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

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 desarrollo 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 grupo[1].

2. Compilación

La herramienta para compilar tanto el código asembly 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*: Imprimer la versión actual del programa;
- *-b [ARGS]*: El programa recibe nombres de archivos de texto o strings ingresados por *stdin*, ordenandolos utilizando el algoritmo Bubblersort. 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.

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

[2] Los archivos se encuentran separados según la implemetación 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 divida 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"*

```c
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA BUBBLESORT
 *
 * ******************************************************************************
 * ******************************************************************************/


#ifndef BUBBLESORT_H
#define BUBBLESORT_H


// Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void bubblesort(char* words[], int arraysize);


#endif
```

Código 2: *"bubblesort.c"*

```c
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA BUBBLESORT
 *
 * ******************************************************************************
 * ******************************************************************************/

#include "bubblesort.h"
#include <stdbool.h>
#include <string.h>


// Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
// arreglo de palabras.
```

```
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void bubblesort(char* words[], int arraysize)
{
  // Variables de procesamiento
  bool huboIntercambio;
  int i;
  int n = arraysize;
  char* sAux;

  // Recorremos el arreglo haciendo intercambios hasta que ya no se registre
  // ningun cambio realizado.
  do
  {
    huboIntercambio = false;

    for(i = 1; i < n; i++)
    {
      // Si el de indice menor es mayor que el de indice superior, los
      // intercambiamos
      if(strcasecmp(words[i-1], words[i]) > 0)
      {
        sAux = words[i-1];
        words[i-1] = words[i];
        words[i] = sAux;

        // Cambiamos el flag para registrar que hubo un cambio
        huboIntercambio = true;
      }
    }

    // Como el elemento del indice superior se encuentra ya ordenado una
    // vez finalizada la pasada, se reduce en uno la cantidad de indices
    // a iterar en la proxima pasada.
    n -= 1;

  } while(huboIntercambio);
}
```

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"*

```
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA HEAPSORT
 *
 * ******************************************************************************
 * ******************************************************************************/



#ifndef HEAPSORT_H
#define HEAPSORT_H



// Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void heapsort(char* words[], int arraysize);



#endif
```

Código 4: *"heapsort.c"*

```c
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA HEAPSORT
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "heapsort.h"
#include <string.h>


void heap(char* words[],int raiz, int fin){
    char finished = 0;
    char* tmp;
    int hijo;
    while (raiz*2+1 <= fin && !finished){
        //Elijo el hijo mayor
        if(raiz*2+1==fin || strcasecmp(words[raiz*2+1],words[raiz*2+2]) > 0 ){
            hijo=raiz*2+1;
        }else{
            hijo=raiz*2+2;
        }
        //Checkeo si es mayor e intercambio
        if(strcasecmp(words[raiz],words[hijo]) < 0){
            tmp=words[raiz];
            words[raiz]=words[hijo];
            words[hijo]=tmp;
            raiz=hijo;
        }else{
            finished=1;
        }
    }
}



// Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void heapsort(char* words[], int arraysize){
    int i;
    char* tmp;
    //Armo Heap
    for (i=(arraysize/2)-1;i>=0;i--){
        heap(words,i,arraysize-1);
    }

    //Extraigo raiz y armo heap
    for(i=arraysize-1;i>0;i--){
        tmp=words[0];
        words[0]=words[i];
        words[i]=tmp;
        heap(words,0,i-1);
    }
}
```

4.2. Implementación en Assembly

La implementación del programa fue divida en los siguientes módulos:

- **tp1**: Solamente recibe un texto como argumento por linea de comandos y lo imprime ordenandolo 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"*

```
#ifndef USE_MIPS_ASSEMBLY
#define USE_MIPS_ASSEMBLY
#include <mips/regdef.h>
#include <sys/syscall.h>


# Utilizacion de registros:
#
# s0: words
# s1: arraysize
# s2: variable local "i"
# s3: variable local "tmp"
# t0: variable local auxiliar
# t1: variable local auxiliar
#


.text
.align  2
.globl  heapsort
.extern heap

      # ABA: 16; SRA: 16; LTA:24
heapsort: subu  sp,sp,56        # Creamos stack frame.
      sw    ra,52(sp)       # Return address
      sw    $fp,48(sp)        # Frame pointer
      sw    gp,44(sp)       # Global pointer
      move  $fp,sp          # Establecemos la base

      # Almacenamos contenido de registros
      sw    s0,40(sp)
      sw    s1,36(sp)
      sw    s2,32(sp)
      sw    s3,28(sp)

      # Almacenamos los parametros recibidos
      sw    a0,56(sp)       # words
      sw    a1,60(sp)       # arraysize

      # Almacenamos los parametros guardados en fp en variables
      move  s0,a0         # words
      move  s1,a1           # arraysize

      # Armamos el heap
      srl   t0,s1,1         # t0 = arraysize / 2
      subu  s2,t0,1         # i = arraysize / 2 - 1
FOR_1:    slt   t0,s2,zero        # i es menor que 0
      bne   t0,zero,FOR_1_FIN   # Si t0 = 1 saltamos a FOR_1_FIN

      add   a1,s2,zero        # Cargamos i en a1
      subu  a2,s1,1         # Cargamos arraysize-1 en a2
      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:

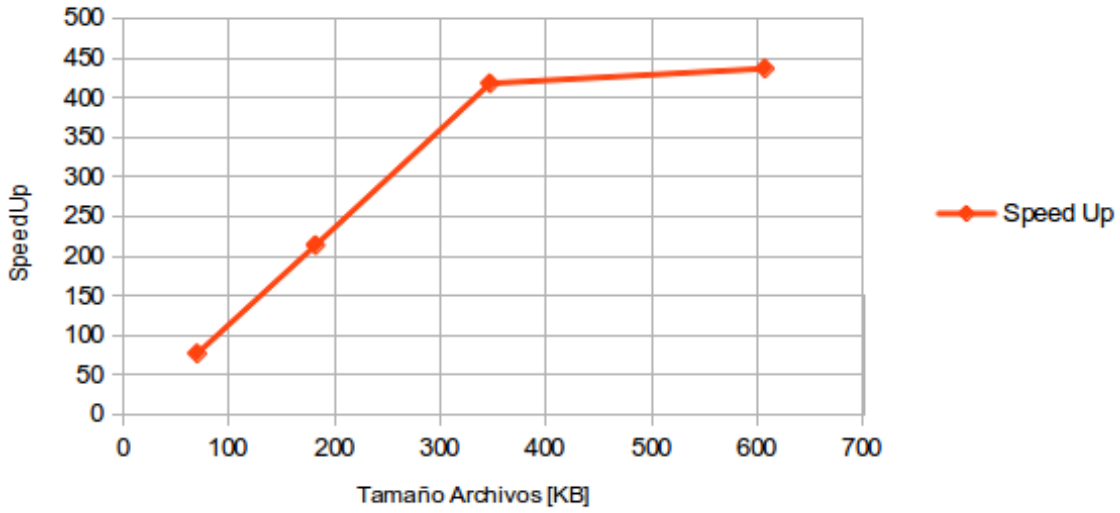

Figura 1: *Gráfico del speedup del Bubblesort contra el Heapsort.*

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.*

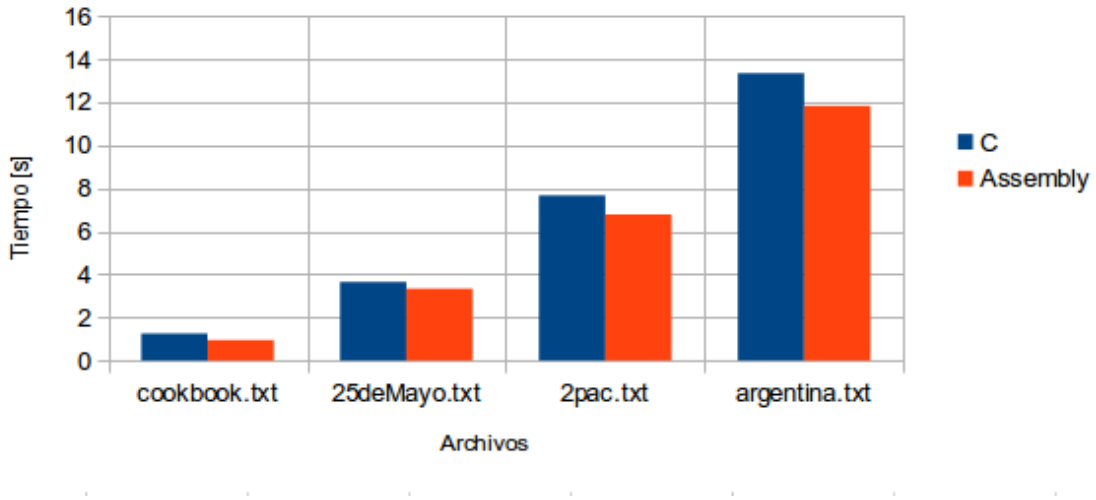

Figura 2: *Gráfico comparando la performance del Heapsort en sus dos implementaciones (C vs Assembly).*

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

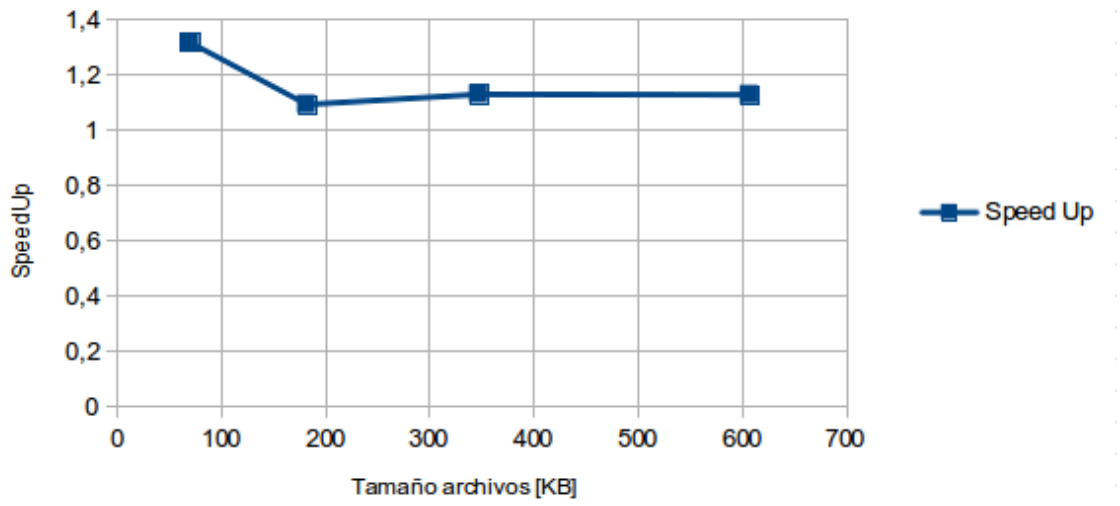

Figura 3: *Gráfico del speedup del Heapsort en C contra la versión en Assembly.*

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 em 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"*

```c
/*
 ============================================================================
 Name        : tp1.c
 Authors     : Belen Beltran (91718)
       Federico Martin Rossi (92086)
       Pablo Rodriguez (93970)
 Version     : 1.0
 Description : Programa de ordenamiento heapsort o bubblesort
 ============================================================================
 */

#include "heapsort.h"
#include "bubblesort.h"
#include <stdio.h>
#include <string.h>
#include <getopt.h>
void* malloc(size_t);
void* realloc(void*,size_t);
void free(void*);
int system(const char* cadena);

/** Funciones Auxiliares **/
void readWord(FILE* fd,char* word){
  int index=0;
  int c;
  while((c=getc(fd))!=EOF && c!=32 && c!='\n'  && index < 49){
    if((c>=48 && c<=57) || (c>=65 && c<=90) ||
      (c>=97 && c<=122) || (c>=130 && c<=165)){
      word[index++]=c;
    }
  }
  word[index]='\0';
}

// Devuelve la cantidad de palabras leidas
char** leerArchivo(char* sourceName, int* cant) {
    int size = 0;
  int tamano = 5000;
  int tam;
  char word[50];
  FILE* sourcefd;
  char** palabras;


  // Se toma '-' como stdin
    if (sourceName[0] == '-')
        sourcefd = stdin;
    // Se abre el archvio correspondiente
    else {
        sourcefd = fopen(sourceName,"r");
        if (!sourcefd) {
          fprintf(stderr, "El archivo no pudo ser abierto\n");
          return 0;
        }
    // Si el archivo esta vacio
    fseek(sourcefd, 0, SEEK_END);
    if ( ftell(sourcefd) == 0 ) {
      fprintf(stderr, "El archivo esta vacio\n");
      return 0;
    }

    // Se vuelve al principio
```

```
    fseek(sourcefd, 0, SEEK_SET);
    }

  /* Cargo el archivo a memoria*/
  palabras = (char**)malloc(tamano*sizeof(char*));

  // Se toma '-' como stdin
  if (sourceName[0] == '-')
    sourcefd = stdin;
  // Se abre el archvio correspondiente
  else
    sourcefd = fopen(sourceName,"r");

  readWord(sourcefd,word);
  while (!feof(sourcefd)){
    if (size >= tamano){
      tamano += 5000;
      palabras = (char**)realloc(palabras, tamano*sizeof(char*));
    }
    tam = strlen(word) + 1;
    if (palabras == NULL) {
      fprintf(stderr,"\n\nNo se pueden alocar mas palabras\n\n");
      break;
    }
    palabras[size] = malloc(tam);
    memcpy(palabras[size], word, tam);
    size++;
    readWord(sourcefd, word);
  }
  // Se cierra el archivo si corresponde
  if (sourceName[0] != '-')
    fclose(sourcefd);

  *cant = size;

  return palabras;
}

void displayAyuda() {
  printf("%s", "Usage: ./tp1 [OPCION] [ARCHIVO]\n\
  \nOPCION:\n\
  -V, --version\tImprime la version del programa.\n\
  -h, --help\tImprime esta ayuda. \n\
  -b, --bubblesort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\
  -p, --heapsort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\
  NOTA: De no recibirse una OPCION, se podran recibir nombres de ARCHIVO, resultando en la\
 concatenacion e impresion por pantalla de estos.\
  \nEjemplos:\n\
  tp1 -b palabras.txt\n\
  tp1 -p palabras.txt\n");
}

void displayVersion() {
  printf("%s", "Esta aplicacion ejecuta un ordenamiento sobre las palabras contenidas en un archivo.\
 Puede ejecutarse el algoritmo heapsort o bubblesort.\
  \nVersion: v1.0\n");
}

void ejecutarHeapsort(char* nombreArchivo) {
  int i, size;
  char** palabras = leerArchivo(nombreArchivo, &size);

  if (!palabras)
    return;

  /*Ordeno*/
    heapsort(palabras,size);

  /*Imprimo el resultado y libero memoria*/
    for(i=0;i<size;i++){
        printf("%s ",palabras[i]);
      free(palabras[i]);
  }
  free(palabras);

  printf("\n");
```

```
}

void ejecutarBubblesort(char* nombreArchivo) {
  int i, size;
  char** palabras = leerArchivo(nombreArchivo, &size);

  if (!palabras)
    return;

  /*Ordeno*/
    bubblesort(palabras,size);

  /*Imprimo el resultado y libero memoria*/
    for(i=0;i<size;i++){
        printf("%s ",palabras[i]);
      free(palabras[i]);
  }
  free(palabras);

  printf("\n");
}


int main (int argc, char* argv[]) {
  int c;
  FILE* sourcefd;

  // Si hay pocos argumentos
  if (argc < 2) {
    fprintf(stderr, "ERROR: No hay suficientes argumentos\n");
    return 1;
  }


  // while (1) {
  static struct option opciones[] = {
    // No setean flags
    {"help", no_argument, 0, 'h'},
    {"version", no_argument, 0, 'V'},
    {"bubble", required_argument, 0, 'b'},
    {"heap", required_argument, 0, 'p'},
    {0,0,0,0}
  };
  // getopt_long stores the option index here.
  int indice_opciones = -1;

  int hayOpciones = 0;

  while( (c = getopt_long (argc, argv, "hVb:p:", opciones, &indice_opciones)) != -1 ) {

    hayOpciones = 1;

    switch (c) {
      case 0:
        printf ("Estoy en 0. Option %s", opciones[indice_opciones].name);
        if (optarg)
          printf (" with arg %s", optarg);
        printf ("\n");
        break;

      case 'h':
        if (argc == 2)
          displayAyuda();
        break;

      case 'V':
        if (argc == 2)
          displayVersion();
        break;

      case 'p':
        if (argc == 3)
          ejecutarHeapsort(optarg);
        break;

      case 'b':
```

```
        if (argc == 3)
          ejecutarBubblesort(optarg);
        break;

      default:
        break;
    }
   }
   // Si hay argumentos que no son opciones, se buscan si nos nombres de archivo
  if (optind < argc && !hayOpciones) {
    char op[256];
    while (optind < argc) {
      sprintf(op, "cat %s", argv[optind++]);
      system(op);
    }
  }

  return 0;
}
```

A.2. *bubblesort.h*. Declaración del algoritmo Bubblesort

Código 7: *"bubblesort.h"*

```
/* ********************************************************************************
 * ********************************************************************************
 *
 * LIBRERIA BUBBLESORT
 *
 * ********************************************************************************
 * ********************************************************************************/



#ifndef BUBBLESORT_H
#define BUBBLESORT_H



// Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void bubblesort(char* words[], int arraysize);



#endif
```

A.3. *bubblesort.c*. Definición del algoritmo Bubblesort

Código 8: *"bubblesort.c"*

```
/* ********************************************************************************
 * ********************************************************************************
 *
 * LIBRERIA BUBBLESORT
 *
 * ********************************************************************************
 * ********************************************************************************/


#include "bubblesort.h"
```

```c
#include <stdbool.h>
#include <string.h>



// Funcion que aplica el algoritmo de ordenamiento Bubblesort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void bubblesort(char* words[], int arraysize)
{
  // Variables de procesamiento
  bool huboIntercambio;
  int i;
  int n = arraysize;
  char* sAux;

  // Recorremos el arreglo haciendo intercambios hasta que ya no se registre
  // ningun cambio realizado.
  do
  {
    huboIntercambio = false;

    for(i = 1; i < n; i++)
    {
      // Si el de indice menor es mayor que el de indice superior, los
      // intercambiamos
      if(strcasecmp(words[i-1], words[i]) > 0)
      {
        sAux = words[i-1];
        words[i-1] = words[i];
        words[i] = sAux;

        // Cambiamos el flag para registrar que hubo un cambio
        huboIntercambio = true;
      }
    }

    // Como el elemento del indice superior se encuentra ya ordenado una
    // vez finalizada la pasada, se reduce en uno la cantidad de indices
    // a iterar en la proxima pasada.
    n -= 1;

  } while(huboIntercambio);
}
```

A.4. *heapsort.h*. Declaración del algoritmo Heapsort

Código 9: *"heapsort.h"*

```c
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA HEAPSORT
 *
 * ******************************************************************************
 * ******************************************************************************/



#ifndef HEAPSORT_H
#define HEAPSORT_H



// Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
```

```
// POST: el arreglo 'words' queda ordenado.
void heapsort(char* words[], int arraysize);


#endif
```

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

Código 10: *"heapsort.c"*

```
/* ******************************************************************************
 * ******************************************************************************
 *
 * LIBRERIA HEAPSORT
 *
 * ******************************************************************************
 * ******************************************************************************/


#include "heapsort.h"
#include <string.h>


void heap(char* words[],int raiz, int fin){
    char finished = 0;
    char* tmp;
    int hijo;
    while (raiz*2+1 <= fin && !finished){
        //Elijo el hijo mayor
        if(raiz*2+1==fin || strcasecmp(words[raiz*2+1],words[raiz*2+2]) > 0 ){
            hijo=raiz*2+1;
        }else{
            hijo=raiz*2+2;
        }
        //Checkeo si es mayor e intercambio
        if(strcasecmp(words[raiz],words[hijo]) < 0){
            tmp=words[raiz];
            words[raiz]=words[hijo];
            words[hijo]=tmp;
            raiz=hijo;
        }else{
            finished=1;
        }
    }
}



// Funcion que aplica el algoritmo de ordenamiento Heapsort para ordenar un
// arreglo de palabras.
// PRE: 'words' es un puntero a un arreglo de punteros a caracter; 'arraysize'
// es el tamanio de dicho arreglo.
// POST: el arreglo 'words' queda ordenado.
void heapsort(char* words[], int arraysize){
    int i;
    char* tmp;
    //Armo Heap
    for (i=(arraysize/2)-1;i>=0;i--){
        heap(words,i,arraysize-1);
    }

    //Extraigo raiz y armo heap
    for(i=arraysize-1;i>0;i--){
        tmp=words[0];
        words[0]=words[i];
        words[i]=tmp;
        heap(words,0,i-1);
    }
}
```

B. Implementación completa en lenguaje Assembly MIPS32

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

Código 11: *"tp1.c"*

```
/*
 ============================================================================
 Name         : tp1.c
 Authors      : Belen Beltran (91718)
        Federico Martin Rossi (92086)
        Pablo Rodriguez (93970)
 Version      : 1.0
 Description  : Programa de ordenamiento heapsort
 ============================================================================
 */

#include <stdio.h>
#include <string.h>
#include <getopt.h>
#include "bubblesort.h"

void* malloc(size_t);
void* realloc(void*,size_t);
void free(void*);
int system(const char* cadena);
extern heapsort(char**, int);

/** Funciones Auxiliares **/
void readWord(FILE* fd,char* word){
  int index=0;
  int c;
  while((c=getc(fd))!=EOF && c!=32 && c!='\n'  && index < 49){
    if((c>=48 && c<=57) || (c>=65 && c<=90) ||
      (c>=97 && c<=122) || (c>=130 && c<=165)){
      word[index++]=c;
    }
  }
  word[index]='\0';
}

// Devuelve la cantidad de palabras leidas
char** leerArchivo(char* sourceName, int* cant) {
    int size = 0;
  int tamano = 5000;
  int tam;
  char word[50];
  FILE* sourcefd;
  char** palabras;


  // Se toma '-' como stdin
    if (sourceName[0] == '-')
        sourcefd = stdin;
    // Se abre el archvio correspondiente
    else {
        sourcefd = fopen(sourceName,"r");
        if (!sourcefd) {
          fprintf(stderr, "El archivo no pudo ser abierto\n");
          return 0;
        }
    // Si el archivo esta vacio
    fseek(sourcefd, 0, SEEK_END);
    if ( ftell(sourcefd) == 0 ) {
      fprintf(stderr, "El archivo esta vacio\n");
      return 0;
    }

    // Se vuelve al principio
    fseek(sourcefd, 0, SEEK_SET);
    }

  /* Cargo el archivo a memoria*/
  palabras = (char**)malloc(tamano*sizeof(char*));
```

```
69
70    // Se toma '-' como stdin
71    if (sourceName[0] == '-')
72      sourcefd = stdin;
73    // Se abre el archvio correspondiente
74    else
75      sourcefd = fopen(sourceName,"r");
76
77    readWord(sourcefd,word);
78    while (!feof(sourcefd)){
79      if (size >= tamano){
80        tamano += 5000;
81        palabras = (char**)realloc(palabras, tamano*sizeof(char*));
82      }
83      tam = strlen(word) + 1;
84      if (palabras == NULL) {
85        fprintf(stderr,"\n\nNo se pueden alocar mas palabras\n\n");
86        break;
87      }
88      palabras[size] = malloc(tam);
89      memcpy(palabras[size], word, tam);
90      size++;
91      readWord(sourcefd, word);
92    }
93    // Se cierra el archivo si corresponde
94    if (sourceName[0] != '-')
95      fclose(sourcefd);
96
97    *cant = size;
98
99    return palabras;
100 }
101
102 void displayAyuda() {
103   printf("%s", "Usage: ./tp1 [OPCION] [ARCHIVO]\n\
104   \nOPCION:\n\
105   -V, --version\tImprime la version del programa.\n\
106   -h, --help\tImprime esta ayuda. \n\
107   -b, --bubblesort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\
108   -p, --heapsort\tEjecuta el algoritmo bubblesort en el ARCHIVO recibido por parametros.\n\
109   NOTA: De no recibirse una OPCION, se podran recibir nombres de ARCHIVO, resultando en la\
110  concatenacion e impresion por pantalla de estos.\
111   \nEjemplos:\n\
112   tp1 -b palabras.txt\n\
113   tp1 -p palabras.txt\n");
114 }
115
116 void displayVersion() {
117   printf("%s", "Esta aplicacion ejecuta un ordenamiento sobre las palabras contenidas en un archivo.\
118  Puede ejecutarse el algoritmo heapsort o bubblesort.\
119   \nVersion: v1.0\n");
120 }
121
122 void ejecutarHeapsort(char* nombreArchivo) {
123   int i, size;
124   char** palabras = leerArchivo(nombreArchivo, &size);
125
126   if (!palabras)
127     return;
128
129   /*Ordeno*/
130     heapsort(palabras,size);
131
132   /*Imprimo el resultado y libero memoria*/
133     for(i=0;i<size;i++){
134         printf("%s ",palabras[i]);
135       free(palabras[i]);
136   }
137   free(palabras);
138
139   printf("\n");
140 }
141
142 void ejecutarBubblesort(char* nombreArchivo) {
143   int i, size;
144   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     }
```

```
   }
   // Si hay argumentos que no son opciones, se buscan si nos nombres de archivo
  if (optind < argc && !hayOpciones) {
    char op[256];
    while (optind < argc) {
      sprintf(op, "cat %s", argv[optind++]);
      system(op);
    }
  }

  return 0;
 }
```

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

Código 12: *"heapsort.S"*

```
#ifndef USE_MIPS_ASSEMBLY
#define USE_MIPS_ASSEMBLY
#include <mips/regdef.h>
#include <sys/syscall.h>


# Utilizacion de registros:
#
# s0: words
# s1: arraysize
# s2: variable local "i"
# s3: variable local "tmp"
# t0: variable local auxiliar
# t1: variable local auxiliar
#


.text
.align  2
.globl  heapsort
.extern heap

      # ABA: 16; SRA: 16; LTA:24
heapsort: subu  sp,sp,56        # Creamos stack frame.
      sw    ra,52(sp)       # Return address
      sw    $fp,48(sp)        # Frame pointer
      sw    gp,44(sp)       # Global pointer
      move  $fp,sp          # Establecemos la base

      # Almacenamos contenido de registros
      sw    s0,40(sp)
      sw    s1,36(sp)
      sw    s2,32(sp)
      sw    s3,28(sp)

      # Almacenamos los parametros recibidos
      sw    a0,56(sp)       # words
      sw    a1,60(sp)       # arraysize

      # Almacenamos los parametros guardados en fp en variables
      move  s0,a0         # words
      move  s1,a1           # arraysize

      # Armamos el heap
      srl   t0,s1,1         # t0 = arraysize / 2
      subu  s2,t0,1         # i = arraysize / 2 - 1
FOR_1:    slt   t0,s2,zero        # i es menor que 0
      bne   t0,zero,FOR_1_FIN   # Si t0 = 1 saltamos a FOR_1_FIN

      add   a1,s2,zero        # Cargamos i en a1
      subu  a2,s1,1         # Cargamos arraysize-1 en a2
      jal   heap          # Saltamos a funcion "heap"

```

```
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
```

```
  sll t3, a1, 1 #raiz*2
  addi t3, t3, 1 #+1
  bgt t3, a2, fin_loop  # > fin
  #comparaciones de if
  subu t4, t3, a2
  beq t4, 0, if
  #Guardo todos los tmp porque llamo una funcion. No hay nada!
  sll t3, t3, 2 # x 4 #cuentas de arrays
  addu t5, a0, t3
  addiu t6, t5, 4
  lw a0, 0(t5) #Cargo los parametros
  lw a1, 0(t6)
  #Llamo a la funcion el return value viene en v0
  jal strcmpi
  #recupero los temp necesarios
  lw a1, 44(sp)
  lw a0, 40(sp)
  bgt v0, 0, if
  #hago el else
  sll t2, a1, 1
  addiu t2, t2, 2
  j fi
if: #hago el if
  sll t2, a1, 1
  addi t2, t2, 1
fi: #Guardo los tmp necesarios
  sw t2, 16(sp)
  #cargo los parametrosi
  sll t2, t2, 2
  sll a1, a1, 2
  addu t3, a0, t2
  addu t4, a0, a1
  lw a0, 0(t4)
  lw a1, 0(t3)
  #llamo a strcmp
  jal strcmpi
  blt v0, 0, if2
  #recupero los tmp necesarios. No hay nada!
  #hago el else
  j fin_loop
if2:  #hago el if
  #Recupero los tmp necesarios.
  lw a0, 40(sp)
  lw a1, 44(sp)
  lw a2, 48(sp)
  lw t2, 16(sp)
  #hago los intercambios
  sll t3, a1, 2 #Index raiz x4 para array
  sll t4, t2, 2 #Index hijo x4 para array
  addu t3, a0, t3
  addu t4, a0, t4
  lw t1, 0(t3) # cargo el valor de words[raiz]
  lw t5, 0(t4) # cargo el valor de words[hijo]
  sw t5, 0(t3)  #guardo en words[raiz] = words[hijo]
  sw t1, 0(t4) #guardo en words[hijo] = tmp
  sw t2, 44(sp) # raiz = hijo
  move a1, t2
  j loop
fin_loop:
  #destruyo el stack
  lw a0, 40(sp)
  lw a1, 44(sp)
  lw a2, 48(sp)
  lw ra, 32(sp)
  lw $fp, 28(sp)
  lw gp, 24(sp)
  addiu sp, sp, 40
  j ra

#endif
```

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

Código 14: *"strcmpi.S"*

```
#ifndef USE_MIPS_ASSEMBLY
#define USE_MIPS_ASSEMBLY
#include <mips/regdef.h>
#include <sys/syscall.h>

# strcmpi.s :   Evalua dos strings para ver si son equivalentes o no.
#         No toma en cuenta mayusculas y minusculas para la evaluacion.
#
# Variables:
#       - a0 -> dir del primer array
#       - a1 -> dir del segundo array
#       - t1 -> int i (indice)
#       - t2 -> puntero para recorrer el array 'a'
#       - t3 -> puntero para recorrer el array 'b'
# Auxiliares:
#       - t4
#       - t5
#       - t6

.text
.align 2
.globl strcmpi


strcmpi:    subu sp, sp, 8 # Se crea el SRA 8 y LTA 0 y ABA 0 = 8 bytes
            sw $30, 4(sp)
            sw gp, 0(sp)
            move $30, sp

            addu t1, zero, zero # Se setea i = 0

recorrer:     addu t5, t1, a0 # t5 <- direccion de a[i]
        lbu t2, 0(t5) # t2 <- a[i]
        addu t5, t1, a1 # t5 <- direccion de b[i]
        lbu t3, 0(t5) # t3 <- b[i]

        beq t2, zero, terminoA # Si 'a' termina, se analiza si 'b' tambien
        beq t3, zero, terminoB # Si 'b' termina, se analiza si 'a' tambien
        beq t2, t3, sonIguales # Si son iguales, sigo analizando otra letra. Sino veo mayusculas y minusculas

        addu t4, t2, zero # Se copia el valor de t2 en t4
        addu t6, t4, zero # Se copia el valor de t4 en t6

caseSensitive:  slti t5, t4, 90 # Se chequea si puede ser un char en mayusculas. (char < 90) -> 1
        beq t5, zero, continue # Si no esta la posibilidad, se continua
        slti t5, t4, 65 # Se chequea si es una letra mayuscula realmente. (char < 65) -> 1
        bne t5, zero, continue # Si no es mayuscula, se continua

        addu t6, t4, zero # Se guarda el valor de t4 para analizar corte
        addiu t4, t4, 32 # Se transforma en minusculas

continue:   beq t6, t3, stop # Si se analizo la 2da letra, se sale del ciclo

        addu t2, t4, zero # Sino, se guarda lo recien calculado en t2
        addu t4, t3, zero # Se copia el valor de t3 en t4
        addu t6, t4, zero # Se copia el valor de t4 en t6
        b caseSensitive # Y se analiza la 2da letra


stop:     addu t3, t4, zero # Se guarda lo calculado en t3
        bne t2, t3, salirDistintas # Son distintas

sonIguales:   addiu t1, t1, 1 # i += 1
        b recorrer # Se continua el loop

terminoA:   beq t3, zero, salirIguales # Si 'b' termina, se trata de dos cadenas iguales
        b salirDistintas # Sino, se trata de cadenas diferentes

terminoB:   beq t2, zero, salirIguales # Si 'a' termina, se trata de dos cadenas iguales
        b salirDistintas # Sino, se trata de cadenas diferentes

salirDistintas: sub v0, t2, t3 # Se devuelve el valor de restar (a - b)
        b salir

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
```