Diseño de algoritmos

Jesús Bermúdez de Andrés. UPV-EHU
Guías para la solución de ejercicios: Programación Dinámica

Curso 2008-09

1. Sea A una tabla de n números enteros con $n > 0$, y sea la función

```
func Suma?(A, 1, i) return boolean
{pre: i > 0}
  if i = 1 ∧ A(1) ≠ 0 then return false
  elsif A(i) = Sumatorio(A, 1, i − 1) then return true
  else return Suma?(A, 1, i − 1)
```

siendo

```
func Sumatorio(A, 1, k) return integer
  s ← 0
  for j ← 1 to k loop s ← s + A(j) end loop
  return s
```

La evaluación de Suma?$(A,\ 1,\ n)$ devuelve un valor booleano que indica si alguno de los elementos de la tabla $A(1..n)$ coincide con la suma de todos los elementos que le preceden. Analice la eficiencia de este algoritmo. Utilice otra técnica para escribir un algoritmo que resuelva el problema de un modo más eficiente. Analice la eficiencia de su propuesta.

Solución:
Es evidente que Sumatorio$(A,\ 1,\ k)$ es $\Theta(k)$. Entonces la función de coste de Sumatorio$(A,\ 1,\ n)$, en el caso peor, es de la forma $f(n) = f(n-1) + \Theta(\mathrm{n})$ que resulta de $\Theta(n^2)$. Además, en el caso peor, usa espacio extra $O(n)$ debido a la pila de recursión.

Un modo simple de resolver más eficientemente el problema, consiste en evitar la repetición innecesaria de cálculos al invocar Sumatorio repetidas veces. Bastaría con memorizar en una tabla $S(1..n)$ los valores que el algoritmo calcula con Sumatorio, de manera que $S(i) =$ Sumatorio$(A,\ 1,\ i)$. Denominamos M_Suma? al nuevo algoritmo propuesto:

```
func M_Suma?(A, 1, n) return boolean
  if A(1) = 0 then return true
  else
```

```
S(1) ← A(1)
for i ← 2 to n loop
  if A(i) = S(i − 1) then return true
  else
    S(i) ← S(i − 1) + A(i)
end loop
return false
```

Es fácil ver que M_Suma?$(A,\ 1,\ n)$ es $O(n)$ en tiempo y que usa espacio extra de $\Theta(n)$ debido a la tabla $S(1..n)$.

2. Sea el siguiente programa, con X e Y números positivos:

```
func Principal (N, X, Y) return real
{pre: N ≥ 2}
T(1..N): array de números reales

  func Aux (N) return real
    if T(N − 1) = 0 then T(N − 1) ← Aux(N − 1)
    if T(N − 2) = 0 then T(N − 2) ← Aux(N − 2)
    return (T(N − 1) + T(N − 2))/2

begin Principal
  T(1) ← X
  T(2) ← Y
  for i ← 3 to N loop T(i) ← 0
  if N ≤ 2 then return T(N)
  else return Aux(N)
end
```

a) Defina la función que calcula Principal$(n,\ X,\ Y)$.

b) Indique la técnica que se ha usado en el diseño del programa anterior.

c) Analice el tiempo de ejecución de Principal$(n\ X,\ Y)$.

d) ¿Usaría este algoritmo en el caso de que $X = Y$? ¿Por qué?

Solución:

a) La función $P(n, X, Y)$ que calcula Principal$(n\ X,\ Y)$ queda definida por la recurrencia siguiente:

$$P(n,X,Y) = \begin{cases} X & \text{si } n = 1 \\ Y & \text{si } n = 2 \\ (P(n-1,X,Y) + P(n-2,X,Y))/2 & \text{si } n > 2 \end{cases}$$

b) La técnica que se ha usado en el diseño de Principal$(n\ X,\ Y)$ es la de funciones con memoria en Programación Dinámica.

c) El tiempo de ejecución de PRINCIPAL($n\ X,\ Y$) es $\Theta(n)$, ya que calcula los valores de una tabla $T(1..n)$ y cada valor lo calcula en $O(1)$.

d) No merece la pena usar PRINCIPAL($n\ X,\ Y$) si $X = Y$ ya que, en ese caso, resulta que $\forall n\ P(n, X, X) = X$; y entonces ese valor puede calcularse en $O(1)$.

3. Dado M, un número natural no negativo, ¿cuál es el mayor valor que podemos conseguir multiplicando n números naturales que sumen M?

   Es decir, deseamos

$$\begin{array}{rl} \mathit{maximizar} & x_1 \times \ldots \times x_n \\ \mathit{sujeto\ a} & x_1 + x_2 + \ldots + x_n = M \end{array}$$

   Pretendemos utilizar la técnica de programación dinámica para resolver el problema. Para ello consideramos la función siguiente:

   > $MAX(k, C)$= máximo valor de $x_1 \times \ldots \times x_k$ sujeto a $x_1 + \ldots + x_k = C$ siendo $x_1, \ldots,\ x_k$ números naturales.

   de la que necesitamos el valor $MAX(n, M)$.

   a) Defina esa función de forma recurrente para aplicar convenientemente la técnica.

   b) Escriba el algoritmo que calcule los valores de esa función siguiendo la recurrencia definida en el punto anterior.

   c) Analice los recursos de tiempo y espacio necesarios para su algoritmo.

   Solución:

   a) La función

   > $MAX(k, C)$= máximo valor de $x_1 \times \ldots \times x_k$ sujeto a $x_1 + \ldots + x_k = C$ siendo $x_1, \ldots,\ x_k$ números naturales.

   puede definirse mediante las ecuaciones siguientes:

$$\begin{array}{rcl} MAX(k, 0) & = & 0 \quad \forall k \geq 0 \\ MAX(1, C) & = & C \quad si\ C > 0 \\ MAX(k, C) & = & \textbf{máximo}\{x \ \times \ MAX(k-1, C-x) \mid 1 \leq x \leq C\} \\ & & si\ k > 1 \ \wedge \ C > 0 \end{array}$$

   b) Diseñaremos un algoritmo que calcule los valores de una tabla $T(0..n, 0..M)$, de manera que $T(k, C) = MAX(k, C)$.

```
func MAX(n, M) return natural
  for k in 0..n loop T(k,0) ← 0
  for C in 1..M loop T(1,C) ← C
  for k in 2..n loop
    for C in 1..M loop
      T(k,C) ← 0
      for x in 1..C loop
        T(k,C) ← máximo{T(k,C), x × T(k−1,C−x)}
  return T(n,M)
```

c) Es evidente que la función de coste temporal de MAX$(n,\ M)$ es $\Theta(nM)$, y que necesita $\Theta(nM)$ de espacio extra.

4. La operación de multiplicación de matrices es asociativa, por lo tanto hay muchas formas distintas de multiplicar n matrices $A_1 \times \ldots \times A_n$. Diseñe un algoritmo, utilizando la técnica de programación dinámica, que calcule el número de formas distintas de poner todos los paréntesis pertinentes para determinar el orden en que deben multiplicarse las n matrices.

Solución:
Definimos recursivamente la función:

> $f(n)$ = número de formas de poner todos los paréntesis para la multiplicación de n matrices.

Obsévese que si ponemos un par de paréntesis agrupando a las i primeras matrices y otro par agrupando al resto de $n-i$ matrices, es decir: $(A_1 \times \ldots \times A_i)(A_{i+1} \times \ldots \times A_n)$, reducimos el problema a calcular $f(i)$ por $(A_1 \times \ldots \times A_i)$ y $f(n-i)$ por $(A_{i+1} \times \ldots \times A_n)$. Además, por cada una de las formas de poner paréntesis a $(A_1 \times \ldots \times A_i)$ tenemos todas las formas de poner paréntesis a $(A_{i+1} \times \ldots \times A_n)$. Por lo tanto, el número de formas de poner el resto de los paréntesis a $(A_1 \times \ldots \times A_i)(A_{i+1} \times \ldots \times A_n)$ sería $f(i)f(n-i)$.

Ahora bien, ese número i puede ser desde 1 a $n-1$; por tanto el número $f(n)$ es la suma de todas esas posibilidades.

$$\begin{aligned} f(n) &= 1 \text{ si } n \leq 2 \\ f(n) &= \sum_{i=1}^{n-1} f(i)f(n-i) \text{ si } n > 2 \end{aligned}$$

Podemos calcular los valores de esa función en una tabla $F(1..n)$ de manera que $F(i) = f(i)$.

```
func FORMAS_PARÉNTESIS(n) return natural
{pre: n ≤ 2}
  F(1) ← 1
  F(2) ← 1
  for i in 3..n loop
```

```
    F(i) ← 0
    for k in 1..i − 1 loop
      F(i) ← F(i) + F(k)F(i − k)
  return F(n)
```

Este algoritmo es $\Theta(n^2)$ en tiempo y necesita $\Theta(n)$ de espacio extra.

5. Disponemos de n tipos de sellos de valores correspondientes $\{v_1, \ldots, v_n\}$ (todos ellos números enteros positivos). ¿De cuántas formas distintas puede franquearse una carta con tarifa postal T? (El importe de franqueo debe ser exactamente T y, naturalmente, no importa el orden en el que se peguen los sellos).

   Diseñe un algoritmo con la técnica de programación dinámica que resuelva este problema. Analice su algoritmo.

   Solución:
   Una ecuación conveniente, para utilizar la técnica de programación dinámica, es la siguiente:

   > $Formas(n, T)$ = número de formas de franquear tarifa T usando sellos de tipo $\{1, \ldots, n\}$

$$
\begin{array}{rcl}
Formas(0, T) & = & 0 \text{ si } T > 0 \\
Formas(n, 0) & = & 1 \text{ si } n \geq 0 \\
Formas(n, T) & = & Formas(n-1, T) + Formas(n, T - v_n) \\
 & & \text{si } n > 0 \wedge T > 0 \wedge T \geq v_n \\
Formas(n, T) & = & Formas(n-1, T) \text{ si } n > 0 \wedge T > 0 \wedge T < v_n
\end{array}
$$

   Diseñamos un algoritmo que calcule los valores de una tabla $F(0..n, 0..T)$, de manera que $F(i, j) = Formas(i, j)$. Nos interesa el valor $F(n, T)$.

```
func Formas(n, T) return natural
  for j in 1..T loop F(0, j) ← 0
  for i in 0..n loop F(i, 0) ← 1
  for i in 1..n loop
    for j in 1..T loop
      F(i, j) ← F(i − 1, j)
      if j ≥ v_i then F(i, j) ← F(i, j) + F(i, j − v_i)
  return F(n, T)
```

   Este algoritmo es $\Theta(nT)$ tanto en tiempo como en espacio extra.

6. En un archipiélago, con multitud de pequeñas islas cercanas, hay puentes que unen ciertos pares de islas entre sí. Para cada puente (que puede ser de dirección única), además de saber la isla de origen y la isla de destino, se conoce su

anchura (número entero mayor que 0). La *anchura de un camino*, formado por una sucesión de puentes, es la anchura mínima de las anchuras de todos los puentes que lo forman. Para cada par de islas se desea saber cuál es el camino de *anchura máxima* que las une (siempre que exista alguno).

Diseñe un algoritmo con la técnica de programación dinámica que resuelva este problema. Analice su algoritmo.

Solución:
Denominaremos $A(i, j)$ la anchura del puente que va de la isla i a la isla j. Si no hay puente en esa dirección entonces $A(i, j) = 0$. Obsérvese que conviene $A(i, i) = \infty$ puesto que para ir de una isla a ella misma no debe existir ninguna restricción.

Definimos, de manera recursiva, la siguiente función:

> $Max\_Anchura(i,\ j,\ k)$ = máxima anchura de los caminos que van de la isla i a la isla j, pudiendo pasar por las islas $\{1, \ldots, k\}$

$$
\begin{aligned}
Max\_Anchura(i,\ j,\ 0) &= A(i,\ j) \\
Max\_Anchura(i,\ j,\ k) &= \mathbf{max}\{Max\_Anchura(i,\ j,\ k-1), \\
&\qquad \mathbf{min}\{Max\_Anchura(i,\ k,\ k-1), \\
&\qquad\quad Max\_Anchura(k,\ j,\ k-1)\}\} \ \text{ si } \ k > 0
\end{aligned}
$$

Diseñamos un algoritmo que calcule los valores de una tabla $M(1..n, 1..n)$, de manera que, al terminar, $M(i, j) = Max\_Anchura(i,\ j,\ n)\ \forall i,\ j$

```
proc MÁXIMA_ANCHURA(A, M)
  M(1..n, 1..n) ← A(1..n, 1..n)
  for k in 1..n loop
    for i in 1..n loop
      for j in 1..n loop
        if M(i,k) < M(k,j) then aux ← M(i,k)
        else aux ← M(k,j)
        if M(i,j) < aux then M(i,j) ← aux
```

Este algoritmo es $\Theta(n^3)$ en tiempo, y necesita espacio extra en $\Theta(n^2)$.

7. Una empresa de inversiones dispone de una tabla *Renta(1..M, 1..p)* en la que tiene registrados los porcentajes de beneficio por invertir en determinados productos financieros, numerados $1, \ldots, p$. En concreto, *Renta(d, f)* es el porcentaje de beneficio por invertir d decenas de euros en el producto financiero f. Es importante considerar que el porcentaje de beneficio para cada producto f, varía con la cantidad de euros d que se inviertan.

   Utilice la técnica de *programación dinámica* para escribir un algoritmo que calcule el máximo beneficio (en euros) que se puede obtener con D decenas de euros (se supone que $D \leq M$).

Solución:
Definimos recursivamente la siguiente función:

> $MB(d,\ f)$ = máximo beneficio que se puede obtener invirtiendo d decenas de euros con los productos $\{1, \ldots, f\}$

$$
\begin{array}{rcl}
MB(d,\ 0) & = & 0 \quad \forall d \geq 0 \\
MB(0,\ f) & = & 0 \quad \forall f \geq 0 \\
MB(1,\ f) & = & (\mathbf{max}\{\mathit{Renta(1,\ i)} \mid 1 \leq i \leq f\} \times 10)/100 \quad \text{si} \ \ f \geq 1 \\
MB(d,\ f) & = & \mathbf{max}\{B(d,\ f-1), \\
 & & \{MB(d-k, f-1) + (\mathit{Renta(k,\ f)} \times k \times 10)/100) \mid 1 \leq k \leq d\}\} \\
 & & \text{si} \ \ d \geq 1 \wedge f \geq 1
\end{array}
$$

Buscamos el valor $MB(D,\ p)$

Diseñamos un algoritmo que calcule los valores de una tabla $B(0..D, 0..p)$, de manera que, al terminar, $B(i,j) = MB(i,\ j,)\ \forall i,\ j$

```
proc MÁXIMO_BENEFICIO(Renta, B)
  for d in 0..D loop B(d,0) ← 0
  for f in 0..p loop B(0,f) ← 0
  for f in 1..p loop
    for d in 1..D loop
      B(d,f) ← B(d,f-1)
      for k in 1..d loop
        aux ← B(d-k,f-1) + (Renta(k, f)×k × 10)/100)
        if B(d,f) < aux then B(d,f) ← aux
```

Este algoritmo es $\Theta(pD^2)$ en tiempo, y necesita espacio extra en $\Theta(pD)$.

8. Maite y Josepo han recibido un montón de regalos por su estupendo trabajo en una serie de televisión de reconocida fama. Cada regalo viene en una caja destinada a ambos. Como no tienen suficiente tiempo para desempaquetar y mirar qué es cada cosa, han decidido utilizar el siguiente criterio para repartirse los regalos: cada uno debe quedarse con la misma cantidad de peso; para ello cuentan con los pesos de cada una de las cajas $P_1, \ldots, P_n$ (números enteros positivos). Al cabo de un buen rato, todavía no han conseguido hacer el reparto según ese criterio. Diseñe un algoritmo, utilizando la técnica de programación dinámica, que resuelva el problema de esta pareja y determine una forma de reparto de los regalos, si es que es posible. Analice su algoritmo.

Solución:
Digamos que la suma de los pesos es $D = P_1 + \ldots + P_n$. Asumimos que D es par, puesto que en otro caso terminamos inmediatemente respondiendo que no

es posible ese reparto paritario. Entonces el problema consiste en determinar si existe un subconjunto de las cajas tal que la suma de sus pesos sea $\frac{D}{2}$.

Definimos la función booleana siguiente para aplicar la técnica de programación dinámica.

> $S(m,k) = True$ si existe un subconjunto de $\{P_1, \ldots, P_k\}$ que sume m.
> En caso contrario $S(m,k) = False$

$$
\begin{aligned}
S(0,k) &= True \;\; \forall k \geq 0 \\
S(m,0) &= False \;\; si \;\; m > 0 \\
S(m,k) &= S(m,k-1) \vee S(m-P_k, k-1) \;\; si \;\; m \geq P_k \;\; y \;\; k > 0 \\
S(m,k) &= S(m,k-1) \;\; si \;\; m < P_k \;\; y \;\; k > 0
\end{aligned}
$$

Necesitamos el valor $S(\frac{D}{2}, n)$. El algoritmo que escribimos a continuación, calcula ese valor con ayuda de una tabla con el mismo nombre S. Además, el algoritmo incluye unas sentencias que dejan las marcas pertinentes para poder construir después la colección de regalos de uno de ellos; el otro tomaría el resto de los regalos.

```
func HAY_REPARTO(P(1..n), S(0..D/2, 0..n)) return (Boolean × Conjunto)
  for k in 0..n loop S(0,k) ← True
  for m in 1..D/2 loop S(m,0) ← False
  for k in 1..n loop
    for m in 1..D/2 loop
      if m < P(k) then
        S(m,k) ← S(m,k-1)
        marca(m,k) ← False
      elsif S(m-P(k), k-1) then
        S(m,k) ← S(m-P(k), k-1)
        marca(m,k) ← True
      else
        S(m,k) ← S(m, k-1)
        marca(m,k) ← False
  end loop
  {Ahora calculamos la colección de regalos}
  if S(D/2, n) then
    i ← D/2
    j ← n
    C ← ∅
    while j > 0 loop
      if marca(i,j) then
```

```
        C ← C ∪ {j}
        i ← i − P(j)
        j ← j − 1
      else
        j ← j − 1
  else
    return (False, ∅)
```

Este algoritmo es $\Theta(nD)$ tanto en tiempo como en espacio extra.

9. Una academia, recién inaugurada, pretende impartir un número L de horas lectivas. Para ello puede realizar contratos de profesores de n clases. Los profesores con contrato de la clase i imparten un máximo de h_i horas lectivas, percibiendo por ese motivo una cantidad p_i de reales de vellón. Determine cuál es el número de contratos necesario para impartir L horas lectivas, de forma que la cantidad de reales de vellón a pagar sea mínima. Analice su algoritmo.

   Solución:
   Primero definimos recursivamente la función de optimización, que es la que se refiere al mínimo precio para impartir L horas lectivas.

   > $Precio(H)$ = mínimo precio para impartir H horas, contando con la posibilidad de realizar contratos de las clases $\{1, \ldots, n\}$.

   $$\begin{aligned} Precio(H) &= 0 \text{ si } H \leq 0 \\ Precio(H) &= \mathbf{min}\{Precio(H - h_i) + p_i \mid 1 \leq i \leq n\} \text{ si } H > 0 \end{aligned}$$

   El número de contratos necesarios viene especificado por la siguiente función:

   > $Contratos(H)$ = número de contratos necesarios para poder impartir H horas, minimizando el precio de los contratos.
   >
   > $Contratos(H) = 1 + Contratos(H - h_i)$
   > siendo i el índice de la clase de contrato que minimizó $Precio(H)$.

   Por tanto, para calcular $Contratos(L)$ hay que calcular $Precio(L)$. Para eso calcularemos un array $P(1..L)$ de manera que $P(x) = Precio(x)$. A la par, podemos ir calculando otro array $C(1..L)$ de manera que $C(x) = Contratos(x)$.

   ```
   func Contratos(h(1..n), p(1..n), L) return entero
     P(0) ← 0
     for H in 1..L loop
       aux ← ∞
       for i in 1..n loop
         if h(i) ≤ H then
   ```

```
        if P(H − h(i)) + p(i) < aux then
            aux ← P(H − h(i)) + p(i)
            clase ← i
    else
        if p(i) < aux then
            aux ← p(i)
            clase ← i
  C(H) ← 1 + C(H − h(clase))
return C(L)
```

Este algoritmo es $\Theta(nL)$ en tiempo y necesita espacio extra de $\Theta(L)$.

10. Dada una cantidad M ¿cuál es el máximo número X, con $X \leq M$, que podemos calcular partiendo de una cantidad inicial I y aplicando repetidamente las operaciones $\times 2$, $\times 3$, $\times 5$? Emplee la técnica de la programación dinámica para determinar dicho número X y analice el orden del algoritmo propuesto.

    Complete el algoritmo anterior con las instrucciones pertinentes para que, además, sepamos qué secuencia de operaciones nos lleva a ese máximo X.

    Solución:
    Primero definimos una función recursiva que especifica el problema de optimización. En este caso puede servir la siguiente, que definimos con un solo argumento, ya que el parámetro M y los multiplicadores $\times 2$, $\times 3$, $\times 5$ son fijos.

    > $Max\_Desde(c)$ = mayor número, menor o igual que M, que puedo alcanzar aplicando repetidamente las operaciones $\times 2$, $\times 3$, $\times 5$, al número c.

$$
\begin{aligned}
Max\_Desde(c) &= c \text{ si } M < c \times 2 \\
Max\_Desde(c) &= Max\_Desde(c \times 2) \text{ si } c \times 2 \leq M < c \times 3 \\
Max\_Desde(c) &= \mathbf{max}\{Max\_Desde(c \times 2), Max\_Desde(c \times 3)\} \\
&\quad \text{si } c \times 3 \leq M < c \times 5 \\
Max\_Desde(c) &= \mathbf{max}\{Max\_Desde(c \times 2), Max\_Desde(c \times 3), \\
&\quad Max\_Desde(c \times 5\} \\
&\quad \text{si } c \times 5 \leq M
\end{aligned}
$$

Interesa el valor $Max\_Desde(I)$. Vamos a calcularlo usando una tabla $MD(I..M)$ tal que $MD(i) = Max\_Desde(i)$. Además, el algoritmo incluye unas sentencias que dejan las marcas pertinentes para poder construir después una secuencia de multiplicaciones $\times 2$, $\times 3$ o $\times 5$, que nos lleve desde I hasta $Max\_Desde(I)$. El vector de marcas debe interpretarse del siguiente modo: $marca(i) = 1$, significa que ya no deben realizarse multiplicaciones desde el valor i; $marca(i) = 2$, $marca(i) = 3$ y $marca(i) = 5$ significan que debe realizarse la multiplicación $\times 2$, $\times 3$ y $\times 5$, respectivamente, desde el valor i.

```
func MAX_DESDE(I, M)) return (Boolean × Secuencia)
  for c ← M downto I loop
    if M < c × 2 then MD(c) ← c;  marca(c) ← 1
    elsif c × 2 ≤ M < c × 3 then MD(c) ← MD(c × 2);  marca(c) ← 2
    elsif c × 3 ≤ M < c × 5 then
      if MD(c × 2) < MD(c × 3) then
        MD(c) ← MD(c × 3);  marca(c) ← 3
      else MD(c) ← MD(c × 2);  marca(c) ← 2
    else {c × 5 ≤ M}
      if MD(c × 2) > MD(c × 3) ∧ MD(c × 2) > MD(c × 5) then
        MD(c) ← MD(c × 2);  marca(c) ← 2
      if MD(c × 3) > MD(c × 2) ∧ MD(c × 3) > MD(c × 5) then
        MD(c) ← MD(c × 3);  marca(c) ← 3
      if MD(c × 5) > MD(c × 2) ∧ MD(c × 5) > MD(c × 3) then
        MD(c) ← MD(c × 5);  marca(c) ← 5
  end loop
  {Ahora calculamos la secuencia de multiplicaciones}
  c ← I
  S ← [ ]
  while marca(c) ≠ 1 loop
    S ← S ⊕ [marca(c)]
    c ← marca(c) × c
  end loop
  return (MD(I), S)
```

Este algoritmo es $\Theta(M - I)$ tanto en tiempo como en espacio extra.