Algorítmica

Curso 2023-2024

Grupo Viterbi

PRÁCTICA 1-ANÁLISIS DE EFICIENCIA DE ALGORITMOS

Integrantes:

Miguel Ángel De la Vega Rodríguez	miguevrod@correo.ugr.es
Alberto De la Vera Sánchez	joaquin724@correo.ugr.es
Joaquín Avilés De la Fuente	adelaveras01@correo.ugr.es
Manuel Gomez Rubio	e.manuelgmez@go.ugr.es
Pablo Linari Perez	e.pablolinari@go.ugr.es

Facultad de Ciencias UGR
Escuela Técnica Ingeniería Informática UGR
Granada
2023-2024

Índice

Participación

- **Miguel Ángel De la Vega Rodríguez:** 20%
  - Plantilla y estructura del documento LaTeX
  - Cómputo de la eficiencia de los algoritmos (Resultados y Ajuste)
- **Joaquín Avilés De la Fuente:** 20%
  - Descripción del Objetivo de la pŕactica
  - Diseño del estudio
- **Alberto De la Vera Sánchez:** 20%
- **Manuel Gomez Rubio** 20%
- **Pablo Linari Pérez:** 20%
  - Estudio y comparación de las gráficas
  - Diseño del estudio

Equipo de trabajo

- **Miguel Ángel De la Vega Rodríguez:** (Ordenador donde se ha realizado el computo)
  - AMD Ryzen 7 2700X 8-Core
  - 16 GB RAM DDR4 3200 MHz
  - NVIDIA GeForce GTX 1660 Ti
  - 1 TB SSD NvMe

Objetivos

En esta práctica, se han implementado los siguientes algoritmos de ordenación: **quicksort, mergesort, inserción, burbuja,** y **selección**. Además, se han implementado los algoritmos de **Floyd**, que calcula el costo del camino mínimo entre cada par de nodos de un grafo dirigido, de **Fibonacci**, que calcula los números de la sucesión de Fibonacci , y de **Hanoi**, que resuelve el famoso problema de las torres de Hanoi. Se ha aplicado la siguiente metodología:

- En primer lugar, aunque tenemos la eficiencia teórica de estos algoritmos, se realizarán los calculos necesarios para demostrar como se obtiene dicha eficiencia utilizando los distintos métodos estudiados en teoría.

- En segundo lugar, se pasará al estudio empírico de los algoritmos de ordenación de vectores para distintos tipos de datos, es decir, para datos tipo **int**, **float**, **double** y **string**. Posteriormente, se creará las gráficas para cada algoritmo en las que visualizaremos el tiempo de ejecución en función del tamaño del vector y del tipo de dato. Finalmente para esta parte, se hara un calculo de **eficiencia híbrida** que se basa en ajustar la gráfica obtenida a la función de su eficiencia teórica por mínimos cuadrados, obteniendo por tanto los literales de dicha función que ajustan la gráfica.

- En tercer lugar, se hará el estudio de los otros tres algoritmos de forma similar, es decir, se estudiará la eficiencia de estos de modo empírica, cuyo estudio se mostrará en las gráficas, y se calculará la eficiencia híbrida de estos, a partir de la eficiencia teórica.

Diseño del estudio

Los estudios empíricos han sido realizados en el ordenador con las características mencionadas anteriormente. Además, hemos realizado el estudio empírico de forma aislada para el algoritmo de ordenación de vectores quicksort en los distintos ordenadores de los participantes del grupo para ver como afectan las características hardware de cada ordenador en el tiempo de ejecución, cuyas gráficas se mostrarán en la sección de Algoritmos. En ambos casos se ha hecho uso del sistema operativo Linux, concretamente de Debian, y se ha utilizado el compilador gcc para la compilación de los programas con el flag -Og para la optimización.

4.1. Algoritmos de ordenación de vectores

Para los algoritmos de ordenación se han usado entradas de datos de tipo int, float, double y string mientras que para los algoritmos de Hanoi, Floyd y Fibonnaci solo se han usado entradas de tipo int ya que no tendría sentido usar entradas de otro tipo.

- En los algoritmos con eficiencia $O(n^2)$ como los de Burbuja, Selección e Inserción los saltos usados entre los tipos de datos int, float y double generados aleatoriamente son de 5000 en 5000 empezando con una muestra de 5000 datos y llegando a un máximo de 125000 datos.

- En los lagoritmos con eficiencia $O(n\log(n))$ como el mergesort o el quicksort los saltos usados entre los tipos de datos int, float y double generados aleatoriamente son de 50000 en 50000 empezando con una muestra de 50000 datos y llegando a un máximo de 1250000 datos.

4.2. Otros algoritmos

En los algoritmos restantes se han usado datos de tipo int generados aleatoriamente y proporcionados en la siguiente medida:

- Para el algoritmo de Floyd que es de orden $O(n^3)$ se han usado enteros aleatorios desde 50 hasta 1250 con saltos de 50 en 50.

- Para el algoritmo de Fibonnaci que es de orden $O((\frac{1+\sqrt{5}}{2})^n)$ se han usado enteros aleatorios desde 50 hasta 1250 con saltos de 50 en 50.

- Para el algoritmo de Hanoi que es de orden $O(2^n)$ se han usado enteros aleatorios desde 3 hasta 33 con saltos de 50 en 50.

Por último para el tipo de dato string se han extraido las muestras del archivo *quijote.txt* para simular una generación aleatoria de palabras, esta entrada de datos no ha sido totalmente aleatoria ya que al usar un lenguaje determinado para el texto, en este caso el español, se repiten con mas frecuencia algunas palabras por tanto esto se verá reflejado en el comportamiento de los algoritmos. En este caso el Quijote tiene un total de 202308 palabras por lo que se comenzará con una muestra de 12308 palabras con saltos de 10000 en 10000 hasta llegar a 202308 palabras.

4.3. Scripts usados para la ejecución

- **[AutoCompile.sh]** Este script se encarga de compilar todos los ficheros en una misma carpeta con las mismas opciones de compilación, para garantizar la máxima igualdad posible entre cada algoritmo y organizar la estructura de ficheros.
- **[AutoFinal.sh]** Este script es el encargado de ejecutar todos los algoritmos varias veces con las opciones respectivas para cada uno, el resultado se pasa por un programa AutoMedia.py que se encarga de realizar la media de las ejecuciones de los algoritmos, este resultado es guardado en una carpeta llamada Resultados de la que posteriormente el mismo script genera las graficas de cada algoritmo.
- **[AutoIndividual.sh]** Este script es como el descrito previamente pero unicamente ejecuta un script, esto ha sido útil para hacer pruebas sin la necesidad de esperar la gran cantidad de tiempo que requiere la ejecución de todos los algoritmos.

Algoritmos

Esta sección esta dedicada a mostrar los resultados obtenidos en el estudio de los algoritmos, la estructura seguida para mostrar los resultados consiste en mostrar, para cada algoritmo, los tiempos de ejecución, junto con las gráficas obtenidas y los ajustes correspondientes. Previo a ello, se analizará en cada caso teoricamente la eficiencia prevista para cada algoritmo.

5.1. Estudio teórico

En esta sección se calculará la eficiencia teórica de cada algoritmo, es decir, la eficiencia que se espera al hacer el estudio empírico de cada algoritmo. Para ello, se utilizarán los métodos estudiados en teoría.

Algoritmo de ordenación Burbuja

Utilizaremos el siguiente fragmento de código para estudiar su eficiencia, pues es el utilizado en la práctica

```
1  void burbuja(int T[], int inicial, int final)
2  {
3  int i, j;
4  int aux;
5  for (i = inicial; i < final - 1; i++)
6      for (j = final - 1; j > i; j--)
7      if (T[j] < T[j - 1])
8      {
9          aux = T[j];
10         T[j] = T[j - 1];
11         T[j - 1] = aux;
```

```
12                  }
13              }
```

El trozo de código dentro del bucle interno, es decir, de la línea 7 a la 12 tiene eficiencia $O(1)$ y por tanto tiene un tiempo de ejecución constante que anotaremos como a. Además, este trozo de código se ejecuta en concreto $(final-1)-(i+1)+1$ veces, es decir, $final-i+1$ veces. Es claro que la ejecución de la línea 3 y 4 y las comparaciones, inicializaciones y actualizaciones de los bucles tienen eficiencia $O(1)$. Sabiendo esto y que el número de veces que se ejecute el bucle interno depende del externo tenemos entonces la siguiente fórmula

$$\sum_{i=inicial}^{final-2}\sum_{j=i+1}^{final-1} a$$

Tomaremos $final = n$ e $inicial = 1$ para simplificar el cálculo y veamos que obtenemos ahora

$$\sum_{i=1}^{n-2}\sum_{j=i+1}^{n-1} a = a\cdot\sum_{i=1}^{n-2}\sum_{j=1}^{n-i-1} 1 = a\cdot\sum_{i=1}^{n-2}(n-i-1)a\cdot(n\sum_{i=1}^{n-2}1-\sum_{i=1}^{n-2}i-\sum_{i=1}^{n-2}1) =$$

$$= a\cdot(n(n-2)-\frac{(n-2)(n-1)}{2}-(n-2)) = a\cdot(n^2-2n-\frac{n^2-3n+2}{2}-n+2) =$$

$$= a\cdot(\frac{n^2}{2}-\frac{3}{2}n+1) = \frac{n^2}{2}a-\frac{3}{2}na+a$$

Es claro que $\frac{n^2}{2}a-\frac{3}{2}na+a \in O(n^2)$ y por tanto la eficiencia teórica del algoritmo de burbuja es $O(n^2)$.

Algoritmo de ordenación Inserción

Utilizaremos el siguiente fragmento de código para estudiar su eficiencia, pues es el utilizado en la práctica

```
1   void insercion(int T[], int inicial, int final)
2   {
3     int i, j;
4     int aux;
5     for (i = inicial + 1; i < final; i++) {
6       j = i;
7       while ((T[j] < T[j-1]) && (j > 0)) {
8         aux = T[j];
9         T[j] = T[j-1];
10        T[j-1] = aux;
11        j--;
12      };
13    };
14  }
```

El trozo de código dentro del bucle interno, es decir, de la línea 8 a la 10 tiene eficiencia $O(1)$ y por tanto tiene un tiempo de ejecución constante que anotaremos como a. Dicho trozo de código se ejecutará en el peor de los casos $i-(0-1)+1 = i$ veces, mientras que el bucle while se ejecutará $(final-1)-(inicial+1)+1 = final-inicial-1$ veces. Sabiendo que la ejecución de la línea 3 y 4 y las comparaciones, inicializaciones y actualizaciones de los bucles tienen eficiencia $O(1)$, tenemos la siguiente fórmula

$$\sum_{i=inicial+1}^{final-1}\sum_{j=1}^{i} a$$

Tomaremos $final = n$ e $inicial = 1$ para simplificar el cálculo y veamos que obtenemos ahora

$$\sum_{i=2}^{n-1}\sum_{j=1}^{i} a = a \cdot \sum_{i=1}^{n-2}\sum_{j=1}^{i} 1 = a \cdot \sum_{i=1}^{n-2} i = a \cdot \frac{(n-2)(n-1)}{2} = \frac{n^2}{2}a - \frac{3n}{2}a + a$$

Es claro que $\frac{n^2}{2}a - \frac{3}{2}na + a \in O(n^2)$ y por tanto la eficiencia teórica del algoritmo de inserción es $O(n^2)$.

Algoritmo de ordenación Selección

Utilizaremos el siguiente fragmento de código para estudiar su eficiencia, pues es el utilizado en la práctica

```
1   void seleccion(int T[], int inicial, int final)
2   {
3     int i, j, indice_menor;
4     int menor, aux;
5     for (i = inicial; i < final - 1; i++) {
6       indice_menor = i;
7       menor = T[i];
8       for (j = i; j < final; j++)
9         if (T[j] < menor) {
10          indice_menor = j;
11          menor = T[j];
12        }
13      aux = T[i];
14      T[i] = T[indice_menor];
15      T[indice_menor] = aux;
16    };
17  }
```

El trozo de código dentro del bucle interno, es decir, de la línea 9 a la 14 tiene eficiencia $O(1)$ y por tanto tiene un tiempo de ejecución constante que anotaremos como a. Este trozo de código se ejecutará en el peor de los casos $(final-1)-i+1 = final-i$ veces, mientras que el bucle for interno se ejecutará $(final-1-1)-inicial+1 = final-inicial-1$ veces. Sabiendo que la ejecución de las líneas 3, 4, 6, 7 y las comparaciones, inicializaciones y actualizaciones de los bucles tienen eficiencia $O(1)$, tenemos la siguien fórmula

$$\sum_{i=inicial}^{final-2}\sum_{j=i}^{final-1} a$$

Tomaremos $final = n$ e $inicial = 1$ para simplificar el cálculo y veamos que obtenemos ahora

$$\begin{aligned}\sum_{i=1}^{n-2}\sum_{j=i}^{n-1} a = a \cdot \sum_{i=1}^{n-2}\sum_{j=1}^{n-1-(i-1)} 1 = a \cdot \sum_{i=1}^{n-2}\sum_{j=1}^{n-i} 1 = a \cdot \sum_{i=1}^{n-2} n - i = a \cdot (n\sum_{i=1}^{n-2} 1 - \sum_{i=1}^{n-2} i)\\ = a \cdot (n(n-2) - \frac{(n-2)(n-1)}{2}) = a \cdot (n^2 - 2n - \frac{n^2-3n+2}{2}) = a \cdot (\frac{n^2}{2} - \frac{n}{2} + 1)\\ = \frac{n^2}{2}a - \frac{n}{2}a + a\end{aligned}$$

Es claro que $\frac{n^2}{2}a - \frac{n}{2}a + a \in O(n^2)$ y por tanto la eficiencia teórica del algoritmo de inserción es $O(n^2)$.

Algoritmo de ordenación Mergesort

Utilizaremos el siguiente fragmento de código para estudiar su eficiencia, pues es el utilizado en la práctica

```
1
2         void fusion(int T[], int inicial, int final, int U[], int V[])
3         {
4             int j = 0;
5             int k = 0;
6             for (int i = inicial; i < final; i++){
7                 if (U[j] < V[k]) {
8                     T[i] = U[j];
9                     j++;
10                } else{
11                    T[i] = V[k];
12                    k++;
13                };
14            };
15        }
16
17        void mergesort(int T[], int inicial, int final)
18        {
19            if (final - inicial < UMBRAL_MS){
20                insercion(T, inicial, final);
21            } else {
22                int k = (final - inicial)/2;
23
24                int * U = new int [k - inicial + 1];
25                assert(U);
26                int l, l2;
27                for (l = 0, l2 = inicial; l < k - inicial; l++, l2++)
28                    U[l] = T[l2];
29                U[l] = INT_MAX;
30
31                P * V = new P [final - k + 1];
32                assert(V);
33                for (l = 0, l2 = k; l < final - k; l++, l2++)
34                    V[l] = T[l2];
35                V[l] = INT_MAX;
36
37                mergesort_lims(U, 0, k - inicial);
38                mergesort_lims(V, 0, final - k);
39                fusion(T, inicial, final, U, V);
40                delete [] U;
41                delete [] V;
42            };
43        }
```

Destacar que tomaremos $final = n$ e $inicial = 0$. Es claro que en el caso de $n = final - incial < UMBRAL\_MS$ la eficiencia del algoritmo en el peor caso es $O(UMBRAL\_MS^2)$, es decir, constante, por tanto, nos centraremos en el caso en el que $n \geq UMBRAL\_MS$. En este caso, el algoritmo se divide en dos partes, la primera parte es la creación de los vectores U y V y la segunda parte es la llamada recursiva a la función mergesort y el resto de código.

La primera parte la podemos dividir en dos: la creación del vector U tomando entonces de la línea 22 a la línea 29, donde podemos ver que el bucle for de la línea 27 se ejecuta $\frac{n}{2}$ veces; y la creación del vector V tomando

entonces de la línea 31 a la línea 35, donde tenemos el mismo resultado. Tenemos entonces que ambas partes tienen eficiecia $O(\frac{n}{2})$, es decir, $O(n)$ y aplicando la regla del máximo obtendríamos hasta la línea 35 un «««< HEAD orden de $O(n)$.
En la segund parte, observamos que la llamada recursiva a la función mergesort se hace dos veces con vectores de tamaño $\frac{n}{2}$. Además, viendo la función **fusion** vemos que el bucle for de la línea 6 se ejecuta n veces, es decir, dicha función tiene eficiencia $O(n)$.
Teniendo en cuenta el razonamiento hecho y aplicando la regla de la suma, obtenemos la siguiente ecuación

$$T(n) = 2T(\frac{n}{2}) + n$$

Pasemos ahora a resolver dicha ecuación de recurrencia. Aplicando el siguiente cambio de variable $n = 2^m$ obtenemos

$$T(2^m) = 2T(2^{m-1}) + 2^m \Longrightarrow T(2^m) - 2T(2^{m-1}) = 2^m$$

Resolvamos la parte homógenea de la ecuación, es decir, la ecuación $T(2^m) - 2T(2^{m-1}) = 0$. Obtenemos el polinomio característico de la parte homógenea que es $p_H(x) = x - 2$ cuya raíz es $x = 2$.
Obtengamos ahora la parte no homógenea

$$2^m = b_1^m q_1(m) \Longrightarrow b_1 = 2 \wedge q_1(m) = 1 \text{ con grado } d_1 = 0$$

Tenemos entonces el siguiente polinómio característico

$$p(x) = (x-2)(x-b_1)^{d_1+1} = (x-2)^2$$

Por tanto la solución general es

$$t_m = c_{10}2^m m^0 + c_{11}2^m m^1 \overset{*}{\Longrightarrow} t_n = c_{10}n + c_{11}n\log_2(n) \Longrightarrow T(n) = c_{10}n + c_{11}n\log_2(n)$$

donde en $(*)$ hemos deshecho el cambio de variable
Aplicando la regla del máximo tenemos $T(n) \in O(n\log(n))$

Algoritmo de ordenación quicksort

Para el estudio de eficiencia de este algoritmo hemos usado el siguiente código:

```
1   void quicksort(int T[], int inicial, int final){
2       int k;
3       if (final - inicial < UMBRAL_QS) {
4           insercion(T, inicial, final);
5       } else {
6           dividir_qs(T, inicial, final, k);    <--- O(n)
7
8           //peor caso
9           O(n-1) ---> quicksort(T, inicial, k);   <--- O(n/2)
10          O(1) ---> quicksort(T, k + 1, final);   <--- O(n/2)
11      }
12  }
13
14  void dividir_qs(int T[], int inicial, int final, int & pp){
15      int pivote, aux;
16      int k, l;
17
18      pivote = T[inicial];
```

```
19              k = inicial;
20              l = final;
21              do {
22                  k++;
23              } while ((T[k] <= pivote) && (k < final-1));    <--- O(n)
24              do {
25                  l--;
26              } while (T[l] > pivote);
27              while (k < l) {              <--- O(n)
28                  aux = T[k];
29                  T[k] = T[l];
30                  T[l] = aux;
31                  do k++; while (T[k] <= pivote);
32                  do l--; while (T[l] > pivote);
33              };
34              aux = T[inicial];
35              T[inicial] = T[l];
36              T[l] = aux;
37              pp = l;
38          };
```

Para el estudio de la eficiencia se ha ido estudiando cada método por separado. El método llamado inserción no se tiene en cuenta para la eficiencia ya que solo se usa cuando el problema es de un tamaño menor a UMBRAL_QS. A simple vista es fácil comprobar que el propósito del algoritmo es dividir la ordenación del vector de tamaño original en otros dos de un tamaño más reducido, en el mejor de los casos este será a la mitad si el pivote es justo la mediana. La parte de la llamada recursiva, es $O(\frac{n}{2})$, y la llamada a dividir_qs es $O(n)$, por tanto obtenemos la siguiente expresión:

$$T(n) = 2T(\frac{n}{2}) + n$$

usando el cambio de variable $n = 2^k$ obtenemos:

$$t_k - 2t_{k-1} = 2^k$$

cuyo polinomio característico es:

$$p(x) = (x-2)^2 \Rightarrow t_k = c_1 2^k + c_2 2^k k$$

Finalmente, desacemos el cambio obteniendo:

$$t_n = c_1 n + c_2 n log_2 n \in O(n log_2 n)$$

Donde vemos que el algorimo es $O(nlogn)$.
En el peor caso, lo que ocurre es que el algoritmo no puede establecer un buen pivote, lo que hace que se obtenga la siguiente ecuación:

$$T(n) = T(n-1) + n + 1 = T(n-2) + 2n + 2 = ... = T(n-k) + kn + k$$

tomando $k = n-1$ para llegar al caso base tenemos que:

$$T(n) = T(n-n+1) + (n-1)n + n - 1 = T(1) + (n-1)n + n - 1 \in O(n^2)$$

Donde se ve que en el peor de los casos el algoritmo es $O(n^2)$ que es lo que ocurre con los string por tener un mayor coste de operación, o con los vectores de números ya ordenados o casi ordenados.

Algoritmo de Floyd

Para analizar la eficiencia de este algoritmo se ha usado este fragmento del código:

```
1        void Floyd(int **M, int dim){
2        for (int k = 0; k < dim; k++)
3            for (int i = 0; i < dim;i++)
4                for (int j = 0; j < dim;j++)
5                {
6                int sum = M[i][k] + M[k][j];
7                  M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
8            }
9        }
```

Para el estudio de la eficiencia de este algoritmo es bastante simple ya que a simple vista ya se puede comprobar que es $O(n^3)$ porque hay tres bucles for anidados y cada uno es $O(n)$ por lo que el algoritmo es $O(n^3)$ como se había mencionado.

$$T(n) \in O(n^3)$$

Algoritmo de Fibonacci

Para el la estudio de la eficiencia de este algoritmo se ha usado el siguiente código:

```
1        int fibo(int n){
2            if (n < 2)
3                return 1;
4            else
5                return fibo(n-1) + fibo(n-2);
6        }
```

Estudiando este código se pueden observar dos llamadas recursivas con n-1 y n-2 respectivamente, por lo que si obtenemos la ecuación de la eficiencia del algorimo, al ser la condición del if constante, tenemos que:

$$T(n) = T(n-1) + T(n-2) + 1$$

tomando $T(n) = x^n$ obtenemos:

$$x^n = x^{n-1} + x^{n-2} + 1 \Leftrightarrow x^n - x^{n-1} - x^{n-2} = 1 \Leftrightarrow x^{n-2}(x^2 - x - 1) = 1 = 1^n n^0$$

como $x^{n-2} \neq 0$ tenemos que calcular las raíces del polinomio característico:

$$p(x) = (x - \frac{1+\sqrt{5}}{2})(x - \frac{1-\sqrt{5}}{2})(x-1)$$

de donde obtenemos:

$$t(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n + c_3 1^n \Rightarrow T(n) \in O((\frac{1+\sqrt{5}}{2})^n)$$

Por tanto se comprueba que el algoritmo es $O((\frac{1+\sqrt{5}}{2})^n)$

Algoritmo de Hanoi

Para el la estudio de la eficiencia de este algoritmo se ha usado el siguiente código:

```
1              void hanoi (int M, int i, int j){
2                   if (M > 0)
3                       {
4                       hanoi(M-1, i, 6-i-j);
5                       hanoi (M-1, 6-i-j, j);
6                   }
7              }
```

Vemos que se llama recursivamente dos veces a la función hanoi con M-1, y sabiendo que la eficiencia del if es constante tenemos la siguiente ecuación de recurrencia:

$$T(n) = 2T(n-1) + 1 \wedge T(0) = 1$$

Destacar que hemos tomado n=M. Resolviendo la ecuación de recurrencia obtenemos:

$$T(n) = 2T(n-1) + 1 = 2(2T(n-2) + 1) + 1 = 2^2T(n-2) + 3 = 2(2^2T(n-3) + 3) + 1 = 2^3T(n-3) + 7 =$$
$$= ... = 2^kT(n-k) + 2k - 1$$

Habiendo obtenido el caso general, veamos que obtenemos con $k = n$

$$T(n) = 2^nT(n-n) + 2n - 1 = 2^nT(0) + 2n - 1 = 2^n + 2n - 1 \Longrightarrow T(n) \in O(2^n)$$

Por tanto se comprueba que el algoritmo es $O(2^n)$

5.2. Estudio empírico

Burbuja

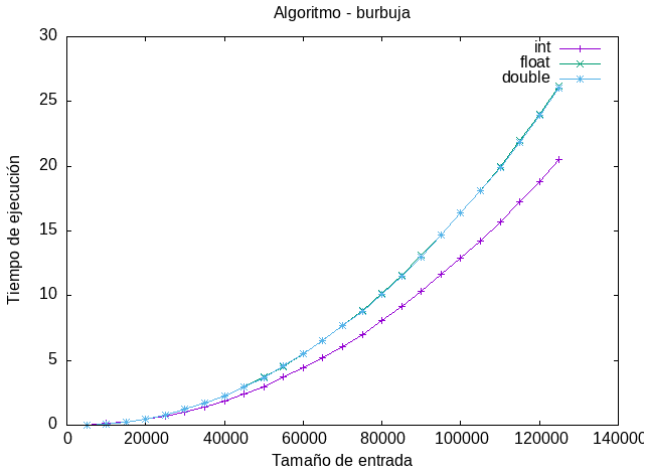

Figura 1: Ejecución algoritmo burbuja

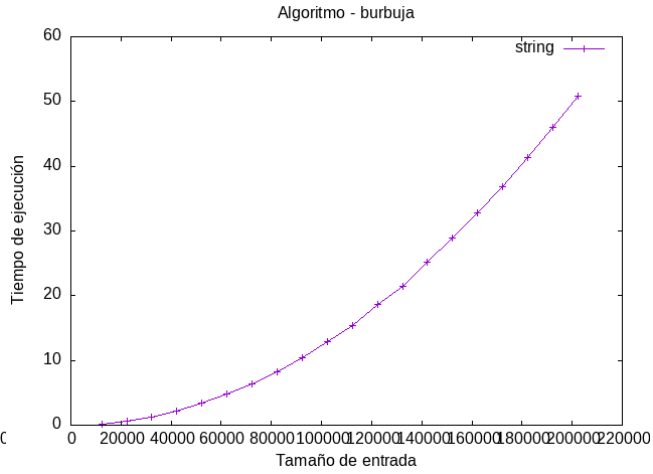

Figura 2: Ejecución algoritmo burbuja con string

BURBUJA INT	
ENTRADA	TIEMPO
5000	0,0349211
10000	0,120458
15000	0,245185
20000	0,441264
25000	0,683406
30000	1,04181
35000	1,38162
40000	1,84603
45000	2,38588
50000	2,98525
55000	3,69388
60000	4,39556
65000	5,17327
70000	6,06058
75000	7,0144
80000	8,04692
85000	9,15881
90000	10,3159
95000	11,6496
100000	12,8843
105000	14,2458
110000	15,7108
115000	17,2545
120000	18,8423
125000	20,5122

BURBUJA FLOAT	
ENTRADA	TIEMPO
5000	0,0217027
10000	0,102747
15000	0,258293
20000	0,492039
25000	0,812521
30000	1,20884
35000	1,69518
40000	2,27424
45000	2,93436
50000	3,69289
55000	4,53819
60000	5,52096
65000	6,56449
70000	7,6747
75000	8,89351
80000	10,2096
85000	11,6127
90000	13,1071
95000	14,6642
100000	16,3917
105000	18,1445
110000	19,9994
115000	21,9713
120000	24,0176
125000	26,1758

BURBUJA DOUBLE	
ENTRADA	TIEMPO
5000	0,021847
10000	0,101877
15000	0,25446
20000	0,49134
25000	0,802523
30000	1,22906
35000	1,69937
40000	2,25218
45000	2,91507
50000	3,66562
55000	4,58012
60000	5,49872
65000	6,50195
70000	7,66039
75000	8,79687
80000	10,1098
85000	11,5094
90000	12,9876
95000	14,7154
100000	16,3819
105000	18,1415
110000	19,9348
115000	21,8762
120000	23,9177
125000	26,0694

BURBUJA STRING	
ENTRADA	TIEMPO
12308	0,211221
22308	0,615473
32308	1,28107
42308	2,1949
52308	3,34899
62308	4,76194
72308	6,41449
82308	8,29372
92308	10,4296
102308	12,827
112308	15,4556
122308	18,6297
132308	21,4492
142308	25,1843
152308	28,8384
162308	32,7271
172308	36,8875
182308	41,3668
192308	46,0167
202308	50,8681

Seleccion

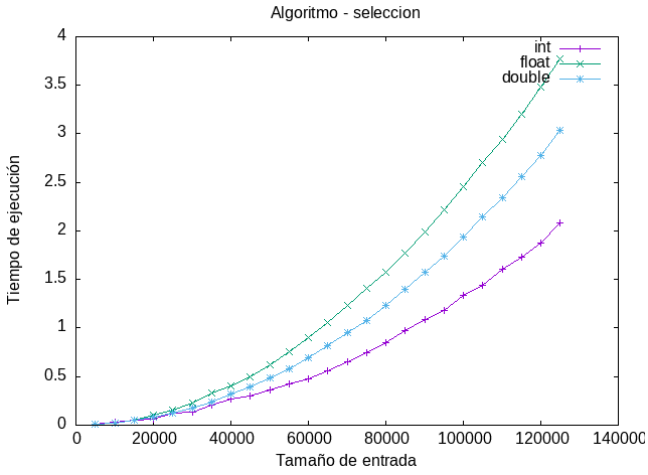

Figura 3: Ejecución algoritmo seleccion

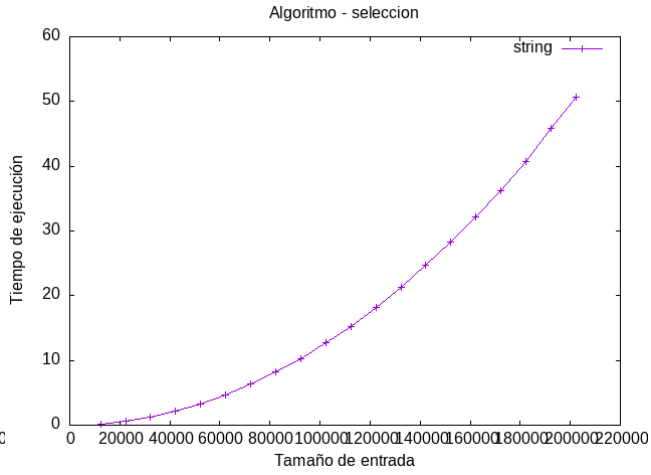

Figura 4: Ejecución algoritmo seleccion con string

SELECCION INT	
ENTRADA	TIEMPO
5000	0,00742912
10000	0,028543
15000	0,0552547
20000	0,0576667
25000	0,119985
30000	0,136593
35000	0,211097
40000	0,266688
45000	0,303022
50000	0,365775
55000	0,425585
60000	0,481344
65000	0,560598
70000	0,648472
75000	0,745198
80000	0,851962
85000	0,970841
90000	1,0899
95000	1,18203
100000	1,33268
105000	1,44115
110000	1,60109
115000	1,73173
120000	1,87346
125000	2,07954

SELECCION FLOAT	
ENTRADA	TIEMPO
5000	0,00656798
10000	0,0255079
15000	0,0567478
20000	0,1013
25000	0,155952
30000	0,22519
35000	0,3326
40000	0,399363
45000	0,50144
50000	0,622559
55000	0,752239
60000	0,902525
65000	1,0589
70000	1,23376
75000	1,40422
80000	1,5725
85000	1,77537
90000	1,99127
95000	2,21625
100000	2,45637
105000	2,70612
110000	2,94571
115000	3,19926
120000	3,47832
125000	3,77295

SELECCION DOUBLE	
ENTRADA	TIEMPO
5000	0,00592686
10000	0,021291
15000	0,0473479
20000	0,079274
25000	0,123864
30000	0,17523
35000	0,24262
40000	0,318302
45000	0,393314
50000	0,488219
55000	0,583393
60000	0,696681
65000	0,815409
70000	0,951775
75000	1,08181
80000	1,22849
85000	1,39556
90000	1,57202
95000	1,74023
100000	1,94182
105000	2,14333
110000	2,34173
115000	2,55887
120000	2,77209
125000	3,03284

SELECCION STRING	
ENTRADA	TIEMPO
12308	0,183608
22308	0,60529
32308	1,26893
42308	2,17589
52308	3,32206
62308	4,71574
72308	6,34248
82308	8,21518
92308	10,3325
102308	12,6803
112308	15,3029
122308	18,1543
132308	21,2342
142308	24,7701
152308	28,326
162308	32,1693
172308	36,2288
182308	40,7869
192308	45,8192
202308	50,7106

Inserción

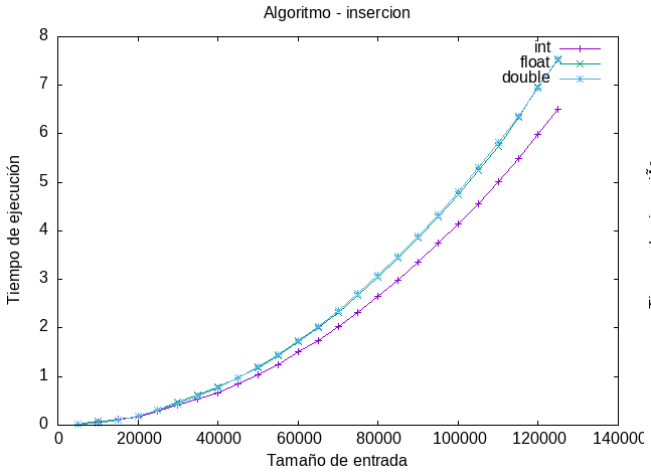

Figura 5: Ejecución algoritmo insercion

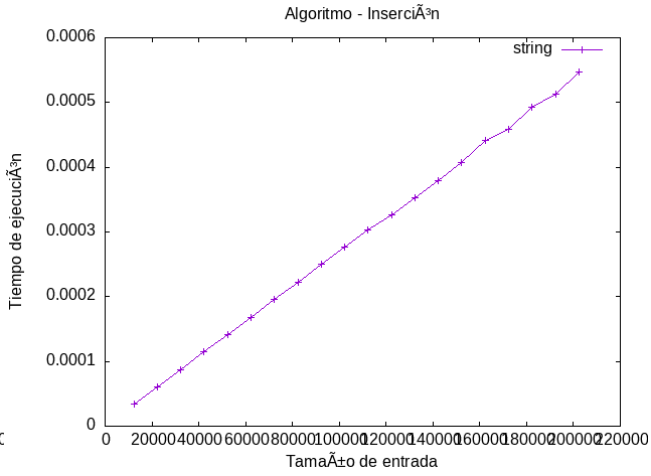

Figura 6: Ejecución algoritmo inserción con string

INSERCION INT	
ENTRADA	TIEMPO
5000	0,0202083
10000	0,0669288
15000	0,118554
20000	0,171463
25000	0,290236
30000	0,4179
35000	0,546578
40000	0,664381
45000	0,844561
50000	1,03853
55000	1,25102
60000	1,51964
65000	1,7472
70000	2,03282
75000	2,32695
80000	2,64902
85000	2,99459
90000	3,35945
95000	3,74271
100000	4,14988
105000	4,56802
110000	5,01684
115000	5,48697
120000	5,98221
125000	6,51173

INSERCION FLOAT	
ENTRADA	TIEMPO
5000	0,0231026
10000	0,083436
15000	0,109163
20000	0,194157
25000	0,303104
30000	0,474516
35000	0,611683
40000	0,788565
45000	0,967047
50000	1,1911
55000	1,43977
60000	1,71672
65000	2,00781
70000	2,32859
75000	2,67209
80000	3,04303
85000	3,43543
90000	3,8531
95000	4,28721
100000	4,75486
105000	5,23354
110000	5,74715
115000	6,3353
120000	6,95729
125000	7,53129

INSERCION DOUBLE	
ENTRADA	TIEMPO
5000	0,0123225
10000	0,0489249
15000	0,10909
20000	0,194458
25000	0,303342
30000	0,437655
35000	0,591596
40000	0,772917
45000	0,97772
50000	1,20692
55000	1,45909
60000	1,73629
65000	2,03246
70000	2,35687
75000	2,70622
80000	3,0808
85000	3,47801
90000	3,90283
95000	4,34073
100000	4,81364
105000	5,30147
110000	5,8236
115000	6,36572
120000	6,93946
125000	7,54045

INSERCION STRING	
ENTRADA	TIEMPO
12308	3,45E-05
22308	6,10E-05
32308	8,77E-05
42308	0,000114708
52308	0,000141829
62308	0,000168388
72308	0,000195308
82308	0,000222477
92308	0,000249857
102308	0,000276907
112308	0,000303237
122308	0,000326027
132308	0,000353216
142308	0,000379146
152308	0,000406996
162308	0,000441945
172308	0,000459225
182308	0,000492385
192308	0,000513575
202308	0,000547894

Mergesort

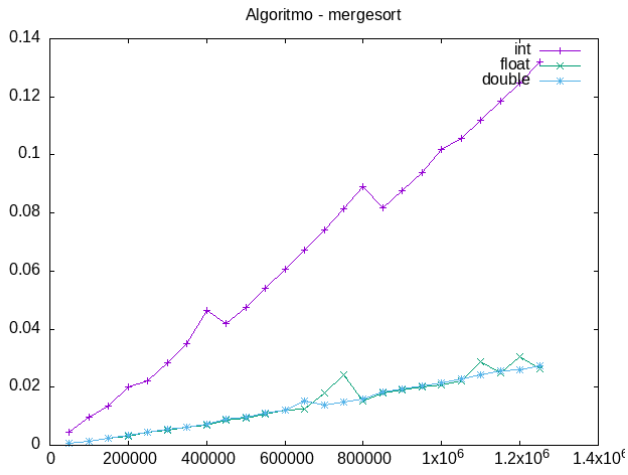

Figura 7: Ejecución algoritmo insercion

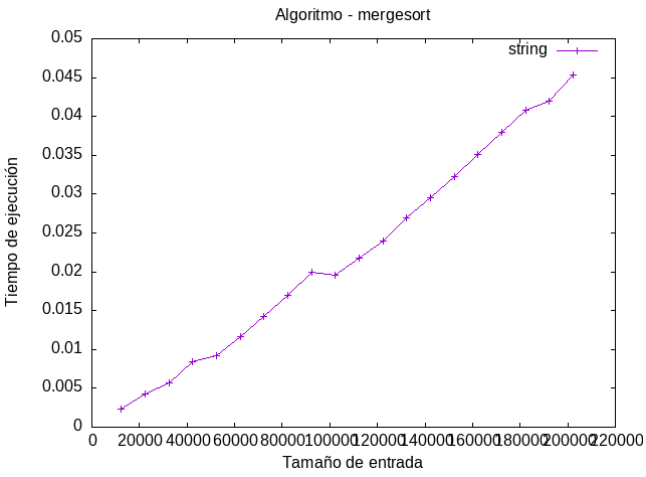

Figura 8: Ejecución algoritmo inserción con string

MERGESORT INT	
ENTRADA	TIEMPO
50000	0,00885563
100000	0,0197725
150000	0,0256754
200000	0,0396519
250000	0,0408458
300000	0,0487663
350000	0,0550213
400000	0,0522138
450000	0,0539998
500000	0,0567018
550000	0,0649299
600000	0,0730137
650000	0,0765446
700000	0,0783553
750000	0,0801539
800000	0,0898059
850000	0,0904214
900000	0,0957652
950000	0,102922
1000000	0,10751
1050000	0,114045
1100000	0,117457
1150000	0,124595
1200000	0,130175
1250000	0,137581

MERGESORT FLOAT	
ENTRADA	TIEMPO
50000	0,00535974
100000	0,0113207
150000	0,0159655
200000	0,0238364
250000	0,0263662
300000	0,0338221
350000	0,0412734
400000	0,0473443
450000	0,0587599
500000	0,0631239
550000	0,0721455
600000	0,078831
650000	0,0864586
700000	0,095748
750000	0,1006187
800000	0,105794
850000	0,11503
900000	0,115988
950000	0,119324
1000000	0,126549
1050000	0,133461
1100000	0,140533
1150000	0,147092
1200000	0,155134
1250000	0,160264

MERGESORT DOUBLE	
ENTRADA	TIEMPO
50000	0,0109506
100000	0,0229505
150000	0,032702
200000	0,0470055
250000	0,0465912
300000	0,045297
350000	0,0516928
400000	0,0576073
450000	0,0581018
500000	0,0668968
550000	0,0680914
600000	0,0821479
650000	0,0840685
700000	0,107731
750000	0,110418
800000	0,113999
850000	0,115276
900000	0,118358
950000	0,125308
1000000	0,130252
1050000	0,138512
1100000	0,145543
1150000	0,147466
1200000	0,161852
1250000	0,175689

MERGESORT STRING	
ENTRADA	TIEMPO
12308	0,00232464
22308	0,00426666
32308	0,00570479
42308	0,00839818
52308	0,00923186
62308	0,0115959
72308	0,0142777
82308	0,0169875
92308	0,0199003
102308	0,0195291
112308	0,0217157
122308	0,0240279
132308	0,0269058
142308	0,0295147
152308	0,0322954
162308	0,0350564
172308	0,0379305
182308	0,040835
192308	0,041929
202308	0,0454001

Quicksort

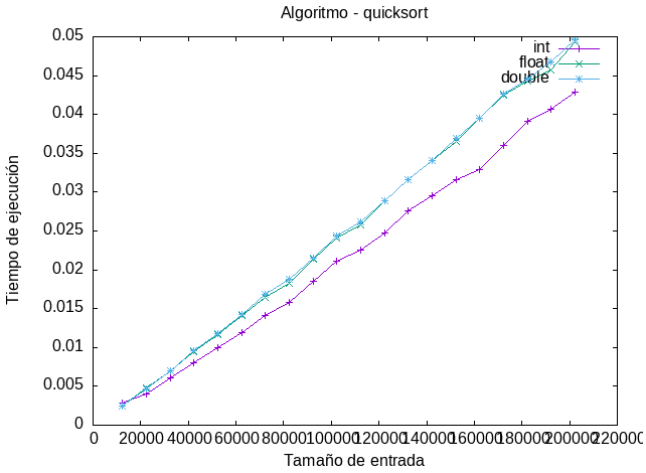

Figura 9: Ejecución algoritmo insercion

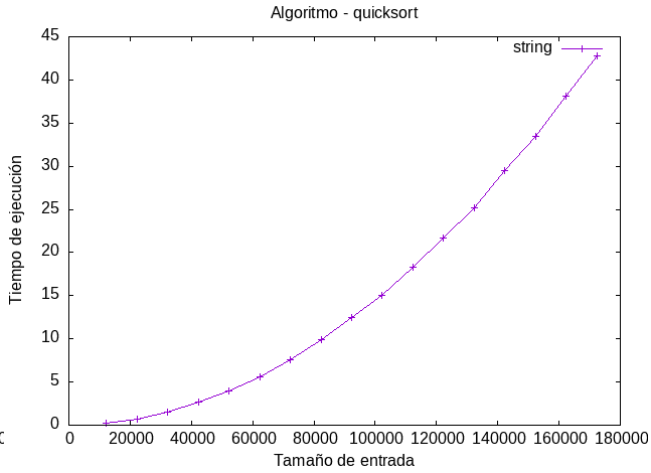

Figura 10: Ejecución algoritmo inserción con string

QUICKSORT INT	
ENTRADA	TIEMPO
50000	0,0054040633
100000	0,0118495667
150000	0,0175899667
200000	0,0238955
250000	0,0302456667
300000	0,0351111667
350000	0,0382649333
400000	0,0422403
450000	0,0438669667
500000	0,04426
550000	0,0485150333
600000	0,0534733667
650000	0,0581094667
700000	0,0614056333
750000	0,0639220667
800000	0,0658048667
850000	0,0690738333
900000	0,0731824333
950000	0,0761217
1000000	0,0791737667
1050000	0,0828587333
1100000	0,0841312
1150000	0,0922743
1200000	0,0955899667
1250000	0,0969886333

QUICKSORT FLOAT	
ENTRADA	TIEMPO
50000	0,00656497
100000	0,0139409333
150000	0,0218207667
200000	0,0290163
250000	0,0369301667
300000	0,0380422
350000	0,0381787667
400000	0,0424919
450000	0,0473254
500000	0,0484824667
550000	0,0545023667
600000	0,0560969
650000	0,0597359333
700000	0,0630458667
750000	0,0668639
800000	0,0677411333
850000	0,0738302667
900000	0,0779973333
950000	0,0811925667
1000000	0,0857580667
1050000	0,0920956
1100000	0,0974753333
1150000	0,100609
1200000	0,1046883333
1250000	0,109222

QUICKSORT DOUBLE	
ENTRADA	TIEMPO
50000	0,0068726067
100000	0,0145315667
150000	0,0226185333
200000	0,0309204333
250000	0,0365530333
300000	0,0399214333
350000	0,0424273
400000	0,0433774333
450000	0,0462596333
500000	0,0489615333
550000	0,0520081333
600000	0,0545633333
650000	0,0557539333
700000	0,0581116333
750000	0,0626529333
800000	0,0661678333
850000	0,0728078333
900000	0,0773843
950000	0,0808991
1000000	0,0875709333
1050000	0,0956117333
1100000	0,0987883667
1150000	0,1001157667
1200000	0,1031563333
1250000	0,10996

QUICKSORT STRING	
ENTRADA	TIEMPO
12308	0,331982
22308	0,433525
32308	0,810498
42308	1,16745
52308	1,97952
62308	2,74097
72308	3,55831
82308	4,67599
92308	5,73606
102308	7,04456
112308	8,1903
122308	10,0859
132308	11,609
142308	13,1231
152308	15,4072
162308	17,1388
172308	19,0906

Hanoi

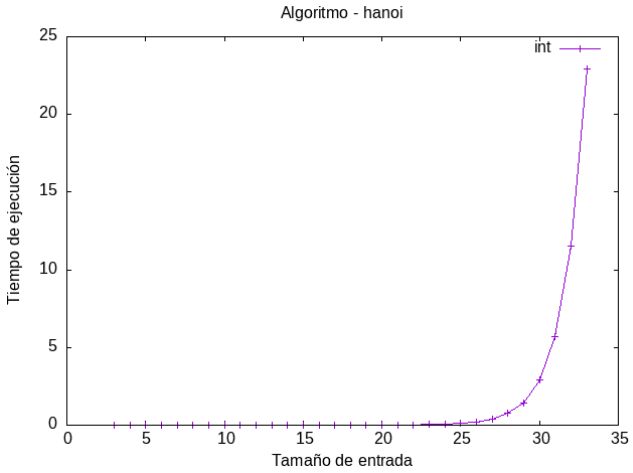

Figura 11: Ejecución algoritmo Hanoi

HANOI	
ENTRADA	TIEMPO
3	3.3e-07
4	3.7e-07
5	6.4e-07
6	9.1e-07
7	1.31e-06
8	2.13e-06
9	3,66E-03
10	6,81E-03
11	9.15e-06
12	2,48E-02
13	4,30E-01
14	9,10E-01
15	180.597
16	370.175
17	728.411
18	143.246
19	283.343
20	583.306
21	113.207
22	23.259
23	446.832
24	678.575
25	115.696
26	205.652
27	387.235
28	757.241
29	145.715
30	291.489
31	573.008
32	11.5402
33	229.485

5.3. Fibonacci

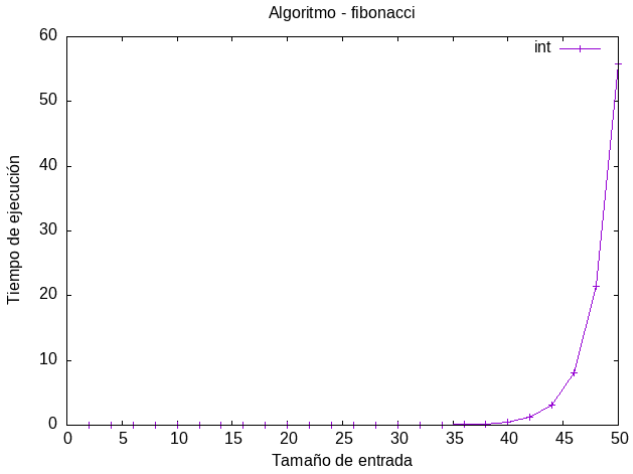

Figura 12: Ejecución algoritmo Fibonacci

FIBONNACCI	
ENTRADA	TIEMPO
2	1,90E-07
4	2,50E-07
6	3,10E-07
8	6,20E-07
10	1,13E-06
12	2,11E-06
14	4,42E-06
16	9,70E-06
18	2,34E-05
20	5,86E-05
22	0,000155419
24	0,000392447
26	0,00103918
28	0,00269092
30	0,00702187
32	0,0188891
34	0,0493608
36	0,101751
38	0,173823
40	0,466558
42	1,23353
44	3,14718
46	8,09745
48	21,4575
50	55,7656

5.4. Floyd

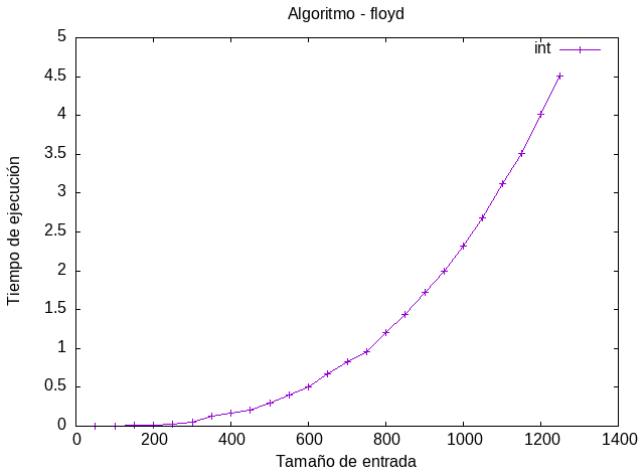

Figura 13: Ejecución algoritmo Floyd

FLOYD	
ENTRADA	TIEMPO
50	0,000317337
100	0,00221242
150	0,006823
200	0,0089683
250	0,0301016
300	0,0513097
350	0,133839
400	0,174386
450	0,207949
500	0,297192
550	0,399401
600	0,509953
650	0,667422
700	0,824563
750	0,956399
800	1,19819
850	1,4429
900	1,71932
950	1,99518
1000	2,31279
1050	2,67816
1100	3,11671
1150	3,51038
1200	4,01139
1250	4,50535

Estudio de las gráficas

En esta sección se mostrarán las gráficas obtenidas en el estudio empírico de los algoritmos.

6.1. Algoritmos $O(n^2)$

Comenzaremos comparando las gráficas obtenidas para los algoritmos de ordenación con eficiencia $O(n^2)$

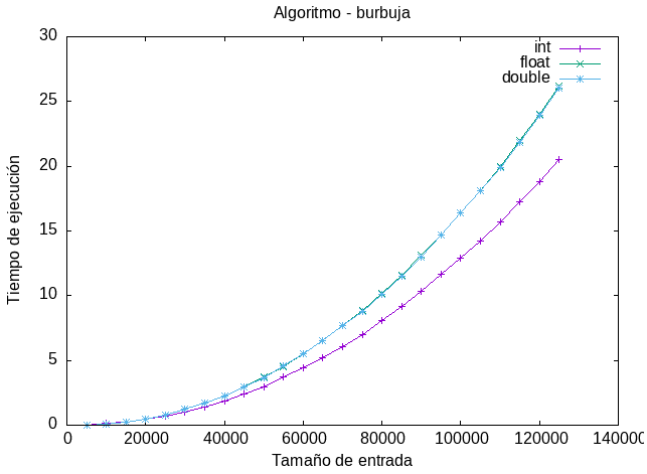

Figura 14: Ejecución algoritmo burbuja

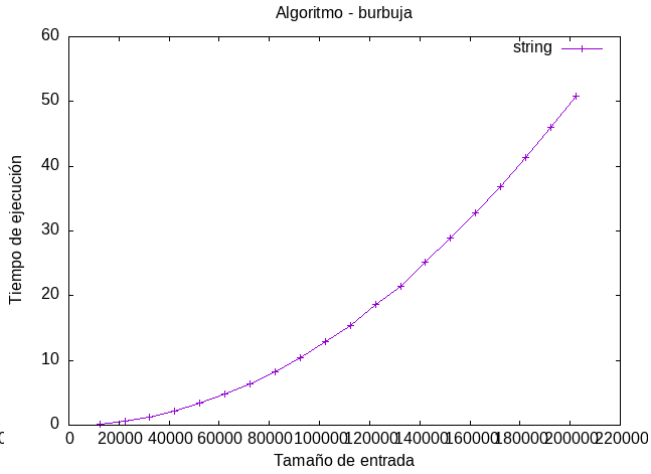

Figura 15: Ejecución algoritmo burbuja con string

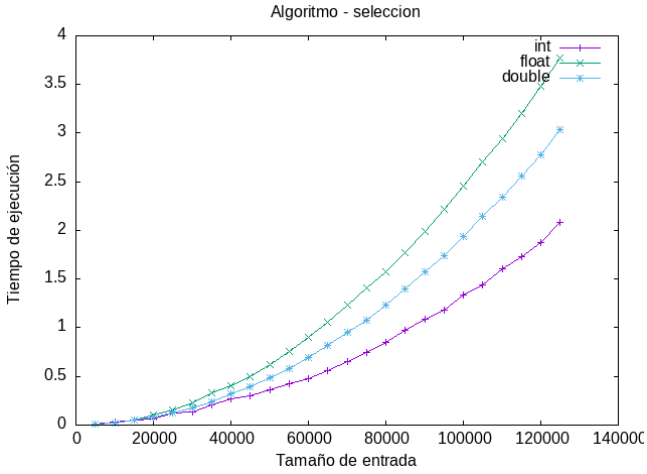

Figura 16: Ejecución algoritmo seleccion

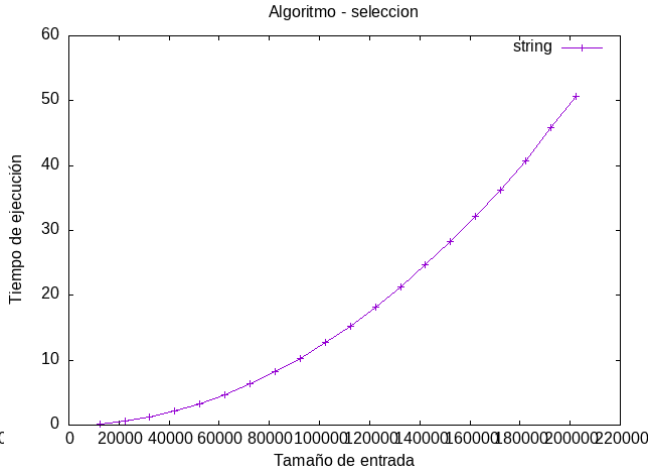

Figura 17: Ejecución algoritmo seleccion con string

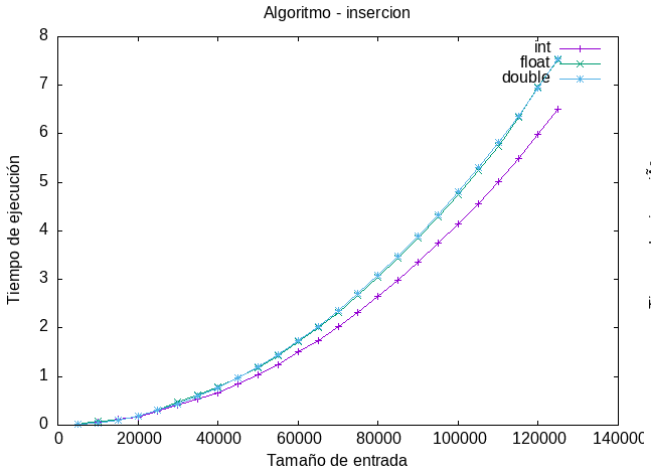

Figura 18: Ejecución algoritmo insercion

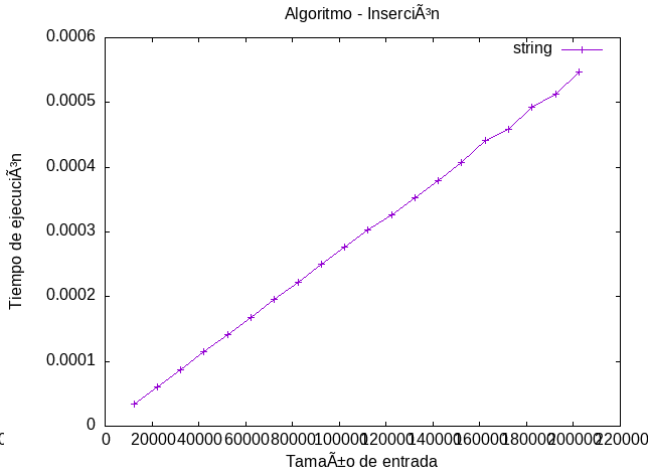

Figura 19: Ejecución algoritmo inserción con string

Comenzaremos analizando primero los casos con tipos de datos int, float y double. Si nos fijamos en los tiempos de ejecución de cada algoritmo podemos ver que el algoritmo de burbuja es el que peor se comporta en todos los casos, seguido del algoritmo de Inserción y finalmente el algoritmo de selección. Dejando asi el algoritmo de burbuja como el peor de los tres y el de Inserción como el mejor. En el algoritmo de burbuja y de insercion se puede ver que el tiempo de ejecución es muy similar en todos los casos llegando a ser practicamente el mismo en los casos con datos double y float . mientras que el algoritmo de seleccion si se ve mas afectado por el tipo de dato que se le pasa siendo los datos int los mas rápidos y los datos float los mas lentos.

si nos fijamos en las graficas de los algoritmos con string podemos ver que los algoritmos de burbuja y seleccion son los que peor se comportan ya que se usa com entrada el libro del quijote en español lo cual hace que haya muchas palabras repetidas y por tanto el algoritmo de burbuja y seleccion tengan que hacer mas comparaciones y por tanto mas tiempo de ejecución. En el caso del algoritmo de inserción vemos que esto le favorece y su tiempo de ejecución se reduce drasticamente en comparación con los datos int, float y double.

6.2. Algoritmos $O(n\log(n))$

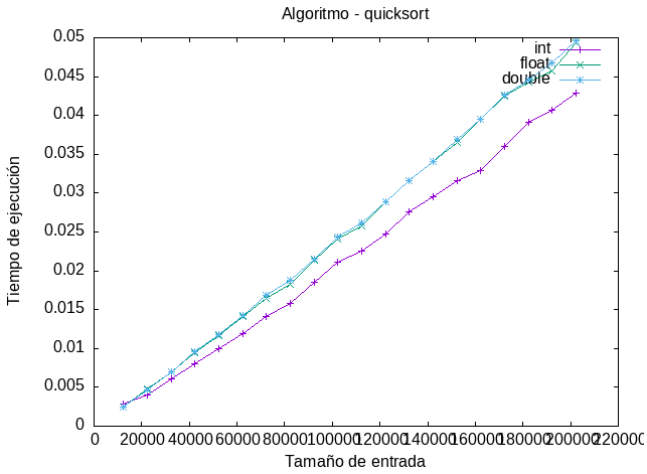

Figura 20: Ejecución algoritmo insercion

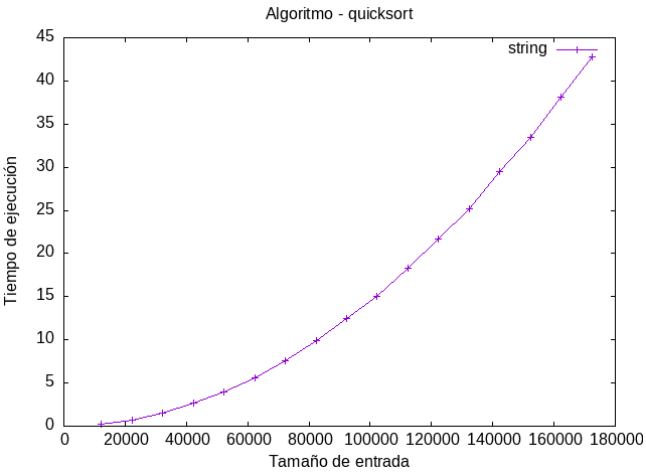

Figura 21: Ejecución algoritmo inserción con string

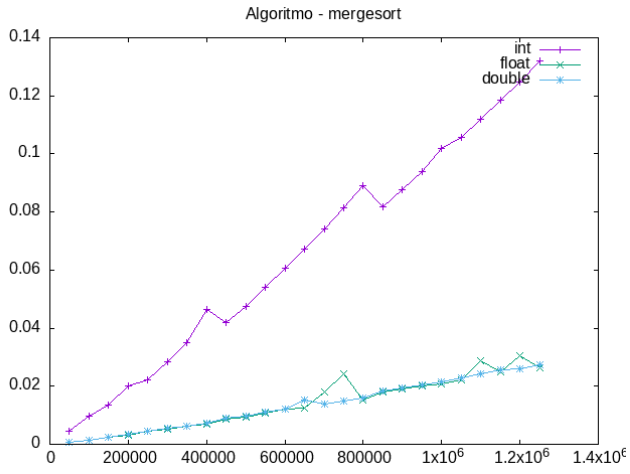

Figura 22: Ejecución algoritmo mergesort

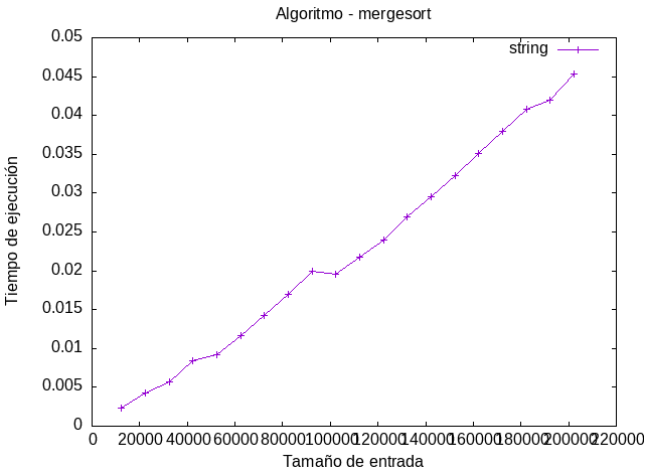

Figura 23: Ejecución algoritmo mergesort con string

Pasamos ahora a estudiar los algoritmos con eficiencia $O(n\log(n))$ los cuales veremos que son mas eficientes que los $O(n^2)$ Si nos fijamos em la gráfica del quicksort veos que no hay casi diferencia entre los datos tipo float y double mientras que los datos tipo int son mas rapidos, en el mergesort pasa justamente lo contrario , los tipos de datos double y float tardan menos en ser ordenados que los datos tipo int . pero ambos son mas eficientes que los anteriormente vistos .

Si nos fijamos en las graficas de los algoritmos cuando los ejecutamos con datos de tipo string vemos que el mergesort gana en tiempo de ejecución al quicksort ya que en los casos donde hay datos repetidos el mergesort se comporta mejor que el quicksort debido a su implementacion.

6.3. Algoritmos Hanoi , Floyd y Fibbonaci

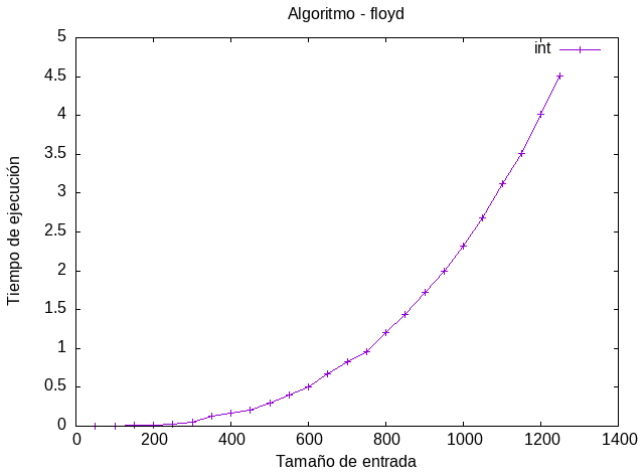

Figura 24: Ejecución algoritmo mergesort

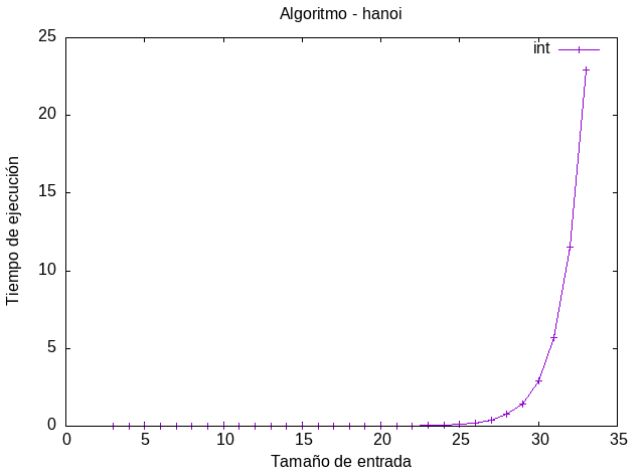

Figura 25: Ejecución algoritmo mergesort

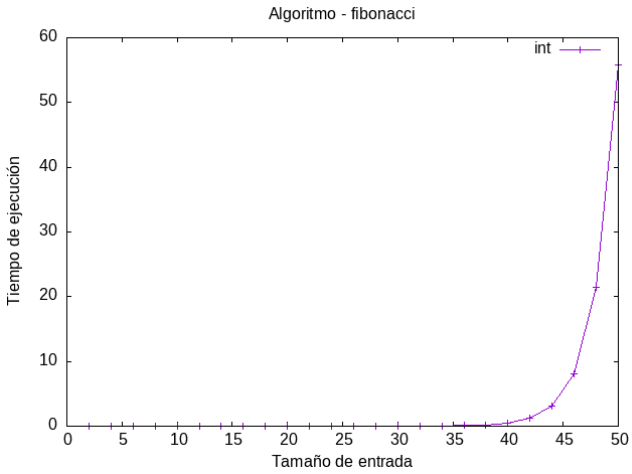

Figura 26: Ejecución algoritmo mergesort

Por último pasamos a estudiar los algoritmos de Hanoi, Floyd y Fibonacci. Si nos fijamos en las gráficas de estos algoritmos vemos que el algoritmo de Floyd es el mas rapido ya que se trata de una algoritmo del orden $O(n^3)$ por tanto supera en velocidad a el algoritmo de Hanoi que es del orden $O(2^n)$ y al algoritmo de Fibonacci que es del orden $O((\frac{1+\sqrt{5}}{2})^n)$ siendo este ultimo el mas lento de todos.

Como conclusion a este apartado hemos podido observar que los algoritmos de ordenación con eficiencia $O(n^2)$ son mas lentos que los de eficiencia $O(n\log(n))$ y que dependiendo del tipo de dato con el que se trabaje y de si hay datos repetidos o no habrá algoritmos a los que les afecte de manera positiva como al mergesort o el de Inserción que pasa a ser practicamente logaritmico y otros como el de burbuja o seleccion que se vean perjudicados por estos datos.

Conclusiones