# Estructuras de Datos y de la Información

Ingeniería Técnica en Informática de Gestión. Curso 2009/2010

Ejercicios del Tema 1

## Análisis de algoritmos iterativos

1. Determina la complejidad temporal en el caso peor y en el caso mejor de los siguientes algoritmos de ordenación.

(a) Ordenación por inserción

```
void ordenaIns ( int v[], int num ) {
  int i, j, x;

  for ( i = 1; i < num; i++ ) {
    x = v[i];
    j = i-1;
    while ( (j >= 0 ) && (v[j] > x) ){
      v[j+1] = v[j];
      j = j-1;
    }
    v[j+1] = x;
  }
}
```

(b) Ordenación por selección

```
void ordenaSel ( int v[], int num ) {
  int i, j, menor, aux;

  for ( i = 0; i < num; i++ ) {
    menor = i;
    for ( j = i+1; j < num; j++ )
      if ( v[j] < v[menor] )
        menor = j;
    if ( i != menor ) {
      aux = v[i];
      v[i] = v[menor];
      v[menor] = aux;
    }
  }
}
```

(c) Método de la burbuja

```
void ordenaBur ( int v[], int num ) {
  int i, j, aux;
  bool modificado;

  i = 0;
  modificado = true;
  while ( (i < num-1) && modificado ) {
    modificado = false;
    for ( j = num-1; j > i; j-- )
      if ( v[j] < v[j-1] ) {
        aux = v[j];
        v[j] = v[j-1];
        v[j-1] = aux;
        modificado = true;
      }
    i++;
  }
}
```

2. Determina la complejidad temporal en el caso peor del siguiente algoritmo.

```
int buscaBin( int v[], int num, int x ) {
  int izq, der, centro;

  izq = -1;
  der = num;
  while ( der != izq+1 ) {
    centro = (izq+der) / 2;
    if ( v[centro] <= x )
      izq = centro;
    else
      der = centro;
  }
  return izq;
}
```

3. El siguiente algoritmo de búsqueda decide si un entero dado aparece o no dentro de un vector de enteros dado:

```
bool busca( int v[], int num, int x ) {
// Pre.: v es un array de al menos num elementos
  int j;
  bool encontrado;

  j = 0;
  encontrado = false;
  while ( (j < num) && ! encontrado ) {
    encontrado = ( v[j] == x );
    j++;
  }
  return encontrado;
// Post.: devuelve true si x aparece en v entre las posiciones 0 .. num-1
//        y false en caso contrario
}
```

Analiza la *complejidad en promedio* del algoritmo, bajo las hipótesis siguientes: (a) la probabilidad de que $x$ aparezca en $v$ es un valor constante $p$, $0 < p < 1$; y (b) la probabilidad de que $x$ aparezca en la posición $i$ de $v$ (y no en posiciones anteriores) es la misma para cada índice $i$, $0 \le i < num$.

4. Las dos implementaciones siguientes calculan la potencia $n^m$:

```
int potencia1(int n, int m)
{
   int p;
   p=1;
   while (m>0)
   {
      p=p*n; m--;
   }
   return p;
}
```

```
int potencia2(int n, int m)
{
   int p;
   p=1;
   while (m>0)
   {
      if (m%2!=0) p=p*n;
      m=m/2;
      n=n*n;
   }
   return p;
}
```

Describir su funcionamiento y determinar su complejidad en el peor caso, ¿cuál es mejor?.

5. (Febrero 2003) Describir brevemente lo que hace el siguiente fragmento de código y determinar su complejidad asintótica en función de n:

```
for (i=1; i<=n; i++){
   for (j=1; j<=n-i; j++) cout << " ";
   for (j=1; j<=i; j++) cout << j;
   for (j=i-1; j>=1; j--) cout << j;
   cout << "\n";
}
```

6. Los algoritmos de ordenación de arrays han sido objeto de estudio meticuloso en el área de la informática, así como la complejidad asociada a los mismos. Los que hemos estudiado nosotros hasta el momento son generales en el sentido de que, en principio, sirven para ordenar cualquier array de enteros (es muy fácil modificarlos para ordenar arrays de cualquier tipo ordenado).

   Sin embargo, para algunos problemas concretos es posible sacar partido de las particularidades del problema. Por ejemplo, si sabemos que el array a ordenar v[1..n] contiene enteros en un rango acotado (y relativamente pequeño), $n_0 \leq v[i] \leq n_1$ para todo $i \in \{1, \ldots, n\}$, podemos utilizar el siguiente algoritmo:

   - construir la tabla de frecuencias t asociada a v. Esta tabla no es más que un array $t[n_0..n_1]$ tal que t[i] contiene el número de apariciones del elemento i en el array inicial v

   - a partir de la tabla de frecuencias, obtener la ordenación del vector v. La idea es: colocar al principio de v $t[n_0]$ elementos $n_0$, a continuación colocar $t[n_0 +1]$ elementos $n_1$ y así sucesivamente.

   Por ejemplo, consideremos v = [1, 3, 2, 4, 5, 3, 4, 2, 3, 1, 4, 5]

   Todos los elementos están en el rango [1.,5]. La tabla de frecuencias asociada es:

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| t | 2 | 2 | 3 | 3 | 2 |

   Ahora, colocamos en v en este orden 2 unos, 2 doses, 3 treses, 3 cuatros y 2 cincos, y obtenemos el array ordenado:

   v = [1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5]

   Implementar este algoritmo en C++ para vectores de enteros en el rango 0.,999. ¿Cuál es la complejidad del algoritmo en el peor caso? ¿Y en el mejor? Ampliar la funcionalidad de modo que el rango de los elementos se determine en tiempo de ejecución y el algoritmo opere de acuerdo con este dato. ¿Cuál es la complejidad de este nuevo algoritmo?

7. (Junio 2003) Hoy es día de excursión para todo el cole y los niños están muy contentos. Como son muchísimos niños, para evitar despistes y no perder a ninguno, la señorita encargada los ha colocado por orden alfabético. Sin embargo, en un despiste de la "seño", algunos niños han hecho de las suyas y han intercambiado su posición con la de uno de los compañeros de al lado (el de la izquierda o el de la derecha). Los niños más revoltosos han repetido este proceso varias veces, mientras que hay otros que no se han movido ni una sola vez, como Remedios que es muy buena. La señorita se enfada mucho cuando descubre el desaguisado: *"He malgastado n*log(n) (unidades de tiempo, siendo n el número de niños) en ordenaros... ¡Os pondré un negativo a todos!"*. Pero Remedios, que tiene un expediente inmaculado, protesta: *"Seño, yo no me he movido ninguna vez. Jaimito es el que más veces se ha movido. Yo lo he visto... y se ha movido 4 veces"*.

   Casualmente un alumno de EDI que ve lo que ha ocurrido y tranquiliza a la señorita diciéndole que la entropía generada por los niños no es excesiva y que de hecho puede diseñar un algoritmo de complejidad lineal para recomponer la ordenación. Asumiendo que los niños están representados en un vector diseña e implementa un algoritmo que reconstruya la ordenación en tiempo lineal. Por simplicidad, puedes suponer que cada niño está representado por un entero con respecto al que se ordena. Razona que el algoritmo construido es de complejidad lineal.

## Órdenes de Complejidad

8. Demostrar:
   a) $f(n) \in O(g(n)) \Leftrightarrow O(f(n)) \subseteq O(g(n))$
   b) $O(f(n)) = O(g(n)) \Leftrightarrow f(n) \in O(g(n))$ y $g(n) \in O(f(n))$
   d) $O(c \cdot f(n)) = O(f(n))$, siendo $c > 0$.

9. Demostrar la regla de la suma para el orden exacto:
   $\Theta(f(n) + g(n)) = \Theta(\max(f(n), g(n)))$

10. Demostrar las siguientes inclusiones estrictas:
    $O(1) \subset O(\log(n)) \subset O(n) \subset O(n \not\subset \log(n)) \subset O(n^2) \subset O(n^3) \subset \ldots \subset O(n^k) \subset O(2^n) \subset O(n!)$

11. Demostrar la propiedad de la dualidad:
    $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

12. Demostrar o refutar las siguientes propiedades:
    a) $f(n) \in O(n^2)$ y $g(n) \in O(n) \Rightarrow f(n)/g(n) \in O(n)$
    b) $f(n) \in \Theta(n^2)$ y $g(n) \in \Theta(n) \Rightarrow f(n)/g(n) \in \Theta(n)$

13. Demostrar o refutar las siguientes afirmaciones:
    a) $2^n + n^{99} \in O(n^{99})$
    b) $2^n + n^{99} \in \Omega(n^{99})$
    c) $2^n + n^{99} \in \Theta(n^{99})$
    d) $2^n + n^{99} \in O(2^n)$
    e) $2^n + n^{99} \in \Omega(2^n)$
    f) $2^n + n^{99} \in \Theta(2^n)$

14. Buscar ejemplos de funciones $f(n)$ y $g(n)$ tales que $f(n) \in O(g(n))$ pero $f(n) \notin \Omega(g(n))$