UNIVERSIDAD DE GRANADA

E.T.S. DE INGENIERÍAS INFORMÁTICA y DE TELECOMUNICACIÓN

Departamento de Ciencias de la Computación e Inteligencia Artificial

Doble Grado en Ingeniería Informática y Matemáticas

Curso 2022-2023

Algorítmica

Guión de Prácticas

Práctica 1: Análisis de eficiencia de algoritmos

Índice

Objetivo

El objetivo de esta práctica es que el estudiante comprenda la importancia del análisis de la eficiencia de los algoritmos y se familiarice con las formas de llevarlo a cabo. Para ello se mostrará cómo realizar un estudio teórico, empírico e híbrido. Cada estudiante deberá realizar los análisis empíricos e híbridos de los algoritmos que se detallan más adelante.

1. Cálculo del tiempo teórico

A partir de la expresión del algoritmo, se aplicarán las reglas conocidas para contar el número de operaciones que realiza un algoritmo. Este valor será expresado como una función $T(n)$ que dará el número de operaciones requeridas para un caso concreto del problema caracterizado por tener un tamaño n.

En el caso de los algoritmos recursivos aparecerá una expresión del tiempo de ejecución con forma recursiva, que habrá que resolver con las técnicas estudiadas (p.e. resolución de recurrencias por el método de la ecuación característica). En el caso de esta práctica nos centraremos en el análisis del peor caso. Así, tras obtener la expresión analítica de $T(n)$, calcularemos el orden de eficiencia del algoritmo empleando la notación $\mathcal{O}()$.

A continuación, desarrollaremos el estudio teórico sobre dos algoritmos de ejemplo.

1.1. Ejemplo 1: Algoritmo de ordenación Burbuja

Vamos a obtener la eficiencia teórica del algoritmo de ordenación burbuja. Para ello vamos a considerar el siguiente código que implementa la ordenación de un vector de enteros, desde las posiciones `inicial` a `final`, mediante el método burbuja.

```
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 }
```

La mayor parte del tiempo de ejecución se emplea en el cuerpo del bucle interno. Esta porción de código lo podemos acotar por una constante a. Por lo tanto, las líneas de $7-12$ se ejecutan exactamente un número de veces igual a $(\texttt{final}-1)-(\texttt{i}+1)+1$, es decir, $\texttt{final}-\texttt{i}-1$. A su vez el bucle interno se ejecuta un número de veces indicado por el bucle externo. En definitiva, tendríamos una fórmula como la siguiente:

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

Renombrando `final` como n e `inicial` como 1 en la expresión (1), pasamos a calcular la siguiente expresión:

$$\sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} a \tag{2}$$

Realizando la sumatoria interior en (2) obtenemos:

$$\sum_{i=1}^{n-2} a(n-i-1) \tag{3}$$

Y, finalmente, tenemos:

$$\frac{a}{2}n^2 - \frac{3a}{2}n + a \tag{4}$$

Claramente, $\frac{a}{2}n^2 - \frac{3a}{2}n + a \in \mathcal{O}(n^2)$. Diremos, por tanto, que el método de ordenación es de orden $\mathcal{O}(n^2)$ o cuadrático.

1.2. Ejemplo 2: Algoritmo de ordenación Mergesort

En este ejemplo vamos a calcular la eficiencia del algoritmo de ordenación Mergesort. Este algoritmo divide el vector en dos partes iguales y se vuelve a aplicar de forma recurrente a cada una de ellas. Una vez hecho esto, fusiona los dos vectores sobre el vector original, de manera que esta parte ya queda ordenada. Si el número de elementos del vector que se está tratando en cada momento de la recursión es menor que una constante `UMBRALMS`, entonces se ordenará mediante el algoritmo burbuja.

Usamos la siguiente implementación del algoritmo Mergesort, que ordena un vector de enteros entre las posiciones `inicial` y `final`:

```
1 void fusion (int T[], int inicial, int final, int U[], int V[])
2 {
3   int j = 0 ;
4   int k = 0 ;
5
6   for (int i = inicial; i < final; i++)
7     if (U[j] < V[k]) {
8       T[i] = U[j];
9       j++;
10    }
11    else {
```

```
12      T[i] = V[k];
13      k++;
14    }
15 }
16
17 void mergesort (int T[], int inicial, int final)
18 {
19   if (final - inicial < UMBRALMS)
20     burbuja (T, inicial, final);
21   else {
22     int k = (final - inicial)/2;
23     int * U = new int [k - inicial + 1];
24     assert (U);
25     int l, l2;
26
27     for (l = 0, l2 = inicial; l < k; l++, l2++)
28       U[l] = T[l2];
29
30     U[l] = INT_MAX;
31     int * V = new int [final - k + 1];
32     assert (V);
33
34     for (l = 0, l2 = k; l < final - k; l++, l2++)
35       V[l] = T[l2];
36
37     V[l] = INT_MAX;
38     mergesort(U, 0, k);
39     mergesort(V, 0, final - k);
40     fusion(T, inicial, final, U, V);
41     delete [] U;
42     delete [] V;
43   }
44 }
```

Una vez entendido el procedimiento `mergesort` vamos a pasar a obtener su eficiencia teórica.

Suponiendo que entramos por la rama else (línea 21) podemos acotar la secuencia de instrucciones de las líneas $22-25$ por una constante. Sin perder generalidad, vamos a tomar esta constante como c. El bucle `for` de la línea 27 se ejecuta un número de veces igual a $n/2$ (en la primera llamada de `mergesort` tomaremos `inicial` como 0 y `final` como n), por lo tanto la

eficiencia hasta aquí sería $\mathcal{O}(n)$. Este mismo razonamiento lo tenemos para las instrucciones desde la línea 30 hasta 37. Aplicando la regla del máximo obtendríamos hasta la línea 37 un orden de $\mathcal{O}(n)$.

A continuación, se hacen dos llamadas de forma recursiva a `mergesort` (líneas 38 y 39) con los dos nuevos vectores construidos para ordenar en cada uno de ellos $n/2$ elementos. En la línea 40 se llama al procedimiento `fusion` que, como se puede observar, tiene una eficiencia de $\mathcal{O}(n)$.

Por lo tanto, para averiguar la eficiencia de `mergesort`, podemos formular la siguiente ecuación:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \text{si } n \geq \texttt{UMBRALMS} \\ n^2 & \text{si } n < \texttt{UMBRALMS} \end{cases} \tag{5}$$

Vamos a resolver la recurrencia en la ecuación (5):

$$T(n) = 2T(\frac{n}{2}) + n, \text{ si } n \geq \texttt{UMBRALMS} \tag{6}$$

Haciendo el cambio de variable $n = 2^m$ en la expresión (6) obtenemos:

$$T(2^m) = 2T(2^{m-1}) + 2^m \text{ si } m \geq \log_2(\texttt{UMBRALMS}) \tag{7}$$

$$T(2^m) - 2T(2^{m-1}) = 2^m \tag{8}$$

Renombrando $T(2^m) = t_m$, resulta

$$t_m - 2t_{m-1} = 2^m \tag{9}$$

La ecuación de recurrencia que se deriva de la ecuación (9) es:

$$(x-2)^2 = 0 \tag{10}$$

Por tanto, la solución general será:

$$t_m = c_1 2^m + c_2 m 2^m \tag{11}$$

Deshaciendo el cambio de variable, obtenemos:

$$T(n) = c_1 n + c_2 n \log_2(n) \tag{12}$$

Por tanto $T(n) \in \mathcal{O}(n \log_2(n))$.

1.2.1. Comentarios adicionales

En este ejemplo cabe preguntarse por qué hemos hecho uso de un algoritmo de ordenación de orden cuadrático (línea 20), el algoritmo burbuja de ordenación para el caso base, cuando el algoritmo `mergesort` es más eficiente.

La respuesta se verá cuando se estudie la técnica de resolución de problemas "Divide y Vencerás". Como idea básica, basta saber que el algoritmo de ordenación burbuja a pesar de ser cuadrático, es lo suficientemente eficiente para aplicarlo sobre un vector con pocos elementos, como se da en este caso con un valor de `UMBRALMS` lo suficientemente pequeño.

También se verá más adelante que los algoritmos recursivos, a igual orden de eficiencia, son menos eficientes que los algoritmos iterativos (hacen uso de llamadas recursivas lo que implica usar la pila donde guardar los valores de las variables en cada recursión).

2. Cálculo de la eficiencia empírica

Veremos ahora como llevar a cabo un estudio puramente empírico del comportamiento de los algoritmos analizados. Para ello mediremos los recursos empleados (tiempo) para cada tamaño dado de las entradas.

En el caso de los algoritmos de ordenación, el tamaño viene dado por el número de componentes del vector a ordenar.

2.1. Cálculo del tiempo de ejecución

Para obtener el tiempo empírico de una ejecución de una parte de un programa usamos la biblioteca `ctime`. Comenzamos definiendo en el código dos variables de tipo `clock_t`:

```
clock_t t_antes;
clock_t t_despues;
```

En la variable `t_antes` capturamos el valor del reloj antes de la ejecución del algoritmo al que queremos medir el tiempo. La variable `t_despues` contendrá el valor del reloj después de la ejecución del algoritmo en cuestión. Así, si deseamos obtener el tiempo del algoritmo de ordenación burbuja tendremos que poner algo parecido a lo siguiente:

```
//Captura el valor del reloj antes de la llamada a burbuja
t_antes = clock();
// Llama al algoritmo de ordenación burbuja
```

```
burbuja(T, 0, n);
//Captura el valor del reloj después de la ejecución de burbuja
t_despues = clock();
```

Para obtener el número de segundos el cálculo es sencillo:

```
cout << (double)(t_despues - t_antes) / CLOCKS_PER_SEC << endl;
```

La diferencia entre las dos medidas de tiempo se convierte en segundos dividiendo por la constante `CLOCKS_PER_SEC`.

2.2. Medidas para tamaños de entrada pequeños

Para aquellos casos donde, por ejemplo, la cantidad de elementos a ordenar es muy pequeña, el tiempo medido será muy pequeño y por lo tanto el resultado será 0 segundos. Estos tiempos tan pequeños se pueden medir de forma indirecta ejecutando la sentencia que nos interesa muchas veces y después dividiendo el tiempo total por el número de veces que se ha ejecutado. Por ejemplo:

```
#include <ctime>
clock_t t_antes, t_despues;
double tiempo_transcurrido;
const int NUM_VECES = 10000;
int i;
t_antes = clock();
for (i = 0; i < NUM_VECES; i++)
  //Sentencia cuyo tiempo se pretende medir
t_despues = clock();
tiempo_transcurrido = ((double)(t_despues - t_antes)/
                       (CLOCKS_PER_SEC * (double)NUM_VECES));
```

2.3. Otra forma más precisa de calcular el tiempo de ejecución

También se dispone de otra biblioteca de la STL, `chrono`, que permite calcular el tiempo de ejecución de forma algo más precisa (necesita C++ 11 o posterior). Para emplearla incluimos en el programa:

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

```
high_resolution_clock::time_point t_antes, t_despues;
duration<double> transcurrido;
```

Luego se mide el valor del reloj antes y después de la ejecución del código y se transforma en segundos:

```
t_antes = high_resolution_clock::now();
  //sentencia o programa a medir
t_despues = high_resolution_clock::now();
transcurrido = duration_cast<duration<double>>(t_despues - t_antes);
cout << "el tiempo empleado es " << transcurrido.count() << " s."
     << endl;
```

En este caso al compilar se debe incluir la directiva `-std=gnu++0x`.
Se pueden consultar otras formas de medir el tiempo en:
`https://ichi.pro/es/8-formas-de-medir-el-tiempo-de-ejecucion-en-c-c-79591185`

2.4. Automatización de las medidas

Para realizar un análisis de eficiencia empírica deberemos ejecutar el mismo algoritmo para diferentes tamaños de entrada. Así, para un algoritmo de ordenación, por ejemplo, lo ejecutaremos para diferentes tamaños del vector a ordenar y obtendremos el tiempo (preferiblemente varias veces para cada tamaño, en cuyo caso obtendremos el tiempo medio por tamaño[1]). Estos tiempos los almacenaremos en un fichero. Para simplificar las tareas, se puede usar un "script" de shell. Un ejemplo es el que se muestra a continuación:

```
#!/bin/bash
i=1000
while [ "$i" -lt 10000 ]
do
    ./burbuja $i >> salida.dat
    i=$(( $i + 1000 ))
done
```

Para crearlo basta teclear su contenido en editor de texto. Puede ponerse cualquier nombre que sea significativo, pero es recomendable que tenga la extensión `.sh`. Por ejemplo, `mide-tiempos.sh`

[1]Esto tiene sentido en tanto en cuanto los tiempos de ejecución del algoritmo puedan variar considerablemente para entradas del mismo tamaño.

Este script escribe en el fichero `salida.dat` el tiempo en segundos que tarda el algoritmo de ordenación en ordenar vectores de 1000 a 100000 elementos. Las muestras se han tomado de 1000 en 1000. Para poder ejecutarlo debemos darle permisos de ejecución mediante la sentencia `chmod +x mide-tiempos.sh`, y a continuación ejecutarlo con `./mide-tiempos.sh`.

2.5. Cómo mostrar los resultados

Para mostrar la eficiencia empírica haremos uso de tablas y de gráficas que recojan el tiempo invertido para cada caso. Para mostrar los datos en una gráfica pondremos en el eje X (abscisas) el tamaño de los casos y en el eje Y (ordenadas) el tiempo, medido en segundos, requerido por la implementación del algoritmo. Existen distintos paquetes de software que permiten construir las representaciones gráficas. Algunas opciones son grace, (xmgrace), gnuplot, R o Matlab. El paquete grace es multiplataforma y se puede descargar desde `http://plasma-gate.weizmann.ac.il/Grace/`. Gnuplot (`http://gnuplot.sourceforge.net`) está disponible para Linux, OS/2, MS Windows, OSX, VMS y muchas otras plataformas. A continuación se ilustra brevemente el uso de gnuplot en Linux.

Partimos de un conjunto de datos, por ejemplo `salida.dat` que contiene en cada fila un par de valores (x, y) separados por espacios en blanco. El primer elemento del par se corresponde con el tamaño del problema y el segundo elemento se corresponde con el tiempo.

```
100 0
5100 0.53
10100 2.12
15100 4.72
20100 8.39
25100 13.11
30100 18.73
35100 25.4
...
```

Desde la línea de órdenes invocamos a `gnuplot`. Para poder representar estos puntos ejecutamos la orden

```
gnuplot> plot 'salida.dat' title 'Eficiencia XXX' with points
```

Como resultado aparecerá una nueva ventana como la de la figura 1. El mandato `plot` indica que queremos representar el fichero `salida.dat`, `with points` indica que en la salida nos muestre los datos como un conjunto de

puntos desconectados (hay otras opciones, como por ejemplo `with lines`). Podemos dar un título significativo al gráfico (en nuestro caso `Eficiencia XXX`.)

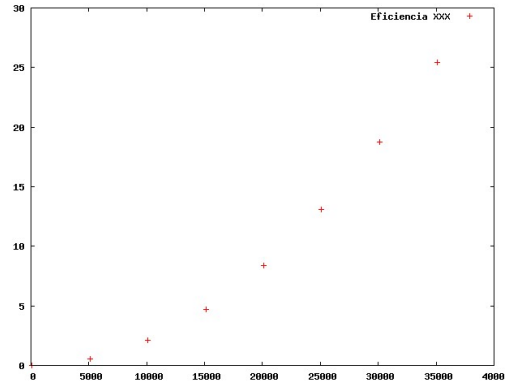

Figura 1: Fichero salida.dat

Además, podemos etiquetar los valores que representa el eje X y el eje Y. Para ello podemos ejecutamos las siguientes órdenes:

```
gnuplot> set xlabel "Tamaño"
gnuplot> set ylabel "Tiempo (seg)"
```

Para volver a mostrar el gráfico, podemos utilizar al orden `replot` (aunque en este caso el gráfico de la figura 2 ha sido obtenido empleando la opción `with lines`).

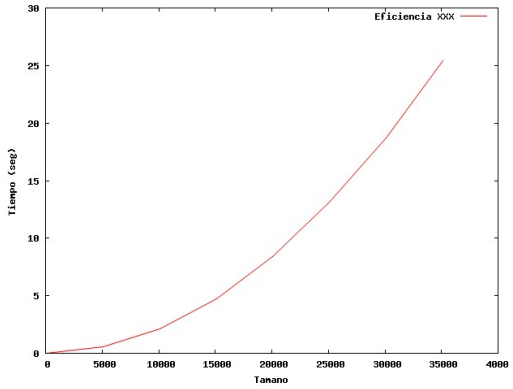

Figura 2: Fichero salida.dat etiquetado

Si se desea que el gráfico se guarde en un fichero de cierto tipo, por ejemplo pgn o jpeg (si queremos ver una lista de formatos disponibles podemos teclear `set terminal`), se puede emplear

```
gnuplot> set terminal jpeg
gnuplot> set output "fichero.jpeg"
```

Para hacerse una idea de la potencia y calidad de los gráficos que puede generar gnuplot, se puede consultar la página web `http://gnuplot.sourceforge.net/demo/`.

3. Cálculo de la eficiencia híbrida

El cálculo teórico del tiempo de ejecución de un algoritmo nos da mucha información. Es suficiente para comparar dos algoritmos cuando los suponemos aplicados a casos de tamaño arbitrariamente grande. Sin embargo, cuando se va a aplicar el algoritmo en una situación concreta, es decir, especificadas la implementación, el compilador utilizado, el ordenador sobre el que se ejecuta, etc., nos interesa conocer de la forma más exacta posible la fórmula de la función tiempo. Así, el cálculo teórico nos da la expresión general, pero asociada a cada término de esta expresión aparece una constante de valor desconocido.

Para describir completamente la fórmula del tiempo, necesitamos conocer el valor de esas constantes. La forma de averiguar estos valores es ajustar la función a un conjunto de puntos. En nuestro caso, la función es la que resulta del cálculo teórico, el conjunto de puntos lo forman los resultados del análisis empírico y para el ajuste emplearemos regresión por mínimos cuadrados. Por ejemplo, en el algoritmo de ordenación burbuja, la función que vamos a ajustar a los puntos obtenidos en el cálculo de la eficiencia empírica será un polinomio de segundo orden:

$$T(n) = a_2 \times n^2 + a_1 \times n + a_0$$

También podríamos usar directamente

$$T(n) = a_2 \times n^2$$

pero el modelado sería menos fiel.

Al ajustar a los puntos obtenidos por mínimos cuadrados obtendremos los valores de a_2, a_1 y a_0, es decir, las constantes ocultas. De esta manera, luego podremos saber cuánto tiempo, aproximadamente, utilizará el algoritmo para cualquier entrada de tamaño n. Además, estas constantes serán determinantes para comparar los tiempos requeridos por distintos algoritmos con el mismo orden de eficiencia.

Cómo obtener las constantes ocultas

Al igual que para el cálculo de la eficiencia empírica, podemos utilizar (entre otros) grace o gnuplot. Ilustraremos a continuación el uso de gnuplot para esta tarea.

Lo primero que tenemos que hacer es definir la función que queremos ajustar a los datos. En nuestro ejemplo, estamos hablando de una función cuadrática, pues hemos visto que el algoritmo tiene un orden de eficiencia $\mathcal{O}(n^2)$. Podemos definir esta función en gnuplot con la siguiente sentencia:

```
gnuplot> f(x) = a2*x*x+a1*x+a0
```

El siguiente paso es indicarle a gnuplot que calcule la regresión. Esto es simple, únicamente le tenemos que indicar

```
gnuplot> fit f(x) 'salida.dat' via a2,a1,a0
```

Tras unas pocas iteraciones del proceso de regresión de la función a ajustar se muestra el resultado del proceso. Tras el último paso, el resultado debería ser algo parecido a esto:

```
 Iteration 12
 WSSR        : 0.00707381        delta(WSSR)/WSSR   : -8.88702e-07
 delta(WSSR) : -6.28651e-09      limit for stopping : 1e-05
 lambda   : 0.000351302

resultant parameter values

a2              = 2.04333e-08
a1              = 7.98476e-06
a0              = -0.0268361

After 12 iterations the fit converged.
final sum of squares of residuals : 0.00707381
rel. change during last iteration : -8.88702e-07

degrees of freedom    (FIT_NDF)                        : 5
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0376133
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00141476

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
```

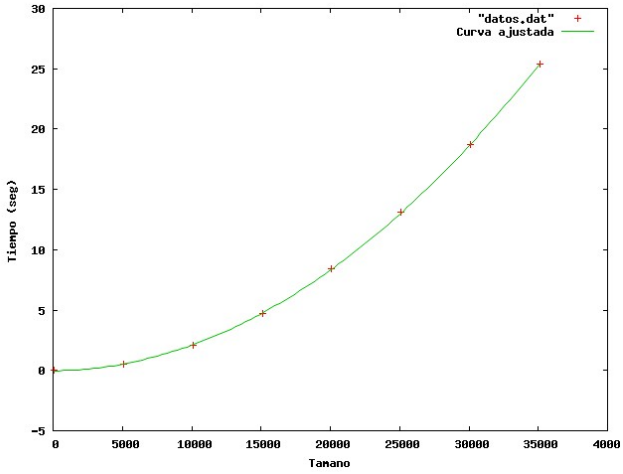

Figura 3: Enfoque híbrido: Ajuste de datos

```
a2              = 2.04333e-08      +/- 1.161e-10    (0.5681%)
a1              = 7.98476e-06      +/- 4.248e-06    (53.2%)
a0              = -0.0268361       +/- 0.03199      (119.2%)


correlation matrix of the fit parameters:

               a2     a1     a0
a2              1.000
a1             -0.962  1.000
a0              0.648 -0.798  1.000
```

Como vemos, hay una gran cantidad de información sobre los datos y sobre la bondad del ajuste. Probablemente, la parte que más nos interesa es cuáles son los valores de los parámetros. Éstos son fácilmente identificables en la sección `Final set of parameters`.

La siguiente pregunta es cómo de bien se ajusta esta función a nuestros datos. Para ello, podemos dibujar ambos en un único gráfico mediante la orden siguiente:

```
gnuplot> plot 'salida.dat', f(x) title 'Curva ajustada'
```

Como resultado tenemos el gráfico de la figura 3.

Por ejemplo, podemos estimar usando esa función ajustada que el tiempo de ejecución del algoritmo para una entrada de tamaño $n = 40000$ sería de $f(40000) = 32{,}98$ segundos.

4. Trabajo a realizar

Para alcanzar el objetivo previsto con esta práctica, el alumno deberá realizar el análisis empírico e híbrido de distintos algoritmos y recoger en una memoria el trabajo realizado. A continuación, se detallan las tareas concretas a realizar y el contenido de la memoria que debe preparar.

4.1. Algoritmos a estudiar

Ahora aplicaremos las pautas y conceptos descritos en las secciones anteriores para los análisis empírico e híbrido de un conjunto de algoritmos. Los algoritmos son los que se describen en la tabla 1. Son los siguientes algoritmos de ordenación de vectores: `burbuja`, `insercion`, `seleccion`, `heapsort`, `mergesort` y `quicksort`[2]

Algoritmo	Eficiencia
burbuja	$\mathcal{O}(n^2)$
inserción	$\mathcal{O}(n^2)$
selección	$\mathcal{O}(n^2)$
heapsort	$\mathcal{O}(n \log_2 n)$
mergesort	$\mathcal{O}(n \log_2 n)$
quicksort	$\mathcal{O}(n \log_2 n)$

Cuadro 1: Algoritmos a analizar

Se facilitan implementaciones de estos algoritmos en C++. Estas implementaciones están disponibles en la plataforma de docencia de la asignatura (Prado). Las implementaciones son sólo una base con la que trabajar. El alumno puede modificarlas para facilitar la obtención de las medidas de tiempos.

Para el estudio de estos algoritmos de ordenación se considerarán distintos tipos de datos base. Los tipos de datos base contemplados se indican en la tabla 2.

[2]El orden de efiencia del algoritmo quicksort para el peor caso es $\mathcal{O}(n^2)$, sin embargo, el análisis del caso promedio es $\mathcal{O}(n \log_2 n)$, que es el orden que consideraremos en este estudio.

Código numérico	Tipo de dato
0	entero: `int`
1	número en coma flotante de precisión sencilla: `float`
2	número en coma flotante en doble precisión: `double`
3	número en coma flotante muy grande: `long double`

Cuadro 2: Codificación de tipos de datos base

Cada alumno debe realizar los estudios sobre vectores de un único tipo de dato base. El tipo de dato básico que debe considerar se obtiene haciendo un cálculo sencillo: sumar las cifras de su DNI y calcular el módulo de esta suma sobre 4. Así, por ejemplo, el alumno con DNI 12.345.678, debe estudiar algoritmos para ordenar vectores de `int` ya que:

$$1 + 2 + \cdots + 8 = 36; \quad 36 \text{ mód } 4 = 0$$

4.2. Tareas

Esta práctica requiere que el alumno realice las siguientes tareas sobre cada uno de los algoritmos indicados.

1. Calcular la eficiencia empírica, siguiendo las indicaciones de la sección 2. Definir adecuadamente los tamaños de entrada de forma tal que se generen al menos 25 datos. Debe tenerse en cuenta que los tamaños con los que probar variarán mucho en función del orden de eficiencia de los algoritmos. Así, los tamaños a utilizar según el orden de eficiencia del algoritmo son los siguientes:

   - Orden $\mathcal{O}(n^2)$: desde 5000 hasta 125000 con saltos de 5000.
   - Orden $\mathcal{O}(n \log_2 n)$: desde 50000 hasta 1250000 con saltos de 50000.

   Una vez definidos los tamaños de las entradas se generarán casos de entrada correspondientes, y se usarán los mismos casos de entrada para medir los tiempos empleados por todos los algoritmos del mismo orden de eficiencia. Los resultados de las mediciones. Los resultados obtenidos se recogerán en tablas y gráficas, agrupados por órdenes de eficiencia. Por tanto, habrá que construir una tabla con los algoritmos de orden $\mathcal{O}(n^2)$ y otra con los $\mathcal{O}(n \log_2 n)$.

2. Para cada una de las tablas anteriores, generar un gráfico comparando los tiempos de los algoritmos.

Indique claramente el significado de cada serie. Para los cuatro algoritmos que realizan la misma tarea (los de ordenación), incluya también una gráfica con todos ellos, para poder apreciar las diferencias en rendimiento de algoritmos con diferente orden de eficiencia.

3. Calcular también la eficiencia híbrida de todos los algoritmos, siguiendo las pautas indicadas en la sección 3. Conviene probar el ajuste de los datos a curvas de otra eficiencia teórica (lineal, cuadrático, cúbico, etc.) y comparar la variación en la calidad del ajuste.

4. Otro aspecto interesante a analizar mediante este tipo de estudio es la variación de la eficiencia empírica en función de parámetros externos tales como: las opciones de compilación utilizada (con/sin optimización), el ordenador donde se realizan las pruebas, el sistema operativo, etc. Plantear estudios empíricos e híbridos variando alguno de estos aspectos y realizarlos, comparando los resultados entre sí y con respecto al estudio inicial.

5. Redactar una memoria del trabajo realizado, de acuerdo a la estructura descrita en la sección que sigue (4.3).

4.3. Estructura de la memoria

Todo el trabajo realizado debe redactarse en una memoria. La estructura de este documento será la siguiente:

1. Portada. Incluyendo las denominaciones de titulación, asignatura y práctica. También el nombre completo del alumno y su dirección de correo electrónico.

2. Objetivos. Descripción del objetivo de la práctica.

3. Diseño del estudio. Descripción de los tamaños e instancias de casos de entrada usadas para los análisis empíricos e híbridos. Descripción completa del entorno de análisis: hardware empleado, sistema operativo, compilador, etc. Descripción del método de medición de tiempo empleado

4. Algoritmos $\mathcal{O}(n^2)$. Esta sección se estructurará en las siguientes subsecciones:

   a) Resultados de mediciones de tiempo para todos los algoritmos de orden de eficiencia cuadrático, expresados en forma de tablas y gráficas.

   b) Ajuste del algoritmo i-ésimo. Para cada algoritmo concreto se obtendrá una curva de tiempo ajustando la función teórica a los datos empíricos obtenidos. Indicar los resultados del ajuste. Probar a realizar el ajuste de los datos empíricos a curvas teóricas distintas, por ejemplo, lineales o cúbicas.

   c) Comparar las curvas de tiempo de todos los algoritmos del grupo indicando cuál sea el más eficiente, si es posible identificar una opción mejor.

   d) Análisis de eficiencia empírica e híbrida modificando factores de hardware, sistema operativo o compilación. Modificar alguno de los parámetros referidos que afecten a los tiempos de ejecución y reproducir los análisis empíricos e híbridos. Comparar los resultados obtenidos con los de la subsección previa.

5. Algoritmos $\mathcal{O}(n \log_2 n)$. Incluir una sección con la misma estructura que la anterior, pero considerando los algoritmos de eficiencia $\mathcal{O}(n \log_2 n)$.

6. Estudio comparativo de curvas de eficiencia de todos los algoritmos.

7. Conclusiones. Enumerar y describir las principales conclusiones derivadas del trabajo realizado en esta práctica.

La memoria debe entregarse en formato pdf.

Se recuerda que esta práctica debe ser realizada individualmente por cada alumno. El documento se entregará a través de la actividad correspondiente incluida en la página de la asignatura de la plataforma Prado. La fecha límite para entregar la memoria es el día **19 de marzo de 2023** a las 23:59 horas.