UNIVERSIDAD DE GRANADA

Algorítmica

Grado en Ingeniería Informática

DECSAI

Guion de Prácticas 1

Cálculo de la eficiencia de algoritmos

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Cálculo de la eficiencia de algoritmos

1. Objetivo

El objetivo de la práctica consiste en que el alumno sea capaz de analizar y calcular la eficiencia teórica, práctica e híbrida de algoritmos, iterativos y recursivos. Para ello, se expondrá al menos un problema que será resuelto en clase por el profesor. Posteriormente, se expone un conjunto de problemas que deberán ser resueltos por el estudiante.

2. Cálculo de la eficiencia teórica

Se debe calcular la eficiencia teórica de los algoritmos propuestos a continuación. Utilícese la explicación dada por el profesor para resolver el ejemplo siguiente como plantilla para solucionar cada problema.

2.1. Algoritmos iterativos: Ejemplo

Calcule la eficiencia, en el peor caso, para el siguiente algoritmo:

```
76 void OrdenaBurbuja(int *v, int n) {
77
78     int i, j, aux;
79     bool haycambios= true;
80
81     i= 0;
82     while (haycambios) {
83
84      haycambios=false; // Suponemos vector ya ordenado
85      for (j= n-1; j>i; j--) { // Recorremos vector de final a i
86
87          if (v[j-1]>v[j]) { // Dos elementos consecutivos mal ordenados
88           aux= v[j]; // Los intercambiamos
89           v[j]= v[j-1];
90           v[j-1]= aux;
91           haycambios= true; // Al intercambiar, hay cambio
92          }
93      }
94      i++;
95     }
96 }
```

Figura 1: Algoritmo de ordenación por burbuja

Solución:

El algoritmo resuelve el problema de ordenación de un subvector v, con un total de n componentes útiles.

Variable o variables de las que dependen el tamaño del caso: *El tamaño del problema depende de un único parámetro* ***n****, que es la longitud del subvector a ordenar.*

Comenzamos analizando la función desde la parte más interna. Las líneas 88-91 contienen accesos a posiciones de un array (v[j], v[j-1]…), y asignaciones de tipos básicos. Todas son operaciones elementales y, por tanto, son O(1). La secuencia de todas estas operaciones también es O(1), aplicando la regla del máximo.

La sentencia condicional "if" de la línea 87: Como no hay parte "else", podemos suponer que el peor caso para la función se dará cuando se cumpla la condición, y podemos asumir que en el peor de los casos esto se dará siempre (basta con que el array esté ordenado al revés en la iteración actual). La evaluación de la condición se corresponde con una comparación "<" y dos accesos a componentes de un array. Todas son operaciones elementales y tienen un orden de eficiencia O(1). De este modo, el "if" completo tiene una eficiencia igual al orden de eficiencia de comprobar la condición + orden de eficiencia de las sentencias de dentro de la estructura. Ambos son O(1), por lo que aplicando la regla del máximo, la eficiencia del algoritmo entre las líneas 87-92 (inclusive) es O(1).

El bucle "for" de la línea 85: Su inicialización es O(1) por ser una operación elemental, al igual que la comprobación y la actualización. En el peor de los casos, el bucle se ejecuta n-1 veces (cuando i=0, al principio del algoritmo). Su orden de eficiencia será (n-1)(O(comprobación) + O(actualización) + O(sentencias del bucle)) que, como todas son O(1), aplicando la regla del máximo la eficiencia es O(n-1), que equivale a O(n).*

*El bucle "while" de la línea 82: Se ejecutará mientras su condición sea cierta. La evaluación de esta condición es O(1). Como antes hemos asumido que se entra al "if" de la línea 87 siempre, entonces también se dará que la variable "hayCambios" valga true siempre, hasta que el bucle for no se ejecute ninguna vez (sólo se dará cuando i=n-1). De este modo, sabemos que el bucle while se ejecutará un máximo de n-1 veces. La sentencia "i++" de la línea 94 es O(1) por ser operación elemental, por lo que las sentencias del bucle while son max(O(n), O(1)), y por tanto la eficiencia del while será (n-1)*O(n)= O(n*(n-1)), que equivale a $O(n^2)$. Las operaciones entre las líneas 76 y 81 son todas elementales y, por tanto, O(1), por lo que la eficiencia total del algoritmo es $O(n^2)$ tras aplicar de nuevo la regla del máximo.*

2.2. Algoritmos recursivos: Ejemplo

Calcule la eficiencia en el peor caso para el siguiente algoritmo (Figura 2 y Figura 3):

```
 8  void fusionaMS(double *v, int posIni, int centro, int posFin, double *vaux) {
 9
10      int i= posIni;
11      int j= centro;
12      int k= 0;
13
14      while (i<centro && j<=posFin) {
15          if (v[i]<=v[j]) {
16              vaux[k]= v[i];
17              i++;
18          } else {
19              vaux[k]= v[j];
20              j++;
21          }
22          k++;
23      }
24
25      while (i<centro) {
26          vaux[k]= v[i];
27          i++, k++;
28      }
29      while (j<=posFin) {
30          vaux[k]= v[j];
31          j++, k++;
32      }
33
34      memcpy(v+posIni, vaux, k*sizeof(double));
35  }
```

Figura 2: Función de fusión del algoritmo MergeSort

```
38  void MergeSort(double *v, int posIni, int posFin, double *vaux) {
39
40      if (posIni>=posFin) return;
41
42      int centro= (posIni+posFin)/2;
43
44      MergeSort(v, posIni, centro, vaux);
45      MergeSort(v, centro+1, posFin, vaux);
46      fusionaMS(v, posIni, centro+1, posFin, vaux);
47  }
```

Figura 3: Función principal del algoritmo MergeSort

Solución:

El algoritmo resuelve el problema de ordenación de un subvector v entre las posiciones posIni y

posFin (inclusive), usando otro vector vaux como auxiliar.

Variable o variables de las que dependen el tamaño del caso: *El tamaño del problema depende del número de componentes del subvector a ordenar. Por tanto, n=posFin-posIni+1.*

Es un algoritmo recursivo. Por tanto, llamemos ***T(n)*** *al tiempo de ejecución que tarda el algoritmo "MergeSort" en resolver un problema de tamaño "n". Según el algoritmo, existen dos casos (uno general, y un caso base):*

- *Caso base: Cuando n<=1. Entra al "if" de la línea 40 y termina. Según las reglas de análisis de eficiencia, en este caso base de la recurrencia el tiempo de ejecución es O(1).*
- *Caso general: Es el que interesa. Se da cuando n>1 . En este caso:*
  - *El algoritmo calcula la parte central del subvector en "centro". Esta operación es O(1) porque todas las sentencias son operaciones simples.*
  - *Luego hace una llamada recursiva en la línea 44, para resolver un subproblema desde posIni hasta centro, cuyo tamaño es n/2. Si T(n) es el tiempo que tarda el algoritmo para resolver el problema de tamaño n, entonces la llamada recursiva de la línea 44 tendrá un tiempo de ejecución T(n/2).*
  - *Luego hace otra llamada recursiva en la línea 45, para resolver un subproblema desde centro+1 hasta posFin, cuyo tamaño asintótico también se puede aproximar por n/2. Al igual que en la línea anterior, la llamada recursiva de la línea 45 tendrá un tiempo de ejecución T(n/2).*
  - *Luego hace una llamada a la función "fusionaMS". Asumiendo que memcpy es O(1) (o también O(n), según su implementación interna), siguiendo las reglas del cálculo de eficiencia, la función íntegra tiene un orden O(n).*
  - *La llamada a la función de la línea 46 se hace sobre un subvector de n componentes, por lo que también podemos asumir que la llamada a esta función es O(n).*

Por tanto, podemos aproximar T(n) en el caso general como T(n)= 2T(n/2)+n . Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de MergeSort.

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, $n=2^m$ *, quedando:*

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

Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

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

Resolvemos primero la "parte homogénea":

$$T(2^m)= 2T(2^{m-1}) = 0$$
$$x^m -2x^{m-1} = 0$$
$$x^{m-1} (x-2) = 0$$

Y obtenemos la parte homogénea del polinomio característico como $P_H(x)= (x-2)$

Para resolver la "parte no homogénea", debemos conseguir un escalar "b_1*" y un polinomio "*$q_1(m)$*"*

de modo que:

$$2^m = b_1{}^m q_1(m)$$

Es fácil haciendo $b_1 = 2$ y $q_1(m)=1$, donde el grado del polinomio es $d_1=0$

El polinomio característico se obtiene como: $P(x)= P_H(x)(x-b_1)^{d1+1}= (x-2)(x-2)= (x-2)^2$

De este modo, tenemos $r=1$ única raíz diferente, con valor $R_1= 2$ y multiplicidad $M_1= 2$

Aplicando la fórmula de la ecuación característica, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

$$T(2^m)=\sum_{i=1}^{r}\sum_{j=0}^{M_i-1} c_{ij} R_i^m m^j = c_{10}2^m + c_{11}2^m m$$

Ahora deshacemos el cambio de variable para volver al espacio de tiempos inicial:

$$T(n)=c_{10}n+c_{11}n\log_2(n)$$

Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es ***O(n·log(n)).***

3. Cálculo de la eficiencia práctica

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de caso, y calcular su tiempo de ejecución. Para ello haremos uso de la biblioteca **chrono** incluida en el estándar **C++11**. No olvide compilar cada programa con el compilador **g++** incluyendo este estándar (o superior) antes de compilar cada programa.

3.1. Ejercicio guiado

El tiempo de ejecución lo mediremos como la diferencia entre el instante de tiempo justo anterior al inicio del algoritmo, y el instante justamente posterior. Para ello, necesitaremos declarar las siguientes variables en nuestro programa:

```
#include<chrono>
...

chrono::time_point<std::chrono::high_resolution_clock> t0, tf; // Para medir el tiempo de ejecución
```

Los instantes de tiempo, y la duración, los obtendremos de la siguiente forma:

```
// Comenzamos a medir el tiempo
t0= std::chrono::high_resolution_clock::now();

// Aquí vendría la ejecución del algoritmo

// Terminamos de medir el tiempo
tf= std::chrono::high_resolution_clock::now();


// Medimos la duración, en MICROSEGUNDOS
unsigned long duration;
duration= std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();
```

Ejecute el código del ejemplo **Burbuja.cpp** o del ejemplo **MergeSort.cpp** proporcionados por el profesor para medir el tiempo de ejecución del algoritmo de ordenación por Burbuja y el algoritmo de MergeSort con diversos tiempos de ejecución.

Se deben medir diferentes tiempos de ejecución (para al menos 5-10 tamaños de casos diferentes), de modo que **descartemos todos los tiempos de ejecución igual a 0 que obtengamos**. Un tiempo de ejecución nunca es de 0 unidades; si esto ocurre, es debido a que la precisión del reloj no es suficiente como para poder medir el tiempo que tarda en ejecutarse un algoritmo. Los tamaños de caso y sus correspondientes tiempos de ejecución serán guardados en un fichero para su posterior procesamiento. Por ejemplo, ejecute los siguientes programas facilitados por el profesor en la práctica:

```
> make
> ./Burbuja.bin burbuja.txt 12345  1000  2000  3000  4000  5000  6000  7000  8000  9000  10000
> ./MergeSort.bin mergesort.txt  12345  10000  20000  30000  40000  50000  60000  70000  80000  90000  100000
```

Estas líneas crearán dos ficheros, burbuja.txt y mergesort.txt, con dos columnas de datos: La primera, conteniendo el tamaño de la instancia del problema (pasado por argumento a cada programa). La segunda, el tiempo de ejecución (en este caso en microsegundos) de cada instancia ejecutada. Los valores de los tiempos obtenidos dependerán, además del tamaño de caso usado, del PC donde se ejecuten y de su frecuencia de reloj. Por ejemplo, se podrían obtener las siguientes tablas de datos:

Tabla 1: Tam. de casos y tiempos de ejecución en Burbuja

Tamaño del caso (n)	Tiempo de ejecución (en us)
1000	2476
2000	6132
3000	13752
4000	25212
5000	40770
6000	61886
7000	85037
8000	113978
9000	148216
10000	187329

Tabla 2: Tam. de casos y tiempos de ejecución en MergeSort

Tamaño del caso (n)	Tiempo de ejecución (en us)
10000	2220
20000	1972
30000	3198
40000	4262
50000	5419
60000	6580
70000	7864
80000	9009
90000	10265
100000	11546

4. Eficiencia híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado 3 se corresponde con la eficiencia teórica calculada en el apartado 2. Partiremos de la definición de orden de eficiencia, en la que se indica que, cuando el tamaño del caso es suficientemente grande (tiende a infinito), entonces el tiempo de ejecución del algoritmo T(n) está acotado por la función del orden de eficiencia O(f(n)), de modo que T(n)<= K·f(n), con K una constante real positiva. A "K" se le denomina *constante oculta*.

4.1. Cálculo de la constante oculta

En apartados previos hemos calculado el orden O(f(n)) de cada algoritmo propuesto. Este orden O(f(n)) quiere decir que existe una constante **K**, para cada algoritmo, tal que el tiempo **T(n)** de ejecución del mismo para un tamaño de caso **n** es:

$$T(n) <= K*f(n)$$

El cálculo de la constante **K** se calcula despejando e igualando la fórmula anterior:

$$K = T(n)/f(n)$$

Este valor de K se calculará para todas las ejecuciones del mismo algoritmo para distintos tamaños de caso, produciendo valores aproximados para K. Aproximaremos el valor final de K como la media de todos estos valores. Las siguientes gráficas muestran un ejemplo del cálculo de este valor en LibreOffice Calc, para los resultados del algoritmo de ordenación por burbuja y de MergeSort previamente expuestos en apartados anteriores, donde $f(n)=n^2$ en el algoritmo de burbuja, y $f(n)=n*\log(n)$ en el algoritmo de MergeSort. El valor final de K será el valor promedio obtenido en todas las ejecuciones.

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	2476	0,002476	1768,20779456412
2000	6132	0,001533	7072,83117825649
3000	13752	0,001528	15913,8701510771
4000	25212	0,00157575	28291,324713026
5000	40770	0,0016308	44205,194864103
6000	61886	0,001719055556	63655,4806043084
7000	85037	0,00173544898	86642,181933642
8000	113978	0,00178090625	113165,298852104
9000	148216	0,00182982716	143224,831359694
10000	187329	0,00187329	176820,779456412
	K promedio:	0,001768207795	

Figura 4: Cálculo de la constante oculta mínima y de los tiempos de ejecución teóricos aproximados para burbuja

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
10000	2220	0,0555	1054,68819046953
20000	1972	0,02292474131	2268,122771641
30000	3198	0,02380994258	3541,4751859897
40000	4262	0,02315267515	4853,73832468589
50000	5419	0,02306462904	6194,93521367967
60000	6580	0,02295169427	7559,18954408524
70000	7864	0,02318691102	8942,61844782264
80000	9009	0,02296765923	10342,4622121796
90000	10265	0,02302179502	11756,6574017125
100000	11546	0,023092	13183,6023808691
	K promedio	0,02636720476	

Figura 5: Cálculo de la constante oculta mínima y de los tiempos de ejecución teóricos aproximados para mergesort

4.2. Comparación gráfica de órdenes de eficiencia

Asumiendo que hemos calculado el orden de eficiencia del algoritmo de ordenación por burbuja como $O(n^2)$ y de MergeSort como $O(n \cdot \log(n))$, y que el valor de la constante oculta en promedio es el valor K calculado, a continuación comprobaremos experimentalmente que el orden O(f(n)) calculado con una constante oculta superior siempre será mayor que el tiempo de ejecución real del algoritmo (basta con

escoger la K con valor mayor al estimado), por lo que hemos conseguido acotar el tiempo de ejecución del mismo y estimar, en el futuro, cuánto tardará en ejecutarse para otros tamaños de casos en el peor de los casos. Se muestra la siguiente gráfica generada por LibreOffice Calc para el tiempo de ejecución real y teórico con la constante oculta calculada.

Para ello, en LibreOffice Calc seleccionamos todas las filas de las columnas "n", "Tiempo" y "Tiempo teórico", y pulsamos sobre el icono "Insertar gráfico". A continuación, en las opciones escogemos gráfico de tipo "XY (dispersión)", sub-opción "Puntos y líneas". Finalmente, pulsamos sobre "Terminar" y se generarán las siguientes gráficas:

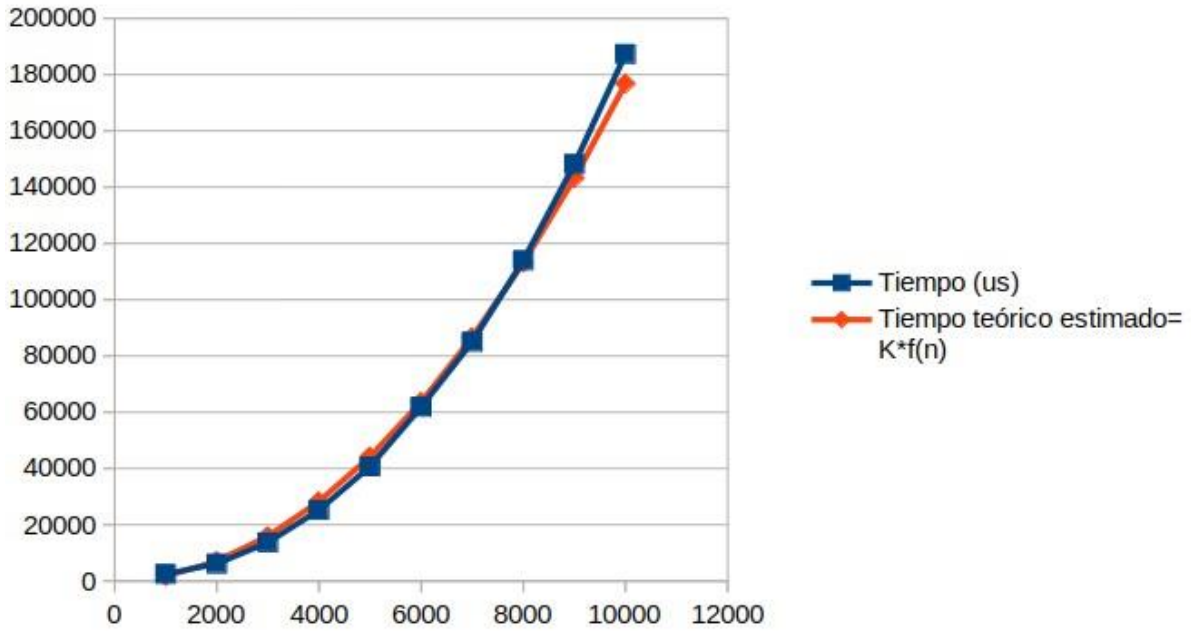

Figura 6: Tiempo de ejecución real vs teórico en burbuja

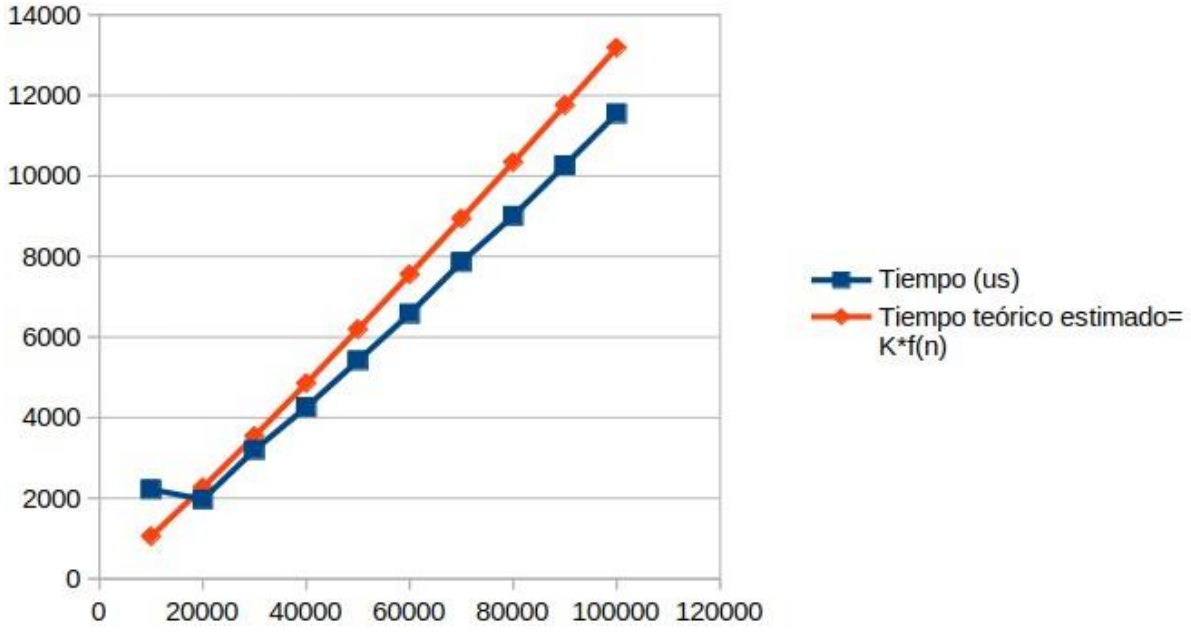

Figura 7: Tiempos de ejecución real vs teórico en mergesort

Podemos observar cómo las gráficas coinciden en el algoritmo de burbuja, por lo que se ha calculado bien tanto el orden teórico como la constante oculta. En el algoritmo MergeSort, esta buena aproximación no se aprecia tan bien como en burbuja. Esto es debido a que los tamaños de casos escogidos son muy pequeños en comparación con la eficiencia práctica del algoritmo, por lo que sería necesario repetir la experimentación para valores más grandes (100000, 500000, 1000000, etc.) de modo que se aprecie debidamente la tendencia de n*log(n).

5. Ejercicios propuestos

Se debe analizar la eficiencia teórica, práctica e híbrida de los siguientes algoritmos de ordenación, de acuerdo a los criterios dados en el apartado 6 de evaluación. Cada uno de ellos tiene un enlace a una página de Wikipedia donde se puede encontrar el funcionamiento del algoritmo. Como para calcular la eficiencia práctica es necesaria la implementación de los algoritmos en C, se permite el uso tanto de implementaciones encontradas en la literatura o en internet (poner la referencia) como de herramientas de IA generativa (comprobar que el código generado por la herramienta es correcto).

1. **Conteo (Counting Sort)**
2. **Inserción (Insertion Sort)**
3. **Rápido (Quicksort)**
4. **Selección (Selection Sort)**
5. **Ordenamiento Shell (Shell sort)**

6. Evaluación de la práctica

La valoración de la práctica se dará como una calificación numérica entre 0 y 10.

Se deben resolver todos los ejercicios propuestos en el apartado 5 de este guion de la siguiente forma:

1. Calcular el orden de eficiencia del algoritmo en el caso peor. Esto requiere:
    a. Analizar de qué variable o variables dependen el tamaño del caso "n". Se valorará que el análisis de las variables, así como el cálculo de "n" en función de estas variables, sea correcto.
    b. Analizar el algoritmo siguiendo las reglas de cálculo de la eficiencia, desde la parte interna hasta la parte externa. Se valorará la correcta toma de decisiones para valorar sentencias condicionales, repetitivas, secuencias de sentencias y llamadas a funciones.
    c. Establecer la ecuación en recurrencias del algoritmo (sólo en algoritmos recursivos). Se valorará que la ecuación resultante sea la correcta, atendiendo al paso anterior.
    d. Resolver la ecuación en recurrencias (sólo en algoritmos recursivos). Se valorará que el procedimiento usado esté correctamente explicado y que la solución final sea correcta.
    e. Dar el orden de eficiencia O del problema. Se valorará que la conclusión del orden de eficiencia final sea consistente con las decisiones tomadas en los pasos previos, y que sea correcto.

2. Calcular la constante oculta, escogiendo libremente distintos tamaños de casos que permitan verificar su correcto cálculo. Se valorará la explicación sobre cómo calcular la constante oculta, así como su validez en cada uno de los problemas planteados.
3. Verificar gráficamente el comportamiento del algoritmo en el caso peor, para el tiempo de ejecución real vs el tiempo de ejecución teórico calculado con la eficiencia híbrida. Se valorará que el cálculo del tiempo de ejecución teórico sea correcto, así como la explicación y conclusiones del estudiante con respecto a la gráfica resultante.
4. Compare la eficiencia teórica e híbrida entre todos los algoritmos analizados (todos resuelven el problema de ordenación). Indique qué algoritmo o algoritmos es/son más eficiente/s asintóticamente, y cuál/es de ellos es mejor considerando la eficiencia híbrida.

7. Entrega y presentación de la práctica

Se deberá entregar un documento (memoria de prácticas) realizado en equipos de 5 personas, conteniendo los siguientes apartados:

1. Solución a los problemas propuestos de eficiencia teórica, para cada ejercicio propuesto.
2. Solución a los problemas propuestos de eficiencia práctica, para cada ejercicio propuesto.
3. Solución a los problemas anteriores con eficiencia híbrida: cálculo de constantes y comparación gráfica, para todos los ejercicios propuestos.
4. Comparación entre la eficiencia teórica e híbrida entre los algoritmos de ordenación, indicando cuál de ellos es más eficiente de forma justificada.

La práctica deberá ser entregada por PRADO, en la fecha y hora límite explicada en clase por el profesor. No se aceptarán, bajo ningún concepto, prácticas entregadas con posterioridad a la fecha límite indicada. La entrega de PRADO permanecerá abierta con, al menos, una semana de antelación antes de la fecha límite, por lo que todo alumno tendrá tiempo suficiente para entregarla. **La práctica deberá ser enviada por un único integrante del equipo de trabajo**. Y deberá contener los nombres de todos los autores del mismo. Si algún equipo sufre una baja, o considera que alguno de los miembros no ha trabajado lo suficiente para ser incluido entre los autores de la práctica, deberá comunicar tal situación al profesor.

El profesor, en clase de prácticas, realizará controles de las prácticas a discreción, que consistirán en presentaciones de los estudiantes de cada equipo (powerpoint) y/o entrevistas individuales con el fin de asegurar de que los estudiantes alcanzan las competencias deseadas. Estas entrevistas y/o presentaciones se realizarán en las sesiones de evaluación de prácticas, previamente anunciadas en clase por el profesor.

La **no asistencia** a una sesión de evaluación de prácticas por un estudiante supondrá la **calificación de 0 (no presentado) a la práctica que deba presentar, independientemente de la calificación obtenida por el resto de su equipo**.

IMPORTANTE: Antes de las sesiones de evaluación, cada estudiante deberá prepararse para:

- Realizar una presentación "powerpoint" de la práctica de **máximo 10 minutos**.
- Conocer a fondo todos los algoritmos resueltos, así como los pasos para realizar el análisis de eficiencia teórico, práctico e híbrido de cada uno de ellos.
- Conocer las respuestas a las preguntas requeridas en el apartado 6.

La calificación otorgada será la misma para todos los integrantes del grupo, por lo que si alguno de los autores no demuestra el nivel exigido en la evaluación, se verá afectada la calificación de TODOS los miembros del equipo.