Algorítmica (2016-2017)

Grado en Ingeniería Informática

Universidad de Granada

Práctica 1:
Análisis de Eficiencia de Algoritmos

Gregorio Carvajal Expósito
Gema Correa Fernández
Jonathan Fernández Mertanen
Eila Gómez Hidalgo
Elías Méndez García
Alex Enrique Tipán Párraga

Contenidos

Introducción

En esta primera parte de la Práctica 1, vamos a comprobar empíricamente los tiempos de ejecución de diferentes algoritmos para distintos tamaños de entrada. En concreto, realizaremos los dos primeros ejercicios de la práctica.

A continuación, compararemos los resultados que producen los distintos algoritmos, para un mismo orden de eficiencia.

Para profundizar un poco más, haremos una comparativa entre los algoritmos de ordenación, los cuales tendrán distintos órdenes de eficiencia, para así poder ver cuales son más rápidos.

Eficiencia Empírica

Procedimiento

Para el cálculo de la eficiencia empírica de los distintos algoritmos, hemos realizado el siguiente procedimiento:

1. Para medir los tiempos en cada programa se ha utilizado la biblioteca `chrono` del estándar de c++11. Para ello, tenemos que añadir a la cabecera de cada programa, lo siguiente:

```
#include <chrono>
using namespace std::chrono;
```

2. Para mejorar la automatización de la obtención de tiempos al programa, se ha editado el código para que reciba dos argumentos. El primero es el número de datos con los que se debe ejecutar el algoritmo y el segundo es el número con el que se debe repetir el mismo procedimiento para sacar una media de tiempos y así intentar evitar las variaciones de tiempos.

```
if (argc != 3) {
    cerr << "El programa necesita dos argumentos." << endl
           << "Uso: " << argv[0]
           << "<número de datos> <número de repeticiones>"
           << endl;

    return (-1);
```

```
}

int n = std::stoi(argv[1]);            // número de datos
int iteraciones = std::stoi(argv[2]);  // número de repeticiones
```

3. Para garantizar que todas las repeticiones se hacen con los mismos datos antes de ser ordenados, lo primero que hacemos, al empezar otra iteración, es crear una copia de los datos originales y trabajar sobre esa copia.

4. Para la medición de tiempos usamos `high_resolution_clock::now()`, que nos da la ventaja de obtener medidas de tiempo muy pequeñas y que no se redondee a cero. Para la medición de tiempo, se consulta el reloj antes y después de lanzar el algoritmo y calculamos la diferencia entre ambos momentos de tiempo para obtener el tiempo de ejecución. Al final haremos la media con la acumulación de tiempos obtenidos.

```
double tiempo = 0.0f;
for (int i = 0; i < iteraciones; ++i) {

    int *copy = new int[n];
    std::copy(T, T + n, copy);

    assert(copy);

    auto t1 = high_resolution_clock::now();
    algoritmo(copy, n);
    auto t2 = high_resolution_clock::now();

    tiempo += duration_cast<duration<double>>(t2 - t1).count();

    delete[] copy;
}
```

5. Una vez terminada la ejecución, se obtiene la media de los tiempos y se muestra por pantalla en un formato compatible con gnuplot.

```
tiempo /= iteraciones;

cout << n << " " << tiempo << endl;
```

Para facilitar la obtención de los tiempos y la generación de las gráficas de puntos con gnuplot se ha usado el siguiente *Makefile* y *script*.

Makefile

```
##############################################################################
# Makefile
##############################################################################

SHELL = /bin/bash # for ubuntu

##############################################################################
SRC = $(wildcard *.cpp)
EXE = $(basename $(SRC))
DAT = $(EXE:=.dat)
SVG = $(EXE:=.svg)

##############################################################################
CFLAGS = -O3
CXXFLAGS =  $(CFLAGS) -std=c++11 # -std=c++1y

##############################################################################

default: $(EXE)

all: default data

clean:
	$(RM) -rfv $(EXE)

purge: clean
	$(RM) -rfv $(SVG) $(DAT)

data: default
	chmod +x generateData.sh
	./generateData.sh
```

generateData.sh

```
#!/bin/bash
# Argumentos
# $1 nombre del programa a ejecutar
# $2 tamaño de datos inicial
# $3 tamaño de datos de la última iteración
# $4 incremento entre cada iteración
# $5 número veces a ejecutar para obtener la media
function generateData() {
  echo "" > $1.dat
  for ((c = $2; c <= $3; c += $4)); do
    ./$1 $c $5 >> $1.dat
  done
}

for I in "burbuja" "heapsort" "insercion" "quicksort" "mergesort"
"seleccion"
do
  echo "Calculando los tiempos del algoritmo: $I"
  generateData $I 1000 25000 1000 10
done

echo "Calculando los tiempos del algoritmo: hanoi"
generateData hanoi 10 35 1 10

echo "Calculando los tiempos del algoritmo: floyd"
generateData floyd 100 2500 100 10

for D in "burbuja" "heapsort" "insercion" "quicksort" "mergesort"
"seleccion" "hanoi" "floyd"
do
  echo "Generando gráfico de: $D"
  gnuplot <<< "\
      set terminal svg; \
      set output '$D.svg'; \
       set xlabel 'Tamaño'; \
       set ylabel 'Tiempo (seg)'; \
       plot '$D.dat' title 'Eficiencia de $D' with points"
done
```

Tablas de los tiempos de ejecución

Vamos a agrupar los tiempos obtenidos en 4 tablas en función de las distintas eficiencias teóricas de cada algoritmo.

En primer lugar, tenemos los tiempos de ejecución para los algoritmos de burbuja, inserción y selección, que comparten el mismo orden de eficiencia O(n²) *(Tabla 1)*, usaremos distintos números de datos (comprendidos entre 1000 y 25000):

Orden de eficiencia O(n²)			
Nº de datos	**Burbuja**	**Inserción**	**Selección**
1000	0,00070484	0,000168415	0,000547321
2000	0,00278126	0,000659599	0,00223252
3000	0,00661067	0,00153272	0,00490875
4000	0,0128572	0,00267263	0,00868611
5000	0,0214832	0,00431276	0,0135702
6000	0,0335224	0,00589975	0,0194873
7000	0,0471374	0,00786774	0,026516
8000	0,06509	0,010325	0,034716
9000	0,0848508	0,0130514	0,0432733
10000	0,109111	0,0160186	0,0541113
11000	0,134892	0,019131	0,0643321
12000	0,162664	0,023165	0,0776892
13000	0,195891	0,0275811	0,0915134
14000	0,228442	0,0318901	0,104928
15000	0,263753	0,0361799	0,119902
16000	0,303609	0,0417343	0,138473
17000	0,347029	0,0466932	0,154819
18000	0,388661	0,0527501	0,174173
19000	0,438423	0,0592765	0,194356
20000	0,485374	0,0644968	0,216541
21000	0,544013	0,072138	0,235675
22000	0,602217	0,0794605	0,257706
23000	0,655833	0,0871451	0,283176
24000	0,721958	0,0936699	0,306367
25000	0,783014	0,102941	0,338052

Tabla 1. Tiempos Obtenidos para O(n²)

En segundo lugar, podemos ver los tiempos de ejecución de los algoritmos Mergesort, Quicksort y Heapsort, que comparten el mismo orden de eficiencia O(n log(n)) *(Tabla 2)*, y usaremos distintos números de datos (comprendidos entre 1000 y 25000):

Orden de eficiencia O(n log(n))			
Nº de datos	**Mergesort**	**Quicksort**	**Heapsort**
1000	0,0000254	2,32E-05	4,84E-05
2000	0,0000743	6,61E-05	0,000108417
3000	0,0001207	0,000105517	0,000178247
4000	0,0001651	0,000152682	0,000251412
5000	0,0002191	0,00019645	0,000323969
6000	0,0002814	0,000253365	0,000396304
7000	0,0003167	0,000287885	0,000466316
8000	0,0003703	0,000327304	0,000537423
9000	0,0004369	0,000385544	0,000610291
10000	0,0004746	0,000415927	0,000689575
11000	0,0005446	0,000471943	0,000767739
12000	0,0006024	0,000522267	0,000843046
13000	0,0006297	0,000570751	0,000916754
14000	0,0006869	0,000603042	0,00101354
15000	0,0007438	0,000646043	0,00108273
16000	0,0008024	0,000702551	0,00116413
17000	0,0008726	0,000767774	0,00124579
18000	0,0009337	0,000813343	0,00133197
19000	0,0009884	0,000856854	0,00138204
20000	0,0010473	0,00090537	0,00149273
21000	0,0011262	0,000974075	0,00157585
22000	0,0011676	0,00100891	0,00165497
23000	0,0012511	0,00108535	0,00171587
24000	0,0013058	0,00112833	0,0018183
25000	0,0013863	0,00117366	0,00190969

Tabla 2. Tiempos Obtenidos para O(n log(n))

En tercer lugar, tenemos la tabla con los tiempos de ejecución del algoritmo Floyd que tiene un orden de eficiencia O(n³) *(Tabla 3)*:

Orden de Eficiencia O(n³)	
Nº de datos	**Floyd**
100	0,000927308
200	0,00686802
300	0,0223873
400	0,0519861
500	0,101854
600	0,176301
700	0,277682
800	0,41186
900	0,5953
1000	0,827693
1100	1,159
1200	1,54931
1300	2,02421
1400	2,64131
1500	3,2443
1600	3,97622
1700	4,79084
1800	5,67815
1900	6,67535
2000	7,77094
2100	9,03474
2200	10,1211
2300	11,6046
2400	13,2707
2500	15,2813

Tabla 3. Tiempos Obtenidos para O(n log(n))

En último lugar, tenemos la tabla con los tiempos de ejecución del algoritmo Hanoi, siendo su orden de eficiencia $O(2^n)$ *(Tabla 4)*:

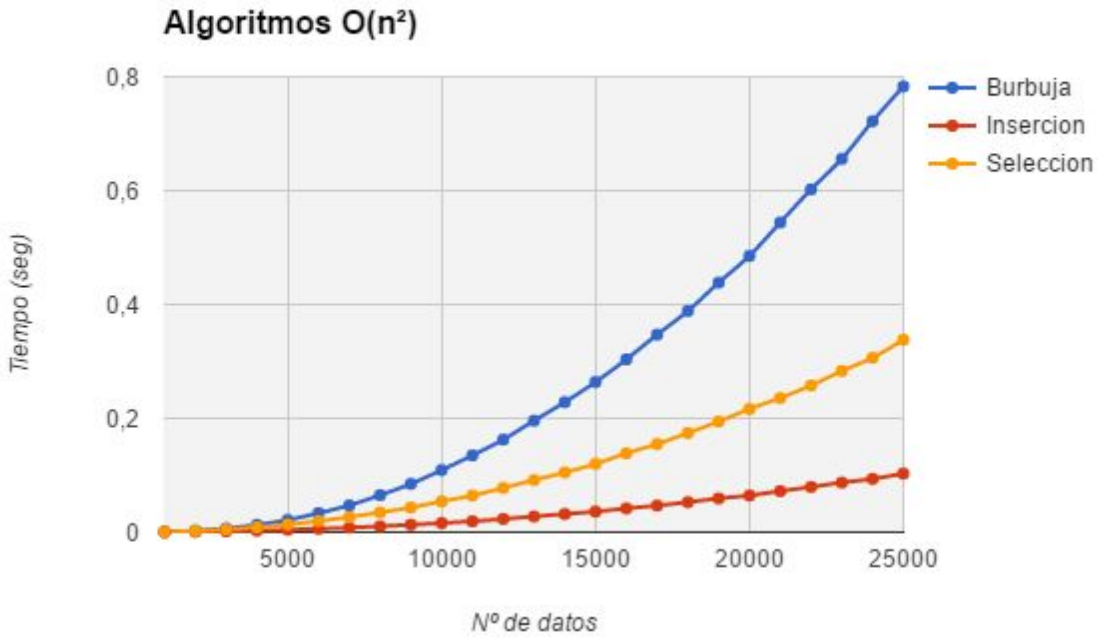
Orden de eficiencia $O(2^n)$	
Nº de datos	**Hanoi**
10	1,50E-06
11	2,72E-06
12	5,19E-06
13	1,02E-05
14	1,94E-05
15	3,84E-05
16	8,33E-05
17	0,000162479
18	0,000332946
19	0,00067568
20	0,00138661
21	0,00273687
22	0,00553412
23	0,0110794
24	0,0220678
25	0,0438211
26	0,0867785
27	0,171589
28	0,340834
29	0,67166
30	1,33313
31	2,64708
32	5,30209
33	10,6107
34	21,4292
35	42,8149

Tabla 4. Tiempos Obtenidos para $O(2^n)$

Gráficas de los tiempos de los algoritmos según su orden de eficiencia

En el siguiente apartado vamos a comparar los tiempos obtenidos dependiendo de su orden de eficiencia $O(n^2)$, $O(n \log(n))$, $O(n^3)$, $O(2^n)$.

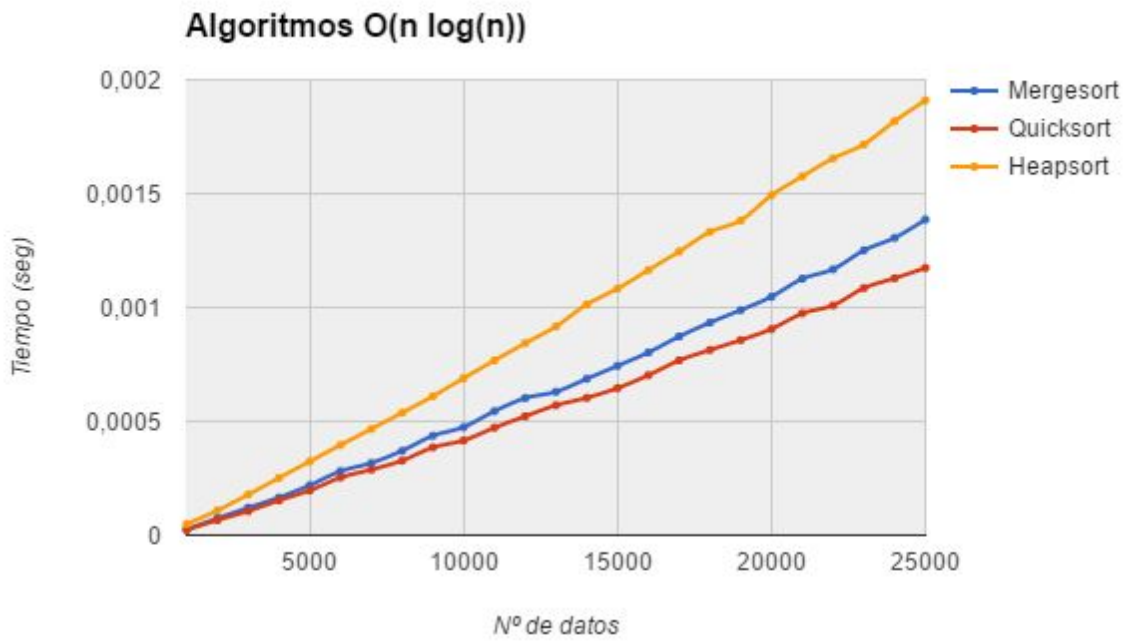
A continuación, se muestran los tiempos obtenidos para los algoritmos con orden de eficiencia $O(n^2)$:



Gráfica 1. Algoritmos de Ordenación con Orden de Eficiencia $O(n^2)$

Como podemos apreciar tenemos tres algoritmos de orden de eficiencia $O(n^2)$, que son la ordenación por burbuja, por inserción y por selección. Cuanto mayor es el número de datos más se aprecia la diferencia de tiempos que hay entre los distintos algoritmos. Como podemos ver en la Gráfica 1, el mejor tiempo se obtiene para la ordenación por selección. Y el peor tiempo obtenido es para la ordenación por burbuja (casi 5 veces más lento que el de ordenación por selección para 25000 datos).

A continuación, se muestran los tiempos obtenidos para los algoritmos con orden de eficiencia O(n log(n)):



Gráfica 2. Algoritmos de Ordenación con Orden de Eficiencia O(n log(n))

Como podemos apreciar tenemos tres algoritmos de orden de eficiencia O(n log(n)), que son la ordenación por Mergesort, Quicksort y Heapsort. Cuanto mayor es el número de datos más se empieza a apreciar la diferencia de tiempos que hay entre los distintos algoritmos. Como podemos ver en la Gráfica 2, el mejor tiempo se obtiene para la ordenación por Quicksort. Y el peor tiempo obtenido es para la ordenación Heapsort (casi 2 veces más lento que el de ordenación por Quicksort para 25000 datos).

A continuación, se muestran los tiempos obtenidos para el algoritmo con orden de eficiencia $O(n^3)$:

Gráfica 3. Algoritmos con Orden de Eficiencia $O(n^3)$

En la siguiente tabla se muestran los tiempos obtenidos para el algoritmo con orden de eficiencia $O(2^n)$:

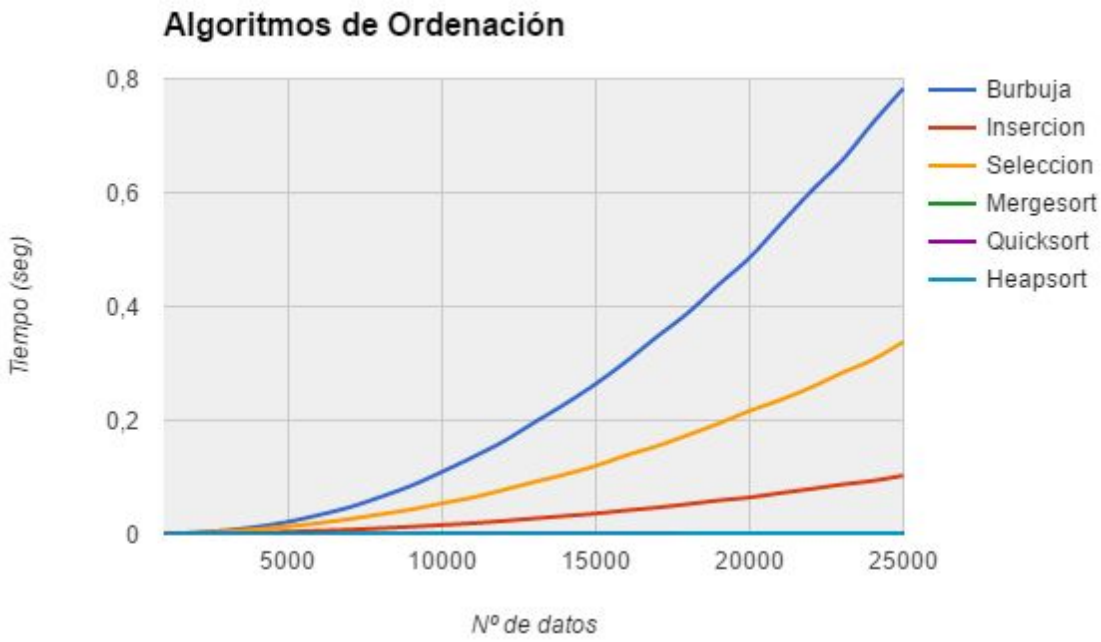
Gráfica 4. Algoritmos con Orden de Eficiencia $O(2^n)$

Comparación de los tiempos de los algoritmos de ordenación

En la Gráfica 5 podemos ver los tiempos que tardan los 6 algoritmos de ordenación, que estamos estudiando, para diferentes cantidades de datos.

Debemos diferenciar entre los dos órdenes de eficiencia que tienen los distintos algoritmos:

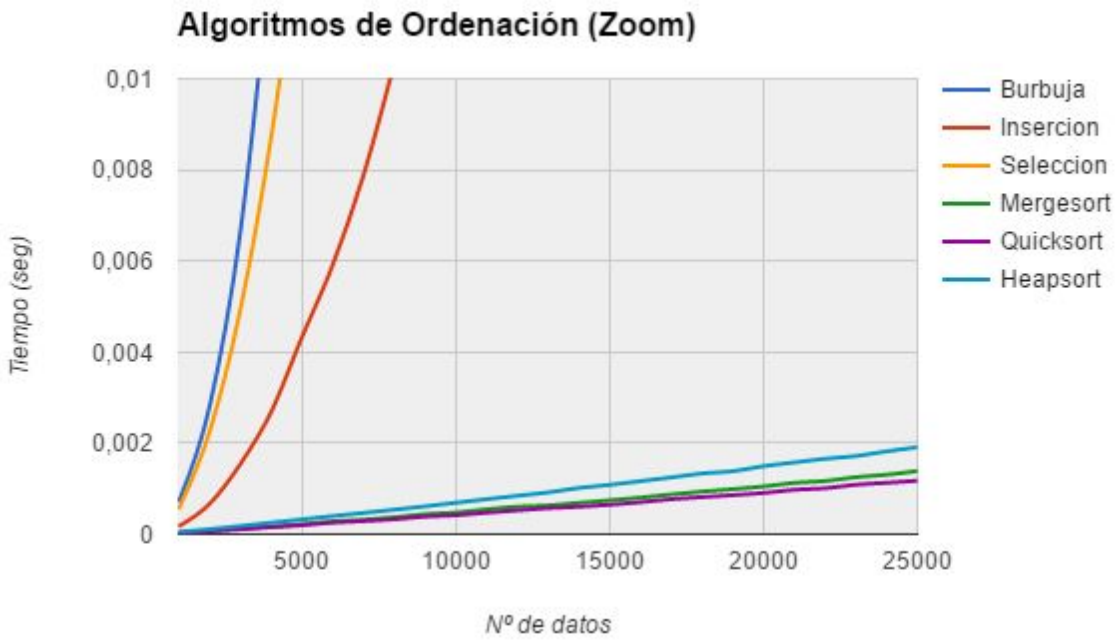
- Burbuja, Inserción y Selección, tienen orden eficiencia $O(n^2)$
- Mergesort, Quicksort y Heapsort, tiene orden de eficiencia O(n log(n))



Gráfica 5. Algoritmos de Ordenación

Como se puede ver el la Gráfica 5 los tiempos de los tres algoritmos con orden $O(n^2)$ empiezan a aumentar cuanto mayor el el número de datos a ordenar, alejándose de forma considerable de los tiempos obtenidos por los de orden O(n log(n)).

Como los algoritmos Mergesort, Quicksort y Heapsort son tan rápidos, en comparación con los otros tres, no se aprecian bien sus líneas en el gráfico. Es por ello que adjuntamos la siguiente imagen con el mismo gráfico pero realizando un poco de zoom (Gráfica 6):



Gráfica 6. Algoritmos de Ordenación Ampliados

Como se puede apreciar en la Gráfica 6, el mejor tiempo lo obtiene Quicksort, seguido muy cerca por Mergesort y Heapsort.