



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

<b>1. Descripción de la práctica</b>	<b>1</b>
<b>2. Código fuente a utilizar</b>	<b>1</b>
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> . . . . .	10
<b>3. Cálculo de la eficiencia empírica</b>	<b>12</b>
3.1. Scripts desarrollados . . . . .	12
3.2. Entornos de ejecución . . . . .	15
3.2.1. PC 1 . . . . .	15
3.2.2. PC 2 . . . . .	15
3.3. Gráficas comparativas . . . . .	16
3.3.1. Algoritmos con eficiencia $O(n^2)$ . . . . .	16
3.3.2. Algoritmo con eficiencia $O(n^3)$ . . . . .	17
3.3.3. Algoritmos con eficiencia $O(n \cdot \log(n))$ . . . . .	18
3.3.4. Algoritmo con eficiencia $O(2^n)$ . . . . .	19
3.3.5. Comparación entre algoritmos de ordenación . . . . .	19
3.4. Variación de la eficiencia empírica . . . . .	21
3.4.1. Algoritmos con eficiencia $O(n^2)$ . . . . .	21
3.4.2. Algoritmos con eficiencia $O(n^3)$ . . . . .	23
3.4.3. Algoritmos con eficiencia $O(n \cdot \log(n))$ . . . . .	23
3.4.4. Algoritmo con eficiencia $O(2^n)$ . . . . .	25
3.4.5. Algoritmos con eficiencia $O(n \cdot \log(n))$ . . . . .	25
<b>4. Cálculo de la eficiencia híbrida</b>	<b>25</b>
4.0.1. Algoritmos con eficiencia $O(n^2)$ . . . . .	25
4.0.2. Algoritmos con eficiencia $O(n^3)$ . . . . .	25
4.0.3. Algoritmos con eficiencia $O(n \cdot \log(n))$ . . . . .	25
4.0.4. Algoritmo con eficiencia $O(2^n)$ . . . . .	25
4.0.5. Algoritmos con eficiencia $O(n \cdot \log(n))$ . . . . .	25

# Índice de cuadros

1. Parámetros de ejecución de cada programa . . . . .	14
2. Comparación entre ambos entornos de ejecución . . . . .	21

# 1. Descripción de la práctica

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

1. **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.
2. **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     clock_t tantes; // Valor del
   ↳ reloj antes de la ejecución
44     clock_t tdespues; // Valor del
   ↳ reloj después de la ejecución
45     tantes = clock();
46     hanoi(M, 1, 2);
47     tdespues = clock();
48     cout << M <<
49     ((double)(tdespues-tantes))
50     /CLOCKS_PER_SEC << endl;
51     return 0;
52 }
```

## 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() %
103                     ↪ MAX_LONG);
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;
124     // clock_t tdespues;
125     int dim;
126
127     //Lectura de los parametros de
128     ↪ entrada
129     if (argc != 2)
130     {
131         cout << "Parámetros de entrada:
132             ↪ " << endl
133             << "1.- Número de nodos" <<
134             ↪ endl << endl;
135         return 1;
136     }
137
138     dim = atoi(argv[1]);
139
140     int ** M = ReservaMatriz(dim);
141
142     RellenaMatriz(M,dim);
143
144     // Empieza el algoritmo de floyd
145     tantes = clock();
146     Floyd(M,dim);
147     tdespues = clock();
148     cout << dim <<
149     ((double)(tdespues-tantes))
150     /CLOCKS_PER_SEC << endl;
151     LiberaMatriz(M,dim);
152
153     return 0;

```

## 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   if (argc != 2){
66     cerr << "Formato " << argv[0] <<
   ↪ " <num_elem>" << endl;
67     return -1;
68   }
69   int n = atoi(argv[1]);
70   int * T = new int[n];
71   assert(T);
72   srand(time(0));
73
74   for (int i = 0; i < n; i++)
75     T[i] = random();
76
77   clock_t tantes;
78   clock_t tdespues;
79   tantes = clock();
80   burbuja(T, n);
81   tdespues = clock();
82   cout << n <<
83   ((double)(tdespues-tantes))
84   /CLOCKS_PER_SEC << endl;
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. Aplica el algoritmo de
   ↪ selección.
17 */
18 inline static
19 void seleccion(int T[], int num_elem);
20
21 /**
22   @brief Ordena parte de un vector
   ↪ por el método de selección.
23
24   @param T: vector de elementos.
   ↪ Tiene un número de elementos
25           mayor o igual a
   ↪ final. Es MODIFICADO.
26   @param inicial: Posición que marca
   ↪ el inicio de la parte del
27           vector a ordenar.
28   @param final: Posición detrás de la
   ↪ última de la parte del
29           vector a ordenar.
30           inicial < final.
31
32   Cambia el orden de los elementos de
   ↪ T entre las posiciones
33   inicial y final - 1 de forma que los
   ↪ dispone en sentido creciente
34   de menor a mayor. Aplica el
   ↪ algoritmo de selección.
35 */
36 static void seleccion_lims(int T[],
   ↪ int inicial, int final);
37 /**
38   Implementación de las funciones
39 */
40
41 void seleccion(int T[], int num_elem){
42     seleccion_lims(T, 0, num_elem);
43 }
44
45 static void seleccion_lims(int T[],
   ↪ int inicial, int final){
46     int i, j, indice_menor;
47     int menor, aux;
48     for (i = inicial; i < final - 1;
   ↪ i++) {
49         indice_menor = i;
50         menor = T[i];
51         for (j = i; j < final; j++)
52             if (T[j] < menor) {
53                 indice_menor = j;
54                 menor = T[j];
55             }
56         aux = T[i];
57         T[i] = T[indice_menor];
58         T[indice_menor] = aux;
59     }
60 }
61
62 int main(int argc, char * argv[]){
63     if (argc != 2){
64         cerr << "Formato " << argv[0] <<
   ↪ " <num_elem>" << endl;
65         return -1;
66     }
67     int n = atoi(argv[1]);
68     int * T = new int[n];
69     assert(T);
70     srand(time(0));
71
72     for (int i = 0; i < n; i++)
73         T[i] = random();
74
75     clock_t tantes; // Valor del
   ↪ reloj antes de la ejecución
76     clock_t tdespues; // Valor del
   ↪ reloj después de la ejecución
77     tantes = clock();
78     seleccion(T, n);
79     tdespues = clock();
80     cout << n <<
   ↪ ((double)(tdespues-tantes))
81     /CLOCKS_PER_SEC << endl;
82
83     delete [] T;
84     return 0;
85 }
```

### 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     insercion_lims(T, 0, num_elem);
46 }
47
48 static void insercion_lims(int T[],
   ↳ int inicial, int final){
49     int i, j;
50     int aux;
51     for (i = inicial + 1; i < final;
   ↳ i++) {
52         j = i;
53         while ((T[j] < T[j-1]) && (j > 0))
   ↳ {
54             aux = T[j];
55             T[j] = T[j-1];
56             T[j-1] = aux;
57             j--;
58         };
59     };
60 }
61
62 int main(int argc, char * argv[]){
63     if (argc != 2){
64         cerr << "Formato " << argv[0] <<
   ↳ " <num_elem>" << endl;
65         return -1;
66     }
67     int n = atoi(argv[1]);
68
69     int * T = new int[n];
70     assert(T);
71
72     srand(time(0));
73
74     for (int i = 0; i < n; i++)
75         T[i] = random();
76
77     clock_t tantes; // Valor del
   ↳ reloj antes de la ejecución
78     clock_t tdespues; // Valor del
   ↳ reloj después de la ejecución
79     tantes = clock();
80     insercion(T, n);
81     tdespues = clock();
82     cout << n <<
   ↳ ((double)(tdespues-tantes))
83     /CLOCKS_PER_SEC << endl;
84     delete [] T;
85
86     return 0;
87 };
```



### 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
   ↪ método de montones.
10
11   @param T: vector de elementos. Debe
   ↪ tener num_elem elementos. Es
   ↪ MODIFICADO.
12   @param num_elem: número de
   ↪ elementos. num_elem > 0.
13
14   Cambia el orden de los elementos de
   ↪ T de forma que los dispone
   ↪ en sentido creciente de menor a
   ↪ mayor. Aplica el algoritmo de
   ↪ ordenación por montones.
15 */
16 inline static
17 void heapsort(int T[], int num_elem);
18
19 /**
20 @brief Reajusta parte de un vector
   ↪ para que sea un montón.
21
22 @param T: vector de elementos. Debe
   ↪ tener num_elem elementos.
   ↪ Es MODIFICADO.
23 @param num_elem: número de
   ↪ elementos. num_elem > 0.
24 @param k: índice del elemento que
   ↪ se toma como raíz
25
26 Reajusta los elementos entre las
   ↪ posiciones k y num_elem - 1
   ↪ de T para que cumpla la propiedad
   ↪ de un montón (APO),
   ↪ considerando al elemento en la
   ↪ posición k como la raíz.
27 */
28 static void reajustar(int T[], int
   ↪ num_elem, int k);
29
30 /**Implementación de las funciones**/
31
32 static void heapsort(int T[], int
   ↪ num_elem){
33     int i;
34     for (i = num_elem/2; i >= 0; i--){
35         reajustar(T, num_elem, i);
36     }
37 }
38
39 for (i = num_elem - 1; i >= 1; i--){
40     int aux = T[0];
41     T[0] = T[i];
42     T[i] = aux;
43     reajustar(T, i, 0);
44 }
45
46 static void reajustar(int T[], int
   ↪ num_elem, int k){
47     int j;
48     int v;
49     v = T[k];
50     bool esAPO = false;
51     while ((k < num_elem/2) && !esAPO)
52     {
53         j = k + k + 1;
54         if ((j < (num_elem - 1)) &&
   ↪ (T[j] < T[j+1]))
55             j++;
56         if (v >= T[j])
57             esAPO = true;
58         T[k] = T[j];
59         k = j;
60     }
61     T[k] = v;
62 }
63
64 int main(int argc, char * argv[]){
65     if (argc != 2){
66         cerr << "Formato " << argv[0] <<
   ↪ " <num_elem>" << endl;
67         return -1;
68     }
69     int n = atoi(argv[1]);
70     int * T = new int[n];
71     assert(T);
72     srand(time(0));
73     for (int i = 0; i < n; i++){
74         T[i] = random();
75     }
76
77     clock_t tantes; // Valor del
   ↪ reloj antes de la ejecución
78     clock_t tdespues; // Valor del
   ↪ reloj después de la ejecución
79     tantes = clock();
80     heapsort(T, n);
81     tdespues = clock();
82     cout << n <<
   ↪ ((double)(tdespues-tantes))
83     /CLOCKS_PER_SEC << endl;
84
85     delete [] T;
86     return 0;
87 };
```

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

```
39  Es MODIFICADO.
40  @param num_elem: número de
   * ↪ elementos. num_elem > 0.
41
42  Cambia el orden de los elementos de
   * ↪ T de forma que los dispone
43  * en sentido creciente de menor a
   * ↪ mayor. Aplica el algoritmo de
   * ↪ inserción.
44  */
45 inline static
46 void insercion(int T[], int num_elem);
47 /**
48  * @brief Ordena parte de un vector
   * ↪ por el método de inserción.
49
50  * @param T: vector de elementos.
   * ↪ Tiene un número de elementos
51  * mayor o igual a
   * ↪ final. Es MODIFICADO.
52  * @param inicial: Posición que marca
   * ↪ el inicio de la parte del
53  * vector a ordenar.
54  * @param final: Posición detrás de la
   * ↪ última de la parte del
55  * vector a ordenar.
   * ↪ inicial < final.
56  * Cambia el orden de los elementos de
   * ↪ T entre las posiciones
57  * inicial y final - 1 de forma que
   * ↪ los dispone en sentido creciente
58  * de menor a mayor. Aplica el
   * ↪ algoritmo de la inserción.
59  */
60 static void insercion_lims(int T[],
   * ↪ int inicial, int final);
61 /**
62  * @brief Mezcla dos vectores
   * ↪ ordenados sobre otro.
63
64  * @param T: vector de elementos.
   * ↪ Tiene un número de elementos
65  * mayor o igual a
   * ↪ final. Es MODIFICADO.
66  * @param inicial: Posición que marca
   * ↪ el inicio de la parte del
67  * vector a escribir.
68  * @param final: Posición detrás de la
   * ↪ última de la parte del
69  * vector a escribir
   * ↪ inicial < final.
70
71  * @param U: Vector con los elementos
   * ↪ ordenados.
72  * @param V: Vector con los elementos
   * ↪ ordenados.
```

```

73      El número de elementos de
74      ↪ U y V sumados debe coincidir
75      ↪ con final - inicial.
76      En los elementos de T entre las
77      ↪ posiciones inicial y final - 1
78      ↪ pone ordenados en sentido
79      ↪ creciente, de menor a mayor, los
80      ↪ elementos de los vectores U y V.
81      */
82      Implementación de las funciones
83      */
84      inline static void insercion(int T[],
85      ↪ int num_elem){
86          insercion_lims(T, 0, num_elem);
87      }
88      static void insercion_lims(int T[],
89      ↪ int inicial, int final){
90          int i, j;
91          int aux;
92          for (i = inicial + 1; i < final;
93          ↪ i++) {
94              j = i;
95              while ((T[j] < T[j-1]) && (j > 0))
96                  ↪ {
97                  aux = T[j];
98                  T[j] = T[j-1];
99                  T[j-1] = aux;
100                  j--;
101              };
102          };
103      }
104      const int UMBRAL_MS = 100;
105      void mergesort(int T[], int num_elem){
106          mergesort_lims(T, 0, num_elem);
107      }
108      static void mergesort_lims(int T[],
109      ↪ int inicial, int final){
110          if (final - inicial < UMBRAL_MS)
111              ↪ insercion_lims(T, inicial,
112              ↪ final);
113          else {
114              int k = (final - inicial)/2;
115              int * U = new int [k - inicial +
116              ↪ 1];
117              assert(U);
118              int l, l2;
119              for (l = 0, l2 = inicial; l < k;
120              ↪ l++, l2++)
121                  U[l] = T[l2];
122              U[l] = INT_MAX;
123          }
124      }
125      static void fusion(int T[], int
126      ↪ inicial, int final, int U[], int
127      ↪ V[]){
128          int j = 0;
129          int k = 0;
130          for (int i = inicial; i < final;
131          ↪ i++)
132              if (U[j] < V[k]) {
133                  T[i] = U[j];j++;
134              } else{
135                  T[i] = V[k];k++;
136              }
137          }
138      }
139      int main(int argc, char * argv[]){
140          if (argc != 2){
141              cerr << "Formato " << argv[0] <<
142              ↪ " <num_elem>" << endl;
143              return -1;
144          }
145          int n = atoi(argv[1]);
146          int * T = new int[n];
147          assert(T);
148          srand(time(0));
149          for (int i = 0; i < n; i++)
150              T[i] = random();
151          clock_t tantes; // Valor del
152          ↪ reloj antes de la ejecución
153          clock_t tdespues; // Valor del
154          ↪ reloj después de la ejecución
155          tantes = clock();
156          mergesort(T, n);
157          tdespues = clock();
158          cout << n <<
159          ↪ ((double)(tdespues-tantes))
160          ↪ /CLOCKS_PER_SEC << endl;
161          delete [] T;
162          return 0;
163      };

```

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

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

```

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

```

### 3. Cálculo de la eficiencia empírica

#### 3.1. Scripts desarrollados

Hemos ejecutado cada código 25 veces mediante la creación de dos scripts en Shell Bash, uno que ejecuta cada programa individualmente y otro que se sirve del primero para ejecutarlos todos con tamaños acordes a su eficiencia ( un algoritmo de eficiencia  $O(n \cdot \log(n))$  se puede ejecutar tranquilamente con un tamaño de problema del orden de millones de datos, pero uno con una eficiencia peor (por ejemplo,  $O(2^n)$  ) tendrá que ejecutarse con un tamaño del orden de decenas.

```
1  #!/bin/bash
2
3  #Primer argumento: programa a ejecutar
4  #Segundo argumento: tamaño inicial
5  #Tercer argumento : incremento
6
7  if [ $# -eq 3 ]
8  then
9      i="0"
10     output="out"
11     tam=$2
12     while [ $i -lt 25 ]
13     do
14         ./$1 $tam >> $1.out
15         i=$((i+1))
16         tam=$((tam+$3))
17     done
18 else
19     echo "Error de argumentos"
20 fi
```

Listing 1: Script individual

```

1  #!/bin/bash
2  echo "Ejecutando burbuja..."
3  ./individual.sh burbuja 1000 1000
4  echo "Ejecutando insercion..."
5  ./individual.sh insercion 1000 1000
6  echo "Ejecutando seleccion..."
7  ./individual.sh seleccion 1000 1000
8  echo "Ejecutando mergesort..."
9  ./individual.sh mergesort 1000000 500000
10 echo "Ejecutando quicksort..."
11 ./individual.sh quicksort 1000000 500000
12 echo "Ejecutando heapsort..."
13 ./individual.sh heapsort 1000000 500000
14 echo "Ejecutando hanoi..."
15 ./individual.sh hanoi 10 1
16 echo "Ejecutando floyd..."
17 ./individual.sh floyd 100 100

```

Listing 2: Script conjunto

También hemos diseñado un archivo *Makefile* para que la compilación sea más sencilla.

```

DOC=doc
SRC=src
OUT=out
BIN=src

all : todos
todos : burbuja floyd hanoi heapsort insercion mergesort quicksort seleccion
        cd $(SRC) ; ./todos.sh
burbuja :
        g++ -o ./$(BIN)/burbuja ./$(SRC)/burbuja.cpp
floyd :
        g++ -o ./$(BIN)/floyd ./$(SRC)/floyd.cpp
hanoi :
        g++ -o ./$(BIN)/hanoi ./$(SRC)/hanoi.cpp
heapsort :
        g++ -o ./$(BIN)/heapsort ./$(SRC)/heapsort.cpp
insercion :
        g++ -o ./$(BIN)/insercion ./$(SRC)/insercion.cpp
mergesort :
        g++ -o ./$(BIN)/mergesort ./$(SRC)/mergesort.cpp
quicksort :
        g++ -o ./$(BIN)/quicksort ./$(SRC)/quicksort.cpp
seleccion :
        g++ -o ./$(BIN)/seleccion ./$(SRC)/seleccion.cpp

```

Listing 3: Makefile

Cada programa ha sido modificado añadiendo las siguientes líneas para que su salida sea el tiempo de ejecución:

```

1      clock_t tantes;
2      clock_t tdespues;
3      tantes = clock();
4      algoritmo_en_cuestion(T, n);
5      tdespues = clock();
6      cout << ((double)(tdespues - tantes))
7      / CLOCKS_PER_SEC << endl;

```

Listing 4: Código fuente modificado

Los parámetros con los que se ejecutan los programas son los siguientes:

Algoritmo	Eficiencia	Tamaño inicial	Incremento
Burbuja	$O(n^2)$	1000	1000
Inserción	$O(n^2)$	1000	1000
Selección	$O(n^2)$	1000	1000
Mergesort	$O(n \cdot \log(n))$	1.000.000	500.000
Quicksort	$O(n \cdot \log(n))$	1.000.000	500.000
Heapsort	$O(n \cdot \log(n))$	1.000.000	500.000
Floyd	$O(n^3)$	100	100
Hanoi	$O(2^n)$	10	1

Cuadro 1: Parámetros de ejecución de cada programa



## 3.2. Entornos de ejecución

Todas las ejecuciones se realizarán en el *PC 1*, excepto las que utilicemos para la comparación en el apartado de *Variación de la eficiencia empírica*.

### 3.2.1. PC 1

#### 1. CPU : AMD FX-8320 @3.50Ghz

- a) Arquitectura : x86\_64
- b) Caché
  - 1) Caché L1
    - $a'$  L1d : 16K
    - $b'$  L1i : 64K
  - 2) Caché L2 : 2048K
  - 3) Caché L3 : 8192K
- c) Frecuencia máxima (Overclock) : 4.20 Ghz
- d) Núcleos físicos : 4
- e) Núcleos lógicos : 8

#### 2. RAM

- a) Capacidad : 16384 MB
- b) Frecuencia : 1600 Mhz
- c) Tecnología : DDR3

### 3.2.2. PC 2

#### 1. CPU : Intel Core i7-6700HQ @2.60Ghz

- a) Arquitectura : x86\_64
- b) Caché
  - 1) Caché L1
    - $a'$  L1d : 32K
    - $b'$  L1i : 32K
  - 2) Caché L2 : 256K
  - 3) Caché L3 : 6144K
- c) Frecuencia máxima (HT) : 3.5 Ghz
- d) Núcleos físicos : 4
- e) Núcleos lógicos : 8

#### 2. RAM

- a) Capacidad : 8192 MB
- b) Frecuencia : 2133 Mhz
- c) Tecnología : DDR4

### 3.3. Gráficas comparativas

#### 3.3.1. Algoritmos con eficiencia $O(n^2)$

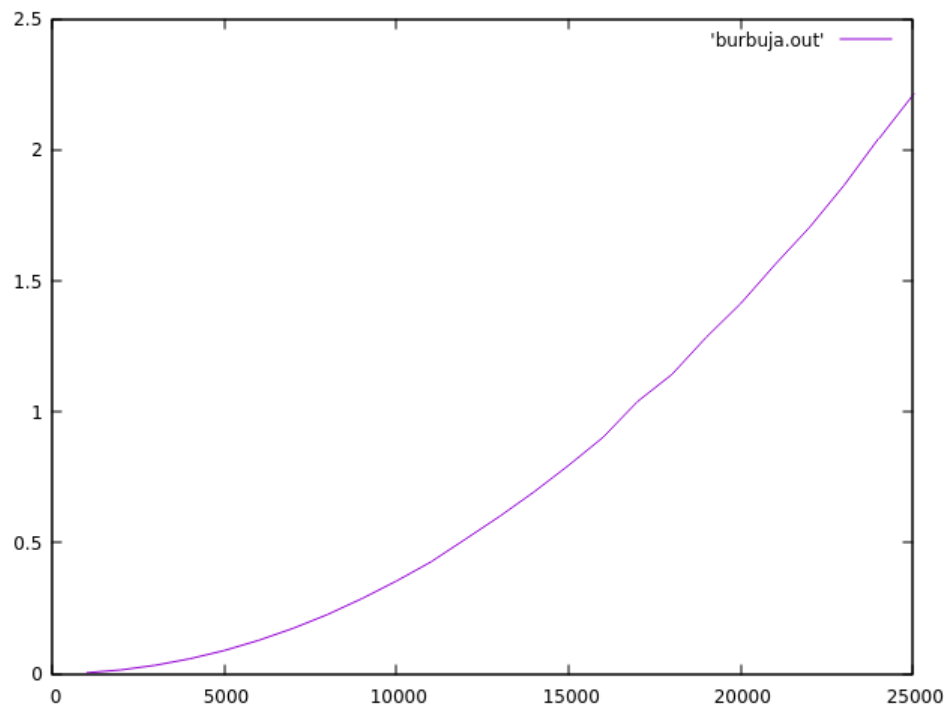


Figura 1: Algoritmo burbuja

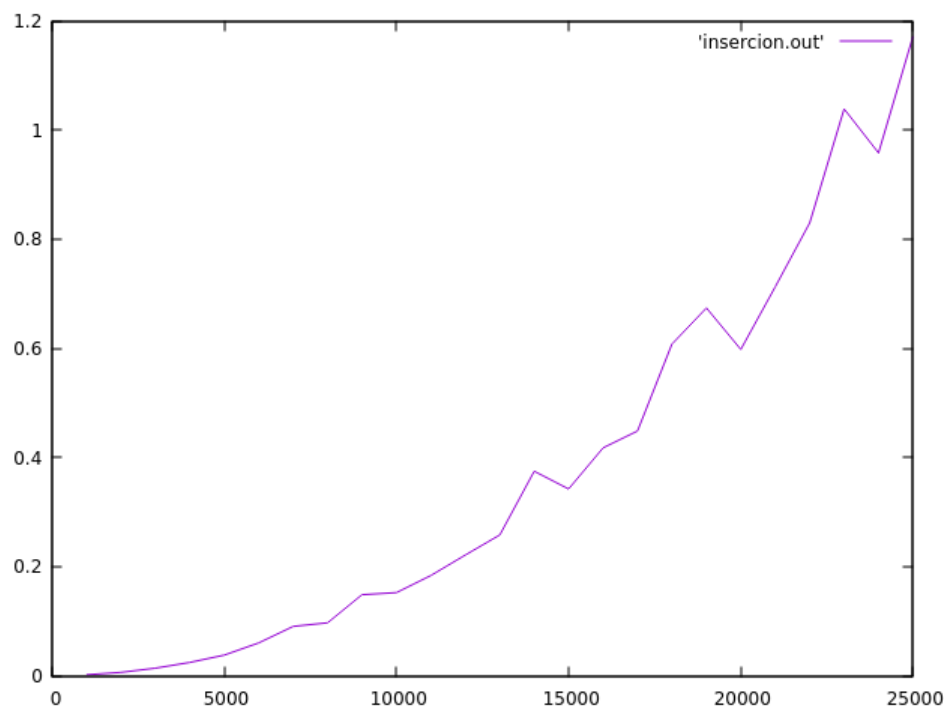


Figura 2: Algoritmo de inserción

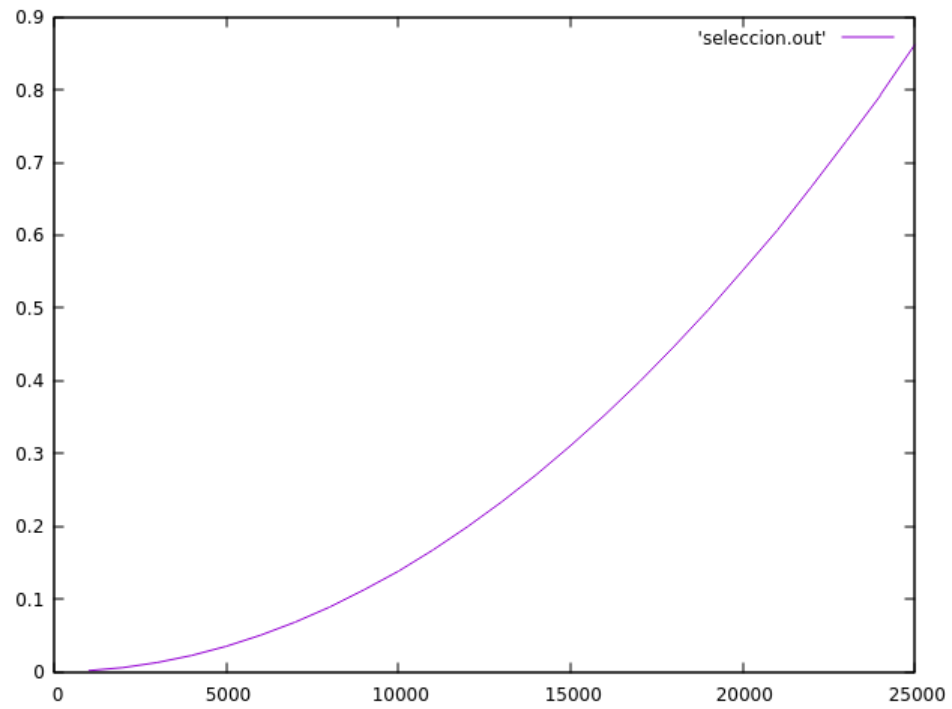


Figura 3: Algoritmo de selección

### 3.3.2. Algoritmo con eficiencia $O(n^3)$

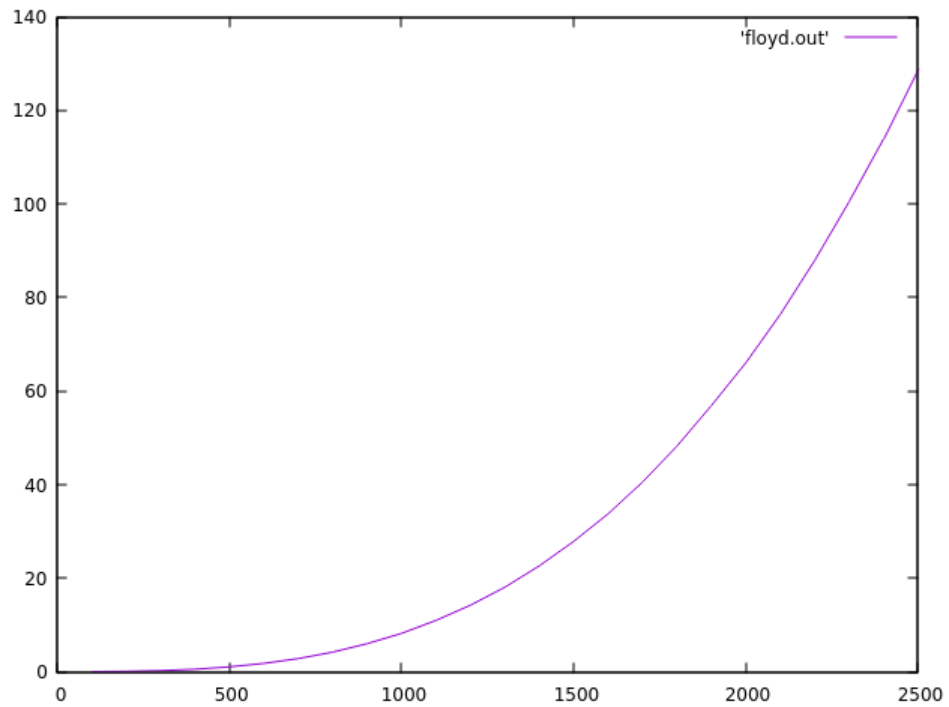


Figura 4: Algoritmo de Floyd

### 3.3.3. Algoritmos con eficiencia $O(n \cdot \log(n))$

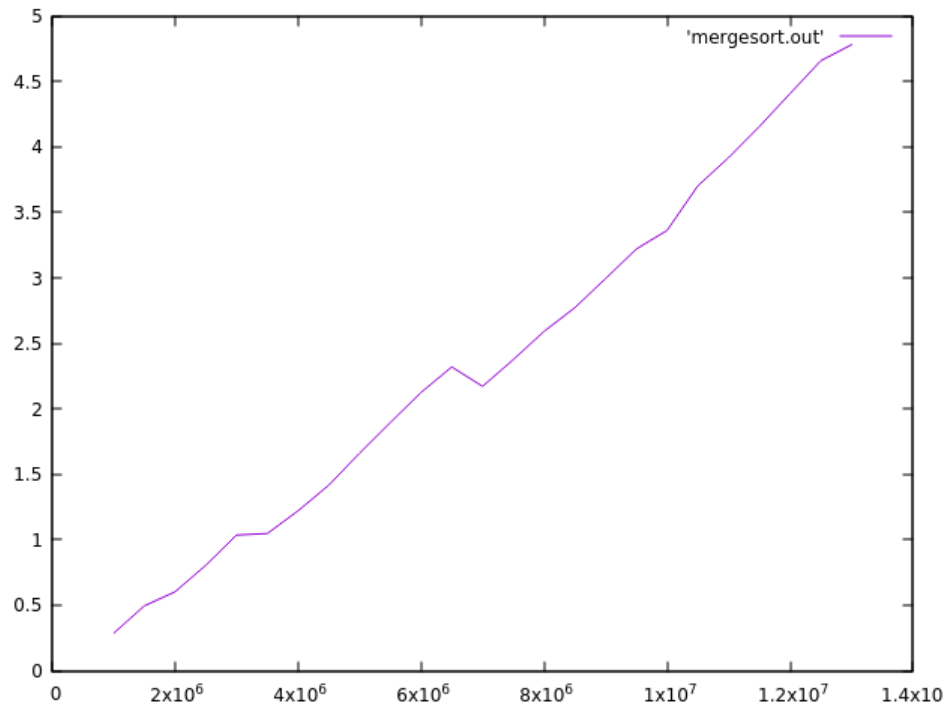


Figura 5: Algoritmo mergesort

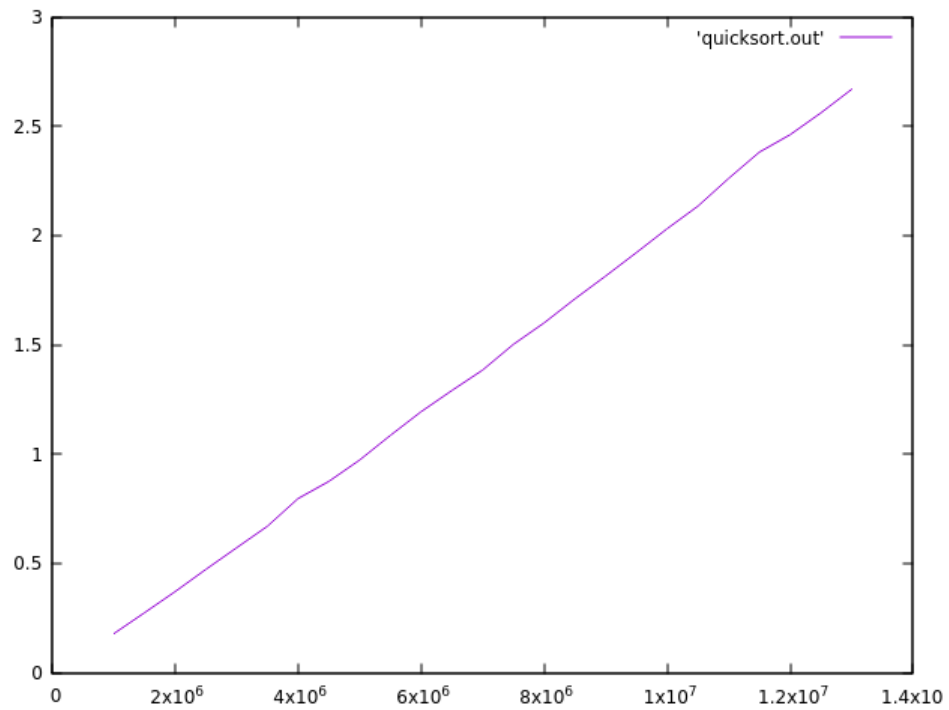


Figura 6: Algoritmo quicksort

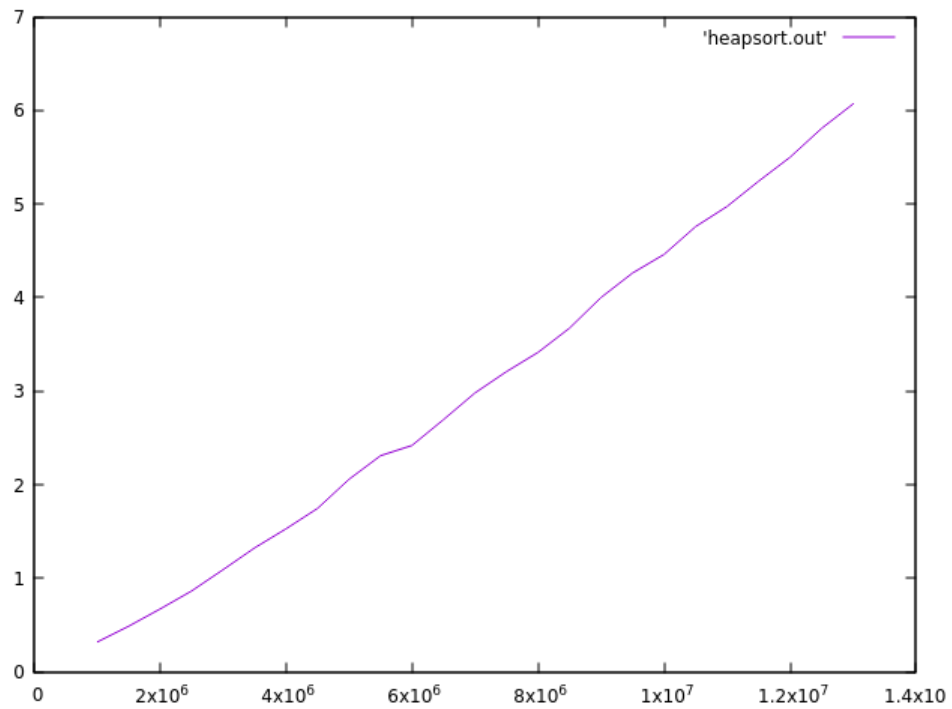


Figura 7: Algoritmo heapsort

#### 3.3.4. Algoritmo con eficiencia $O(2^n)$

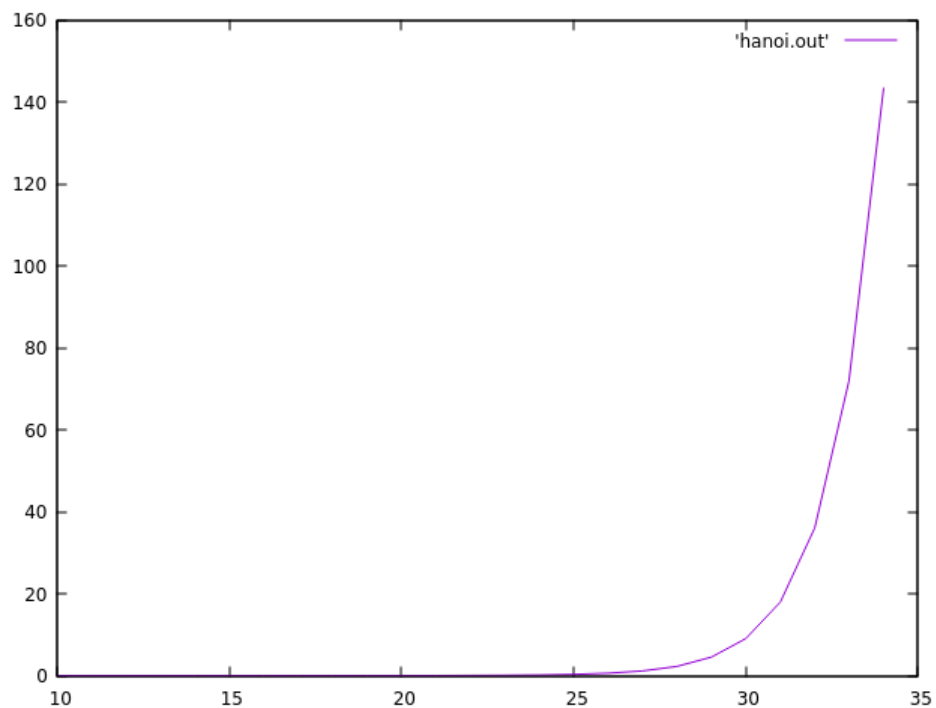


Figura 8: Algoritmo Hanoi

#### 3.3.5. Comparación entre algoritmos de ordenación

A simple vista solo podremos ver el trabajo de los algoritmos rápidos (*heapsort*, *mergesort* y *quicksort*), ya que trabajan con tamaños de problema muy superiores al resto de

algoritmos.

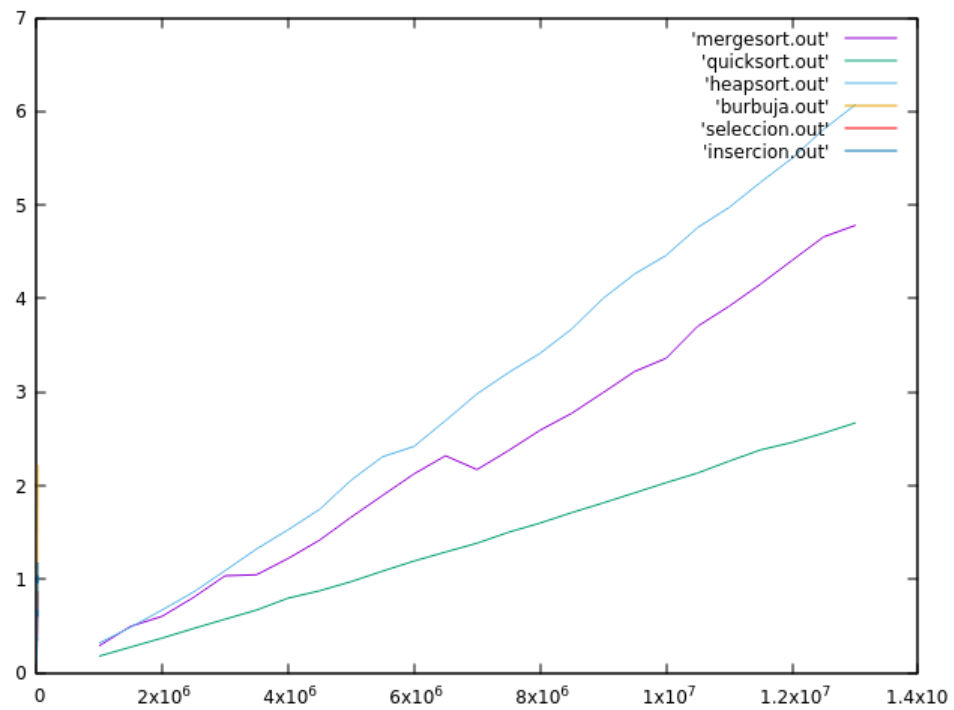


Figura 9: Comparación de algoritmos de ordenación

Si hacemos zoom, podremos ver mejor la diferencia:

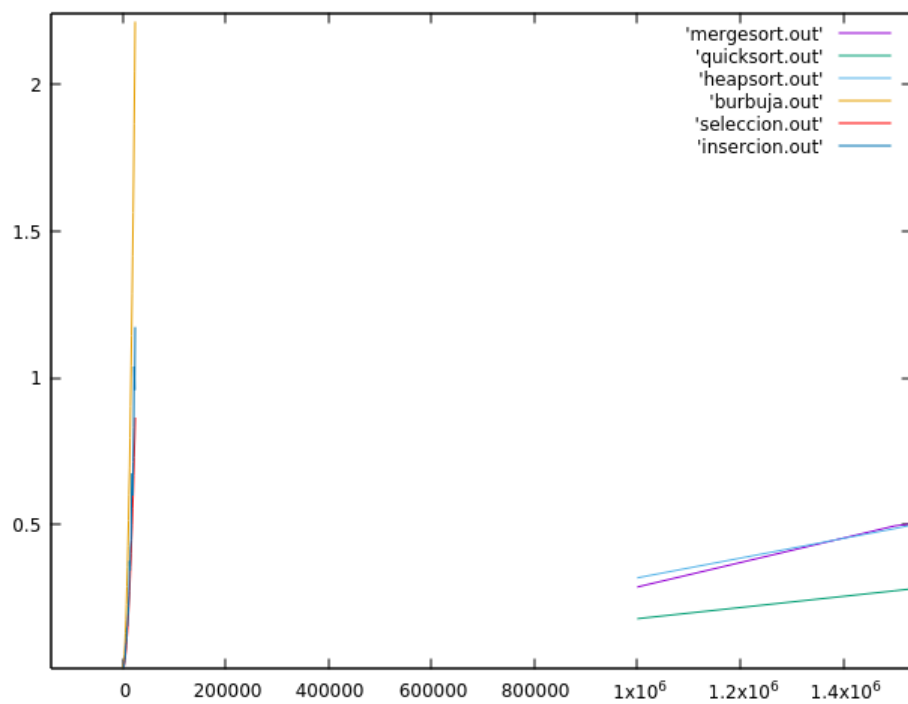


Figura 10: Comparación de algoritmos de ordenación (zoom)

### 3.4. Variación de la eficiencia empírica

En esta sección demostraremos empíricamente el *Principio de invarianza*.

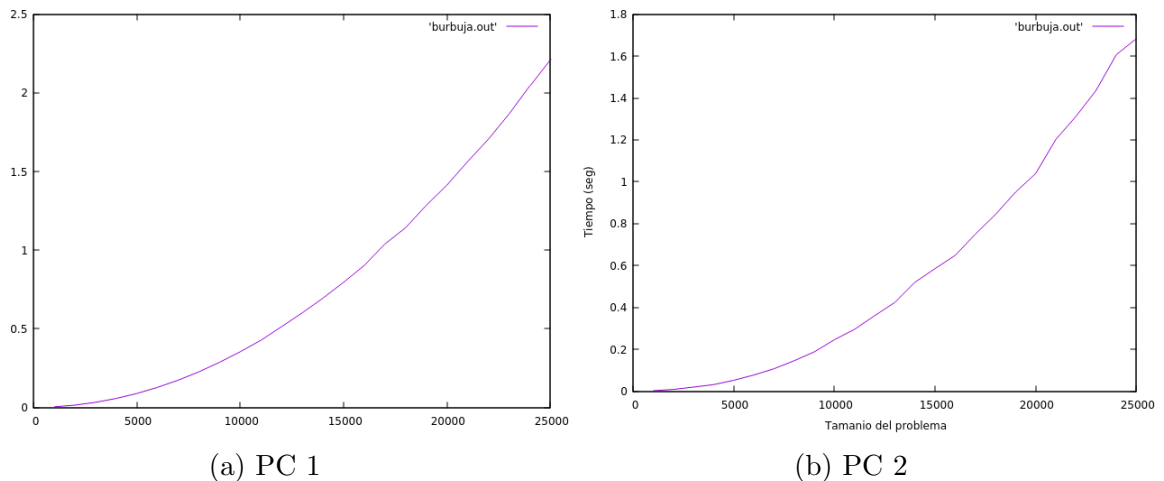
**Principio 1 (de Invarianza)** *Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia más que, a lo sumo, en una constante multiplicativa.*

Para ello, ejecutaremos los algoritmos en una arquitectura distinta, el *PC 2*. Comenzamos con una tabla donde podremos observar la constante en cuestión según el tiempo medio de ejecución de cada algoritmo:

Algoritmo	Tiempo medio PC 1	Tiempo medio PC 2	Constante
Burbuja	pc1	pc2	cte
Inserción	pc1	pc2	cte
Selección	pc1	pc2	cte
Mergesort	pc1	pc2	cte
Quicksort	pc1	pc2	cte
Heapsort	pc1	pc2	cte
Floyd	pc1	pc2	cte
Hanoi	pc1	pc2	cte

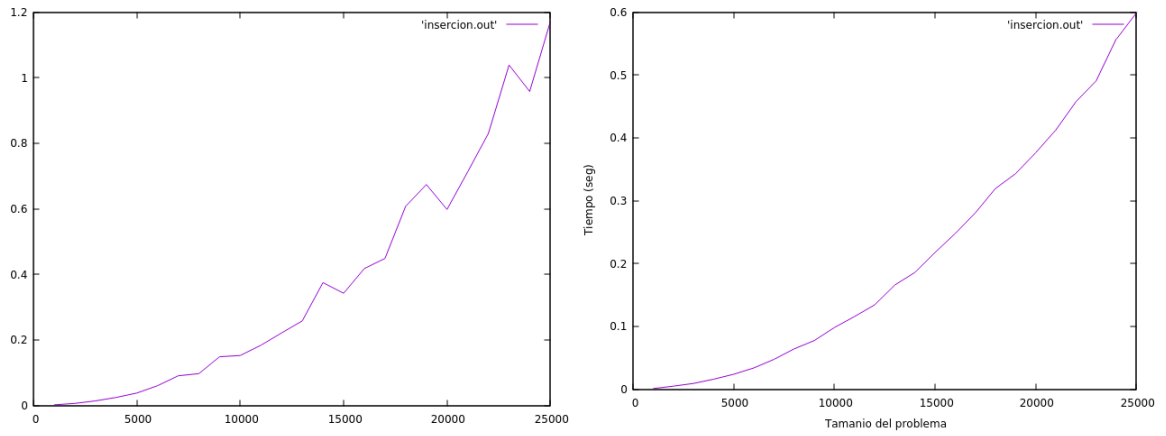
Cuadro 2: Comparación entre ambos entornos de ejecución

#### 3.4.1. Algoritmos con eficiencia $O(n^2)$



Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 11: Algoritmo burbuja

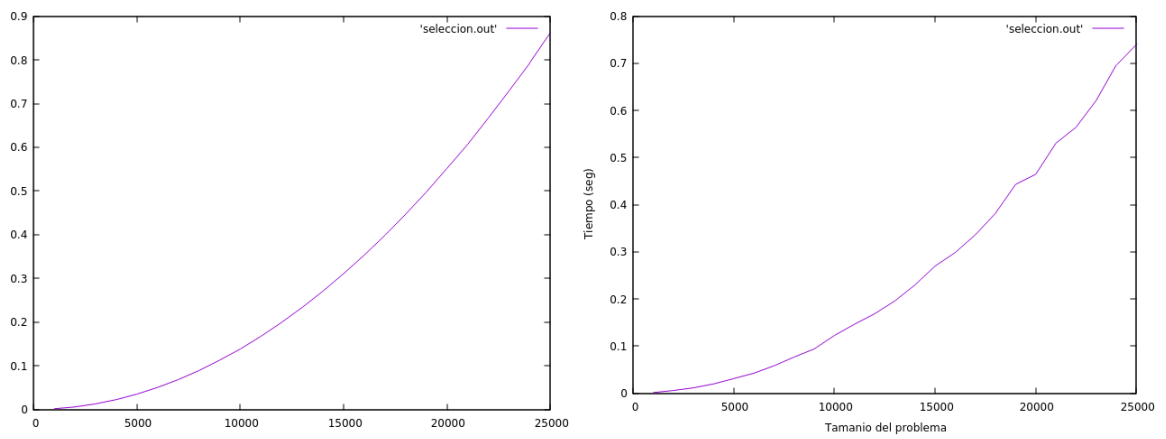


(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 12: Algoritmo de inserción



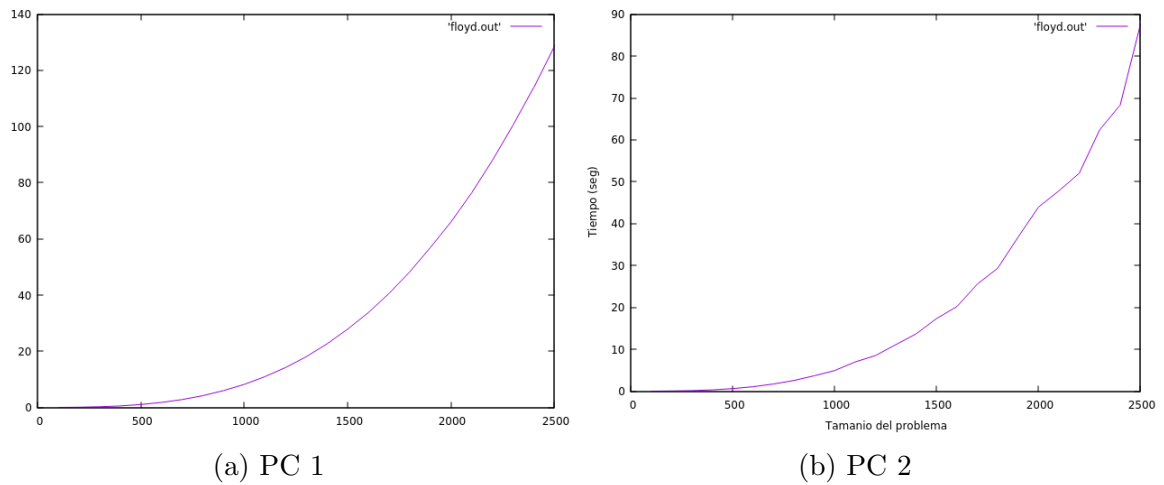
(a) PC 1

(b) PC 2

Figura 13: Algoritmo de selección



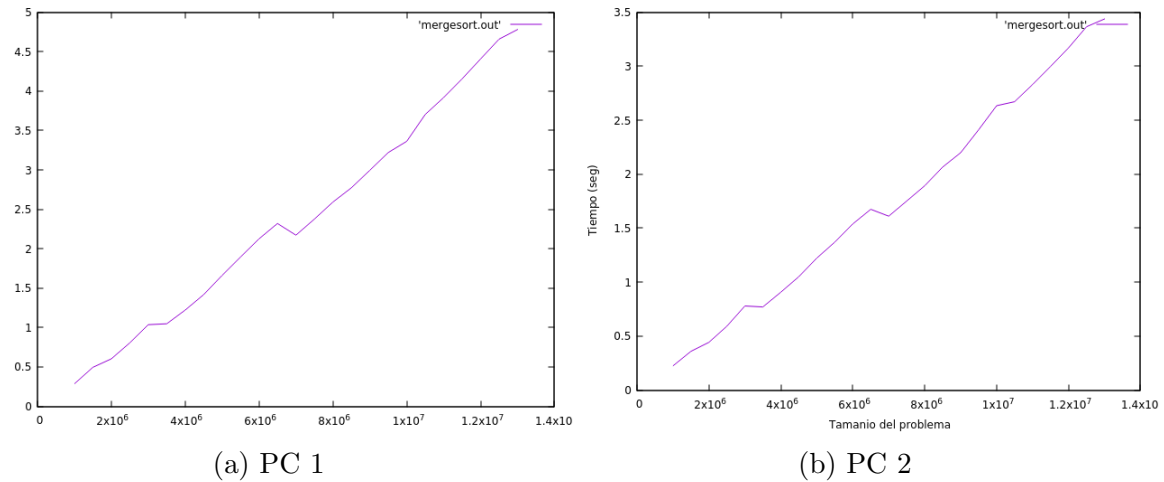
### 3.4.2. Algoritmos con eficiencia $O(n^3)$



Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

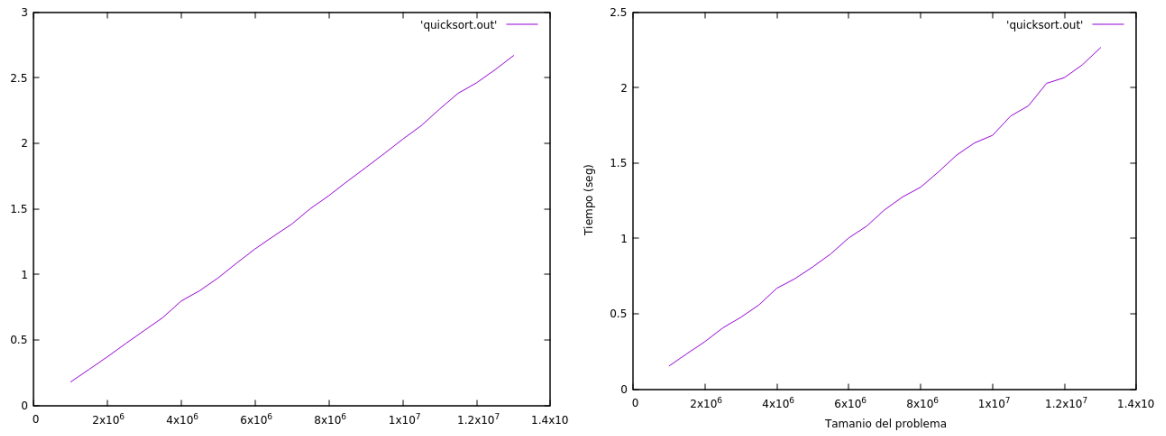
Figura 14: Algoritmo de Floyd

### 3.4.3. Algoritmos con eficiencia $O(n \cdot \log(n))$



Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 15: Algoritmo mergesort

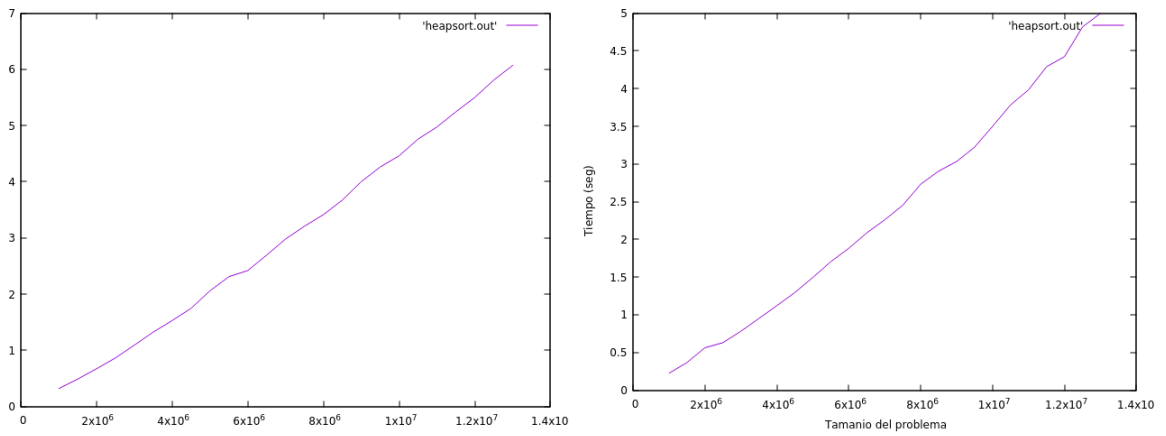


(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 16: Algoritmo quicksort



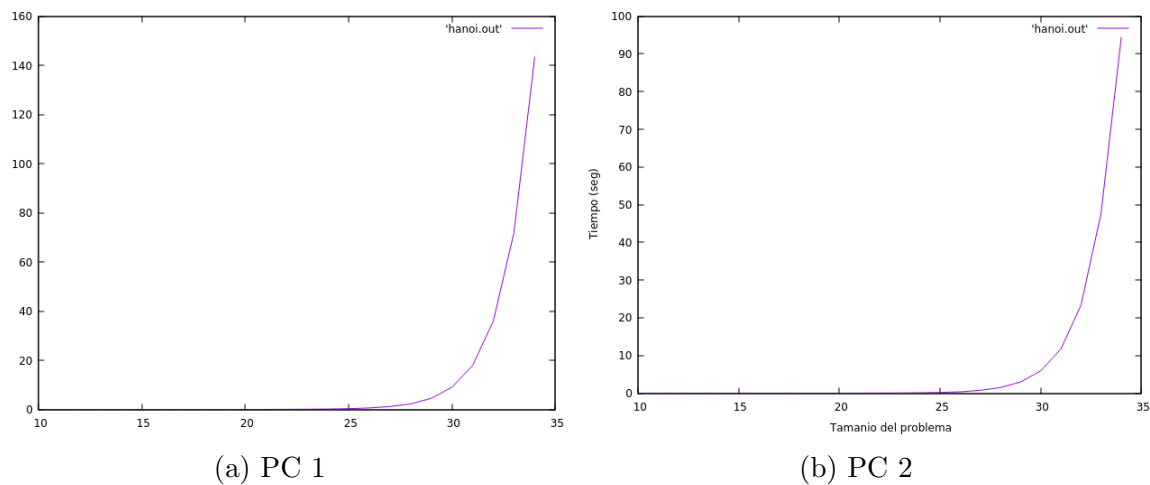
(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 17: Algoritmo heapsort

3.4.4. Algoritmo con eficiencia  $O(2^n)$



Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 18: Algoritmo de Hanoi

3.4.5. Algoritmos con eficiencia  $O(n \cdot \log(n))$

4. Cálculo de la eficiencia híbrida

- 4.0.1. Algoritmos con eficiencia  $O(n^2)$
- 4.0.2. Algoritmos con eficiencia  $O(n^3)$
- 4.0.3. Algoritmos con eficiencia  $O(n \cdot \log(n))$
- 4.0.4. Algoritmo con eficiencia  $O(2^n)$
- 4.0.5. Algoritmos con eficiencia  $O(n \cdot \log(n))$