
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 tres métodos:

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

# 2. Código fuente a utilizar

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

## 2.1. Hanoi

```
#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]);

  hanoi(M, 1, 2);

  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;    // Valor del reloj antes de la ejecución
//  clock_t tdespues;  // Valor del reloj después de la ejecución
  int dim;           // Dimensión de la matriz

  //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();
<<<<<<< HEAD:Prácticas/Práctica 1/doc/memoria.tex
//  cout << "Tiempo: " << ((double)(tdespues-tantes))/CLOCKS_PER_SEC << " s" << endl;
=======
//  cout << "Tiempo: " << ((double)(tdespues-tantes))//CLOCKS_PER_SEC
// << " s" << endl;
>>>>>>> 90e22ad6eebc655f85de509ad749c265edeac2f9:Práctica 1/memoria.tex
  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();
    }

  burbuja(T, n);

  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();
    }

  seleccion(T, n);

  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();
    };

  insercion(T, n);

  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();

  heapsort(T, n);

  delete [] T;

  return 0;
};
```

### 2.3.5. *Mergesort*

```cpp
#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.
```

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

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

### 2.3.6. *Quicksort*

```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]);
```

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

## 3. Cálculo de la eficiencia empírica

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.

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

Primer script

```
#!/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
```

Segundo script

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;
```

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

## 3.1. Gráficas comparativas

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

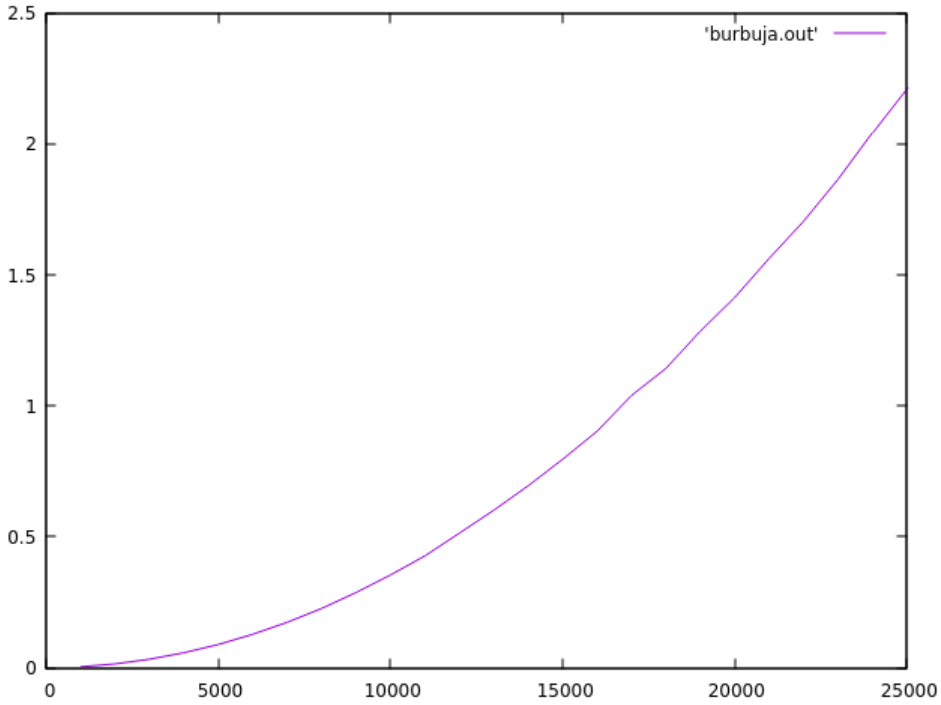


Figura 1: Algoritmo burbuja

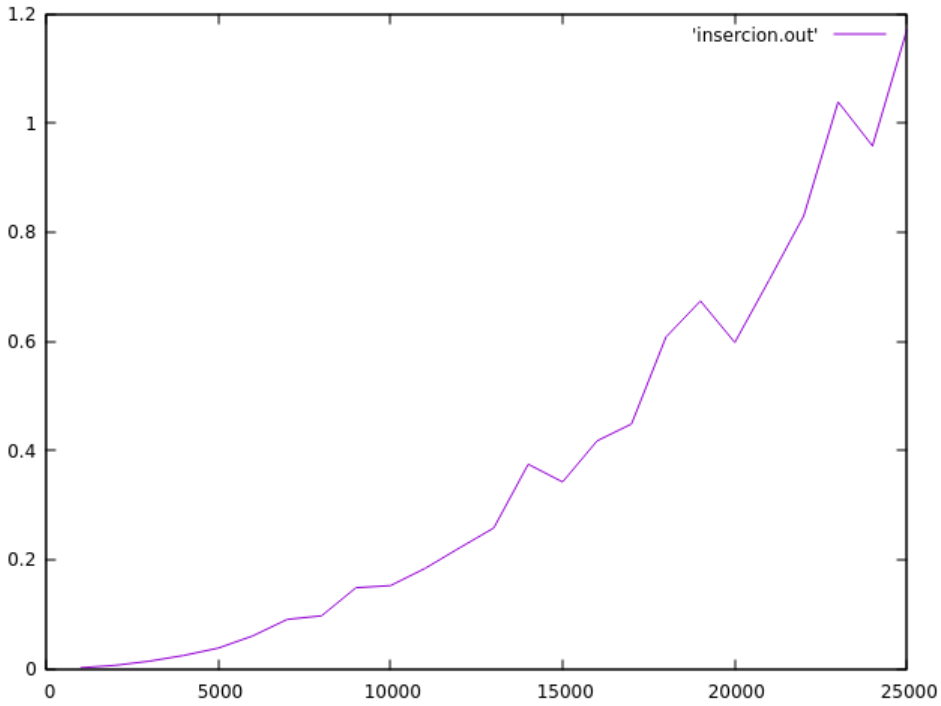


Figura 2: Algoritmo de inserción

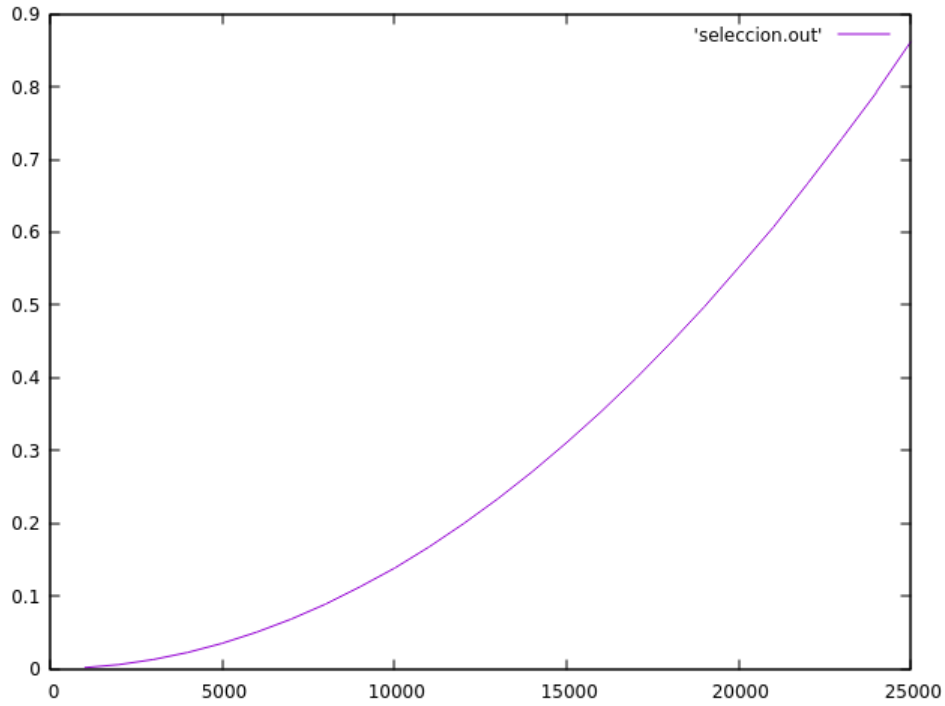


Figura 3: Algoritmo de selección

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

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

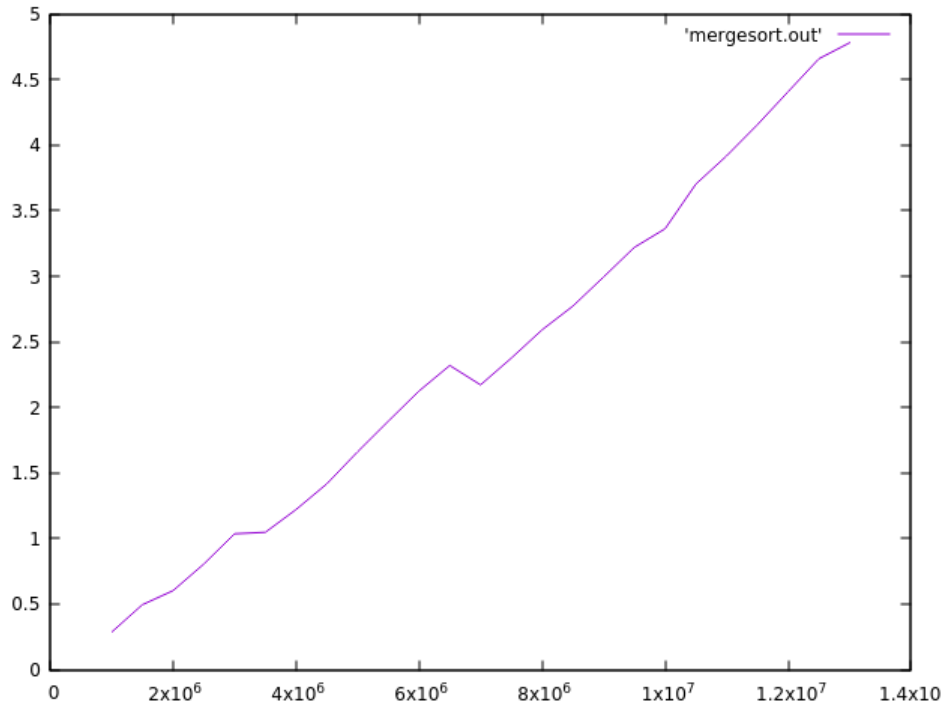


Figura 4: Algoritmo mergesort

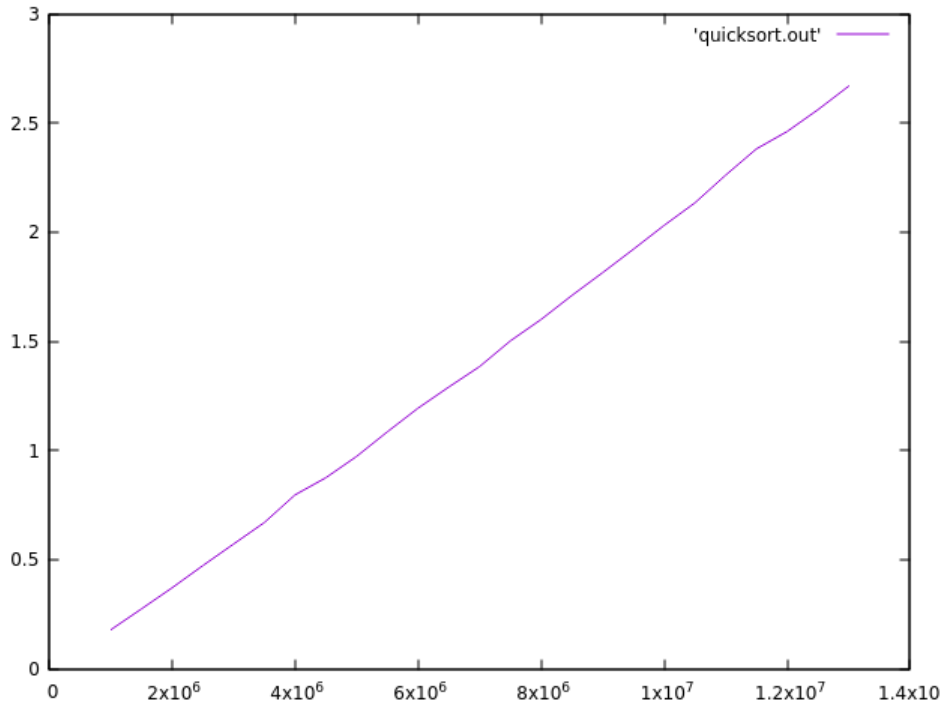


Figura 5: Algoritmo quicksort

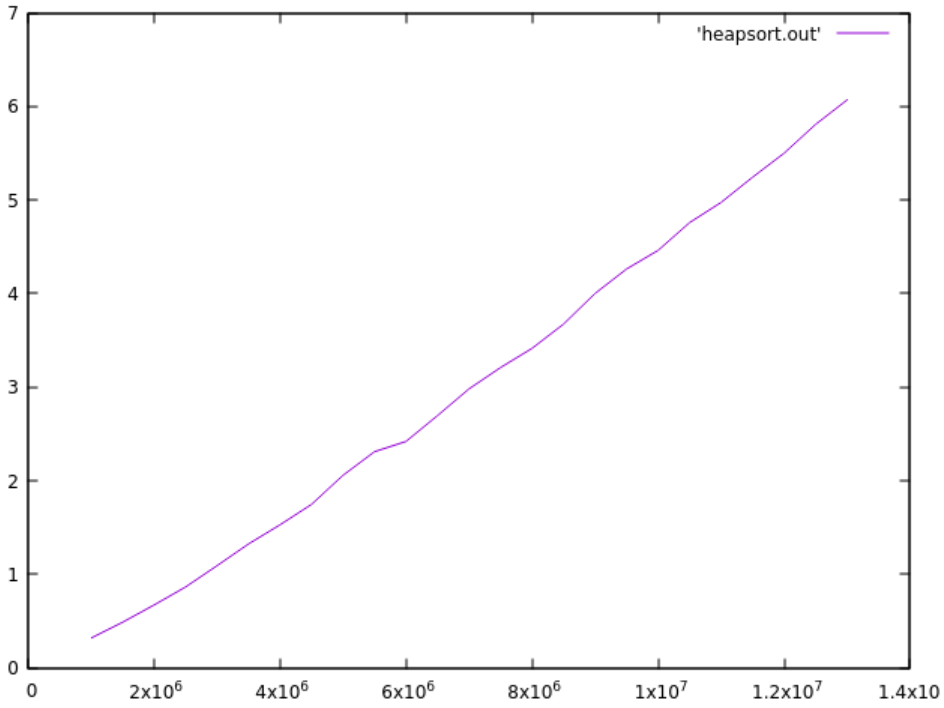


Figura 6: Algoritmo heapsort

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

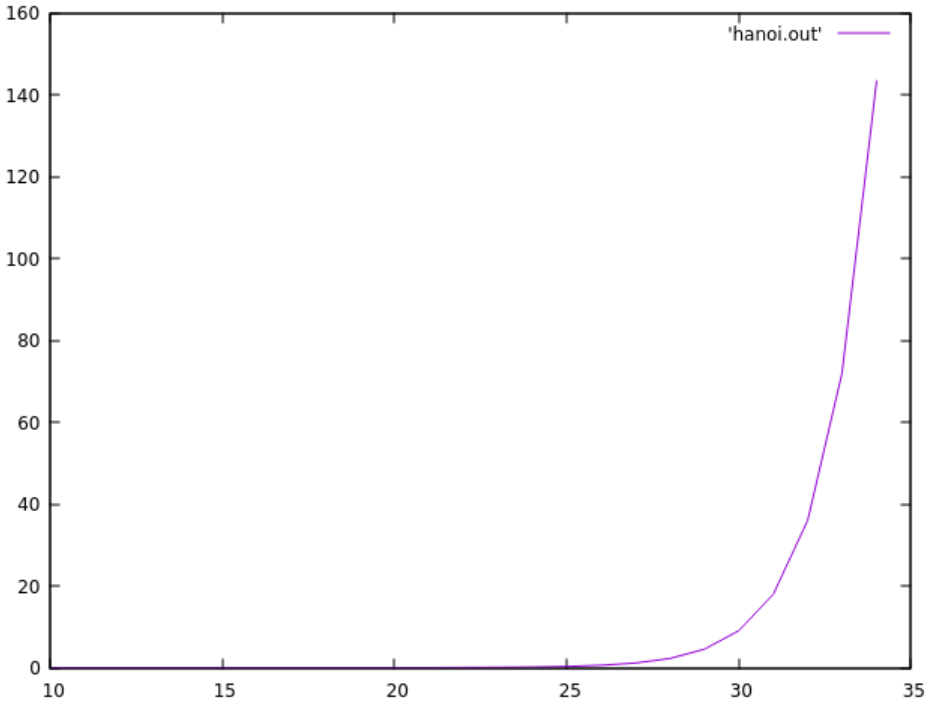


Figura 7: Algoritmo Hanoi

### 3.1.5. 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.

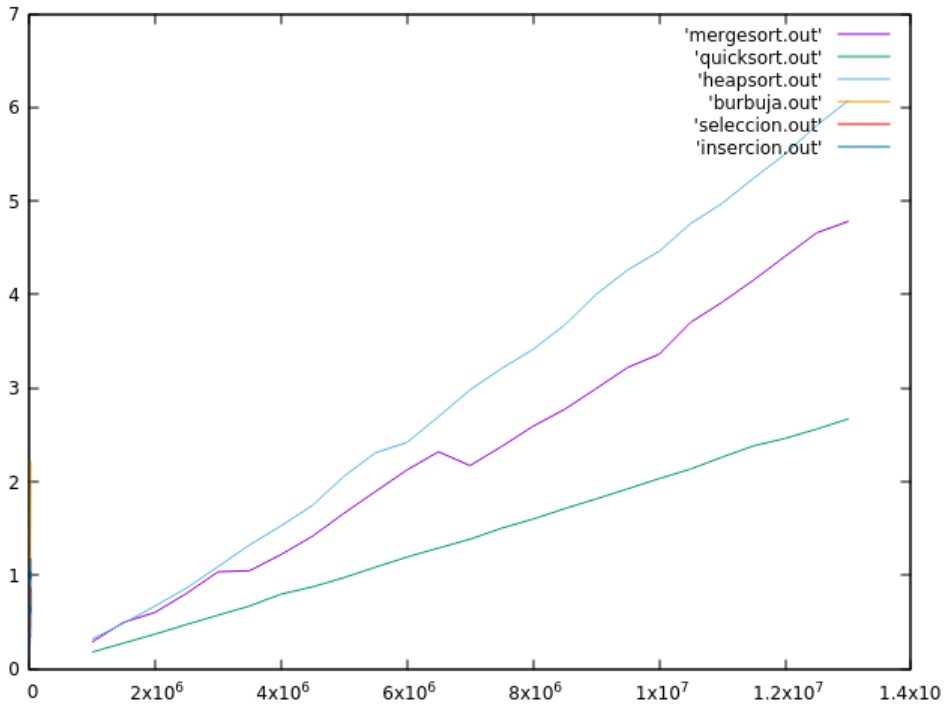


Figura 8: Comparación de algoritmos de ordenación

Si hacemos zoom, podremos ver mejor la diferencia:

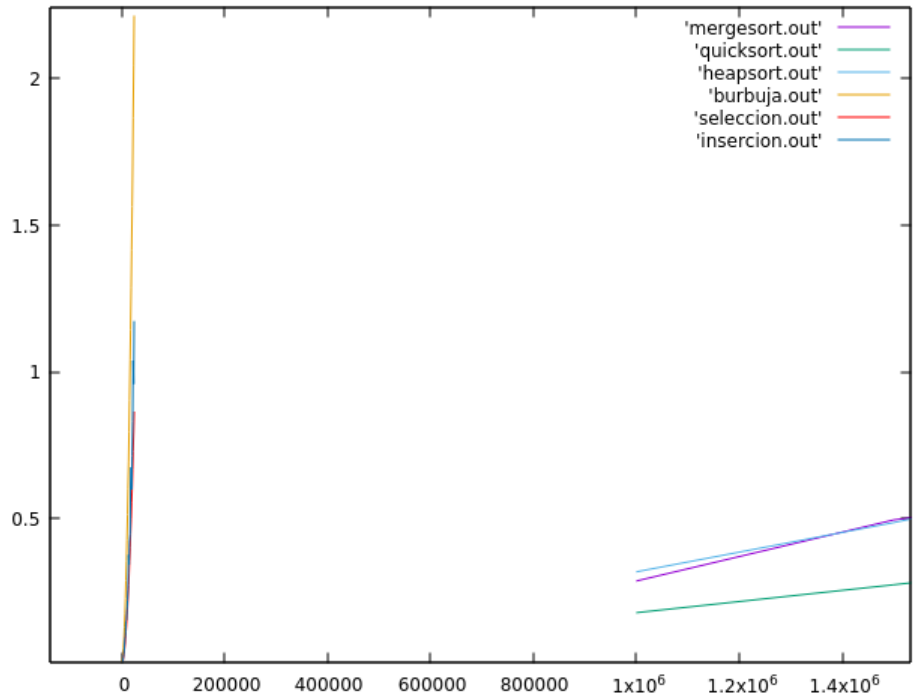


Figura 9: Comparación de algoritmos de ordenación (zoom)

### 3.2. Variación de la eficiencia empírica

# 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 $O(n \cdot log(n))$