Universidad Rey Juan Carlos

Diseño y Análisis de Algoritmos – Grado en Ingeniería Informática

Curso 2018-2019 – 10 de mayo de 2019

Duración: **2 horas y 45 minutos**

Soluciones

1. Análisis de algoritmos

Ejercicio 1 [1 punto]

Ordena de menor a mayor orden de complejidad $\mathcal{O}$ las siguientes funciones (algunas pueden tener la misma complejidad):

$$10n, \quad n\log_{10} n, \quad n^2\log n, \quad 5\log n, \quad \log n^2, \quad n^2, \quad n^{1+a} \text{ con } 0<a<1$$

Solución:

$$\log n^2 \;=\; 5\log n \;<\; 10n \;<\; n\log_{10} n \;<\; n^{1+a}\ (\text{con } 0<a<1) \;<\; n^2 \;<\; n^2\log n$$

Ejercicio 2 [3 puntos]

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

```
1  for (int i=0; i<n; i++)
2    for (int j=i+1; j<=n; j++)
3      if (condicion(i))    // una operación
4        for (int k=0; k<j; k++)
5          procesa(i,j,k);  // dos operaciones
```

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

Solución:

El código consta de 3 bucles, que podemos descomponer de la siguiente manera:

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

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

$$T_{\text{If}} = \begin{cases} 1 & \text{en el mejor caso} \\ 1 + 1 + \sum_{k=0}^{j-1}(1+2+1) + 1 = 3 + 4j & \text{en el peor caso} \end{cases}$$

Veamos primero el coste en el mejor caso. Si $T_{\text{If}} = 1$, entonces sustituyendo en $T_{\text{For 2}}$ tenemos:

$$T_{\text{For 2}} = 2 + \sum_{j=i+1}^{n} 3 = 2 + 3(n-i) = 3n + 2 - 3i.$$

Sustituyendo en $T(n)$ obtenemos:

$$T(n) = 2 + \sum_{i=0}^{n-1}(3n + 4 - 3i) = 2 + 3n^2 + 4n - 3\sum_{i=0}^{n-1} i$$

$$= 2 + 3n^2 + 4n - 3\frac{(n-1)n}{2} = \frac{4 + 6n^2 + 8n - 3n^2 + 3n}{2} = \frac{3n^2 + 11n + 4}{2} \in \Theta(n^2)$$

En el peor caso, sustituyendo T_{If} en $T_{\text{For 2}}$ tenemos:

$$T_{\text{For 2}} = 2 + \sum_{j=i+1}^{n} (5 + 4j) = 2 + 5(n-i) + 4\sum_{j=i+1}^{n} j$$

$$= 2 + 5n - 5i + 4\left[\sum_{j=1}^{n} j - \sum_{j=1}^{i} j\right]$$

$$= 2 + 5n - 5i + 4\left[\frac{n(n+1)}{2} - \frac{i(i+1)}{2}\right]$$

Simplificando obtenemos:

$$T_{\text{For 2}} = 2 + 5n - 5i + 2n^2 + 2n - 2i^2 - 2i = 2n^2 + 7n + 2 - 7i - 2i^2$$

Sustituyendo en la expresión para el primer bucle tenemos:

$$T(n) = 2 + \sum_{i=0}^{n-1}(2n^2 + 7n + 4 - 7i - 2i^2) = 2 + 2n^3 + 7n^2 + 4n - 7\sum_{i=0}^{n-1} i - 2\sum_{i=0}^{n-1} i^2$$

$$= 2 + 2n^3 + 7n^2 + 4n - 7\frac{(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6}$$

$$= 2 + 2n^3 + 7n^2 + 4n - 7\frac{n^2}{2} + 7\frac{n}{2} - 2\frac{2n^3 - 3n^2 + n}{6}$$

$$= \frac{12 + 12n^3 + 42n^2 + 24n - 21n^2 + 21n - 4n^3 + 6n^2 - 2n}{6}$$

Finalmente:

$$T(n) = \frac{8n^3 + 27n^2 + 43n + 12}{6} \in \theta(n^3)$$

Ejercicio 3 [3 puntos]

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

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/4) + n^2 & \text{si } n > 1. \end{cases}$$

Solución:

Para poder aplicar el método primero tenemos que hacer el cambio de variable $n = 4^k$. Además, en ese caso:

$$n^2 = (4^k)^2 = 4^{2k} = (4^2)^k = 16^k.$$

Podemos reescribir la expresión para $T(n)$ de la siguiente manera:

$$T(n) = T(4^k) = T(4^k/4) + 16^k = T(4^{k-1}) + 16^k$$

A continuación hacemos un cambio de función $t(k) = T(4^k)$:

$$T(n) = T(4^k) = t(k) = t(k-1) + 16^k$$

El polinomio característico de la recurrencia $t(k) = t(k-1) + 16^k$ es:

$$(x-1)(x-16)$$

Por tanto, la recurrencia tiene la siguiente forma:

$$t(k) = C_1 1^k + C_2 16^k = C_1 + C_2 16^k$$

En este momento podemos deshacer el cambio de variable fácilmente:

$$T(n) = C_1 + C_2 n^2$$

Para hallar las constantes necesitamos dos casos base. Procedemos a calcular $T(4) = T(1) + 4^2 = 17$. Por tanto, el sistema de ecuaciones a resolver es:

$$\left.\begin{array}{rcrcr} T(1) & = & C_1 + C_2 & = & 1 \\ T(4) & = & C_1 + 16C_2 & = & 17 \end{array}\right\}$$

Las soluciones son $C_1 = -1/15$ y $C_2 = 16/15$. Por tanto:

$$T(n) = \frac{16}{15}n^2 - \frac{1}{15} \in \Theta(n^2)$$

Ejercicio 4 [3 puntos]

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

$$T(n) = \begin{cases} 1 & \text{si } n = 1, \\ T(n/4) + n^2 & \text{si } n > 1. \end{cases}$$

Solución:

Procedemos a expandir la recurrencia:

$$\begin{aligned} T(n) &= T(n/4) + n^2 \\ &= \left[T(n/4^2) + \left(\frac{n}{4}\right)^2\right] + n^2 = T(n/4^2) + n^2\left(1 + \frac{1}{16}\right) \\ &= \left[T(n/4^3) + \left(\frac{n}{4^2}\right)^2\right] + n^2\left(1 + \frac{1}{16}\right) = T(n/4^3) + n^2\left(1 + \frac{1}{16} + \frac{1}{16^2}\right) \\ &\vdots \\ &= T(n/4^i) + n^2\left(1 + \frac{1}{16} + \frac{1}{16^2} + \cdots + \frac{1}{16^{i-1}}\right) = T(n/4^i) + n^2 \sum_{j=0}^{i-1} \frac{1}{16^j} \end{aligned}$$

El último sumatorio es una serie geométrica:

$$\sum_{j=0}^{i-1} \frac{1}{16^j} = \frac{\frac{1}{16^i} - 1}{\frac{1}{16} - 1}.$$

Por tanto, tenemos:

$$T(n) = T(n/4^i) + n^2\left[\frac{\frac{1}{16^i} - 1}{\frac{1}{16} - 1}\right] = T(n/4^i) + n^2\left[\frac{\frac{1}{16^i} - 1}{-\frac{15}{16}}\right] T(n/4^i) + n^2\left[-\frac{16}{15} \cdot \frac{1}{16^i} + \frac{16}{15}\right]$$

Se llega al caso base cuando $n/4^i = 1$. Es decir, cuando $n = 4^i$. En ese caso $n^2 = 16^i$. Sustituyendo:

$$T(n) = T(1) + n^2\left[-\frac{16}{15}\frac{1}{n^2} + \frac{16}{15}\right] = 1 - \frac{16}{15} + \frac{16}{15}n^2 = \frac{16}{15}n^2 - \frac{1}{15} \in \Theta(n^2)$$

2. Diseño de algoritmos

Ejercicio 5 [5 puntos]

Se pide implementar un algoritmo basado en la estrategia de divide y vencerás para resolver el problema de la sublista de máxima suma. Dada una lista de números el objetivo es hallar una sublista de elementos contiguos cuya suma sea máxima. Por ejemplo, dada la lista $[-1, -4, 5, 2, -3, 4, 2, -5]$, la lista óptima es $[5, 2, -3, 4, 2]$, cuyos elementos suman 10. Se asumirá que la lista inicial no es vacía, y está compuesta de números enteros.

Solución:

El algoritmo debe descomponer la lista dada en dos mitades. La sublista de suma máxima estará en alguna de esas mitades (esto se calcula mediante dos llamadas recursivas), o será la concatenación de dos sublistas de ambas mitades, que sean contiguas en la original. Por ejemplo, la sublista de suma máxima de la mitad izquierda contendrá el elemento en la posición (mitad-1), mientras que la sublista de suma máxima de la mitad derecha contendrá el elemento en la posición (mitad). Ambas sublistas máximas se pueden hallar mediante simples bucles, y la suma máxima de su concatenación es la suma sus respectivas sumas máximas. El siguiente código describe el algoritmo:

Java:

```java
import java.io.*;

public class MaxSumList {

    public static int max(int a, int b) {
        if (a>b)
            return a;
        else
            return b;
    }
```

```java
    public static int maxima_suma_sublista(int a[], int ini, int fin) {
        int n = a.length;
        if (ini==fin)
            return a[ini];
        else {
            int mitad = (ini+fin)/2;

            // Calculas la suma máxima de ambas mitades de la lista a
            int max_mitad_izq = maxima_suma_sublista(a,ini,mitad);
            int max_mitad_dcha = maxima_suma_sublista(a,mitad+1,fin);

            // suma máxima de las listas de la mitad izquierda
            // que contienen el índice (mitad-1)
            int max_izq = -Integer.MAX_VALUE;
            int aux_suma = 0;
            for (int i=mitad; i>=0; i--) {
                aux_suma = aux_suma + a[i];
                max_izq = max(max_izq,aux_suma);
            }

            // suma máxima de las listas de la mitad derecha
            // que contienen el índice (mitad)
            int max_dcha = -Integer.MAX_VALUE;
            aux_suma = 0;
            for (int i=mitad+1; i<n; i++) {
                aux_suma = aux_suma + a[i];
                max_dcha = max(max_dcha,aux_suma);
            }

            return max(max(max_mitad_izq, max_mitad_dcha), max_izq+max_dcha);
        }
    }

    // función no pedida en el enunciado:
    public static void main(String args[]) throws Exception {
        //lectura:
        BufferedReader cin =
            new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(cin.readLine());

        int a[] = new int[n];

        //lectura:
        String[] listaEnteros = cin.readLine().split(" ");
        for (int i=0; i<n; i++) {
            int m = Integer.parseInt(listaEnteros[i]);
            a[i] = m;
        }

        //escritura:
        System.out.println(maxima_suma_sublista(a,0,n-1));
    }
}
```

Python:

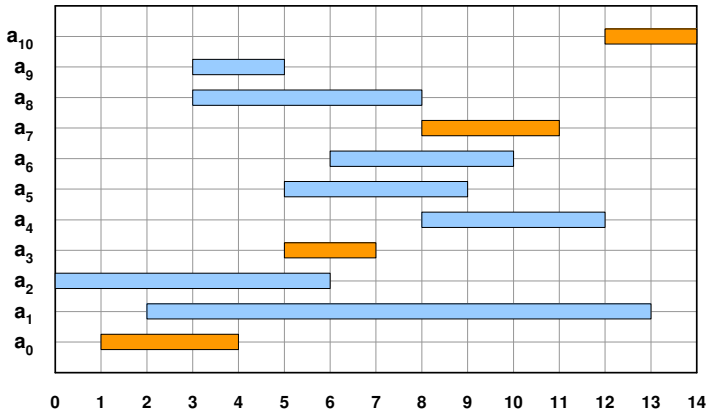
```python
import math

def maxima_suma_sublista(a):
    n = len(a)
    if n==1:
        return a[0]
    else:
        mitad = n//2

        # Calculas la suma máxima de ambas mitades de la lista a
        max_mitad_izq = maxima_suma_sublista(a[0:mitad])
        max_mitad_dcha = maxima_suma_sublista(a[mitad:])

        # suma máxima de las listas de la mitad izquierda
        # que contienen el índice (mitad-1)
        max_izq = -math.inf
        aux_suma = 0
        for i in range(mitad-1,-1,-1):
            aux_suma = aux_suma + a[i]
            max_izq = max(max_izq,aux_suma)

        # suma máxima de las listas de la mitad derecha
        # que contienen el índice (mitad)
        max_dcha = -math.inf
        aux_suma = 0
        for i in range(mitad,n):
            aux_suma = aux_suma + a[i]
            max_dcha = max(max_dcha,aux_suma)

        return max(max_mitad_izq, max_mitad_dcha, max_izq+max_dcha)

# No pedido en el enunciado:
a = [int(x) for x in input().split()]
print(maxima_suma_sublista(a))
```

Ejercicio 6 [5 puntos]

Se pide implementar un algoritmo basado en la técnica de *backtracking* para resolver el problema de selección de actividades. Sea un conjunto A de n actividades $\{a_0, a_1, \ldots, a_{n-1}\}$ que necesitan utilizar un recurso común, por ejemplo, una sala de reuniones. El recurso solo puede ser usado por una actividad en cada momento. Cada actividad tiene un instante de comienzo c_i y un instante de finalización f_i, donde $0 \leq c_i < f_i < \infty$. Si se selecciona la actividad a_i, se desarrolla en el intervalo semiabierto de tiempo $[c_i, f_i)$. Las actividades a_i y a_j son compatibles si sus intervalos $[c_i, f_i)$ y $[c_j, f_j)$ no se solapan, es decir, si $c_i \geq f_j$ o $c_j \geq f_i$.

El objetivo de este ejercicio es determinar un subconjunto de actividades compatibles cuya cardinalidad sea máxima. Por ejemplo, sea el siguiente conjunto de actividades:

i	0	1	2	3	4	5	6	7	8	9	10
c_i	1	2	0	5	8	5	6	8	3	3	12
f_i	4	13	6	7	12	9	10	11	8	5	14



Un subconjunto S de actividades compatibles es $\{a_2, a_4, a_{10}\}$. Sin embargo, no es un subconjunto de cardinalidad máxima, como lo son $\{a_0, a_3, a_7, a_{10}\}$ (en naranja en la figura) y $\{a_9, a_3, a_4, a_{10}\}$ (ya que tienen 4 elementos).

Aunque este problema se puede resolver con un algoritmo voraz, el método desarrollado deberá seguir una estrategia basada en *backtracking*. Se asumirá que las tareas no están ordenadas. Además, el algoritmo desarrollado no ordenará las tareas de ninguna manera. El algoritmo podrá usar funciones auxiliares para verificar la validez de los candidatos a introducir en la solución parcial.

Solución:

La solución consiste en analizar todos los subconjuntos posibles de tareas. Para ello el siguiente código usa el esquema para buscar subconjuntos basado en un árbol binario. Para podar el árbol de recursión se usa una función que comprueba si hay solapamiento entre la tarea i y las contenidas en la solución parcial (desde el índice 0 hasta el $i-1$). En caso de haber analizado todos los posibles candidatos (tareas), se actualiza la mejor solución y su cardinalidad si la cardinalidad del conjunto definido por solución parcial es mayor que la encontrada previamente por el algoritmo de *backtracking*.

Java:

```java
import java.io.*;

public class Tareas {

    public static boolean es_candidato_valido(int i, int sol[],
                                              int c[], int f[]) {

        boolean es_valido = true;

        int j=0;
        while ((j<i) && es_valido) {
            if (sol[j]==1)
                es_valido = (c[i]>=f[j] || c[j]>=f[i]);

            j = j + 1;
        }

        return es_valido;
    }


    public static int busca_maxima_cardinalidad(int i, int card_actual,
                                                int sol[], int opt_sol[],
                                                int card_opt,
                                                int c[], int f[]) {

        int n = sol.length;

        if (i==n) {
            if (card_actual>card_opt) {
                card_opt = card_actual;
                for (int k=0; k<n; k++)
                    opt_sol[k] = sol[k];
            }
        }
        else {
            for (int k=0; k<=1; k++) {

                if ((k==0) || es_candidato_valido(i,sol,c,f)) {

                    sol[i] = k;

                    card_actual = card_actual + k;

                    card_opt = busca_maxima_cardinalidad(i+1,card_actual,
                                                sol,opt_sol,card_opt,c,f);
                }
            }
        }

        return card_opt;
    }
```

```java
    public static int planificacion_tareas(int c[], int f[]) {

        int n = c.length;

        int sol[] = new int[n];
        int opt_sol[] = new int[n];

        int card_opt =  busca_maxima_cardinalidad(0,0,sol,opt_sol,-1,c,f);

        // opcional
        for (int i=0; i<n-1; i++)
            System.out.print(opt_sol[i] + " ");
        System.out.println(opt_sol[n-1]);

        return card_opt;
    }

    // función no pedida en el enunciado:
    public static void main(String args[]) throws Exception {
        //lectura:
        BufferedReader cin =
            new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(cin.readLine());

        int c[] = new int[n];
        int f[] = new int[n];

        //lectura:
        String[] listaEnteros = cin.readLine().split(" ");
        for (int i=0; i<n; i++) {
            int m = Integer.parseInt(listaEnteros[i]);
            c[i] = m;
        }

        listaEnteros = cin.readLine().split(" ");
        for (int i=0; i<n; i++) {
            int m = Integer.parseInt(listaEnteros[i]);
            f[i] = m;
        }

        //escritura:
            System.out.println(planificacion_tareas(c,f));
    }
}
```

Python:

```python
def es_candidato_valido(i,sol,c,f):

    es_valido = True

    j=0
    while j<i and es_valido:
       # ver si la tarea i se solapa con las anteriores,
       # previamente incluidas en la solución parcial
        if sol[j]==1:
            es_valido = (c[i]>=f[j] or c[j]>=f[i])

        j = j + 1

    return es_valido

def busca_maxima_cardinalidad(i,card_actual,sol,opt_sol,card_opt,c,f):
    n = len(sol)

    if i==n:
        if card_actual>card_opt:  # actualizar si se encuentra una solución
            mejor
            card_opt = card_actual
            for k in range(n):
                opt_sol[k] = sol[k]
    else:
        for k in range(0,2):  # árbol de recursión binario

            if k==0 or es_candidato_valido(i,sol,c,f):

                sol[i]=k

                card_actual = card_actual + k

                card_opt = busca_maxima_cardinalidad(i+1,card_actual,sol,
                    opt_sol,card_opt,c,f)

    return card_opt

def planificacion_tareas(c,f):
    n = len(c)

    sol = [None]*n
    opt_sol = [None]*n

    card_opt =  busca_maxima_cardinalidad(0,0,sol,opt_sol,-1,c,f)

    print(opt_sol)  # opcional
    return card_opt

# No pedido en el enunciado:
c = [int(x) for x in input().split()]
f = [int(x) for x in input().split()]
print(planificacion_tareas(c,f))
```