguiente comando:  
36 *  
37 * # ./tp0 [OPCION] [archivos...]  
38 *  
39 * donde, la opcion puede ser:  
40 *  
41 * -h, Muestra la ayuda  
42 * -V, Muestra la informacion de la version  
43 * -b, Indica la utilizacion del algoritmo Bubblesort  
44 * -s, Indica la utilizacion del algoritmo Shellsort  
45 *  
46 * y en archivos se debe especificar los nombres de archivo (incluyendo su  
47 * extension) separados uno del otro por un espacio.  
48 * En caso de desear ingresar el texto a traves de la entrada estandar, debe  
49 * utilizarse la siguiente variacion del comando anterior:  
50 *  
51 * # ./tp0 [OPCION]  
52 *  
53 *  
54 */  
55  
56  
57  
58 #include <ctype.h>  
59 #include <stdio.h>  
60 #include <stdlib.h>  
61 #include <string.h>  
62 #include <unistd.h>
```

```

63 #include "bubblesort.h"
64 #include "shellsort.h"
65 #include "fileloader.h"
66
67
68
69
70
71 /* *****
72  * CONSTANTES
73  * *****/
74
75
76 // Maximo de caracteres permitidos por texto de entrada estandar
77 #define MAX_CHARS 1000
78
79
80
81 /* *****
82  * FUNCIONES AUXILIARES
83  * *****/
84
85
86 // Funcion que imprime por pantalla el mensaje de ayuda del programa.
87 void ayudaImprimirSalidaEstandar()
88 {
89     printf("tp0 [OPTIONS] [files...]\n");
90     printf("-h, \t\t\t display this help and exit.\n");
91     printf("-V, \t\t\t display version information and exit.\n");
92     printf("-b, \t\t\t use the bubblesort algorithm.\n");
93     printf("-s, \t\t\t use the shellsort algorithm.\n\n");
94     printf("$echo -n echo 'El tractorcito rojo que silbo y bufo' > entrada.txt\n");
95     printf("$tp0 -b entrada.txt\n");
96     printf("bufo El que rojo silbo tractorcito y\n\n");
97     printf("cat letters.txt\n");
98     printf("aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ$\n\n");
99     printf("$tp0 letters.txt\n");
100    printf("aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ\n$\n\n");
101    printf("$tp0 letters.txt entrada.txt\n");
102    printf("aAbBcCdDeEfFgGhHiIjJkKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ bufo El que rojo silbo tractorcito y\n$\n");
103 }
104
105
106 // Funcion que imprime por pantalla la version del programa.
107 void versionImprimirSalidaEstandar()
108 {
109     printf("Programa de ordenamiento de palabras\nVersion: v0.1\n");
110 }
111
112
113 // Funcion que imprime los resultados del ordenamiento a traves de la salida
114 // estandar.
115 // PRE: 'palabras' es un arreglo de palabras; 'arraysize' es el tamaño del
116 // arreglo
117 void ordenamientoImprimirSalidaEstandar(char* palabras[], int arraysize)
118 {
119     int i;
120     for(i = 0; i < arraysize; printf("%s ", palabras[i++]));
121     printf("\n");
122 }
123
124
125 // Funcion que se encarga leer archivos de texto devolviendo una cadena con
126 // el contenido de estos.
127 // PRE: 'iniArchivos' es el indice inicial desde donde 'argv' posee nombres
128 // de archivo; 'argc' y 'argv' son los parametros de entrada de la funcion
129 // main.
130 // POST: se devuelve una unica cadena con el contenido completo de archivos.
131 char* cargarTextosDeArchivos(int iniArchivos, int argc, char **argv)
132 {
133     int i;
134     char * texto = "";
135
136     // Sensamos cada archivo ingresado y concatenamos su contenido
137     for(i = iniArchivos; i < argc; i++)
138     {

```

```

139 // Cargamos contenido de archivo
140 char * texto_tmp = file_loader(argv[i]);
141
142 // Recalculamos el tamaño del arreglo
143 int len = strlen(texto) + strlen(texto_tmp) + 2;
144
145 // Solicitamos espacio de memoria nuevo
146 char * texto_alloc = (char *) malloc(len * sizeof(char));
147
148 // Copiamos contenido antiguo y concatenamos el nuevo
149 strcpy(texto_alloc, texto);
150 if(i > iniArchivos) strcat(texto_alloc, " ");
151 strcat(texto_alloc, texto_tmp);
152 free(texto_tmp);
153
154 // Liberamos espacio de memoria antiguo
155 if(i > iniArchivos) free(texto);
156
157 // Asociamos al nuevo arreglo
158 texto = texto_alloc;
159 }
160
161 return texto;
162 }
163
164
165 // Funcion que se encarga de leer el texto ingresado por entrada estandar.
166 // POST: se devuelve una unica cadena con el texto ingresado.
167 char* cargarTextosDeEntradaEstandar()
168 {
169     char * texto = (char *) malloc(MAX_CHARS * sizeof(char) + 1);
170     if(fgets(texto, MAX_CHARS + 1, stdin) == NULL) return "";
171
172     return texto;
173 }
174
175
176 void testFileLoader(const char * url)
177 {
178     char * text;
179     char ** words = NULL;
180     int wordsSize = 0;
181
182     text = file_loader(url);
183
184     if (text != NULL)
185     {
186         //fputs(text, stdout);
187     }
188
189     wordsSize = to_words(text, &words);
190
191     for(int i = 0; i < wordsSize; i++)
192     {
193         fputs("\n", stdout);
194         fputs(words[i], stdout);
195         fputs("\n", stdout);
196     }
197 }
198
199
200
201
202 /* *****
203 * PROGRAMA PRINCIPAL
204 * ***** */
205
206
207 int main(int argc, char **argv)
208 {
209     int c;
210
211     //File loader variables
212     char * text;
213     char ** words = NULL;
214     int wordsSize = 0;

```

```

215
216 opterr = 0;
217
218 while ((c = getopt(argc, argv, "hVb:s:")) != -1)
219 {
220     switch (c)
221     {
222         // Ayuda
223         case 'h':
224             ayudaImprimirSalidaEstandar();
225             break;
226
227         // Version
228         case 'V':
229             versionImprimirSalidaEstandar();
230             break;
231
232         // Ejecucion de bubblesort
233         case 'b':
234
235             // Cargamos contenidos de archivos
236             text = cargarTextosDeArchivos(2, argc, argv);
237
238             // Si el/los archivos esta/n vacio/s, salimos
239             if (text == NULL) break;
240
241             wordsSize = to_words(text, &words);
242
243             if (wordsSize == -1) break;
244
245             // Ejecutamos bubblesort
246             bubblesort(words, wordsSize);
247
248             // Enviamos a salida estandar
249             ordenamientoImprimirSalidaEstandar(words, wordsSize);
250
251             free(words);
252             free(text);
253             break;
254
255         // Ejecucion de shellsort
256         case 's':
257
258             // Cargamos contenidos de archivos
259             text = cargarTextosDeArchivos(2, argc, argv);
260
261             // Si el/los archivos esta/n vacio/s, salimos
262             if (text == NULL) break;
263
264             wordsSize = to_words(text, &words);
265
266             if (wordsSize == -1) break;
267
268             // Ejecutamos shellsort
269             shellsort(words, wordsSize);
270
271             // Enviamos a salida estandar
272             ordenamientoImprimirSalidaEstandar(words, wordsSize);
273
274             free(words);
275             free(text);
276             break;
277
278         // No se especifica nombre de archivo
279         case '?':
280
281             // Ejecutar bubblesort con texto desde entrada estandar
282             if (optopt == 'b')
283             {
284                 // Cargamos texto
285                 text = cargarTextosDeEntradaEstandar();
286
287                 // Si no se ingreso texto, salimos
288                 if (text == NULL) break;
289                 wordsSize = to_words(text, &words);
290

```



```

291     if (wordsSize == -1) break;
292
293     // Ejecutamos bubblesort
294     bubblesort(words, wordsSize);
295
296     // Enviamos a salida estandar
297     ordenamientoImprimirSalidaEstandar(words, wordsSize);
298
299     free(words);
300     free(text);
301     break;
302 }
303
304 // Ejecutar shellsort con texto desde entrada
305 // estandar
306 else if(optopt == 's')
307 {
308     // Cargamos texto
309     text = cargarTextosDeEntradaEstandar();
310
311     // Si no se ingreso texto, salimos
312     if (text == NULL) break;
313     wordsSize = to_words(text, &words);
314
315     if (wordsSize == -1) break;
316
317     // Ejecutamos shellsort
318     shellsort(words, wordsSize);
319
320     // Enviamos a salida estandar
321     ordenamientoImprimirSalidaEstandar(words, wordsSize);
322
323     free(words);
324     free(text);
325     break;
326 }
327
328 default:
329     opterr = 1;
330     break;
331 }
332 }
333
334 // Manejo de casos en que no se ingresan argumentos validos
335 if(opterr || argc <= 1)
336 {
337     fprintf(stderr, "Los argumentos no son validos.\n");
338     printf("Tipee 'tp0 -h' para ver el modo de uso.\n");
339 }
340
341
342 return 0;
343 }

```

A.2. *bubblesort.h*. Declaración de la librería Bubblesort

Código 7: “*bubblesort.h*”

```

1 /* *****
2 * *****
3 *
4 * LIBRERIA BUBBLESORT
5 *
6 * *****
7 * *****/
8
9
10
11 #ifndef BUBBLESORT_H
12 #define BUBBLESORT_H
13

```

```

14
15
16 // Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
17 // arreglo de palabras.
18 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
19 // es el tamaño de dicho arreglo.
20 // POST: el arreglo 'words' queda ordenado.
21 void bubblesort(char* words[], int arraysize);
22
23
24
25 #endif

```

A.3. *bubblesort.c*. Definición de la librería Bubblesort

Código 8: “*bubblesort.c*”

```

1 /* *****
2 * *****
3 *
4 * LIBRERIA BUBBLESORT
5 *
6 * *****
7 * *****/
8
9
10 #include "bubblesort.h"
11 #include <stdbool.h>
12 #include <string.h>
13
14
15
16 // Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
17 // arreglo de palabras.
18 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
19 // es el tamaño de dicho arreglo.
20 // POST: el arreglo 'words' queda ordenado.
21 void bubblesort(char* words[], int arraysize)
22 {
23     // Variables de procesamiento
24     bool huboIntercambio;
25     int i;
26     int n = arraysize;
27     char* sAux;
28
29     // Recorremos el arreglo haciendo intercambios hasta que ya no se registre
30     // ningun cambio realizado.
31     do
32     {
33         huboIntercambio = false;
34
35         for(i = 1; i < n; i++)
36         {
37             // Si el de indice menor es mayor que el de indice superior, los
38             // intercambiamos
39             if(strcasecmp(words[i-1], words[i]) > 0)
40             {
41                 sAux = words[i-1];
42                 words[i-1] = words[i];
43                 words[i] = sAux;
44
45                 // Cambiamos el flag para registrar que hubo un cambio
46                 huboIntercambio = true;
47             }
48         }
49
50         // Como el elemento del indice superior se encuentra ya ordenado una
51         // vez finalizada la pasada, se reduce en uno la cantidad de indices
52         // a iterar en la proxima pasada.
53         n -= 1;
54

```

```

55 } while(huboIntercambio);
56 }

```

A.4. *shellsort.h*. Declaración de la librería Shellsort

Código 9: “*shellsort.h*”

```

1  /* *****
2  * *****
3  *
4  * LIBRERIA SHELLSORT
5  *
6  * *****
7  * *****/
8
9
10
11 #ifndef SHELLSORT_H
12 #define SHELLSORT_H
13
14
15
16 // Funcion que aplica el algoritmo de ordenamiento Shellsort para ordenar un
17 // arreglo de palabras.
18 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
19 // es el tamaño de dicho arreglo.
20 // POST: el arreglo 'words' queda ordenado.
21 void shellsort(char* words[], int arraysize);
22
23
24
25 #endif

```

A.5. *shellsort.c*. Definición de la librería Shellsort

Código 10: “*shellsort.c*”

```

1  /* *****
2  * *****
3  *
4  * LIBRERIA SHELLSORT
5  *
6  * *****
7  * *****/
8
9
10 #include "shellsort.h"
11 #include <string.h>
12
13
14
15 // Funcion que aplica el algoritmo de ordenamiento Shellsort para ordenar un
16 // arreglo de palabras.
17 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
18 // es el tamaño de dicho arreglo.
19 // POST: el arreglo 'words' queda ordenado.
20 void shellsort(char* words[], int arraysize)
21 {
22     int intervalo, k, j, i;
23     char* sAux;
24
25     // Tomamos como intervalo el modulo de la mitad del arreglo
26     intervalo = arraysize / 2;
27

```

```

28 // Iteramos hasta que el intervalo sea nulo
29 while(intervalo > 0)
30 {
31     for(i = intervalo-1; i < arraysize; i++)
32     {
33         j = i - intervalo;
34
35         // Procesamos hasta caer en un indice fuera de rango del arreglo
36         while(j >= 0)
37         {
38             k = j + intervalo;
39
40             // Si el de indice menor es mayor que el de indice superior, los
41             // intercambiamos
42             if(strcasecmp(words[j], words[k]) > 0)
43             {
44                 sAux = words[j];
45                 words[j] = words[k];
46                 words[k] = sAux;
47             }
48             else
49                 j = 0;
50
51             j -= intervalo;
52         }
53     }
54
55     // Tomamos como nuevo intervalo el modulo de la mitad del intervalo
56     // anterior
57     intervalo = intervalo / 2;
58 }
59 }

```

A.6. *fileloader.h*. Declaración de la librería File Loader

Código 11: “*fileloader.h*”

```

1 /* *****
2 * *****
3 *
4 * LIBRERIA DE CARGA DE ARCHIVO DE PALABRAS
5 *
6 * *****
7 * *****/
8
9
10
11 #ifndef FILELOADER_H
12 #define FILELOADER_H
13
14 #define INITIAL_BUFFER_SIZE 100
15
16 #include <stdbool.h>
17
18
19 // Funcion que carga un archivo de texto a memoria
20 // PRE: 'fileUrl' es la url completa al archivo a ordenar.
21 // POST: devuelve la lista de palabras cargadas en memoria
22 char* file_loader(const char * fileUrl);
23
24 // Funcion que separa las palabras de un texto en unidades.
25 // PRE: 'text' es una cadena que contiene el texto; 'result' es un puntero en donde se
26 // insertan las palabras del texto.
27 // POST: en 'result' se almacenaron las palabras.
28 int to_words(char * text, char *** result);
29
30 // Funcion que verifica si un caracter es un fin de palabra o un separador.
31 // PRE: 'letter' es el caracter a procesar.
32 // POST: devuelve true si el caracter es un fin de palabra o separador, o
33 // false en su defecto.

```

```

34 bool isEndOfWord(char letter);
35
36 #endif

```

A.7. *fileloader.c*. Definición de la librería File Loader

Código 12: “*fileloader.c*”

```

1  /* *****
2  * *****
3  *
4  * LIBRERIA DE CARGA DE ARCHIVO DE PALABRAS
5  *
6  * *****
7  * *****/
8
9
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include "fileloader.h"
14
15
16
17
18 // Funcion que carga un archivo de texto a memoria
19 // PRE: 'fileUrl' es la url completa al archivo a ordenar.
20 // POST: devuelve la lista de palabras cargadas en memoria
21 char * file_loader(const char * fileUrl)
22 {
23     FILE * file;
24     long fileSize;
25     char * buffer;
26     size_t result;
27
28     file = fopen (fileUrl, "r");
29
30     if (file==NULL)
31     {
32         //Fail to load file
33         fputs ("The file can not be open or not exist \n", stderr);
34         return NULL;
35     }
36
37     //Get file size
38     fseek (file , 0 , SEEK_END);
39     fileSize = ftell (file);
40     rewind (file);
41
42     //Allocate memory for the whole file
43     buffer = (char *) malloc ((sizeof(char) * fileSize) + 1);
44
45     if (buffer == NULL)
46     {
47         fputs ("There is no enoght memory to allocate the file in memory \n", stderr);
48         return NULL;
49     }
50
51     //Copy the file to the buffer
52     result = fread (buffer, 1, fileSize, file);
53     buffer[(sizeof(char) * fileSize)] = '\0';
54
55     if (result != fileSize)
56     {
57         fputs ("Fail to read the whole file \n", stderr);
58         return NULL;
59     }
60
61     /* the whole file is now loaded in the memory buffer. */
62

```

```

63     fclose (file);
64
65     return buffer;
66 }
67
68
69 // Funcion que separa las palabras de un texto en unidades.
70 // PRE: 'text' es una cadena que contiene el texto; 'result' es un puntero en donde se
71 // insertan las palabras del texto.
72 // POST: en 'result' se almacenaron las palabras.
73 int to_words(char * text, char *** result)
74 {
75     char ** words;
76     char ** reallocatedWords = NULL;
77     bool isNewWord = true;
78     size_t textSize = strlen(text);
79     int wordsIndex = 0;
80     size_t bufferSize = INITIAL_BUFFER_SIZE * sizeof(char*);
81     size_t spaceForWords = INITIAL_BUFFER_SIZE;
82
83     //initially allocate memory for 100 words
84     words = (char **) malloc ( INITIAL_BUFFER_SIZE * sizeof(char*) );
85
86     if(words == NULL)
87     {
88         fputs ("Not enough memory to allocate the words \n", stderr);
89         return -1;
90     }
91
92     for (int i = 0; i < textSize; i++)
93     {
94         if ( !isEndOfWord (text[i]) )
95         {
96             if(isNewWord)
97             {
98                 words[wordsIndex] = &text[i];
99                 isNewWord = false;
100
101                 if(wordsIndex >= spaceForWords-1)
102                 {
103                     //Reallocate for more space and copy the old array to the new one
104                     bufferSize *= 2;
105                     spaceForWords *= 2;
106                     reallocatedWords = realloc(words, bufferSize);
107
108                     if(reallocatedWords != NULL)
109                     {
110                         words = reallocatedWords;
111                         reallocatedWords = NULL;
112                     }
113                     else
114                     {
115                         fputs ("Not enough memory to allocate the words \n", stderr);
116                         return -1;
117                     }
118                 }
119
120                 wordsIndex++;
121             }
122         }
123         else
124         {
125             text[i] = '\0';
126             isNewWord = true;
127         }
128     }
129
130     *result = words;
131
132     return wordsIndex;
133 }
134
135
136 // Funcion que verifica si un caracter es un fin de palabra o un separador.
137 // PRE: 'letter' es el caracter a procesar.
138 // POST: devuelve true si el caracter es un fin de palabra o separador, o

```

```

139 // false en su defecto.
140 bool isEndOfWord(char letter)
141 {
142     //A-Z es de 65 a 90
143     //a-z es de 97 a 122
144     //0-9 es de 48 a 57
145     return !( (letter > 64 && letter < 91) || (letter > 96 && letter < 123) || (letter > 47 && letter < 58));
146 }

```

B. Implementación completa de Shellsort en assembly MIPS

B.1. *tp0.c*. Implementación del main del programa

Código 13: “*tp0.c*”

```

1 /* *****
2  * PROGRAMA DE ORDENAMIENTO DE PALABRAS
3  * *****
4  *
5  * Facultad de Ingenieria - UBA
6  * 66.20 Organizacion de computadoras
7  * Trabajo Practico: Conjunto de instrucciones MIPS
8  *
9  * ALUMNOS:
10 *
11 * Facundo Rossi (86707) - frossi85@gmail.com
12 * Federico Colangelo (89869) - federico.colangelo@semperti.com
13 * Federico Martin Rossi (92086) - federicomrossi@gmail.com
14 *
15 * *****
16 *
17 * Programa que permite aplicar metodos de ordenamiento a un texto o textos
18 * especificados por el usuario. Estos textos pueden ser ingresados a traves de
19 * la entrada estandar o bien a traves de la especificacion de los nombres de
20 * archivo que se desean procesar. Cabe resaltar que al especificar varios
21 * nombres de archivos, los metodos seran aplicados al conjunto total de los
22 * textos contenidos en estos. Es decir, el resultado del ordenamiento sera
23 * unico ya que contendra la mezcla de palabras de todos los archivos.
24 * Los metodos de ordenamiento soportados por la presente version del programa
25 * son:
26 *
27 * - Shellsort
28 *
29 */
30
31 #include <stdio.h>
32 #include <ctype.h>
33 #include <stdlib.h>
34 #include <string.h>
35 #include <unistd.h>
36
37 #include "fileloader.h"
38
39 extern int shellsort(char **, size_t);
40
41 int main (int argc, char** argv)
42 {
43     char* texto;
44     char** words;
45     size_t size;
46     int i;
47
48     texto=file_loader(argv[1]);
49     size=to_words(texto,&words);
50
51     shellsort(words, size);
52     i=0;
53     while (i<size)
54     {

```

```

55         puts(words[i]);
56         i++;
57     }
58
59     return 0;
60 }

```

B.2. *shellsort.S*. Implementación del algoritmo Shellsort

Código 14: “*shellsort.S*”

```

1  #ifndef USE_MIPS_ASSEMBLY
2  #define USE_MIPS_ASSEMBLY
3  #include <mips/regdef.h>
4  #include <sys/syscall.h>
5
6  .text
7  .align 2
8  .globl shellsort
9  .extern swap
10 .extern compare
11
12
13 shellsort:  subu sp,sp,24    # creo stack frame, es de 24 porque necesito guardar ra
14             sw ra,16(sp)
15             sw $fp,12(sp)
16             sw gp,8(sp)
17             move $fp,sp
18
19             #Guardo los parametros recibidos en fp
20             sw a0,24($fp)    # words = char **
21             sw a1,28($fp)    # size
22
23             #Guardo los parametros cargados en fp a variables temporales
24             lw s6, 24($fp)    #en 24 esta el words
25             lw s7, 28($fp)    #en 28 esta el size
26
27             #Iniciaizo variables locales
28             srl s0, s7, 1    # intervalo = size/2
29             li s1, 0        # k
30             li s2, 0        # j
31             li s3, 0        # i
32             li s4, 0        # sAux
33
34
35             #add v0, s0, zero
36             #j fin
37
38             #while(intervalo > 0)
39 loop1:      bgt s0, zero, initloop2 #branch grater than, salta si s0 es mayor a 0
40             j fin
41
42             #for(i = intervalo-1; i < arraysize; i++)
43 initloop2:  addi s3, s0, -1    # i = interlvalo - 1
44 loop2:      bge s3, s7, endloop2 # si i >= size fin de for, bge=branch greater than or equal
45             sub s2, s3, s0    #j = i - intervalo;
46
47
48             #while(j >= 0)
49 loop3:      bge s2, zero, loop3task
50             addi s3, s3, 1
51             j loop2
52
53
54
55
56 loop3task:  add s1, s2, s0    #k = j + intervalo;
57
58             #Para comparar cadenas, la forma mas sencilla es iterar sobre cada caracter de cada cadena, y restarlos. ←
59             Si el resultado es 0, son

```



```

59 #iguales. Si no, a continuacion, si el resultado es> 0, entonces la primera cadena es mayor de que la otra ←
    cadena, de lo contrario la segunda cadena es inferior y #habria que intercambiarlas. Si te quedas sin ←
    alguna de las cadenas antes que el otro, y son iguales hasta llegar a ese punto, la cadena mas corta es ←
    menor que el que #mas.
60 #CARGO LAS CADENAS A COMPARAR
61 #words[j]
62 ##Guardo en variables auxiliares las direcciones de memoria a intercambiar
63 #words = s6
64 #Guardo i y j por q las voy a modificar y no las quiero perder
65 sw s1, 0($fp)
66 sw s2, 4($fp)
67
68
69
70 sll s1, s1, 2
71 sll s2, s2, 2
72 addu s1, s1, s6
73 addu s2, s2, s6
74
75 #Cargo los strings a comparar en ax que luego compare utiliza para trabajar
76 lw a0, 0(s1) #OPTIMIZAR CODIGO PARA ISAR MENOS ti
77 lw a1, 0(s2)
78 jal compare #llamo a compare
79
80 lw s1, 0($fp) #Cargo el valor de k en stack
81 lw s2, 4($fp) #Cargo el valor de j en stack
82
83 bltz v0, swapping #swap si compare > 0
84 add s2, zero, zero #j=0
85 j endif
86
87 # poner el valor a utilizar en los registros correctos y llamar a swap
88 swapping: add a0, s6,0
89 add a1, s1, 0
90 add a2, s2, 0
91 jal swap
92
93 endif: sub s2, s2, s0 #j = j - intervalo;
94 j loop3
95
96
97 endloop2: srl s0, s0, 1 #intervalo = intervalo / 2;
98 j loop1
99
100
101
102 ##Comienzo retorno
103 fin: move sp, $fp
104 lw gp, 8(sp)
105 lw $fp, 12(sp)
106 lw ra, 16(sp)
107 addiu sp, sp, 24
108 jr ra #ra = return address es la direccion a la q quiero volver luego de llamar la funcion
109
110 #endif

```

B.3. *compare.S*. Implementación del comparador de palabras

Código 15: “*compare.S*”

```

1
2 #ifndef USE_MIPS_ASSEMBLY
3 #define USE_MIPS_ASSEMBLY
4 #include <mips/regdef.h>
5 #include <sys/syscall.h>
6
7 .text
8 .align 2
9 .globl compare
10

```

```

11 # t1 = posicion caracter actual en cadena 1
12 # t2 = posicion caracter actual en cadena 2
13
14
15 compare:
16     subu sp,sp,48    # creo stack frame
17     sw $fp,12(sp)
18     sw gp,8(sp)
19     move $fp,sp
20     sw s0,16(sp)
21     sw s1,20(sp)
22     sw s2,24(sp)
23     sw s3,28(sp)
24     sw s4,32(sp)
25     sw s5,36(sp)
26     sw s6,40(sp)
27     sw s7,44(sp)
28
29     add t0, zero, zero # result =0
30     add t1, zero, a0
31     add t2, zero, a1
32
33 nextchar:
34     #Cargo los elementos a comparar en t4 y t5
35     lb t3, 0(t1)
36     lb t4, 0(t2)
37
38     #Voy a pasar todas las mayusculas a minusculas para que compare sea case insensitive como en C
39
40     addi t5, t3, -91
41     bgtz t5, aMinusculas2
42     addi t5, t3, -64
43     bltz t5, aMinusculas2
44     addi t3, t3, 32
45
46 aMinusculas2: addi t5, t4, -91
47     bgtz t5, comparacion
48     addi t5, t4, -64
49     bltz t5, comparacion
50     addi t4, t4, 32
51
52
53     #Comparo caracter para ver si es final del string
54 comparacion: beqz t3, fincadena1
55     beqz t4, fincadena2
56
57 continue: sub t0, t3, t4    # cadena1[i] - cadena2[i]
58     add t1, t1, 1    #NO VA sll x,y,2: por q char es un byte
59     add t2, t2, 1
60
61     beqz t0, nextchar
62     j fin
63
64
65 fincadena1: beqz t4, fin
66     add t0, zero, -1
67     j fin
68
69 fincadena2: add t0, zero, 1
70     j fin
71
72 fin:    ##Comienzo retorno
73     addi v0, t0, 0
74
75     lw s7, 44(sp)
76     lw s6, 40(sp)
77     lw s5, 36(sp)
78     lw s4, 32(sp)
79     lw s3, 28(sp)
80     lw s2, 24(sp)
81     lw s1, 20(sp)
82     lw s0, 16(sp)
83     move sp, $fp
84     lw gp, 8(sp)
85     lw $fp, 12(sp)
86     addiu sp, sp, 48

```

```

87     jr ra      #ra = return address es la direccion a la q quiero volver luego de llamar la funcion
88
89 #endif

```

B.4. *swap.S*. Implementación del comparador de palabras

Código 16: “*swap.S*”

```

1  #ifndef USE_MIPS_ASSEMBLY
2  #define USE_MIPS_ASSEMBLY
3  #include <mips/regdef.h>
4  #include <sys/syscall.h>
5
6  .text
7  .align 2
8  .globl swap
9
10
11
12 swap:
13     subu sp,sp,48    # creo stack frame
14     sw $fp,12(sp)
15     sw gp,8(sp)
16     move $fp,sp
17     sw s0,16(sp)
18     sw s1,20(sp)
19     sw s2,24(sp)
20     sw s3,28(sp)
21     sw s4,32(sp)
22     sw s5,36(sp)
23     sw s6,40(sp)
24     sw s7,44(sp)
25
26     #Guardo los parametros recibidos en fp
27     sw a0,48($fp)
28     sw a1,52($fp)
29     sw a2,56($fp)
30
31     ##Guardo en variables auxiliares las direcciones de memoria a intercambiar
32     lw t0, 48($fp)    #en 48 esta el words
33     lw t1, 52($fp)    #en 52 esta el i
34     lw t2, 56($fp)    #en 56 esta el j
35
36     ##Realizo el intercambio
37     #t3 = direccion de fp+(i*4)
38     #t4 = direccion de fp+(j+4)
39     sll t1, t1, 2
40     sll t2, t2, 2
41     addu t1, t1, t0
42     addu t2, t2, t0
43
44
45     #Cargo los elementos a intercambiar en t3 y t4
46     lw t3, 0(t1)
47     lw t4, 0(t2)
48
49     #Guardo los elementos de manera intercambiada
50     sw t4, 0(t1)
51     sw t3, 0(t2)
52
53
54     lw v0, 0(t2)
55
56     ##Comienzo retorno
57     lw s7, 44(sp)
58     lw s6, 40(sp)
59     lw s5, 36(sp)
60     lw s4, 32(sp)
61     lw s3, 28(sp)
62     lw s2, 24(sp)

```

```
63  lw s1, 20(sp)
64  lw s0, 16(sp)
65
66  move sp, $fp
67  lw gp, 8(sp)
68  lw $fp, 12(sp)
69  addiu sp, sp, 48
70  jr ra      #ra = return address es la direccion a la q quiero volver luego de llamar la funcion
71
72  #endif
```