Algorítmica

Grado en Ingeniería Informática

Práctica 1

Análisis de eficiencia de algoritmos

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

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Curso 2017-2018

Índice

Índice de cuadros

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);
        }
```

```
85      }
86    return M;
87  }
88
89  void LiberaMatriz(int ** & M, int dim)
90  {
91    for (int i = 0; i < dim; i++)
92      delete [] M[i];
93    delete [] M;
94    M = NULL;
95  }
96
97
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                     M[i][j] = (M[i][j]
                      ↪  > sum) ? sum :
                      ↪  M[i][j];
114               }
115 }
116
117 int main (int argc, char **argv)
118 {
119 //  clock_t tantes;
120 //  clock_t tdespues;
121     int dim;
122
123   //Lectura de los parametros de
    ↪  entrada
124   if (argc != 2)
125     {
126       cout << "Parámetros de entrada:
        ↪  " << endl
127            << "1.- Número de nodos" <<
            ↪  endl << endl;
128       return 1;
129     }
130
131   dim = atoi(argv[1]);
132   int ** M = ReservaMatriz(dim);
133
134   RellenaMatriz(M,dim);
135
136
137  // Empieza el algoritmo de floyd
138   tantes = clock();
139   Floyd(M,dim);
140   tdespues = clock();
141   cout << dim  <<
142   ((double)(tdespues-tantes))
143   /CLOCKS_PER_SEC << endl;
144   LiberaMatriz(M,dim);
145
146   return 0;
```

2.3. Algoritmos de ordenación

2.3.1. Burbuja

```
#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.
                  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 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*

```
#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

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

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

Cuadro 1: Parámetros de ejecución de cada programa

3.2. Entornos de ejecución

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

3.2.1. PC 1

1. CPU : AMD FX-8320 @3.50Ghz
    a) Arquitectura : x86_64
    b) Caché
        1) Caché L1
            a′ L1d : 16K
            b′ L1i : 64K
        2) Caché L2 : 2048K
        3) Caché L3 : 8192K
    c) Frecuencia máxima (Overclock) : 4.20 Ghz
    d) Núcleos físicos : 4
    e) Núcleos lógicos : 8
2. RAM
    a) Capacidad : 16384 MB
    b) Frecuencia : 1600 Mhz
    c) Tecnología : DDR3

3.2.2. PC 2

1. CPU : Intel Core i7-6700HQ @2.60Ghz
    a) Arquitectura : x86_64
    b) Caché
        1) Caché L1
            a′ L1d : 32K
            b′ L1i : 32K
        2) Caché L2 : 256K
        3) Caché L3 : 6144K
    c) Frecuencia máxima (HT) : 3.5 Ghz
    d) Núcleos físicos : 4
    e) Núcleos lógicos : 8
2. RAM
    a) Capacidad : 8192 MB
    b) Frecuencia : 2133 Mhz
    c) Tecnología : DDR4

3.3. Gráficas comparativas

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

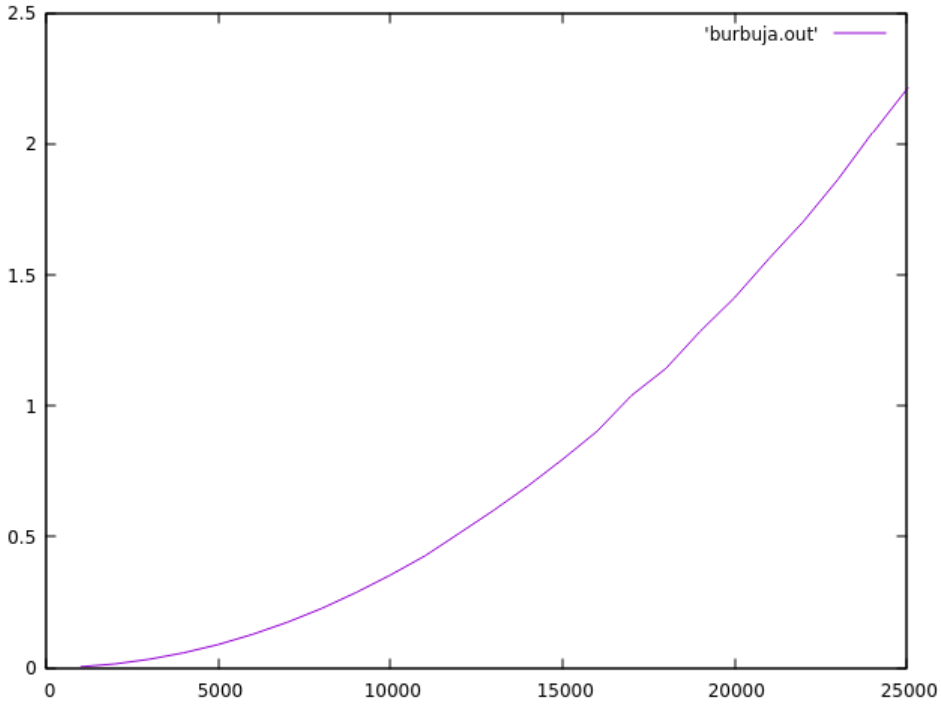


Figura 1: Algoritmo burbuja

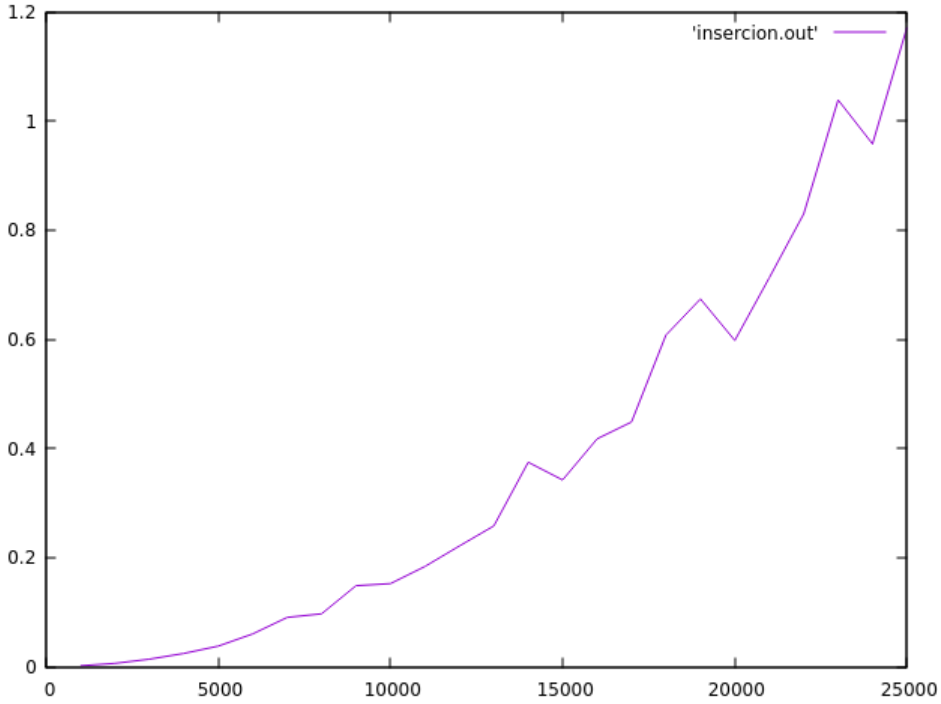


Figura 2: Algoritmo de inserción

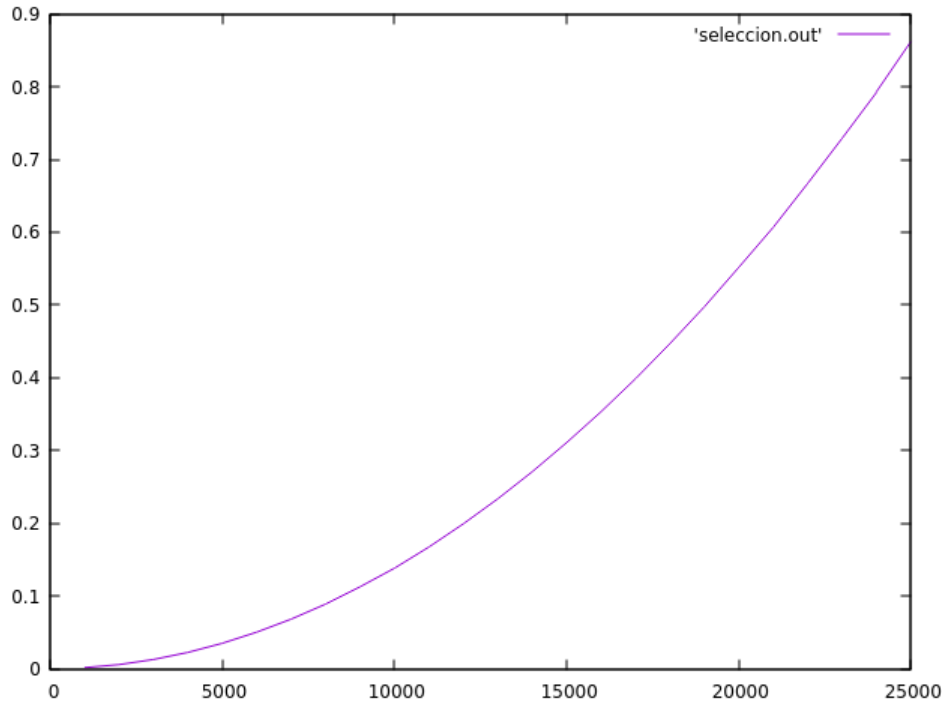


Figura 3: Algoritmo de selección

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

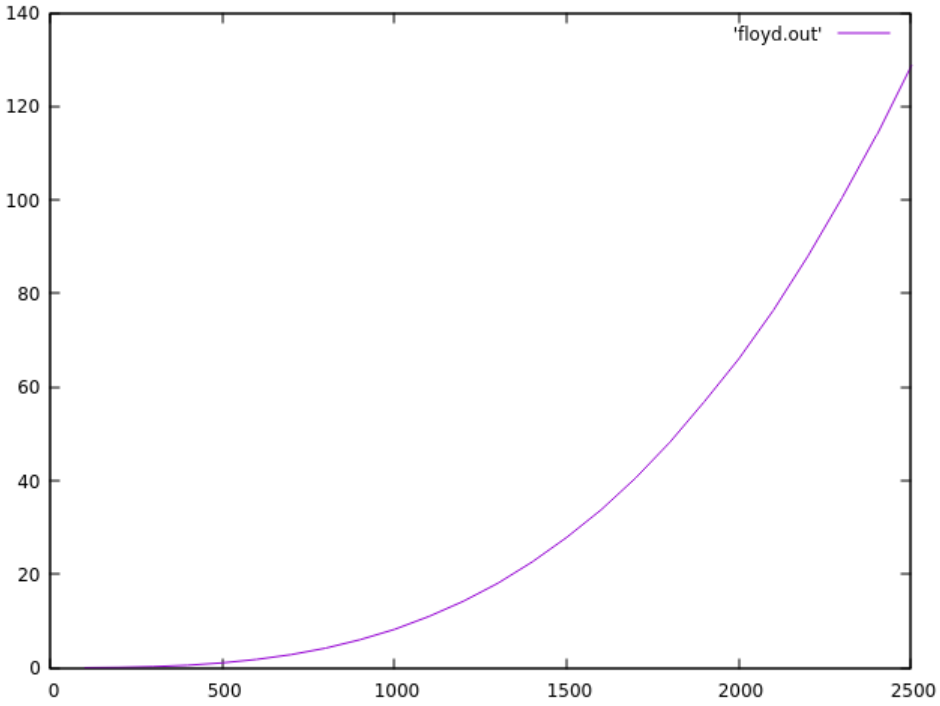


Figura 4: Algoritmo de Floyd

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

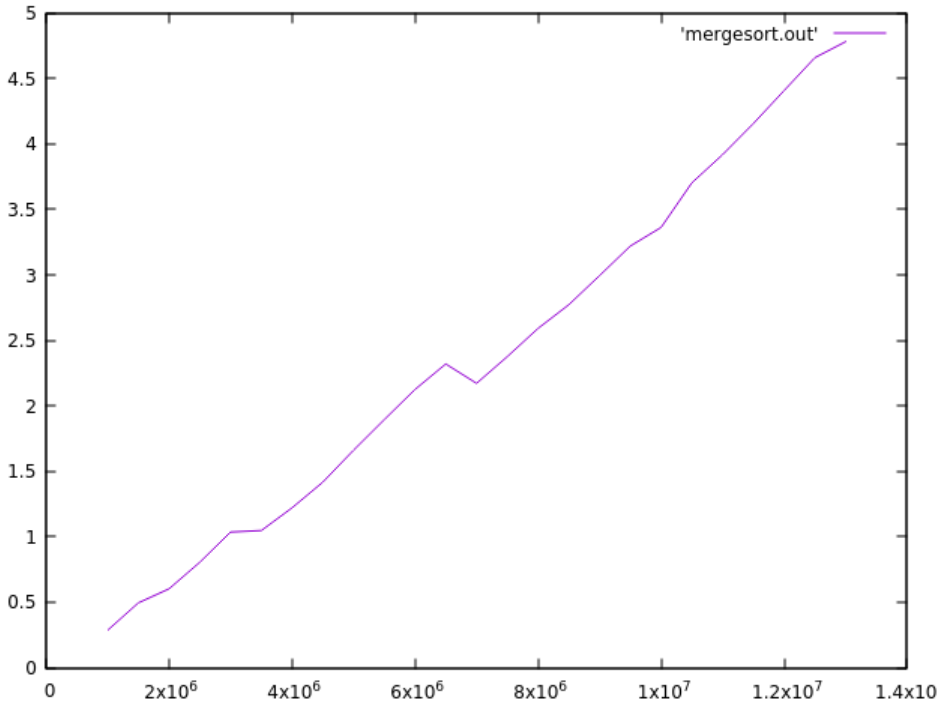


Figura 5: Algoritmo mergesort

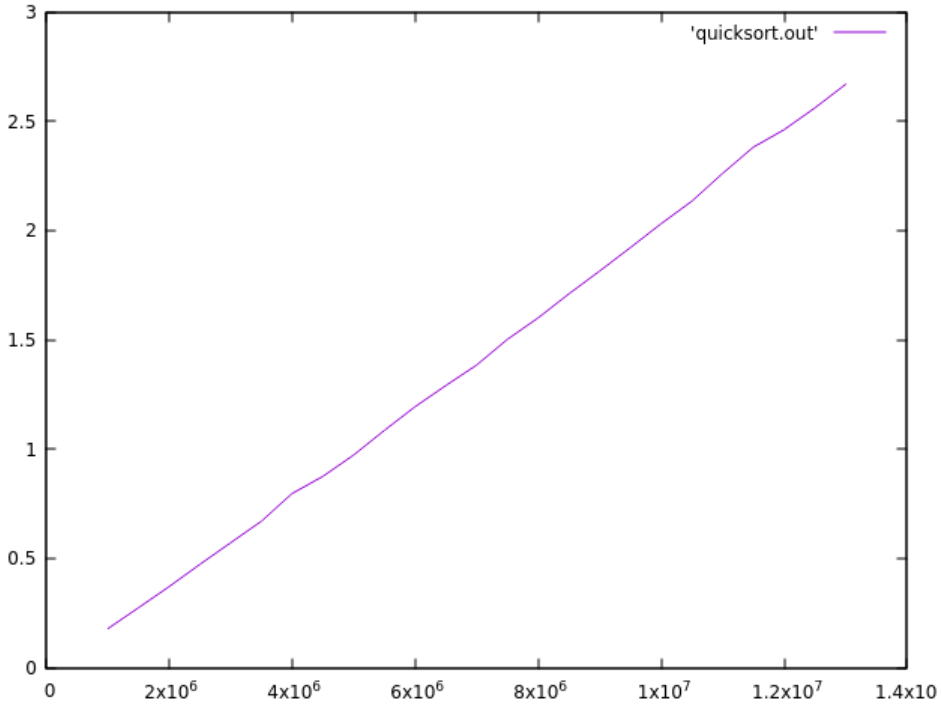


Figura 6: Algoritmo quicksort

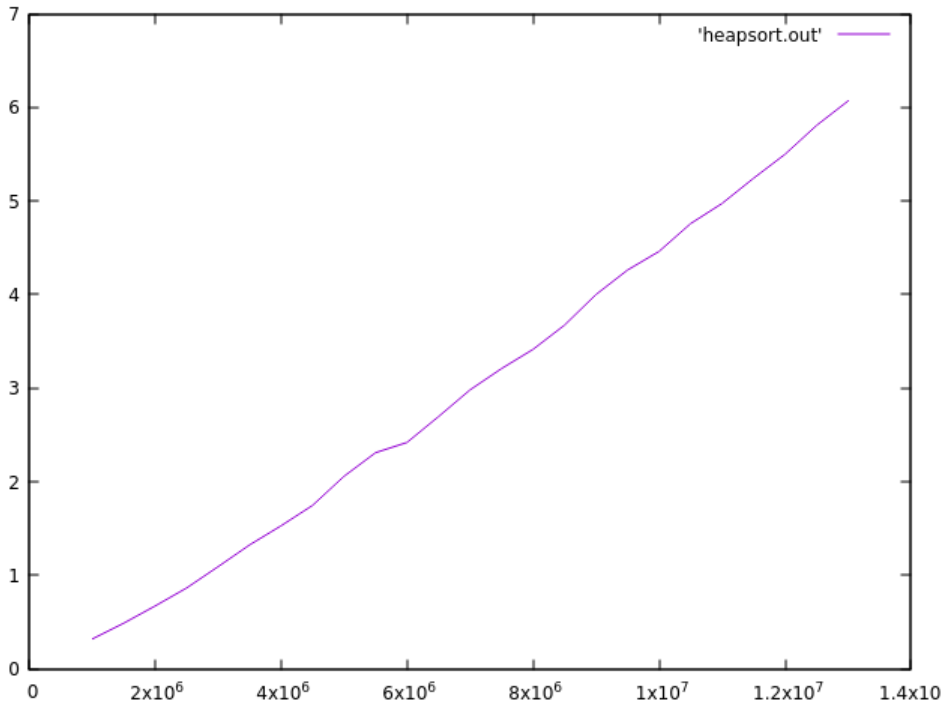


Figura 7: Algoritmo heapsort

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

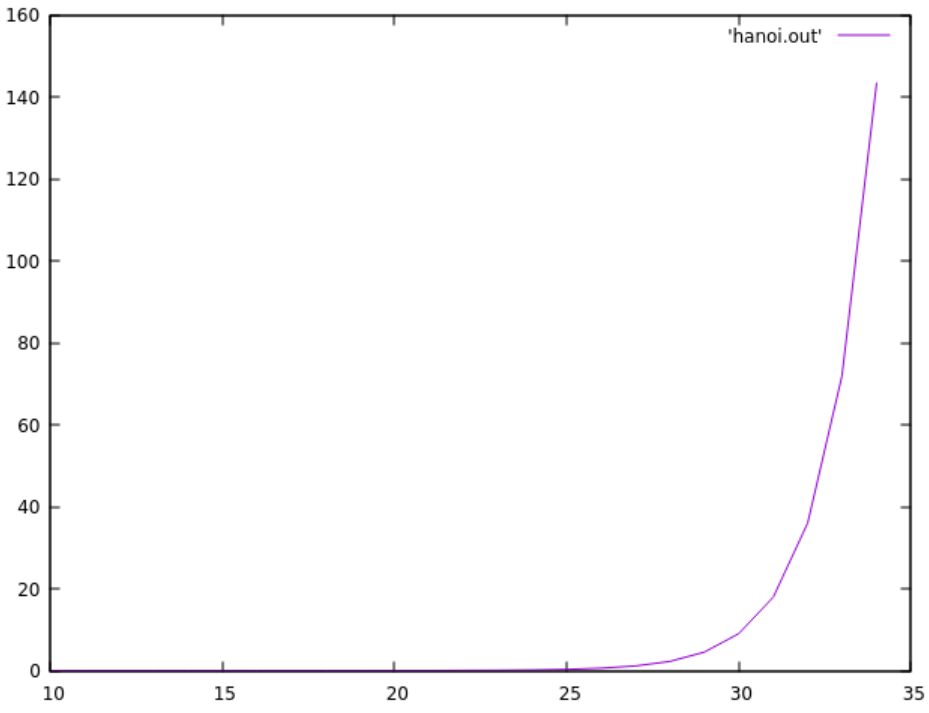


Figura 8: Algoritmo Hanoi

3.3.5. Comparación entre algoritmos de ordenación

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

algoritmos.

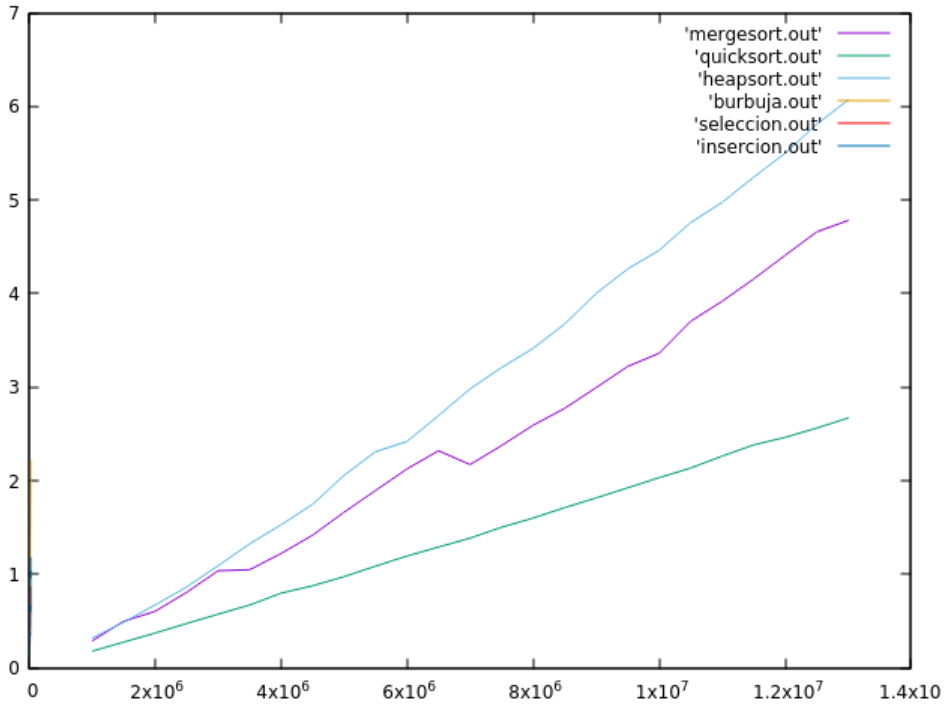


Figura 9: Comparación de algoritmos de ordenación

Si hacemos zoom, podremos ver mejor la diferencia:

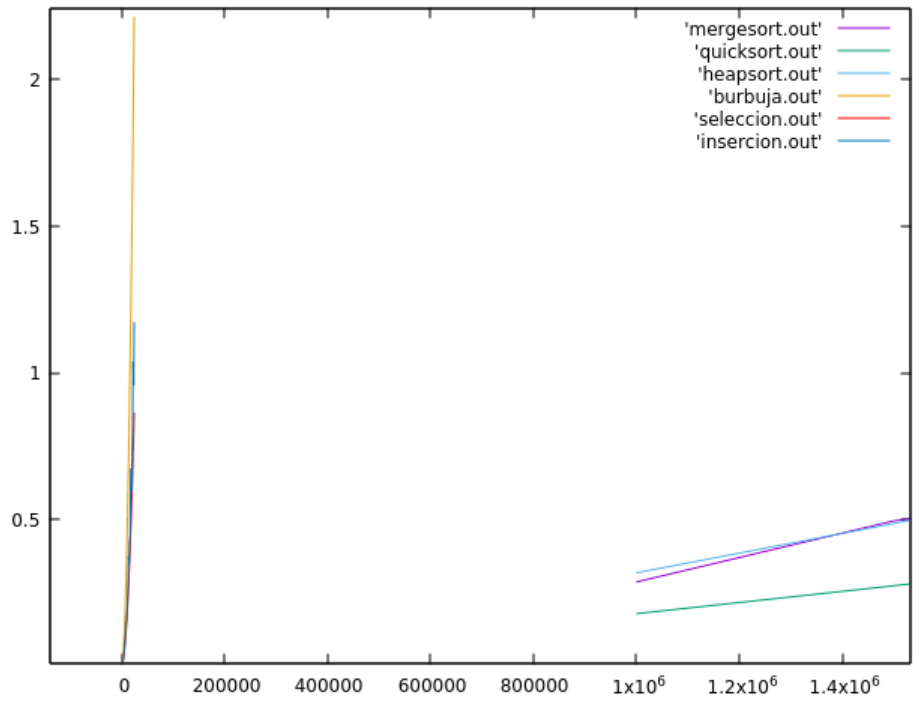


Figura 10: Comparación de algoritmos de ordenación (zoom)

3.4. Variación de la eficiencia empírica

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

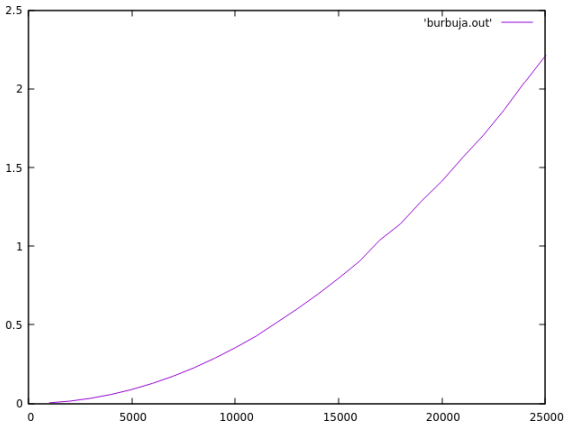
Principio 1 (de Invarianza) *Dos implementaciones diferentes de un mismo algoritmo no difieren en eficiencia más que, a lo sumo, en una constante multiplicativa.*

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

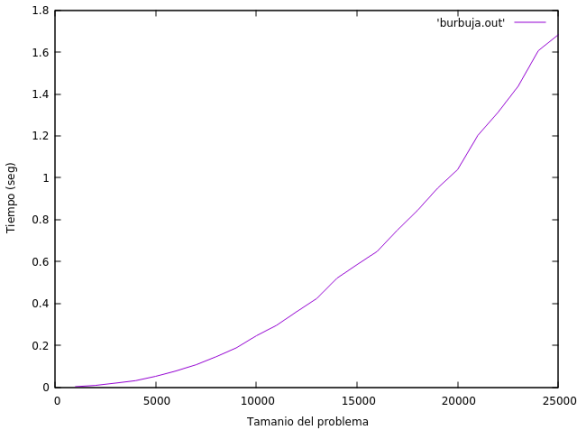
Algoritmo	**Tiempo medio PC 1**	**Tiempo medio PC 2**	**Constante**
Burbuja	pc1	pc2	cte
Inserción	pc1	pc2	cte
Selección	pc1	pc2	cte
Mergesort	pc1	pc2	cte
Quicksort	pc1	pc2	cte
Heapsort	pc1	pc2	cte
Floyd	pc1	pc2	cte
Hanoi	pc1	pc2	cte

Cuadro 2: Comparación entre ambos entornos de ejecución

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



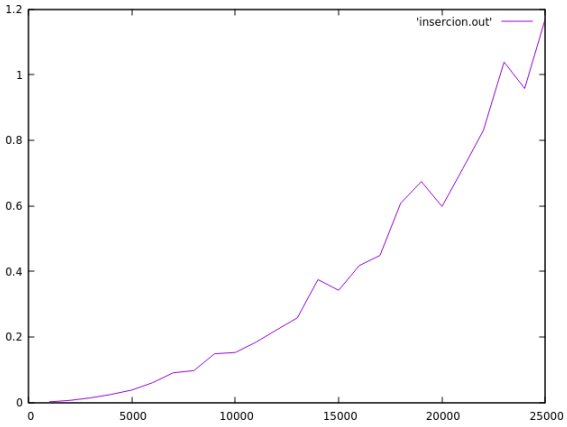
(a) PC 1



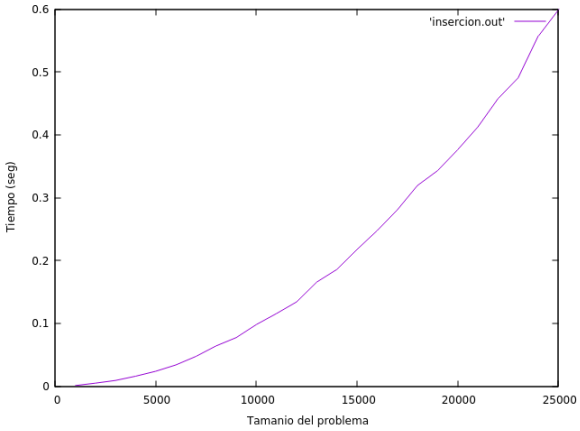
(b) PC 2

Tamaño	**Tiempo PC 1**	**Tiempo PC 2**
t	t1	t2

Figura 11: Algoritmo burbuja



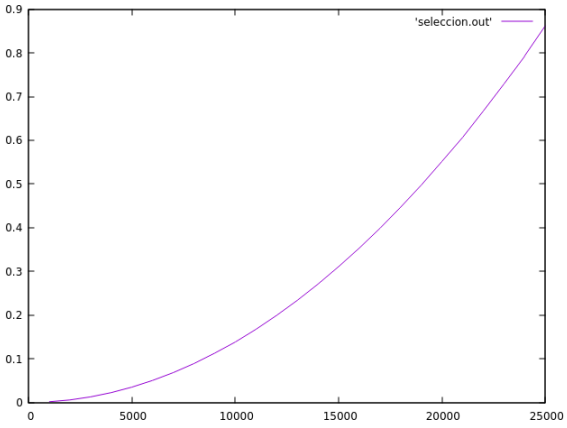
(a) PC 1



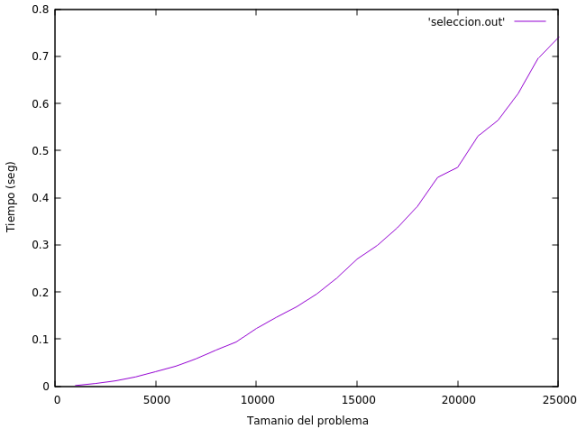
(b) PC 2

Tamaño	**Tiempo PC 1**	**Tiempo PC 2**
t	t1	t2

Figura 12: Algoritmo de inserción



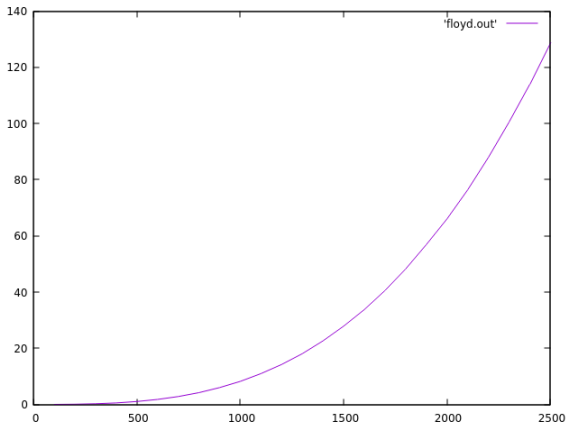
(a) PC 1



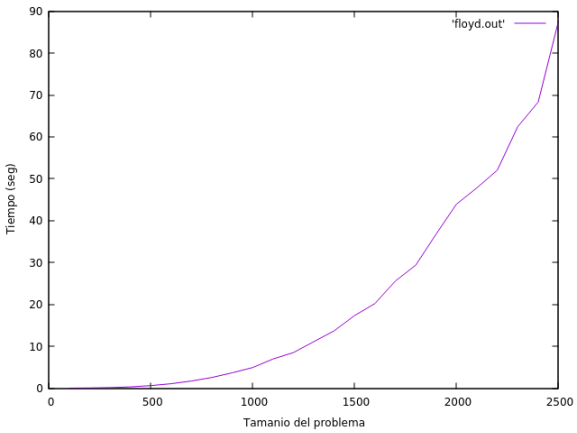
(b) PC 2

Figura 13: Algoritmo de selección

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



(a) PC 1

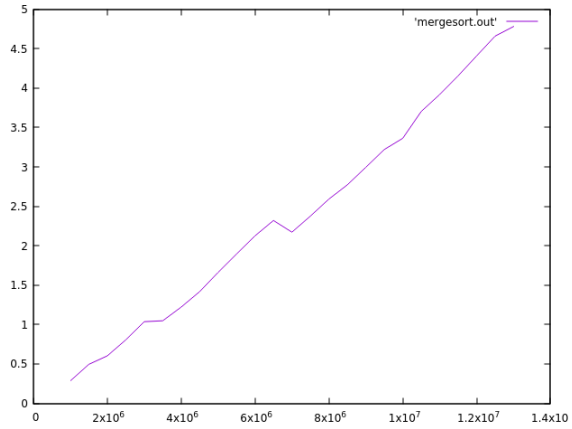


(b) PC 2

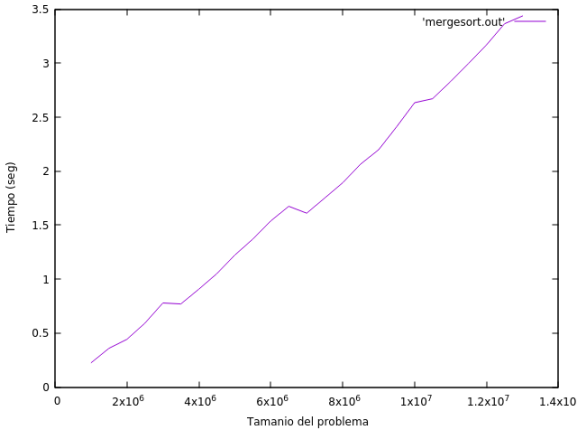
Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 14: Algoritmo de Floyd

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



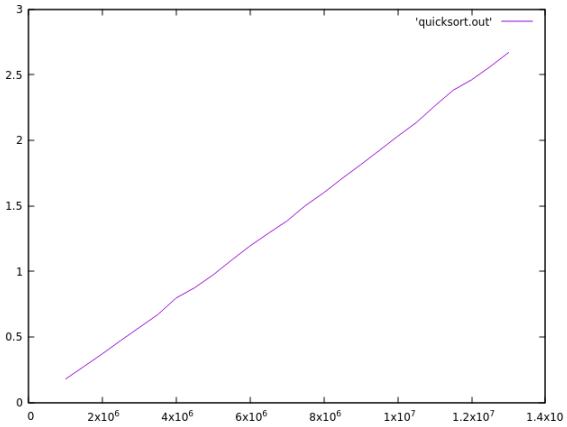
(a) PC 1



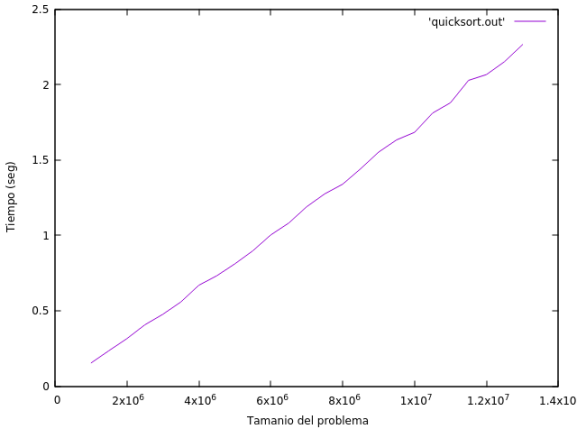
(b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 15: Algoritmo mergesort



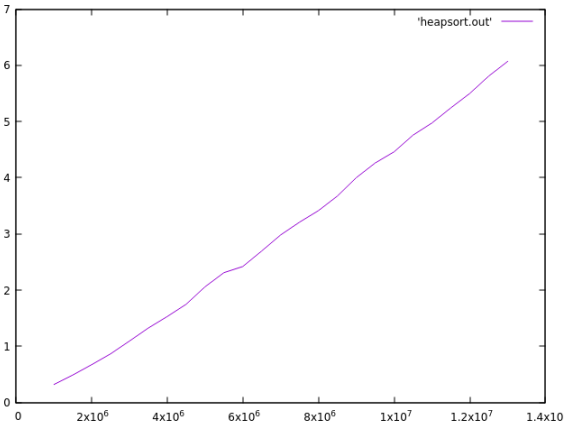
(a) PC 1



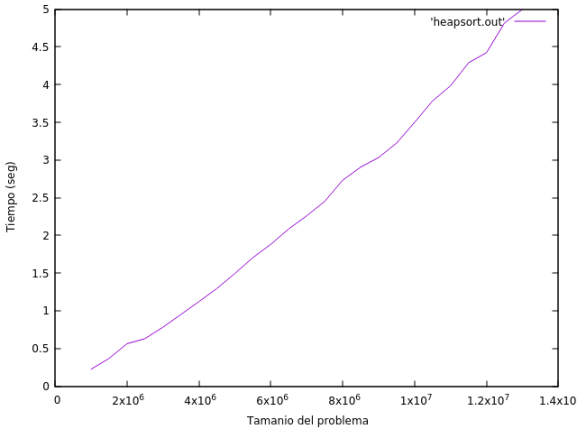
(b) PC 2

Tamaño	**Tiempo PC 1**	**Tiempo PC 2**
t	t1	t2

Figura 16: Algoritmo quicksort



(a) PC 1

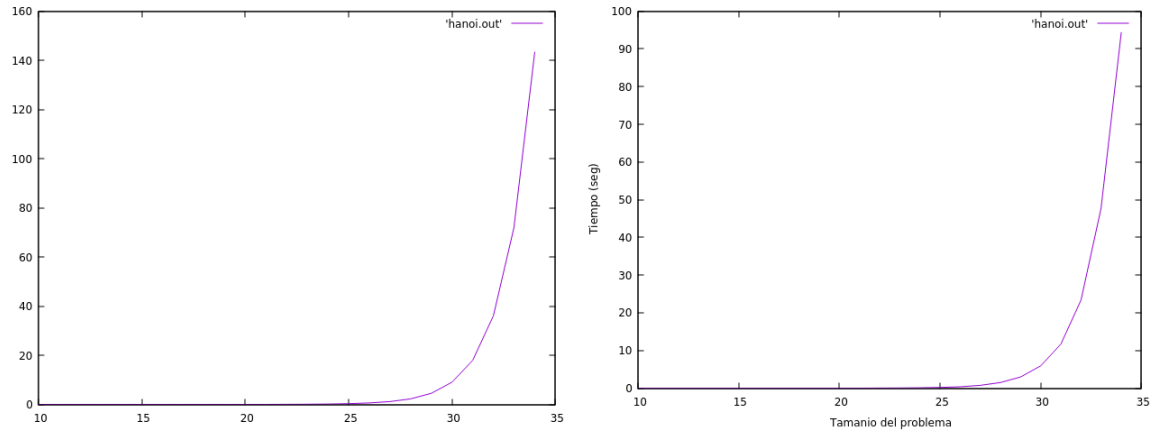


(b) PC 2

Tamaño	**Tiempo PC 1**	**Tiempo PC 2**
t	t1	t2

Figura 17: Algoritmo heapsort

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



(a) PC 1 (b) PC 2

Tamaño	Tiempo PC 1	Tiempo PC 2
t	t1	t2

Figura 18: Algoritmo de Hanoi

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

4. Cálculo de la eficiencia híbrida

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

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

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

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

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