UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Reto 1: Eficiencia

J. Fdez-Valdivia
Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Usando la ***notación O,*** *determinar la eficiencia de las siguientes* `funciones`:

(a)

```
  void eficiencia1(int n)
{
  int x=0; int i,j,k;
          for(i=1; i<=n; i+=4)
               for(j=1; j<=n; j+=[n/4])                              |
                    for(k=1; k<=n; k*=2)
                            x++;
}
```

(b)

```
  int eficiencia2 (bool existe)
   {
      int sum2=0; int k,j,n;

      if (existe)
            for(k=1; k<=n; k*=2)
                 for(j=1; j<=k; j++)
                         sum2++;
      else
             for(k=1; k<=n; k*=2)
               for(j=1; j<=n; j++)
                         sum2++;
    return sum2;
   }
```

(c)

```
 void eficiencia3 (int n)              |    void eficiencia4 (int n)
{                                      |      {
  int j; int i=1; int x=0;             |      int j; int i=2; int x=0;
   do{                                 |       do{
      j=1;                             |           j=1;
      while (j <= n){                  |           while (j <= i){
         j=j*2;                        |               j=j*2;
         x++;                          |               x++;
      }                                |           }
      i++;                             |           i++;
    }while (i<=n);                     |         }while (i<=n);
}                                      |       }
```

2.- Considerar el siguiente segmento de código con el que se pretende buscar un entero ***x*** *en una lista de enteros L de tamaño n (el bucle* ***for*** *se ejecuta* ***n veces****):*

```
    void eliminar (Lista L, int x)
       {
          int aux, p;
          for (p=primero(L); p!=fin(L);)
              {
                 aux=elemento (p,L);
                 if (aux==x)
                    borrar (p,L);
                 else p++;
              }
       }
```

Analizar la eficiencia de la función **eliminar** si:

(a) **primero es O(1)** y **fin, elemento y borrar son O(n)**. ¿Cómo mejorarías esa eficiencia con un ligero cambio en el código?

(b) **primero, elemento y borrar son O(1)** y **fin es O(n)**. ¿Cómo mejorarías esa eficiencia con un ligero cambio en el código?

(c) **todas las funciones son O(1)**. ¿Puede en ese caso mejorarse la eficiencia con un ligero cambio en el código?

Consideraciones:

*1.- El reto es **individual***

2.- la solución deberá entregarse obligatoriamente en un fichero pdf (se sugiere como nombre reto1.pdf)

3.- Si la solución es correcta, se puntuará con 0.2 para la evaluación contínua

4.- El plazo límite de entrega es el 3 de Octubre a las 23.55h

(a)

```
void eficiencia1(int n)
{
  int x=0; int i,j,k;
          for(i=1; i<=n; i+=4)
               for(j=1; j<=n; j+=[n/4])
                    for(k=1; k<=n; k*=2)
                         x++;
}
```

$\{ O(I)$ → Las asignaciones y declaraciones de variables son de orden $O(I)$

$\{ O(I)$

$\{ O(\log_2 n)$ $\}$ $O(\log_2 n)$ $\}$ $O(n \cdot \log_2 n)$ $\}$ $O(n \cdot \log_2 n)$

En este código hay 3 bucles anidados. En el más interno, x++ es lo mismo que x=x+1, lo cual tiene un coste de orden O(I). Como en cada iteración k*=2 y k=1, terminará cuando $2^i > n \Leftrightarrow \log_2 n$ iteraciones, de ahí que su orden de eficiencia sea $O(\log_2 n)$. El bucle inmediatamente externo a este realiza como mucho 4 iteraciones ya que j+=n/4 y $\frac{n}{4} \cdot 4 = n$, y el bucle se ejecuta siempre que $j \leq n$. Finalmente, el bucle mayor tiene una eficiencia de $n/4$ pues se ejecuta mientras que $i \leq n$ y i+=4 en cada iteración. Pero sabemos que $O\left(\frac{n}{4}\right) = O(n)$, y aplicando la regla del producto por tener bucles anidados, la eficiencia del código es $O(n \cdot \log_2 n)$.

(b)

```
int eficiencia2 (bool existe)
 {
    int sum2=0; int k,j,n;                          } O(I)
    if (existe)                                     } O(I)
          for(k=1; k<=n; k*=2)
               for(j=1; j<=k; j++)                  } O(k)
                     sum2++;                        } O(I)
    else
           for(k=1; k<=n; k*=2)
             for(j=1; j<=n; j++)                    } O(n)
                     sum2++;                        } O(I)
  return sum2;
 }
```

Asignación y declaración de variables tiene un coste de O(I)

La comprobación de un if con un dato bool es O(I)

$O(k)$ y $O(I)$ → $O(k \cdot \log_2 n)$

$O(n)$ y $O(I)$ → $O(n \cdot \log_2 n)$

$O(n \cdot \log_2 n)$

Análisis if

Tenemos dos bucles, con uno anidado en el otro. En el más interno, sum2++ es sum2 = sum2 + 1 (operación simple) por lo que es O(I). Viendo la declaración del for, está claro que el bucle se ejecuta k veces → O(k). En el bucle grande, como k*=2 en cada iteración y se ejecuta siempre que k<=n, parará cuando $2^i > n$ → $\log_2 n$ iteraciones ($O(\log_2 n)$). Finalmente se aplica la regla del producto y la eficiencia del if es $O(k \cdot \log_2 n)$.

Análisis else

La estructura es análoga a la anterior y se obtiene $O(n \cdot \log_2 n)$

Conclusión

Entre $O(k \cdot \log_2 n)$ y $O(n \cdot \log_2 n)$, como $k \leq n$, sabemos que la eficiencia de este código es $O(n \cdot \log_2 n)$ ya que siempre debemos quedarnos con el peor de los casos.

(c)

```
void eficiencia3 (int n)
{
  int j; int i=1; int x=0;
   do{
      j=1;
      while (j <= n){
         j=j*2;
         x++;
      }
      i++;
    }while (i<=n);
}
```

O(I) · O(I) · O(log₂n) · O(I) · O(I) · O(I) → O(n · log₂n)

```
void eficiencia4 (int n)
{
int j; int i=2; int x=0;
 do{
     j=1;
     while (j <= i){
         j=j*2;
         x++;
     }
     i++;
  }while (i<=n);
}
```

O(I) · O(I) · O(log₂i) · O(I) · O(I) · O(I) → $\sum_{i=2}^{n} \log_2 i$

$$\sum_{i=2}^{n} \log_2 i = \log_2(n!) \Rightarrow O(\log_2 n)$$

Como $O(\log_2 n) < O(n \cdot \log_2 n)$, es más eficiente eficiencia 4.

Empecemos analizando el bucle while. x++ es una operación simple, por lo que O(I). Lo mismo ocurre con j=j*2. Como la condición del bucle es j≤n y j=j*2, el bucle parará cuando $2^i > n \Rightarrow \log_2 n$ iteraciones. Por lo tanto, la eficiencia del bucle while es O(log₂n). Fuera del bucle, j=1 e i++ son operaciones básicas, con eficiencia O(I). Finalmente, se aplica la regla del producto por tener bucles anidados, siendo la eficiencia total O(n·log₂n)

Al igual que antes se empieza con el bucle while. j=j*2 y x++ son operaciones básicas de eficiencia O(I). Como la condición del bucle es j≤i y j=j*2, mismo razonamiento que antes, la eficiencia del bucle es O(log₂i). Fuera de este bucle, j=1 e i++ son también operaciones básicas de eficiencia O(I). Como el bucle do-while va desde i=1 hasta i=n y el bucle while tiene eficiencia O(log₂i), la eficiencia total será de orden $O(\sum_{i=2}^{n} \log_2 i)$. Se efectúa el cálculo y O(log₂n).

a)

```
void eliminar (Lista L, int x)
   {
      int aux, p;                                   } O(I)
      for (p=primero(L); p!=fin(L);)                O(I)  O(n)
          {
             aux=elemento (p,L);                    } O(n)   } O(n²)   } O(n³)
             if (aux==x)                            } O(I)
                borrar (p,L);                       } O(n)
             else p++;                              } O(I)
          }
   }
```

Se ha aplicado la regla del producto

Las declaraciones de variables, la condición del if (aux==x) y el aumento p++ son operaciones básicas por lo que su eficiencia es O(I). Hemos de hacer una apreciación con la condición del bucle for. Esta se comprueba en cada iteración, así que es como una sentencia más del cuerpo del bucle. Como elemento, fin y borrar son O(n) y el bucle se ejecuta n veces, la eficiencia total del código es $O(n^3)$ (tras aplicar la regla del producto)

Mejora ⇒ Para no tener que llamar a "fin" en cada iteración, que tiene un coste O(n), podemos guardar justo antes de la declaración el valor de fin(L):

```
int aux, p, fin;
fin = fin(L)
for (p = primero(L), p != fin)
```

Aunque seguiría siendo $O(n^3)$, es algo más eficiente dentro de la familia $O(n^3)$

b)

```
void eliminar (Lista L, int x)
   {
      int aux, p;                                   } O(I)
      for (p=primero(L); p!=fin(L);)                O(I)  O(n)
          {
             aux=elemento (p,L);                    } O(I)   } O(n²)   } O(n³)
             if (aux==x)                            } O(I)
                borrar (p,L);                       } O(I)
             else p++;                              } O(I)
          }
   }
```

Lo único que cambia con respecto al caso anterior es que la única función con eficiencia O(n) es "fin". Al igual que antes, la condición del bucle se comprueba en cada iteración, y al tener "fin" en la condición y al ejecutarse n veces el bucle, la eficiencia total es $O(n^3)$.

Mejora ⇒ Haríamos exactamente el mismo cambio que antes. En este caso, como todas las sentencias del bucle son O(I), por lo que la eficiencia del bucle es O(n) y la nueva sentencia fin = fin(L) está fuera del bucle y también es O(n), y por la regla de la suma, la eficiencia final sería O(n).

c)

```
void eliminar (Lista L, int x)
   {
      int aux, p;
      for (p=primero(L); p!=fin(L);)
          {
             aux=elemento (p,L);
             if (aux==x)
                borrar (p,L);
             else p++;
          }
   }
```

O(I)
O(I)
O(I)
O(I)
O(I)
O(I) } O(n)
O(n)

En este código todas las operaciones son O(I) y por lo tanto, como el bucle se ejecuta n veces, la eficiencia total del código es O(n).

Mejora ⇒ Lo único que podemos hacer es no declarar "aux" y trabajar con elemento(p,L), evitando hacer la asignación aux = elemento(p,L). La eficiencia seguiría siendo O(n) aunque muy ligeramente mejor.