Trabajo práctico 0: Infraestructura básica

Contini, Agustín - *Padrón 89180*
agscontini@gmail.com
Farina, Federico - *Padrón 90177*
federicojosefarina@gmail.com
Prystupiuk, Maximiliano - *Padrón 94*
mprystupiuk@gmail.com

1er. Cuatrimestre de 2017

66.20 Organización de Computadoras
Facultad de Ingeniería, Universidad de Buenos Aires

March 30, 2017

Abstract

El trabajo práctico consiste en implementar el algoritmo de ordenamiento Quicksort y BubbleSort en lenguaje C, las funciones swap y compare, para luego comparar sus respectivas performance y la cantidad de llamados a swap y compare.

Contents

1 Introducción **3**
1.1 Bubblesort . . . 3
1.2 Quicksort . . . 3

2 El programa **4**
2.1 Compilado . . . 4
2.2 Uso . . . 4

3 Desarrollo **5**
3.1 Estructura interna del programa . . . 5
3.2 Aclaraciones . . . 5
3.3 Mediciones de tiempo de ejecución . . . 6
3.3.1 Resultado de las ejecuciones . . . 6
3.3.2 Gráficos comparando los swaps de los sort . . . 6
3.3.3 Gráficos comparando los compare de los sort . . . 6
3.3.4 Gráficos comparando los tiempos de los sort por palabras . . . 6
3.3.5 Cálculo del Speedup . . . 8

4 Conclusiones **9**

5 Código **10**
5.1 sorters.h . . . 10
5.2 sorters.c . . . 10
5.3 main.c . . . 11

List of Figures

1 Tabla de los tiempos del algoritmo de ordenamiento bubblesort . . . 6
2 Tabla de los tiempos del algoritmo de ordenamiento quicksort . . . 6
3 Gráfico comparando la cantidad de swaps de los sorts corridos en Linux . . . 6
4 Gráfico comparando la cantidad de compare de los sorts corridos en Linux . . . 7
5 Gráfico comparando la cantidad de tiempo de los sorts corridos en Linux por palabras del documento . . . 7
6 Tabla de los speedups obtenidos de los sorts . . . 8
7 Gráfico del speedup entre Bubblsort en C y Quicksort en C . . . 8

1 Introducción

1.1 Bubblesort

El Bubblesort es un algoritmo de ordenamiento que funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Dado que sólo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

1.2 Quicksort

Se elige un elemento de la lista de elementos a ordenar, al que llamaremos pivote. Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada. La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha. Se repite este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados. Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es O(n·log n). En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente. En el caso promedio, el orden es O(n·log n). No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

2 El programa

2.1 Compilado

Se incluye junto a este trabajo un Makefile para compilar el programa. Simplemente debe abrirse una Terminal en el directorio root. Poner "mkdir build", "cd build", "cmake ..", "make".

2.2 Uso

El programa puede leer tanto datos desde stdin como de archivos pasados por parámetro. La salida del programa es por stdout y la de errores por stderr. Forma de uso:

```
Usage:
  ./tp0 -h
  ./tp0 -V
./tp0 [OPTIONS] [file...]
  -h, --help       Print this and exit.
  -V, --version    Print version and exit.
  -o, --output     Path to output file.
  -i, --input      Path to input file.
  -b, --bsort      Use bubblesort.
  -q, --qsort      Use quicksort.
```

3 Desarrollo

3.1 Estructura interna del programa

Se desarrollaron los algoritmos con vistas a cumplir el objetivo del programa. A continuación se realiza una descripción de cada uno:

- 1. Se detectan los argumentos de entrada por línea de comandos, mediante una comparación de caracteres.
- 2. Para los argumentos help y Version, se imprimen cadenas de texto con la información básica de uso, y de la versión e integrantes del trabajo.
- 3. Para los argumentos Bubblesort y Quicksort, se siguen una serie de pasos:
  - a) Si se detectan más argumentos de entrada, se lo utiliza como archivos de entrada. En caso contrario, se abre la entrada estándar stdin.
  - b) Se abren los archivos de entrada en el orden de la línea de comandos y se realiza un ciclo sobre cada uno de ellos, hasta el fin de archivo. En dicho ciclo se realiza un parseo del archivo y se almacena cada palabra en un vector común para todos los archivos. Este vector trabaja con memoria dinámica.
  - c) Luego de procesar todos los archivos se llama al ordenamiento pedido por parámetro.
  - d) Luego se procede a imprimir por la salida estándar stdout. Se imprime palabra por palabra, recorriendo el vector ya ordenado.
  - e) Por último, se libera la memoria pedida.

3.2 Aclaraciones

La razón por la cual decidimos hacer una versión corrida para Linux y otra para Gxemul es para poder tener tiempos comparables. Para los speedups, decidimos correr las pruebas de tiempos de Bubblesort y Quicksort en C en Linux y en Gxemul. Para que los tiempos para las comparaciones y speedups fueran comparables, cada sort para cada texto a comparar fueron corridas en ambos sistemas operativos.

3.3 Mediciones de tiempo de ejecución

3.3.1 Resultado de las ejecuciones

BubbleSort	Alice	Beowulf	Cyclopedia	Elquijote
#palabras	30355	37048	105582	354258
#swap()	230926667	350206392	2908894943	36655791348
#compare()	459751835	686185857	5572929618	73826819192
ejecución (segundos)	345,383	538,274	5424,19	71833,06

Figure 1: Tabla de los tiempos del algoritmo de ordenamiento bubblesort

Quicksort	Alice	Beowulf	Cyclopedia	Elquijote
#swap()	233313	286457	1025182	3362463
#compare()	3876645	8253090	43621725	767627337
ejecución (segundos)	3	6	36	633
Tamaños	177	225	659	2.199

Figure 2: Tabla de los tiempos del algoritmo de ordenamiento quicksort

3.3.2 Gráficos comparando los swaps de los sort

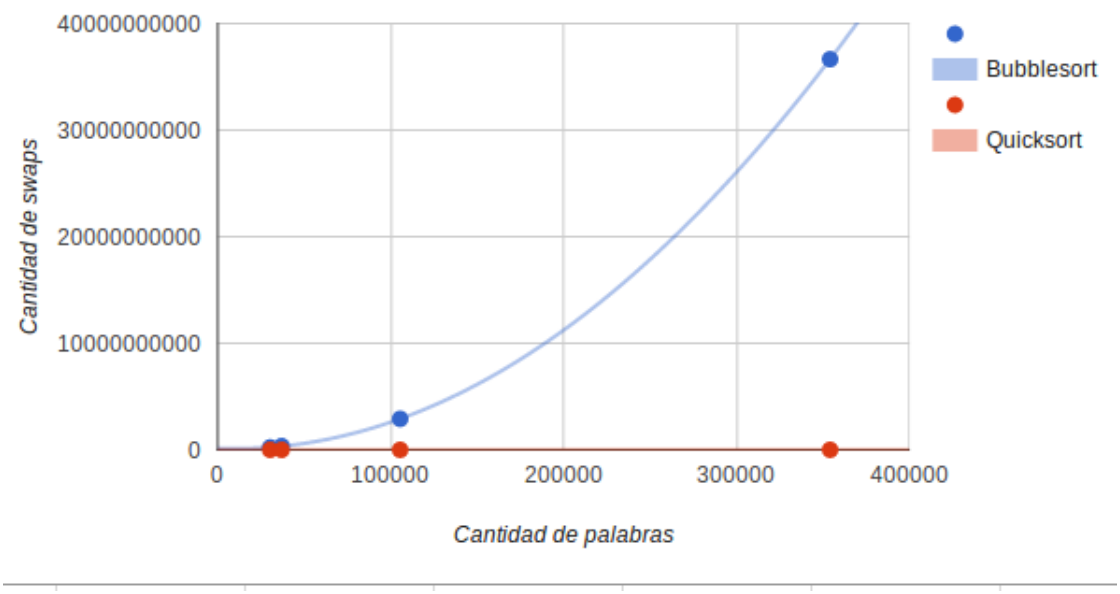

Figure 3: Gráfico comparando la cantidad de swaps de los sorts corridos en Linux

3.3.3 Gráficos comparando los compare de los sort

Como podemos apreciar, con escala lineal para el eje y (tiempo tardado del sort), por la abismal diferencia entre el Bubblesort y Quicksort, ésta última no posee pendiente. Así que

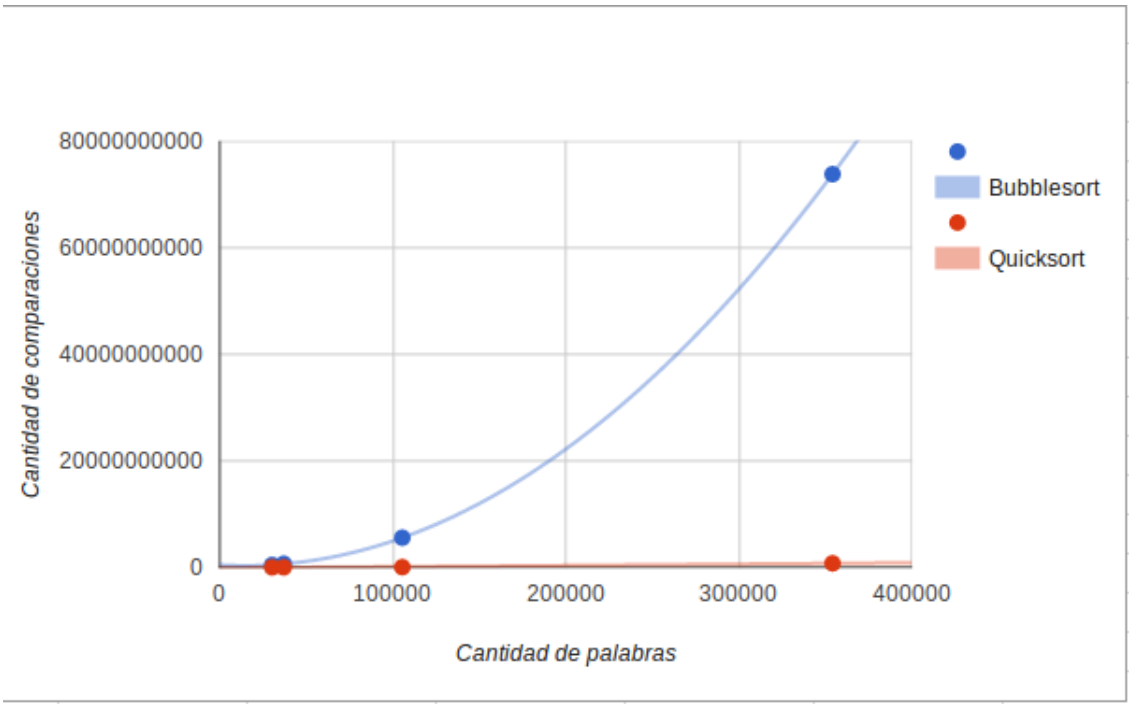

Figure 4: Gráfico comparando la cantidad de compare de los sorts corridos en Linux

presentamos el mismo gráfico, pero con escala logarítmica en el eje y, para poder ver mejor las diferencias.

3.3.4 Gráficos comparando los tiempos de los sort por palabras

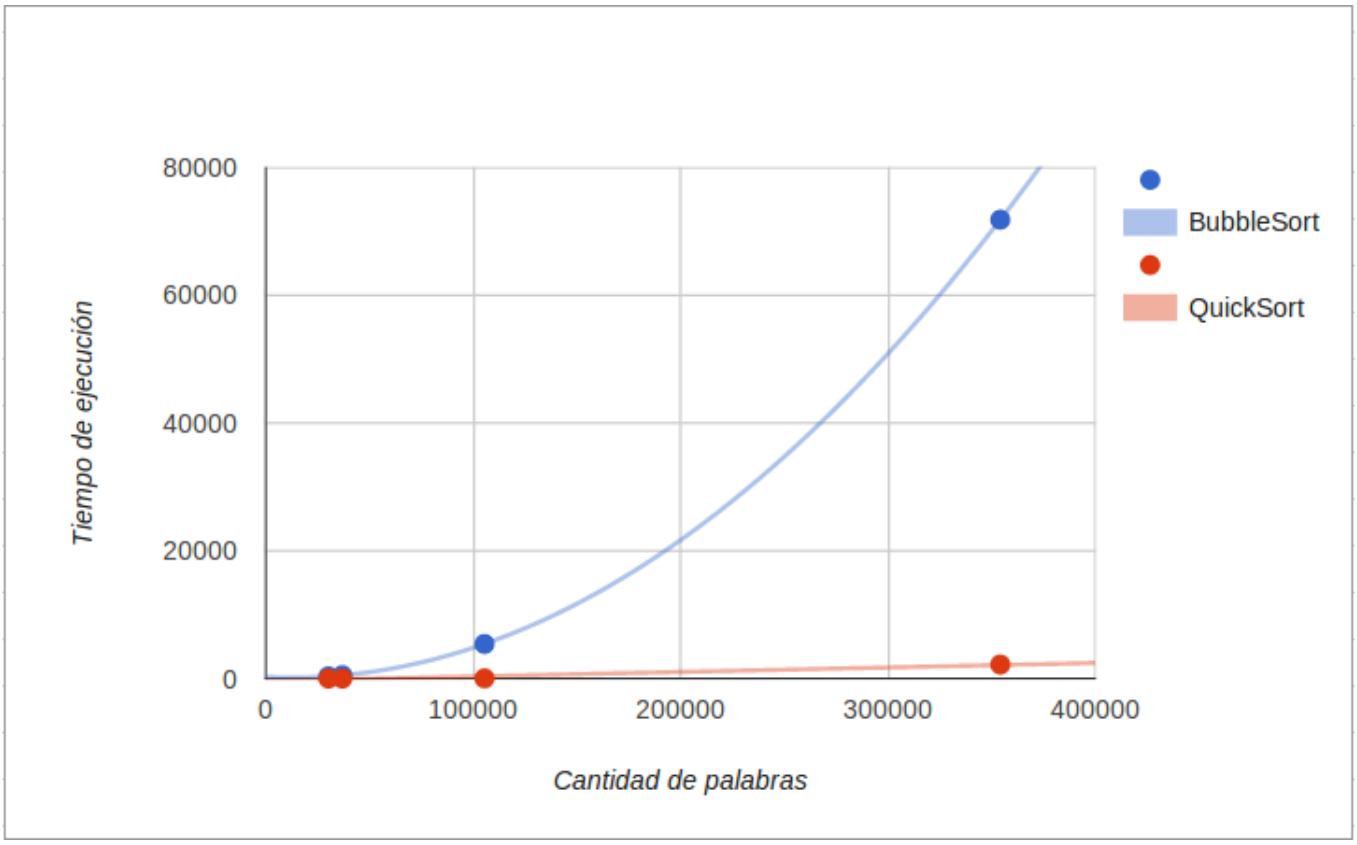

Figure 5: Gráfico comparando la cantidad de tiempo de los sorts corridos en Linux por palabras del documento

Podemos observar que a medida que el tamaño de los archivos aumenta, el Bubblesort tarda cada vez más en ordenar las palabras, mientras que el Quicksort tiene una pendiente menos pronunciada. Esto se debe a las características inherentes de los algoritmos. El Bubblesort no es un algoritmo eficiente ya que realiza muchas comparaciones innecesarias al no presentar un procedimiento que aproveche las comparaciones ya realizadas. Esto hace que se vuelva muy lento, lo cual se acentúa cuanto mayor es el número de palabras a ordenar.

Para el caso del algoritmo Quicksort implementado en C se puede observar que es ampliamente superior al Bubblesort y que el tiempo demorado es muy inferior en ambos ssoo. El compilador de C, al estar ya optimizado para pasar del lenguaje C a assembler, realiza la menor cantidad de accesos a memoria posible y así haciendo, en nuestro caso, el algoritmo de ordenamiento más óptimo.

La cantidad de llamados a compare y swap entre ambos algoritmos es muy marcada, dando esto un claro indicio del porque de la diferencia en los tiempos de ejecución en cada caso.

3.3.5 Cálculo del Speedup

El speedup es una relación entre el tiempo mejorado y el tiempo anterior a la mejora de, en nuestro caso, una aplicación. Este valor nos dará una idea de cuánto más rápido realiza una rutina un programa respecto de otro (en nuestro caso, Bubblesort contra Quicksort).

La fórmula para calcularlo es la siguiente:

$$SpeedUp = \frac{T_{original}}{T_{mejorado}}$$

En nuestro caso:

$$SpeedUp = \frac{T_{bubblesort}}{T_{quicksort}}$$

Reemplazando con los datos antes mencionados:

SpeedUp entre ambos algoritmos		File size	Speed Up
		177	105,7510716
		225	87,76683515
		659	151,6577196
		2.199	113,5512049

Figure 6: Tabla de los speedups obtenidos de los sorts

Pasamos los datos antes calculados a un gráfico para observar mejor cómo evoluciona el Speedup.

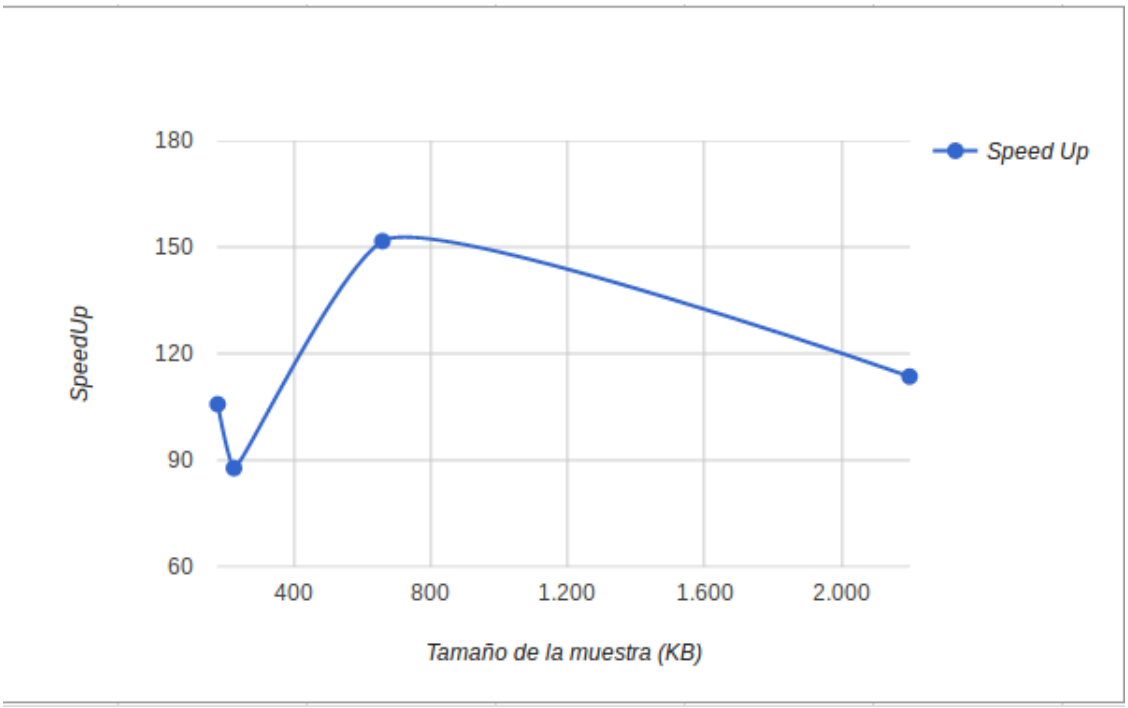

Figure 7: Gráfico del speedup entre Bubblsort en C y Quicksort en C

A medida que el tamaño del archivo aumenta, el Speedup del Quicksort respecto del Bubblesort también aumenta. Esto es debido al comportamiento de los algoritmos a la hora de ordenar los archivos. Como se vio en la sección anterior, cuanto más grande es la cantidad de

palabras del archivo, el Bubblesort tarda mucho más que el Quicksort. Por lo tanto, viendo la fórmula del Speedup, la relación de los tiempos cada vez va aumentando, haciendo que aumente el Speedup.

4 Conclusiones

En el presente trabajo se analizó el comportamiento del ordenamiento de archivos de distintos tamaños con los algoritmos Bubblesort y Quicksort. Se pudo observar que el ordenamiento del Quicksort es más rápido que el del Bubblesort. En las sucesivas mediciones se pudo comprobar que a medida que aumentaba el tamaño del archivo a ordenar el Bubblesort realizaba el ordenamiento considerablemente más lento que el otro algoritmo.

Los posteriores cálculos de los Speedups permitieron dar una idea más concisa sobre la mejora de un algoritmo respecto de otro.

5 Código

5.1 sorters.h

```
1  #ifndef SORTERS_H_INCLUDED
2  #define SORTERS_H_INCLUDED
3
4  void bubbleSort(char *arrayOfWords[], unsigned int arraySize);
5
6  void quickSort(char *arrayOfWords[], unsigned int arraySize);
7
8  int compare(char *word1, char *word2);
9
10 void swap(char *arrayOfWords[], unsigned int position1, unsigned int
       position2);
11
12 #endif // SORTERS_H_INCLUDED
```

5.2 sorters.c

```
1  #include "sorters.h"
2  #include <string.h>
3
4  #define true 1
5  #define false 0
6  typedef int bool;
7
8  void bubbleSort(char *arrayOfWords[], unsigned int length) {
9      unsigned int i, j;
10     for (i = 0; i < length; i++) {
11         bool swapped = false;
12         for (j = 0; j < length - i - 1; j++) {
13             if (compare(arrayOfWords[j], arrayOfWords[j + 1]) > 0) {
14                 swapped = true;
15                 swap(arrayOfWords, j, j + 1);
16             }
17         }
18         //Mejora: Deja de comparar si esta ordenado
19         if (!swapped) {
20             break;
21         }
22     }
23 }
24
25 int compare(char *word1, char *word2) {
26     return strcmp(word1, word2);
27 }
28
29 void swap(char *arrayOfWords[], unsigned int position1, unsigned int
       position2) {
30     char *auxPtr = arrayOfWords[position2];
31     arrayOfWords[position2] = arrayOfWords[position1];
32     arrayOfWords[position1] = auxPtr;
33 }
34
35 void quickSort(char *arrayOfWords[], unsigned int length) {
36     unsigned int i, pivot = 0;
```

```
37
38      if (length <= 1) {
39          return;
40      }
41
42      // swap a randomly selected value to the last node
43      //swap(arrayOfWords, arrayOfWords+((unsigned int)rand() % length),
            arrayOfWords+length-1);
44      // reset the pivot index to zero, then scan
45      for (i = 0; i < length - 1; ++i) {
46          if (compare(arrayOfWords[i], arrayOfWords[length - 1]) < 0) {
47              swap(arrayOfWords, i, pivot++);
48          }
49      }
50
51      // move the pivot value into its place
52      swap(arrayOfWords, pivot, length - 1);
53
54      // and invoke on the subsequences. does NOT include the pivot-slot
55      quickSort(arrayOfWords, pivot++);
56      quickSort(arrayOfWords + pivot, length - pivot);
57  }
```

5.3 main.c

```
1   #include "sorters.h"
2   #include <ctype.h>
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <unistd.h>
6   #include <getopt.h>
7   #include <string.h>
8   #include <sys/stat.h>
9
10  #define true 1
11  #define false 0
12  #define LONGEST_LINE 1024
13
14  /*#define WORD_SEPARATORS "\" .,;:  ?' !  -_  &*#\n\r\t\f\v()[]{}~%<>|"*/
15  #define WORD_SEPARATORS "\r\t\n \r\n" //Separadores: Tab, nueva linea, y
        espacio en blanco.
16  typedef int bool;
17  bool useBubbleSort = true;
18  bool haveQuickSort = false;
19  bool haveBubbleSort = false;
20
21  void freeArray(char *wordsArray[], unsigned int amountOfWords);
22
23  void printArray(char *arrayWords[], unsigned int amountOfWords, FILE *
        outputStream);
24
25  void closeFile(FILE *inputFile);
26
27  void printError(char *message);
28
29  int main(int argc, char *argv[]) {
30      int parameter;
```

```
31      char *input = NULL;
32      char *output = NULL;
33
34      static struct option long_options[] =
35      {
36      /* These options don't set a flag.
37       We distinguish them by their indices. */
38      {"version", no_argument,       0, 'V'},
39      {"help",    no_argument,       0, 'h'},
40      {"output",  required_argument, 0, 'o'},
41      {"input",   required_argument, 0, 'i'},
42      {"qsort",   no_argument,       0, 'q'},
43      {"bsort",   no_argument,       0, 'b'},
44      {0, 0,                         0, 0}
45      };
46
47      opterr = 0;
48      int option_index = 0;
49
50      while ((parameter = getopt_long(argc, argv, "Vhqb:o:i:",
51                                      long_options, &option_index)) != -1)
52          switch (parameter) {
53              case 'V':
54                  printf("This is version 1.0 from tpo: Basic infrastructure
                        .\n");
55                  exit(EXIT_FAILURE);
56              case 'h':
57                  printf("Usage:\n"
58                         " tp0 -h\n"
59                         " tp0 -V\n"
60                         " tp0 [options] file\n"
61                         "Options:\n"
62                         " -V, --version print version and quit.\n"
63                         " -h, --help Print this information.\n"
64                         " -o, --output Path to output file.\n"
65                         " -i, --intput Path to input file.\n"
66                         " -q, --qsort Use quicksort.\n"
67                         " -b, --bsort Use bubblesort.");
68                  exit(EXIT_FAILURE);
69              case 'o':
70                  output = optarg;
71                  break;
72              case 'i':
73                  input = optarg;
74                  break;
75              case 'q':
76                  useBubbleSort = false;
77                  haveQuickSort = true;
78                  break;
79              case 'b':
80                  useBubbleSort = true;
81                  haveBubbleSort = true;
82                  break;
83              case '?':
84                  if (optopt == 'o' || optopt == 'i') {
85                      fprintf(stderr, "Option -%c requires an argument.\n",
                            optopt);
86                  } else if (isprint (optopt)) {
87                      fprintf(stderr, "Unknown argument '-%c'.\n", optopt);
```

```
88                  } else {
89                      fprintf(stderr, "Unknown option character '\\x%x'.\n",
                            optopt);
90
91                  }
92                  exit(EXIT_FAILURE);
93              default:
94                  exit(EXIT_FAILURE);
95          }
96
97      if (haveBubbleSort && haveQuickSort) {
98          printError("fatal: cannot use quicksort and bubblesort at the same
                time.");
99          exit(EXIT_FAILURE);
100     }
101
102     FILE *inputStream = NULL;
103     FILE *outputStream = NULL;
104
105     /* input value */
106     if (input != NULL) {
107         inputStream = fopen(input, "r");
108         if (inputStream != NULL) {
109             struct stat st;
110             if (stat(input, &st) != 0) {
111                 printError("Invalid input file.\n");
112                 return EXIT_FAILURE;
113             } else if (st.st_size == 0) {
114                 printError("Empty input file.\n");
115                 exit(EXIT_FAILURE);
116             }
117         } else {
118             printError("fatal: cannot open input file.");
119             exit(EXIT_FAILURE);
120         }
121     }
122
123 /* ouptut value */
124     if (output != NULL) {
125         outputStream = fopen(output, "w");
126         if (outputStream == NULL) {
127             printError("fatal: cannot open output file.");
128             closeFile(inputStream);
129             exit(EXIT_FAILURE);
130         }
131     }
132
133     char **wordsArray = NULL;
134
135
136     unsigned int amountOfWords = 0L;
137     char *token;
138     char line[LONGEST_LINE];
139     while (fgets(line, sizeof(line), inputStream) != NULL) {
140
141         token = strtok(line, WORD_SEPARATORS);
142 /* walk through other tokens */
143         while (token != NULL) {
144             size_t wordLength = strlen(token);
```

```
145             char *word = malloc(sizeof(char) * (wordLength + 1));
146
147             if (word == NULL) {
148                 printError("fatal: Malloc failed");
149                 fclose(inputStream);
150                 fclose(outputStream);
151                 freeArray(wordsArray, amountOfWords);
152                 exit(EXIT_FAILURE);
153             }
154
155             strcpy(word, token
156             );
157             wordsArray = realloc(wordsArray, (amountOfWords + 1) * sizeof(
                    char *));
158
159             if (wordsArray == NULL) {
160                 printError("fatal: Malloc failed");
161                 fclose(inputStream);
162                 fclose(outputStream);
163                 freeArray(wordsArray, amountOfWords);
164                 exit(EXIT_FAILURE);
165             }
166
167             wordsArray[amountOfWords] = word;
168             //printf("Palabra: %s\n", wordsArray[amountOfWords]);
169             amountOfWords++;
170             token = strtok(NULL, WORD_SEPARATORS);
171         }
172     }
173
174     //printf("Amount of words: %u\n", amountOfWords);
175     closeFile(inputStream);
176
177     if (useBubbleSort) {
178         bubbleSort(wordsArray, amountOfWords
179         );
180     } else {
181         quickSort(wordsArray, amountOfWords
182         );
183     }
184
185     printArray(wordsArray, amountOfWords, outputStream);
186     freeArray(wordsArray, amountOfWords);
187     closeFile(outputStream);
188     return EXIT_SUCCESS;
189 }
190
191 void freeArray(char *wordsArray[], unsigned int amountOfWords) {
192
193     if (amountOfWords == 0) {
194         return;
195     }
196
197     unsigned int i;
198     for (i = 0; i <= amountOfWords - 1; i++) {
199         free(wordsArray[i]);
200     }
201
202     free(wordsArray);
```

```
203 }
204
205 void printArray(char *wordsArray[], unsigned int amountOfWords, FILE *
        outputStream) {
206
207     if (amountOfWords == 0) {
208         return;
209     }
210
211     FILE *out = outputStream;
212
213     if (out == NULL) {
214         out = stdout;
215     }
216
217     unsigned int i;
218     for (i = 0; i <= amountOfWords - 1; i++) {
219         fprintf(out, "%s\n", wordsArray[i]);
220         //Check if write was successful
221         if (ferror(out)) {
222             freeArray(wordsArray, amountOfWords);
223             closeFile(out);
224             exit(EXIT_FAILURE);
225         }
226     }
227 }
228
229 void closeFile(FILE *file) {
230     if (file != NULL) {
231         fclose(file);
232     }
233 }
234
235
236 void printError(char *message) {
237     fputs(message, stderr);
238 }
```