



ugr

Universidad
de Granada

ALGORÍTMICA
GRADO EN INGENIERÍA INFORMÁTICA

Práctica 1

Análisis de eficiencia de algoritmos

Autores

Maria Jesús López Salmerón
Nazaret Román Guerrero
Laura Hernández Muñoz
José Baena Cobos
Carlos Sánchez Páez



DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

CURSO 2017-2018

Índice

1. Descripción de la práctica	1
2. Código fuente a utilizar	1
2.1. Hanoi	1
2.2. Floyd	2
2.3. Algoritmos de ordenación	4
2.3.1. Burbuja	4
2.3.2. Selección	5
2.3.3. Inserción	6
2.3.4. <i>Heapsort</i>	7
2.3.5. <i>Mergesort</i>	8
2.3.6. <i>Quicksort</i>	11
3. Cálculo de la eficiencia empírica	13
3.1. Gráficas comparativas	13
3.2. Variación de la eficiencia empírica	13
4. Cálculo de la eficiencia híbrida	13

Índice de cuadros

1. Descripción de la práctica

El objetivo de la práctica es analizar la eficiencia de distintos algoritmos mediante tres métodos:

1. **Teórico:** obteniendo una expresión $T(n)$ que será convertida a notación $O(n)$
2. **Empírico:** ejecutando dicho algoritmo con distintos tamaños de problema y analizando el tiempo de realización del mismo frente a la cantidad de datos de entrada.
3. **Híbrido:** Hayando las constantes ocultas en la expresión $T(n)$ mediante los datos empíricos obtenidos anteriormente.

2. Código fuente a utilizar

Los algoritmos que utilizaremos para realizar la práctica han sido descargados de la plataforma decsai.ugr.es.

2.1. Hanoi

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5
6  /**
7   @brief Resuelve el problema de las
   ↳ Torres de Hanoi
8   @param M: número de discos. M > 1.
9   @param i: número de columna en que
   ↳ están los discos.
10          i es un valor de {1, 2,
   ↳ 3}. i != j.
11   @param j: número de columna a que
   ↳ se llevan los discos.
12          j es un valor de {1, 2,
   ↳ 3}. j != i.
13
14   Esta función imprime en la salida
   ↳ estándar la secuencia de
15   movimientos necesarios para
   ↳ desplazar los M discos de la
16   columna i a la j, observando la
   ↳ restricción de que ningún
17   disco se puede situar sobre otro de
   ↳ tamaño menor. Utiliza
18   una única columna auxiliar.
19  */
20 void hanoi (int M, int i, int j);
21
22 void hanoi (int M, int i, int j)
23 {
24     if (M > 0)
```

```
25     {
26         hanoi(M-1, i, 6-i-j);
27         cout << i << " -> " << j <<
   ↳ endl;
28         hanoi (M-1, 6-i-j, j);
29     }
30 }
31
32 int main(int argc, char * argv[])
33 {
34
35     if (argc != 2)
36     {
37         cerr << "Formato " << argv[0] <<
   ↳ " <num_discos>" << endl;
38         return -1;
39     }
40
41     int M = atoi(argv[1]);
42
43     hanoi(M, 1, 2);
44
45     return 0;
46 }
```

2.2. Floyd

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5  #include <climits>
6  #include <cassert>
7  #include <cmath>
8
9  static int const MAX_LONG = 10;
10
11 /**
12  @brief Reserva espacio en memoria
13  ↪ dinámica para una matriz
14  ↪ cuadrada.
15
16  @param dim: dimensión de la matriz.
17  ↪ dim > 0.
18
19  @returns puntero a la zona de
20  ↪ memoria reservada.
21 */
22 int ** ReservaMatriz(int dim);
23
24 /**
25  @brief Libera el espacio asignado a
26  ↪ una matriz cuadrada.
27
28  @param M: puntero a la zona de
29  ↪ memoria reservada. Es MODIFICADO.
30  @param dim: dimensión de la matriz.
31  ↪ dim > 0.
32
33  Liberar la zona memoria asignada a
34  ↪ M y lo pone a NULL.
35 */
36 void LiberaMatriz(int ** & M, int
37  ↪ dim);
38
39 /**
40  @brief Rellena una matriz cuadrada
41  ↪ con valores aleatorios.
42
43  @param M: puntero a la zona de
44  ↪ memoria reservada. Es MODIFICADO.
45  @param dim: dimensión de la matriz.
46  ↪ dim > 0.
47
48  Asigna un valor aleatorio entero de
49  ↪ [0, MAX_LONG - 1] a cada
50  ↪ elemento de la matriz M, salvo los
51  ↪ de la diagonal principal
52  ↪ que quedan a 0..
53 */
54 void RellenaMatriz(int **M, int dim);
55
56 /**
57  @brief Cálculo de caminos mínimos.
58
59  @param M: Matriz de longitudes de
60  ↪ los caminos. Es MODIFICADO.
61  @param dim: dimensión de la matriz.
62  ↪ dim > 0.
63
64  Calcula la longitud del camino
65  ↪ mínimo entre cada par de nodos
66  ↪ (i,j),
67  ↪ que se almacena en M[i][j].
68 */
69 void Floyd(int **M, int dim);
70
71 /**
72  Implementación de las funciones
73 */
74
75 int ** ReservaMatriz(int dim)
76 {
77     int **M;
78     if (dim <= 0)
79     {
80         cerr<< "\n ERROR: Dimension de
81         ↪ la matriz debe ser mayor que
82         ↪ 0" << endl;
83         exit(1);
84     }
85     M = new int * [dim];
86     if (M == NULL)
87     {
88         cerr << "\n ERROR: No puedo
89         ↪ reservar memoria para un
90         ↪ matriz de "
91         << dim << " x " << dim <<
92         ↪ "elementos" << endl;
93         exit(1);
94     }
95     for (int i = 0; i < dim; i++)
96     {
97         M[i]= new int [dim];
98         if (M[i] == NULL)
99         {
100             cerr << "ERROR: No puedo
101             ↪ reservar memoria para un
102             ↪ matriz de "
103             << dim << " x " << dim
104             ↪ << endl;
105             for (int j = 0; j < i; j++)
106                 delete [] M[i];
107             delete [] M;
108             exit(1);
109         }
110     }
111 }
```

```

85     }
86     return M;
87 }
88
89 void LiberaMatriz(int ** &M, int dim)
90 {
91     for (int i = 0; i < dim; i++)
92         delete [] M[i];
93     delete [] M;
94     M = NULL;
95 }
96
97 void RellenaMatriz(int **M, int dim)
98 {
99     for (int i = 0; i < dim; i++)
100         for (int j = 0; j < dim; j++)
101             if (i != j)
102                 M[i][j] = (rand() % MAX_LONG);
103 }
104
105
106 void Floyd(int **M, int dim)
107 {
108     for (int k = 0; k < dim; k++)
109         for (int i = 0; i < dim; i++)
110             for (int j = 0; j <
111                 ↪ dim; j++)
112                 {
113                     int sum = M[i][k] +
114                     ↪ M[k][j];
115                     M[i][j] = (M[i][j]
116                     ↪ > sum) ? sum :
117                     ↪ M[i][j];
118                 }
119 }
120
121 int main (int argc, char **argv)
122 {
123     // clock_t tantes;    // Valor del
124     ↪ reloj antes de la ejecución
125     // clock_t tdespues; // Valor del
126     ↪ reloj después de la ejecución
127     int dim;             // Dimensión de
128     ↪ la matriz
129
130     //Lectura de los parametros de
131     ↪ entrada
132     if (argc != 2)
133     {
134         cout << "Parámetros de entrada:
135         ↪ " << endl
136         << "1.- Número de nodos" <<
137         ↪ endl << endl;
138         return 1;
139     }
140
141     dim = atoi(argv[1]);
142     int ** M = ReservaMatriz(dim);
143
144     RellenaMatriz(M,dim);
145
146     // Empieza el algoritmo de floyd
147     // tantes = clock();
148     Floyd(M,dim);
149     // tdespues = clock();
150     // cout << "Tiempo: " << ((double)
151     // (tdespues-tantes))/CLOCKS_PER_SEC
152     // << " s" << endl;
153     LiberaMatriz(M,dim);
154
155     return 0;
156 }

```

2.3. Algoritmos de ordenación

2.3.1. Burbuja

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5  #include <climits>
6  #include <cassert>
7
8  /**
9   @brief Ordena un vector por el
   ↪ método de la burbuja.
10
11   @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
12           Es MODIFICADO.
13   @param num_elem: número de
   ↪ elementos. num_elem > 0.
14
15   Cambia el orden de los elementos de
   ↪ T de forma que los dispone
16   en sentido creciente de menor a
   ↪ mayor.
17   Aplica el algoritmo de la burbuja.
18 */
19 inline static
20 void burbuja(int T[], int num_elem);
21
22 /**
23   @brief Ordena parte de un vector
   ↪ por el método de la burbuja.
24
25   @param T: vector de elementos.
   ↪ Tiene un número de elementos
26           mayor o igual a
   ↪ final. Es MODIFICADO.
27
28   @param inicial: Posición que marca
   ↪ el inicio de la parte del
29           vector a ordenar.
30   @param final: Posición detrás de la
   ↪ última de la parte del
31           vector a ordenar.
32           inicial < final.
33
34   Cambia el orden de los elementos de
   ↪ T entre las posiciones
35   inicial y final - 1 de forma que los
   ↪ dispone en sentido creciente
36   de menor a mayor.
37   Aplica el algoritmo de la burbuja.
38 */
39 static void burbuja_lims(int T[], int
   ↪ inicial, int final);
40
41 /**
42   Implementación de las funciones
43   */
44
45 inline void burbuja(int T[], int
   ↪ num_elem)
46 {
47   burbuja_lims(T, 0, num_elem);
48 }
49
50 static void burbuja_lims(int T[], int
   ↪ inicial, int final)
51 {
52   int i, j;
53   int aux;
54   for (i = inicial; i < final - 1;
   ↪ i++)
55     for (j = final - 1; j > i; j--)
56       if (T[j] < T[j-1])
57         {
58           aux = T[j];
59           T[j] = T[j-1];
60           T[j-1] = aux;
61         }
62 }
63
64 int main(int argc, char * argv[])
65 {
66   if (argc != 2)
67   {
68     cerr << "Formato " << argv[0] <<
   ↪ " <num_elem>" << endl;
69     return -1;
70   }
71
72   int n = atoi(argv[1]);
73
74   int * T = new int[n];
75   assert(T);
76
77   srand(time(0));
78
79   for (int i = 0; i < n; i++)
80   {
81     T[i] = random();
82   }
83
84   burbuja(T, n);
85
86   delete [] T;
87
88   return 0;
89 }
```

2.3.2. Selección

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5  #include <climits>
6  #include <cassert>
7
8  /**
9   @brief Ordena un vector por el
   ↪ método de selección.
10
11  @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
12         Es MODIFICADO.
13  @param num_elem: número de
   ↪ elementos. num_elem > 0.
14
15  Cambia el orden de los elementos de
   ↪ T de forma que los dispone
16  en sentido creciente de menor a
   ↪ mayor.
17  Aplica el algoritmo de selección.
18 */
19 inline static
20 void seleccion(int T[], int num_elem);
21
22 /**
23  @brief Ordena parte de un vector
   ↪ por el método de selección.
24
25  @param T: vector de elementos.
   ↪ Tiene un número de elementos
26         mayor o igual a
   ↪ final. Es MODIFICADO.
27  @param inicial: Posición que marca
   ↪ el inicio de la parte del
28         vector a ordenar.
29  @param final: Posición detrás de la
   ↪ última de la parte del
30         vector a ordenar.
31         inicial < final.
32
33  Cambia el orden de los elementos de
   ↪ T entre las posiciones
34  inicial y final - 1 de forma que los
   ↪ dispone en sentido creciente
35  de menor a mayor.
36  Aplica el algoritmo de selección.
37 */
38 static void seleccion_lims(int T[],
   ↪ int inicial, int final);
39
40 /**
41  Implementación de las funciones
42 */
43
44 void seleccion(int T[], int num_elem)
45 {
46     seleccion_lims(T, 0, num_elem);
47 }
48
49 static void seleccion_lims(int T[],
   ↪ int inicial, int final)
50 {
51     int i, j, indice_menor;
52     int menor, aux;
53     for (i = inicial; i < final - 1;
   ↪ i++) {
54         indice_menor = i;
55         menor = T[i];
56         for (j = i; j < final; j++)
57             if (T[j] < menor) {
58                 indice_menor = j;
59                 menor = T[j];
60             }
61         aux = T[i];
62         T[i] = T[indice_menor];
63         T[indice_menor] = aux;
64     }
65 }
66
67 int main(int argc, char * argv[])
68 {
69     if (argc != 2){
70         cerr << "Formato " << argv[0] <<
   ↪ " <num_elem>" << endl;
71         return -1;
72     }
73
74     int n = atoi(argv[1]);
75
76     int * T = new int[n];
77     assert(T);
78
79     srand(time(0));
80
81     for (int i = 0; i < n; i++)
82     {
83         T[i] = random();
84     }
85
86     seleccion(T, n);
87
88     delete [] T;
89
90     return 0;
91 }
```

2.3.3. Inserción

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5  #include <climits>
6  #include <cassert>
7
8  /**
9   @brief Ordena un vector por el
   ↪ método de inserción.
10
11   @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
12           Es MODIFICADO.
13   @param num_elem: número de
   ↪ elementos. num_elem > 0.
14
15   Cambia el orden de los elementos de
   ↪ T de forma que los dispone
16   en sentido creciente de menor a
   ↪ mayor.
17   Aplica el algoritmo de inserción.
18 */
19 inline static
20 void insercion(int T[], int num_elem);
21
22 /**
23   @brief Ordena parte de un vector
   ↪ por el método de inserción.
24
25   @param T: vector de elementos.
   ↪ Tiene un número de elementos
26           mayor o igual a
   ↪ final. Es MODIFICADO.
27   @param inicial: Posición que marca
   ↪ el inicio de la parte del
28           vector a ordenar.
29   @param final: Posición detrás de la
   ↪ última de la parte del
30           vector a ordenar.
31           inicial < final.
32
33   Cambia el orden de los elementos de
   ↪ T entre las posiciones
34   inicial y final - 1 de forma que los
   ↪ dispone en sentido creciente
35   de menor a mayor.
36   Aplica el algoritmo de inserción.
37 */
38 static void insercion_lims(int T[],
   ↪ int inicial, int final);
39
40 /**
41   Implementación de las funciones
42   **/
43
44 inline static void insercion(int T[],
   ↪ int num_elem)
45 {
46   insercion_lims(T, 0, num_elem);
47 }
48
49 static void insercion_lims(int T[],
   ↪ int inicial, int final)
50 {
51   int i, j;
52   int aux;
53   for (i = inicial + 1; i < final;
   ↪ i++) {
54     j = i;
55     while ((T[j] < T[j-1]) && (j > 0))
   ↪ {
56       aux = T[j];
57       T[j] = T[j-1];
58       T[j-1] = aux;
59       j--;
60     };
61   };
62 }
63
64 int main(int argc, char * argv[])
65 {
66
67   if (argc != 2)
68   {
69     cerr << "Formato " << argv[0] <<
   ↪ " <num_elem>" << endl;
70     return -1;
71   }
72
73   int n = atoi(argv[1]);
74
75   int * T = new int[n];
76   assert(T);
77
78   srand(time(0));
79
80   for (int i = 0; i < n; i++)
81   {
82     T[i] = random();
83   };
84
85   insercion(T, n);
86
87   delete [] T;
88
89   return 0;
90 };
```


2.3.4. Heapsort

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5  #include <climits>
6  #include <cassert>
7
8  /**
9   * @brief Ordena un vector por el
10   ↪ método de montones.
11
12   * @param T: vector de elementos. Debe
13   ↪ tener num_elem elementos. Es
14   ↪ MODIFICADO.
15   * @param num_elem: número de
16   ↪ elementos. num_elem > 0.
17
18   * Cambia el orden de los elementos de
19   ↪ T de forma que los dispone
20   ↪ en sentido creciente de menor a
21   ↪ mayor. Aplica el algoritmo de
22   ↪ ordenación por montones.
23   */
24 inline static
25 void heapsort(int T[], int num_elem);
26
27 /**
28   * @brief Reajusta parte de un vector
29   ↪ para que sea un montón.
30
31   * @param T: vector de elementos. Debe
32   ↪ tener num_elem elementos.
33   ↪ Es MODIFICADO.
34   * @param num_elem: número de
35   ↪ elementos. num_elem > 0.
36   * @param k: índice del elemento que
37   ↪ se toma como raíz
38
39   * Reajusta los elementos entre las
40   ↪ posiciones k y num_elem - 1
41   ↪ de T para que cumpla la propiedad
42   ↪ de un montón (APO),
43   ↪ considerando al elemento en la
44   ↪ posición k como la raíz.
45   */
46 static void reajustar(int T[], int
47 ↪ num_elem, int k);
48
49 /**Implementación de las funciones**/
50
51 static void heapsort(int T[], int
52 ↪ num_elem)
53 {
54     int i;
55     for (i = num_elem/2; i >= 0; i--)
```

```

56         reajustar(T, num_elem, i);
57     for (i = num_elem - 1; i >= 1; i--){
58         int aux = T[0];
59         T[0] = T[i];
60         T[i] = aux;
61         reajustar(T, i, 0);
62     }
63 }
64
65 static void reajustar(int T[], int
66 ↪ num_elem, int k)
67 {
68     int j;
69     int v;
70     v = T[k];
71     bool esAPO = false;
72     while ((k < num_elem/2) && !esAPO)
73     {
74         j = k + k + 1;
75         if ((j < (num_elem - 1)) &&
76             ↪ (T[j] < T[j+1]))
77             j++;
78         if (v >= T[j])
79             esAPO = true;
80         T[k] = T[j];
81         k = j;
82     }
83     T[k] = v;
84 }
85
86 int main(int argc, char * argv[]){
87
88     if (argc != 2){
89         cerr << "Formato " << argv[0] <<
90         ↪ " <num_elem>" << endl;
91         return -1;
92     }
93
94     int n = atoi(argv[1]);
95
96     int * T = new int[n];
97     assert(T);
98
99     srand(time(0));
100
101     for (int i = 0; i < n; i++)
102         T[i] = random();
103
104     heapsort(T, n);
105
106     delete [] T;
107
108     return 0;
109 };
```

2.3.5. Mergesort

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5  #include <climits>
6  #include <cassert>
7
8  /**
9   @brief Ordena un vector por el
   ↪ método de mezcla.
10
11   @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
12       Es MODIFICADO.
13   @param num_elem: número de
   ↪ elementos. num_elem > 0.
14
15   Cambia el orden de los elementos de
   ↪ T de forma que los dispone
16   en sentido creciente de menor a
   ↪ mayor.
17   Aplica el algoritmo de mezcla.
18 */
19 inline static
20 void mergesort(int T[], int num_elem);
21
22 /**
23   @brief Ordena parte de un vector
   ↪ por el método de mezcla.
24
25   @param T: vector de elementos.
   ↪ Tiene un número de elementos
26       mayor o igual a
   ↪ final. Es MODIFICADO.
27   @param inicial: Posición que marca
   ↪ el inicio de la parte del
28       vector a ordenar.
29   @param final: Posición detrás de la
   ↪ última de la parte del
30       vector a ordenar.
31       inicial < final.
32
33   Cambia el orden de los elementos de
   ↪ T entre las posiciones
34   inicial y final - 1 de forma que
   ↪ los dispone en sentido creciente
35   de menor a mayor.
36   Aplica el algoritmo de la mezcla.
37 */
38 static void mergesort_lims(int T[],
   ↪ int inicial, int final);
39
40 /**
41   @brief Ordena un vector por el
   ↪ método de inserción.
```

```
42
43   @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
44       Es MODIFICADO.
45   @param num_elem: número de
   ↪ elementos. num_elem > 0.
46
47   Cambia el orden de los elementos de
   ↪ T de forma que los dispone
48   en sentido creciente de menor a
   ↪ mayor.
49   Aplica el algoritmo de inserción.
50 */
51 inline static
52 void insercion(int T[], int num_elem);
53
54 /**
55   @brief Ordena parte de un vector
   ↪ por el método de inserción.
56
57   @param T: vector de elementos.
   ↪ Tiene un número de elementos
58       mayor o igual a
   ↪ final. Es MODIFICADO.
59   @param inicial: Posición que marca
   ↪ el inicio de la parte del
60       vector a ordenar.
61   @param final: Posición detrás de la
   ↪ última de la parte del
62       vector a ordenar.
63       inicial < final.
64
65   Cambia el orden de los elementos de
   ↪ T entre las posiciones
66   inicial y final - 1 de forma que
   ↪ los dispone en sentido creciente
67   de menor a mayor.
68   Aplica el algoritmo de la
   ↪ inserción.
69 */
70 static void insercion_lims(int T[],
   ↪ int inicial, int final);
71
72 /**
73   @brief Mezcla dos vectores
   ↪ ordenados sobre otro.
74
75   @param T: vector de elementos.
   ↪ Tiene un número de elementos
76       mayor o igual a
   ↪ final. Es MODIFICADO.
77   @param inicial: Posición que marca
   ↪ el inicio de la parte del
78       vector a escribir.
```

```

79      @param final: Posición detrás de la
      ↪ última de la parte del
80          vector a escribir
81          inicial < final.
82      @param U: Vector con los elementos
      ↪ ordenados.
83      @param V: Vector con los elementos
      ↪ ordenados.
84          El número de elementos de
      ↪ U y V sumados debe coincidir
85          con final - inicial.
86
87      En los elementos de T entre las
      ↪ posiciones inicial y final - 1
88      pone ordenados en sentido
      ↪ creciente, de menor a mayor, los
89      elementos de los vectores U y V.
90  */
91  static void fusion(int T[], int
      ↪ inicial, int final, int U[], int
      ↪ V[]);
92
93  /**
94      Implementación de las funciones
95  **/
96
97  inline static void insercion(int T[],
      ↪ int num_elem)
98  {
99      insercion_lims(T, 0, num_elem);
100 }
101
102 static void insercion_lims(int T[],
      ↪ int inicial, int final)
103 {
104     int i, j;
105     int aux;
106     for (i = inicial + 1; i < final;
      ↪ i++) {
107         j = i;
108         while ((T[j] < T[j-1]) && (j > 0))
      ↪ {
109             aux = T[j];
110             T[j] = T[j-1];
111             T[j-1] = aux;
112             j--;
113         };
114     };
115 }
116
117 const int UMBRAL_MS = 100;
118
119 void mergesort(int T[], int num_elem)
120 {
121     mergesort_lims(T, 0, num_elem);
122 }
123
124 static void mergesort_lims(int T[],
      ↪ int inicial, int final)
125 {
126     if (final - inicial < UMBRAL_MS)
127     {
128         insercion_lims(T, inicial,
      ↪ final);
129     } else {
130         int k = (final - inicial)/2;
131
132         int * U = new int [k - inicial +
      ↪ 1];
133         assert(U);
134         int l, l2;
135         for (l = 0, l2 = inicial; l < k;
      ↪ l++, l2++)
136             U[l] = T[l2];
137         U[l] = INT_MAX;
138
139         int * V = new int [final - k +
      ↪ 1];
140         assert(V);
141         for (l = 0, l2 = k; l < final -
      ↪ k; l++, l2++)
142             V[l] = T[l2];
143         V[l] = INT_MAX;
144
145         mergesort_lims(U, 0, k);
146         mergesort_lims(V, 0, final - k);
147         fusion(T, inicial, final, U, V);
148         delete [] U;
149         delete [] V;
150     };
151 }
152
153 static void fusion(int T[], int
      ↪ inicial, int final, int U[], int
      ↪ V[])
154 {
155     int j = 0;
156     int k = 0;
157     for (int i = inicial; i < final;
      ↪ i++)
158     {
159         if (U[j] < V[k]) {
160             T[i] = U[j];
161             j++;
162         } else{
163             T[i] = V[k];
164             k++;
165         };
166     };
167 }

```

```

168
169 int main(int argc, char * argv[])
170 {
171
172     if (argc != 2)
173     {
174         cerr << "Formato " << argv[0] <<
            ↳ " <num_elem>" << endl;
175         return -1;
176     }
177
178     int n = atoi(argv[1]);
179
180     int * T = new int[n];
181     assert(T);
182
183     srand(time(0));
184
185     for (int i = 0; i < n; i++)
186     {
187         T[i] = random();
188     }
189
190     mergesort(T, n);
191
192     delete [] T;
193
194     return 0;
195 };

```

2.3.6. Quicksort

```
1  #include <iostream>
2  using namespace std;
3  #include <ctime>
4  #include <cstdlib>
5  #include <climits>
6  #include <cassert>
7
8  /**
9   @brief Ordena un vector por el
   ↪ método quicksort.
10
11  @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
12      Es MODIFICADO.
13  @param num_elem: número de
   ↪ elementos. num_elem > 0.
14
15  Cambia el orden de los elementos de
   ↪ T de forma que los dispone
16  en sentido creciente de menor a
   ↪ mayor.
17  Aplica el algoritmo quicksort.
18 */
19 inline static
20 void quicksort(int T[], int num_elem);
21
22 /**
23  @brief Ordena parte de un vector
   ↪ por el método quicksort.
24
25  @param T: vector de elementos.
   ↪ Tiene un número de elementos
26      mayor o igual a
   ↪ final. Es MODIFICADO.
27  @param inicial: Posición que marca
   ↪ el inicio de la parte del
28      vector a ordenar.
29  @param final: Posición detrás de la
   ↪ última de la parte del
30      vector a ordenar.
31      inicial < final.
32
33  Cambia el orden de los elementos de
   ↪ T entre las posiciones
34  inicial y final - 1 de forma que
   ↪ los dispone en sentido creciente
35  de menor a mayor.
36  Aplica el algoritmo quicksort.
37 */
38 static void quicksort_lims(int T[],
   ↪ int inicial, int final);
39
40 /**
41
42  @brief Ordena un vector por el
   ↪ método de inserción.
43
44  @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
45      Es MODIFICADO.
46  @param num_elem: número de
   ↪ elementos. num_elem > 0.
47
48  Cambia el orden de los elementos de
   ↪ T de forma que los dispone
49  en sentido creciente de menor a
   ↪ mayor.
50  Aplica el algoritmo de inserción.
51 */
52 inline static
53 void insercion(int T[], int num_elem);
54
55 /**
56  @brief Ordena parte de un vector
   ↪ por el método de inserción.
57
58  @param T: vector de elementos.
   ↪ Tiene un número de elementos
59      mayor o igual a
   ↪ final. Es MODIFICADO.
60  @param inicial: Posición que marca
   ↪ el inicio de la parte del
61      vector a ordenar.
62  @param final: Posición detrás de la
   ↪ última de la parte del
63      vector a ordenar.
64      inicial < final.
65
66  Cambia el orden de los elementos de
   ↪ T entre las posiciones
67  inicial y final - 1 de forma que
   ↪ los dispone en sentido creciente
68  de menor a mayor.
69  Aplica el algoritmo de inserción.
70 */
71 static void insercion_lims(int T[],
   ↪ int inicial, int final);
72
73 /**
74
75  @brief Redistribuye los elementos
   ↪ de un vector según un pivote.
76
77  @param T: vector de elementos.
   ↪ Tiene un número de elementos
78      mayor o igual a
   ↪ final. Es MODIFICADO.
79  @param inicial: Posición que marca
   ↪ el inicio de la parte del
```

```

80         vector a ordenar.
81         @param final: Posición detrás de la
82         ↪ última de la parte del
83         vector a ordenar.
84         inicial < final.
85         @param pp: Posición del pivote. Es
86         ↪ MODIFICADO.
87
88         Selecciona un pivote los elementos
89         ↪ de T situados en las posiciones
90         ↪ entre inicial y final - 1.
91         ↪ Redistribuye los elementos,
92         ↪ situando los
93         ↪ menores que el pivote a su
94         ↪ izquierda, después los iguales y a
95         ↪ la
96         ↪ derecha los mayores. La posición
97         ↪ del pivote se devuelve en pp.
98         */
99         static void dividir_qs(int T[], int
100         ↪ inicial, int final, int & pp);
101
102         /**
103         Implementación de las funciones
104         **/
105
106         inline static void insercion(int T[],
107         ↪ int num_elem)
108         {
109             insercion_lims(T, 0, num_elem);
110         }
111
112         static void insercion_lims(int T[],
113         ↪ int inicial, int final)
114         {
115             int i, j;
116             int aux;
117             for (i = inicial + 1; i < final;
118             ↪ i++) {
119                 j = i;
120                 while ((T[j] < T[j-1]) && (j > 0))
121                 ↪ {
122                     aux = T[j];
123                     T[j] = T[j-1];
124                     T[j-1] = aux;
125                     j--;
126                 };
127             };
128         }
129
130         const int UMBRAL_QS = 50;
131
132         inline void quicksort(int T[], int
133         ↪ num_elem)
134         {
135             quicksort_lims(T, 0, num_elem);
136         }
137
138         static void quicksort_lims(int T[],
139         ↪ int inicial, int final)
140         {
141             int k;
142             if (final - inicial < UMBRAL_QS) {
143                 insercion_lims(T, inicial, final);
144             } else {
145                 dividir_qs(T, inicial, final, k);
146                 quicksort_lims(T, inicial, k);
147                 quicksort_lims(T, k + 1, final);
148             };
149         }
150
151         static void dividir_qs(int T[], int
152         ↪ inicial, int final, int & pp)
153         {
154             int pivote, aux;
155             int k, l;
156
157             pivote = T[inicial];
158             k = inicial;
159             l = final;
160             do {
161                 k++;
162             } while ((T[k] <= pivote) && (k <
163             ↪ final-1));
164             do {
165                 l--;
166             } while (T[l] > pivote);
167             while (k < l) {
168                 aux = T[k];
169                 T[k] = T[l];
170                 T[l] = aux;
171                 do k++; while (T[k] <= pivote);
172                 do l--; while (T[l] > pivote);
173             };
174             aux = T[inicial];
175             T[inicial] = T[l];
176             T[l] = aux;
177             pp = l;
178         };
179
180         int main(int argc, char * argv[])
181         {
182             if (argc != 2)
183             {
184                 cerr << "Formato " << argv[0] <<
185                 ↪ " <num_elem>" << endl;
186                 return -1;
187             }
188
189             int n = atoi(argv[1]);

```

```
172
173     int * T = new int[n];
174     assert(T);
175
176     srand(time(0));
177
178     for (int i = 0; i < n; i++)
179     {
180         T[i] = random();
181     };
182
183     quicksort(T, n);
184
185     delete [] T;
186
187     return 0;
188 };
```

3. Cálculo de la eficiencia empírica

3.1. Gráficas comparativas

3.2. Variación de la eficiencia empírica

4. Cálculo de la eficiencia híbrida