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

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2017-2018

Índice

Índice de cuadros

Índice de figuras

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

```
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>

/**
   @brief Resuelve el problema de las Torres de Hanoi
   @param M: número de discos. M > 1.
   @param i: número de columna en que están los discos.
             i es un valor de {1, 2, 3}. i != j.
   @param j: número de columna a que se llevan los discos.
             j es un valor de {1, 2, 3}. j != i.

   Esta función imprime en la salida estándar la secuencia de
   movimientos necesarios para desplazar los M discos de la
   columna i a la j, observando la restricción de que ningún
   disco se puede situar sobre otro de tamaño menor. Utiliza
   una única columna auxiliar.
*/
void hanoi (int M, int i, int j);

void hanoi (int M, int i, int j)
{
  if (M > 0)
    {
      hanoi(M-1, i, 6-i-j);
      cout << i << " -> " << j << endl;
      hanoi (M-1, 6-i-j, j);
    }
}

int main(int argc, char * argv[])
{

    if (argc != 2)
    {
      cerr << "Formato " << argv[0] << " <num_discos>" << endl;
      return -1;
    }

  int M = atoi(argv[1]);

  clock_t tantes;    // Valor del reloj antes de la ejecución
  clock_t tdespues;  // Valor del reloj después de la ejecución
  tantes = clock();
  hanoi(M, 1, 2);
  tdespues = clock();
  cout << M <<
  ((double)(tdespues-tantes))
  /CLOCKS_PER_SEC << endl;
  return 0;
}
```

2.2. Floyd

```cpp
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>
#include <cmath>

static int const MAX_LONG  = 10;

/**
   @brief Reserva espacio en memoria dinámica para una matriz cuadrada.

   @param dim: dimensión de la matriz. dim > 0.

   @returns puntero a la zona de memoria reservada.
*/
int ** ReservaMatriz(int dim);

/**
   @brief Libera el espacio asignado a una matriz cuadrada.

   @param M: puntero a la zona de memoria reservada. Es MODIFICADO.
   @param dim: dimensión de la matriz. dim > 0.

   Liberar la zona memoria asignada a M y lo pone a NULL.
*/
void LiberaMatriz(int ** & M, int dim);

/**
   @brief Rellena una matriz cuadrada con valores aleaotorias.

   @param M: puntero a la zona de memoria reservada. Es MODIFICADO.
   @param dim: dimensión de la matriz. dim > 0.

   Asigna un valor aleatorio entero de [0, MAX_LONG - 1] a cada
   elemento de la matriz M, salvo los de la diagonal principal
   que quedan a 0..
*/
void RellenaMatriz(int **M, int dim);


/**
   @brief Cálculo de caminos mínimos.

   @param M: Matriz de longitudes de los caminos. Es MODIFICADO.
   @param dim: dimensión de la matriz. dim > 0.

   Calcula la longitud del camino mínimo entre cada par de nodos (i,j),
   que se almacena en M[i][j].
*/
void Floyd(int **M, int dim);


/**
   Implementación de las funciones
**/

int ** ReservaMatriz(int dim)
{
  int **M;
  if (dim  <= 0)
    {
      cerr<< "\n ERROR: Dimension de la matriz debe ser mayor que 0" << endl;
      exit(1);
    }
  M = new int * [dim];
  if (M == NULL)
    {
      cerr << "\n ERROR: No puedo reservar memoria para un matriz de "
 &    << dim << " x " << dim << "elementos" << endl;
      exit(1);
    }
  for (int i = 0; i < dim; i++)
    {
      M[i]= new int [dim];
      if (M[i] == NULL)
 & {
 &   cerr << "ERROR: No puedo reservar memoria para un matriz de "
 &        << dim << " x " << dim << endl;
 &   for (int j = 0; j < i; j++)
 &     delete [] M[i];
 &   delete [] M;
 &   exit(1);
 & }
    }
```

```
  return M;
} &

void LiberaMatriz(int ** & M, int dim)
{
  for (int i = 0; i < dim; i++)
    delete [] M[i];
  delete [] M;
  M = NULL;
} &  &


void RellenaMatriz(int **M, int dim)
{
  for (int i = 0; i < dim; i++)
    for (int j = 0; j < dim; j++)
      if (i != j)
 & M[i][j]= (rand() \% MAX_LONG);
} &  &  &
 &
void Floyd(int **M, int dim)
{
 & for (int k = 0; k < dim; k++)
 &   for (int i = 0; i < dim;i++)
 &     for (int j = 0; j < dim;j++)
 &       {
 &  & int sum = M[i][k] + M[k][j];
 ↪  &
 &      & M[i][j] = (M[i][j] > sum) ?
 ↪  sum : M[i][j];
 &       }
} &       &

int main (int argc, char **argv)
{
//  clock_t tantes;
//  clock_t tdespues;
    int dim;

  //Lectura de los parametros de
  ↪  entrada
  if (argc != 2)
    {
      cout << "Parámetros de entrada:
      ↪  " << endl
 &    << "1.- Número de nodos" << endl
 ↪  << endl;
      return 1; &
    } &

  dim = atoi(argv[1]); &
  int ** M = ReservaMatriz(dim);

  RellenaMatriz(M,dim);
 &  &  &  &
 // Empieza el algoritmo de floyd
  tantes = clock();
  Floyd(M,dim);
  tdespues = clock();
  cout << dim  <<
  ((double)(tdespues-tantes))
  /CLOCKS_PER_SEC << endl;
  LiberaMatriz(M,dim);

  return 0; &
```

2.3. Algoritmos de ordenación

2.3.1. Burbuja

```cpp
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

/**
   @brief Ordena un vector por el método de la burbuja.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo de la burbuja.
*/
inline static
void burbuja(int T[], int num_elem);

/**
   @brief Ordena parte de un vector por el método de la burbuja.

   @param T: vector de elementos. Tiene un número de elementos
                    mayor o igual a final.Es MODIFICADO.

   @param inicial: Posición que marca el incio de la parte del
                    vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                    vector a ordenar.
    &&    inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de la burbuja.
*/
static void burbuja_lims(int T[], int inicial, int final);

/**
   Implementación de las funciones
**/

inline void burbuja(int T[], int num_elem)
{
  burbuja_lims(T, 0, num_elem);
}

static void burbuja_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial; i < final - 1; i++)
    for (j = final - 1; j > i; j--)
      if (T[j] < T[j-1])
 & {
 &   aux = T[j];
 &   T[j] = T[j-1];
 &   T[j-1] = aux;
 & }
}

int main(int argc, char * argv[]){
    if (argc != 2){
      cerr << "Formato " << argv[0] << " <num_elem>" << endl;
      return -1;
    }
  int n = atoi(argv[1]);
  int * T = new int[n];
  assert(T);
  srandom(time(0));

  for (int i = 0; i < n; i++)
      T[i] = random();

  clock_t tantes;
  clock_t tdespues;
  tantes = clock();
  burbuja(T, n);
  tdespues = clock();
  cout << n  <<
  ((double)(tdespues-tantes))
  /CLOCKS_PER_SEC << endl;

  delete [] T;

  return 0;
}
```

2.3.2. Selección

```cpp
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

/**
   @brief Ordena un vector por el método de selección.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor. Aplica el algoritmo de selección.
*/
inline static
void seleccion(int T[], int num_elem);

/**
   @brief Ordena parte de un vector por el método de selección.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                  vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                  vector a ordenar.
    && inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1de forma que los dispone en sentido creciente
   de menor a mayor. Aplica el algoritmo de selección.
*/
static void seleccion_lims(int T[], int inicial, int final);
/**
   Implementación de las funciones
**/

void seleccion(int T[], int num_elem){
  seleccion_lims(T, 0, num_elem);
}

static void seleccion_lims(int T[], int inicial, int final){
  int i, j, indice_menor;
  int menor, aux;
  for (i = inicial; i < final - 1; i++) {
    indice_menor = i;
    menor = T[i];
    for (j = i; j < final; j++)
      if (T[j] < menor) {
 & indice_menor = j;
 & menor = T[j];
      }
    aux = T[i];
    T[i] = T[indice_menor];
    T[indice_menor] = aux;
  }
}

int main(int argc, char * argv[]){
  if (argc != 2){
      cerr << "Formato " << argv[0] << " <num_elem>" << endl;
      return -1;
   }
  int n = atoi(argv[1]);
  int * T = new int[n];
  assert(T);
  srandom(time(0));

  for (int i = 0; i < n; i++)
      T[i] = random();

  clock_t tantes;    // Valor del reloj antes de la ejecución
  clock_t tdespues;  // Valor del reloj después de la ejecución
  tantes = clock();
  seleccion(T, n);
  tdespues = clock();
  cout << n << ((double)(tdespues-tantes))/CLOCKS_PER_SEC << endl;

  delete [] T;
  return 0;
}
```

2.3.3. Inserción

```cpp
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

/**
   @brief Ordena un vector por el método de inserción.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo de inserción.
*/
inline static
void insercion(int T[], int num_elem);

/**
   @brief Ordena parte de un vector por el método de inserción.

   @param T: vector de elementos. Tiene un número de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                   vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                   vector a ordenar.
    && inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de inserción.
*/
static void insercion_lims(int T[], int inicial, int final);

/**
   Implementación de las funciones
**/


inline static void insercion(int T[], int num_elem){
  insercion_lims(T, 0, num_elem);
}

static void insercion_lims(int T[], int inicial, int final){
  int i, j;
  int aux;
  for (i = inicial + 1; i < final; i++) {
    j = i;
    while ((T[j] < T[j-1]) && (j > 0)) {
      aux = T[j];
      T[j] = T[j-1];
      T[j-1] = aux;
      j--;
    };
  };
}

int main(int argc, char * argv[]){
    if (argc != 2){
      cerr << "Formato " << argv[0] << " <num_elem>" << endl;
      return -1;
    }
  int n = atoi(argv[1]);

  int * T = new int[n];
  assert(T);

  srandom(time(0));

  for (int i = 0; i < n; i++)
      T[i] = random();

  clock_t tantes;    // Valor del reloj antes de la ejecución
  clock_t tdespues;  // Valor del reloj después de la ejecución
  tantes = clock();
  insercion(T, n);
  tdespues = clock();
  cout << n << ((double)(tdespues-tantes))/CLOCKS_PER_SEC << endl;
  delete [] T;

  return 0;
};
```

2.3.4. *Heapsort*

```cpp
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>

/**
   @brief Ordena un vector por el método de montones.

   @param T: vector de elementos. Debe tener num_elem elementos.Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   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.
*/
inline static
void heapsort(int T[], int num_elem);

/**
   @brief Reajusta parte de un vector para que sea un montón.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.
   @param k: índice del elemento que se toma com raíz

   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.
*/
static void reajustar(int T[], int num_elem, int k);

/**Implementación de las funciones**/

static void heapsort(int T[], int num_elem){
  int i;
  for (i = num_elem/2; i >= 0; i--)
    reajustar(T, num_elem, i);
  for (i = num_elem - 1; i >= 1; i--){
      int aux = T[0];
      T[0] = T[i];
      T[i] = aux;
      reajustar(T, i, 0);
    }
}

static void reajustar(int T[], int num_elem, int k){
  int j;
  int v;
  v = T[k];
  bool esAPO = false;
  while ((k < num_elem/2) && !esAPO)
    {
      j = k + k + 1;
      if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
 & j++;
      if (v >= T[j])
 & esAPO = true;
      T[k] = T[j];
      k = j;
    }
  T[k] = v;
}

int main(int argc, char * argv[]){
  if (argc != 2){
      cerr << "Formato " << argv[0] << " <num_elem>" << endl;
      return -1;
   }
  int n = atoi(argv[1]);
  int * T = new int[n];
  assert(T);
  srandom(time(0));
  for (int i = 0; i < n; i++)
      T[i] = random();

  clock_t tantes;    // Valor del reloj antes de la ejecución
  clock_t tdespues;  // Valor del reloj después de la ejecución
  tantes = clock();
  heapsort(T, n);
  tdespues = clock();
  cout << n << ((double)(tdespues-tantes))/CLOCKS_PER_SEC << endl;

  delete [] T;
  return 0;
};
```

2.3.5. *Mergesort*

```
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>
/**
   @brief Ordena un vector por el método de mezcla.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor. Aplica el algoritmo de mezcla.
*/
inline static
void mergesort(int T[], int num_elem);
/**
   @brief Ordena parte de un vector por el método de mezcla.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                  vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                  vector a ordenar.
    && inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor. Aplica el algoritmo de la mezcla.
*/
static void mergesort_lims(int T[], int inicial, int final);
/**
   @brief Ordena un vector por el método de inserción.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor. Aplica el algoritmo de inserción.
*/
inline static
void insercion(int T[], int num_elem);
/**
   @brief Ordena parte de un vector por el método de inserción.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                  vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                  vector a ordenar.
   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor. Aplica el algoritmo de la inserción.
*/
static void insercion_lims(int T[], int inicial, int final);
/**
   @brief Mezcla dos vectores ordenados sobre otro.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                  vector a escribir.
   @param final: Posición detrás de la última de la parte del
                  vector a escribir
    && inicial < final.
   @param U: Vector con los elementos ordenados.
   @param V: Vector con los elementos ordenados.
```

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

2.3.6. *Quicksort*

```cpp
#include <iostream>
using namespace std;
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>
/**
   @brief Ordena un vector por el método quicksort.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo quicksort.
*/
inline static
void quicksort(int T[], int num_elem);
/**
   @brief Ordena parte de un vector por el método quicksort.

   @param T: vector de elementos. Tiene un número de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                   vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                   vector a ordenar.
    && inicial < final.
   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo quicksort.
*/
static void quicksort_lims(int T[], int inicial, int final);

/**
   @brief Ordena un vector por el método de inserción.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.

   Cambia el orden de los elementos de T de forma que los dispone
   en sentido creciente de menor a mayor.
   Aplica el algoritmo de inserción.
*/
inline static
void insercion(int T[], int num_elem);

/**
   @brief Ordena parte de un vector por el método de inserción.

   @param T: vector de elementos. Tiene un número de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                   vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                   vector a ordenar.
    && inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1 de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de inserción.
*/
static void insercion_lims(int T[], int inicial, int final);

/**
   @brief Redistribuye los elementos de un vector según un pivote.

   @param T: vector de elementos. Tiene un número de elementos
                   mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                   vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                   vector a ordenar.
    && inicial < final.
```

```
   @param pp: Posición del pivote. Es MODIFICADO.

   Selecciona un pivote los elementos de T situados en las posiciones entre inicial y final - 1. Redistribuye los elementos, situando los menores que el pivote a su izquierda, después los iguales y a la derecha los mayores. La posición del pivote se devuelve en pp.
*/
static void dividir_qs(int T[], int inicial, int final, int & pp);
/**
   Implementación de las funciones
**/

inline static void insercion(int T[], int num_elem){
  insercion_lims(T, 0, num_elem);
}
static void insercion_lims(int T[], int inicial, int final){
  int i, j;
  int aux;
  for (i = inicial + 1; i < final; i++) {
    j = i;
    while ((T[j] < T[j-1]) && (j > 0)) {
      aux = T[j];
      T[j] = T[j-1];
      T[j-1] = aux;
      j--;
    };
  };
}
const int UMBRAL_QS = 50;
inline void quicksort(int T[], int num_elem){
  quicksort_lims(T, 0, num_elem);
}
static void quicksort_lims(int T[], int inicial, int final){
  int k;
  if (final - inicial < UMBRAL_QS)
    insercion_lims(T, inicial, final);
   else {
    dividir_qs(T, inicial, final, k);
    quicksort_lims(T, inicial, k);
    quicksort_lims(T, k + 1, final);
  };
}
static void dividir_qs(int T[], int inicial, int final, int & pp){
  int pivote, aux;
  int k, l;
  pivote = T[inicial];
  k = inicial;
  l = final;
  do {
    k++;
  } while ((T[k] <= pivote) && (k < final-1));
  do {
    l--;
  } while (T[l] > pivote);
  while (k < l) {
    aux = T[k];
    T[k] = T[l];
    T[l] = aux;
    do k++; while (T[k] <= pivote);
    do l--; while (T[l] > pivote);
  };
  aux = T[inicial];
  T[inicial] = T[l];
  T[l] = aux;
  pp = l;
};

int main(int argc, char * argv[]){
    if (argc != 2){
      cerr << "Formato " << argv[0] << " <num_elem>" << endl;
      return -1;
    }
  int n = atoi(argv[1]);
  int * T = new int[n];
  assert(T);
  srandom(time(0));
  for (int i = 0; i < n; i++)
      T[i] = random();

  clock_t tantes;    // Valor del reloj antes de la ejecución
  clock_t tdespues;  // Valor del reloj después de la ejecución
  tantes = clock();
  quicksort(T, n);
  tdespues = clock();
  cout << n << ((double)(tdespues-tantes))/CLOCKS_PER_SEC << endl;

  delete [] T;
  return 0;
};
```

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.

```
#!/bin/bash

#Primer argumento: programa a ejecutar
#Segundo argumento: tamaño inicial
#Tercer argumento : incremento

if [ $# -eq 3 ]
then
 & i="0"
 & output="out"
 & tam=$2
 & while [ $i -lt 25 ]
 & do
 &  & ./$1 $tam >> $1.out
 &  & i=$[$i+1]
 &  & tam=$[$tam+$3]
 & done
else
 & echo "Error de argumentos"
fi
```

Listing 1: Script individual

```
#!/bin/bash
echo "Ejecutando burbuja..."
./individual.sh burbuja 1000 1000
echo "Ejecutando insercion..."
./individual.sh insercion 1000 1000
echo "Ejecutando seleccion..."
./individual.sh seleccion 1000 1000
echo "Ejecutando mergesort..."
./individual.sh mergesort 1000000 500000
echo "Ejecutando quicksort..."
./individual.sh quicksort 1000000 500000
echo "Ejecutando heapsort..."
./individual.sh heapsort 1000000 500000
echo "Ejecutando hanoi..."
./individual.sh hanoi 10 1
echo "Ejecutando floyd..."
./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:

```
& clock_t tantes;
& clock_t tdespues;
& tantes = clock();
& algoritmo_en_cuestion(T, n);
& tdespues = clock();
& cout << ((double)(tdespues - tantes))
& / CLOCKS_PER_SEC << endl;
```

Listing 4: Código fuente modificado

Por último, este script genera las gráficas mediante gnuplot:

```
#!/usr/bin/gnuplot
#Burbuja

set xlabel "Tamanio del problema"
set ylabel "Tiempo (seg)"
set terminal png size 640,480
set output 'empirica_burbuja.png'
plot 'burbuja.out' with lines

#Floyd

set terminal png size 640,480
set output 'empirica_floyd.png'
plot 'floyd.out' with lines

#Hanoi

set terminal png size 640,480
set output 'empirica_hanoi.png'
plot 'hanoi.out' with lines

#Heapsort

set terminal png size 640,480
set output 'empirica_heapsort.png'
plot 'heapsort.out' with lines

#Insercion

set terminal png size 640,480
set output 'empirica_insercion.png'
plot 'insercion.out' with lines

#Mergesort

set terminal png size 640,480
set output 'empirica_mergesort.png'
plot 'mergesort.out' with lines

#Quicksort

set terminal png size 640,480
set output 'empirica_quicksort.png'
plot 'quicksort.out' with lines

#Selección

set terminal png size 640,480
set output 'empirica_seleccion.png'
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 $O(n^2)$

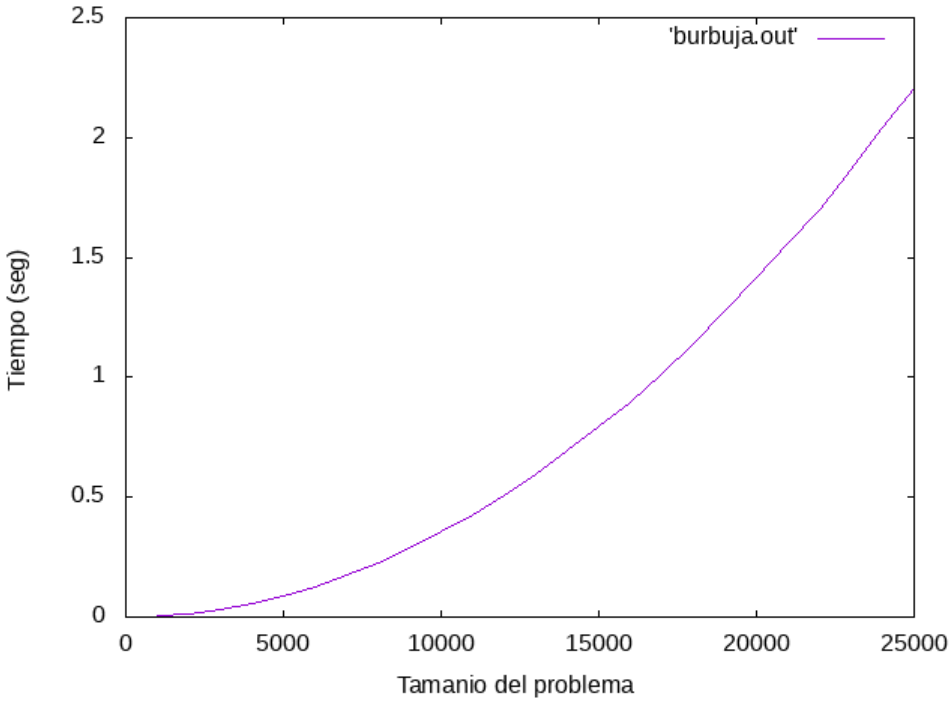


Figura 1: Algoritmo burbuja

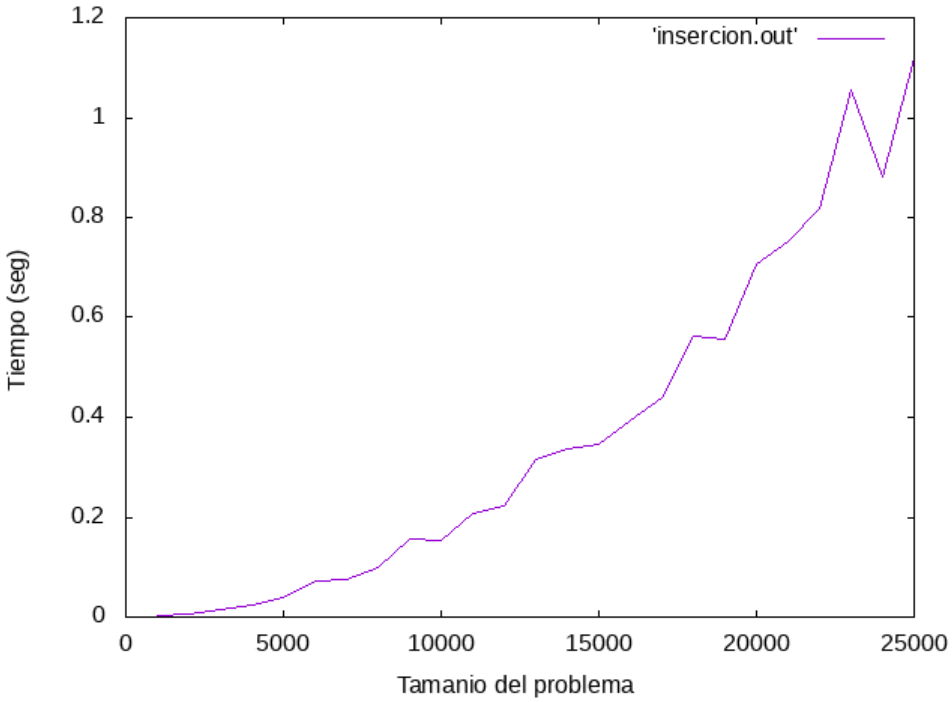


Figura 2: Algoritmo de inserción

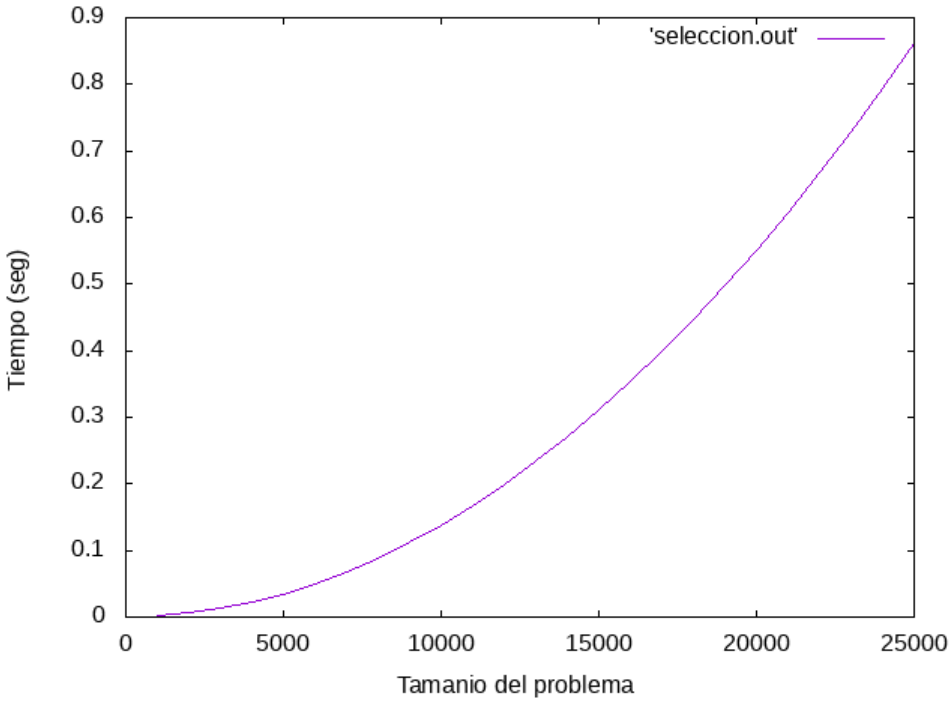


Figura 3: Algoritmo de selección

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

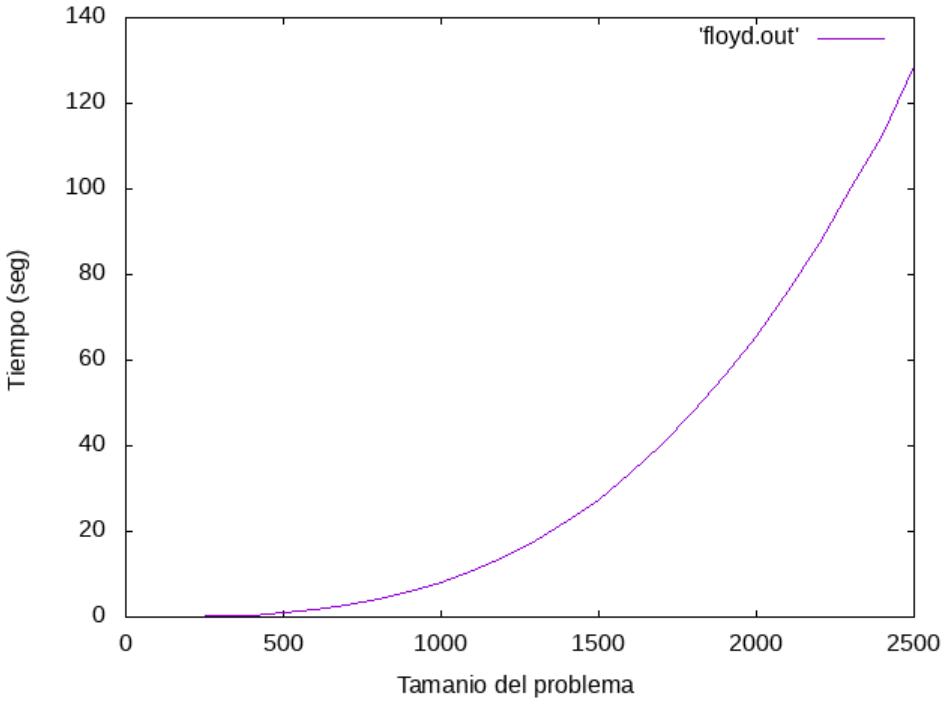


Figura 4: Algoritmo de Floyd

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

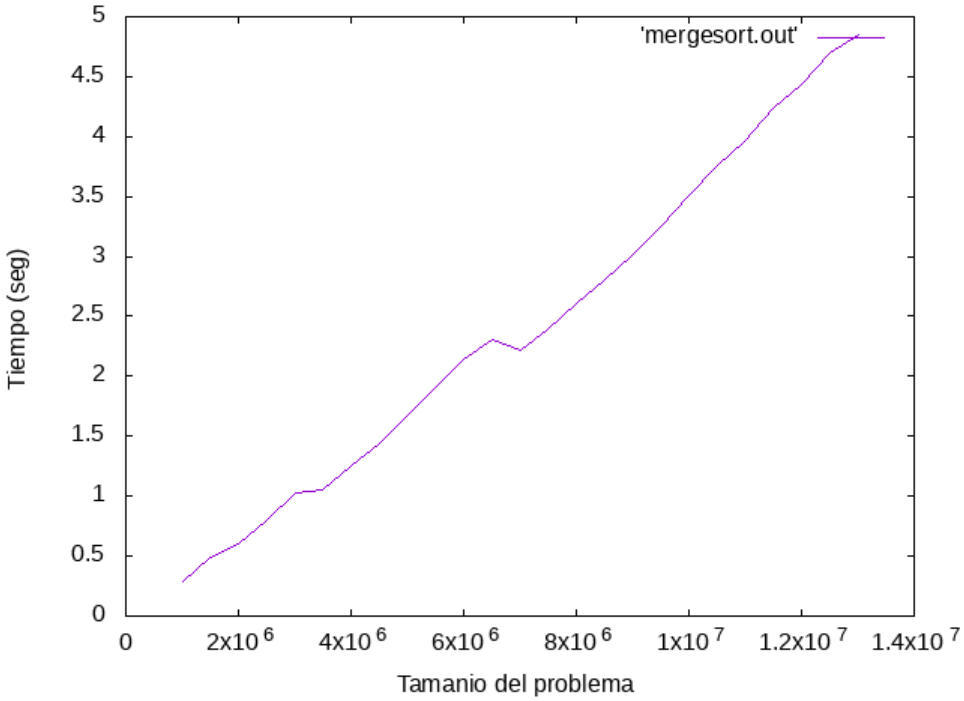


Figura 5: Algoritmo mergesort

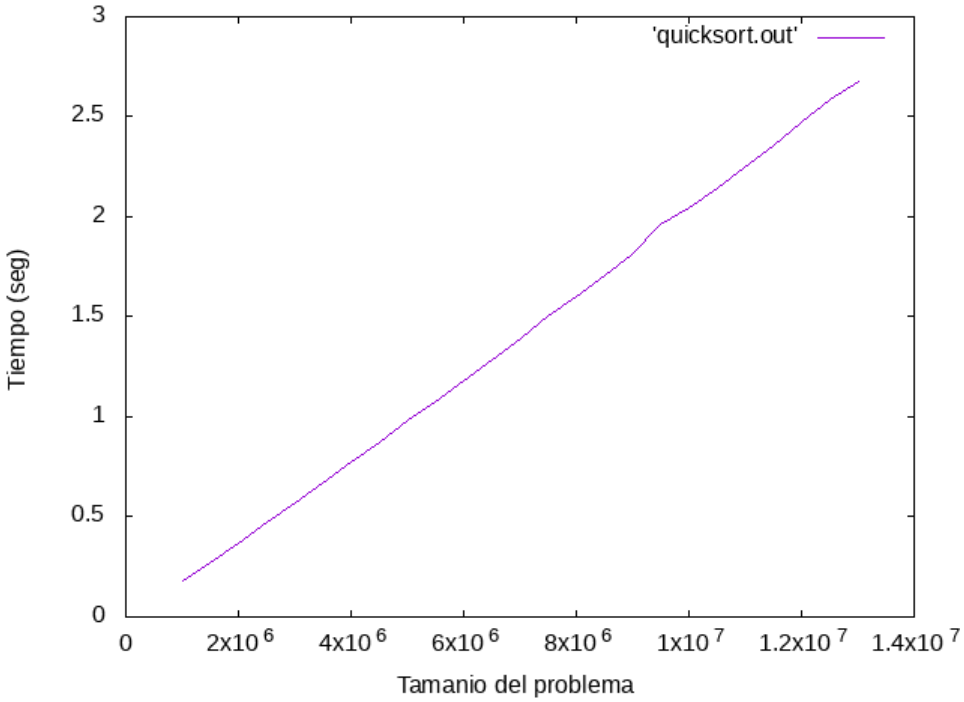


Figura 6: Algoritmo quicksort

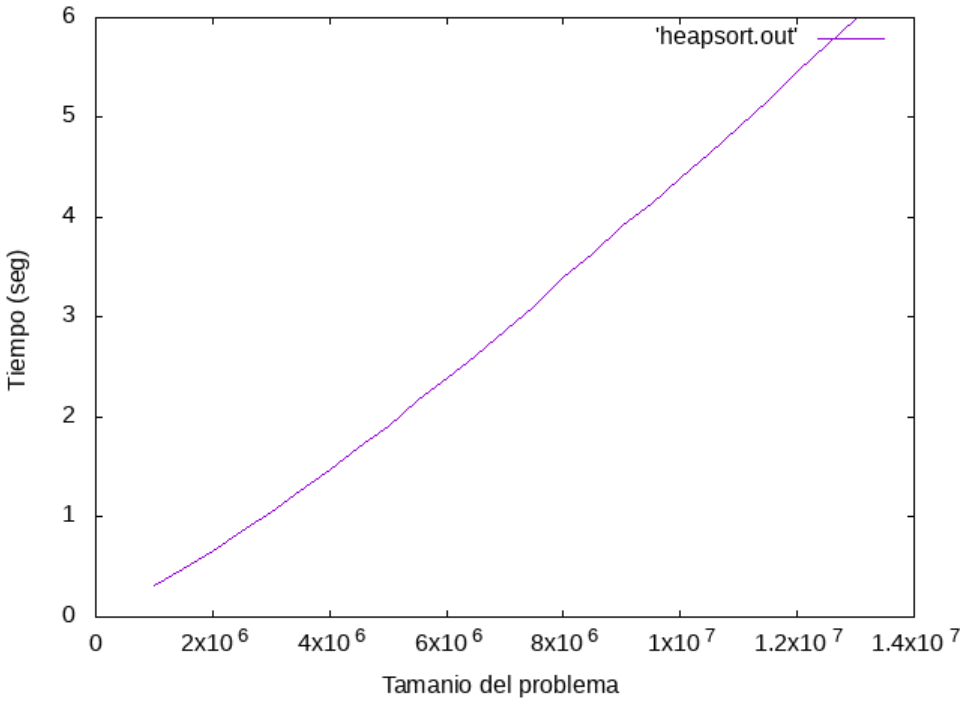


Figura 7: Algoritmo heapsort

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

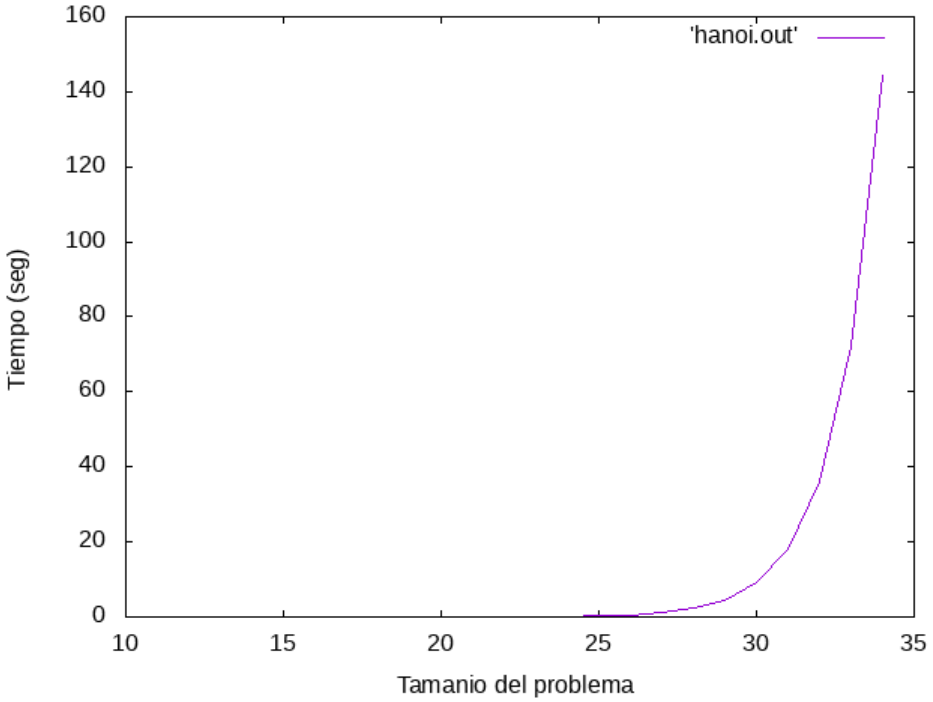


Figura 8: Algoritmo Hanoi

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.

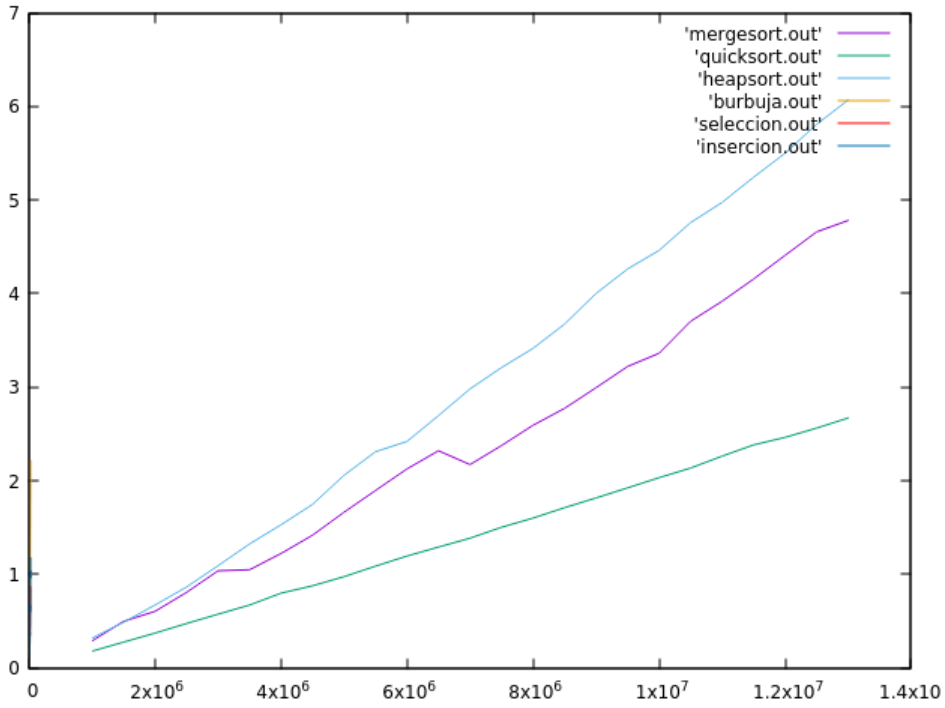


Figura 9: Comparación de algoritmos de ordenación

Si hacemos zoom, podremos ver mejor la diferencia:

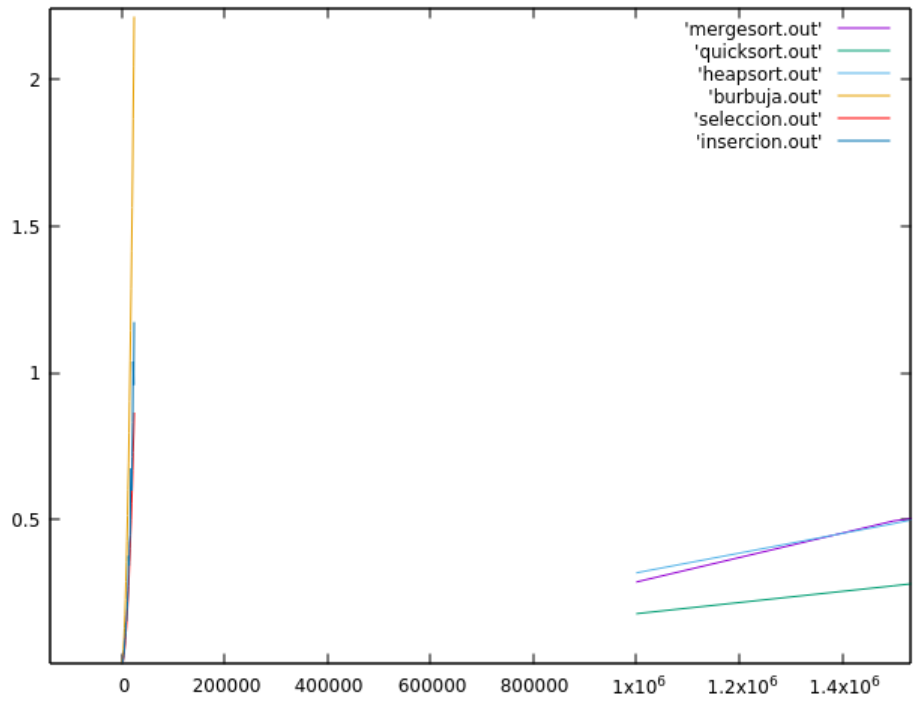


Figura 10: Comparación de algoritmos de ordenación (zoom)

3.4. Variación de la eficiencia empírica

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

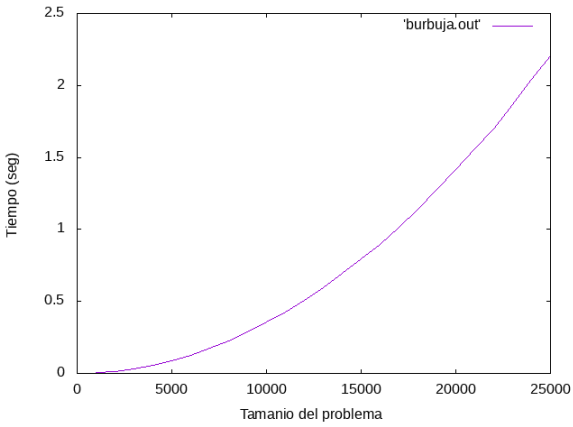
Principio 1 (de Invarianza) *Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia más que, a lo sumo, en una constante multiplicativa.*

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:

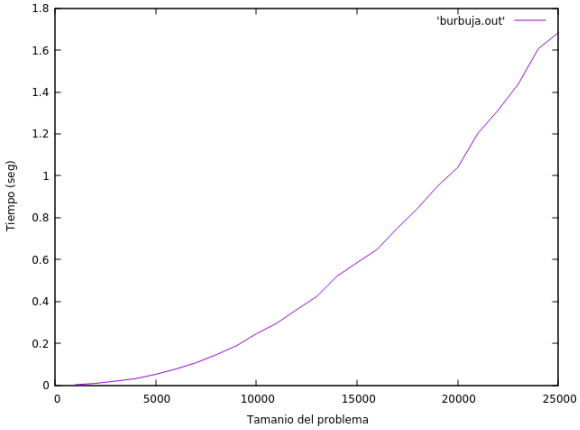
Algoritmo	**Tiempo medio PC 1**	**Tiempo medio PC 2**	**Constante**
Burbuja	0,366	0,251	1,456
Inserción	0,172	0,100	1,715
Selección	0,144	0,124	1,159
Mergesort	1,948	1,422	1,371
Quicksort	1,144	0,965	1,186
Heapsort	2,314	1,821	1,271
Floyd	8,636	5,348	1,615
Hanoi	0,036	0,023	1,538

Cuadro 2: Comparación de tiempos entre ambos entornos de ejecución

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



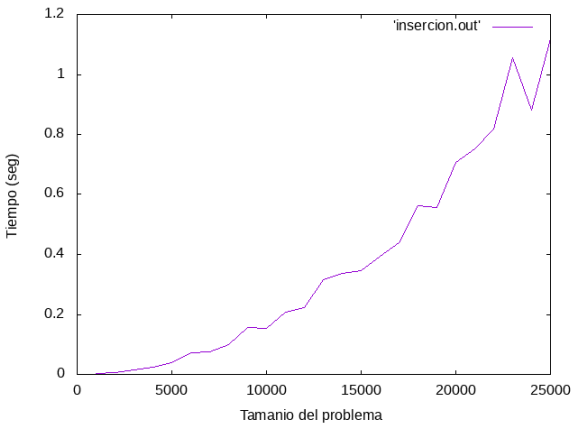
(a) PC 1



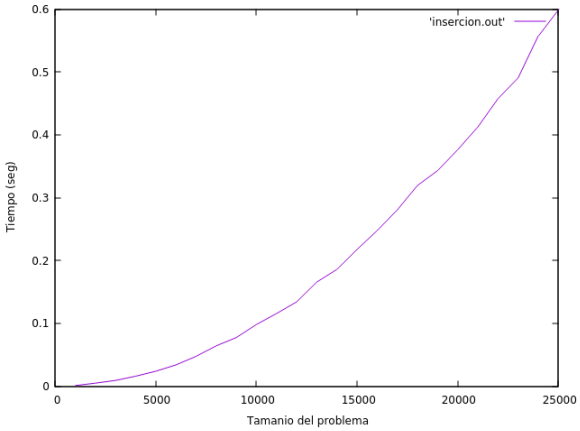
(b) PC 2

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



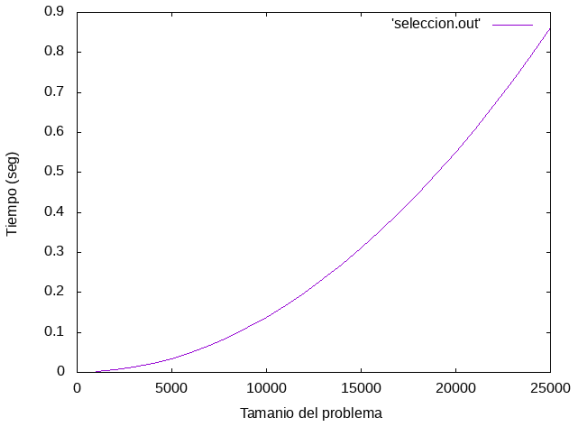
(a) PC 1



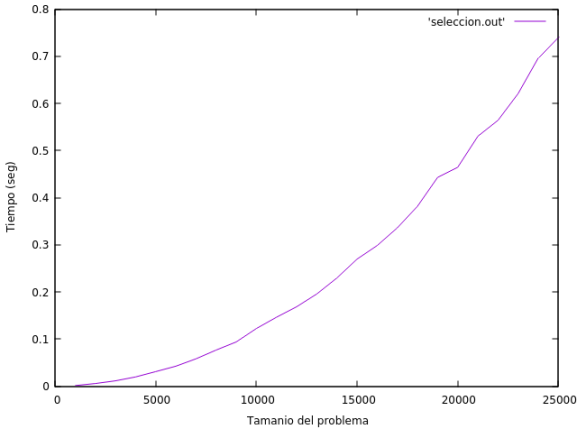
(b) PC 2

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

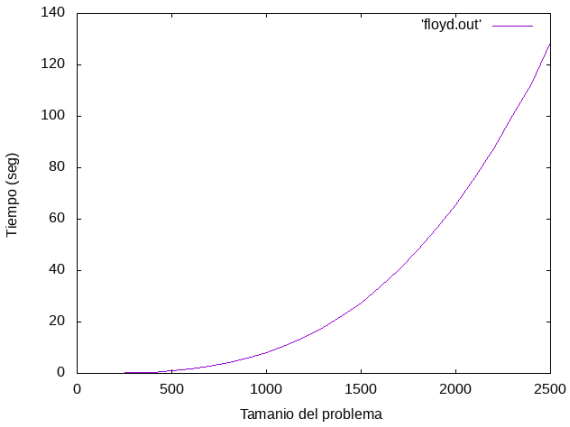


(b) PC 2

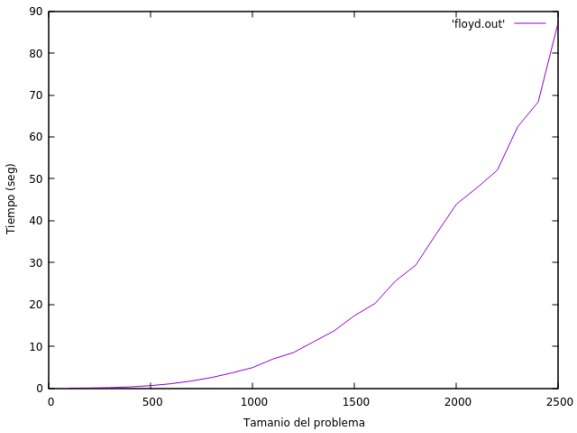
Tamaño	**Tiempo PC 1**	**Tiempo PC 2**
1000	0,001469	0,001203
2000	0,005661	0,005539
3000	0,012639	0,011269
4000	0,022373	0,019565
5000	0,0348	0,030867
6000	0,050395	0,042368
7000	0,06795	0,057941
8000	0,088633	0,076493
9000	0,112056	0,093877
10000	0,138403	0,122324
11000	0,167213	0,146299
12000	0,198844	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

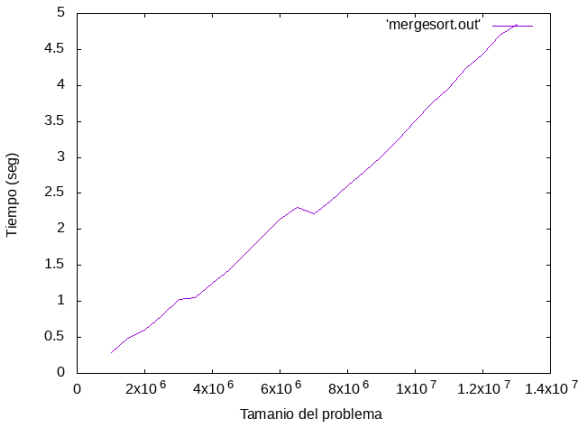


(b) PC 2

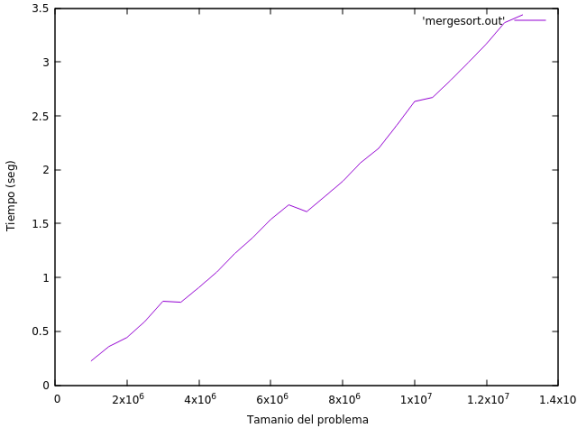
Tamaño	**Tiempo PC 1**	**Tiempo PC 2**
100	0,008394	0,005113
200	0,066017	0,041535
300	0,220825	0,138818
400	0,521858	0,311018
500	1,01887	0,602487
600	1,75665	1,05512
700	2,80066	1,70498
800	4,17319	2,5442
900	5,95085	3,65579
1000	8,15234	4,91424
1100	10,8509	6,95897
1200	14,0868	8,47521
1300	17,9348	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 $O(n \cdot log(n))$



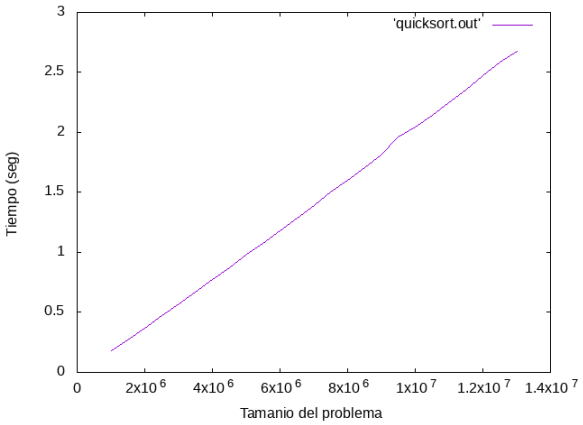
(a) PC 1



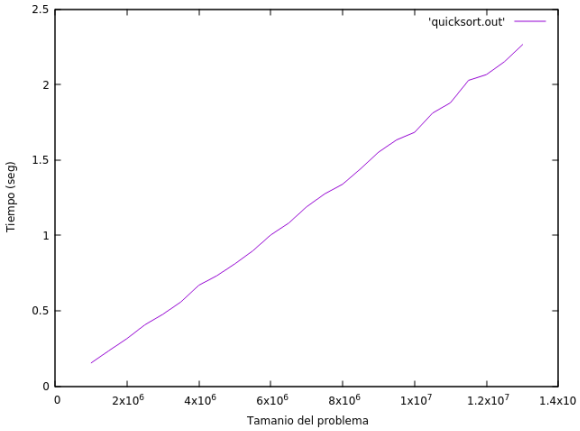
(b) PC 2

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



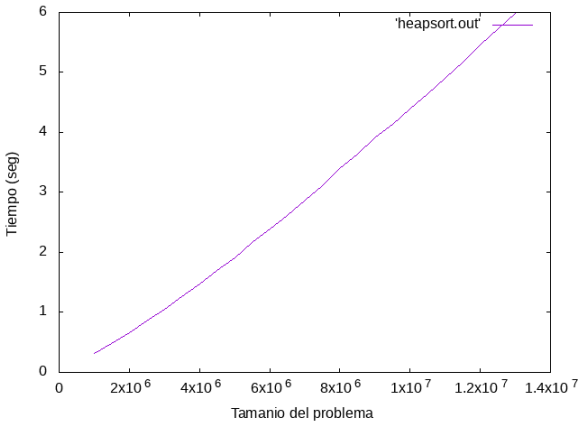
(a) PC 1



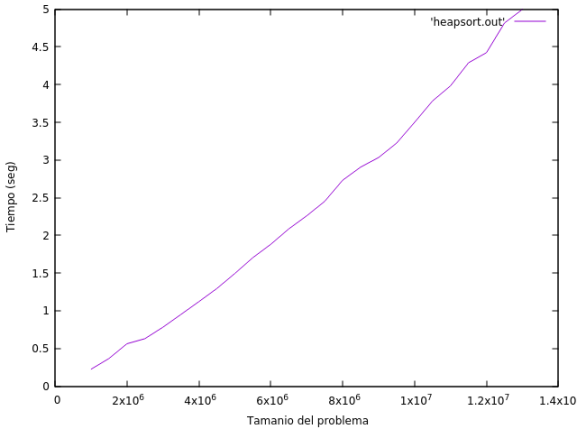
(b) PC 2

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

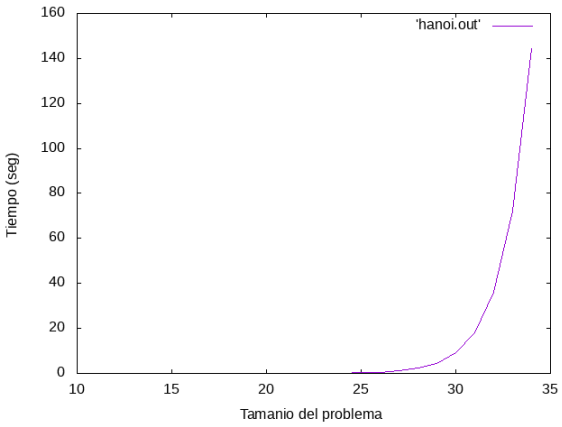


(b) PC 2

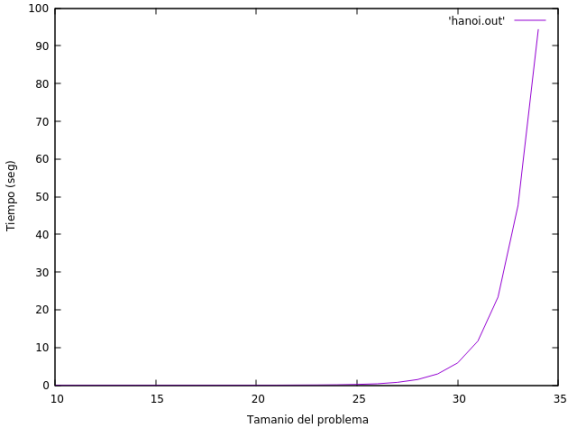
Tamaño	**Tiempo PC 1**	**Tiempo PC 2**
1000000	0,313202d	0,223356
1500000	0,482248d	0,366004
2000000	0,664047d	0,561854
2500000	0,855399d	0,628741
300000	1,05215d	0,781531
350000	1,27027d	0,949929
400000	1,47906d	1,1197
450000	1,69867d	1,29438
500000	1,91449d	1,49264
550000	2,15878d	1,70317
600000	2,38723d	1,88031
650000	2,61876d	2,08545
700000	2,86654d	2,25945
750000	3,12018d	2,45073
800000	3,39958d	2,73144
850000	3,63556d	2,90442
900000	3,90373d	3,03281
950000	4,14154d	3,22381
1000000	4,39101d	3,50009
1050000	4,64831d	3,78361
1100000	4,90958d	3,98295
1150000	5,17252d	4,28962
1200000	5,45655d	4,42294
1250000	5,72861d	4,81687
1300000	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	n^2	2.253e-12 (0.06377 %)
Selección		3.047e-13 (0.02211 %)
Inserción		3.085e-11 (1.805 %)
Heapsort	$n \cdot log(n)$	2.071e-10 (0.7626 %)
Mergesort		1.893e-10 (0.8614 %)
Quicksort		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 $O(n^2)$

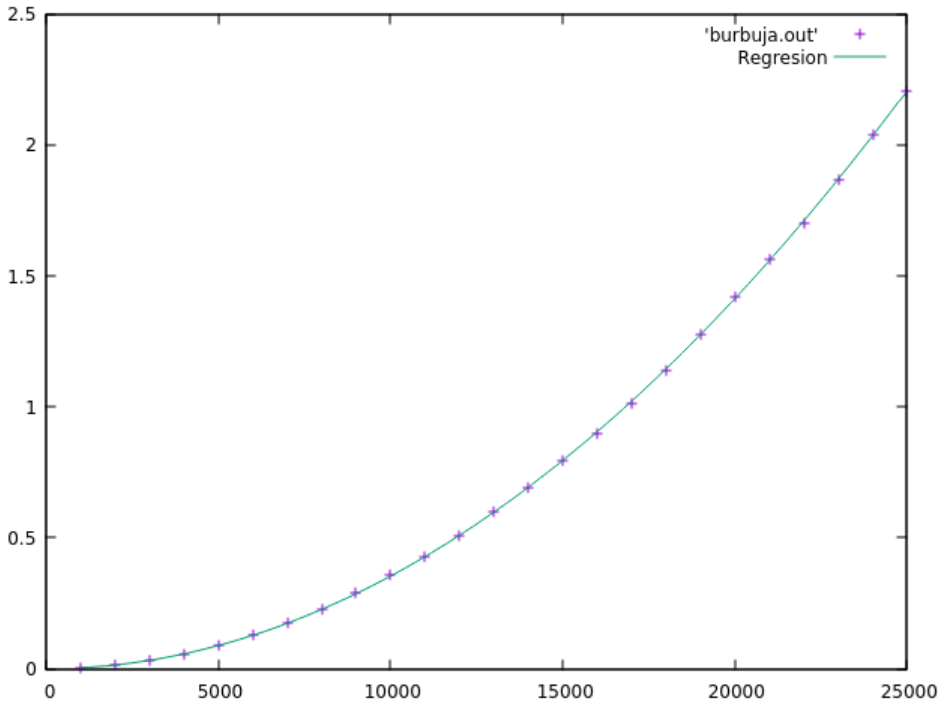


Figura 19: Algoritmo burbuja

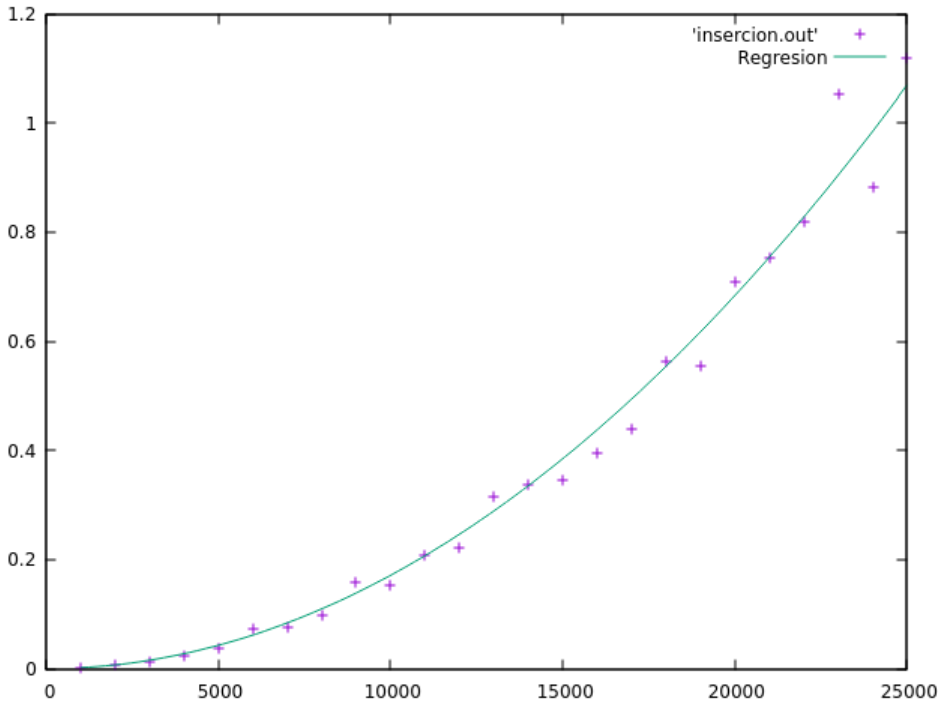


Figura 20: Algoritmo de inserción

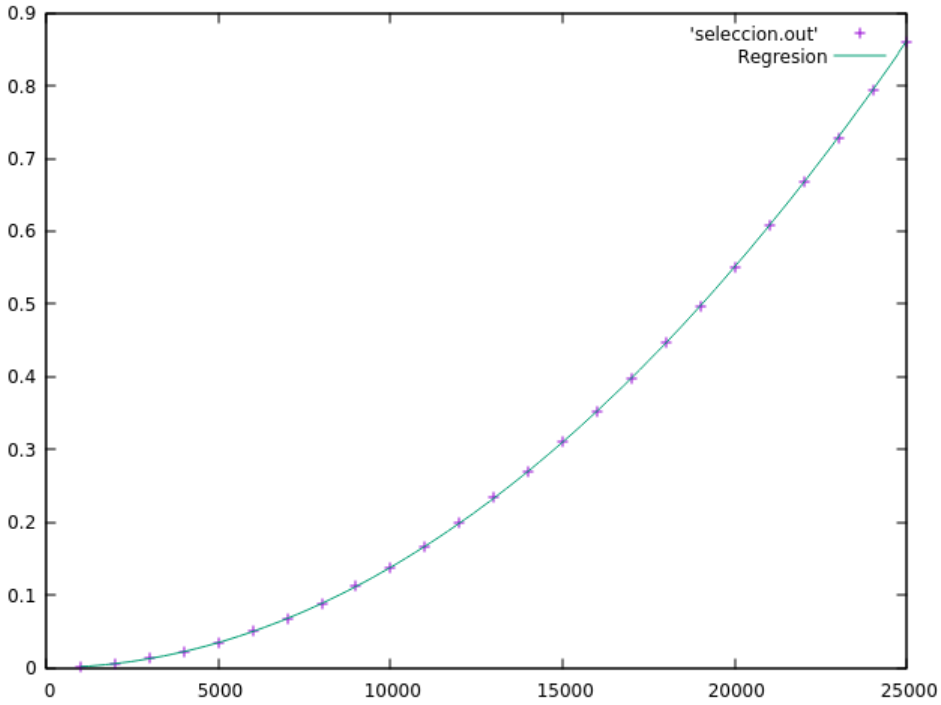


Figura 21: Algoritmo de selección

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

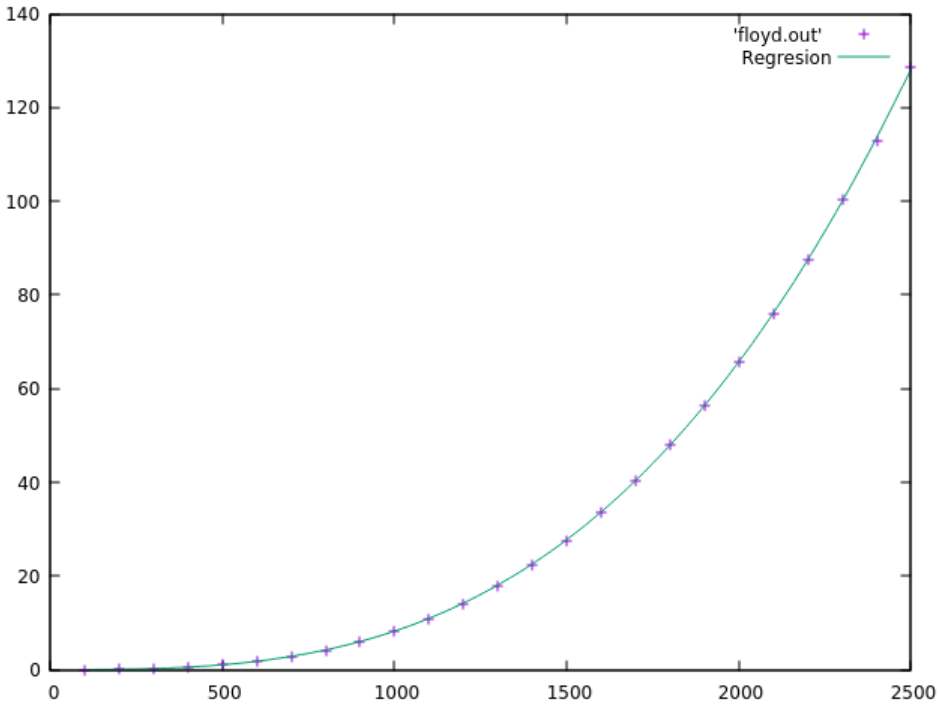


Figura 22: Algoritmo de Floyd

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

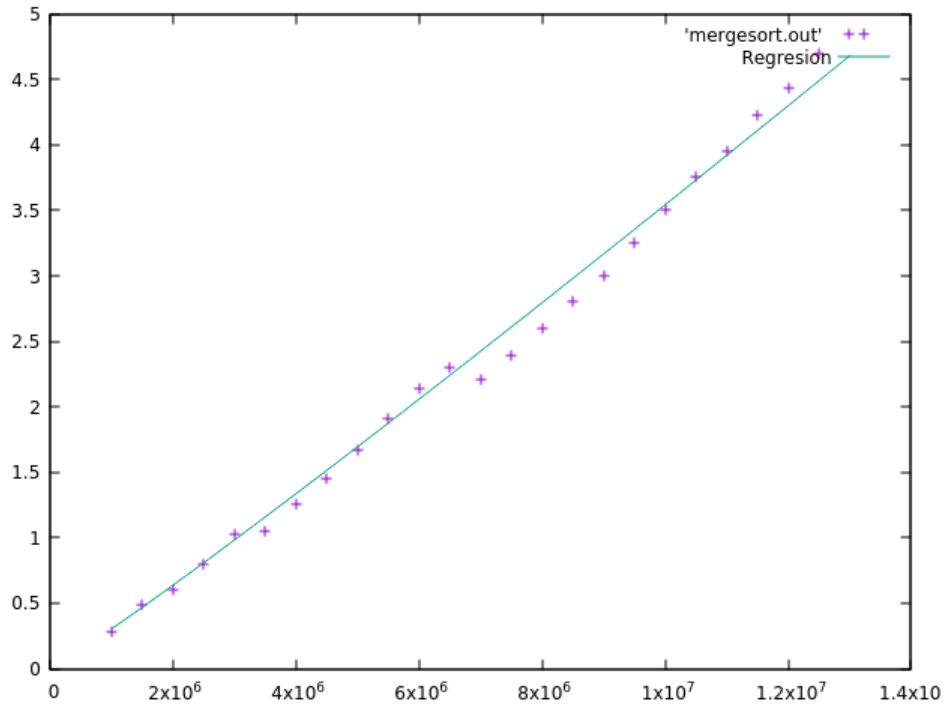


Figura 23: Algoritmo mergesort

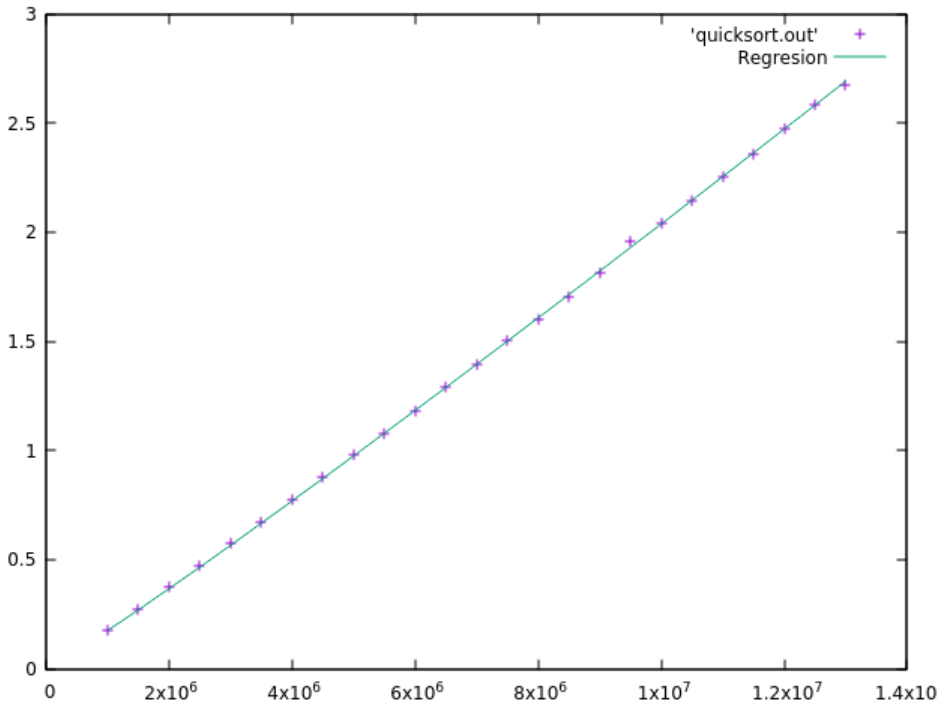


Figura 24: Algoritmo quicksort

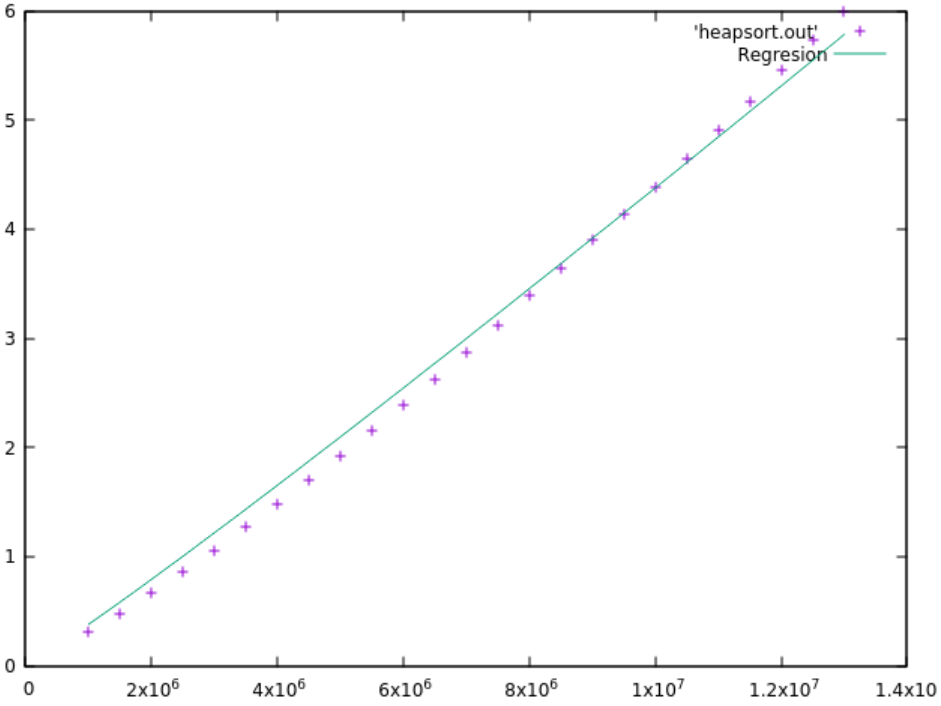


Figura 25: Algoritmo heapsort

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

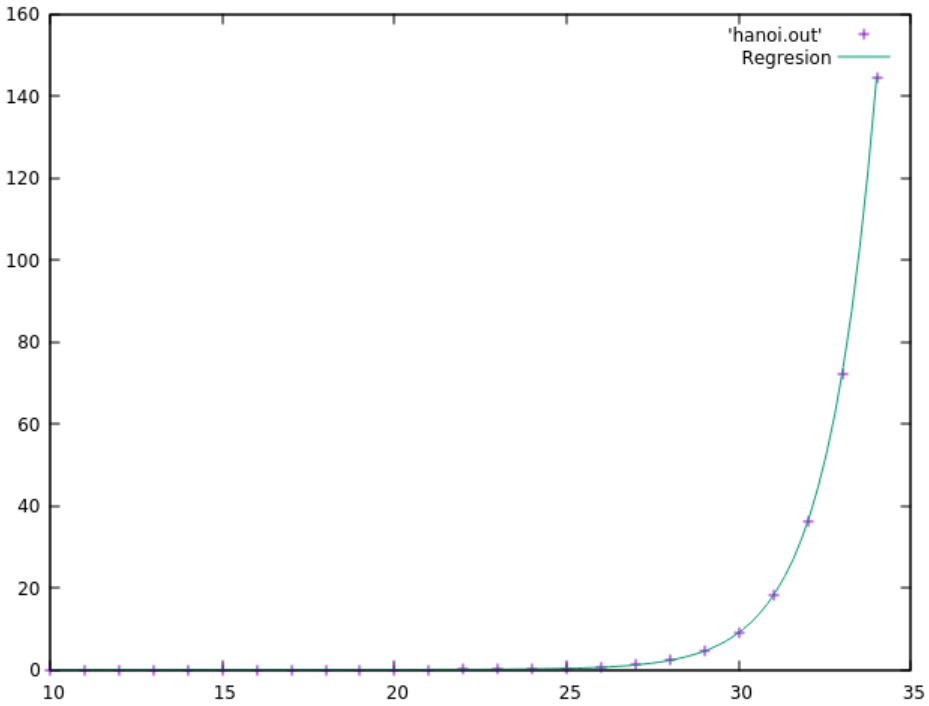


Figura 26: Algoritmo Hanoi

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.

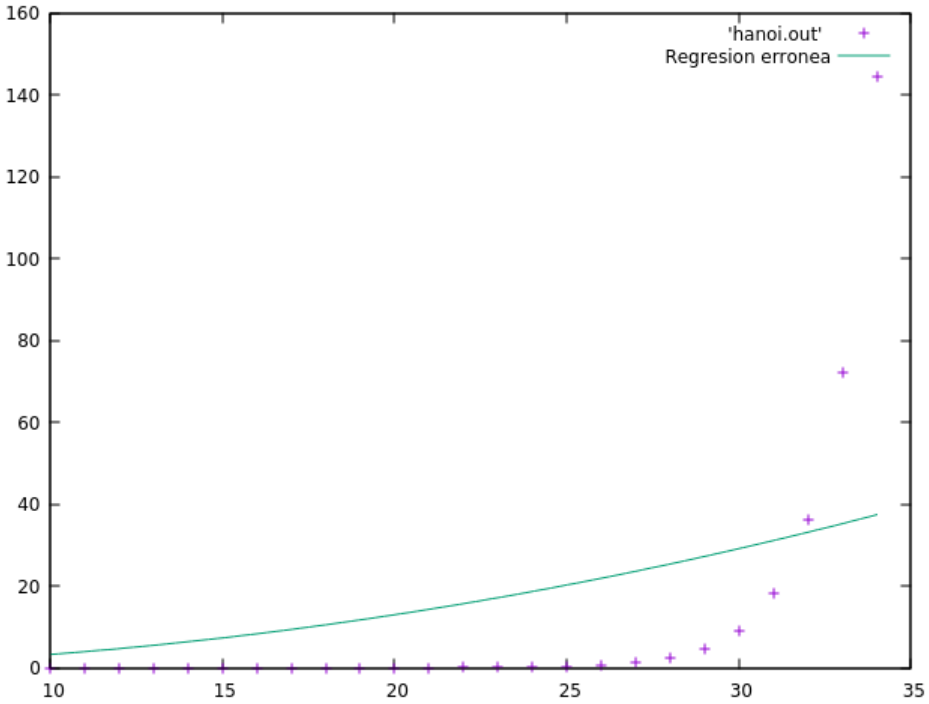


Figura 27: Regresión errónea