# Ejercicio 1. Programación dinámica

[ITIS/ITIG, Feb 2007]

Dada una cantidad de dinero $C$, deseamos encontrar el mínimo número de monedas que cubra exactamente, si es posible, esta cantidad. Para ello conocemos los valores de los $N$ tipos de monedas y el número de monedas disponibles de cada tipo. Diseña un algoritmo dinámico que resuelva el problema detallando lo siguiente:

- La estructura de cálculos intermedios. (0,5 puntos)
- La relación recursiva. (1 punto)
- La función o procedimiento que implemente este algoritmo. (2,5 puntos).

# Solución Ejercicio 1. Programación dinámica

- Este ejercicio es muy parecido al que hemos visto anteriormente de devolución del cambio
- La diferencia es que el número de monedas disponibles de cada tipo es limitado
- Recordando el caso de estudio de devolución del cambio, la expresión recurrente que lo definía era:

$$C[i,j] = \text{mín}(C[i-1,j], 1 + C[i, j-d_i])$$

- De los dos accesos recurrentes de la expresión anterior:
  a) $C[i-1,j]$ corresponde con la decisión de no tomar ninguna moneda de valor $d_i$. Este caso no modifica el número de monedas utilizado para obtener $C[i,j]$. No afecta la limitación en el número de monedas.
  b) $1 + C[i, j-d_i]$ corresponde con la selección de una moneda de valor $d_i$ (al menos). Este caso sí modifica el número de monedas que se necesita para obtener $C[i,j]$ en función de $C[i, j-d_i]$, pues se utiliza una moneda de valor $d_i$.

# Solución Ejercicio 1. Programación dinámica (cont.)

- Una posible solución es considerar tantos casos como monedas de valor $d_i$ estén disponibles, y después calcular el mínimo de todos ellos

$$C[i,j] = \min_{0 \le k \le m_i} (k + C[\mathbf{i-1}, j - d_i * k])$$

donde $m_i$ es el número de monedas disponibles de valor $d_i$.

- La recurrencia completa es:

$$C(i,j) = \begin{cases} 0 & si\ j = 0 \\ +\infty & si\ j < 0 \\ +\infty & si\ i = 1 \\ & y\ (j < d_i\ o\ j > d_i * m_i \\ & o\ j\ mod\ d_i \neq 0) \\ \lfloor j/d_i \rfloor & si\ i = 1\ y\ j\ mod\ d_i = 0 \\ & y\ j \le d_i * m_i \\ \min\limits_{0 \le k \le m_i} (k + C[i-1, j - d_i * k]) & \text{en otro caso} \end{cases}$$

- Para cada uno de los valores de $k$ (desde 1 hasta $m_i$) se considera tomar las $k$ monedas de valor $d_i$, en lugar de hacerlo de una en una como en la versión original del problema de devolución del cambio

# Solución Ejercicio 1. Programación dinámica (cont.)

```
fun monedas2(d[1..N], m[1..N],Cant)
  crear C[1..N,0..Cant]
  desde i← 1 hasta N hacer C[i,0] ← 0
  desde i← 1 hasta N hacer
    desde j← 1 hasta Cant hacer
      si i=1 Y j<d[i] entonces C[i,j]← +∞
      si no si i=1 entonces
        si j mod d[i] = 0 Y j div d[i] ≤ m[i] entonces C[i,j]← j div d[i]
        si no C[i,j]← +∞
      si no
        min ← +∞
        desde k← 0 hasta mín(j div d[i], m[i]) hacer
          si min > k+C[i-1,j-k*d[i]] entonces min ← k+C[i-1,j-k*d[i]]
        fin desde
        C[i,j] ← min
      fin si
    fin desde
  fin desde
  devolver C[N,Cant]
fin fun
```

# Ejercicio 2. Programación dinámica

[ITIS/ITIG, Sep 2007] En el departamento de una empresa de traducciones se desea hacer traducciones de textos entre varios idiomas. Se dispone de algunos diccionarios. Cada diccionario permite la traducción (bidireccional) entre dos idiomas. En el caso más general, no se dispone de diccionarios para cada par de idiomas por lo que es preciso realizar varias traducciones. Dados $N$ idiomas y $M$ diccionarios, determina si es posible realizar la traducción entre dos idiomas dados y, en caso de ser posible, determina la cadena de traducciones de longitud mínima.
Diseña un algoritmo dinámico que resuelva el problema detallando lo siguiente:

1. La estructura de cálculos intermedios utilizada (0,5 puntos).
2. La relación recursiva entre subproblemas (0,5 puntos).
3. El procedimiento o función que implemente el algoritmo (2 puntos).

# Ejercicio 2. Programación dinámica (cont.)

- Ejemplo 1:
  Traducir del latín al arameo disponiendo de los siguientes diccionarios:
  *latín-griego*, *griego-etrusco*, *griego-demótico*, *demótico-arameo*
  Solución: sí es posible la traducción: latín-griego-demótico-arameo, realizando tres traducciones.
- Ejemplo 2:
  Traducir del latín al arameo disponiendo de los siguientes diccionarios:
  *latín-griego*, *arameo-etrusco*, *griego-demótico*, *demótico-hebreo*
  Solución: No es posible la traducción.

# Solución ejercicio 2. Programación dinámica

- Una forma sencilla de representar este problema es considerando las secuencias de traducciones como "caminos" de un idioma a otro
- De esta forma, podemos considerar los idiomas como los nodos de un grafo, y los diccionarios como las aristas
- Es importante tener en cuenta lo que se pide como resultado del algoritmo:
  a) saber si es posible la traducción entre dos idiomas dados
  b) y la cadena de traducciones de longitud mínima
- Vemos que es un problema de optimización (*cadena de traducciones de longitud mínima*), y conocemos un algoritmo para calcular los caminos mínimos entre cualquier par de nodos de un grafo: el *algoritmo de Floyd*
- El único dato que nos falta para aplicarlo directamente es la longitud y orientación de las aristas

# Solución ejercicio 2. Programación dinámica (cont.)

- Si sólo nos pidieran saber si es posible la traducción entre dos idiomas, podríamos utilizar una versión simplificada del algoritmo de Floyd: el algoritmo de Warshall, que utiliza una matriz de adyacencia booleana para saber si dos nodos están conectados o no (en lugar de una matriz de distancias), [GV00], p. 197
- Como nos piden la cadena mínima, debemos asignar una distancia en cada arista, por ejemplo 1
- También se dice que los diccionarios permiten traducciones bidireccionales: por cada diccionario habrá dos aristas, cada una en un sentido (la matriz de adyacencia será simétrica)
- Respecto a las estructuras de datos intermedias:
  - En el algoritmo de Floyd vimos que la estructura de datos intermedia coincide con la matriz resultado, y es una matriz cuadrada $N \times N$
  - También se nos pide la cadena de traducciones por las que tenemos que pasar: necesitamos una matriz para almacenar los nodos intermedios por los que pasan los caminos mínimos

# Solución ejercicio 2. Programación dinámica (cont.)

```
fun trad(L[1..n,1..n],i,j,c[1..n])
   crear D[1..n,1..n],P[1..n,1..n]
   floyd(L,D,P)
   si D[i,j] ≠ +∞ entonces
      k ← 1
      cam(P,i,j,c,k)
      c[k] ← -1
   fin si
   devolver D[i,j]
fin fun

proc cam(P[1..n,1..n],i,j,c[1..n],k)
   si P[i,j] ≠ 0 entonces
      cam(P,i,P[i,j],c,k)
      c[k] ← P[i,j]
      k ← k+1
      cam(P,P[i,j],j,c,k)
   fin si
fin proc
```

```
proc floyd(L[1..n,1..n],D[1..n,1..n],P[1..n,1..n])

   desde i ← 1 hasta n hacer
      desde j ← 1 hasta n hacer
         P[i,j] ← 0
      fin desde
   fin desde
   D ← L
   desde k ← 1 hasta n hacer
      desde i ← 1 hasta n hacer
         desde j ← 1 hasta n hacer
            si D[i,k]+D[k,j] < D[i,j] entonces
               D[i,j] ← D[i,k]+D[k,j] ; P[i,j] ← k
            fin si
         fin desde
      fin desde
   fin desde
fin proc
```

# Ejercicio 3. Programación dinámica

[ITIS/ITIG, Jun 2007] Un excéntrico nutricionista va a un restaurante. En la carta aparecen todos los platos disponibles con el número de calorías. El nutricionista conoce el número mínimo de calorías que su cuerpo necesita en esa comida. Su objetivo es encontrar el menú que cubra exactamente esa cantidad de calorías o las supere de forma mínima. Además, no quiere repetir platos.
Diseña un algoritmo dinámico que determine qué platos forman parte del menú óptimo y el número de calorías del menú óptimo. Detalla lo siguiente:

1. Las estructuras y/o variables necesarias para representar la información del problema (0,5 puntos).
2. La estructura de cálculos intermedios utilizada (1 punto).
3. La relación recursiva entre subproblemas (1 punto).

4 y 5. El procedimiento o función que implementa el algoritmo (2,5 puntos).

# Solución ejercicio 3. Programación dinámica

- Este problema es de optimización (menú con mínimo número de calorías por encima de una cantidad)
- Entre los problemas que ya hemos visto, el más parecido es el de la mochila 0-1, pues los platos de la carta se eligen completos o no se eligen
- Las diferencias con este problema son: a) cómo se calcula el óptimo (mínimo en lugar de máximo), y b) la existencia de un valor de referencia (número mínimo de calorías necesario)
- Estas diferencias pueden resolverse utilizando una expresión recurrente apropiada para llenar la tabla
- Para conocer las estructuras necesarias para representar el problema es necesario analizar lo que se nos proporciona y se nos pide:
  - Se nos proporciona el número de calorías de cada plato: un vector $C[1..n]$, donde $n$ es el número de platos de la carta
  - Se nos pide el menú elegido: un vector $M[1..n]$ de los platos elegidos (el tamaño del vector es el número máximo de platos que podemos elegir)
  - También se nos pide el número de calorías de este menú

# Solución ejercicio 3. Programación dinámica (cont.)

- La estructura de datos intermedia puede ser, como en el problema de la mochila, una matriz $D[1..n, 0..Cal]$ con una fila por cada plato de la carta (desde 1 hasta $n$), y el número total de calorías en las columnas
- $D[i, j]$ contiene el número de calorías del menú óptimo considerando los $i$ primeros platos de la carta: aquél con el menor número de calorías (mayor o igual a $j$)
- El valor que buscamos es el correspondiente a $D[n, Cal]$
- Para calcular $D[i, j]$, se pueden dar dos casos:
  - Si no se elige el plato $i$ de la carta, entonces el menú elegido para $D[i, j]$ contendrá exclusivamente platos de 1 a $i - 1$: $D[i - 1, j]$
  - Si se elige el plato $i$, entonces el número de calorías total del menú $D[i, j]$ es $c_i + D[i - 1, j - c_i]$

# Solución ejercicio 3. Programación dinámica (cont.)

- Hay que tener en cuenta algunos casos especiales:
  - $D[1,j] = +\infty, j > c_1$, porque no es posible conseguir un menú de más de $c_1$ calorías utilizando exclusivamente el plato 1 (no se puede repetir el mismo plato)
  - $D[1,j] = c_1, 0 < j \leq c_1$
  - $D[i,j] = 0, j \leq 0$, pues el menú óptimo con al menos 0 (o menos) calorías consiste en no elegir ningún plato
- Por tanto, la expresión recurrente es la siguiente:

$$D[i,j] = \begin{cases} 0 & si\ j \leq 0 \\ +\infty & si\ i = 1\ y\ j > c_i \\ c_1 & si\ i = 1\ y\ 0 < j \leq c_i \\ \text{mín}(C[i-1,j], c_i + C[i-1, j-c_i]) & \text{en otro caso} \end{cases}$$

- la función mín() permite elegir el menú de menos calorías
- Los valores de la tabla con valor $+\infty$ garantizan que no se va a elegir un menú que tenga menos calorías que el mínimo necesario

# Solución ejercicio 3. Programación dinámica (cont.)

```
fun menu(c[1..n],Cal,m[1..n])
   crear D[1..n,0..Cal]
   desde pl ← 1 hasta n hacer
      D[pl,0] ← 0
      desde ncal ← 1 hasta Cal hacer
         si pl=1 Y ncal > c[1] entonces D[pl,ncal] ← +∞
         si no si pl=1 entonces D[pl,ncal] ← c[1]
         si no si ncal < c[pl] entonces
            D[pl,ncal] ← mín(D[pl-1,ncal],c[pl])
         si no
            D[pl,ncal] ← mín(D[pl-1,ncal],c[pl]+D[pl-1,ncal-c[pl]])
         fin si
      fin desde
   fin desde
   si D[n,Cal] < +∞ entonces obtener_menu(D,c,m)
   devolver D[n,Cal]
fin fun
```

# Solución ejercicio 3. Programación dinámica (cont.)

```
proc obtener_menu(D[1..n,0..Cal],c[1..n],m[1..n])
   plato ← 1
   i ← n
   j ← Cal
   mientras i > 0 Y j > 0 hacer
      si D[i,j]=D[i-1,j] entonces
         i ← i-1
      si no
         m[plato] ← i
         plato ← plato + 1
         j ← j-c[i]
         i ← i-1
      fin si
   fin mientras
   desde i ← plato hasta n hacer
      m[i] ← -1
   fin desde
fin proc
```

# Ejercicio 4. Programación dinámica

[GV00] Supongamos que existen $n$ bancos en los que podemos invertir, y disponemos de una cantidad *Cant* para invertirla. Cada banco nos proporciona intereses según una función monótona creciente, $f_i(x)$, donde $x$ es el importe a invertir e $i$ el banco en el que se invierte.

Diseñad un algoritmo dinámico que encuentre la inversión óptima: la cantidad que se debe invertir en cada banco para maximizar los intereses obtenidos.

# Solución ejercicio 4. Programación dinámica

- Este problema es muy parecido al anterior, aunque en este caso lo que se debe hacer es maximizar la expresión
$$f_1(x_1) + f_2(x_2) + ... + f_n(x_n)$$
con la restricción: $x_1 + x_2 + ... + x_n = Cant$
- Como en el caso anterior, vamos a intentar encontrar una expresión recurrente respecto al número de bancos considerados
- Representamos con $I(i,j)$ al interés máximo al invertir la cantidad $j$ en los primeros $i$ bancos:
$$I(i,j) = f_1(x_1) + f_2(x_2) + ... + f_i(x_i)$$
con la restricción $x_1 + x_2 + ... + x_i = j$.
- Se verifica el principio del óptimo: si $I(i,j)$ es el interés máximo para $i$ bancos, entonces $I(i-1, j-x_i)$ es el interés máximo para $i-1$ bancos (se verifica para cualquier número de bancos menor a $i$)
- Por tanto, la expresión recurrente resultante es:
$$I(i,j) = \begin{cases} f_1(j) & si\ i = 1 \\ \max\limits_{0 \le t \le j} (I(i-1, j-t) + f_i(t)) & \text{en otro caso} \end{cases}$$

# Solución ejercicio 4. Programación dinámica (cont.)

- Podemos considerar que las funciones $f_i(x)$ se proporcionan como valores de una matriz $F[1..n, 0..Cant]$, donde $F[i, j]$ contiene el importe de los intereses que proporciona el banco $i$ al invertir la cantidad $j$
- Como estructura de datos intermedia, utilizamos una matriz $I[1..n, 0..Cant]$
- Dada la estructura de la recurrencia, se puede generar la matriz $I$ por filas

# Solución ejercicio 4. Programación dinámica (cont.)

```
fun intereses(F[1..n,0..Cant],Cant)
  crear I[1..n,0..Cant]
  desde i ← 1 hasta n hacer
    I[i,0] ← 0
  fin desde
  desde j ← 1 hasta Cant hacer
    I[1,j] ← F[1,j]
  fin desde
  desde i ← 2 hasta n hacer
    desde j ← 1 hasta Cant hacer
      I[i,j] ← F[i,j]
      desde t ← 0 hasta j-1 hacer
        si I[i,j] < I[i-1,j-t]+F[i,t] entonces
          I[i,j] ← I[i-1,j-t]+F[i,t]
        fin si
      fin desde
    fin desde
  fin desde
  devolver I[n,Cant]
fin fun
```

- Falta la obtención de los importes invertidos en cada uno de los bancos
- ¿Cómo podría incluirse en este algoritmo?

# Ejercicio 5. Programación dinámica

[GV00] **Subsecuencia común máxima.** Dada una secuencia $X = \{x_1\ x_2\ ...\ x_m\}$, se dice que $Z = \{z_1\ z_2\ ...\ z_k\}$ $(k \leq m)$ es una subsecuencia de $X$ si existe una secuencia creciente de índices de $X$ $\{i_1\ i_2\ ...\ i_k\}$ tales que para todo $j = 1, 2...k$, $x_{i_j} = z_j$.

Dadas dos secuencias $X$ e $Y$, $Z$ es una subsecuencia común de $X$ e $Y$ si es subsecuencia de $X$ y subsecuencia de $Y$. Determinar utilizando programación dinámica la subsecuencia de longitud máxima común a $X$ e $Y$.

**Ejemplo**[1]**:** dadas $X = \{A, B, C, B, D, A, B\}$ e $Y = \{B, D, C, A, B, A\}$, la secuencia $\{B, C, A\}$ es una subsecuencia comun de ambas. Las secuencias $\{B, C, B, A\}$ y $\{B, D, A, B\}$ son subsecuencias comunes de máxima longitud.

[1] http://webdiis.unizar.es/~kftricas/Asignaturas/ea/Ejercicios/4-EjProgramacionDinamica.pdf

# Solución ejercicio 5. Programación dinámica

- Las secuencias de entrada son $X = \{x_1\ x_2\ ...\ x_m\}$ e $Y = \{y_1\ y_2\ ...\ y_n\}$
- Llamaremos $L(i, j)$ a la longitud de la secuencia común máxima de $X_i$ e $Y_j$, siendo éstas los prefijos de $X$ e $Y$ de longitudes $i$ y $j$, respectivamente:

$$X_i = \{x_1\ x_2\ ...\ x_i\}$$
$$Y_j = \{y_1\ y_2\ ...\ y_j\}$$

- ¿Se cumple el principio de optimalidad?

# Solución ejercicio 5. Programación dinámica

- Las secuencias de entrada son $X = \{x_1\ x_2\ ...\ x_m\}$ e $Y = \{y_1\ y_2\ ...\ y_n\}$
- Llamaremos $L(i,j)$ a la longitud de la secuencia común máxima de $X_i$ e $Y_j$, siendo éstas los prefijos de $X$ e $Y$ de longitudes $i$ y $j$, respectivamente:

  $X_i = \{x_1\ x_2\ ...\ x_i\}$
  $Y_j = \{y_1\ y_2\ ...\ y_j\}$
- ¿Se cumple el principio de optimalidad?
- En cada paso del algoritmo, podemos decidir si el último carácter de $X_i$ e $Y_j$ forma parte de la secuencia común máxima:
  - Si el último carácter de ambas secuencias es igual, entonces se añade a la secuencia común máxima: en este caso, $L(i,j)$ tiene el siguiente valor: $L(i-1,j-1)+1$
  - Si el último carácter de $X_i$ e $Y_j$ no es igual, entonces la secuencia común máxima no lo contiene, y el valor de $L(i,j)$ será el mayor de $L(i,j-1)$ y $L(i-1,j)$

# Solución ejercicio 5. Programación dinámica (cont.)

- Por tanto, la expresión recurrente es:

$$L(i,j) = \begin{cases} 0 & si\ i = 0\ o\ j = 0 \\ L(i-1,j-1)+1 & si\ i \neq 0, j \neq 0, x_i = y_j \\ \text{máx}(L(i,j-1), L(i-1,j)) & si\ i \neq 0, j \neq 0, x_i \neq y_j \end{cases}$$

- ¿Qué elemento de la tabla de cálculos intermedios contiene la longitud de la subsecuencia común máxima?
- ¿Cómo sería la implementación de esta recurrencia?
- ¿Cómo se puede obtener la subsecuencia común máxima?

# Ejercicio 6. Programación dinámica

[ITIS/ITIG, Feb 2006] Nicanor Cienfuegos quiere impresionar a su novia. Ha decidido gastarse todo su dinero en flores. Según los criterios estéticos de Nicanor el ramo ideal es aquel que minimiza el número de flores. Dado el prestigio de la floristería, piensa que para cada tipo de flor pueden vender un número infinito de copias. Ante la cara de asombro de su novia Nicanor recapacita, quizás no fue correcta su suposición. Mediante programación dinámica, diseñad dos algoritmos que resuelvan el problema detallando lo siguiente:

- Suposición 1: número infinito de copias (2 puntos):
  - Estructura de datos intermedia.
  - Relación recursiva entre casos y subcasos.
  - Implementación del código.
- Suposición 2: número finito de copias (2 puntos):
  - Estructura de datos intermedia.
  - Relación recursiva entre casos y subcasos.
  - Implementación del código.

# Solución ejercicio 6. Programación dinámica

- Este problema es muy parecido al de la devolución del cambio: hay que minimizar el número de flores, sin sobrepasar otra medida (el precio total)
- La diferencia está en que en este caso puede darse la situación de que no se gaste exactamente el dinero disponible, sino algo menos
- Para que el gasto sea lo más próximo a la cantidad disponible, debemos considerarlas en orden creciente de precio. Denominamos $p_i$ al precio de la flor $i$, $p_1 \leq p_2 \leq ... \leq p_n$
- Así, el importe no gastado será inferior a $p_1$
- Con $C(i,j)$ se representa el número mínimo de flores de 1 a $i$ que se pueden utilizar para gastar la cantidad $j$ de dinero
- La expresión de la recurrencia es:

$$C(i,j) = \begin{cases} +\infty & si\ j < 0 \\ 0 & si\ j = 0 \\ \lfloor j/p_1 \rfloor & si\ i = 1\ y\ j > 0 \\ \text{mín}(C(i-1,j), 1 + C(i,j-p_i)) & \text{en otro caso} \end{cases}$$

# Solución ejercicio 6. Programación dinámica (cont.)

- En la recurrencia anterior se ha considerado que se puede elegir cualquier número de flores de cada tipo (la expresión $1 + C(i, j - p_i)$ permite volver a seleccionar el mismo tipo de flor indefinidamente)
- Si no hay un número ilimitado de flores, la recurrencia debe considerar el número máximo de flores disponible por cada tipo
- Denominamos $f_i$ al número de flores del tipo $i$
- La expresión de la recurrencia pasa a ser la siguiente:

$$C(i,j) = \begin{cases} +\infty & si\ j < 0 \\ 0 & si\ j = 0 \\ \lfloor j/p_i \rfloor & si\ i = 1\ y\ j \leq p_i f_i \\ f_i & si\ i = 1\ y\ j > p_i f_i\ (*) \\ \min\limits_{0 \leq k \leq f_i} \{k + C(i-1, j - k * p_i)\} & \text{en otro caso} \end{cases}$$

(*) Podemos considerar que compramos todas las flores de la tienda y además nos sobra dinero

- La especificación del algoritmo es similar a las vistas en casos anteriores

# Ejercicio 7. Programación dinámica

[ITIS/ITIG, feb 2008] A Nicanor Cienfuegos le han hecho un regalo. Como no le gusta ha decidido cambiarlo por otros productos. Su cambio ideal es el siguiente: el valor de los productos tiene que ser igual al valor del regalo o superarlo de forma mínima. No le importa tener varias copias del mismo producto. Suponiendo conocidos los productos de la tienda, sus precios y el número de unidades de cada producto:

a) (3 puntos) Diseña un **algoritmo dinámico** que determine el valor de los productos elegidos en el canje, detallando
   - La estructura de cálculos intermedios,
   - La relación recursiva, y
   - La función o procedimiento que implemente este algoritmo.

b) (1 punto) Proporciona una función con memoria que implemente este algoritmo.

# Solución ejercicio 7. Programación dinámica

- Este ejercicio es una combinación de otros dos vistos en clase: cambio de moneda con un número limitado de monedas (feb 07), y el nutricionista (jun 07).
- La suma de los valores de los productos debe ser igual o superior (de forma mínima) a la cantidad de dinero disponible
- Existe un número limitado de objetos de cada tipo en la tienda
- Estructura para cálculos intermedios una tabla $C[1..N, 0..R]$, donde $N$ es el número de tipos de objetos en la tienda, y $R$ es el valor del regalo a cambiar
- La relación recursiva es la siguiente ($p_i$ es el precio y $u_i$ el número de unidades del objeto $i$):

$$C(i,j) = \begin{cases} 0 & si\ j \leq 0 \\ \lceil j/p_i \rceil * p_i & si\ i = 1\ y\ j \leq p_i * u_i \\ +\infty & si\ i = 1\ y\ j > p_i * u_i \\ \min\limits_{0 \leq k \leq K_i} (p_i * k + C[i-1, j - p_i * k]) & \text{en otro caso} \end{cases}$$

donde $K_i = \min(u_i, \lceil j/p_i \rceil)$.

# Solución ejercicio 7. Programación dinámica (cont.)

```
fun regalos(p[1..N], u[1..N],Cant)
   crear C[1..N,0..Cant]
   desde i← 1 hasta N hacer C[i,0] ← 0
   desde i← 1 hasta N hacer
      desde j← 1 hasta Cant hacer
         si i=1 entonces
            si j/p[i] ≤ u[i] entonces C[i,j]← ⌈j/p[i]⌉*p[i]
            si no C[i,j]← +∞
         si no
            min ← +∞
            desde k← 0 hasta mín(⌈j/p[i]⌉, u[i]) hacer
               si k*p[i]>j entonces min ← mín(min,k*p[i])
               si no min ← mín(min,k*p[i]+C[i-1,j-k*p[i]])
            fin desde
            C[i,j] ← min
         fin si
      fin desde
   fin desde
   devolver C[N,Cant]
fin fun
```

# Solución ejercicio 7. Programación dinámica (cont.)

- Una posible solución para el apartado b) es la siguiente:

```
//C[1..N,1..Cant] es una tabla global con valor inicial -1
fun regalos_mem(p[1..N], u[1..N],i,j)
   si j≤0 entonces devolver 0
   si C[i,j]≥0 entonces devolver C[i,j]
   si i=1 entonces
      si j/p[1] ≤ u[1] entonces C[1,j]← ⌈j/p[1]⌉*p[1]
      si no C[i,j]← +∞
   si no
      min ← +∞
      desde k← 0 hasta mín(⌈j/p[i]⌉, u[i]) hacer
         aux ← k*p[i]+regalos_mem(p,u,i-1,j-k*p[i])
         si min > aux entonces min ← aux
      fin desde
      C[i,j] ← min
   fin si
   devolver C[i,j]
fin fun
```

# Ejercicio 8. Programación dinámica

[ITIS/ITIG, Jun 2008] Una Organización No Gubernamental dispone de un fondo para la financiación de proyectos de ayuda, y se quiere financiar la realización del máximo número de proyectos posible.
Para ello, se dispone de una serie de tipos de proyecto diferentes, con la siguiente información disponible:

- Presupuesto del tipo de proyecto, $p_i$
- Número de regiones en las que se puede realizar, $r_i$

Mediante la técnica de programación dinámica, diseña un algoritmo que permita determinar cuántos proyectos se pueden realizar sin superar el importe total del fondo, indicando lo siguiente:

1. Estructura de datos intermedia (0,5 puntos)
2. Relación recursiva entre casos y subcasos (1 punto)
3. Código del algoritmo (3,5 puntos)

# Solución ejercicio 8. Programación dinámica

1) La estructura de datos intermedia es una tabla $C[1..n, 0..fondo]$ con tantas filas como tipos de proyectos, ordenados en orden creciente de presupuesto, y tantas columnas como el importe del fondo. $C[i,j]$ representa el número máximo de proyectos (entre los $i$ primeros tipos de proyecto) que se pueden financiar con un importe $j$.
2) La relación recursiva entre casos y subcasos es la siguiente:

$$C(i,j) = \begin{cases} -\infty & si\ j < 0 \\ 0 & si\ j = 0 \\ \lfloor j/p_i \rfloor & si\ i = 1\ y\ j \leq p_i r_i \\ r_i & si\ i = 1\ y\ j > p_i r_i \\ \max\limits_{0 \leq k \leq r_i} \{k + C(i-1, j - k * p_i)\} & \text{en otro caso} \end{cases}$$

donde $p_i$ es el presupuesto del proyecto $i$, $p_1 \leq p_2 \leq ... \leq p_n$, y $r_i$ es el número de regiones en las que se puede realizar el proyecto.

# Solución ejercicio 8. Programación dinámica (cont.)

```
fun ong(p[1..n],r[1..n],Fondo)
   crear C[1..n,0..Fondo]
   desde pr ← 1 hasta n hacer
      C[pr,0] ← 0
      desde impte ← 1 hasta Fondo hacer
         si pr=1 ∧ impte ≤ p[1]*r[1] entonces C[pr,impte] ← ⌊impte/p[1]⌋
         si no si pr=1 entonces C[pr,impte] ← r[1]
         si no
            C[pr,impte] ← 0
            desde k ← 0 hasta mín(r[pr],⌊impte/p[pr]⌋) hacer
               si C[pr,impte] < k+C[pr-1,impte-k*p[pr]] entonces
                  C[pr,impte] ← k+C[pr-1,impte-k*p[pr]]
               fin si
            fin desde
         fin si
      fin desde
   fin desde
   devolver C[n,Fondo]
fin fun
```