UNIVERSIDAD DE GRANADA

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

Departamento de Ciencias de la Computación e Inteligencia Artificial

Algorítmica

Guión de Prácticas

Práctica 1: Análisis de Eficiencia de Algoritmos

Curso 2023-2024

Grado en Informática

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 equipo de estudiantes 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). El análisis que nos interesa será el del peor caso. Así, tras obtener la expresión analítica de $T(n)$, calcularemos el orden de eficiencia del algoritmo empleando la notación $O()$.

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 la posición inicial a la final de éste, 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 $(final - 1) - (i + 1) + 1$, es decir, $final - i - 1$. 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:

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

Renombrando en la ecuación (1) *final* como n e *inicial* como 1, pasamos a resolver la siguiente ecuación:

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

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

El código del algoritmo MergeSort es el siguiente:

```
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==0);
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==0);
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 $O(n)$. 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 $O(n)$.

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 fusión que como se puede observar tiene una eficiencia de $O(n)$.

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

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

Vamos a resolver la recurrencia en la ec.(5):

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

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

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

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

Renombrando $T(2^m) = t_m$ obtenemos

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

La ecuación de recurrencia que se deduce de la ec.(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 $T(n) \in O(n \log_2(n))$.

1.2.1. Comentarios adicionales sobre la constante UMBRALMS

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 donde internamente se debe gestionar una 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. En el caso del algoritmo para calcular los números de la sucesión de Fibonacci, el tamaño se corresponde con el valor del entero. En el caso del algoritmo para resolver el problema de las torres de Hanoi, el tamaño se corresponde con el valor del entero que representa el número de discos. Para el caso de los algoritmos (Floyd) que calculan los caminos mínimos entre todos los pares de nodos en un grafo dirigido, el tamaño es el número de nodos del grafo.

2.1. Cálculo del tiempo de ejecución

Para obtener el tiempo empírico de una ejecución de un algoritmo captuar el valor del reloj antes de la ejecución del algoritmo al que queremos medir el tiempo así como 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:

```
tantes = Captura el valor del reloj antes de la llamada al algoritmo

  // Llama al algoritmo de ordenación burbuja
  burbuja(T,0,n);

tdespues = Captura el valor del reloj después de la ejecución del algoritmo
```

La diferencia entre los dos los dos instantes de reloj nos dice el tiempo dedicado a la ejecución del algoritmo

```
 (tdespues - tantes)
```

2.2. Problemas 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:

```
const int NUM_VECES=10000;

tantes= Captura el reloj antes

for (int i=0; i<NUM_VECES;i++)
   //Sentencia cuyo tiempo se pretende medir

tdespues = Captura el reloj despues
tiempo_transcurrido= (tdespues-tantes)/ NUM_VECES
```

2.3. Cómo calcular el tiempo utilizando la biblioteca chrono

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

```
1 #include <chrono>
2 using namespace std::chrono;
3
4 high_resolution_clock::time_point tantes, tdespues;
5 duration<double> transcurrido;
```

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

```
1 tantes = high_resolution_clock::now();
2    //sentencia o programa a medir
3 tdespues = high_resolution_clock::now();
4
5 transcurrido = duration_cast<duration<double>>(tdespues - tantes);
6 cout << "el tiempo empleado es " << transcurrido.count() << " segundos." <<
     endl;}
```

En este caso al compilar se debe incluir la directiva `-std=gnu++0x`.
Para otras formas de medir el tiempo, consultar por ejemplo:
`https://ichi.pro/es/8-formas-de-medir-el-tiempo-de-ejecucion-en-c-c-79591185633529`

2.4. Cómo proceder

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 una "macro", como la que se muestra a continuación:

```
#!/bin/csh -vx
echo "" >> salida.dat
@ i = 1000
while ( $i < 100000 )
./insercion $i >> salida.dat
@ i += 1000
end
```

Así en esta macro se va a escribir 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 ejecutarla debemos darle permisos de ejecución mediante la sentencia `chmod +x macro`, y a continuación ejecutarla como `./macro`.

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

2.5. Cómo mostrar los resultados

Para mostrar la eficiencia empírica haremos uso de tablas que recojan el tiempo invertido para cada caso y también de gráficas. Para mostrar los datos en gráfica pondremos en el eje X (abscisa) el tamaño de los casos y en el eje Y (ordenada) el tiempo, medido en segundos, requerido por la implementación del algoritmo. Para hacer esta representación de los resultados podemos hacer uso, bien del software grace (xmgrace) o de gnuplot (entre otras opciones). El software grace trabaja sobre Linux aunque ha sido exportado a plataformas como Windows 2000/NT/XP, y se puede descargar desde `http://plasma-gate.weizmann.ac.il/Grace/`. Gnuplot 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 pares de elementos (x, y) separados por espacios en blanco, como se indica a continuación. 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 línea de comandos hacemos una llamada a `gnuplot`, y para poder representar estos puntos podemos ejecutar el comando

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

Como resultado aparecerá una nueva ventana como la de la figura 1. El comando `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 grafico (en nuestro caso `Eficiencia XXX`.)

Además podemos etiquetar los valores que representa el eje x y el eje y. Para ello podemos ejecutar los siguientes comandos:

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

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

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

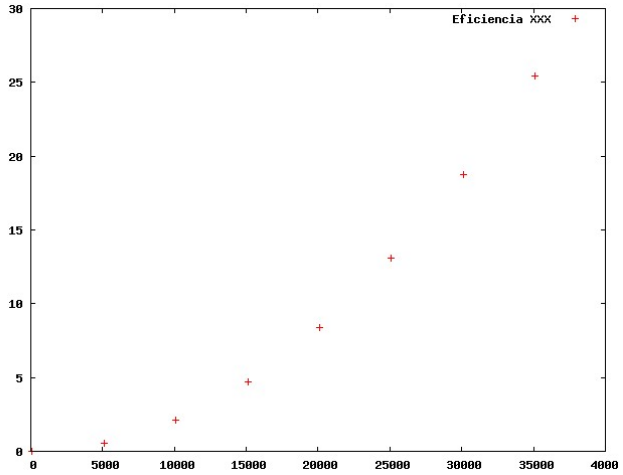

Figura 1: Fichero salida.dat

```
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 ecuación del 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 ecuación 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:

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

aunque también podríamos usar directamente

$$T(n) = a_0 \times n^2$$

Al ajustar a los puntos obtenidos por mínimos cuadrados obtendremos los valores de a_0, a_1 y a_2, es decir, las constantes ocultas. De esta manera, luego podremos saber cuánto tiempo aproximadamente utilizará el algoritmo para cualquier entrada de tamaño n.

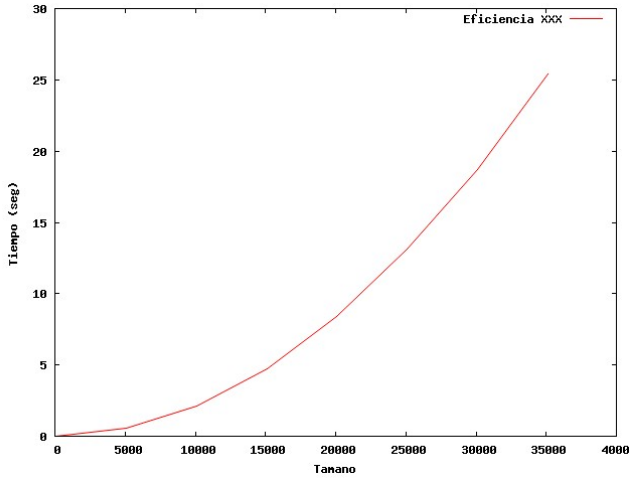

Figura 2: Fichero salida.dat etiquetado

3.1. 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 $O(n^2)$. Podemos definir esta función en gnuplot mediante el siguiente comando:

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

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

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

Lo que se debe ver es el resultado, tras unas pocas iteraciones, del proceso de regresión de la función a ajustar. El último paso debería ser algo como:

```
 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

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

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


correlation matrix of the fit parameters:

               a0     a1     a2
a0              1.000
a1             -0.962  1.000
a2              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 importa es cuáles son los valores de los parámetros. Éstos son fácilmente identificables bajo la sección `Final set of parameters`.

La siguiente pregunta es cómo se ajusta esta función a nuestros datos. Para ello, podemos dibujar ambos en un único gráfico mediante el comando siguiente. Como resultado tenemos el gráfico de la figura 3.

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

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. Tareas a realizar

Ahora aplicaremos las pautas y conceptos descritos en las secciones anteriores para el análisis de un conjunto de algoritmos. Estos ocho algoritmos son los que se describen en la tabla 1. `burbuja`, `insercion`, `mergesort` y `quicksort` son algoritmos de ordenación de vectores; `floyd` es un algoritmo que calcula el costo del camino mínimo entre cada par de nodos de un grafo dirigido; `fibonacci` es un algoritmo para calcular los números de la sucesión de Fibonacci, y `hanoi` es un algoritmo para resolver el famoso problema de las torres de Hanoi.

Las implementaciones de los algoritmos están disponibles en la plataforma de docencia de la asignatura. Una vez compilados los códigos (se pueden modificar ligeramente para que

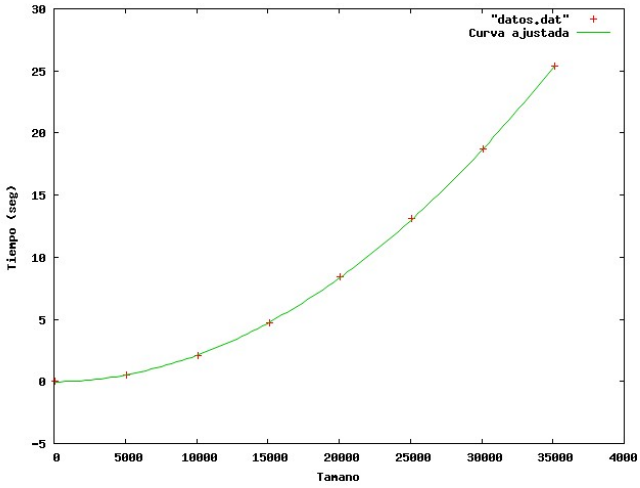

Figura 3: Enfoque híbrido: Ajuste de datos

Algoritmo	Eficiencia	
burbuja	$O(n^2)$	
insercion	$O(n^2)$	
mergesort	$O(n \log n)$	
quicksort	$O(n \log n)$	
floyd	$O(n^3)$	
fibonacci	$O((\frac{1+\sqrt{5}}{2})^n)$	$\frac{1+\sqrt{5}}{2} \approx 1{,}618$
hanoi	$O(2^n)$	

Tabla 1: Algoritmos a emplear

el posterior procesamiento de los datos sea más automático), se deben realizar las siguientes tareas.

Para el estudio de estos algoritmos de ordenación se considerarán distintos tipos de datos base. Los tipos de datos base contemplados serán: `int, float, double, string`

1. Calcule la eficiencia empírica, siguiendo las indicaciones de la sección 2. Defina adecuadamente los tamaños de entrada de forma tal que se generen al menos 25 datos. Tenga 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 tipo de algoritmo son los siguientes:

   - Orden $O(n^2)$: desde 5000 hasta 125000 con saltos de 5000.
   - Orden $O(n \log n)$: desde 50000 hasta 1250000 con saltos de 50000.
   - Orden $O(n^3)$: desde 50 hasta 1250 con saltos de 50.
   - Orden $O((\frac{1+\sqrt{5}}{2})^n)$: desde 2 hasta 50 con saltos de 2.
   - Orden $O(2^n)$: desde 3 hasta 33 con saltos de 1.

- Para abordar el problema de ordenación con datos de tipo string, se considerará como entradas para el algoritmo la obra "Don Quijote de la Mancha", de Miguel de Cervantes, que la podremos descargar del proyecto Gutenberg en su versión txt del siguiente enlace:

  https://www.gutenberg.org/cache/epub/2000/pg2000.txt.

  Partiremos de un vector de string, donde en cada posición irán cada una de las palabras del libro. En este caso, el Quijote tiene un total de 202.308 palabras, por lo que los intervalos irán desde 12.308 hasta 202.308 con saltos de 10.000 palabras.

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.

Por tanto, habrá que construir una tabla con los algoritmos de orden $O(n^2)$, otra con los $O(n \log n)$, otra con los $O(n^3)$ y otra con los exponenciales $O((\frac{1+\sqrt{5}}{2})^n)$ y $O(2^n)$).

2. Con cada una de las tablas anteriores, genere 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. Para los algoritmos de ordenación incluir una gráfica donde se comparen los tiempos de ejecución cuando varían los tipos de datos de datos considerados.

4. Calcule también la eficiencia híbrida de todos los algoritmos, siguiendo las pautas indicadas en la sección 3. Pruebe también con otros ajustes que no se correspondan con la eficiencia teórica (ajuste lineal, cuadrático, etc) y compruebe la variación en la calidad del ajuste.

5. A partir de las distintas gráficas realizar un análisis comparativo de los distintos algoritmos de ordenación.

6. 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. Proponga algún estudio de este tipo para alguno de los algoritmos considerados y llévelo a cabo.

5. Estructura de la memoria

Escriba una memoria con todas las tareas realizadas, incluyendo todos los detalles relevantes referidos a los cálculos de eficiencia empírica e híbrida.

La estructura de este documento será la siguiente:

1. Portada. Incluyendo las denominaciones de titulación, asignatura y práctica. También el nombre completo del grupo y los alumnos que lo forman con su dirección de correo electrónico.

2. Participación: En esta sección deben aparecer los nombres de todos los miembros del equipo que hayan participado en la realización de la práctica, así como el porcentaje de participación de cada miembro junto a los ítems en los que ha participado.

3. Objetivos. Descripciónn del objetivo de la práctica.

4. 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.

5. Algoritmos considedados. Esta sección se estructurará en 4 subsecciones (cuadráticos, linear-logarítmicos, cúbicos y exponenciales), para cada una se considerará:

   a) Resultados de mediciones de tiempo para todos los algoritmos expresados en forma de gráficas (se pueden incluir también las tablas si se consideran necesarias).

   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.

   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. Razonar la respuesta

   d) En caso de los algoritmos de ordenación, análisis de eficiencia empírica e híbrida en función de los tipos de datos utilizados

   e) Análisis de eficiencia empírica e híbrida en función factores de hardware, sistema operativo o compilación. Comparar los resultados obtenidos con los de la subsección previa.

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. 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 8 de marzo de 2024 a las 23:59 h