Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos – Grado en Ingeniería Informática

Curso 2018-2019 – 14 de mayo de 2019

Duración: **2 horas y 45 minutos**

Soluciones

1. Análisis de algoritmos

Ejercicio 1 [2 puntos]

Demostrar **mediante la definición** de complejidad asintótica $\mathcal{O}$, si se verifican:

a) ¿ $2^{n+1} \in \mathcal{O}(2^n)$? **[1 punto]**

b) ¿ $2^{2n} \in \mathcal{O}(2^n)$? **[1 punto]**

Solución:

En primer lugar, $2^{n+1} \in \mathcal{O}(2^n)$ es cierto. Aplicando la definición debemos buscar una constante $c > 0$ y otra $n_0 > 0$ tal que:

$$2^{n+1} \leq c \cdot 2^n$$

en un intervalo $[n_0, \infty)$. Como $2^{n+1} = 2 \cdot 2^n$, tenemos que encontrar una c que cumpla:

$$2 \cdot 2^n \leq c \cdot 2^n$$

Escogiendo $c = 2$, la desigualdad anterior naturalmente se cumple para todo n. Por tanto, se cumple en un intervalo $[n_0, \infty)$ para cualquier valor de de n_0. Como al aplicar la definición $n_0 > 0$, podemos escoger $n_0 = 1$. De esta manera, hemos hallado una pareja de constantes que hacen que $2^{n+1} \in \mathcal{O}(2^n)$ sea cierto.

En segundo lugar, $2^{2n} \in \mathcal{O}(2^n)$ no es cierto. Por eso, en vez de intentar encontrar las constantes c y n_0, tenemos que demostrar que no va a ser posible hallarlas. Inicialmente, al intentar aplicar la definición deberíamos encontrar una constante $c > 0$ y otra $n_0 > 0$ tal que:

$$2^{2n} \leq c \cdot 2^n$$

en un intervalo $[n_0, \infty)$. Aplicamos logaritmos (en base 2) a ambos lados:

$$\log_2(2^{2n}) \leq \log_2 c \cdot 2^n$$

$$2n \leq \log_2 c + n$$

$$n \leq \log_2 c$$

La última expresión indica que la desigualdad se cumple para todo n menor o igual que la constante $\log_2 c$. Por tanto, sea cual sea el valor de c, la desigualdad se cumple en el intervalo $(-\infty, \log_2 c]$, pero nunca en un intervalo $[n_0, \infty)$. Por tanto, es imposible hallar la pareja de constantes c y n_0 que necesitamos para verificar $2^{2n} \in \mathcal{O}(2^n)$.

Ejercicio 2 [2 puntos]

Calcula el número de operaciones ($T(n)$) que realiza el siguiente código:

```
1  int i=0;
2  while (i<=n){
3      int j=i;
4      while (j<=n){
5          procesa(i,j);   // dos operaciones
6          j++;
7      }
8      i++;
9  }
```

Se considera que las inicializaciones, comparaciones, e incrementos siempre necesitan una sola operación.

Solución:

El código consta de 2 bucles, que podemos descomponer de la siguiente manera:

$$T(n) = 1 + \sum_{i=0}^{n}(1 + T_{\text{For 2}} + 1) + 1$$

$$T_{\text{For 2}} = 1 + \sum_{j=i}^{n}(1 + 2 + 1) + 1 = 2 + 4(n - i + 1) = 4n + 6 - 4i$$

Sustituyendo $T_{\text{For 2}}$ en $T(n)$ obtenemos:

$$T(n) = 2 + \sum_{i=0}^{n}(2 + 4n + 6 - 4i) = 2 + \sum_{i=0}^{n}(4n + 8 - 4i)$$

$$= 2 + 4n(n+1) + 8(n+1) - 4\sum_{i=0}^{n} i$$

$$= 2 + 4n^2 + 4n + 8n + 8 - 4\frac{n(n+1)}{2}$$

$$= 2n^2 + 10n + 10 \in \Theta(n^2)$$

Ejercicio 3 [3 puntos]

Resolved la siguiente relación de recurrencia por el **método general de resolución de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + log_{10}(n^2) & \text{si } n > 1. \end{cases}$$

Solución:

En primer lugar podemos expresar $log_{10}(n^2)$ como $2\log_{10}(n)$. Luego, para poder aplicar el método, tenemos que hacer el cambio de variable $n = 10^k$. Además, en ese caso $k = \log_{10}(n)$. Por tanto, podemos reescribir la expresión para $T(n)$ de la siguiente manera:

$$T(n) = T(10^k) = T(10^k/10) + 2k = T(10^{k-1}) + 2k$$

A continuación hacemos un cambio de función $t(k) = T(10^k)$:

$$T(n) = T(10^k) = t(k) = t(k-1) + 2k \cdot 1^k$$

El polinomio característico de la recurrencia $t(k) = t(k-1) + 2k \cdot 1^k$ es:

$$(x-1)^3$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 1^k + C_2 k \cdot 1^k + C_2 k^2 \cdot 1^k = C_1 + C_2 k + C_3 k^2$$

Para hallar las constantes necesitamos tres casos base. En primer lugar conocemos $T(1) = t(0) = 1$. Aplicando $t(k) = t(k-1) + 2k$ vemos que:

$$t(1) = t(0) + 2 \cdot 1 = 1 + 2 = 3$$

$$t(2) = t(1) + 2 \cdot 2 = 3 + 4 = 7$$

Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcrcrcl} t(0) & = & C_1 & + & & & & = & 1 \\ t(1) & = & C_1 & + & C_2 & + & C_3 & = & 3 \\ t(2) & = & C_1 & + & 2C_2 & + & 4C_3 & = & 7 \end{array}\right\}$$

Las soluciones son $C_1 = 1$, $C_2 = 1$ y $C_3 = 1$. Por tanto:

$$t(k) = 1 + k + k^2$$

Finalmente, deshaciendo el cambio de variable y función obtenemos:

$$T(n) = 1 + \log_{10}(n) + [\log_{10}(n)]^2 \quad \in \quad \Theta[(\log n)^2]$$

Ejercicio 4 [3 puntos]

Resolved la siguiente relación de recurrencia por el **método de expansión de recurrencias**:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + log_{10}(n^2) & \text{si } n > 1. \end{cases}$$

Solución:

En primer lugar podemos expresar $log_{10}(n^2)$ como $2\log_{10}(n)$, quedando:

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/10) + 2\log_{10}(n) & \text{si } n > 1. \end{cases}$$

Procedemos a expandir la recurrencia:

$$\begin{aligned}
T(n) &= T(n/10) + 2\log_{10} n \\
&= \left[T(n/10^2) + 2\log_{10}(n/10)\right] + 2\log_{10} n \\
&= T(n/10^2) + 2\log_{10} n - 2\log_{10} 10 + 2\log_{10} n \\
&= \left[T(n/10^3) + 2\log_{10}(n/100)\right] + 2\log_{10} n - 2\log_{10} 10 + 2\log_{10} n \\
&= T(n/10^3) + 2\log_{10} n - 2\log_{10} 100 + 2\log_{10} n - 2\log_{10} 10 + 2\log_{10} n \\
&= T(n/10^3) + 3 \cdot 2\log_{10} n - 2(1+2) \\
&\vdots \\
&= T(n/10^4) + 4 \cdot 2\log_{10} n - 2(1+2+3) \\
&\vdots \\
&= T(n/10^i) + 2i\log_{10} n - 2\sum_{j=1}^{i-1} j \\
&= T(n/10^i) + 2i\log_{10} n - (i-1)i
\end{aligned}$$

Se llega al caso base cuando $n/10^i = 1$. Es decir, cuando $n = 10^i$, o de manera equivalente, cuando $i = \log_{10} n$. Sustituyendo:

$$T(n) = T(1) + 2(\log_{10} n)(\log_{10} n) - (\log_{10} n - 1)(\log_{10} n)$$

$$T(n) = 1 + \log_{10}(n) + [\log_{10}(n)]^2 \quad \in \quad \Theta[(\log n)^2]$$

2. Diseño de algoritmos

Ejercicio 5 [5 puntos]

Se pide implementar un algoritmo basado en la estrategia de divide y vencerás para determinar si una matriz **A** de números enteros está "ordenada" por filas y columnas independientemente. Para ello, los elementos de cada fila y cada columna deben aparecer en orden no-decreciente. La matriz **A** no es necesariamente cuadrada. La siguiente matriz estaría ordenada según el criterio descrito:

$$\mathbf{A} = \begin{pmatrix} 1 & 3 & 6 & 8 & 10 \\ 2 & 4 & 7 & 9 & 10 \\ 4 & 8 & 11 & 14 & 14 \\ 5 & 10 & 12 & 15 & 16 \end{pmatrix}$$

Solución:

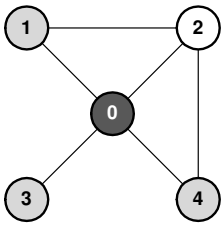
Una posible solución consiste en dividir la matriz en cuatro submatrices (partimos la matriz por la fila central y por la columna central). En el caso recursivo cada una de las submatrices debe estar "ordenada" según especifica el enunciado. Para ello se realizarían cuatro llamadas recursivas. Además, los elementos de las dos columnas centrales, y de las dos filas centrales deben estar ordenados de menor a mayor. Esto se puede ver empleando un bucle en cada caso. El siguiente código ilustra una posible implementación de la función.

```
def esta_ordenada_matriz(A):
    (n,m) = A.shape

    if n==1 and m==1:
        return True
    elif n==0 or m==0:
        return True
    else:

        mitad_n = n//2
        mitad_m = m//2

        elementos_centrales_ordenados = True;
        i = 0
        while i<n and elementos_centrales_ordenados:
            elementos_centrales_ordenados = A[i,mitad_m-1]<=A[i,mitad_m]
            i = i+1

        j = 0
        while j<m and elementos_centrales_ordenados:
            elementos_centrales_ordenados = A[mitad_n-1,j]<=A[mitad_n,j]
            j = j+1

        return (elementos_centrales_ordenados and
                esta_ordenada_matriz(A[0:mitad_n,0:mitad_m]) and
                esta_ordenada_matriz(A[0:mitad_n,mitad_m:m]) and
                esta_ordenada_matriz(A[mitad_n:n,0:mitad_m]) and
                esta_ordenada_matriz(A[mitad_n:n,mitad_m:m]))
```

Ejercicio 6 [5 puntos]

Sea un grafo G no dirigido, y no ponderado, compuesto por n vértices. El grafo G estará definido mediante una matriz de adyacencia simétrica y cuadrada $\mathbf{A}$ de tamaño $n \times n$, cuyos elementos serán 0 o 1. Si el elemento $a_{i,j} = 0$, no habrá una arista conectando a los vértices i y j, mientras que si $a_{i,j} = 1$ sí que habrá una arista conectándolos y serán adyacentes. Se asumirá que no habrá aristas desde un vértice a sí mismo (es decir, $a_{i,i} = 0$ para todo i).

Dado un número entero $m \leq n$, se pide implementar un algoritmo basado en la técnica de *backtracking* que determine si es posible colorear los vértices del grafo con m colores diferentes, de manera que no se asigne el mismo color a dos vértices adyacentes. El siguiente grafo se puede colorear usando 3 colores, pero no es posible colorearlo usando menos colores:



$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

El algoritmo podrá usar funciones auxiliares para verificar la validez de los candidatos a introducir en la solución parcial.

Solución:

En este ejercicio la solución parcial será una lista de n enteros, correspondientes a los n vértices del grafo. Los elementos de la solución parcial podrán tomar m valores diferentes (por ejemplo, desde 0 hasta $m-1$), donde cada uno representará un color diferente. Por tanto, el algoritmo de *backtracking* tendrá un bucle para generar los m posibles candidatos (colores). El bucle será de tipo While, para para en cuanto se haya encontrado una coloración válida. A continuación, se analiza si el color para el vértice i-ésimo es válido. Para ello se puede llamar a una función que compruebe si el color que pretendemos usar para el vértice i-ésimo ya aparece en uno de sus vértices adyacentes. Si el candidato es válido se actualiza la solución parcial y se llama a la función recursiva para expandir la solución parcial a partir del vértice $i+1$. El caso recursivo termina devolviendo si ha encontrado una coloración válida. Por último, el caso base del algoritmo simplemente comprueba si la solución parcial se ha completado, en cuyo caso la coloración sería válida, y la función puede devolver `True`. El siguiente código ilustra una posible implementación.

```python
import numpy as np

def es_candidato_valido(k,i,sol,A):

    es_valido = True

    j=0
    while j<i and es_valido:
        if A[j,i]==1 and sol[j]==k:
            es_valido = False
        j = j + 1

    return es_valido


def colorear_backtracking(i,m,sol,A):
    n = A.shape[0]

    if i==n:
        print(sol)  # opcional
        return True
    else:

        exito = False
        k = 0
        while k<m and not exito:

            if es_candidato_valido(k,i,sol,A):

                sol[i] = k

                exito = colorear_backtracking(i+1,m,sol,A)

            k = k + 1

    return exito


def puede_colorear(A,m):
    n = A.shape[0]

    sol = np.empty(n)

    return colorear_backtracking(0,m,sol,A)
```