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         static 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.

5.2. Burbuja

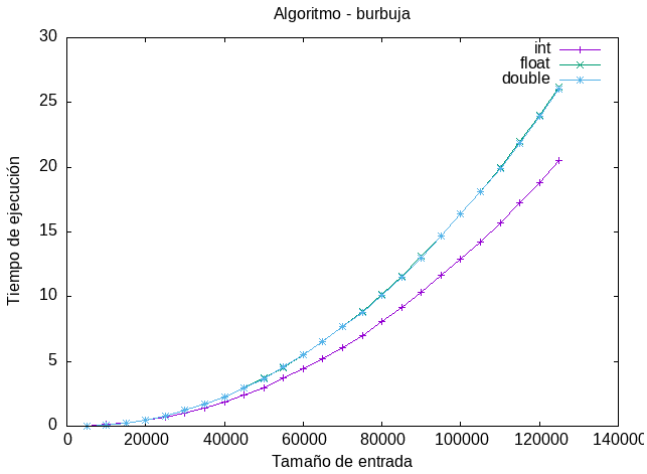

Figura 1: Ejecución algoritmo burbuja

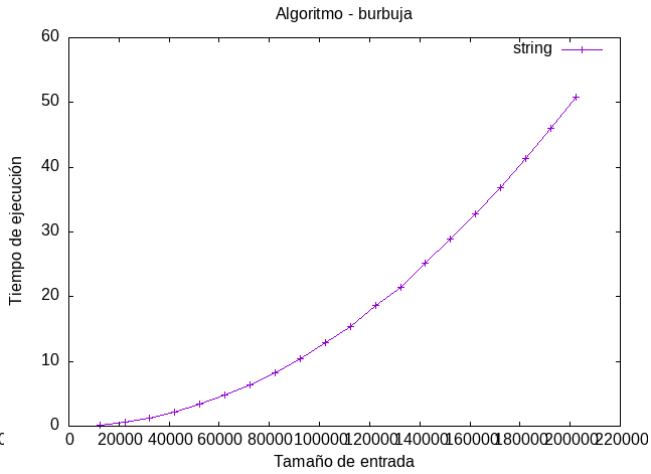

Figura 2: Ejecución algoritmo burbuja con string

5.3. Seleccion

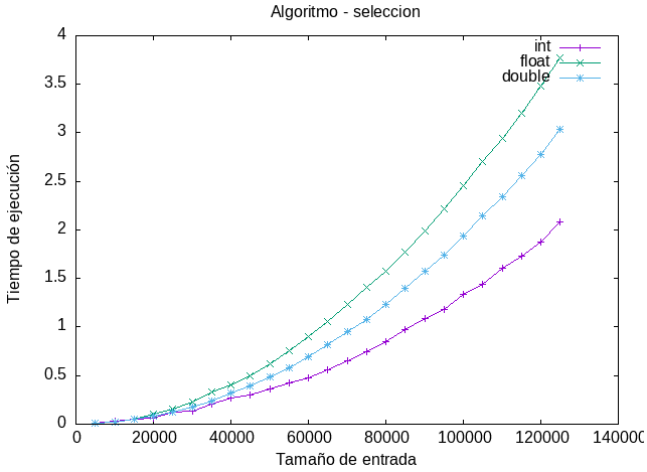

Figura 3: Ejecución algoritmo seleccion

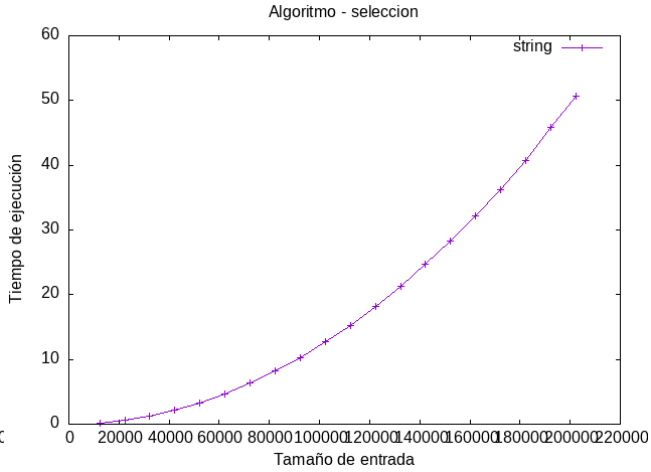

Figura 4: Ejecución algoritmo seleccion con string

5.4. Inserción

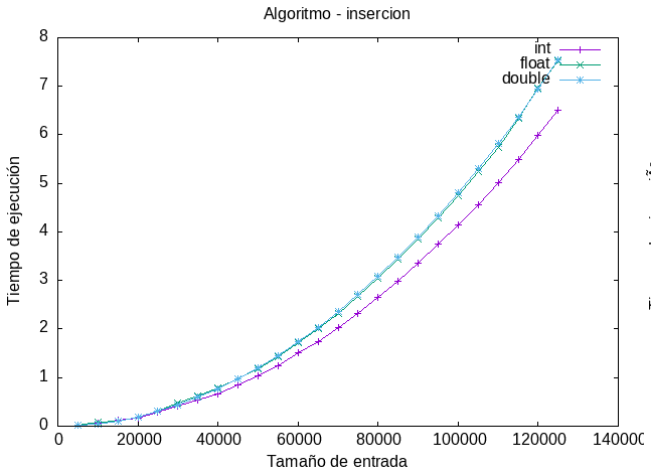

Figura 5: Ejecución algoritmo insercion

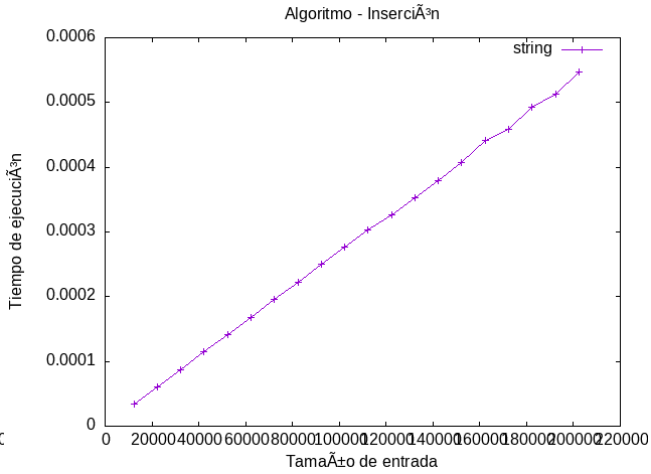

Figura 6: Ejecución algoritmo inserción con string

5.5. Hanoi

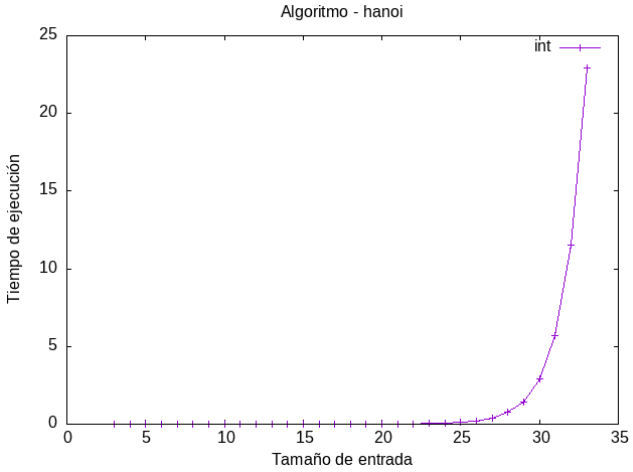

Figura 7: Ejecución algoritmo Hanoi

5.6. Fibonacci

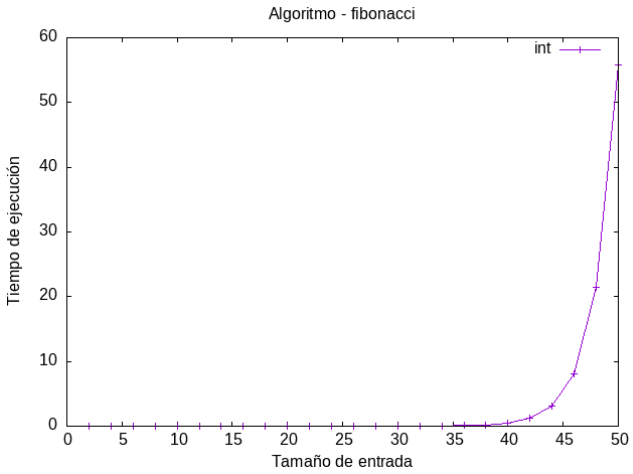

Figura 8: Ejecución algoritmo Fibonacci

5.7. Floyd

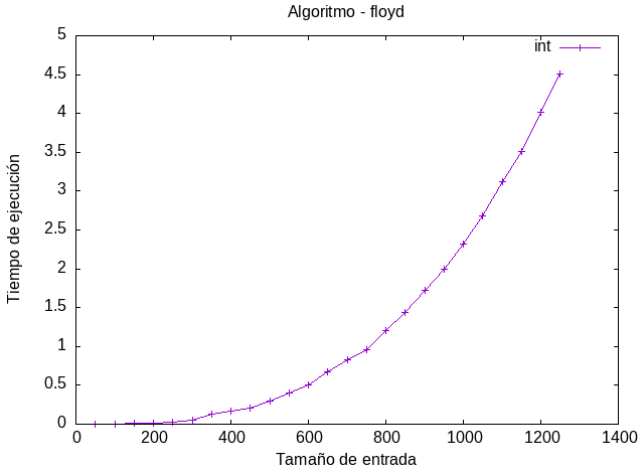

Figura 9: Ejecución algoritmo Floyd

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 $O(n^2)$

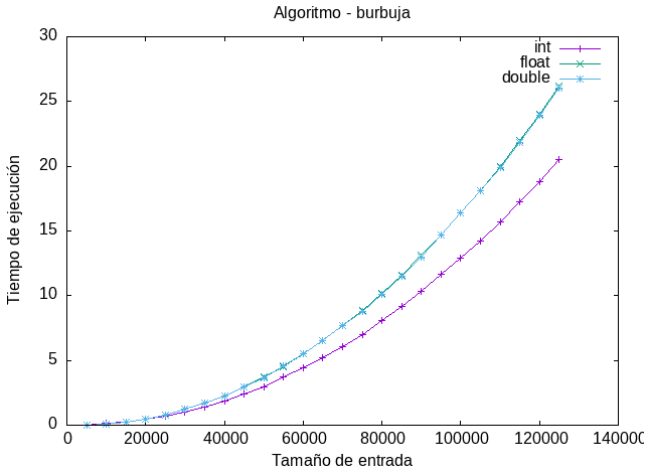

Figura 10: Ejecución algoritmo burbuja

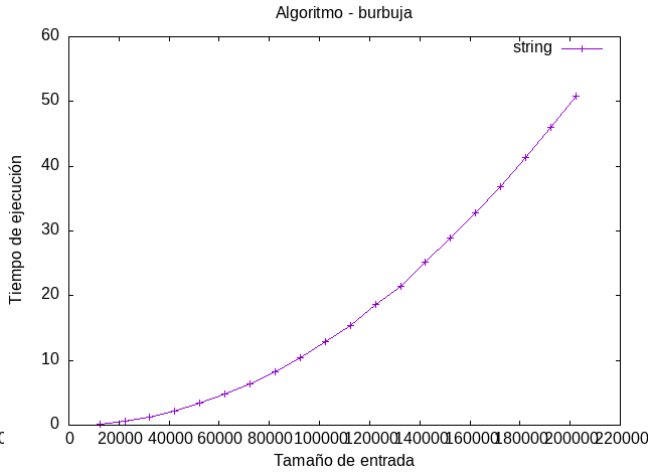

Figura 11: Ejecución algoritmo burbuja con string

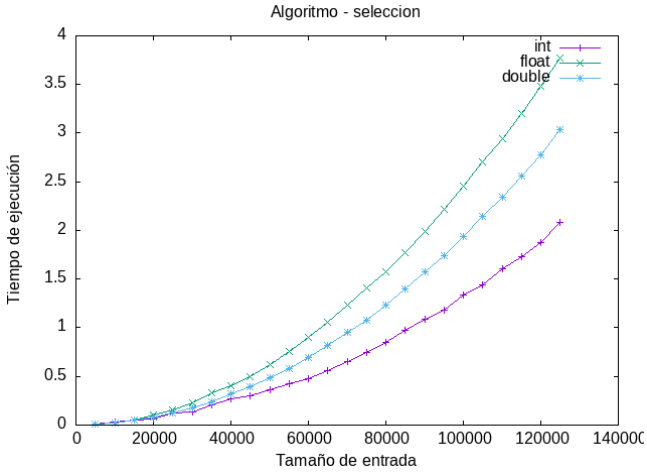

Figura 12: Ejecución algoritmo seleccion

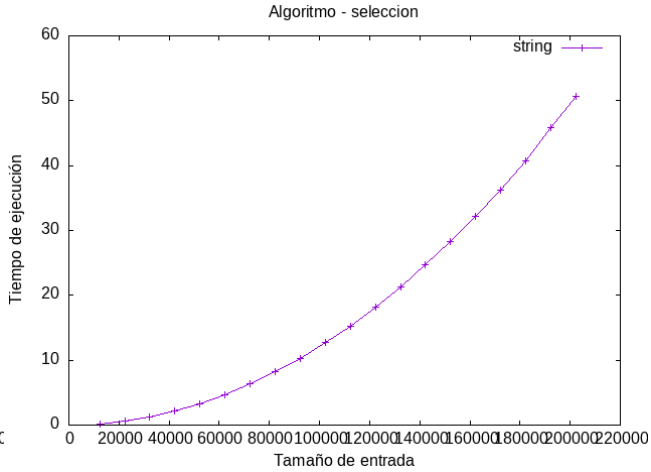

Figura 13: Ejecución algoritmo seleccion con string

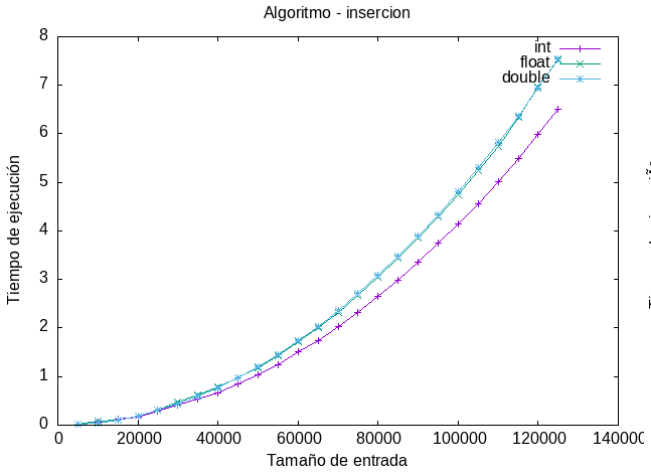

Figura 14: Ejecución algoritmo insercion

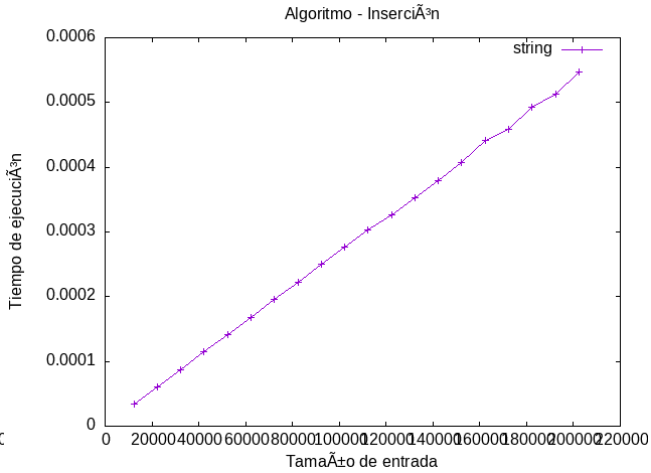

Figura 15: Ejecución algoritmo inserción con string

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 $O(n\log(n))$

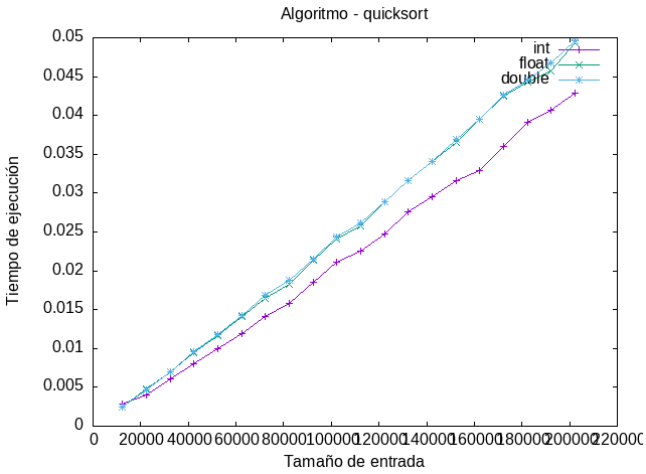

Figura 16: Ejecución algoritmo insercion

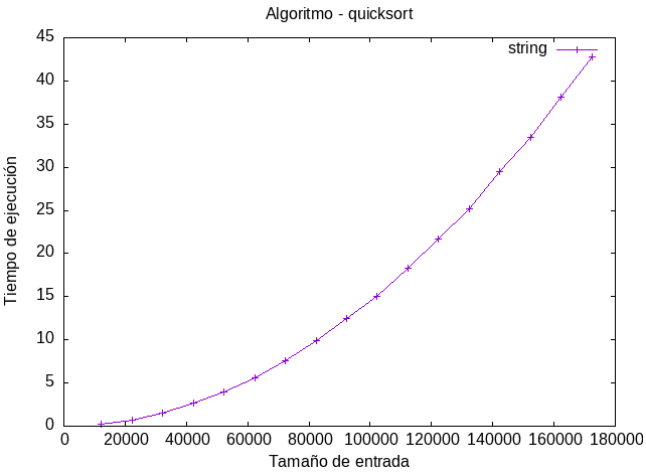

Figura 17: Ejecución algoritmo inserción con string

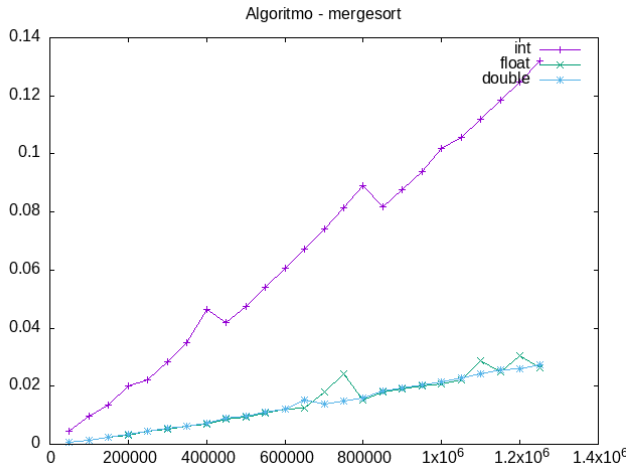

Figura 18: Ejecución algoritmo mergesort

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

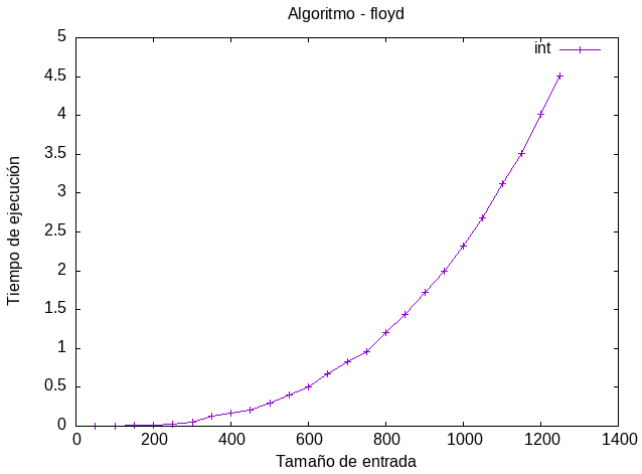

Figura 19: Ejecución algoritmo mergesort

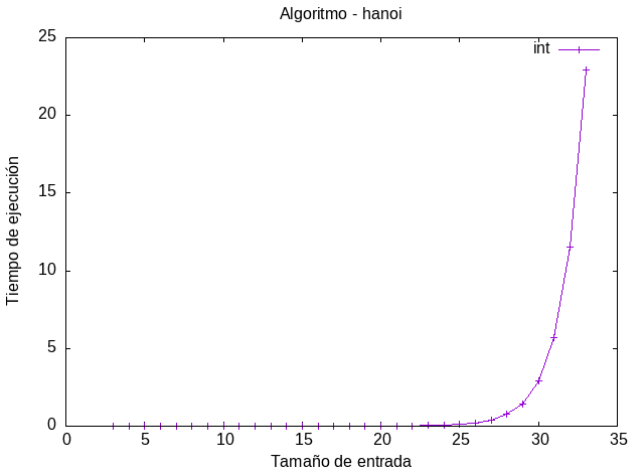

Figura 20: Ejecución algoritmo mergesort

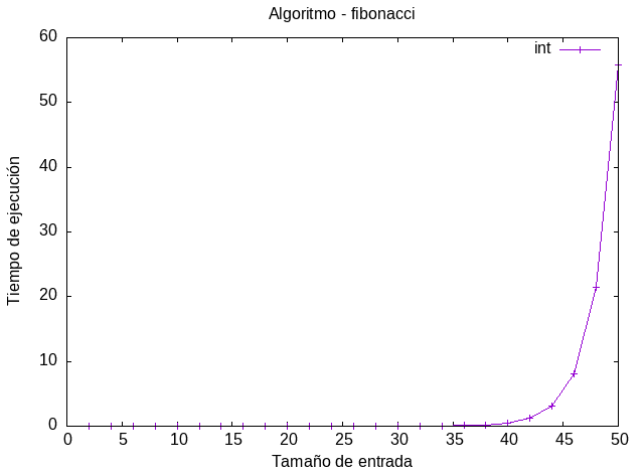

Figura 21: Ejecución algoritmo mergesort

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