



ugr

Universidad
de Granada

ALGORÍTMICA
GRADO EN INGENIERÍA INFORMÁTICA

Práctica 1

Análisis de eficiencia de algoritmos

Autores

María 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>	10
3. Cálculo de la eficiencia empírica	12
3.1. Scripts desarrollados	12
3.2. Entornos de ejecución	15
3.2.1. PC 1	15
3.2.2. PC 2	16
3.3. Gráficas comparativas	17
3.3.1. Algoritmos con eficiencia $O(n^2)$	17
3.3.2. Algoritmo con eficiencia $O(n^3)$	18
3.3.3. Algoritmos con eficiencia $O(n \cdot \log(n))$	19
3.3.4. Algoritmo con eficiencia $O(2^n)$	20
3.3.5. Comparación entre algoritmos de ordenación	20
3.4. Variación de la eficiencia empírica	22
3.4.1. Algoritmos con eficiencia $O(n^2)$	23
3.4.2. Algoritmos con eficiencia $O(n^3)$	26
3.4.3. Algoritmos con eficiencia $O(n \cdot \log(n))$	27
3.4.4. Algoritmo con eficiencia $O(2^n)$	30
4. Cálculo de la eficiencia híbrida	31
4.1. Algoritmos con eficiencia $O(n^2)$	31
4.2. Algoritmo con eficiencia $O(n^3)$	33
4.3. Algoritmos con eficiencia $O(n \cdot \log(n))$	33
4.4. Algoritmo con eficiencia $O(2^n)$	35
4.5. Ajuste erróneo	36

Índice de cuadros

1. Parámetros de ejecución de cada programa	15
2. Comparación de tiempos entre ambos entornos de ejecución	22
3. Bondad de los ajustes	31

Índice de figuras

1.	Algoritmo burbuja	17
2.	Algoritmo de inserción	17
3.	Algoritmo de selección	18
4.	Algoritmo de Floyd	18
5.	Algoritmo mergesort	19
6.	Algoritmo quicksort	19
7.	Algoritmo heapsort	20
8.	Algoritmo Hanoi	20
9.	Comparación de algoritmos de ordenación	21
10.	Comparación de algoritmos de ordenación (zoom)	21
11.	Algoritmo burbuja	23
12.	Algoritmo de inserción	24
13.	Algoritmo de selección	25
14.	Algoritmo de Floyd	26
15.	Algoritmo mergesort	27
16.	Algoritmo quicksort	28
17.	Algoritmo heapsort	29
18.	Algoritmo de Hanoi	30
19.	Algoritmo burbuja	31
20.	Algoritmo de inserción	32
21.	Algoritmo de selección	32
22.	Algoritmo de Floyd	33
23.	Algoritmo mergesort	33
24.	Algoritmo quicksort	34
25.	Algoritmo heapsort	34
26.	Algoritmo Hanoi	35
27.	Regresión errónea	36

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}.  $i \neq j$ .
11   @param j: número de columna a que
   ↳ se llevan los discos.
12          j es un valor de {1, 2,
   ↳ 3}.  $j \neq i$ .
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 reservar
101             ↪ memoria para un matriz de "
102             & << dim << " x " << dim <<
103             ↪ endl;
104             & for (int j = 0; j < i; j++)
105             & delete [] M[i];
106             & delete [] M;
107             & exit(1);
108             & }
109     }
110 }
```

```

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
98 void RellenaMatriz(int **M, int dim)
99 {
100     for (int i = 0; i < dim; i++)
101         for (int j = 0; j < dim; j++)
102             if (i != j)
103                 & M[i][j] = (rand() % 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 < dim; j++)
111     &             {
112     &                 & int sum = M[i][k] + M[k][j];
113     &                 ↪ &
114     &                 & M[i][j] = (M[i][j] > sum) ?
115     &                 ↪ sum : M[i][j];
116     &             }
117 } & &
118
119 int main (int argc, char **argv)
120 {
121     // clock_t tantes;
122     // clock_t tdespues;
123     int dim;
124
125     //Lectura de los parametros de
126     ↪ entrada
127     if (argc != 2)
128     {
129         cout << "Parámetros de entrada:
130         ↪ " << endl
131         & << "1.- Número de nodos" << endl
132         ↪ << endl;
133         return 1; &
134     } &
135
136     dim = atoi(argv[1]); &
137     int ** M = ReservaMatriz(dim);
138
139     RellenaMatriz(M,dim);
140
141     & & &
142     // Empieza el algoritmo de floyd
143     tantes = clock();
144     Floyd(M,dim);
145     tdespues = clock();
146     cout << dim <<
147     ((double)(tdespues-tantes))
148     /CLOCKS_PER_SEC << endl;
149     LiberaMatriz(M,dim);
150
151     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)) &&
55             ↳ (T[j] < T[j+1]))
56             & j++;
57         if (v >= T[j])
58             & esAPO = true;
59         T[k] = T[j];
60         k = j;
61     }
62     T[k] = v;
63 }
64
65 int main(int argc, char * argv[]){
66     if (argc != 2){
67         cerr << "Formato " << argv[0] <<
68             ↳ " <num_elem>" << endl;
69         return -1;
70     }
71     int n = atoi(argv[1]);
72     int * T = new int[n];
73     assert(T);
74     srandom(time(0));
75     for (int i = 0; i < n; i++){
76         T[i] = random();
77     }
78     clock_t tantes; // Valor del
79     ↳ reloj antes de la ejecución
80     clock_t tdespues; // Valor del
81     ↳ reloj después de la ejecución
82     tantes = clock();
83     heapsort(T, n);
84     tdespues = clock();
85     cout << n <<
86         ↳ ((double)(tdespues-tantes))
87         ↳ /CLOCKS_PER_SEC << endl;
88     delete [] T;
89     return 0;
90 }
```

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
70  * & & inicial < final.
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  static void fusion(int T[], int
83      ↪ inicial, int final, int U[], int
84      ↪ V[]);
85  /**
86      Implementación de las funciones
87  **/
88  inline static void insercion(int T[],
89      ↪ int num_elem){
90      insercion_lims(T, 0, num_elem);
91  }
92  static void insercion_lims(int T[],
93      ↪ int inicial, int final){
94      int i, j;
95      int aux;
96      for (i = inicial + 1; i < final;
97          ↪ i++) {
98          j = i;
99          while ((T[j] < T[j-1]) && (j > 0))
100              ↪ {
101              aux = T[j];
102              T[j] = T[j-1];
103              T[j-1] = aux;
104              j--;
105          }
106      };
107  }
108  const int UMBRAL_MS = 100;
109  void mergesort(int T[], int num_elem){
110      mergesort_lims(T, 0, num_elem);
111  }
112  static void mergesort_lims(int T[],
113      ↪ int inicial, int final){
114      if (final - inicial < UMBRAL_MS)
115          insercion_lims(T, inicial,
116              ↪ final);
117      else {
118          int k = (final - inicial)/2;
119          int * U = new int [k - inicial +
120              ↪ 1];
121          assert(U);
122          int l, l2;
123          for (l = 0, l2 = inicial; l < k;
124              ↪ l++, l2++)
125              & & U[l] = T[l2];
126          U[l] = INT_MAX;
127
128          int * V = new int [final - k +
129              ↪ 1];
130          assert(V);
131          for (l = 0, l2 = k; l < final -
132              ↪ k; l++, l2++)
133              & & V[l] = T[l2];
134          V[l] = INT_MAX;
135
136          mergesort_lims(U, 0, k);
137          mergesort_lims(V, 0, final - k);
138          fusion(T, inicial, final, U, V);
139          delete [] U;
140          delete [] V;
141      };
142  }
143
144  static void fusion(int T[], int
145      ↪ inicial, int final, int U[], int
146      ↪ V[]){
147      int j = 0;
148      int k = 0;
149      for (int i = inicial; i < final;
150          ↪ i++)
151          if (U[j] < V[k]) {
152              & & T[i] = U[j];j++;
153          } else{
154              & & T[i] = V[k];k++;
155          }
156      }
157
158  int main(int argc, char * argv[]){
159      if (argc != 2){
160          cerr << "Formato " << argv[0] <<
161              ↪ " <num_elem>" << endl;
162          return -1;
163      }
164      int n = atoi(argv[1]);
165      int * T = new int[n];
166      assert(T);
167      srand(time(0));
168      for (int i = 0; i < n; i++)
169          T[i] = random();
170
171      clock_t tantes; // Valor del
172      ↪ reloj antes de la ejecución
173      clock_t tdespues; // Valor del
174      ↪ reloj después de la ejecución
175      tantes = clock();
176      mergesort(T, n);
177      tdespues = clock();
178      cout << n <<
179          ↪ ((double)(tdespues-tantes))
180          ↪ /CLOCKS_PER_SEC << endl;
181      delete [] T;
182      return 0;
183  };

```

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 inicio 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 inicio 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 inicio 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

Por último, este script genera las gráficas mediante gnuplot:

```
1  #!/usr/bin/gnuplot
2  #Burbuja
3
4  set xlabel "Tamano del problema"
5  set ylabel "Tiempo (seg)"
6  set terminal png size 640,480
7  set output 'empirica_burbuja.png'
8  plot 'burbuja.out' with lines
9
10 #Floyd
11
12 set terminal png size 640,480
13 set output 'empirica_floyd.png'
14 plot 'floyd.out' with lines
15
16 #Hanoi
17
18 set terminal png size 640,480
19 set output 'empirica_hanoi.png'
20 plot 'hanoi.out' with lines
21
22 #Heapsort
23
24 set terminal png size 640,480
25 set output 'empirica_heapsort.png'
26 plot 'heapsort.out' with lines
27
28 #Insercion
29
30 set terminal png size 640,480
31 set output 'empirica_insercion.png'
32 plot 'insercion.out' with lines
33
34 #Mergesort
35
36 set terminal png size 640,480
37 set output 'empirica_mergesort.png'
38 plot 'mergesort.out' with lines
39
```

```
40 #Quicksort
41
42 set terminal png size 640,480
43 set output 'empirica_quicksort.png'
44 plot 'quicksort.out' with lines
45
46 #Selección
47
48 set terminal png size 640,480
49 set output 'empirica_seleccion.png'
50 plot 'seleccion.out' with lines
```

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

Algoritmo	Eficiencia	Tamaño inicial	Tamaño final	Incremento
Burbuja Inserción Selección	$O(n^2)$	1000	25000	1000
Mergesort Quicksort Heapsort	$O(n \cdot \log(n))$	1.000.000	13.000.000	500.000
Floyd	$O(n^3)$	100	2500	100
Hanoi	$O(2^n)$	10	34	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 $O(n^2)$

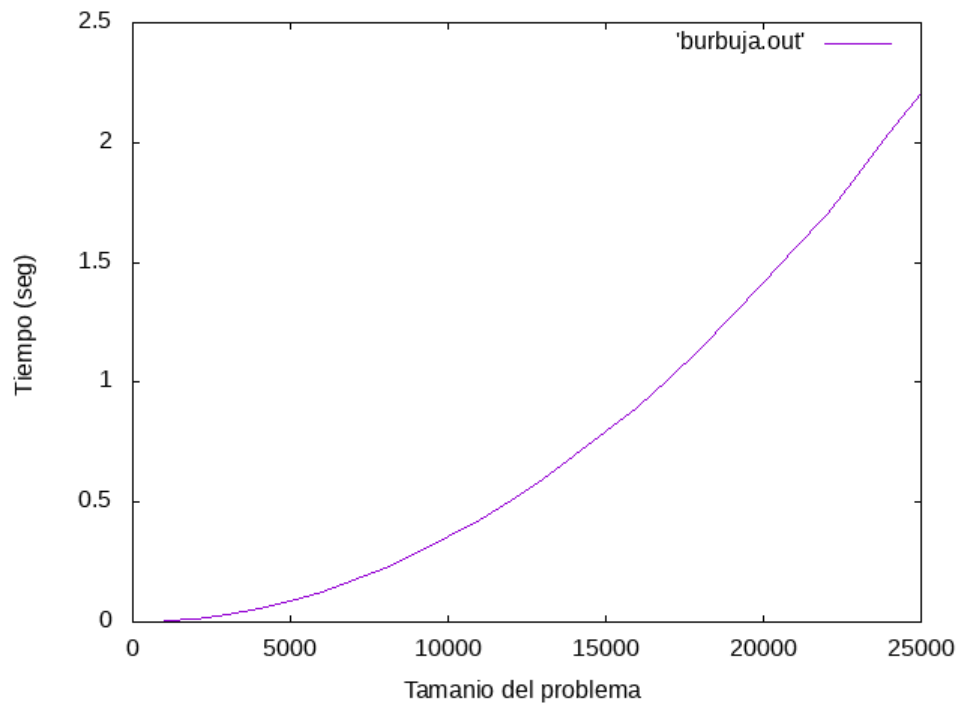


Figura 1: Algoritmo burbuja

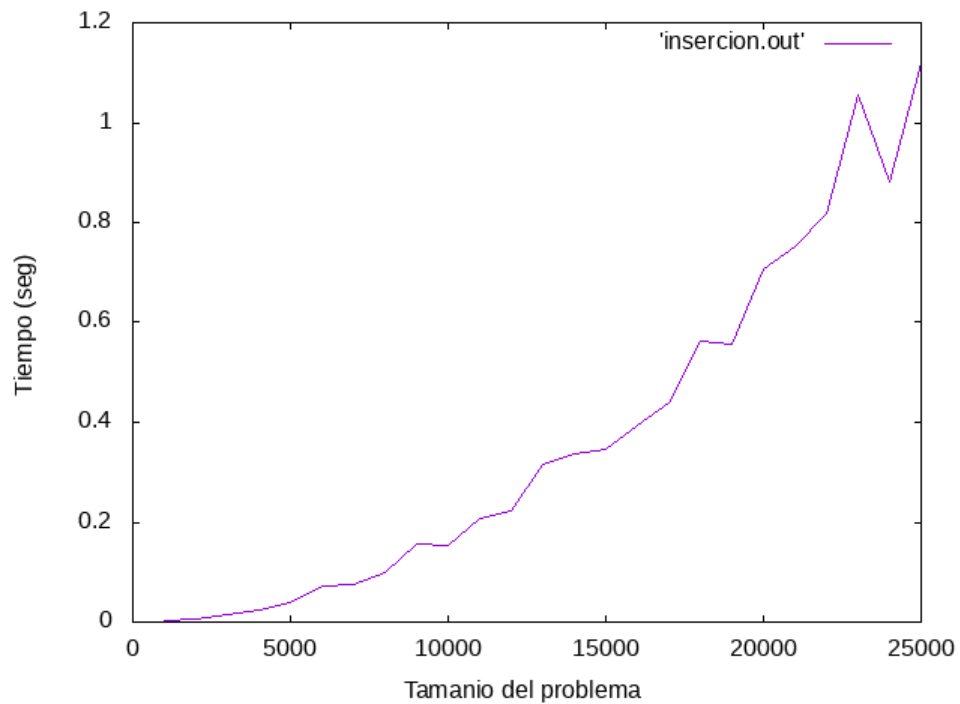


Figura 2: Algoritmo de inserción

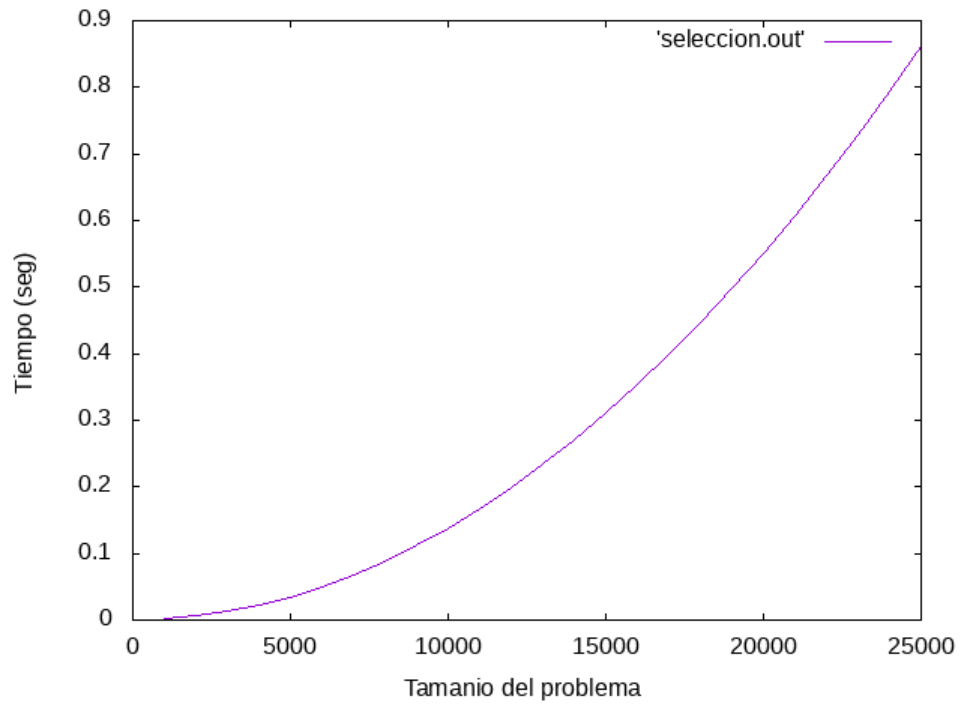


Figura 3: Algoritmo de selección

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

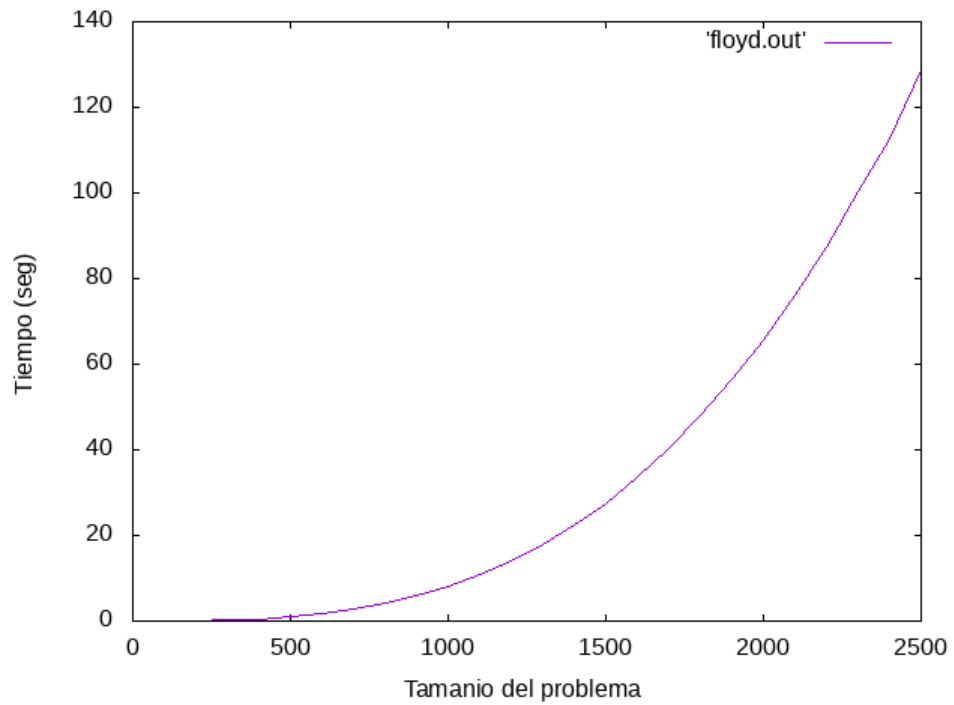


Figura 4: Algoritmo de Floyd

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

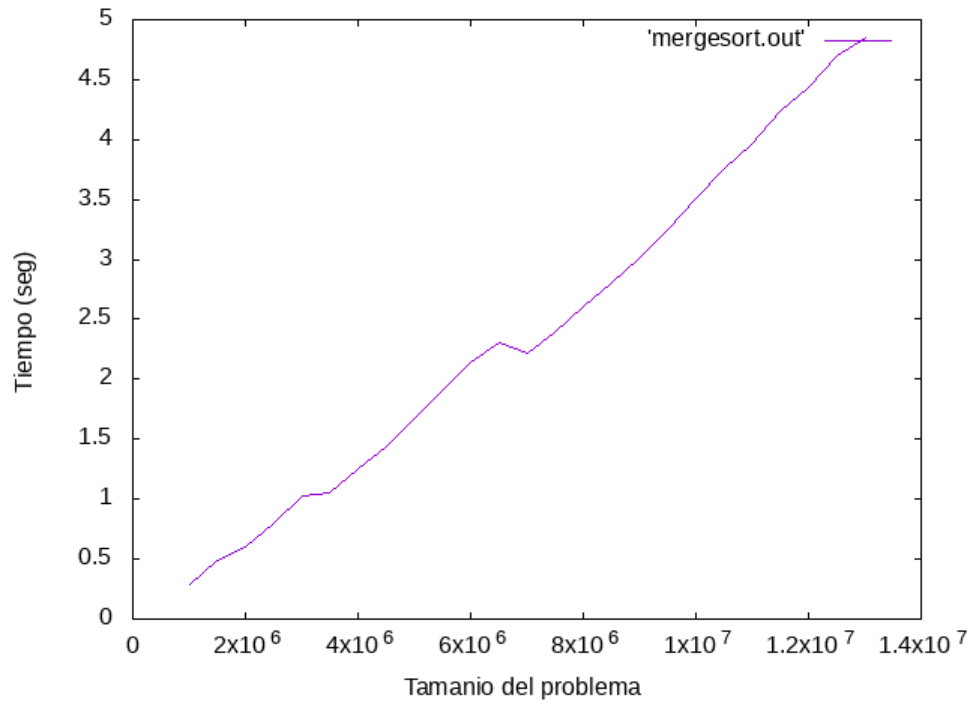


Figura 5: Algoritmo mergesort

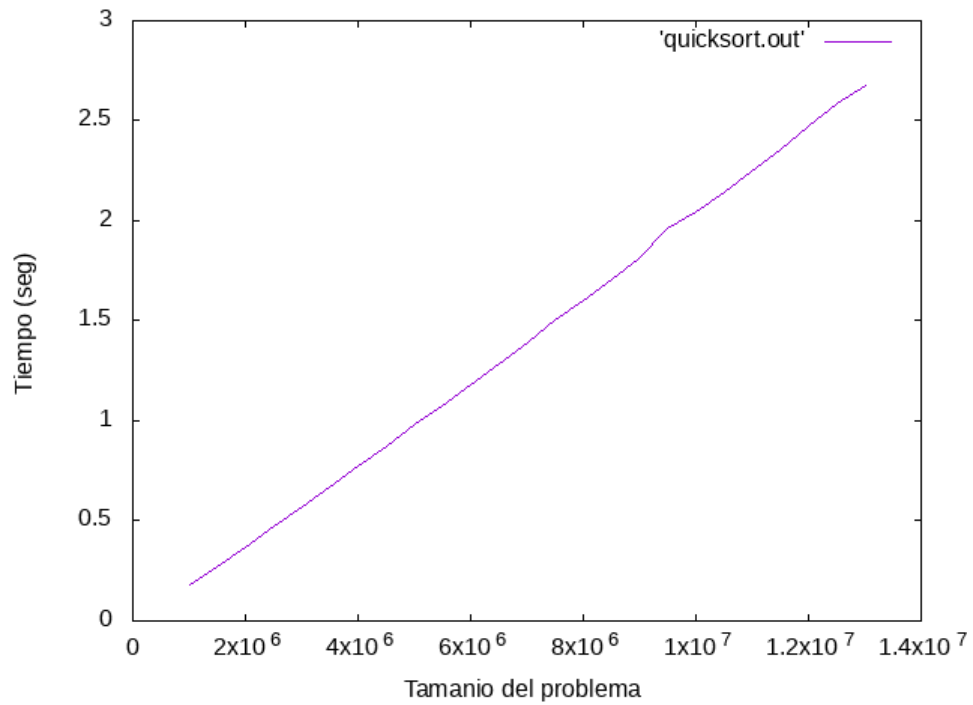


Figura 6: Algoritmo quicksort

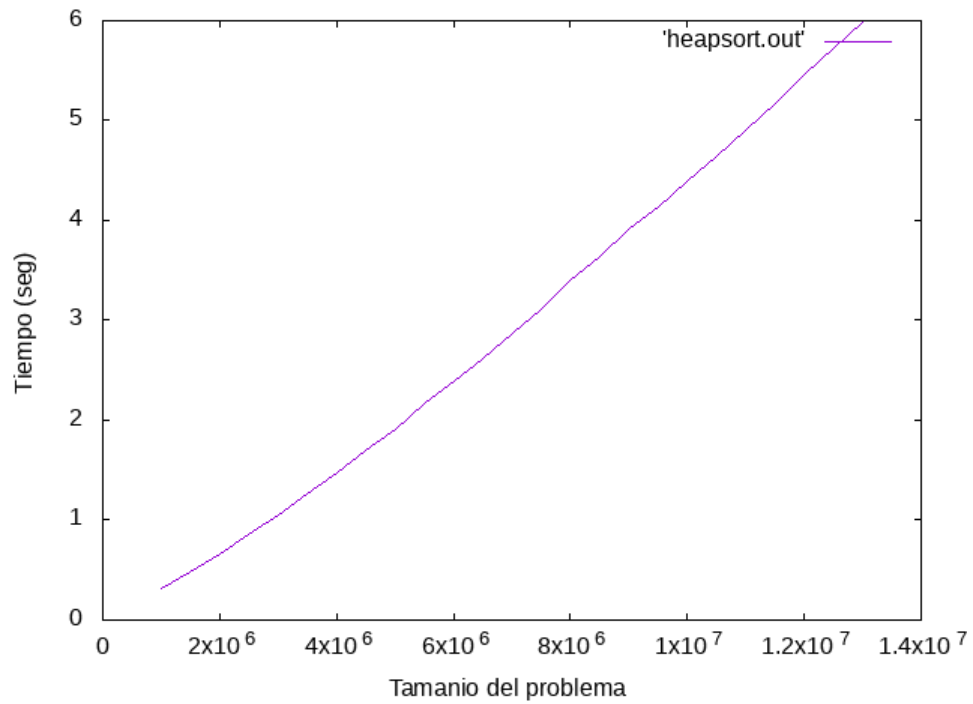


Figura 7: Algoritmo heapsort

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

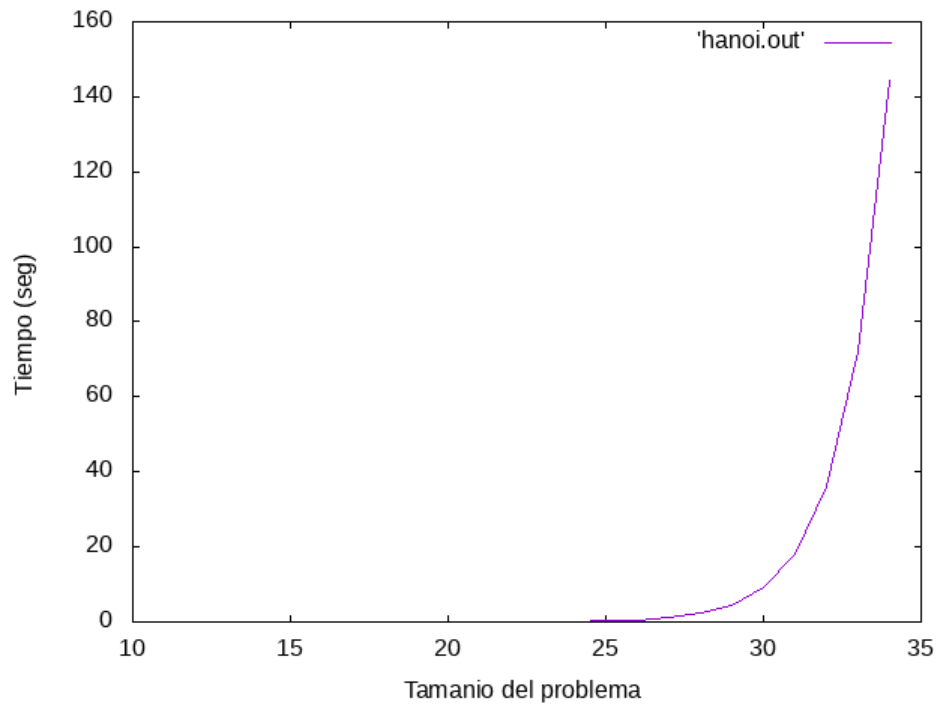


Figura 8: Algoritmo Hanoi

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.

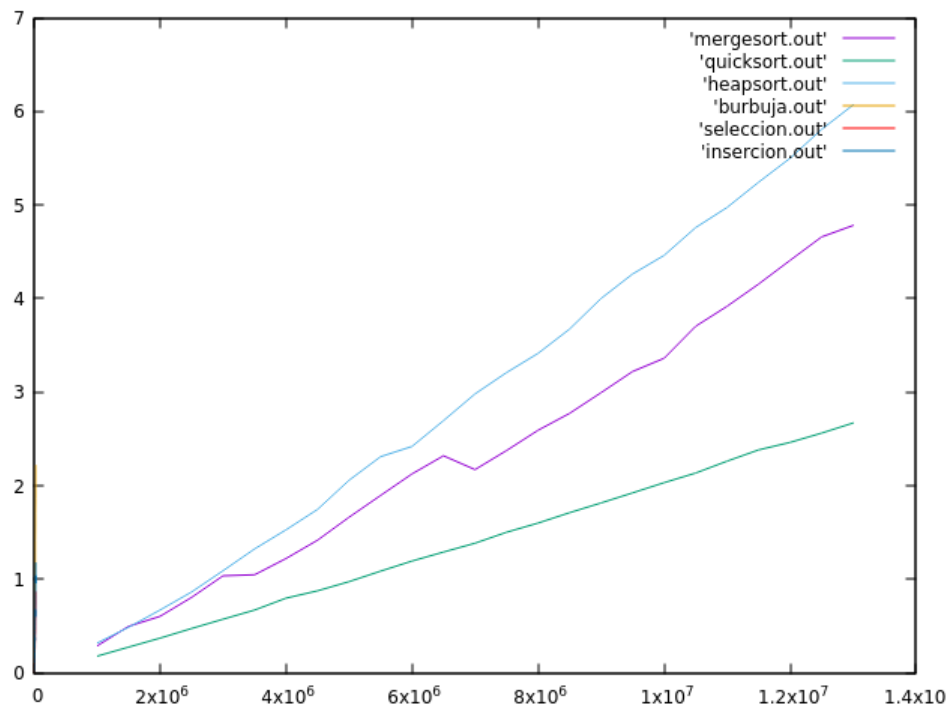


Figura 9: Comparación de algoritmos de ordenación

Si hacemos zoom, podremos ver mejor la diferencia:

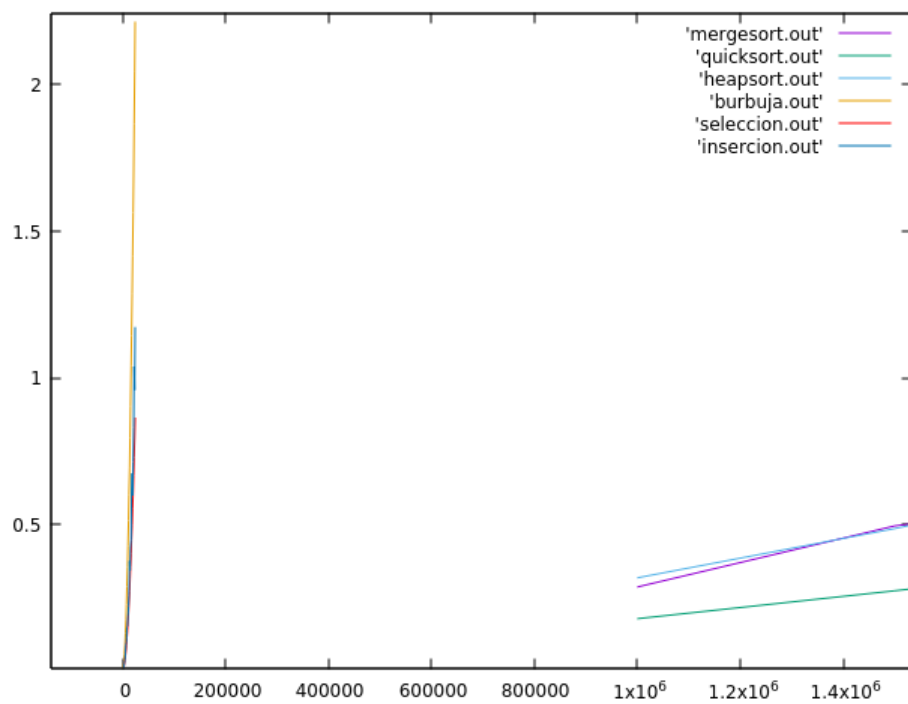


Figura 10: Comparación de algoritmos de ordenación (zoom)

3.4. Variación de la eficiencia empírica

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

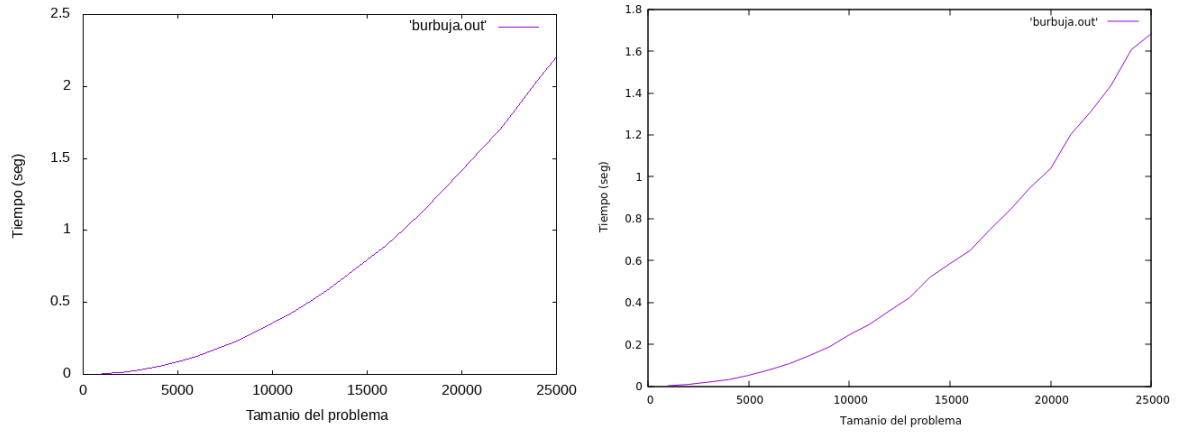
Principio 1 (de Invarianza) *Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia más que, a lo sumo, en una constante multiplicativa.*

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	0,366	0,251	1,456
Inserción	0,172	0,100	1,715
<i>Selección</i>	0,144	0,124	1,159
<i>Mergesort</i>	1,948	1,422	1,371
<i>Quicksort</i>	1,144	0,965	1,186
<i>Heapsort</i>	2,314	1,821	1,271
Floyd	8,636	5,348	1,615
Hanoi	0,036	0,023	1,538

Cuadro 2: Comparación de tiempos entre ambos entornos de ejecución

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

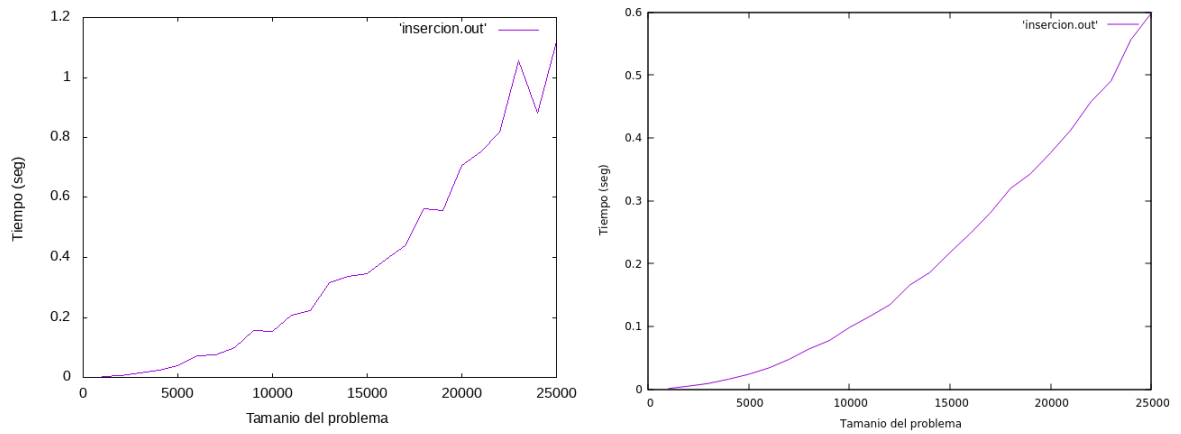


(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
1000	0.00347	0.002096
2000	0.014092	0.008327
3000	0.031775	0.019441
4000	0.056423	0.031218
5000	0.088117	0.051639
6000	0.127298	0.07696
7000	0.174768	0.106872
8000	0.227671	0.145037
9000	0.287853	0.187409
10000	0.355994	0.245409
11000	0.427654	0.295032
12000	0.509095	0.359974
13000	0.597057	0.423534
14000	0.693367	0.520076
15000	0.796559	0.584748
16000	0.898766	0.648439
17000	1.01542	0.749665
18000	1.14065	0.843255
19000	1.27787	0.950054
20000	1.41778	1.03928
21000	1.56071	1.20255
22000	1.70092	1.31179
23000	1.86982	1.4365
24000	2.04169	1.60665
25000	2.20676	1.68318

Figura 11: Algoritmo burbuja

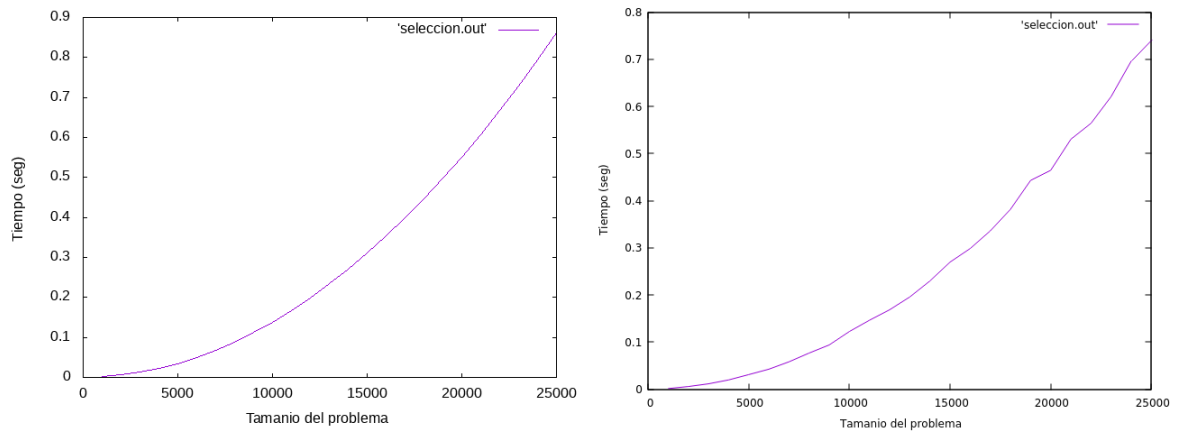


(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
1000	0,001566	0,000911
2000	0,006863	0,004746
3000	0,013775	0,009101
4000	0,024662	0,015753
5000	0,038397	0,023742
6000	0,073059	0,033688
7000	0,075398	0,047426
8000	0,097854	0,064051
9000	0,157555	0,077217
10000	0,15269	0,098011
11000	0,207507	0,115522
12000	0,221672	0,134134
13000	0,315095	0,165736
14000	0,337785	0,185779
15000	0,345627	0,217502
16000	0,393965	0,247678
17000	0,438195	0,280534
18000	0,563423	0,319248
19000	0,555218	0,343008
20000	0,707812	0,376268
21000	0,752603	0,41259
22000	0,817934	0,457614
23000	1,05414	0,490588
24000	0,881149	0,557605
25000	1,11894	0,599629

Figura 12: Algoritmo de inserción



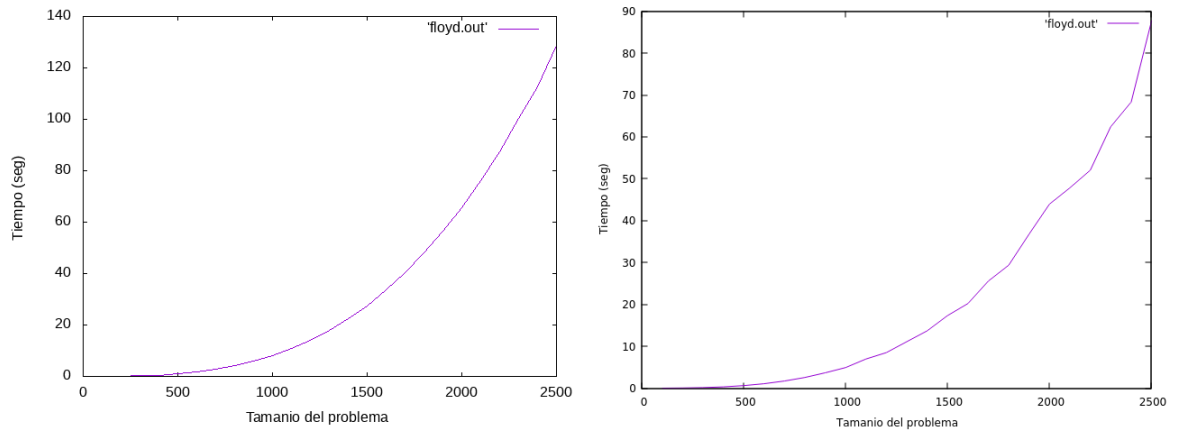
(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
1000	0,001469	0,001203
2000	0,005661	0,005539
3000	0,012639	0,011269
4000	0,022373	0,019565
5000	0,0348	0,030867
6000	0,050395	0,042368
7000	0,06795	0,057941
8000	0,088633	0,076493
9000	0,112056	0,093877
10000	0,138403	0,122324
11000	0,167213	0,146299
12000	0,198844	0,168477
13000	0,233619	0,195566
14000	0,270472	0,229449
15000	0,310389	0,269606
16000	0,353077	0,298446
17000	0,398466	0,336024
18000	0,447165	0,38157
19000	0,497554	0,442948
20000	0,551222	0,464486
21000	0,607534	0,530445
22000	0,667603	0,564229
23000	0,728555	0,620878
24000	0,793231	0,696617
25000	0,8607	0,740901

Figura 13: Algoritmo de selección

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



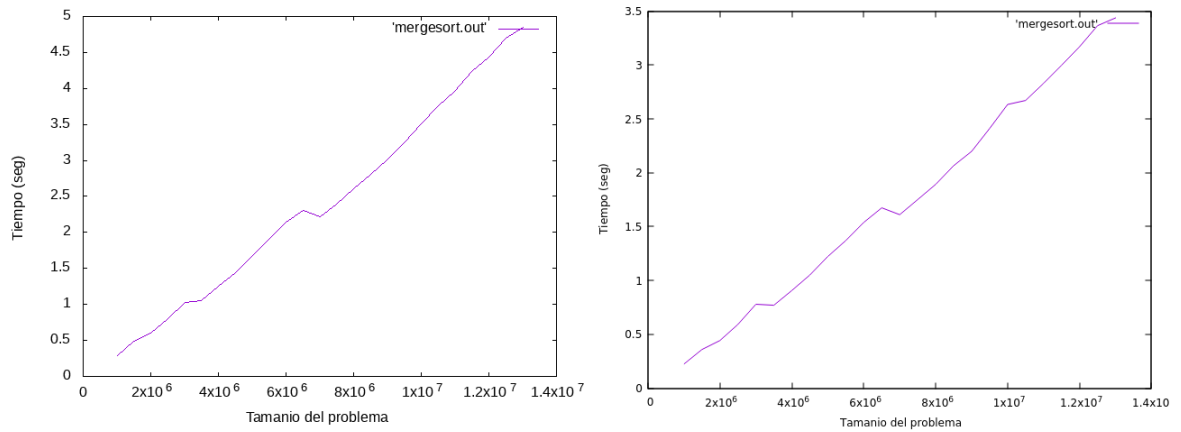
(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
100	0,008394	0,005113
200	0,066017	0,041535
300	0,220825	0,138818
400	0,521858	0,311018
500	1,01887	0,602487
600	1,75665	1,05512
700	2,80066	1,70498
800	4,17319	2,5442
900	5,95085	3,65579
1000	8,15234	4,91424
1100	10,8509	6,95897
1200	14,0868	8,47521
1300	17,9348	11,0851
1400	22,3625	13,6728
1500	27,532	17,3226
1600	33,5096	20,2078
1700	40,3014	25,5719
1800	47,9103	29,3598
1900	56,4474	36,773
2000	65,7778	43,9358
2100	76,1326	47,8074
2200	87,528	52,0403
2300	100,508	62,3899
2400	112,763	68,2729
2500	128,779	87,6574

Figura 14: Algoritmo de Floyd

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

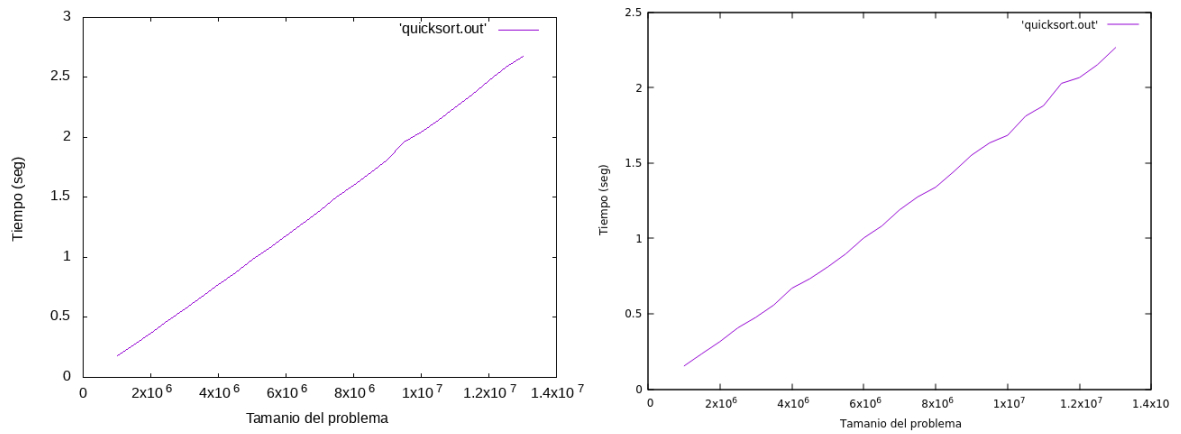


(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
1000000	0,284222d	0,224028
1500000	0,493673d	0,358624
2000000	0,597351d	0,442203
2500000	0,796508d	0,591356
3000000	1,03026d	0,778666
3500000	1,05322d	0,770087
4000000	1,25594d	0,905945
4500000	1,44694d	1,0499
5000000	1,6732d	1,22218
5500000	1,90594d	1,37029
6000000	2,14261d	1,53875
6500000	2,30352d	1,67414
7000000	2,21224d	1,61097
7500000	2,39566d	1,75056
8000000	2,60288d	1,89201
8500000	2,81016d	2,06572
9000000	3,00176d	2,19856
9500000	3,24843d	2,41008
10000000	3,50923d	2,63424
10500000	3,76182d	2,67053
11000000	3,95387d	2,82914
11500000	4,23099d	2,99723
12000000	4,43627d	3,16933
12500000	4,7013d	3,36511
13000000	4,84492d	3,43713

Figura 15: Algoritmo mergesort

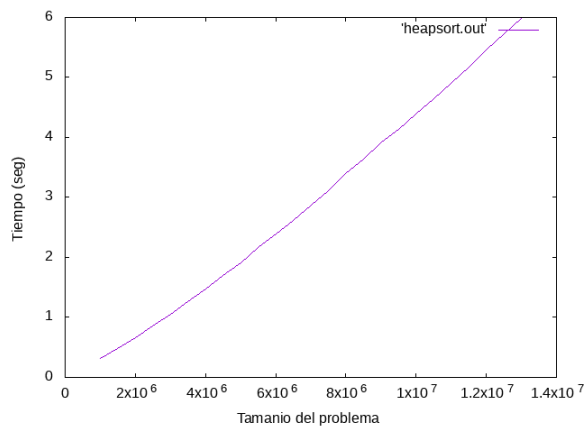


(a) PC 1

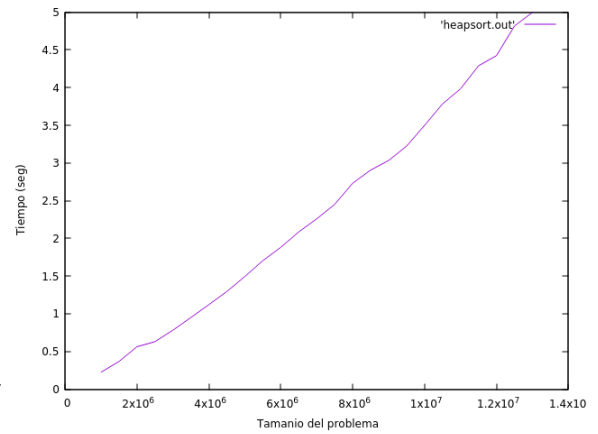
(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
1000000	0,179104	0,153138
1500000	0,272898	0,235349
2000000	0,372015	0,315383
2500000	0,474955	0,40695
3000000	0,57355	0,476882
3500000	0,671565	0,557988
4000000	0,772	0,668801
4500000	0,874961	0,732228
5000000	0,984274	0,810474
5500000	1,07864	0,897322
6000000	1,18198	1,00255
6500000	1,2892	1,08152
7000000	1,39126	1,19
7500000	1,50179	1,2749
8000000	1,60031	1,33935
8500000	1,70584	1,44181
9000000	1,81074	1,55247
9500000	1,95951	1,63309
10000000	2,04412	1,68327
10500000	2,14102	1,81085
11000000	2,25131	1,87967
11500000	2,35475	2,02783
12000000	2,47356	2,06599
12500000	2,58437	2,15211
13000000	2,67631	2,26403

Figura 16: Algoritmo quicksort



(a) PC 1

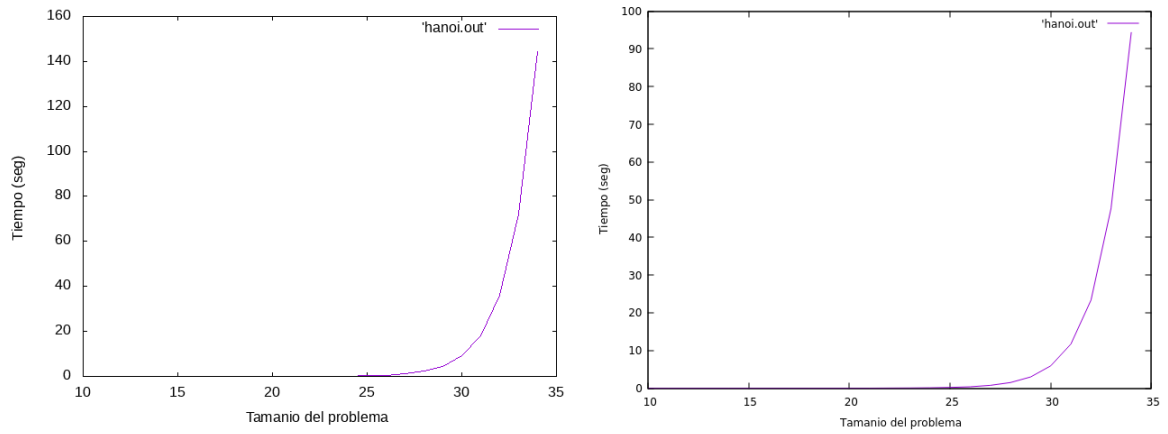


(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
1000000	0,313202d	0,223356
1500000	0,482248d	0,366004
2000000	0,664047d	0,561854
2500000	0,855399d	0,628741
3000000	1,05215d	0,781531
3500000	1,27027d	0,949929
4000000	1,47906d	1,1197
4500000	1,69867d	1,29438
5000000	1,91449d	1,49264
5500000	2,15878d	1,70317
6000000	2,38723d	1,88031
6500000	2,61876d	2,08545
7000000	2,86654d	2,25945
7500000	3,12018d	2,45073
8000000	3,39958d	2,73144
8500000	3,63556d	2,90442
9000000	3,90373d	3,03281
9500000	4,14154d	3,22381
10000000	4,39101d	3,50009
10500000	4,64831d	3,78361
11000000	4,90958d	3,98295
11500000	5,17252d	4,28962
12000000	5,45655d	4,42294
12500000	5,72861d	4,81687
13000000	5,99094d	4,99741

Figura 17: Algoritmo heapsort

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



(a) PC 1

(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
10	1,00E-05	7,00E-06
11	1,80E-05	1,20E-05
12	3,60E-05	2,50E-05
13	7,30E-05	4,50E-05
14	0,000145	8,90E-05
15	0,000288	0,000176
16	0,000587	0,000352
17	0,001131	0,000853
18	0,002277	0,001464
19	0,004491	0,002866
20	0,008952	0,00571
21	0,017783	0,011502
22	0,035699	0,022844
23	0,071711	0,045985
24	0,142219	0,092678
25	0,284257	0,181169
26	0,564716	0,364107
27	1,13561	0,741404
28	2,25519	1,49178
29	4,51148	2,96788
30	9,03096	5,9088
31	18,0955	11,7069
32	36,0193	23,3482
33	72,2035	47,761
34	144,474	94,3148

Figura 18: Algoritmo de Hanoi

4. Cálculo de la eficiencia híbrida

Para realizar este cálculo, realizamos una regresión de la expresión de eficiencia teórica con respecto a los datos empíricos obtenidos.

Si nuestros datos son correctos, el porcentaje de error será muy bajo.

Algoritmo	Orden de eficiencia	Porcentaje de error
Burbuja Selección Inserción	n^2	2.253e-12 (0.06377 %) 3.047e-13 (0.02211 %) 3.085e-11 (1.805 %)
Heapsort Mergesort Quicksort	$n \cdot \log(n)$	2.071e-10 (0.7626 %) 1.893e-10 (0.8614 %) 1.407e-11 (0.1113 %)
Hanoi	2^n	1.095e-12 (0.01302 %)
Floyd	n^3	7.291e-12 (0.08874 %)
Ajuste erróneo	2^n a n^2	0.00868(26.86 %)

Cuadro 3: Bondad de los ajustes

4.1. Algoritmos con eficiencia $O(n^2)$

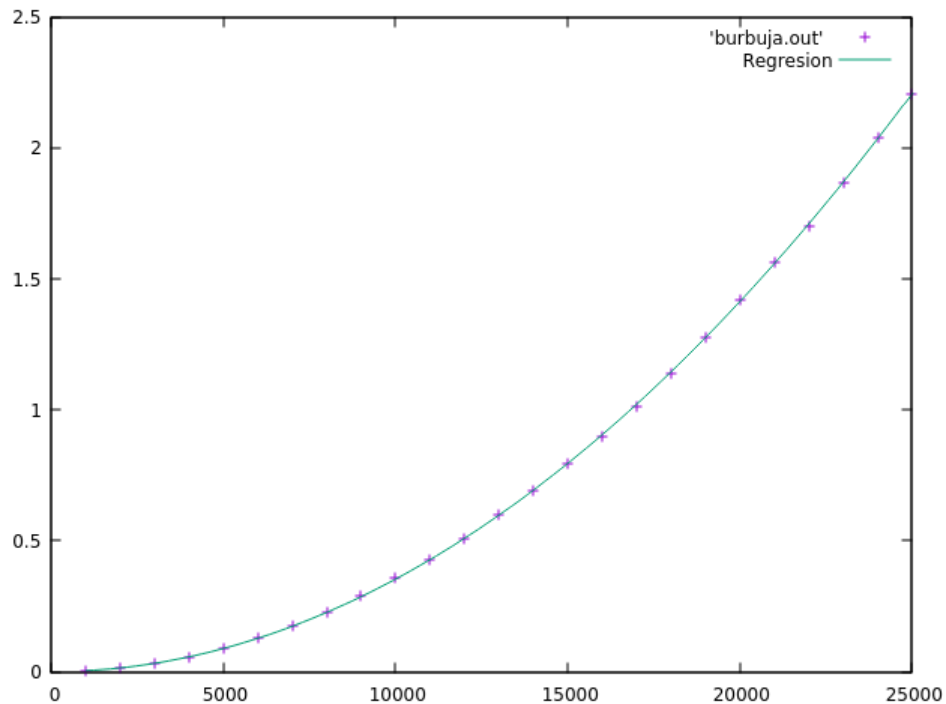


Figura 19: Algoritmo burbuja

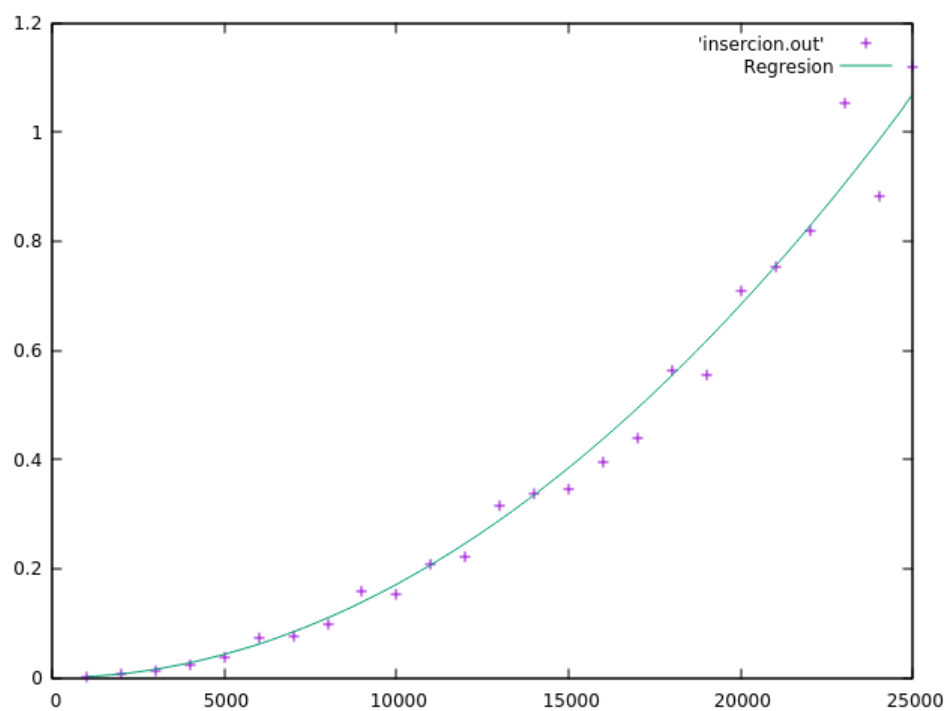


Figura 20: Algoritmo de inserción

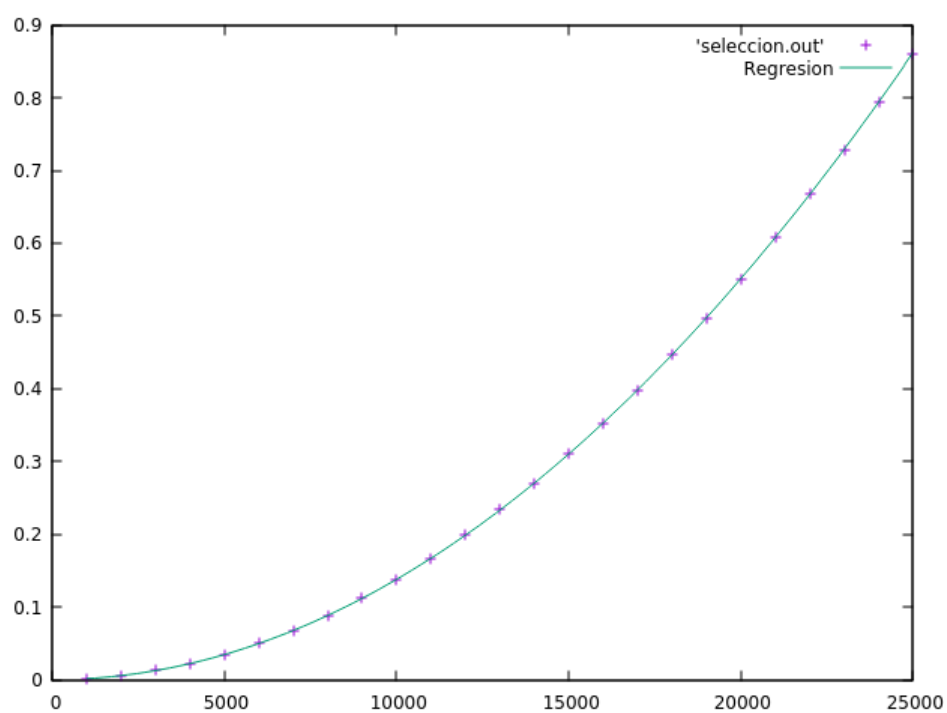


Figura 21: Algoritmo de selección

4.2. Algoritmo con eficiencia $O(n^3)$

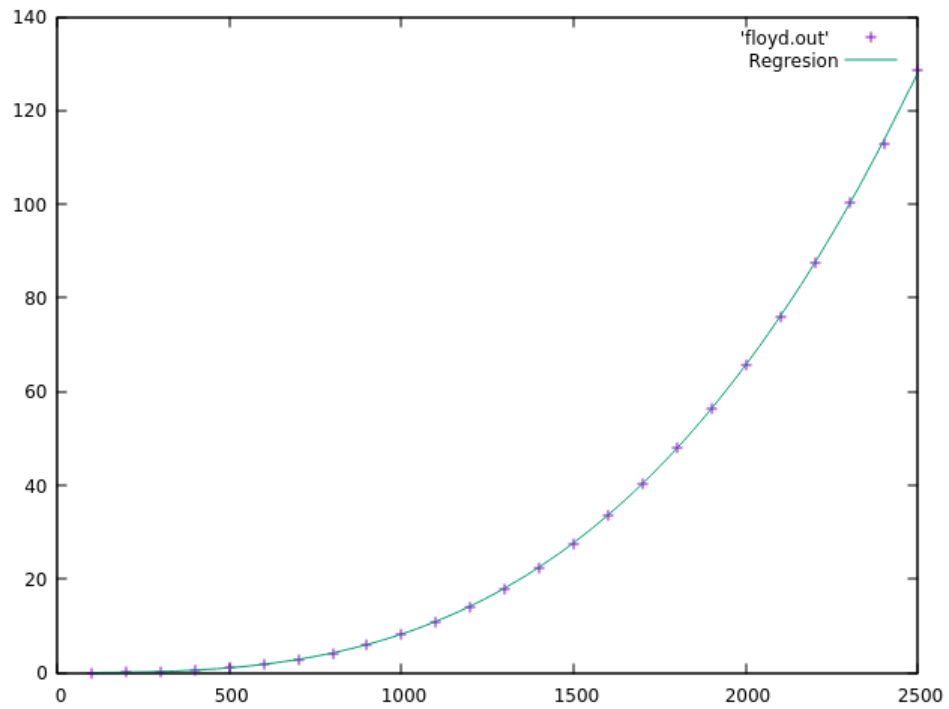


Figura 22: Algoritmo de Floyd

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

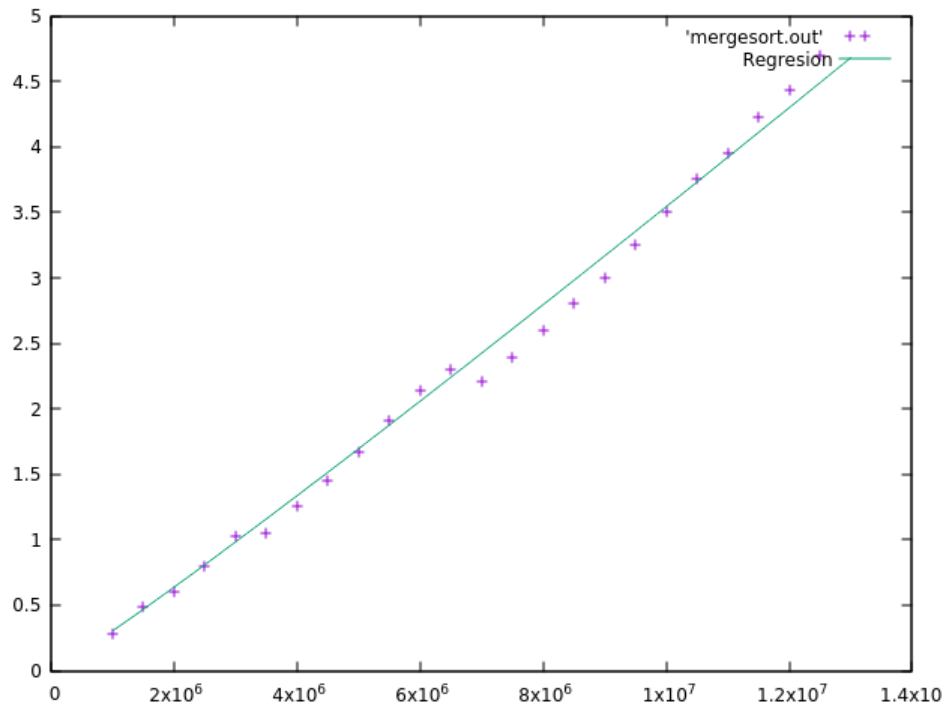


Figura 23: Algoritmo mergesort

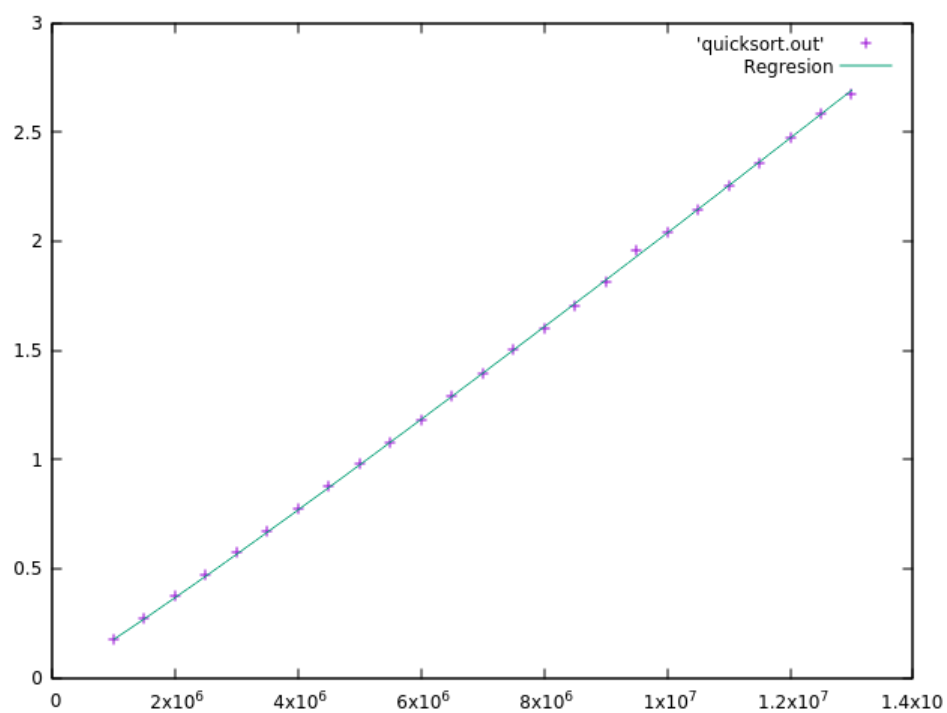


Figura 24: Algoritmo quicksort

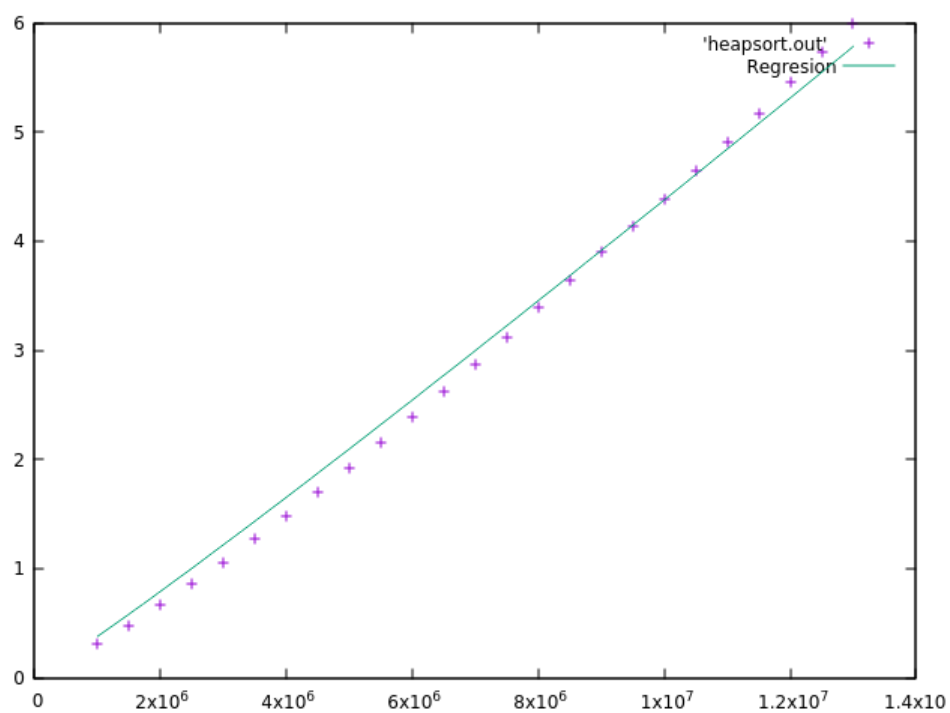


Figura 25: Algoritmo heapsort

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

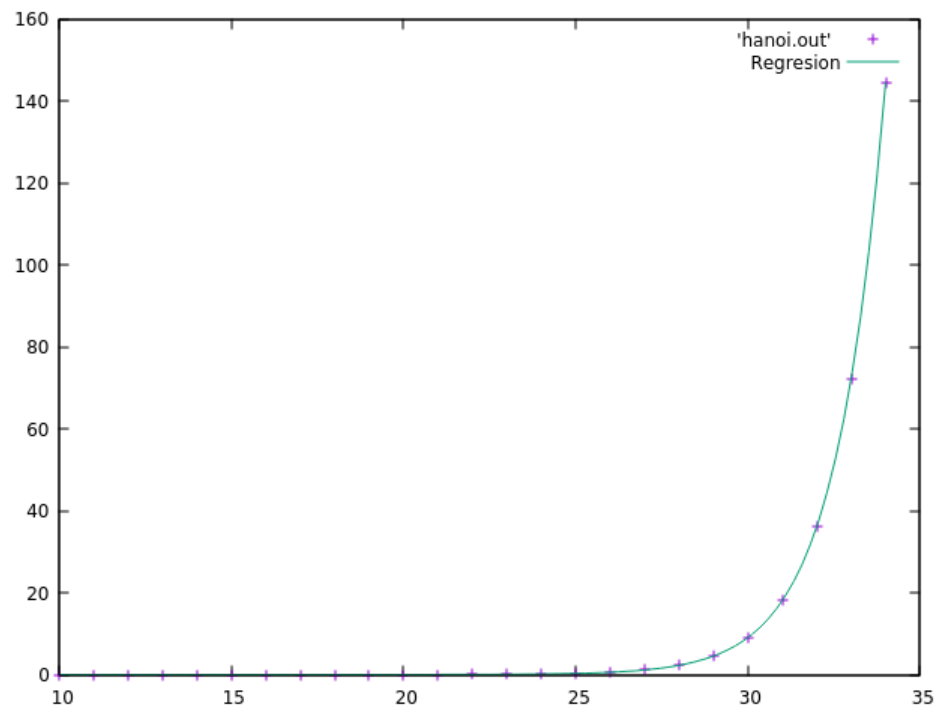


Figura 26: Algoritmo Hanoi

4.5. Ajuste erróneo

Aquí ajustamos los datos del Algoritmo de Hanoi ($O(2^n)$) a una eficiencia teórica cuadrática ($O(n^2)$). Vemos que además de obtener muchísimo error, la gráfica no es consistente.

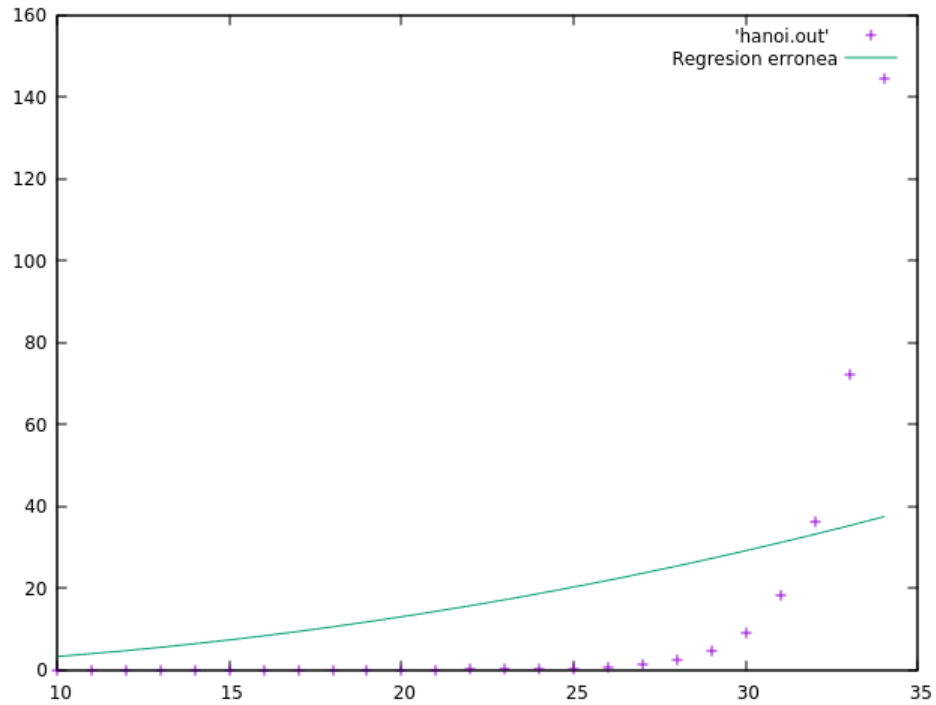


Figura 27: Regresión errónea