UNIVERSIDAD DE GRANADA

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

---

ALGORÍTMICA
Práctica 1 : Eficiencia de algoritmos

Autores :
Elvira Castillo Fernández
David Gil Bautista
Jose Luis Izquierdo Mañas
Freddy A. Jaramillo López
Alejandro Jerónimo Fuentes
Gregorio Vidoy Fajardo

Fecha : 15 de marzo de 2017
Grupo de prácticas : C1
Subgrupo : 4

Índice

1. Descripción del problema

El objetivo de la práctica es comprender la importancia del análisis de la eficiencia de los algoritmos y familiarizarse con las formas de llevarlo a cabo. Para ello se realizará un estudio teórico, empírico e híbrido.

En esta primera parte se cumplirán los siguientes objetivos:

1. Calcular la eficiencia empírica de 8 algoritmos definiendo adecuadamente los tamaños de entrada de forma tal que se generen al menos 25 datos. Se incluirá en la memoria tablas diferentes para los algoritmos de distinto orden de eficiencia (una con los algoritmos de orden $O(n^2)$, otra con los $O(n \log n)$, otra con $O(n^3)$ y otra con $O(2^n)$).

2. Con cada una de las tablas anteriores, se generará un gráfico comparando los tiempos de los algoritmos. Para los algoritmos que realizan la misma tarea (los de ordenación), se incluirá también una gráfica con todos ellos,para poder apreciar las diferencias en rendimiento de algoritmos con diferente orden de eficiencia.

2. Eficiencia empírica

Para medir la eficiencia empírica hemos usado la biblioteca `<ctime>` siguiendo el siguiente esquema:

```
#include <ctime>

clock_t tantes;
clocl_t tdespues;

tantes = clock();       //Valor del reloj antes de la ejecucion del algoritmo
algoritmo();           //Algoritmo a medir
tdespues = clock();    //Valor del reloj después de la ejecucion del algoritmo

cout << (double)(tdespues - tantes) / CLOCKS PER SEC << endl; //Diferencia entre los dos instantes
```

2.1. Algoritmos de ordenación

En primer lugar mostraremos los resultados obtenidos experimentalmente de los algoritmos de ordenación clásicos, mostrando los resultados de ejecución en forma de tabla.

2.1.1. Eficiencia $O(n^2)$

Burbuja

A continuación se muestra el código fuente correspondiente al algoritmo de la burbuja que se ha usado para medir los tiempos de ejecución:

```
1  /**
2     @file Ordenacin por burbuja
3  */
4
5
6  #include <iostream>
7  using namespace std;
8  #include <ctime>
9  #include <cstdlib>
10 #include <climits>
11 #include <cassert>
12
13
14
15
16
17
18 /* ↩
       ************************************************************↩
        */
19 /*  Mtodo de ordenacin por burbuja  */
20
21 /**
22    @brief Ordena un vector por el mtodo de la burbuja.
23
24    @param T: vector de elementos. Debe tener num_elem ↩
          elementos.
25               Es MODIFICADO.
26    @param num_elem: nmero de elementos. num_elem > 0.
27
28    Cambia el orden de los elementos de T de forma que los ↩
          dispone
29    en sentido creciente de menor a mayor.
30    Aplica el algoritmo de la burbuja.
31 */
32 inline static
33 void burbuja(int T[], int num_elem);
34
35
36
37 /**
38    @brief Ordena parte de un vector por el mtodo de la ↩
```

```
        burbuja.
39
40      @param T: vector de elementos. Tiene un nmero de elementos
41                      mayor o igual a final.Es MODIFICADO.
42
43      @param inicial: Posicin que marca el incio de la parte del
44                      vector a ordenar.
45      @param final: Posicin detrs de la ltima de la parte del
46                      vector a ordenar.
47          inicial < final.
48
49      Cambia el orden de los elementos de T entre las posiciones
50      inicial y final - 1de forma que los dispone en sentido creciente
51      de menor a mayor.
52      Aplica el algoritmo de la burbuja.
53  */
54  static void burbuja_lims(int T[], int inicial, int final);
55
56
57
58  /**
59      Implementacin de las funciones
60  **/
61
62  inline void burbuja(int T[], int num_elem)
63  {
64    burbuja_lims(T, 0, num_elem);
65  };
66
67
68  static void burbuja_lims(int T[], int inicial, int final)
69  {
70    int i, j;
71    int aux;
72    for (i = inicial; i < final - 1; i++)
73      for (j = final - 1; j > i; j--)
74        if (T[j] < T[j-1])
75    {
76      aux = T[j];
77      T[j] = T[j-1];
78      T[j-1] = aux;
79    }
80  }
81
82  void sintaxis()
```

```
83 {
84   cerr << "Sintaxis:" << endl;
85   cerr << "  TAM: Tamao del vector (>0)" << endl;
86   cerr << "Se genera un vector de tamao TAM con elementos aleatorios en [0,VMAX[" << endl;
87   exit(EXIT_FAILURE);
88 }
89
90
91 int main(int argc, char **argv)
92 {
93
94   if (argc!=2)
95     sintaxis();
96   int tam=atoi(argv[1]);      // Tamao del vector
97
98   clock_t tantes;    // Valor del reloj antes de la ejecucin
99   clock_t tdespues;  // Valor del reloj despus de la ejecucin
100
101  int * T = new int[tam];
102  assert(T);
103
104  srandom(time(0));
105  //Rellenamos el vector con numeros aleatorios
106  for (int i = 0; i < tam; i++)
107    {
108      T[i] = random();
109    };
110
111  // Empieza el algoritmo Burbuja
112  tantes = clock();// Anotamos el tiempo de inicio
113  burbuja(T, tam);
114  tdespues = clock();// Anotamos el tiempo de finalizacin
115
116  // Mostramos resultados
117  cout << tam << "\t" << (double) (tdespues-tantes) / CLOCKS_PER_SEC << endl;
118  delete [] T;
119
120 };
```

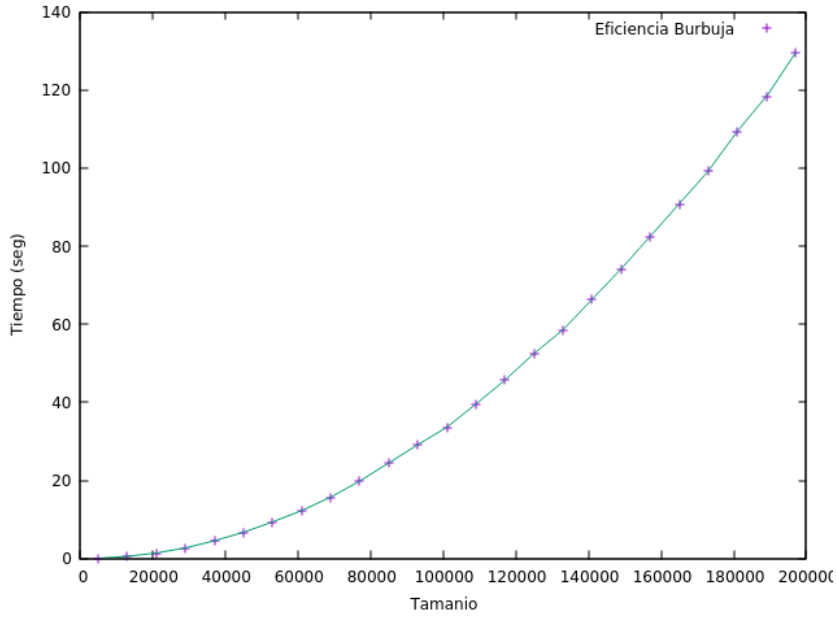

Figura 1: Gráfica algoritmo burbuja

A la hora de ejecutar el algoritmo en un computador obtenemos la siguiente gráfica creada con el software *gnuplot*. Los valores de esta gráfica se pueden observar en la tabla (1)

Podemos observar en la gráfica que efectivamente el algoritmo de la burbuja pertenece a $O(n^2)$ dada la forma que tiene la función n^2.

Inserción

Este es el código fuente correspondiente al algoritmo de ordenación por inserción:

```
1  /**
2     @file Ordenación por inserción
3  */
4
5
6  #include <iostream>
7  using namespace std;
8  #include <ctime>
9  #include <cstdlib>
10 #include <climits>
11 #include <cassert>
12
13
```

```
14
15
16
17
18 /* ↩
     ************************************************************↩
      */
19 /*  Método de ordenación por inserción  */
20
21 /**
22    @brief Ordena un vector por el método de inserción.
23
24    @param T: vector de elementos. Debe tener num_elem ↩
         elementos.
25              Es MODIFICADO.
26    @param num_elem: número de elementos. num_elem > 0.
27
28    Cambia el orden de los elementos de T de forma que los ↩
         dispone
29    en sentido creciente de menor a mayor.
30    Aplica el algoritmo de inserción.
31 */
32 inline static
33 void insercion(int T[], int num_elem);
34
35
36
37 /**
38    @brief Ordena parte de un vector por el método de ↩
         inserción.
39
40    @param T: vector de elementos. Tiene un número de ↩
         elementos
41                    mayor o igual a final. Es MODIFICADO.
42    @param inicial: Posición que marca el incio de la parte ↩
         del
43                    vector a ordenar.
44    @param final: Posición detrás de la última de la parte ↩
         del
45                    vector a ordenar.
46        inicial < final.
47
48    Cambia el orden de los elementos de T entre las ↩
         posiciones
49    inicial y final - 1de forma que los dispone en sentido ↩
         creciente
50    de menor a mayor.
51    Aplica el algoritmo de inserción.
52 */
```

```
53 static void insercion_lims(int T[], int inicial, int final);
54
55
56
57 /**
58    Implementación de las funciones
59 **/
60
61 inline static void insercion(int T[], int num_elem)
62 {
63   insercion_lims(T, 0, num_elem);
64 }
65
66
67 static void insercion_lims(int T[], int inicial, int final)
68 {
69   int i, j;
70   int aux;
71   for (i = inicial + 1; i < final; i++) {
72     j = i;
73     while ((T[j] < T[j-1]) && (j > 0)) {
74       aux = T[j];
75       T[j] = T[j-1];
76       T[j-1] = aux;
77       j--;
78     };
79   };
80 }
81
82
83
84 int main(int argc, char * argv[])
85 {
86
87   if(argc != 2){
88     cout << "Numero de argumentos incorrecto";
89     return 1;
90   }
91
92   int n = atoi(argv[1]);
93
94   int * T = new int[n];
95   assert(T);
96
97   //Declaramos las variables para medir el tiempo
98   clock_t tantes,tdespues;
99
100
101
```

```
102   srandom(time(0));
103
104   for (int i = 0; i < n; i++)
105     {
106       T[i] = random();
107     };
108
109   tantes = clock(); //Medimos el instante antes de la ↩
          ejecución
110   insercion(T, n);
111   tdespues = clock(); // Medimos el instante después de la ↩
          ejecución
112
113
114   cout << (double) (tdespues - tantes) / CLOCKS_PER_SEC << ↩
          endl; //Calculamos la diferencia entre los dos ↩
          instantes y lo pasamos  a segundos
115
116
117   delete [] T;
118
119   return 0;
120 };
```

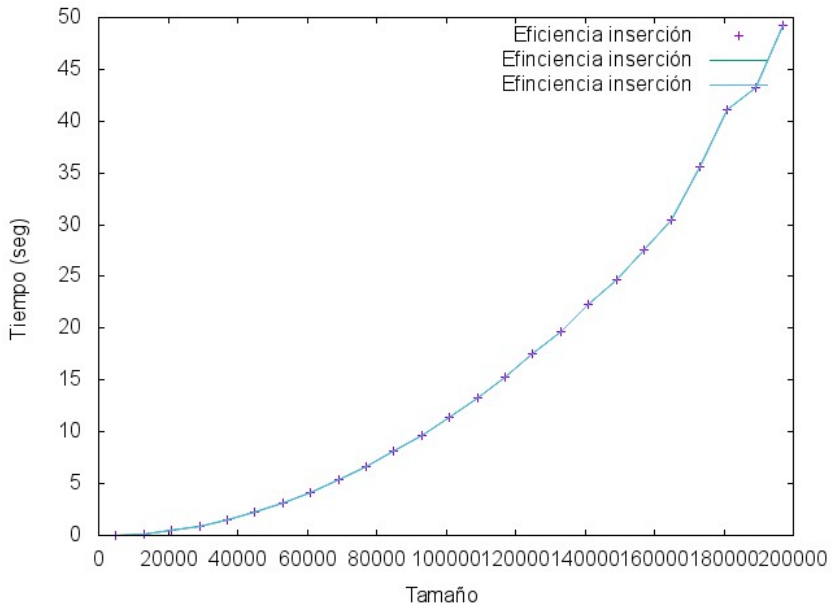

Figura 2: Gráfica algoritmo inserción

También podemos ver en la gráfica que pertenece a $O(n^2)$.

Selección

El código del algoritmo es:

```
1  /**
2     @file Ordenacin por seleccin
3  */
4
5
6  #include <iostream>
7  using namespace std;
8  #include <ctime>
9  #include <cstdlib>
10 #include <climits>
11 #include <cassert>
12
13
14
15
16
17
18 /* ↩
       ************************************************************↩
        */
19 /*  Mtodo de ordenacin por seleccin  */
20
21 /**
22    @brief Ordena un vector por el mtodo de seleccin.
23
24    @param T: vector de elementos. Debe tener num_elem ↩
          elementos.
25           Es MODIFICADO.
26    @param num_elem: nmero de elementos. num_elem > 0.
27
28    Cambia el orden de los elementos de T de forma que los ↩
          dispone
29    en sentido creciente de menor a mayor.
30    Aplica el algoritmo de seleccin.
31 */
32 inline static
33 void seleccion(int T[], int num_elem);
34
35
36
37 /**
38    @brief Ordena parte de un vector por el mtodo de seleccin↩
          .
39
40    @param T: vector de elementos. Tiene un nmero de ↩
```

```
        elementos
41                      mayor o igual a final. Es MODIFICADO.
42      @param inicial: Posicin que marca el incio de la parte ↩
           del
43                      vector a ordenar.
44      @param final: Posicin detrs de la ltima de la parte del
45                      vector a ordenar.
46          inicial < final.
47
48      Cambia el orden de los elementos de T entre las ↩
           posiciones
49      inicial y final - 1de forma que los dispone en sentido ↩
           creciente
50      de menor a mayor.
51      Aplica el algoritmo de seleccin.
52  */
53  static void seleccion_lims(int T[], int inicial, int final);
54
55
56
57  /**
58      Implementacin de las funciones
59  **/
60
61  void seleccion(int T[], int num_elem)
62  {
63    seleccion_lims(T, 0, num_elem);
64  }
65
66  static void seleccion_lims(int T[], int inicial, int final)
67  {
68    int i, j, indice_menor;
69    int menor, aux;
70    for (i = inicial; i < final - 1; i++) {
71      indice_menor = i;
72      menor = T[i];
73      for (j = i; j < final; j++)
74        if (T[j] < menor) {
75    indice_menor = j;
76    menor = T[j];
77        }
78      aux = T[i];
79      T[i] = T[indice_menor];
80      T[indice_menor] = aux;
81    };
82  }
83
84
85
```

```
86 int main(int argc, char * argv[])
87 {
88
89   if (argc != 2)
90     {
91       cerr << "Formato " << argv[0] << " <num_elem>" << endl↩
             ;
92       return -1;
93     }
94
95   int n = atoi(argv[1]);
96
97   int * T = new int[n];
98   assert(T);
99
100  srandom(time(0));
101
102  for (int i = 0; i < n; i++)
103    {
104      T[i] = random();
105    };
106
107
108  const int TAM_GRANDE = 2000;
109  const int NUM_VECES = 100;
110
111  if (n > TAM_GRANDE)
112    {
113      clock_t t_antes = clock();
114
115      seleccion(T, n);
116
117      clock_t t_despues = clock();
118
119      cout << n << "  " << ((double)(t_despues - t_antes)) /↩
             CLOCKS_PER_SEC
120     << endl;
121    } else {
122      int * U = new int[n];
123      assert(U);
124
125      for (int i = 0; i < n; i++)
126  U[i] = T[i];
127
128      clock_t t_antes_vacio = clock();
129      for (int veces = 0; veces < NUM_VECES; veces++)
130  {
131    for (int i = 0; i < n; i++)
132      U[i] = T[i];
```

```
133   }
134       clock_t t_despues_vacio = clock();
135
136       clock_t t_antes = clock();
137       for (int veces = 0; veces < NUM_VECES; veces++)
138   {
139     for (int i = 0; i < n; i++)
140       U[i] = T[i];
141     seleccion(U, n);
142   }
143       clock_t t_despues = clock();
144       cout << n << " \t  "
145      << ((double) ((t_despues - t_antes) -
146        (t_despues_vacio - t_antes_vacio))) /
147   (CLOCKS_PER_SEC * NUM_VECES)
148      << endl;
149
150       delete [] U;
151     }
152
153   delete [] T;
154
155   return 0;
156 };
```

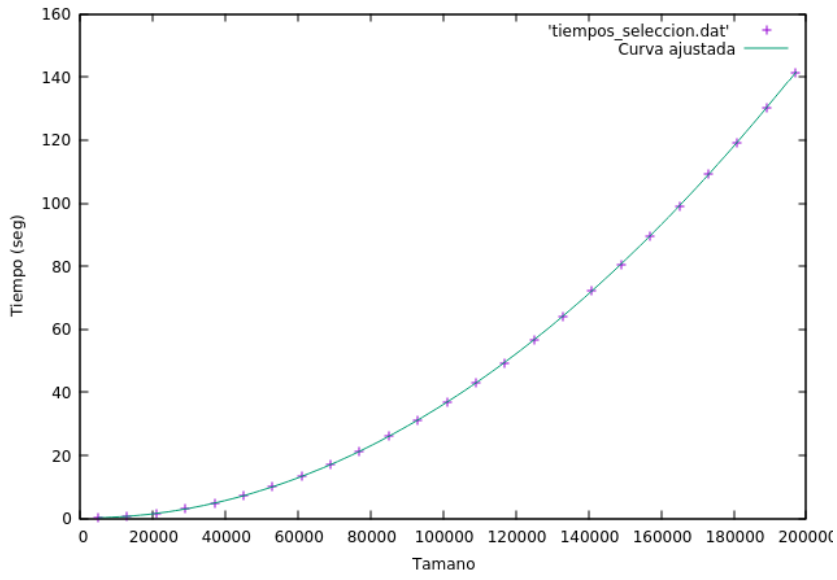

Figura 3: Gráfica algoritmo selección

Para obtener los tiempos de ejecución que se ven en la tabla se ha usado

el siguiente script.

```
#!/bin/csh
@ inicio = 5000
@ fin = 200000
@ incremento = 8000
set ejecutable = burbuja
set salida = tiempos_busqueda_burbuja.dat

@ i = $inicio
echo > $salida
while ( $i <= $fin )
  echo Ejecución tam = $i
  echo `./{$ejecutable} $i` >> $salida
  @ i += $incremento
end
```

Como se puede ver el número de componentes del vector empieza en 5000 hasta 200000 con un incremento de 8000.

La suma de todas las gráficas de los algoritmos ha sido:

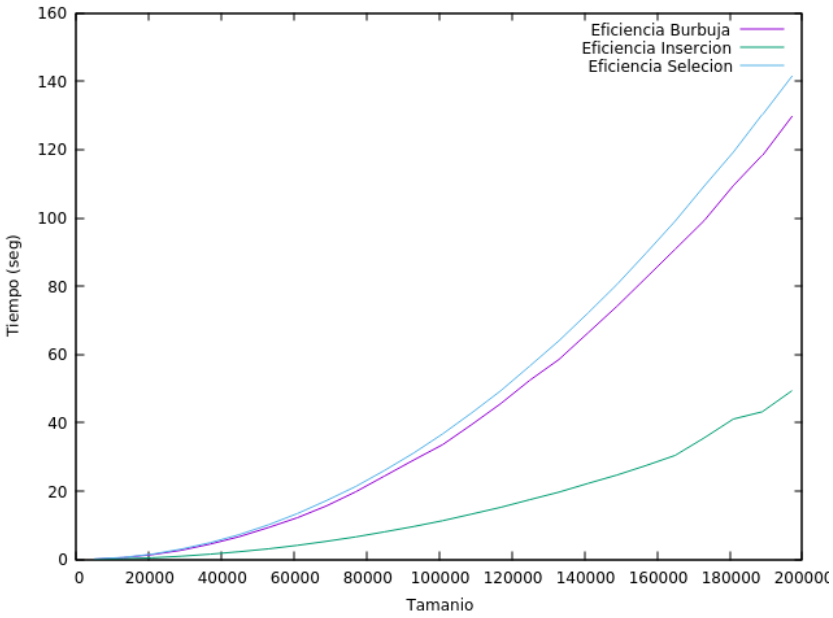

Figura 4: Gráfica algoritmos $O(n^2)$

Podemos concluir que pese a tener la misma eficiencia en el peor de los casos algunos algoritmos tienen tiempos de ejecución más bajos que otros ya que se ejecutan en distinto hardware.

Tamaño del vector	Burbuja	Inserción	Selección
5000	0.0625	0.046875	0.149065
13000	0.515625	0.1875	0.603133
21000	1.40625	0.5	1.57351
29000	2.6875	0.9375	3.03391
37000	4.51562	1.54688	4.92873
45000	6.70312	2.26562	7.29809
53000	9.34375	3.125	10.141
61000	12.2188	4.14062	13.4508
69000	15.6875	5.32812	17.2353
77000	19.8125	6.625	21.346
85000	24.4219	8.10938	26.0582
93000	29.1094	9.64062	31.1696
101000	33.625	11.3438	36.781
109000	39.5312	13.3125	42.9186
117000	45.6406	15.2656	49.4047
125000	52.4531	17.5156	56.6685
133000	58.5625	19.7188	64.0087
141000	66.3438	22.2656	72.1494
149000	74.1562	24.7188	80.5634
157000	82.4375	27.5156	89.669
165000	90.8125	30.4688	99.0533
173000	99.2188	35.5938	109.298
181000	109.422	41.1094	119.24
189000	118.344	43.25	130.293
197000	129.625	49.2656	141.419

Tabla 1: Tiempos de ejecución de la familia $O(n^2)$

2.1.2. Eficiencia $O(n \log n)$

Mergesort

El código del algoritmo es:

```
1 /**
2    @file Ordenacin por mezcla
3 */
4
5
6 #include <iostream>
7 using namespace std;
```

```
8  #include <ctime>
9  #include <cstdlib>
10 #include <climits>
11 #include <cassert>
12
13
14
15
16
17
18 /* ↩
       ************************************************************↩
        */
19 /*  Mtodo de ordenacin por mezcla  */
20
21 /**
22    @brief Ordena un vector por el mtodo de mezcla.
23
24    @param T: vector de elementos. Debe tener num_elem ↩
         elementos.
25              Es MODIFICADO.
26    @param num_elem: nmero de elementos. num_elem > 0.
27
28    Cambia el orden de los elementos de T de forma que los ↩
         dispone
29    en sentido creciente de menor a mayor.
30    Aplica el algoritmo de mezcla.
31 */
32 inline static
33 void mergesort(int T[], int num_elem);
34
35
36
37 /**
38    @brief Ordena parte de un vector por el mtodo de mezcla.
39
40    @param T: vector de elementos. Tiene un nmero de ↩
         elementos
41                  mayor o igual a final. Es MODIFICADO.
42    @param inicial: Posicin que marca el incio de la parte ↩
         del
43                  vector a ordenar.
44    @param final: Posicin detrs de la ltima de la parte del
45                  vector a ordenar.
46        inicial < final.
47
48    Cambia el orden de los elementos de T entre las ↩
         posiciones
49    inicial y final - 1 de forma que los dispone en sentido ↩
```

```
        creciente
50      de menor a mayor.
51      Aplica el algoritmo de la mezcla.
52  */
53  static void mergesort_lims(int T[], int inicial, int final);
54
55
56  /**
57      @brief Ordena un vector por el mtodo de insercin.
58
59      @param T: vector de elementos. Debe tener num_elem ↩
            elementos.
60                  Es MODIFICADO.
61      @param num_elem: nmero de elementos. num_elem > 0.
62
63      Cambia el orden de los elementos de T de forma que los ↩
            dispone
64      en sentido creciente de menor a mayor.
65      Aplica el algoritmo de insercin.
66  */
67  inline static
68  void insercion(int T[], int num_elem);
69
70
71  /**
72      @brief Ordena parte de un vector por el mtodo de insercin↩
            .
73
74      @param T: vector de elementos. Tiene un nmero de ↩
            elementos
75                      mayor o igual a final. Es MODIFICADO.
76      @param inicial: Posicin que marca el incio de la parte ↩
            del
77                      vector a ordenar.
78      @param final: Posicin detrs de la ltima de la parte del
79                      vector a ordenar.
80          inicial < final.
81
82      Cambia el orden de los elementos de T entre las ↩
            posiciones
83      inicial y final - 1 de forma que los dispone en sentido ↩
            creciente
84      de menor a mayor.
85      Aplica el algoritmo de la insercin.
86  */
87  static void insercion_lims(int T[], int inicial, int final);
88
89
90  /**
```

```
91    @brief Mezcla dos vectores ordenados sobre otro.
92
93    @param T: vector de elementos. Tiene un nmero de elementos
94                    mayor o igual a final. Es MODIFICADO.
95    @param inicial: Posicin que marca el incio de la parte del
96                    vector a escribir.
97    @param final: Posicin detrs de la ltima de la parte del
98                    vector a escribir
99        inicial < final.
100   @param U: Vector con los elementos ordenados.
101   @param V: Vector con los elementos ordenados.
102             El nmero de elementos de U y V sumados debe coincidir
103             con final - inicial.
104
105   En los elementos de T entre las posiciones inicial y final - 1
106   pone ordenados en sentido creciente, de menor a mayor, los
107   elementos de los vectores U y V.
108 */
109 static void fusion(int T[], int inicial, int final, int U[], int V[]);
110
111
112
113 /**
114    Implementacin de las funciones
115 **/
116
117
118 inline static void insercion(int T[], int num_elem)
119 {
120   insercion_lims(T, 0, num_elem);
121 }
122
123
124 static void insercion_lims(int T[], int inicial, int final)
125 {
126   int i, j;
127   int aux;
128   for (i = inicial + 1; i < final; i++) {
129     j = i;
130     while ((T[j] < T[j-1]) && (j > 0)) {
131       aux = T[j];
132       T[j] = T[j-1];
133       T[j-1] = aux;
```

```
134       j--;
135     };
136   };
137 }
138 
139 
140 const int UMBRAL_MS = 100;
141 
142 void mergesort(int T[], int num_elem)
143 {
144   mergesort_lims(T, 0, num_elem);
145 }
146 
147 static void mergesort_lims(int T[], int inicial, int final)
148 {
149   if (final - inicial < UMBRAL_MS)
150     {
151       insercion_lims(T, inicial, final);
152     } else {
153       int k = (final - inicial)/2;
154 
155       int * U = new int [k - inicial + 1];
156       assert(U);
157       int l, l2;
158       for (l = 0, l2 = inicial; l < k; l++, l2++)
159   U[l] = T[l2];
160       U[l] = INT_MAX;
161 
162       int * V = new int [final - k + 1];
163       assert(V);
164       for (l = 0, l2 = k; l < final - k; l++, l2++)
165   V[l] = T[l2];
166       V[l] = INT_MAX;
167 
168       mergesort_lims(U, 0, k);
169       mergesort_lims(V, 0, final - k);
170       fusion(T, inicial, final, U, V);
171       delete [] U;
172       delete [] V;
173     };
174 }
175 
176 
177 static void fusion(int T[], int inicial, int final, int U[],↩
      int V[])
178 {
179   int j = 0;
180   int k = 0;
181   for (int i = inicial; i < final; i++)
```

```
182     {
183       if (U[j] < V[k]) {
184   T[i] = U[j];
185   j++;
186       } else{
187   T[i] = V[k];
188   k++;
189       };
190     };
191 }
192
193
194
195
196
197 int main(int argc, char * argv[])
198 {
199
200   if (argc != 2)
201     {
202       cerr << "Formato " << argv[0] << " <num_elem>" << endl↩
              ;
203       return -1;
204     }
205
206   int n = atoi(argv[1]);
207
208   int * T = new int[n];
209   assert(T);
210
211   srandom(time(0));
212
213   for (int i = 0; i < n; i++)
214     {
215       T[i] = random();
216     };
217
218   const int TAM_GRANDE = 10000;
219   const int NUM_VECES = 1000;
220
221   if (n > TAM_GRANDE)
222     {
223       clock_t t_antes = clock();
224
225       mergesort(T, n);
226
227       clock_t t_despues = clock();
228
229       cout << n << "  " << ((double)(t_despues - t_antes)) /↩
```

```
            CLOCKS_PER_SEC
230      << endl;
231    } else {
232      int * U = new int[n];
233      assert(U);
234
235      for (int i = 0; i < n; i++)
236  U[i] = T[i];
237
238      clock_t t_antes_vacio = clock();
239      for (int veces = 0; veces < NUM_VECES; veces++)
240  {
241    for (int i = 0; i < n; i++)
242      U[i] = T[i];
243  }
244      clock_t t_despues_vacio = clock();
245
246      clock_t t_antes = clock();
247      for (int veces = 0; veces < NUM_VECES; veces++)
248  {
249    for (int i = 0; i < n; i++)
250      U[i] = T[i];
251    mergesort(U, n);
252  }
253      clock_t t_despues = clock();
254      cout << n << " \t  "
255     << ((double) ((t_despues - t_antes) -
256       (t_despues_vacio - t_antes_vacio))) /
257  (CLOCKS_PER_SEC * NUM_VECES)
258     << endl;
259
260      delete [] U;
261    }
262
263
264  delete [] T;
265
266  return 0;
267 };
```

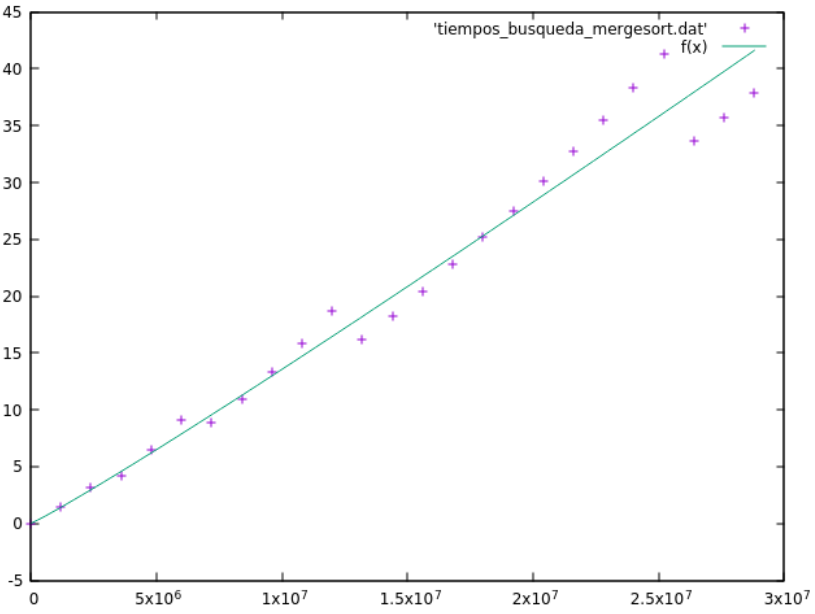

Figura 5: Gráfica algoritmo mergesort

El Mergesort tiene una gráfica con "dientes de sierra" porque a partir de un cierto umbral (para tamaños iguales 100), no profundiza más en la recursividad y lanza un algoritmo cuadrático (inserción) para ordenar .

Quicksort

El código es:

```
1  /**
2     @file Ordenacin rpida (quicksort).
3  */
4
5
6  #include <iostream>
7  using namespace std;
8  #include <ctime>
9  #include <cstdlib>
10 #include <climits>
11 #include <cassert>
12
13
14
15
16
17
18 /* ↩
      ************************************************************↩
       */
```

```
19 /*  Mtodo de ordenacin rpida  */
20
21 /**
22    @brief Ordena un vector por el mtodo quicksort.
23
24    @param T: vector de elementos. Debe tener num_elem ↩
      elementos.
25              Es MODIFICADO.
26    @param num_elem: nmero de elementos. num_elem > 0.
27
28    Cambia el orden de los elementos de T de forma que los ↩
      dispone
29    en sentido creciente de menor a mayor.
30    Aplica el algoritmo quicksort.
31 */
32 inline static
33 void quicksort(int T[], int num_elem);
34
35
36
37 /**
38    @brief Ordena parte de un vector por el mtodo quicksort.
39
40    @param T: vector de elementos. Tiene un nmero de ↩
      elementos
41                    mayor o igual a final. Es MODIFICADO.
42    @param inicial: Posicin que marca el incio de la parte ↩
      del
43                    vector a ordenar.
44    @param final: Posicin detrs de la ltima de la parte del
45                    vector a ordenar.
46        inicial < final.
47
48    Cambia el orden de los elementos de T entre las ↩
      posiciones
49    inicial y final - 1 de forma que los dispone en sentido ↩
      creciente
50    de menor a mayor.
51    Aplica el algoritmo quicksort.
52 */
53 static void quicksort_lims(int T[], int inicial, int final);
54
55
56 /**
57    @brief Ordena un vector por el mtodo de insercin.
58
59    @param T: vector de elementos. Debe tener num_elem ↩
      elementos.
60              Es MODIFICADO.
```

```
61     @param num_elem: nmero de elementos. num_elem > 0.
62
63     Cambia el orden de los elementos de T de forma que los ↩
          dispone
64     en sentido creciente de menor a mayor.
65     Aplica el algoritmo de insercin.
66  */
67  inline static
68  void insercion(int T[], int num_elem);
69
70
71  /**
72     @brief Ordena parte de un vector por el mtodo de insercin↩
          .
73
74     @param T: vector de elementos. Tiene un nmero de ↩
          elementos
75                    mayor o igual a final. Es MODIFICADO.
76     @param inicial: Posicin que marca el incio de la parte ↩
          del
77                    vector a ordenar.
78     @param final: Posicin detrs de la ltima de la parte del
79                    vector a ordenar.
80         inicial < final.
81
82     Cambia el orden de los elementos de T entre las ↩
          posiciones
83     inicial y final - 1 de forma que los dispone en sentido ↩
          creciente
84     de menor a mayor.
85     Aplica el algoritmo de insercin.
86  */
87  static void insercion_lims(int T[], int inicial, int final);
88
89
90  /**
91     @brief Redistribuye los elementos de un vector segn un ↩
          pivote.
92
93     @param T: vector de elementos. Tiene un nmero de ↩
          elementos
94                    mayor o igual a final. Es MODIFICADO.
95     @param inicial: Posicin que marca el incio de la parte ↩
          del
96                    vector a ordenar.
97     @param final: Posicin detrs de la ltima de la parte del
98                    vector a ordenar.
99         inicial < final.
100    @param pp: Posicin del pivote. Es MODIFICADO.
```

```
101
102     Selecciona un pivote los elementos de T situados en las ↩
           posiciones
103     entre inicial y final - 1. Redistribuye los elementos, ↩
           situando los
104     menores que el pivote a su izquierda, despus los iguales ↩
           y a la
105     derecha los mayores. La posicin del pivote se devuelve en↩
            pp.
106 */
107 static void dividir_qs(int T[], int inicial, int final, int ↩
       & pp);
108
109
110
111 /**
112     Implementacin de las funciones
113 **/
114
115
116 inline static void insercion(int T[], int num_elem)
117 {
118   insercion_lims(T, 0, num_elem);
119 }
120
121
122 static void insercion_lims(int T[], int inicial, int final)
123 {
124   int i, j;
125   int aux;
126   for (i = inicial + 1; i < final; i++) {
127     j = i;
128     while ((T[j] < T[j-1]) && (j > 0)) {
129       aux = T[j];
130       T[j] = T[j-1];
131       T[j-1] = aux;
132       j--;
133     };
134   };
135 }
136
137
138 const int UMBRAL_QS = 50;
139
140
141 inline void quicksort(int T[], int num_elem)
142 {
143   quicksort_lims(T, 0, num_elem);
144 }
```

```
145
146 static void quicksort_lims(int T[], int inicial, int final)
147 {
148   int k;
149   if (final - inicial < UMBRAL_QS) {
150     insercion_lims(T, inicial, final);
151   } else {
152     dividir_qs(T, inicial, final, k);
153     quicksort_lims(T, inicial, k);
154     quicksort_lims(T, k + 1, final);
155   };
156 }
157
158
159 static void dividir_qs(int T[], int inicial, int final, int ↩
     & pp)
160 {
161   int pivote, aux;
162   int k, l;
163
164   pivote = T[inicial];
165   k = inicial;
166   l = final;
167   do {
168     k++;
169   } while ((T[k] <= pivote) && (k < final-1));
170   do {
171     l--;
172   } while (T[l] > pivote);
173   while (k < l) {
174     aux = T[k];
175     T[k] = T[l];
176     T[l] = aux;
177     do k++; while (T[k] <= pivote);
178     do l--; while (T[l] > pivote);
179   };
180   aux = T[inicial];
181   T[inicial] = T[l];
182   T[l] = aux;
183   pp = l;
184 };
185
186
187
188
189 int main(int argc, char * argv[])
190 {
191   int n=atoi(argv[1]);;
192
```

```
193
194   clock_t tantes;
195   clock_t tdespues;
196
197   int * T = new int[n];
198   assert(T);
199
200   srandom(time(0));
201
202   for (int i = 0; i < n; i++)
203     {
204       T[i] = random();
205     };
206
207   tantes = clock();
208   quicksort(T, n);
209   tdespues = clock();
210
211
212   cout << n << "\t"<<(tdespues-tantes)/(double)↩
        CLOCKS_PER_SEC <<endl;
213
214   delete [] T;
215
216   return 0;
217 };
```

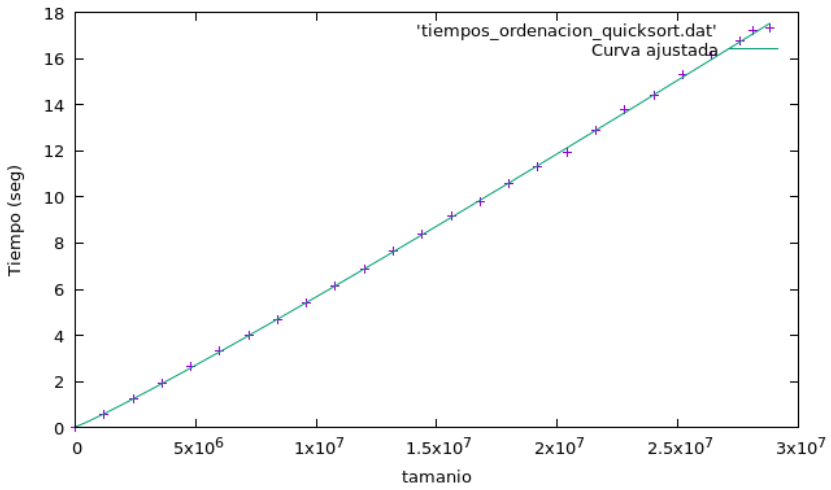

Figura 6: Gráfica algoritmo quicksort

Heapsort

El código para el algoritmo *heapsort* es:

```
1  /**
2     @file Ordenacin por montones
3  */
4
5
6  #include <iostream>
7  #include <chrono>
8  using namespace std;
9  using namespace chrono;
10 #include <ctime>
11 #include <cstdlib>
12 #include <climits>
13 #include <cassert>
14
15
16
17
18
19
20 /* ↩
       ************************************************************↩
        */
21 /*  Mtodo de ordenacin por montones  */
22
23 /**
24    @brief Ordena un vector por el mtodo de montones.
25
26    @param T: vector de elementos. Debe tener num_elem ↩
          elementos.
27           Es MODIFICADO.
28    @param num_elem: nmero de elementos. num_elem > 0.
29
30    Cambia el orden de los elementos de T de forma que los ↩
          dispone
31    en sentido creciente de menor a mayor.
32    Aplica el algoritmo de ordenacin por montones.
33 */
34 inline static
35 void heapsort(unsigned int T[], unsigned int num_elem);
36
37
38
39 /**
40    @brief Reajusta parte de un vector para que sea un montn.
41
42    @param T: vector de elementos. Debe tener num_elem ↩
          elementos.
43           Es MODIFICADO.
```

```
44    @param num_elem: nmero de elementos. num_elem > 0.
45    @param k: ndice del elemento que se toma com raz
46
47    Reajusta los elementos entre las posiciones k y num_elem ↩
          - 1
48    de T para que cumpla la propiedad de un montn (APO),
49    considerando al elemento en la posicin k como la raz.
50 */
51 static void reajustar(unsigned int T[], unsigned int ↩
       num_elem, unsigned int k);
52
53
54
55
56 /**
57    Implementacin de las funciones
58 **/
59
60
61 static void heapsort(unsigned int T[], unsigned int num_elem↩
       )
62 {
63   unsigned int i;
64   for (i = num_elem/2; i >= 0; i--)
65     reajustar(T, num_elem, i);
66   for (i = num_elem - 1; i >= 1; i--)
67     {
68       unsigned int aux = T[0];
69       T[0] = T[i];
70       T[i] = aux;
71       reajustar(T, i, 0);
72     }
73 }
74
75
76 static void reajustar(unsigned int T[], unsigned int ↩
       num_elem, unsigned int k)
77 {
78   unsigned int j;
79   unsigned int v;
80   v = T[k];
81   bool esAPO = false;
82   while ((k < num_elem/2) && !esAPO)
83     {
84       j = k + k + 1;
85       if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
86   j++;
87       if (v >= T[j])
88   esAPO = true;
```

```
89        T[k] = T[j];
90        k = j;
91      }
92    T[k] = v;
93  }
94
95
96  int main(int argc,  char *argv[])
97  {
98    if (argc != 2){
99      return 1;
100   }
101
102   unsigned int n = atoi(argv[1]);
103
104   high_resolution_clock::time_point tantes, tdespues;
105   duration<double> transcurrido;
106
107   int * T = new int[n];
108   assert(T);
109
110   srandom(time(0));
111
112   for (unsigned int i = 0; i < n; i++)
113     {
114       T[i] = random();
115     };
116
117   // escribe_vector(T, n);
118
119
120   tantes = high_resolution_clock::now();
121   heapsort(T, n);
122   tdespues = high_resolution_clock::now();
123   transcurrido = duration_cast<duration<double>>(tdespues - ↩
        tantes);
124
125   cout << n << " " << transcurrido.count() << endl;
126
127   // escribe_vector(T, n);
128
129
130   delete [] T;
131
132   return 0;
133 };
```

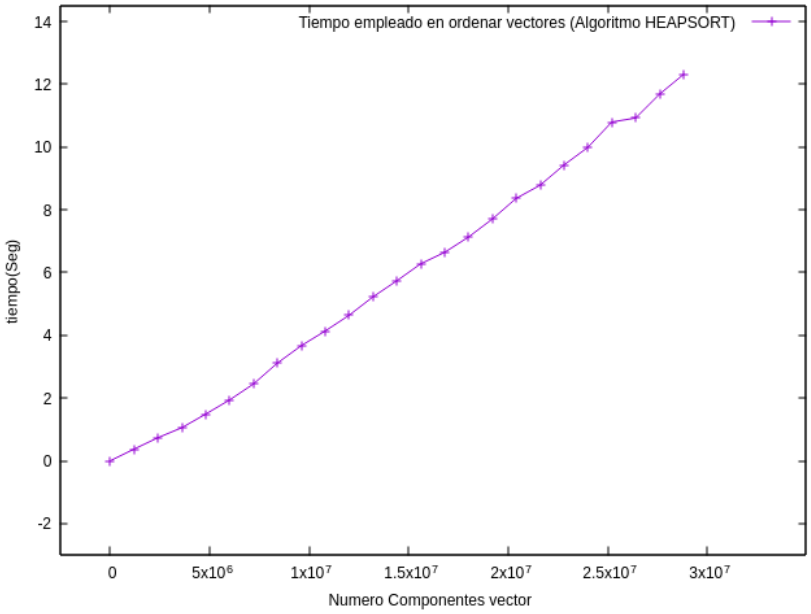

Figura 7: Gráfica algoritmo heapsort

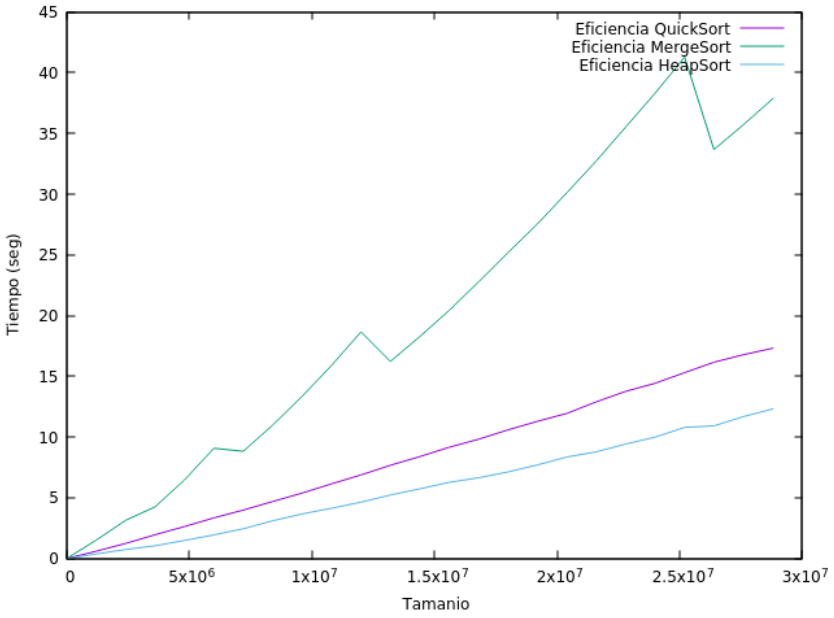

Figura 8: Gráfica algoritmos

En este caso también ocurre que algunos algoritmos son más rápidos que otros. El *heapsort* es el más rápido de los tres y su peor caso es $n \log n$ mientras que el *mergesort* es el más lento. En el *Quicksort* en su peor caso se obtienen tiempos cuadráticos, aunque en tiempo medio actúa como un algoritmo de familia $n \log n$. Como se ha dicho anteriormente, los tiempos de ejecución están condicionados por el hardware que ejecuta los algoritmos.

Tamaño del vector	Mergesort	Quicksort	Heapsort
1	6.7e-08	3e-06	4.98e-07
1200001	1.51248	0.601518	0.369181
2400001	3.13742	1.23973	0.734327
3600001	4.25173	1.9505	1.05172
4800001	6.4652	2.62888	1.48573
6000001	9.07117	3.34089	1.94025
7200001	8.83143	3.98903	2.44891
8400001	10.9863	4.69163	3.11307
9600001	13.35	5.3884	3.66539
10800001	15.8962	6.146	4.13483
12000001	18.6556	6.88565	4.63526
13200001	16.2255	7.6687	5.22195
14400001	18.276	8.3837	5.73521
15600001	20.4301	9.15389	6.27549
16800001	22.7814	9.81892	6.64332
18000001	25.1908	10.5894	7.12804
19200001	27.5473	11.2911	7.70287
20400001	30.1202	11.942	8.3599
21600001	32.7305	12.9	8.78117
22800001	35.5346	13.7612	9.42521
24000001	38.3152	14.4307	9.98922
25200001	41.2853	15.3049	10.7944
26400001	33.6681	16.1586	10.9223
27600001	35.7178	16.7763	11.6846
28800001	37.8503	17.3169	12.3082

Tabla 2: Tiempos de ejecución de la familia $O(n \log n)$

2.2. Algoritmo de Floyd

Este algoritmo calcula los caminos mínimos entre todos los pares de nodos en un grafo dirigido, su código es:

```
1  /**
2     @file Clculo del coste de los caminos mnimos. Algoritmo de Floyd.
3  */
4
5
6  #include <iostream>
7  using namespace std;
8  #include <ctime>
9  #include <cstdlib>
10 #include <climits>
11 #include <cassert>
12 #include <cmath>
13
14
15 static int const MAX_LONG  = 10;
16
17 /*************************************************************************
     */
18
19 /**
20    @brief Reserva espacio en memoria dinmica para una matriz cuadrada.
21
22    @param dim: dimensin de la matriz. dim > 0.
23
24    @returns puntero a la zona de memoria reservada.
25 */
26 int ** ReservaMatriz(int dim);
27
28
29 /*************************************************************************
     */
30
31 /**
32    @brief Libera el espacio asignado a una matriz cuadrada.
33
34    @param M: puntero a la zona de memoria reservada. Es MODIFICADO.
35    @param dim: dimensin de la matriz. dim > 0.
36
37    Liberar la zona memoria asignada a M y lo pone a NULL.
```

```
38 */
39 void LiberaMatriz(int ** & M, int dim);
40
41 /*↩
       **********************************************************************↩
       */
42
43 /**
44    @brief Rellena una matriz cuadrada con valores ↩
          aleaotorias.
45
46    @param M: puntero a la zona de memoria reservada. Es ↩
          MODIFICADO.
47    @param dim: dimensin de la matriz. dim > 0.
48
49    Asigna un valor aleatorio entero de [0, MAX_LONG - 1] a ↩
          cada
50    elemento de la matriz M, salvo los de la diagonal ↩
          principal
51    que quedan a 0..
52 */
53 void RellenaMatriz(int **M, int dim);
54
55 /*↩
       **********************************************************************↩
       */
56 /**
57    @brief Clculo de caminos mnimos.
58
59    @param M: Matriz de longitudes de los caminos. Es ↩
          MODIFICADO.
60    @param dim: dimensin de la matriz. dim > 0.
61
62    Calcula la longitud del camino mnimo entre cada par de ↩
          nodos (i,j),
63    que se almacena en M[i][j].
64 */
65 void Floyd(int **M, int dim);
66
67 /*↩
       **********************************************************************↩
       */
68
69
70 /**
71    Implementacin de las funciones
72 **/
73
74
```

```
75 int ** ReservaMatriz(int dim)
76 {
77   int **M;
78   if (dim  <= 0)
79     {
80       cerr<< "\n ERROR: Dimension de la matriz debe ser ↩
             mayor que 0" << endl;
81       exit(1);
82     }
83   M = new int * [dim];
84   if (M == NULL)
85     {
86       cerr << "\n ERROR: No puedo reservar memoria para un ↩
             matriz de "
87      << dim << " x " << dim << "elementos" << endl;
88       exit(1);
89     }
90   for (int i = 0; i < dim; i++)
91     {
92       M[i]= new int [dim];
93       if (M[i] == NULL)
94   {
95     cerr << "ERROR: No puedo reservar memoria para un matriz↩
           de "
96           << dim << " x " << dim << endl;
97     for (int j = 0; j < i; j++)
98       delete [] M[i];
99     delete [] M;
100    exit(1);
101  }
102    }
103  return M;
104 }
105
106
107 /*↩
     **********************************************************************↩
     */
108
109 void LiberaMatriz(int ** & M, int dim)
110 {
111   for (int i = 0; i < dim; i++)
112     delete [] M[i];
113   delete [] M;
114   M = NULL;
115 }
116
117
118 /*↩
```

```
     ************************************************************************↩
     */
119 void RellenaMatriz(int **M, int dim)
120 {
121   for (int i = 0; i < dim; i++)
122     for (int j = 0; j < dim; j++)
123       if (i != j)
124   M[i][j]= (rand() % MAX_LONG);
125 }
126
127
128 /*↩
     ************************************************************************↩
     */
129 void Floyd(int **M, int dim)
130 {
131   for (int k = 0; k < dim; k++)
132     for (int i = 0; i < dim;i++)
133       for (int j = 0; j < dim;j++)
134         {
135     int sum = M[i][k] + M[k][j];
136         M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
137         }
138 }
139
140
141 /*↩
     ************************************************************************↩
     */
142 int main (int argc, char **argv)
143 {
144   clock_t tantes;     // Valor del reloj antes de la ejecucin
145   clock_t tdespues;  // Valor del reloj despus de la ↩
       ejecucin
146   int dim;            // Dimensin de la matriz
147
148   //Lectura de los parametros de entrada
149   if (argc != 2)
150     {
151       cout << "Parmetros de entrada: " << endl
152      << "1.- Nmero de nodos" << endl << endl;
153       return 1;
154     }
155
156   dim = atoi(argv[1]);
157   int ** M = ReservaMatriz(dim);
158
159   RellenaMatriz(M,dim);
160
```

```
161
162   // Empieza el algoritmo de floyd
163   tantes = clock();
164   Floyd(M,dim);
165   tdespues = clock();
166   cout << "Tiempo: " << ((double)(tdespues-tantes))/↩
          CLOCKS_PER_SEC
167         << " s" << endl;
168   LiberaMatriz(M,dim);
169
170   return 0;
171 }
```

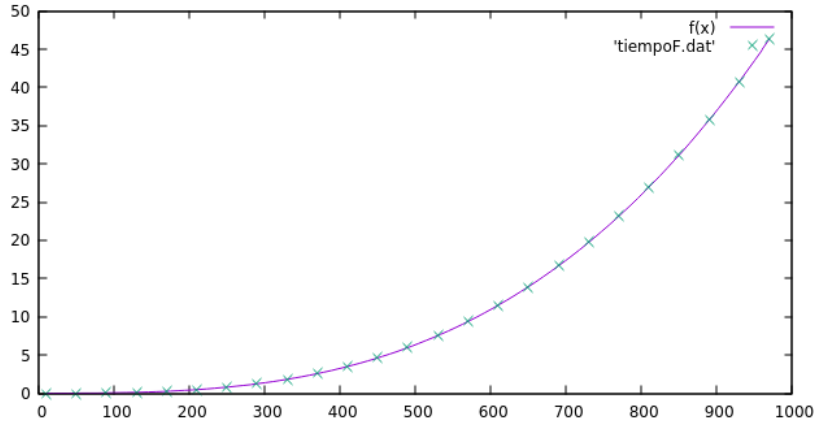

Figura 9: Gráfica algoritmo de Floyd

El script que se ha usado para calcular los tiempos de ejecución ha sido:

```
1  #!/bin/csh
2  @ inicio = 10
3  @ fin = 1000
4  @ incremento = 40
5  set ejecutable = floyd
6  set salida = tiempoFloyd.dat
7
8  @ i = $inicio
9  echo > $salida
10 while ( $i <= $fin )
11   echo Ejecución tam = $i
12   echo `./{$ejecutable} $i` >> $salida
13   @ i += $incremento
14 end
```

Tamaño del vector	Floyd
10	6.6e-05
50	0.007293
90	0.039217
130	0.113225
170	0.250322
210	0.467227
250	0.786888
290	1.23084
330	1.81614
370	2.57111
410	3.49241
450	4.61768
490	5.96742
530	7.55589
570	9.40896
610	11.5164
650	13.9302
690	16.6884
730	19.7581
770	23.1473
810	26.9431
850	31.1451
890	35.8169
930	40.7663
970	46.3376

Tabla 3: Tiempos de ejecución de Floyd $O(n^3)$

2.3. Algoritmo de las torres de Hanoi

El código de este algoritmo es:

```
1 /**
2    @file Resolucin del problema de las Torres de Hanoi
3 */
4
5
6 #include <iostream>
7 using namespace std;
8 #include <ctime>
```

```
9  #include <cstdlib>
10 #include <climits>
11 #include <cassert>
12
13
14 /**
15    @brief Resuelve el problema de las Torres de Hanoi
16    @param M: nmero de discos. M > 1.
17    @param i: nmero de columna en que estn los discos.
18              i es un valor de {1, 2, 3}. i != j.
19    @param j: nmero de columna a que se llevan los discos.
20              j es un valor de {1, 2, 3}. j != i.
21
22    Esta funcin imprime en la salida estndar la secuencia de
23    movimientos necesarios para desplazar los M discos de la
24    columna i a la j, observando la restriccin de que ningn
25    disco se puede situar sobre otro de tamao menor. Utiliza
26    una nica columna auxiliar.
27 */
28 void hanoi (int M, int i, int j);
29
30
31
32
33 void hanoi (int M, int i, int j)
34 {
35   if (M > 0)
36     {
37       hanoi(M-1, i, 6-i-j);
38       //cout << i << " -> " << j << endl;
39       hanoi (M-1, 6-i-j, j);
40   }
41 }
42
43 int main(int argc, char * argv[])
44 {
45
46   if(argc != 2){
47     cout << "Uso: " << argv[0] << " numDiscos" <<endl;
48   }
49
50   int M = atoi(argv[1]);
51
52   clock_t antes, despues;
53   antes = clock();
54
55   hanoi(M, 1, 2);
56
57   despues = clock();
```

```
58
59    cout << M << ' ' <<  despues-antes << endl;
60
61    return 0;
62 }
```

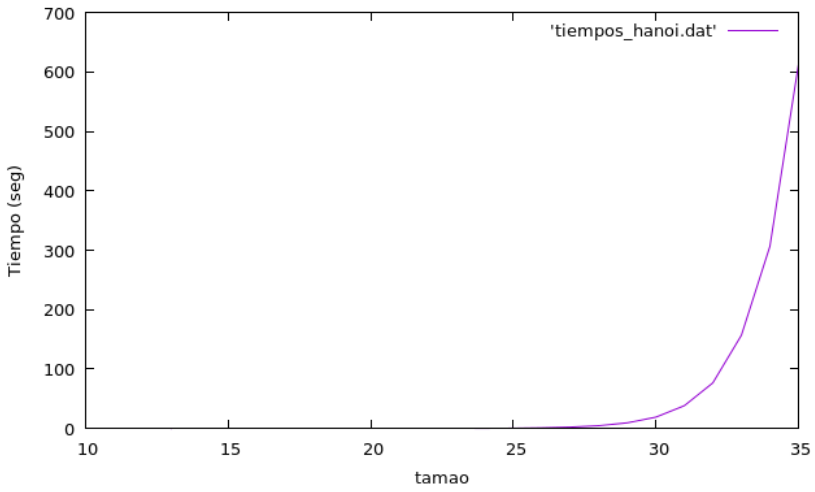

Figura 10: Gráfica algoritmo de Hanoi

Los valores obtenidos en la ejecución de Hanoi han sido:

Tamaño del vector	Hanoi
13	0.0002
14	0.000299
15	0.000634
16	0.001209
17	0.002486
18	0.004823
19	0.009548
20	0.019202
21	0.03786
22	0.075991
23	0.151952
24	0.303134
25	0.600663
26	1.20005
27	2.3963
28	4.80708
29	9.62316
30	19.111
31	38.4526
32	76.8348
33	157.017
34	306.513
35	611.917

Tabla 4: Tiempos de ejecución de Hanoi $O(2^n)$

A continuación mostramos la gráfica para todos los algoritmos de ordenación. En la gráfica vemos que los algoritmos $O(n^2)$ son más lentos que los algoritmos $O(n \log n)$ ya que para cantidades pequeñas del eje x tienen mucho más tiempo en ejecutarse mientras que los logarítmicos para valores muy grandes del eje x tienen unos tiempos de ejecución muy pequeños.

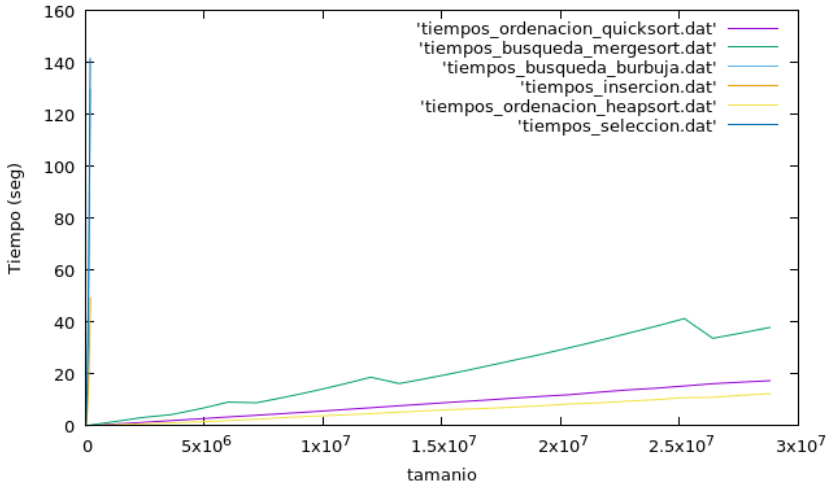

Figura 11: Gráfica algoritmos de ordenación

Si comparamos los algoritmos de Floyd y Hanoi en una gráfica podemos observar que la diferencia es bastante notable. Hanoi tarda 10 minutos con tamaños de 35 elementos.

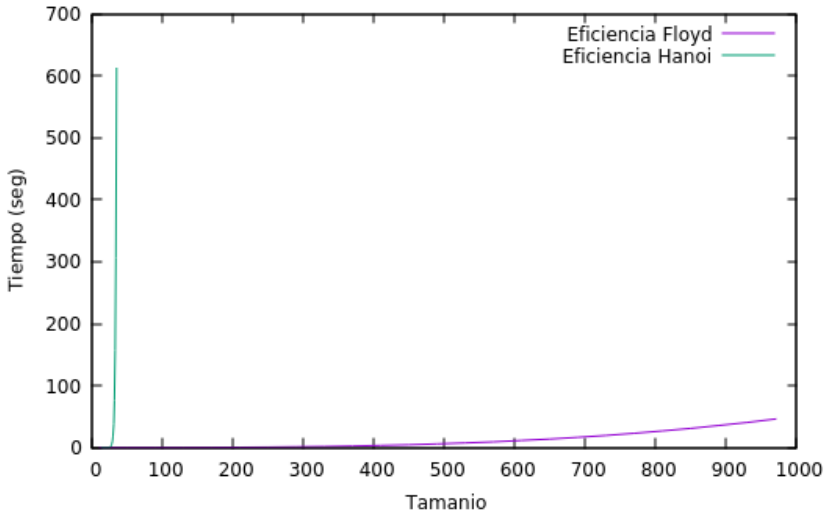

Figura 12: Gráfica algoritmos de Floyd y Hanoi