Programación dinámica

January 25, 2011

Programación dinámica y Divide y Vencerás:

- El método de divide y vencerás consiste en un refinamiento progresivo. Planteamos la resolución de un caso a partir de subcasos del mismo problema pero de menor tamaño.
- La programación dinámica permite resolver problemas típicamente recursivos de manera eficiente de forma ascendente: resolvemos primero los problemas de menor tamaño y, a partir de ellos, los de tamaño superior.
- Se evitan los solapamientos de cálculos $\rightarrow$ Eficiencia!!!.
- Son iterativos.

Programación dinámica y algoritmos voraces:

- Algoritmos voraces:
  - Búsqueda de la solución a través de un conjunto de etapas.
  - Elección voraz.
- Programación dinámica:
  - Búsqueda de la solución a través de un conjunto de etapas.
  - Resolución de un caso a partir del otros casos de menor tamaño → Principio de Optimalidad.
  - Los resultados intermedios los almacenamos en una tabla → Pro-blemas de complejidad espacial.

Definición de los números de Fibonacci:

Definición

$$F(N) = \begin{cases} 1 & \textit{si N=0 ó N=1} \\ F(N-1) + F(N-2) & \textit{si } N > \textit{1} \end{cases}$$

Problema

Calcula el n-ésimo número de Fibonacci

Estructura de datos intermedios:

$$T[1..N] \rightarrow T[K]\text{: K-ésimo número de Fibonacci}$$

La solución estará en $T[N]$.

Algoritmo

fun $fibonacci(N)$
  si $N \leq 1$ **entonces**
    $Fibonacci \leftarrow 1$
  si no
    $T[0] \leftarrow 1$
    $T[1] \leftarrow 1$
    desde $i \leftarrow 1$ **hasta** n **hacer**
      //En cada etapa construimos el I-ésimo
      //número de Fibonacci
      $T[I] \leftarrow T[I-1] + T[I-2]$
    fin desde
    $fibonacci \leftarrow T[N]$
  fin si
  devolver $fibonacci$
fin fun

La complejidad es $\Theta(N)$.

Propiedad de los coeficientes binomiales:

$$\binom{N}{K} = \binom{N-1}{K-1} + \binom{N-1}{K}$$

$$\text{si } 0 < K < N$$

donde:

$$\binom{N}{0} = 1$$

$$\binom{N}{N} = 1$$

$$\binom{N}{1} = N$$

Solución: Para almacenar los resultados intermedios utilizaremos la matriz: $C[0..N, 0..K]$, donde:

$$C[I, J] = \binom{I}{J}$$

Teniendo en cuenta la dependencia de datos:

$$\begin{tikzcd}
C[I-1, J-1] \arrow[dr] & C[I-1, J] \arrow[d] \\
 & C[I, J]
\end{tikzcd}$$

Un ejemplo de matriz para C[5,4]:

$$\begin{bmatrix}
1 & - & - & - \\
1 & 1 & - & - \\
1 & 2 & 1 & - \\
1 & 3 & x & 1 \\
1 & 4 & x & x
\end{bmatrix}$$

Algoritmo

fun $coeficiente(N, K)$
  desde $I \leftarrow 0$ **hasta** N **hacer**
    $C[I, 0] \leftarrow 1$
  fin desde
  desde $I \leftarrow 1$ **hasta** N **hacer**
    $C[I, 1] \leftarrow I$
  fin desde
  desde $I \leftarrow 2$ **hasta** N **hacer**
    $C[I, I] \leftarrow 1$
  fin desde
  desde $I \leftarrow 3$ **hasta** N **hacer**
    desde $J \leftarrow 2$ **hasta** K **hacer**
      si $J \leq K$ **entonces**
        $C[I, J] \leftarrow C[I-1, J-1] + C[I-1, J]$
      fin si
    fin desde
  fin desde
  devolver $C[N, K]$
fin fun

La complejidad es $\Theta(NK)$.

Problema

Sea un sistema monetario con N *tipos de monedas. Sea* $V[K]$ *el valor del tipo de momeda* K*. Suponemos que tenemos una cantidad indefinida de cada tipo de moneda. Diseñad el algoritmo dinámico que determine el menor número de monedas para alcanzar exactamente una cantidad dada.*

Planteamiento

- La solución depende de la cantidad a satisfacer y de los tipos de monedas que podemos usar.
- Crearemos una matriz para guardar los resultados intermedios: $C[1..N, 0..Cant]$ donde $C[I, J]$: es el mínimo número de monedas de tipo 1 a J necesarios para satisfacer J.
- Rellenaremos la tabla por filas. La primera fila representa un caso base. Podemos considerar la solución de las sucesivas filas como una posible mejora al tener un nuevo tipo de moneda disponible.

Traza:

$N = 3$, $V = (1, 4, 6)$ y $Cant = 8$

	0	1	2	3	4	5	6	7	8
1	0^1	1	2	3	4	5	6	7	8
2	0	1^2	2	3	1^3	2	3	4	2
3	0	1	2	3	1	2	1	2	2

Cuestiones:

1. Para una cantidad 0 no necesitamos monedas $\longrightarrow C[I,0] = 0 \ \forall I$.
2. No podemos usar una moneda de tipo 2 porque desbordaría la cantidad:

$$C[I,J] = C[I-1,J] \text{ si } J < V[I]$$

3. Tenemos dos alternativas:
   - No utilizar el nuevo tipo de moneda disponible $\longrightarrow C[I,J] = C[I-1,J]$.
   - Utilizar el nuevo tipo de moneda $\longrightarrow C[I,J] = 1 + C[I, J - V[I]]$.

   Como queremos optimizar cogemos el mínimo:

$$C[I,J] = minimo\{C[I-1,J], 1 + C[I, J - V[I]]\}$$

- Las soluciones al problema verifican la siguiente relación recursiva:

$$C[I, J] = minimo\{C[I-1, J], 1 + C[I, J - V[I]]\}$$

- Utilizaremos una matriz paralela para almacenar la composición del cambio: $M[1..N, 0..Cant]$ donde $M[I, J]$: composición del mínimo número de monedas de tipo 1 a J necesarios para satisfacer J.

Algoritmo

```
fun cambio(V[1..N], cant)
  desde I ← 1 hasta N hacer
    C[I, 0] ← 0
  fin desde
  desde I ← 1 hasta N hacer
    desde J ← 1 hasta cant hacer
      si I = 1 ∧ J < V[I] entonces
        // No hay solución para este subcaso
        C[I, J] ← ∞
      si no
        si I = 1 entonces
          C[I, J] ← 1 + C[1, J − V[1]]
        si no
          si J < V[I] entonces
            // No podemos insertar una nueva moneda
            C[I, J] ← C[I − 1, J]
          si no
            C[I, J] = minimo{C[I − 1, J], 1 + C[I, J − V[I]]}
          fin si
        fin si
      fin si
    fin desde
  fin desde
  devolver C[N, cant]
fin fun
```

Si tenemos un sistema monetario sin un tipo de moneda con valor unidad puede que el problema no tenga solución:

Traza:

$N = 3$, $V = (3, 10, 20)$ y $Cant = 6$

	0	1	2	3	4	5	6
1	0	∞^1	∞	1	∞^2	∞	2
2	0	∞	∞	1	∞	∞	2
3	0	∞	∞	1	∞	∞	2

Cuestiones:

1. Para este caso no podemos coger ninguna moneda sin provocar desbordamiento.
2. Es preciso cubrir exactamente la cantidad. Por lo tanto, tampoco tiene solución.

Algoritmo

fun $cambio(V[1..N], cant)$
  desde $I \leftarrow 1$ **hasta** N **hacer**
    $C[I, 0] \leftarrow 0$
    $M[I, 0] \leftarrow \emptyset$
  fin desde
  desde $I \leftarrow 1$ **hasta** N **hacer**
    desde $J \leftarrow 1$ **hasta** $cant$ **hacer**
      si $I = 1 \wedge J < V[I]$ **entonces**
        // No hay solución para este subcaso
        $C[I, J] \leftarrow \infty$
        $M[I, J] \leftarrow \emptyset$
      si no
        si $I = 1$ **entonces**
          $C[I, J] \leftarrow 1 + C[1, J - V[1]]$
          si $C[1, J - V[1]] = \infty$ **entonces**
            $M[I, J] \leftarrow \emptyset$
          si no
            $M[I, J] \leftarrow \{1\} \cup M[1, J - V[1]]$
          fin si
        si no
          ...
        fin si
      fin si
    fin desde
  fin desde
  devolver $C[N, cant], M[N, cant]$
fin fun

Algoritmo

fun $cambio(V[1..N], cant)$
  ...
  si $J < V[I]$ **entonces**
    // No podemos insertar una nueva moneda
    $C[I, J] \leftarrow C[I-1, J]$
    $M[I, J] \leftarrow M[I-1, J]$
  si no
    $C[I, J] = minimo\{C[I-1, J], 1 + C[I, J - V[I]]\}$
    si $C[I-1, J] > 1 + C[I, J - V[I]]$ **entonces**
      $M[I, J] \leftarrow \{I\} \cup M[1, J - V[I]]$
    si no
      $M[I, J] \leftarrow M[I-1, J]$
    fin si
  fin si
fin fun

Ejercicio

Considera como cambia el algoritmo anterior si tenemos un número finito de copias de cada moneda.

Solución:

Nueva relación recursiva:

$$C[I,J] = minimo\{C[I-1,J], q + C[I-1, J - q * V[I]]\}$$
$$\forall q : 1..cp[I]$$

donde $cp[I]$ es el número de copias de la moneda I.

Problema

Supongamos que existen N bancos en los que podemos invertir. Disponemos de una cantidad inicial, $Cant$, para la inversión. Cada banco nos ofrece interés según el capital invertido. Este interés viene dado por la matriz:

$B[1..N, 0..Cant] \rightarrow B[p, q]$: *"interés" del banco p al invertir q.*

Diseñad un algoritmo dinámico que encuentre la inversión óptima.

Planteamiento:

- La inversión óptima depende de los bancos disponibles y del capital.
- Creamos una matriz $I[1..N, 0..Cant]$ donde:

  $I[s,t]$: beneficio de invertir la cantidad t en los bancos $1..s$.
- La solución está en la posición $I[N, Cant]$.
- Si solamente de disponemos de un banco: $I[1,t] = B[1,t]\ \forall t$.
- Obviamente: $I[s,0] = 0\ \forall s$.
- Además, en la matriz $Comp[1..N, 0..Cant]$ almacenaremos la composición de la inversión óptima.

Cálculo de $I[s,t]$:

- Teniendo disponible el banco s, podemos no invertir en él. Entonces: $I[s,t] = I[s-1,t]$.
- Teniendo disponible el banco s, podemos invertir una cantidad z. Entonces: $I[s,t] = I[s-1,t-z] + B[s,z]$.
- Elegiremos la mejor combinación:

$$I[s,t] = maximo\{I[s-1,t], I[s-1,t-z] + B[s,z]\ \} \\ \forall z : 1..t.$$

Algoritmo

fun $funcionbancos(B[1..N, 0..Cant], Cant)$
  desde $K \leftarrow 1$ **hasta** N **hacer**
    $I[K, 0] \leftarrow 0$
    $Comp[K, 0] \leftarrow \emptyset$
  fin desde
  desde $J \leftarrow 1$ **hasta** $Cant$ **hacer**
    $I[1, J] \leftarrow B[1, J]$
    $Comp[1, J] \leftarrow \{(1, J)\}$
  fin desde
  desde $banc \leftarrow 2$ **hasta** N **hacer**
    desde $cantidad \leftarrow 1$ **hasta** $Cant$ **hacer**
      //No invertir en banc
      $beneficio \leftarrow I[banc - 1, cantidad]$
      $Comp[banc, cantidad] \leftarrow Comp[banc - 1, cantidad]$
      desde $z \leftarrow 1$ **hasta** $cantidad$ **hacer**
        si $beneficio < I[banc - 1, cantidad - z] + B[banc, z]$ **entonces**
          $beneficio \leftarrow I[banc - 1, cantidad - z] + B[banc, z]$
          $Comp[banc, cantidad] \leftarrow \{(banc, z)\} \cup Comp[banc - 1, Cant - z]$
        fin si
      fin desde
      $I[banc, cantidad] \leftarrow beneficio$
    fin desde
  fin desde
  devolver $I[N, Cant], Comp[N, Cant]$
fin fun

Traza

Sea B:

$$\begin{bmatrix} 1 & 1 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 & 3 \\ 4 & 4 & 4 & 4 & 4 \end{bmatrix}$$

- $I[1,t] = B[1,t]$
- $I[s,0] = 0$

- Cálculo de $I[s,t]$:
  - Cálculo de $I[2,1]$. Alternativas:
    - $z = 0 \Rightarrow I[2,1] = I[1,1] = 1.$
    - $z = 1 \Rightarrow I[2,1] = B[2,1] = 2 \Rightarrow I[2,1] = 2.$
  - Cálculo de $I[2,2]$. Alternativas:
    - $z = 0 \Rightarrow I[2,2] = I[1,2] = 1.$
    - $z = 1 \Rightarrow I[2,2] = B[2,1] + I[1,1] = 3.$
    - $z = 2 \Rightarrow I[2,2] = B[2,2] = 2 \Rightarrow I[2,2] = 3.$
  - Cálculo de $I[2,3]$. Alternativas:
    - $z = 0 \Rightarrow I[2,3] = I[1,3] = 2.$
    - $z = 1 \Rightarrow I[2,3] = B[2,1] + I[1,2] = 3.$
    - $z = 2 \Rightarrow I[2,3] = B[2,2] + I[1,1] = 3.$
    - $z = 3 \Rightarrow I[2,3] = B[2,3] = 2 \Rightarrow I[2,3] = 3.$

Problema

En un río hay N *embarcaderos. Situado en un embarcadero solamente se puede ir a los embarcaderos siguientes (no se puede ir río arriba). Existe una tabla de tasas para ir de un embarcadero a otro. Es posible ir de un embarcadero a otro haciendo escala en algún embarcadero intermedio. No existe coste adicional por cambiar de bote. Queremos encontrar la ruta del embarcadero* 1 *al embarcadero* N *de menor coste.*

Planteamiento:

- Con una matriz podemos tratar las tasas: $T[1..N, 1..N]$ donde $T[I, J]$ es la tasa para ir de I a J con $I < J$.
- Los resultados intermedios pueden ser almacenados en un vector:

  $$C[1..N] \rightarrow C[K]\text{: coste mínimo para ir de 1 a } K.$$

- Cálculo de $C[K]$:
  - La ruta óptima desde 1 hasta K está contenida en el conjunto:

    $$\{C[q] + T[q, K]\} \ \forall q : 1..K - 1$$

  - Supongamos que la ruta óptima, R, no estuviera contenida en este conjunto.
    - Existiría un último embarcadero visitado, $\widehat{q}$, antes de ir directamente a N (pudiendo ser el 1).
    - El camino de 1 a $\widehat{q}$ es óptimo. Si no lo fuera R tampoco sería óptimo.
    - R está contenido en el conjunto considerado.
  - Por lo tanto, cogemos el valor que minimiza el coste:

    $$C[K] = minimo\{C[q] + T[q, K]\} \ \forall q : 1..K - 1.$$

Algoritmo

fun $funcionbajar-rio(T[1..N,1..N])$
  $C[1] \leftarrow 0$
  $R[1] \leftarrow \emptyset$
  $C[2] \leftarrow T[1,2]$
  $R[2] \leftarrow (1,2)$
  desde $K \leftarrow 3$ **hasta** N **hacer**
    $min \leftarrow \infty$
    desde $q \leftarrow 1$ **hasta** $K-1$ **hacer**
      si $C[q]+T[q,K] < min$ **entonces**
        $minimo \leftarrow C[q]+T[q,K]$
        $ultimo-camino \leftarrow (q,K)$
        $ultimo-emb \leftarrow q$
      fin si
    fin desde
    $C[K] \leftarrow min$
    $R[K] \leftarrow R[ultimo-emb] \cup \{ultimo-camino\}$
  fin desde
  devolver $retornoC[N], R[N]$
fin fun

Traza: Sea T:

$$\begin{bmatrix} 0 & 10 & 15 & 15 & 20 \\ - & 0 & 15 & 20 & 25 \\ - & - & 0 & 5 & 1 \\ - & - & - & 0 & 10 \\ - & - & - & - & 0 \end{bmatrix}$$

- $C[1] = 0$
  $R[1] = \emptyset$
- $C[2] = 10$
  $R[2] = \{(1, 2)\}$
- $C[3]$:
  - $C[1] + T[1, 3] = 15$
  - $C[2] + T[2, 3] = 25 \Rightarrow C[3] = 15 \; R[3] = \{(1, 3)\}$
- $C[4]$:
  - $C[1] + T[1, 4] = 15$
  - $C[2] + T[2, 4] = 30$
  - $C[3] + T[3, 4] = 20 \Rightarrow C[4] = 15 \; R[4] = \{(1, 4)\}$
- $C[5]$:
  - $C[1] + T[1, 5] = 20$
  - $C[2] + T[2, 5] = 35$
  - $C[3] + T[3, 5] = 16$
  - $C[4] + T[4, 5] = 25 \Rightarrow C[5] = 16 \; R[5] = \{(1, 3), (3, 5)\}$

Problema

Dada una mochila (con capacidad en peso conocida) y N elementos no fraccionables con un peso y valor conocido. Determina el valor de la mochila óptima.

Planteamiento:

- La solución depende de la capacidad de la mochila y de los elementos disponibles.
- Utilizaremos una matriz para almacenar los cálculos intermedios:

  $V[1..N, 0..P_{max}] \rightarrow V[I, P]$: beneficio óptimo de la mochila utilizando los I primeros objetos con la capacidad P de la mochila.
- El resultado está en la posición: $V[N, P_{max}]$
- $V[I, 0] = 0 \; \forall I : 1..N$
- Cálculo de $V[1, P]$:
  - Si no cabe el elemento 1 $(P < w[1])$ entonces: $V[1, P] = 0$.
  - En caso contrario: $V[1, P] = v[1]$.
- Cálculo de $V[I, P]$:
  - Si el elemento I no cabe: $V[I, P] = V[I - 1, P]$.
  - En caso contrario vemos qué alternativa es mejor:
    - No cogerlo: $V[I, P] = V[I - 1, P]$.
    - Cogerlo: $v[I] + V[I - 1, P - w[I]]$
  - Cogemos la opción óptima:

$$V[I, P] = maximo\{V[I-1, P], v[I] + \underbrace{V[I-1, P-w[I]]}_{BeneficioRestoMochila}\}$$

Algoritmo

fun $mochila(w[1..N], v[1..N], Pmax)$
  desde $I \leftarrow 1$ **hasta** N **hacer**
    $V[I, 0] \leftarrow 0$
  fin desde
  desde $J \leftarrow 1$ **hasta** N **hacer**
    desde $P \leftarrow 0$ **hasta** $Pmax$ **hacer**
      si $J = 1 \wedge p < w[1]$ **entonces**
        $V[J, P] \leftarrow 0$
      si no
        si $J = 1$ **entonces**
          $V[J, P] \leftarrow v[1]$
        si no
          si $p < w[J]$ **entonces**
            $V[J, P] \leftarrow V[J-1, P]$
          si no
            $V[J, P] \leftarrow max\{V[J-1, P], V[J-1, P-w[J]] + v[J]\}$
          fin si
        fin si
      fin si
    fin desde
  fin desde
  devolver $V[N, Pmax]$
fin fun

Traza:

N=4, w=(3,4,5,10), v=(1,10,15,20)

$P_{max} = 10$

	0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	1	1	1	1	1	1	1	1
2	0	0	0	1	10	10	10	10	11	11	11
3	0	0	0	1	10	15	15	15	16	25	25
4	0	0	0	1	10	15	15	15	16	25	25

Mochila con N tipos de objetos

- Utilizamos la misma estructura de cálculos intermedios: $V[1..N, 0..P_{max}]$
- $V[I, 0] = 0 \; \forall I$
- Cálculo de la primera fila:

$$V[1, P] \leftarrow P div w[1] * v[1]$$

- Si el objeto de tipo I no cabe:

$$V[J, P] \leftarrow V[J-1, P]$$

- En caso contrario:

$$V[J, P] \leftarrow max V[J-1, P], V[J, P - w[J]] + v[J]$$

- Si consideramos un número de copias finito, $c[1..N]$:

$$V[J, P] = max\{V[J-1, P], V[J-1, P - w[J] * q] + v[J] * q\}$$
$$\forall q : 1..c[J] \text{ si } q * w[P] \leq P$$

Problema

Sea $G = (V, A)$ *un grafo dirigido con matriz de adyacencia* $L[1..n, 1, , n]$. *Determina la longitud del camino mínimo para cada par de vértices.*

Planteamiento:

- Formulamos la solución a través de una serie de etapas: en cada una de ellas calculamos la distancia mínima para una pareja de vértices utilizando como nodos intermedios un subconjunto de V.
- Estructura de cálculos intermedios: $D_Q[1..n, 1..n] \rightarrow D_Q[i,j]$: distancia mínima entre i y j utilizando como nodos intermedios los del conjunto Q.
- Inicialmente: $D_\emptyset[i,j] \leftarrow L[i,j]$.
- En cada etapa añadimos un nodo al conjunto de vértices, Q.
- Al final de la ejecución: $Q = V$.
- Relación recursiva:
  $D_{Q\cup\{k\}} = minimo\{D_Q[i,j], D_Q[i,k] + D_Q[k,j]\}$.

Algoritmo

fun $floyd(L[1..N, 1..N])$
  desde $I \leftarrow 1$ **hasta** N **hacer**
    desde $J \leftarrow 1$ **hasta** N **hacer**
      $D[I, J] \leftarrow L[I, J]$
    fin desde
  fin desde
  desde $K \leftarrow 1$ **hasta** N **hacer**
    desde $I \leftarrow 1$ **hasta** N **hacer**
      desde $J \leftarrow 1$ **hasta** N **hacer**
        $D[I, J] \leftarrow min\{D[I, J], D[I, K] + D[K, J]\}$
      fin desde
    fin desde
  fin desde
  devolver $retornoD[1..N]$
fin fun

Problema

Sea $G = (V, A)$ un grafo dirigido determina si hay un camino entre dos vértices cualesquiera.

Se trata de un caso particular del algoritmo de Floyd:

- Estructura de cálculos intermedios: $D_Q[1..n, 1..n] \rightarrow D_Q[i, j]$: *cierto* si existe camino entre i y j utilizando como nodos intermedios los del conjunto Q.
- Relación recursiva: $D_{Q \cup \{k\}} = D_Q[i, j] \vee D_Q[i, k] \wedge D_Q[k, j]$.

Problema

Dadas n *matrices,* $M_1 * M_2 * ... * M_n$*, determina el número mínimo de multiplicaciones necesarias para calcular su producto.*

- Dadas dos matices A, $p \times q$, y B, $q \times r$, el cálculo de $A * B$ necesita $p * q * r$ operaciones escalares.
- El producto de dos matrices es asociativo: podemos realizar las multiplicaciones en el orden que queramos.
- La elección del orden de los productos influye en el número de multiplicaciones escalares $\longrightarrow$ Influye en la eficiencia.
- Estructura de cálculos intermedios: $M[1..n, 1..n] \longrightarrow M[i, j]$: número mínimo de operaciones escalares necesario para realizar el producto $M_i * M_{i+1} * \ldots * M_j$.
- La solución está en $M[1, n]$.
- Supongamos que la forma óptima de multiplicar $M_i * M_{i+1} * \ldots * M_j$ sea:

$$\underbrace{(M_i * M_{i+1} * \ldots * M_k)}_{M_a} * \underbrace{(M_{k+1} * \ldots * M_j)}_{M_b}$$

  Entonces: $M[i, j] = M[i, k] + M[k + 1, j] +$ *operaciones de multiplicación de* M_a *y* M_b.
- Con $n + 1$ datos podemos almacenar las dimensiones de las n matrices. Sea $D[0..n]$ donde $D[i - 1]$ y $D[i]$ contiene las dimensiones de la matriz i. Entonces:

$$M[i, j] = M[i, k] + M[k + 1, j] + D[i - 1] * D[k] * D[j]$$

- Para cada subproducto de matrices buscamos la descomposición óptima:

$$M[i, j] = minimo\{M[i, k] + M[k + 1, j] + D[i - 1] * D[k] * D[j]\} \ \forall k : i..j - 1.$$

- $M[i, i] = 0 \ \forall i$.

Algoritmo

fun $matriz(D[0..N])$
  // D: array de dimensiones
  // N: número de matrices
  desde $I \leftarrow 1$ **hasta** N **hacer**
    $M[I, I] \leftarrow 0$
  fin desde
  desde $Diag \leftarrow 1$ **hasta** $N - 1$ **hacer**
    desde $I \leftarrow 1$ **hasta** $N - Diag$ **hacer**
      $M[I, I + Diag] \leftarrow calculo - minimo(D, M, I, I + Diag)$
    fin desde
  fin desde
  devolver $retornoM[1..N]$
fin fun
fun $calculo - minimo(D[0..N], M[1..N, 1..N], I, J)$
  $min \leftarrow \infty$
  desde $q \leftarrow I$ **hasta** $J - 1$ **hacer**
    si $M[I, q] + M[q + 1, J] + D[I - 1] * D[q] * D[J] < min$ **entonces**
      $min \leftarrow M[I, q] + M[q + 1, J] + D[I - 1] * D[q] * D[J]$
    fin si
  fin desde
  devolver $retornomin$
fin fun

Problema

Combinando las metodologías de Divide y vencerás y Programación dinámica, diseñad un algoritmo que calcule los números de Fibonacci eficientemente (sin repetición de cálculos).

- Elementos de Divide y vencerás: Planteamiento descendente.
- Elementos de Programación dinámica: Planteamiento ascendente, estructura de cálculos intermedios.
- Combinaremos el planteamiento descendente con una estructura de cálculos intermedios:
  $M[1..N] \rightarrow M[K]$: K-ésimo número de Fibonacci generado.
- Según se generen los valores se almacenan en M. Solamente si no han sido genereados antes se realiza la llamada recursiva.
- Inicialmente $M[K] = 0 \; \forall K$, pues no se corresponde con ningún resultado.

Algoritmo

fun $fibonac(N, M[1..N])$
  si $N = 1 \vee N = 0$ **entonces**
    $fib \leftarrow 1$
    $M[N] \leftarrow 1$
  si no
    si $M[N-1] = 0$ **entonces**
      //fibonac(N-1) no está calculado
      $fib1 \leftarrow fibonac(N-1)$
    si no
      $fib1 \leftarrow M[N-1]$
    fin si
    si $M[N-2] = 0$ **entonces**
      //fibonac(N-2) no está calculado
      $fib2 \leftarrow fibonac(N-2)$
    si no
      $fib2 \leftarrow M[N-2]$
    fin si
    $fib \leftarrow fib1 + fib2$
    $M[N] \leftarrow fib1 + fib2$
  fin si
  devolver fib
fin fun

Problema

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 disponibles y el número de copias de cada tipo de moneda. Diseña un algoritmo dinámico que resuelva el problema.

Algoritmo

fun $cambio(V[1..N], copias[1..N], cant)$
  desde $I \leftarrow 1$ **hasta** N **hacer**
    $C[I, 0] \leftarrow 0$
  fin desde
  desde $I \leftarrow 1$ **hasta** N **hacer**
    desde $J \leftarrow 1$ **hasta** $Cant$ **hacer**
      si $I = 1 \wedge J < V[I]$ **entonces**
        $C[I, J] \leftarrow \infty$ *// No hay solución para este subcaso*
      si no
        si $I = 1$ **entonces**
          si $(JmodV[I] = 0) \wedge (\lfloor J/V[I] \rfloor \leq copias[I])$ **entonces**
            $C[I, J] \leftarrow \lfloor J/V[I] \rfloor$ *//tiene solución*
          si no
            $C[I, J] \leftarrow \infty$ *//no tiene solución*
          fin si
        si no
          $min \leftarrow \infty$
          desde $K \leftarrow 0$ **hasta** $minimo\{\lfloor J/V[I] \rfloor, copias[I]\}$ **hacer**
            si $min > K + C[I-1, J-K*V[I]]$ **entonces**
              $min \leftarrow K + C[I-1, J-K*V[I]]$
            fin si
          fin desde
          $C[I, J] \leftarrow min$
        fin si
      fin si
    fin desde
  fin desde
  devolver $C[N, cant]$
fin fun

Problema

En una montaña hay N puertos. El puerto N está en la cima mientras que el puerto 1 está en la base. Para ascenderla disponemos de una cantidad de oxigeno inicial. Podemos hacer escala en cada uno de los puertos intermedios. Para ascender de un puerto I a otro J, $I < J$, necesitaremos consumir una cierta cantidad de oxigeno conocida. Podremos ascender la montaña y, si no es así, hasta qué puerto podremos ascender?.

Planteamiento:

$Ox[1..N, 1..N] \rightarrow Ox[I, J]$: cantidad de oxigeno necesario para ir de I a J.

Estructura de datos intermedia:

$R[1..N] \rightarrow R[K]$: valor óptimo de las reservas de oxígeno al llegar al puerto K.

Relación recursiva:

$R[K] = max\{R[J] - Ox[J, K]\}$ con $J : 1..K - 1$

Solución:

Mayor K cuyo $R[K] \neq -\infty$

Algoritmo

fun $escalada(Ox[1..N, 1..N], Oxigeno - inicial)$
  $R[1] \leftarrow Oxigeno - inicial$
  desde $K \leftarrow 2$ **hasta** N **hacer**
    $Max \leftarrow -\infty$
    desde $q \leftarrow 1$ **hasta** $K - 1$ **hacer**
      si $Max < R[q] - Ox[q, K] \wedge 0 < R[q] - Ox[q, K]$ **entonces**
        $Max \leftarrow R[q] - Ox[q, K]$
      fin si
    fin desde
    $R[K] \leftarrow Max$
  fin desde
  $K \leftarrow N$
  mientras $R[K] = -\infty$ **hacer**
    $K \leftarrow K - 1$
  fin mientras
  devolver K
fin fun

Problema

Sea un archipiélago de n islas. Existen varios puentes que unen algunas parejas de islas. Cada puente posee una anchura. Puede ocurrir que alguno de estos puentes sea de dirección única. La anchura de un camino es el valor mínimo de los puentes que hay que atravesar. Diseña un algoritmo que para cada pareja de islas determine la anchura máxima de los caminos que las unen.

Planteamiento:

- Representamos el mapa mediante un grafo dirigido no valorado.
- Convenciones: la anchura del los puentes que no existan se representará con $-\infty$ y la auto-arista con ∞. En ambos casos los puentes no existen. Esta suposición simplifica el código.
- Planteamos una estructura de cálculos intermedios similar a la del algoritmo de Floyd:

  $A_k(i,j)$: anchura máxima de los caminos que van de i a j pasando por las k primeras filas.
- Si se pasa por la isla q en el trayecto de i a j se verifica $A_s(i,j) = min\{A_s(i,q), A_s(q,j)\}$.
- Para la k-ésima isla consideramos la mejora que puede proporcionar:

$$A_k(i,j) = max\{A_{k-1}(i,j), min\{A_{k-1}(i,k), A_{k-1}(k,j)\}$$

Algoritmo

fun $puentes(L[1..N, 1..N])$
  $camino[1..N, 1..N] \leftarrow [0]$
  desde $I \leftarrow 1$ **hasta** N **hacer**
    desde $J \leftarrow 1$ **hasta** N **hacer**
      $A[I, J] \leftarrow L[I, J]$
    fin desde
  fin desde
  desde $K \leftarrow 1$ **hasta** N **hacer**
    desde $I \leftarrow 1$ **hasta** N **hacer**
      desde $J \leftarrow 1$ **hasta** N **hacer**
        $aux \leftarrow min\{A[I, K], A[K, J]\}$
        si $aux > A[I, J]$ **entonces**
          $A[I, J] \leftarrow aux$
          $camino[I, J] \leftarrow k$
        fin si
      fin desde
    fin desde
  fin desde
  devolver $retornoD[1..N], camino[1..N, 1..N]$
fin fun

Nota:

- $camino[I, J] = 0$ indica que el camino de anchura máxima entre I y J es directo.
- $camino[I, J] = K$ indica que K es la última isla por la que el camino el camino de anchura máxima entre I y J pasa. Mirando en $camino[I, K]$ podemos encontrar otras islas por las que pasa el camino entre I y K.

Problema

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, diseña un algoritmo dinámico que determine el valor de los productos elegidos en el canje.

Algoritmo

fun $cambio(V[1..N], copias[1..N], cant)$
  desde $I \leftarrow 1$ **hasta** N **hacer**
    $C[I, 0] \leftarrow 0$
  fin desde
  desde $I \leftarrow 1$ **hasta** N **hacer**
    desde $J \leftarrow 1$ **hasta** $Cant$ **hacer**
      si $I = 1 \wedge J < V[I]$ **entonces**
        $C[I, J] \leftarrow V[I]$
      si no
        si $I = 1$ **entonces**
          si $sig - entero(J, V[I]) \leq copias[I]$ **entonces**
            si $JmodV[I] = 0$ **entonces**
              $C[I, J] \leftarrow J$
            si no
              $C[I, J] \leftarrow V[I](Jdiv[I]) + V[I]$
            fin si
          si no
            $C[I, J] \leftarrow \infty$
          fin si
        si no
          ...
        fin si
      fin si
    fin desde
  fin desde
  devolver $C[N, cant]$
fin fun

Algoritmo

fun $cambio(V[1..N], copias[1..N], cant)$
  desde $I \leftarrow 1$ **hasta** N **hacer**
    $C[I, 0] \leftarrow 0$
  fin desde
  desde $I \leftarrow 1$ **hasta** N **hacer**
    desde $J \leftarrow 1$ **hasta** $Cant$ **hacer**
      ...
      $min \leftarrow \infty$
      desde $K \leftarrow 0$ **hasta** $minimo\{sig - entero(J, V[I]), copias[I]\}$ **hacer**
        si $min > K * V[I] + C[I-1, J - K * V[I]] \wedge K * V[I] + C[I-1, J - K * V[I]] > J$
        entonces
          $min \leftarrow K * V[I] + C[I-1, J - K * V[I]]$
        fin si
      fin desde
      $C[I, J] \leftarrow min$
    fin desde
  fin desde
  devolver $C[N, cant]$
fin fun
fun $sig - entero(X, Y)$
  // Devuelve el menor entero, Z, tal que $X/Y \leq Z$
  si $X mod Y = 0$ **entonces**
    $sig - entero \leftarrow X div Y$
  si no
    $sig - entero \leftarrow (X div Y) + 1$
  fin si
  devolver $sig - entero$
fin fun