

Trabajo Práctico N°1
“Conjunto de instrucciones MIPS”

Belén Beltran, Padrón Nro. 91.718
belubeltran@gmail.com

Pablo Ariel Rodriguez, Padrón Nro. 93.970
prodriguez@fi.uba.ar

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

2do. 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. Implementación | 1 |
| 4.1. Implementación en C | 2 |
| 4.1.1. Algoritmo <i>Bubblesort</i> | 2 |
| 4.1.2. Algoritmo <i>Heapsort</i> | 3 |
| 4.2. Implementación en Assembly | 5 |
| 4.2.1. Algoritmo <i>Heapsort</i> | 5 |
| 5. Debugging | 6 |
| 6. Tiempos de ejecución | 6 |
| 7. Conclusiones | 8 |
| Appendices | 10 |
| A. Implementación completa en lenguaje C | 10 |
| A.1. <i>tp1.c</i> . Implementación del main del programa | 10 |
| A.2. <i>bubblesort.h</i> . Declaración del algoritmo Bubblesort | 13 |
| A.3. <i>bubblesort.c</i> . Definición del algoritmo Bubblesort | 13 |
| A.4. <i>heapsort.h</i> . Declaración del algoritmo Heapsort | 14 |
| A.5. <i>heapsort.c</i> . Definición del algoritmo Heapsort | 15 |
| B. Implementación completa en lenguaje Assembly MIPS32 | 16 |
| B.1. <i>tp1.c</i> . Implementación del main del programa | 16 |
| B.2. <i>heapsort.S</i> . Definición del algoritmo Heapsort | 19 |
| B.3. <i>heap.S</i> . Definición del algoritmo Heap | 20 |
| B.4. <i>strcmpi.S</i> . Definición del algoritmo Strcmpi | 21 |

1. Introducción

En el presente trabajo se tiene como objetivo la comparación entre dos algoritmos de ordenamiento: el *Bubblesort*[1] y el *Heapsort*[2]. Para realizar dicha comparación entre ambos se realizó la implementación en lenguaje C de cada uno de estos. A su vez, para comparar el rendimiento entre código de alto nivel y de código nativo, se desarrolló la implementación del Heapsort en assembly MIPS, con el fin de poder comparar los tiempos de ejecución de ambos programas y realizar así un estudio de las mejoras que se producen.

La totalidad del trabajo se ha realizado en una plataforma *NetBSD/MIPS-32* mediante el *GXEmul* [3].

Todos los archivos y códigos fuente aquí mencionados, así como también el presente informe, pueden ser descargados como un archivo comprimido ZIP del repositorio del grupo¹.

2. Compilación

La herramienta para compilar tanto el código assembly como C será el *GCC* [4].

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 trabajo ².

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 MIPS.

3.1. Implementación en C

El resultado de compilación utilizando el comando *make* será un archivo ejecutable de nombre *tp1*, 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 ingresarse el caracter '-' y luego introducir las palabras;
- *-p [ARGS]*: El programa recibe nombres de archivos de texto o strings ingresados por *stdin*, ordenandolos utilizando el algoritmo Heapsort. Para utilizar *stdin* deberá ingresarse el caracter '-' y luego introducir las palabras.

3.2. Implementación en Assembly

El resultado de compilación utilizando el comando *make* será un archivo ejecutable de nombre *tp1*, el cual aceptará un archivo de texto como argumento y lo ordenará con el algoritmo Heapsort.

4. Implementación

En lo que sigue de la sección, se presentarán los códigos fuente de la implementación del algoritmo. Aquellos lectores interesados en la implementación completa del programa, pueden dirigirse al apéndice ubicado al final del presente informe.

¹URI del Repositorio: <https://github.com/federicomrossi/6620-trabajos-practicos-2C2013/tree/master/tp1>

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:

- **tp1**: 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;
- **bubblesort**: Módulo encargado de implementar el algoritmo de ordenamiento Bubblesort. Recibe como parámetros un arreglo de palabras desordenado y el tamaño del mismo. Como resultado devuelve dicho arreglo ordenado.
- **heapsort**: Módulo encargado de implementar el algoritmo de ordenamiento Heapsort. 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     // ningún cambio realizado.
31     do
32     {
33         huboIntercambio = false;
34
35         for(i = 1; i < n; i++)
36         {
37             // Si el de índice menor es mayor que el de índice 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 índice superior se encuentra ya ordenado una
51         // vez finalizada la pasada, se reduce en uno la cantidad de índices
52         // a iterar en la próxima pasada.
53         n -= 1;
54
55     } while(huboIntercambio);
56 }

```

4.1.2. Algoritmo *Heapsort*

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

Código 3: "heapsort.h"

```

1 /* *****
2  * *****
3  *
4  * LIBRERIA HEAPSORT
5  *
6  * *****
7  * *****/
8
9
10
11 #ifndef HEAPSORT_H
12 #define HEAPSORT_H
13
14
15
16 // Funcion que aplica el algoritmo de ordenamiento Heapsort 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 heapsort(char* words[], int arraysize);
22
23
24
25 #endif

```

Código 4: "heapsort.c"

```
1 /* *****
2 * *****
3 *
4 * LIBRERIA HEAPSORT
5 *
6 * *****
7 * *****/
8
9
10 #include "heapsort.h"
11 #include <string.h>
12
13
14 void heap(char* words[],int raiz, int fin){
15     char finished = 0;
16     char* tmp;
17     int hijo;
18     while (raiz*2+1 <= fin && !finished){
19         //Elijo el hijo mayor
20         if(raiz*2+1==fin || strcmp(words[raiz*2+1],words[raiz*2+2]) > 0 ){
21             hijo=raiz*2+1;
22         }else{
23             hijo=raiz*2+2;
24         }
25         //Checkeo si es mayor e intercambio
26         if(strcmp(words[raiz],words[hijo]) < 0){
27             tmp=words[raiz];
28             words[raiz]=words[hijo];
29             words[hijo]=tmp;
30             raiz=hijo;
31         }else{
32             finished=1;
33         }
34     }
35 }
36
37
38
39 // Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
40 // arreglo de palabras.
41 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
42 // es el tamaño de dicho arreglo.
43 // POST: el arreglo 'words' queda ordenado.
44 void heapsort(char* words[], int arraysize){
45     int i;
46     char* tmp;
47     //Armo Heap
48     for (i=(arraysize/2)-1;i>=0;i--){
49         heap(words,i,arraysize-1);
50     }
51
52     //Extraigo raiz y armo heap
53     for(i=arraysize-1;i>0;i--){
54         tmp=words[0];
55         words[0]=words[i];
56         words[i]=tmp;
57         heap(words,0,i-1);
58     }
59 }
```


4.2. Implementación en Assembly

La implementación del programa fue dividida en los siguientes módulos:

- **tp1**: Solamente recibe un texto como argumento por línea de comandos y lo imprime ordenándolo mediante `heapsort`. Esta implementado en lenguaje C;
- **strcmpi**: Función implementada en assembly MIPS que se encarga de realizar la comparación de strings sin ser sensible a mayúsculas;
- **heap**: Función implementada en assembly MIPS que se encarga de armar la estructura heap que es necesaria en cada paso del `heapsort`;
- **heapsort**: Implementación en assembly MIPS del algoritmo de ordenamiento Heapsort.

4.2.1. Algoritmo *Heapsort*

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

Código 5: “*heapsort.S*”

```
1 #ifndef USE_MIPS_ASSEMBLY
2 #define USE_MIPS_ASSEMBLY
3 #include <mips/regdef.h>
4 #include <sys/syscall.h>
5
6
7 # Utilizacion de registros:
8 #
9 # s0: words
10 # s1: arraysize
11 # s2: variable local "i"
12 # s3: variable local "tmp"
13 # t0: variable local auxiliar
14 # t1: variable local auxiliar
15 #
16
17
18 .text
19 .align 2
20 .globl heapsort
21 .extern heap
22
23     # ABA: 16; SRA: 16; LTA:24
24 heapsort: subu sp,sp,56      # Creamos stack frame.
25     sw ra,52(sp)           # Return address
26     sw $fp,48(sp)          # Frame pointer
27     sw gp,44(sp)           # Global pointer
28     move $fp,sp            # Establecemos la base
29
30     # Almacenamos contenido de registros
31     sw s0,40(sp)
32     sw s1,36(sp)
33     sw s2,32(sp)
34     sw s3,28(sp)
35
36     # Almacenamos los parametros recibidos
37     sw a0,56(sp)           # words
38     sw a1,60(sp)           # arraysize
39
40     # Almacenamos los parametros guardados en fp en variables
41     move s0,a0             # words
42     move s1,a1             # arraysize
43
44     # Armamos el heap
45     srl t0,s1,1            # t0 = arraysize / 2
46     subu s2,t0,1           # i = arraysize / 2 - 1
47 FOR_1: slt t0,s2,zero       # i es menor que 0
48     bne t0,zero,FOR_1_FIN   # Si t0 = 1 saltamos a FOR_1_FIN
49
50     add a1,s2,zero         # Cargamos i en a1
51     subu a2,s1,1           # Cargamos arraysize-1 en a2
52     jal heap               # Saltamos a funcion "heap"
```

```

53
54     subu s2,s2,1      # i = i-1 (decrementamos i)
55     j    FOR_1        # Saltamos a etiqueta FOR_1
56
57     # Iteramos sobre el heap intercambiando la raiz con el ultimo
58     # elemento del arreglo, rearmando luego un heap desde los indices
59     # 0 al n-1
60 FOR_1_FIN: subu s2,s1,1      # i = arraysize-1
61 FOR_2:    beq  s2,zero,FOR_2_FIN # Si i = 0 saltamos a FOR_2_FIN
62
63     lw  s3,0(s0)      # tmp = words[0]
64     sll t1,s2,2       # t1 = i * 4
65     add t1,t1,s0      # t1 = words[i]
66     lw  t2,0(t1)      # t2 = dato words[i]
67     sw  t2,0(s0)      # words[0] = words[i]
68     sw  s3,0(t1)      # words[i] = tmp
69
70     add a0,s0,zero    # Cargamos words en a0
71     add a1,zero,zero  # Cargamos 0 en a1
72     subu a2,s2,1      # a2 = i -1
73     jal heap         # Saltamos a funcion "heap"
74
75     subu s2,s2,1      # i = i-1 (decrementamos i)
76     j    FOR_2        # Saltamos a etiqueta FOR_2
77
78     # Reestablecemos valores iniciales
79 FOR_2_FIN: lw  a0,56(sp) # words
80           lw  a1,60(sp) # arraysize
81
82     lw  s0,40(sp)
83     lw  s1,36(sp)
84     lw  s2,32(sp)
85     lw  s3,28(sp)
86
87     move sp,$fp
88     lw  gp,44(sp)      # Global pointer
89     lw  $fp,48(sp)    # Frame pointer
90     lw  ra,52(sp)     # Return address
91     addi sp,sp,56
92
93     jr   ra
94
95 #endif

```

5. Debugging

Para analizar el correcto funcionamiento del programa se crearon casos de prueba pertinentes, considerando combinaciones diferentes en el ingreso de parámetros al programa principal, como también tomando en cuenta las diferentes salidas obtenidas a partir de los algoritmos *Bubblesort* como *Heapsort*. Los resultados fueron comparados con los casos esperados y así se determinó el correcto funcionamiento del programa en su totalidad.

6. Tiempos de ejecución

Se desea medir el tiempo que tarda el programa en ejecutarse en la máquina virtual MIPS32. Para ello se utilizó el comando *GNU "time"* [5], que mide los tiempos de ejecución de un programa. Existen dos casos de prueba interesantes a comparar.

El primero es la comparación entre el *Bubblesort* y el *Heapsort*, ambos realizados en el programa en *C*. Como se observa en el *Cuadro 1*, el primer algoritmo resulta muchísimo más lento que el segundo. Esto se debe a que la performance del *Bubblesort* (en el peor caso de n^2) es mucho mayor a la del *Heapsort* (en el peor caso de $n * \log(n)$).

| Nombre de Archivo | Tiempo bubblesort | Tiempo heapsort |
|-------------------|-------------------|-----------------|
| 25deMayo.txt | 15m 33,414s | 0m 3,883s |
| argentina.txt | 13m 41,020s | 0m 15,469s |
| cookbook.txt | 2m 0,629s | 0m 1,383s |
| 2pac.txt | 54m 8,105s | 0m 7,371s |

Cuadro 1: *Tiempos obtenidos en la ejecución del programa en C para distintos archivos de entrada.*

En la *Figura 1* se puede observar el speedup del Bubblesort contra el Heapsort:

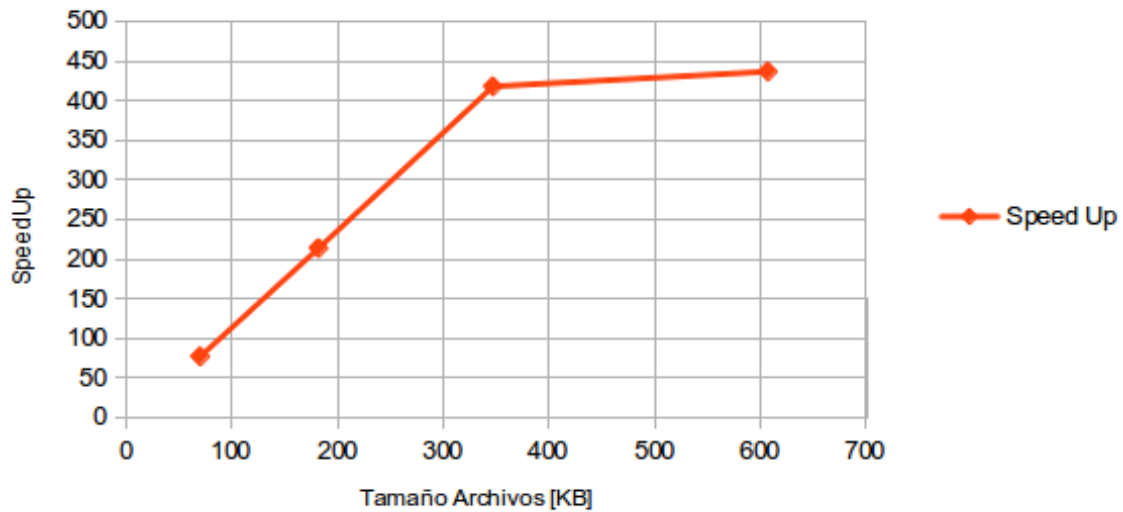


Figura 1: *Gráfico del speedup del Bubblesort contra el Heapsort.*

En el segundo caso, se compara el tiempo de ejecución del algoritmo *Heapsort* en sus dos versiones: la que genera el compilador de C y la que fue diseñada e implementada en Assembly de MIPS32. Como se observa en la *Figura 2*, el algoritmo con mejor performance fue el diseñado para Assembly de MIPS32. Se debe notar que al compilar con *GCC* no se le agregó ninguna optimización al código de Assembly (opción por default al compilar).

| Nombre de Archivo | Tiempo Heapsort MIPS32 | Tiempo heapsort C |
|-------------------|------------------------|-------------------|
| 25deMayo.txt | 3,395s | 3,883s |
| argentina.txt | 13,367s | 15,469s |
| cookbook.txt | 1,148s | 1,383s |
| 2pac.txt | 6,715s | 7,371s |

Cuadro 2: *Tiempos obtenidos en la ejecución del algoritmo Heapsort para distintos archivos de entrada.*

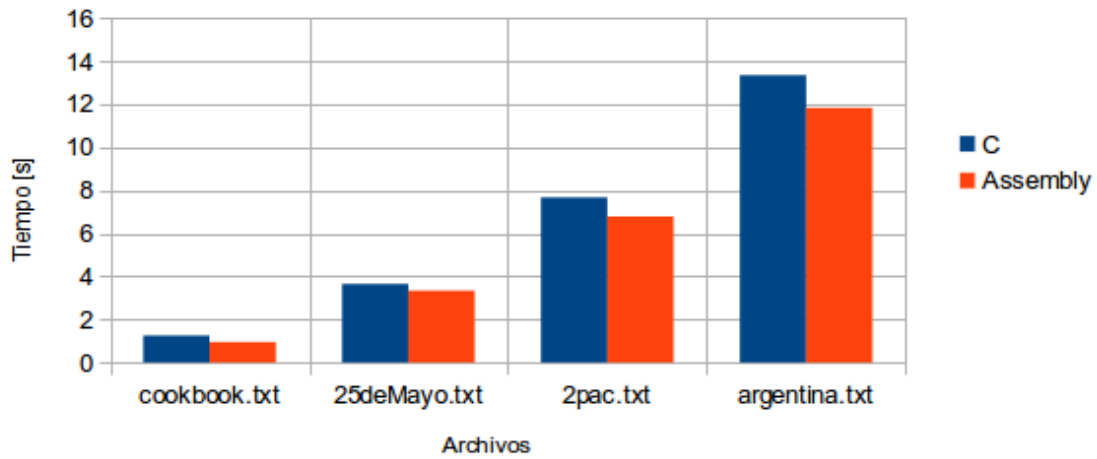


Figura 2: Gráfico comparando la performance del Heapsort en sus dos implementaciones (C vs Assembly).

En la *Figura 3* se observa el speedup del Heapsort en C contra la versión en Assembly.

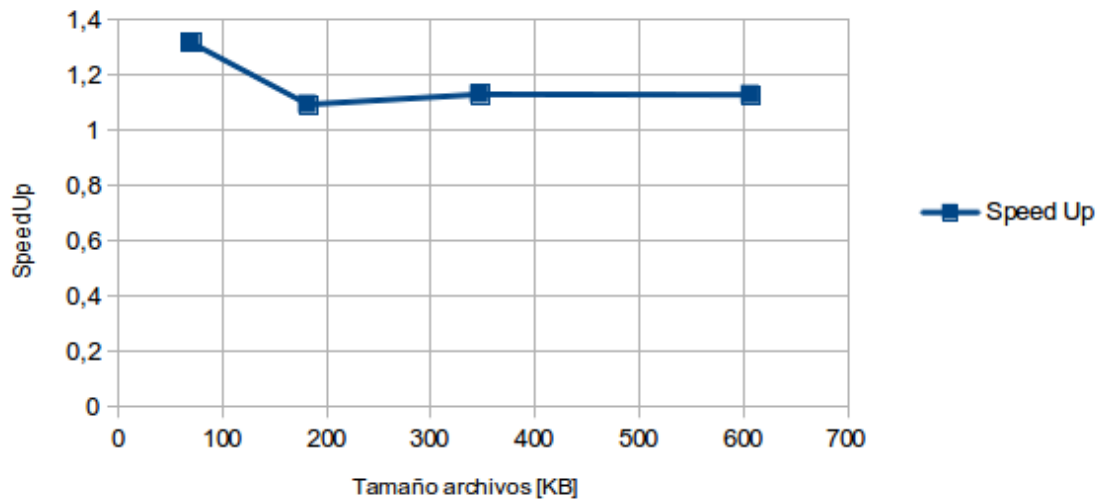


Figura 3: Gráfico del speedup del Heapsort en C contra la versión en Assembly.

7. Conclusiones

Se puede observar que existen diferencias significativas entre los algoritmos Bubblesort y el Heapsort, con respecto a los tiempos de ejecución. Es notable que el Bubblesort es inferior en cuanto a la performance comparado con el Heapsort (ambos implementados en C). Esto se puede corroborar observando el speedup obtenido.

También se pueden ver las diferencias en el Heapsort en sus dos implementaciones: en C y en Assembly MIPS32. Los tiempos de ejecución de ambos son muy similares, pero se puede observar que la versión diseñada para Assembly es apenas mejor. También se puede corroborar esto observando el speedup obtenido.

Referencias

- [1] Bubblesort, http://en.wikipedia.org/wiki/Bubble_sort
- [2] Heapsort, <http://en.wikipedia.org/wiki/Heapsort>
- [3] The NetBSD project, <http://www.netbsd.org/>
- [4] GCC, the GNU Compiler Collection, <http://gcc.gnu.org/>
- [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. *tp1.c*. Implementación del main del programa

Código 6: “*tp1.c*”

```
1  /*
2  =====
3  Name      : tp1.c
4  Authors   : Belen Beltran (91718)
5             Federico Martin Rossi (92086)
6             Pablo Rodriguez (93970)
7  Version   : 1.0
8  Description : Programa de ordenamiento heapsort o bubblesort
9  =====
10 */
11
12 #include "heapsort.h"
13 #include "bubblesort.h"
14 #include <stdio.h>
15 #include <string.h>
16 #include <getopt.h>
17 void* malloc(size_t);
18 void* realloc(void*,size_t);
19 void free(void*);
20 int system(const char* cadena);
21
22 /** Funciones Auxiliares **/
23 void readWord(FILE* fd,char* word){
24     int index=0;
25     int c;
26     while((c=getc(fd))!=EOF && c!=32 && c!='\n' && index < 49){
27         if((c>=48 && c<=57) || (c>=65 && c<=90) ||
28            (c>=97 && c<=122) || (c>=130 && c<=165)){
29             word[index++]=c;
30         }
31     }
32     word[index]='\0';
33 }
34
35 // Devuelve la cantidad de palabras leídas
36 char** leerArchivo(char* sourceName, int* cant) {
37     int size = 0;
38     int tamano = 5000;
39     int tam;
40     char word[50];
41     FILE* sourcefd;
42     char** palabras;
43
44
45     // Se toma '-' como stdin
46     if (sourceName[0] == '-')
47         sourcefd = stdin;
48     // Se abre el archivo correspondiente
49     else {
50         sourcefd = fopen(sourceName,"r");
51         if (!sourcefd) {
52             fprintf(stderr, "El archivo no pudo ser abierto\n");
53             return 0;
54         }
55     }
56     // Si el archivo esta vacio
57     fseek(sourcefd, 0, SEEK_END);
58     if ( ftell(sourcefd) == 0 ) {
59         fprintf(stderr, "El archivo esta vacio\n");
60         return 0;
61     }
62     // Se vuelve al principio
```

```

63     fseek(sourcefd, 0, SEEK_SET);
64 }
65
66 /* Cargo el archivo a memoria*/
67 palabras = (char**)malloc(tamano*sizeof(char*));
68
69 // Se toma '-' como stdin
70 if (sourceName[0] == '-')
71     sourcefd = stdin;
72 // Se abre el archivo correspondiente
73 else
74     sourcefd = fopen(sourceName, "r");
75
76 readWord(sourcefd, word);
77 while (!feof(sourcefd)){
78     if (size >= tamano){
79         tamano += 5000;
80         palabras = (char**)realloc(palabras, tamano*sizeof(char*));
81     }
82     tam = strlen(word) + 1;
83     if (palabras == NULL) {
84         fprintf(stderr, "\n\nNo se pueden alocar mas palabras\n\n");
85         break;
86     }
87     palabras[size] = malloc(tam);
88     memcpy(palabras[size], word, tam);
89     size++;
90     readWord(sourcefd, word);
91 }
92 // Se cierra el archivo si corresponde
93 if (sourceName[0] != '-')
94     fclose(sourcefd);
95
96 *cant = size;
97
98 return palabras;
99 }
100
101 void displayAyuda() {
102     printf("%s", "Usage: ./tp1 [OPCION] [ARCHIVO]\n\n");
103     printf("\nOPCION:\n\n");
104     printf("-V, --version\tImprime la version del programa.\n\n");
105     printf("-h, --help\tImprime esta ayuda. \n\n");
106     printf("-b, --bubblesort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\n");
107     printf("-p, --heapsort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\n");
108     printf("NOTA: De no recibirse una OPCION, se podran recibir nombres de ARCHIVO, resultando en la\n");
109     printf("concatenacion e impresion por pantalla de estos.\n\n");
110     printf("\nEjemplos:\n\n");
111     printf("tp1 -b palabras.txt\n\n");
112     printf("tp1 -p palabras.txt\n");
113 }
114
115 void displayVersion() {
116     printf("%s", "Esta aplicacion ejecuta un ordenamiento sobre las palabras contenidas en un archivo.\n");
117     printf("Puede ejecutarse el algoritmo heapsort o bubblesort.\n\n");
118     printf("\nVersion: v1.0\n");
119 }
120
121 void ejecutarHeapsort(char* nombreArchivo) {
122     int i, size;
123     char** palabras = leerArchivo(nombreArchivo, &size);
124
125     if (!palabras)
126         return;
127
128     /*Ordeno*/
129     heapsort(palabras, size);
130
131     /*Imprimo el resultado y libero memoria*/
132     for(i=0; i<size; i++){
133         printf("%s ", palabras[i]);
134         free(palabras[i]);
135     }
136     free(palabras);
137
138     printf("\n");

```

```

139 }
140
141 void ejecutarBubblesort(char* nombreArchivo) {
142     int i, size;
143     char** palabras = leerArchivo(nombreArchivo, &size);
144
145     if (!palabras)
146         return;
147
148     /*Ordeno*/
149     bubblesort(palabras, size);
150
151     /*Imprimo el resultado y libero memoria*/
152     for(i=0; i<size; i++){
153         printf("%s ", palabras[i]);
154         free(palabras[i]);
155     }
156     free(palabras);
157
158     printf("\n");
159 }
160
161
162 int main (int argc, char* argv[]) {
163     int c;
164     FILE* sourcefd;
165
166     // Si hay pocos argumentos
167     if (argc < 2) {
168         fprintf(stderr, "ERROR: No hay suficientes argumentos\n");
169         return 1;
170     }
171
172
173     // while (1) {
174     static struct option opciones[] = {
175         // No setean flags
176         {"help", no_argument, 0, 'h'},
177         {"version", no_argument, 0, 'V'},
178         {"bubble", required_argument, 0, 'b'},
179         {"heap", required_argument, 0, 'p'},
180         {0, 0, 0, 0}
181     };
182     // getopt_long stores the option index here.
183     int indice_opciones = -1;
184
185     int hayOpciones = 0;
186
187     while( (c = getopt_long (argc, argv, "hVb:p:", opciones, &indice_opciones)) != -1 ) {
188
189         hayOpciones = 1;
190
191         switch (c) {
192             case 0:
193                 printf ("Estoy en 0. Option %s", opciones[indice_opciones].name);
194                 if (optarg)
195                     printf (" with arg %s", optarg);
196                 printf ("\n");
197                 break;
198
199             case 'h':
200                 if (argc == 2)
201                     displayAyuda();
202                 break;
203
204             case 'V':
205                 if (argc == 2)
206                     displayVersion();
207                 break;
208
209             case 'p':
210                 if (argc == 3)
211                     ejecutarHeapsort(optarg);
212                 break;
213
214             case 'b':

```



```

215     if (argc == 3)
216         ejecutarBubblesort(optarg);
217     break;
218
219     default:
220         break;
221 }
222 }
223 // Si hay argumentos que no son opciones, se buscan si nos nombres de archivo
224 if (optind < argc && !hayOpciones) {
225     char op[256];
226     while (optind < argc) {
227         sprintf(op, "cat %s", argv[optind++]);
228         system(op);
229     }
230 }
231
232 return 0;
233 }

```

A.2. *bubblesort.h*. Declaración del algoritmo 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 del algoritmo 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. *heapsort.h*. Declaración del algoritmo Heapsort

Código 9: “*heapsort.h*”

```

1 /* *****
2 * *****
3 *
4 * LIBRERIA HEAPSORT
5 *
6 * *****
7 * *****/
8
9
10
11 #ifndef HEAPSORT_H
12 #define HEAPSORT_H
13
14
15
16 // Funcion que aplica el algoritmo de ordenamiento Heapsort 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 heapsort(char* words[], int arraysize);
22
23
24
25 #endif

```

A.5. *heapsort.c*. Definición del algoritmo Heapsort

Código 10: "*heapsort.c*"

```

1  /* *****
2  * *****
3  *
4  * LIBRERIA HEAPSORT
5  *
6  * *****
7  * *****/
8
9
10 #include "heapsort.h"
11 #include <string.h>
12
13
14 void heap(char* words[],int raiz, int fin){
15     char finished = 0;
16     char* tmp;
17     int hijo;
18     while (raiz*2+1 <= fin && !finished){
19         //Elijo el hijo mayor
20         if(raiz*2+1==fin || strcasecmp(words[raiz*2+1],words[raiz*2+2]) > 0 ){
21             hijo=raiz*2+1;
22         }else{
23             hijo=raiz*2+2;
24         }
25         //Checkeo si es mayor e intercambio
26         if(strcasecmp(words[raiz],words[hijo]) < 0){
27             tmp=words[raiz];
28             words[raiz]=words[hijo];
29             words[hijo]=tmp;
30             raiz=hijo;
31         }else{
32             finished=1;
33         }
34     }
35 }
36
37
38
39 // Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
40 // arreglo de palabras.
41 // PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
42 // es el tamaño de dicho arreglo.
43 // POST: el arreglo 'words' queda ordenado.
44 void heapsort(char* words[], int arraysize){
45     int i;
46     char* tmp;
47     //Armo Heap
48     for (i=(arraysize/2)-1;i>=0;i--){
49         heap(words,i,arraysize-1);
50     }
51
52     //Extraigo raiz y armo heap
53     for(i=arraysize-1;i>0;i--){
54         tmp=words[0];
55         words[0]=words[i];
56         words[i]=tmp;
57         heap(words,0,i-1);
58     }
59 }

```

B. Implementación completa en lenguaje Assembly MIPS32

B.1. *tp1.c*. Implementación del main del programa

Código 11: “*tp1.c*”

```
1  /*
2  =====
3  Name      : tp1.c
4  Authors   : Belen Beltran (91718)
5             Federico Martin Rossi (92086)
6             Pablo Rodriguez (93970)
7  Version    : 1.0
8  Description : Programa de ordenamiento heapsort
9  =====
10 */
11
12 #include <stdio.h>
13 #include <string.h>
14 #include <getopt.h>
15 #include "bubblesort.h"
16
17 void* malloc(size_t);
18 void* realloc(void*,size_t);
19 void free(void*);
20 int system(const char* cadena);
21 extern heapsort(char**, int);
22
23 /** Funciones Auxiliares **/
24 void readWord(FILE* fd,char* word){
25     int index=0;
26     int c;
27     while((c=getc(fd))!=EOF && c!=32 && c!='\n' && index < 49){
28         if((c>=48 && c<=57) || (c>=65 && c<=90) ||
29            (c>=97 && c<=122) || (c>=130 && c<=165)){
30             word[index++]=c;
31         }
32     }
33     word[index]='\0';
34 }
35
36 // Devuelve la cantidad de palabras leídas
37 char** leerArchivo(char* sourceName, int* cant) {
38     int size = 0;
39     int tamano = 5000;
40     int tam;
41     char word[50];
42     FILE* sourcefd;
43     char** palabras;
44
45
46     // Se toma '-' como stdin
47     if (sourceName[0] == '-')
48         sourcefd = stdin;
49     // Se abre el archivo correspondiente
50     else {
51         sourcefd = fopen(sourceName,"r");
52         if (!sourcefd) {
53             fprintf(stderr, "El archivo no pudo ser abierto\n");
54             return 0;
55         }
56     }
57     // Si el archivo esta vacio
58     fseek(sourcefd, 0, SEEK_END);
59     if ( ftell(sourcefd) == 0 ) {
60         fprintf(stderr, "El archivo esta vacio\n");
61         return 0;
62     }
63
64     // Se vuelve al principio
65     fseek(sourcefd, 0, SEEK_SET);
66
67     /* Cargo el archivo a memoria*/
68     palabras = (char**)malloc(tamano*sizeof(char*));
```

```

69 // Se toma '-' como stdin
70 if (sourceName[0] == '-')
71     sourcefd = stdin;
72 // Se abre el archivo correspondiente
73 else
74     sourcefd = fopen(sourceName, "r");
75
76 readWord(sourcefd, word);
77 while (!feof(sourcefd)){
78     if (size >= tamano){
79         tamano += 5000;
80         palabras = (char**)realloc(palabras, tamano*sizeof(char*));
81     }
82     tam = strlen(word) + 1;
83     if (palabras == NULL) {
84         fprintf(stderr, "\n\nNo se pueden alocar mas palabras\n\n");
85         break;
86     }
87     palabras[size] = malloc(tam);
88     memcpy(palabras[size], word, tam);
89     size++;
90     readWord(sourcefd, word);
91 }
92 // Se cierra el archivo si corresponde
93 if (sourceName[0] != '-')
94     fclose(sourcefd);
95
96 *cant = size;
97
98 return palabras;
99 }
100
101 void displayAyuda() {
102     printf("%s", "Usage: ./tp1 [OPCION] [ARCHIVO]\n\n");
103     printf("\nOPCION:\n\n");
104     printf("-V, --version\tImprime la version del programa.\n\n");
105     printf("-h, --help\tImprime esta ayuda. \n\n");
106     printf("-b, --bubblesort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\n");
107     printf("-p, --heapsort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\n");
108     printf("NOTA: De no recibirse una OPCION, se podran recibir nombres de ARCHIVO, resultando en la\n");
109     printf("concatenacion e impresion por pantalla de estos.\n\n");
110     printf("\nEjemplos:\n\n");
111     printf("tp1 -b palabras.txt\n\n");
112     printf("tp1 -p palabras.txt\n\n");
113 }
114
115 void displayVersion() {
116     printf("%s", "Esta aplicacion ejecuta un ordenamiento sobre las palabras contenidas en un archivo.\n");
117     printf("Puede ejecutarse el algoritmo heapsort o bubblesort.\n");
118     printf("\nVersion: v1.0\n");
119 }
120
121 void ejecutarHeapsort(char* nombreArchivo) {
122     int i, size;
123     char** palabras = leerArchivo(nombreArchivo, &size);
124
125     if (!palabras)
126         return;
127
128     /*Ordeno*/
129     heapsort(palabras, size);
130
131     /*Imprimo el resultado y libero memoria*/
132     for(i=0; i<size; i++){
133         printf("%s ", palabras[i]);
134         free(palabras[i]);
135     }
136     free(palabras);
137
138     printf("\n");
139 }
140
141 void ejecutarBubblesort(char* nombreArchivo) {
142     int i, size;
143     char** palabras = leerArchivo(nombreArchivo, &size);

```

```

145
146 if (!palabras)
147     return;
148
149 /*Ordeno*/
150     bubblesort(palabras,size);
151
152 /*Imprimo el resultado y libero memoria*/
153     for(i=0;i<size;i++){
154         printf("%s ",palabras[i]);
155         free(palabras[i]);
156     }
157     free(palabras);
158
159     printf("\n");
160 }
161
162 int main (int argc, char* argv[]) {
163     int c;
164
165     // Si hay pocos argumentos
166     if (argc < 2) {
167         fprintf(stderr, "ERROR: No hay suficientes argumentos\n");
168         return 1;
169     }
170
171
172     // while (1) {
173     static struct option opciones[] = {
174         // No setean flags
175         {"help", no_argument, 0, 'h'},
176         {"version", no_argument, 0, 'V'},
177         {"bubble", required_argument, 0, 'b'},
178         {"heap", required_argument, 0, 'p'},
179         {0,0,0,0}
180     };
181     // getopt_long stores the option index here.
182     int indice_opciones = -1;
183
184     int hayOpciones = 0;
185
186     while( (c = getopt_long (argc, argv, "hVb:p:", opciones, &indice_opciones)) != -1 ) {
187
188         hayOpciones = 1;
189
190         switch (c) {
191             case 0:
192                 printf ("Estoy en 0. Option %s", opciones[indice_opciones].name);
193                 if (optarg)
194                     printf (" with arg %s", optarg);
195                 printf ("\n");
196                 break;
197
198             case 'h':
199                 if (argc == 2)
200                     displayAyuda();
201                 break;
202
203             case 'V':
204                 if (argc == 2)
205                     displayVersion();
206                 break;
207
208             case 'p':
209                 if (argc == 3)
210                     ejecutarHeapsort(optarg);
211                 break;
212
213             case 'b':
214                 if (argc == 3)
215                     ejecutarBubblesort(optarg);
216                 break;
217
218             default:
219                 break;
220         }

```

```

221 }
222 // Si hay argumentos que no son opciones, se buscan si nos nombres de archivo
223 if (optind < argc && !hayOpciones) {
224     char op[256];
225     while (optind < argc) {
226         sprintf(op, "cat %s", argv[optind++]);
227         system(op);
228     }
229 }
230
231 return 0;
232 }

```

B.2. *heapsort.S*. Definición del algoritmo Heapsort

Código 12: “*heapsort.S*”

```

1  #ifndef USE_MIPS_ASSEMBLY
2  #define USE_MIPS_ASSEMBLY
3  #include <mips/regdef.h>
4  #include <sys/syscall.h>
5
6
7  # Utilizacion de registros:
8  #
9  # s0: words
10 # s1: arraysize
11 # s2: variable local "i"
12 # s3: variable local "tmp"
13 # t0: variable local auxiliar
14 # t1: variable local auxiliar
15 #
16
17
18 .text
19 .align 2
20 .globl heapsort
21 .extern heap
22
23     # ABA: 16; SRA: 16; LTA:24
24 heapsort: subu    sp,sp,56      # Creamos stack frame.
25         sw      ra,52(sp)      # Return address
26         sw      $fp,48(sp)     # Frame pointer
27         sw      gp,44(sp)      # Global pointer
28         move    $fp,sp        # Establecemos la base
29
30     # Almacenamos contenido de registros
31         sw      s0,40(sp)
32         sw      s1,36(sp)
33         sw      s2,32(sp)
34         sw      s3,28(sp)
35
36     # Almacenamos los parametros recibidos
37         sw      a0,56(sp)      # words
38         sw      a1,60(sp)      # arraysize
39
40     # Almacenamos los parametros guardados en fp en variables
41         move    s0,a0          # words
42         move    s1,a1          # arraysize
43
44     # Armamos el heap
45         srl     t0,s1,1        # t0 = arraysize / 2
46         subu    s2,t0,1        # i = arraysize / 2 - 1
47 FOR_1:  slt     t0,s2,zero      # i es menor que 0
48         bne     t0,zero,FOR_1_FIN # Si t0 = 1 saltamos a FOR_1_FIN
49
50         add     a1,s2,zero      # Cargamos i en a1
51         subu    a2,s1,1        # Cargamos arraysize-1 en a2
52         jal     heap           # Saltamos a funcion "heap"
53

```

```

54     subu s2,s2,1      # i = i-1 (decrementamos i)
55     j     FOR_1      # Saltamos a etiqueta FOR_1
56
57     # Iteramos sobre el heap intercambiando la raiz con el ultimo
58     # elemento del arreglo, rearmando luego un heap desde los indices
59     # 0 al n-1
60 FOR_1_FIN: subu s2,s1,1      # i = arraysize-1
61 FOR_2:     beq  s2,zero,FOR_2_FIN  # Si i = 0 saltamos a FOR_2_FIN
62
63     lw  s3,0(s0)      # tmp = words[0]
64     sll t1,s2,2      # t1 = i * 4
65     add t1,t1,s0      # t1 = words[i]
66     lw  t2,0(t1)      # t2 = dato words[i]
67     sw  t2,0(s0)      # words[0] = words[i]
68     sw  s3,0(t1)      # words[i] = tmp
69
70     add a0,s0,zero    # Cargamos words en a0
71     add a1,zero,zero  # Cargamos 0 en a1
72     subu a2,s2,1      # a2 = i -1
73     jal heap         # Saltamos a funcion "heap"
74
75     subu s2,s2,1      # i = i-1 (decrementamos i)
76     j     FOR_2      # Saltamos a etiqueta FOR_2
77
78     # Reestablecemos valores iniciales
79 FOR_2_FIN: lw  a0,56(sp)      # words
80           lw  a1,60(sp)      # arraysize
81
82     lw  s0,40(sp)
83     lw  s1,36(sp)
84     lw  s2,32(sp)
85     lw  s3,28(sp)
86
87     move sp,$fp
88     lw  gp,44(sp)      # Global pointer
89     lw  $fp,48(sp)     # Frame pointer
90     lw  ra,52(sp)      # Return address
91     addi sp,sp,56
92
93     jr   ra
94
95 #endif

```

B.3. *heap.S*. Definición del algoritmo Heap

Código 13: “*heap.S*”

```

1  #ifndef USE_MIPS_ASSEMBLY
2  #define USE_MIPS_ASSEMBLY
3  #include <mips/regdef.h>
4  #include <sys/syscall.h>
5
6
7  .text
8  .align 2
9  .globl heap
10 .extern strcmpi
11
12 heap:
13     subu sp, sp, 40 #creo el SRA 16 y LTA 8 y ABA 16 = 40
14     sw ra, 32(sp)
15     sw $fp, 28(sp)
16     sw gp, 24(sp)
17     move $fp, sp
18     sw a0, 40(sp) #Guardo los parametros en el ABA
19     sw a1, 44(sp)
20     sw a2, 48(sp)
21     #t1, sp+16 = tmp
22     #t2, sp+20 = hijo
23 loop: #comparacion de while. No hago la comp de finished, Break instead

```



```

24  sll t3, a1, 1 #raiz*2
25  addi t3, t3, 1 #+1
26  bgt t3, a2, fin_loop # > fin
27  #comparaciones de if
28  subu t4, t3, a2
29  beq t4, 0, if
30  #Guardo todos los tmp porque llamo una funcion. No hay nada!
31  sll t3, t3, 2 # x 4 #cuentas de arrays
32  addu t5, a0, t3
33  addiu t6, t5, 4
34  lw a0, 0(t5) #Cargo los parametros
35  lw a1, 0(t6)
36  #Llamo a la funcion el return value viene en v0
37  jal strcmpi
38  #recupero los temp necesarios
39  lw a1, 44(sp)
40  lw a0, 40(sp)
41  bgt v0, 0, if
42  #hago el else
43  sll t2, a1, 1
44  addiu t2, t2, 2
45  j fi
46  if: #hago el if
47  sll t2, a1, 1
48  addi t2, t2, 1
49  fi: #Guardo los tmp necesarios
50  sw t2, 16(sp)
51  #carga los parametrosi
52  sll t2, t2, 2
53  sll a1, a1, 2
54  addu t3, a0, t2
55  addu t4, a0, a1
56  lw a0, 0(t4)
57  lw a1, 0(t3)
58  #llamo a strcmp
59  jal strcmpi
60  blt v0, 0, if2
61  #recupero los tmp necesarios. No hay nada!
62  #hago el else
63  j fin_loop
64  if2: #hago el if
65  #Recupero los tmp necesarios.
66  lw a0, 40(sp)
67  lw a1, 44(sp)
68  lw a2, 48(sp)
69  lw t2, 16(sp)
70  #hago los intercambios
71  sll t3, a1, 2 #Index raiz x4 para array
72  sll t4, t2, 2 #Index hijo x4 para array
73  addu t3, a0, t3
74  addu t4, a0, t4
75  lw t1, 0(t3) # cargo el valor de words[raiz]
76  lw t5, 0(t4) # cargo el valor de words[hijo]
77  sw t5, 0(t3) #guardo en words[raiz] = words[hijo]
78  sw t1, 0(t4) #guardo en words[hijo] = tmp
79  sw t2, 44(sp) # raiz = hijo
80  move a1, t2
81  j loop
82  fin_loop:
83  #destruyo el stack
84  lw a0, 40(sp)
85  lw a1, 44(sp)
86  lw a2, 48(sp)
87  lw ra, 32(sp)
88  lw $fp, 28(sp)
89  lw gp, 24(sp)
90  addiu sp, sp, 40
91  j ra
92
93  #endif

```

B.4. *strcmpi.S*. Definición del algoritmo Strcmpi

Código 14: "strcmpi.S"

```
1 #ifndef USE_MIPS_ASSEMBLY
2 #define USE_MIPS_ASSEMBLY
3 #include <mips/regdef.h>
4 #include <sys/syscall.h>
5
6 # strcmpi.s :   Evalua dos strings para ver si son equivalentes o no.
7 #               No toma en cuenta mayusculas y minusculas para la evaluacion.
8 #
9 # Variables:
10 #   - a0 -> dir del primer array
11 #   - a1 -> dir del segundo array
12 #   - t1 -> int i (indice)
13 #   - t2 -> puntero para recorrer el array 'a'
14 #   - t3 -> puntero para recorrer el array 'b'
15 # Auxiliares:
16 #   - t4
17 #   - t5
18 #   - t6
19
20 .text
21 .align 2
22 .globl strcmpi
23
24
25 strcmpi:    subu sp, sp, 8 # Se crea el SRA 8 y LTA 0 y ABA 0 = 8 bytes
26             sw $30, 4(sp)
27             sw gp, 0(sp)
28             move $30, sp
29
30             addu t1, zero, zero # Se setea i = 0
31
32 recorrer:   addu t5, t1, a0 # t5 <- direccion de a[i]
33             lbu t2, 0(t5) # t2 <- a[i]
34             addu t5, t1, a1 # t5 <- direccion de b[i]
35             lbu t3, 0(t5) # t3 <- b[i]
36
37             beq t2, zero, terminoA # Si 'a' termina, se analiza si 'b' tambien
38             beq t3, zero, terminoB # Si 'b' termina, se analiza si 'a' tambien
39             beq t2, t3, sonIguales # Si son iguales, sigo analizando otra letra. Sino veo mayusculas y minusculas
40
41             addu t4, t2, zero # Se copia el valor de t2 en t4
42             addu t6, t4, zero # Se copia el valor de t4 en t6
43
44 caseSensitive: slti t5, t4, 90 # Se chequea si puede ser un char en mayusculas. (char < 90) -> 1
45               beq t5, zero, continue # Si no esta la posibilidad, se continua
46               slti t5, t4, 65 # Se chequea si es una letra mayuscula realmente. (char < 65) -> 1
47               bne t5, zero, continue # Si no es mayuscula, se continua
48
49               addu t6, t4, zero # Se guarda el valor de t4 para analizar corte
50               addiu t4, t4, 32 # Se transforma en minusculas
51
52 continue:   beq t6, t3, stop # Si se analizo la 2da letra, se sale del ciclo
53
54             addu t2, t4, zero # Sino, se guarda lo recién calculado en t2
55             addu t4, t3, zero # Se copia el valor de t3 en t4
56             addu t6, t4, zero # Se copia el valor de t4 en t6
57             b caseSensitive # Y se analiza la 2da letra
58
59
60 stop:       addu t3, t4, zero # Se guarda lo calculado en t3
61             bne t2, t3, salirDistintas # Son distintas
62
63 sonIguales: addiu t1, t1, 1 # i += 1
64             b recorrer # Se continua el loop
65
66 terminoA:   beq t3, zero, salirIguales # Si 'b' termina, se trata de dos cadenas iguales
67             b salirDistintas # Sino, se trata de cadenas diferentes
68
69 terminoB:   beq t2, zero, salirIguales # Si 'a' termina, se trata de dos cadenas iguales
70             b salirDistintas # Sino, se trata de cadenas diferentes
71
72 salirDistintas: sub v0, t2, t3 # Se devuelve el valor de restar (a - b)
73               b salir
74
75 salirIguales: addu v0, zero, zero # Se devuelve un 0
```

```
76         b salir
77
78 salir:    lw $30, 4(sp)
79          lw gp, 0(sp)
80          addiu sp, sp, 8
81          j ra
82 #endif
```
