

## Capítulo 4. Programación Dinámica

### Introducción

Una forma razonable y comúnmente empleada de resolver un problema es definir o caracterizar su solución en términos de las soluciones de subproblemas del mismo. Esta idea, cuando se emplea recursivamente, proporciona métodos eficientes de solución para problemas en los que los subproblemas son versiones mas pequeñas del problema original.

Una técnica de diseño de algoritmos, que sigue esta idea, es la conocida como divide y vencerás que hemos estudiado en temas anteriores. Como allí se explicó, consiste en descomponer cada problema en un numero dado de subproblemas, resolver cada uno de ellos (quizás empleando el mismo método) y combinar estas soluciones parciales para obtener una solución global. Desde el punto de vista de la complejidad de los algoritmos que se obtienen por este procedimiento, lo mejor es que todos los subproblemas tengan dimensión similar y que la suma de estas sea lo menor posible. No obstante, en muchos casos, dado un problema de tamaño  $n$  solo puede obtenerse una caracterización efectiva de su solución en términos de la solución de los subproblemas de tamaño  $n-1$  ( $n$  en total) lo que puede aplicarse recursivamente. En estos casos, la técnica conocida como Programación Dinámica (PD) proporciona algoritmos bastante eficientes. Algunos de estos problemas ya los conocemos. Es el caso, por ejemplo, del Problema de la Mochila o del Problema del Camino Mínimo.

En efecto, para el primero, su solución puede entenderse como el resultado de una sucesión de decisiones. Tenemos que decidir los valores de  $x_i$ ,  $1 \leq i \leq n$ . Así, primero podríamos tomar una decisión sobre  $x_1$ , luego sobre  $x_2$ , etc. Una sucesión optimal de decisiones, verificando las restricciones del problema, será la que maximice la correspondiente función objetivo. En el caso del camino mínimo, una forma de encontrar ese camino mínimo desde un vértice  $i$  a otro  $j$  en un grafo dirigido  $G$ , es decidiendo que vértice debe ser el segundo, cual el tercero, cual el cuarto, etc. hasta que alcanzáramos el vértice  $j$ . Una sucesión optimal de decisiones proporcionaría entonces el camino de longitud mínima que buscamos.

El nombre Programación Dinámica y la metodología que subyace en él lo ha tomado la Teoría de Algoritmos de la Teoría de Control, donde la Programación Dinámica es una técnica para obtener la política optima en problemas de control con  $n$  etapas, caracterizando esta en términos de la política optima para un problema similar, pero con  $n-1$  etapas. Por este motivo comenzamos nuestra exposición analizando el método de la Programación Dinámica en su contexto original, para pasar después a su formulación dentro de la Teoría de Algoritmos.

### El Lugar de la Programación Dinámica

El problema mas importante en la Teoría de Control es el del control optimal. Un típico proceso supone en este contexto cuatro tipos de variables,

1. Variables independientes, que son las variables manipulables para el control del proceso.
2. Variables dependientes, que sirven para medir y describir el estado del proceso en cualquier instante de tiempo.

3. Variables producto, que se usan para indicar y medir la calidad de la tarea que realiza el sistema de control, y
4. Ruidos, que son variables incontrolables que dependen del ambiente en el que el sistema lleve a cabo su operación.

El problema general del control optimal de un proceso consiste en determinar la forma en que las variables producto pueden mantenerse en valores optimales, independientemente de las fluctuaciones que los ruidos, o los cambios de parámetros, puedan causar. Las variables producto se usan para describir el índice de eficacia del control del sistema. Así el problema del control optimo supone la optimización (maximización o minimización) de ese índice de eficacia.

Generalmente, un sistema físico de n-esimo orden puede describirse en cualquier instante de tiempo  $t$  por medio de un conjunto finito de cantidades  $(x_1(t), x_2(t), \dots, x_n(t))$ . Estas cantidades se conocen como las variables de estado del sistema, y constituyen las componentes del vector de estado del sistema:  $x(t)$ . La relación entre los cambios del parámetro tiempo y las variables de estado del sistema, se establece mediante una sencilla y útil hipótesis como es que la derivada del vector de estado,  $dx/dt$ , solo dependa del estado del sistema en curso y no de su historia anterior. Esta hipótesis básica lleva a una caracterización matemática del proceso por medio de una ecuación diferencial vectorial,

$$dx(t)/dt = f[x(t), m(t), t]$$

con la condición inicial  $x(0) = x_0$ . En esa ecuación  $m(t)$  nota el vector de control, y  $f$  es una función vectorial de las variables de estado, las señales de control, el tiempo y, posiblemente, las variables ambientales o ruidos externos. En cada instante, el vector de control  $m$  debe satisfacer la condición,

$$g(m) \leq 0$$

que refleja las restricciones que el control del sistema tenga. La función  $g$  es una función conocida de la señal de control. Entonces, el problema del diseño de un control optimal puede plantearse en los siguientes términos: Dado un proceso que debemos controlar, determinar la ley de control, o sucesión, tal que el conjunto de índices de eficacia, previamente fijado, sea optimo.

A veces, las variables de estado de un sistema son todas accesibles para medirlas y observarlas. Para los sistemas lineales que tienen esta característica, aun en presencia de ruido, puede tratarse directamente de determinar la ley de control optimal como una función de las variables de estado. Sin embargo, es bastante frecuente que las variables de estado no sean todas accesibles para medirlas y observarlas. Entonces, la ley de control optimal se determina como una función de las mejores estimaciones de las variables de estado, que se calculan a partir de las señales de salida del sistema que se hayan medido. Por tanto, en el caso mas general, tendremos que considerar la resolución de problemas de control optimal y de estimación optimal.

La Teoría de Control parte de la caracterización de un sistema mediante sus variables de estado y del diseño del sistema por medio de técnicas basadas en su representación como espacio de estados. En una formulación general, el diseño del control optimal se ve generalmente como un problema variacional. Pero hay muchos métodos variacionales para optimizar un funcional sobre un espacio de funciones. El rango de tales métodos va desde los métodos clásicos del calculo de variaciones, a las técnicas numéricas de aproximaciones sucesivas de modelos experimentales. Entre los métodos mas frecuentemente usados para el diseño de sistemas de control están,

- 1.- El Calculo de Variaciones,
- 2.- El Principio de Máximo de Pontryaguin, y
- 3.- La Programación Dinámica.

En todos los casos, el propósito es encontrar la ley de control óptima tal que dado el funcional del índice de eficacia del sistema, lo optimice. Pero es muy significativo que los tres métodos tienen en común el uso de principios variacionales: Cada uno de estos tres métodos está íntimamente ligado a alguna formulación bien conocida de la Mecánica clásica. Los primeros, con la ecuación de Euler-Lagrange, los segundos con el Principio de Hamilton y los terceros con la teoría de Hamilton-Jacobi. Particularmente, el Principio de Máximo de Pontryaguin emplea procedimientos, más o menos directos, del cálculo de variaciones, mientras que la Programación Dinámica (PD), aun basándose en principios variacionales, usa relaciones de recurrencia.

La PD la desarrolló Richard E. Bellman y es una técnica simple, pero poderosa, que se ha demostrado muy útil para la resolución de problemas de decisión multietápicos. La idea fundamental que subyace en el método de la PD es el principio del invariante encajado, según el cual un problema difícil de resolver, está encajado en una clase de problemas más simples de resolver, lo que permite encontrar una solución al problema original.

Naturalmente, el objetivo de este tema es el estudio del uso de la PD para el diseño de algoritmos que resuelvan eficientemente problemas por lo que, una vez que la hemos colocado en un lugar de referencias exacto, a continuación nos centramos en ella. Comenzaremos por introducir los procesos de decisión multietápicos estudiándolos desde el punto de vista del diseño de su control óptimo para, después, presentar el Principio de Optimalidad de Bellman y emplearlo para el diseño de algoritmos que, basados en él, resuelven algunos problemas notables en las Ciencias de la Computación.

### **Procesos de Decisión Multietápicos: Enfoque Funcional**

Uno de los enfoques posibles para el diseño de sistemas con control óptimo es la búsqueda sistemática de una estrategia óptima de control en cierto espacio multidimensional de control. En este sentido, el problema de diseñar el sistema óptimo puede entenderse como un problema de decisión multietápica. En la práctica son abundantes los ejemplos de procesos de decisión multietápica. Quizás el más inmediato de todos ellos sea el del juego de naipes. Pero, en el ambiente financiero, los problemas de inversión y de determinación de políticas para hacer seguros, también, son buenos ejemplos de problemas de decisión multietápicos.

Por ejemplo, supongamos que se dispone de una cantidad inicial de dinero  $x$  para invertir en un negocio de alquiler de computadores. Este dinero se usará para comprar computadores para tratamiento de textos (CTT) y para conexiones a la red (CCR). Supongamos que se gastan  $y$  pesetas en la compra de CTT y, el resto, se dedica a la compra de CCR. Lo que producen anualmente los CTT es una función de la cantidad inicial invertida,  $y$ , e igual a  $g(y)$  el primer año. Lo que producen anualmente los CCR es función del capital restante,  $x-y$ , e igual a  $h(x-y)$  el primer año. La política de la empresa determina que al final de cada año, todos los computadores usados de uno y otro tipo se cambien. El cambio de todos los CTT es una función de la cantidad total de dinero invertida en CTT e igual a  $ay$  el primer año,  $0 < a < 1$ . Por su parte el valor del cambio de los CCR es una función igual a  $b(x-y)$ ,  $0 < b < 1$ . Los expertos que dirigen la empresa tienen que tomar una sucesión de decisiones óptimas de modo que el beneficio total a lo largo de un número  $N$  de años sea máximo. Para simplificar el problema, supondremos que los beneficios anuales no se reinvierten en el negocio.

Los problemas de decisión multietápica pueden resolverse muy bien por medio del enfoque funcional, según el cual el problema de maximización original se convierte en un problema de determinar la solución de una ecuación funcional que se obtiene como sigue. El beneficio durante el primer año es,

$$Y_1(x,y) = g(y) + h(x-y)$$

Así, para un periodo de un año, la máxima recompensa la da,

$$f_1(x) = \text{Max}_{0 \leq y \leq x} \{Y_1(x,y)\} = \text{Max}_{0 \leq y \leq x} \{g(y) + h(x-y)\} \quad (1)$$

Nótese que la recompensa máxima es función de la cantidad a invertir inicialmente. Cuando g y h se conocen, ese máximo puede evaluarse mediante un proceso sencillo de derivación. Así que, en este caso de un año, el problema es trivial.

En nuestro caso, el dinero que queda para tratar, tras un año de operación, es la cantidad destinada al cambio de los CTT y los CCR, debido a la hipótesis de que los beneficios no se reinvierten. Por tanto, la cantidad de capital durante el segundo año de operación es,

$$x_1 = ay + b(x-y)$$

que escribiremos como

$$x_1 = y_1 + (x_1 - y_1)$$

donde  $y_1$  es la cantidad gastada en la adquisición de nuevos CTT y  $x_1 - y_1$  la gastada en CCR. Entonces, durante el segundo año de operación tenemos,

- Beneficios de la inversión en CTT =  $g(y_1)$
- Beneficios de la inversión en CCR =  $h(x_1 - y_1)$

y, al final del segundo año,

- Valor del cambio de CTT =  $ay_1$
- Valor del cambio de CCR =  $b(x_1 - y_1)$

Con lo que el beneficio total tras dos años de operación es,

$$Y_2(x,y,y_1) = g(y) + h(x-y) + g(y_1) + h(x_1 - y_1)$$

y la recompensa máxima tras esos dos años es,

$$f_2(x) = \text{Max}_{0 \leq y \leq x, 0 \leq y_1 \leq x_1} \{Y_2(x,y,y_1)\} = \text{Max}_{0 \leq y \leq x, 0 \leq y_1 \leq x_1} \{g(y) + h(x-y) + g(y_1) + h(x_1 - y_1)\}$$

o bien, de acuerdo con la expresión (1),

$$f_2(x) = \text{Max}_{0 \leq y \leq x, 0 \leq y_1 \leq x_1} \{g(y) + h(x-y) + f_1(x_1)\}$$

con lo que, sustituyendo tenemos para el máximo beneficio,

$$f_2(x) = \text{Max}_{0 \leq y \leq x} \{g(y) + h(x-y) + f_1[ay + b(x-y)]\}$$

Esta ecuación supone que el máximo producido durante un periodo de dos años, puede determinarse derivando las funciones en ella, de acuerdo con las restricciones que comportan. El valor de  $y(x)$  que maximice esas funciones será la decisión óptima a tomar al principio de una operación bianual que comienza con una cantidad x.

El análisis puede repetirse para una operación trianual y obtendríamos que,

$$f_3(x) = \text{Max}_{0 \leq y \leq x} \{g(y) + h(x-y) + f_2[ay + b(x-y)]\}$$

y así sucesivamente. Por tanto, para un periodo de N años, la recompensa máxima vendrá dada por,

$$f_N(x) = \text{Max}_{0 \leq y \leq x} \{g(y) + h(x-y) + f_{N-1}[ay + b(x-y)]\} \quad (2)$$

Esta es la ecuación funcional básica para este proceso de decisión N-etápico. Comenzando con  $f_1(x)$ , como lo determina (1), esta ecuación funcional se usa para calcular  $f_2(x)$ , que luego se usa para calcular  $f_3(x)$ ,... En cada etapa, se obtiene también la decisión óptima  $y_j(x)$  a tomar al comienzo de la j-esima etapa. El valor de  $y(x)$  que maximiza las funciones entre corchetes en (2) es la decisión óptima a tomar al comienzo de la operación del año N, supuesto que inicialmente comenzamos con una cantidad x.

Para la obtención de la solución por un método directo, tendríamos que resolver las N ecuaciones simultáneas que se obtienen igualando a cero las correspondientes derivadas parciales, supuesto que esas ecuaciones fueran derivables, lo que se prevé complicado.

### El Principio de Optimalidad

Como hemos visto, el enfoque funcional expuesto sirve para el estudio de los procesos de decisión multietápico de modo que, aplicando repetidamente la ecuación funcional, se puede encontrar la sucesión de decisiones óptimas para el proceso N-etápico considerado. Aunque el problema de inversión comentado se ha escogido como ilustración, el enfoque funcional proporciona una herramienta muy útil en general para resolver problemas de decisión multietápico. Sin embargo, ahora nos vamos a dedicar a introducir el Principio de Optimalidad, que nos permitirá obtener la ecuación funcional que describe un proceso de decisión multietápico como una consecuencia inmediata.

Consideremos, en primer lugar, un proceso de decisión uni-etápico. Sea  $x$  el vector de estado que caracteriza al sistema físico en cuestión en cualquier instante de tiempo. Suponemos que  $x$  es un vector columna k-dimensional. Si el estado del sistema se transforma de  $x^1$  a  $x^2$  por medio de la transformación,

$$x^2 = g(x^1, m_1)$$

esta transformación producirá una respuesta, o recompensa,

$$R_1 = r(x^1, m_1)$$

El problema consiste en elegir una decisión  $m_1$  que maximice la respuesta. A la decisión  $m_1$  se le llama política uni-etápica. Esta claro que la solución a este problema de decisión uni-etápico no tiene dificultad, porque el máximo de la respuesta esta dado por,

$$f_1(x^1) = \text{Max}_{m_1} \{r(x^1, m_1)\}$$

La decisión que da el máximo valor de la salida, se llama una decisión óptima, o estrategia de control óptima.

En un proceso de decisión bi-etápico, si el estado del sistema se transforma, primero, de  $x^1$  a  $x^2$  por la transformación,

$$x^2 = g(x^1, m_1)$$

y después, de  $x^2$  a  $x^3$  por,

$$x^3 = g(x^2, m_2)$$

esta sucesión de operaciones produce una única salida, o recompensa total,

$$R_2 = r(x^1, m_1) + r(x^2, m_2)$$

Entonces, en este caso, el problema del diseño óptimo consiste en elegir una sucesión de decisiones factibles  $m_1$  y  $m_2$  que maximicen la recompensa total. Este es

un proceso de decisión bi-etápico en el que  $r(x^1, m_1)$  es la salida debida a la primera elección de una decisión, y  $r(x^2, m_2)$  es la salida tras la elección de la segunda decisión. Esta sucesión de decisiones  $\{m_1, m_2\}$  se llama política bi-etápica.

La máxima recompensa estará dada por,

$$f_2(x^1) = \text{Max}_{m_1, m_2} \{r(x^1, m_1) + r(x^2, m_2)\}$$

La función de recompensa total ahora se maximiza sobre la política  $\{m_1, m_2\}$ . A la política que maximiza a  $R_2$  se le llama política optimal. Como fácilmente puede verse, manejar un proceso de decisión bi-etápico es mas complicado que manejar el uni-etápico anterior. La dificultad y la complejidad aumentan conforme lo hace el numero de etapas del problema en consideración.

En general, para un proceso N-etápico, el problema consiste en elegir aquella política N-etápica  $\{m_1, m_2 \dots, m_N\}$  tal que maximice la recompensa total,

$$R_N = \sum_j r(x^j, m_j), \quad j \in J = \{1, 2, \dots, N\}$$

El estado del sistema se transforma de  $x^1$  a  $x^2$  por  $x^2 = g(x^1, m_1)$ , de  $x^2$  a  $x^3$  por  $x^3 = g(x^2, m_2), \dots$ , y finalmente desde  $x^{N-1}$  hasta  $x^N$  por  $x^N = g(x^{N-1}, m_{N-1})$ . La recompensa máxima de este proceso N-etápico estará dada por,

$$f_N(x^1) = \text{Max}_{\{m_j\}} \{ \sum_j r(x^j, m_j) \}, \quad j \in J$$

La política  $\{m_j\}$  que determina  $f_N(x^1)$  es la política optimal, o estrategia de control optimal. En este caso, las  $m_j$  forman una política de control N-etápica.

Como antes se ha comentado, llevar a cabo el procedimiento de maximización por un método directo, requiere la resolución de las N ecuaciones simultaneas que se obtienen igualando a cero las derivadas parciales, con respecto a  $m_j, j \in J$ , de las cantidades entre los corchetes, lo que puede ser formidablemente complicado en tiempo. Es evidente, entonces, que para resolver un problema de decisión optimal con un elevado numero de etapas es necesario un procedimiento sistemático que mantenga al problema en niveles tratables. Tal procedimiento sistemático de resolución puede obtenerse haciendo uso del principio fundamental de la PD, el Principio de Optimalidad de Bellman, que establece que:

***Una política optimal, o estrategia de control optimal, tiene la propiedad de que, cualquiera que sea el estado inicial y la decisión inicial elegidos, la decisión restante forma una estrategia de control optimal con respecto al estado que resulta a consecuencia de la primera decisión.***

Otra forma, equivalente, de expresarlo es,

***Una política es optima si en un periodo dado, cualesquiera que sean las decisiones precedentes, las decisiones que quedan por tomar constituyen una política optima teniendo en cuenta los resultados de las decisiones precedentes, o de forma mucho mas concisa: Una política optima solo puede estar formada por subpolíticas optimas.***

En resumen, el Principio de Optimalidad de Bellman lo que establece es que una política optima solo puede estar formada por subpolíticas optimas

El Principio de Optimalidad, que describe las propiedades básicas de las estrategias de control optimal, se basa en el concepto de invariante encajado, según el cual para resolver un problema de decisión optimal especifico, el problema inicial se encaja en una familia de problemas similares que son mas fáciles de resolver. Para procesos de decisión multietápicos esto permitirá reemplazar, el problema de

optimización multietápico original, por el problema de resolver una sucesión de procesos de decisión uni-etápico que, indudablemente, son mas fáciles de manejar.

Como antes hemos deducido, para el proceso N-etápico que considerábamos el problema consiste en elegir aquella política N-etápica  $\{m_1, m_2 \dots, m_N\}$  tal que maximice la recompensa total,

$$R_N = \sum_j r(x^j, m_j), \quad j \in J = \{1, 2, \dots, N\}$$

Ahora, de acuerdo con el Principio de Optimalidad, la recompensa total del mismo proceso de decisión multietápico puede escribirse como,

$$R_N = r(x^1, m_1) + f_{N-1}[g(x^1, m_1)]$$

donde el primer termino, del miembro de la derecha, es la recompensa inicial y el segundo representa la recompensa máxima debida a las ultimas N-1 etapas. Por tanto, la recompensa máxima estará dada por,

$$f_N(x^1) = \text{Max}_{m_1} \{r(x^1, m_1) + f_{N-1}[g(x^1, m_1)]\}$$

Esta ecuación es valida solo para  $N \geq 2$ , pero para  $N = 1$  la recompensa máxima estaría dada por,

$$f_1(x^1) = \text{Max}_{m_1} \{r(x^1, m_1)\}$$

Es obvio que, aplicando este principio fundamental, un proceso de decisión N-etápico se reduce a una sucesión de N procesos de decisión uni-etápico, con lo que disponemos de un procedimiento iterativo y sistemático para resolver eficientemente aquel problema.

## El enfoque de la PD

A la vista de lo anterior, para abordar la resolución de un problema con la técnica de la PD, habría que verificar, en primer lugar, la naturaleza N-etápica del problema bajo consideración, para caracterizar la estructura de una solución optimal. Posteriormente, se tendría que comprobar el cumplimiento del Principio de Optimalidad de Bellman como condición absolutamente necesaria para su aplicación, y si este se verificara, apoyándose en su propio significado en el caso concreto que estemos resolviendo, formular una ecuación de recurrencias que represente la forma de ir logrando etapa por etapa la solución optimal correspondiente. A partir de esa ecuación, a continuación, habrá que determinar la solución optimal resolviendo problemas encajados para, por ultimo, construir la solución. Con esos ingredientes, se podría iniciar la solución del problema que, a su vez, podría realizarse desde el primer estado hasta el ultimo, o bien a la inversa.

De forma mas concisa, el desarrollo de un algoritmo de PD correspondería a las siguientes etapas:

- 1.- Caracterizar la estructura de una solución optimal
- 2.- Definir recursivamente el valor de una solución optimal
- 3.- Calcular el valor de una solución optimal en forma encajada de menos a mas, y
- 4.- Construir una solución optimal a partir de la información previamente calculada.

Las etapas 1-3 constituyen la base de la solución de PD para un cierto problema. La etapa 4 puede omitirse si lo único que buscamos es el valor de una solución optimal. Cuando se realiza la etapa 4, a veces mantenemos información adicional durante la etapa 3 para facilitar la construcción de una solución optimal.

Vemos así que la PD, igual que la técnica Divide y Vencerás, resuelve problemas combinando las soluciones de subproblemas. Como sabemos, los algoritmos Divide y Vencerás particionan el problema en subproblemas independientes, resuelven estos recursivamente, y combinan sus soluciones para obtener la solución del problema original. En contraste, la PD es aplicable cuando los subproblemas no son independientes, es decir, cuando los subproblemas comparten sub-subproblemas. En este contexto, un algoritmo Divide y Vencerás hace mas trabajo del necesario, ya que resuelve repetidamente sub-subproblemas comunes. Un algoritmo de PD resuelve cada sub-subproblema solo una vez, salvando su solución en una tabla de cara a evitar el trabajo de recalcular la solución cada vez que se encuentra ese sub-subproblema.

Por otro lado, el enfoque que define la PD tiene también ciertos parecidos con el que propone la técnica greedy, pero la diferencia esencial entre el método greedy y el de la PD es que en el primero siempre se generaba solo una sucesión de decisiones, mientras que en PD pueden generarse muchas sucesiones de decisiones. Sin embargo las sucesiones que contengan subsucesiones suboptimales no pueden ser optimales (si se verifica el Principio de Optimalidad), y por tanto no se generaran. Pensemos por ejemplo en el problema del camino mínimo. Si queremos encontrar el camino de longitud mínima entre el vértice  $i$  y el  $j$ , sea  $A$  el conjunto de vértices adyacentes desde  $i$ . ¿Cual de los vértices en  $A$  debe ser el segundo vértice del camino?. No hay forma de tomar esa decisión en ese momento y garantizar que las decisiones futuras producirán una solución optimal.

Por ultimo, en lo que se refiere a similitudes, otra forma de resolver problemas en los que no es posible conseguir una sucesión de decisiones que, etapa por etapa, formen una sucesión optimal, es intentarlo sobre todas las posibles sucesiones de decisiones, es decir, lo que comúnmente se llama enfoque de la fuerza bruta. Según esa metodología, lo que se hace es enumerar todas esas sucesiones, y entonces tomar la mejor respecto al criterio que se este usando como objetivo. La PD haciendo uso explícito del Principio de Optimalidad de Bellman, a menudo, reduce drásticamente la cantidad de enumeraciones que hay que hacer, evitando la enumeración de algunas sucesiones que posiblemente nunca podrán ser optimales. De hecho mientras que el numero total de sucesiones de decisión diferentes es exponencial en el numero de decisiones (si hay  $d$  elecciones para cada una de las  $n$  decisiones, entonces hay  $d^n$  posibles sucesiones de decisiones), los algoritmos de PD suelen tener eficiencia polinomial.

A continuación nos centramos en la aplicación de la técnica de la PD para el diseño de algoritmos, es decir en la ilustración de la aplicación de las anteriores fases para la resolución de un problema con PD, considerando algunas situaciones que ya conocemos de capítulos anteriores, como son el problema de la mochila y el del camino mínimo.

### **a) La naturaleza N-etápica de los problemas**

En efecto, la solución al problema de la mochila puede entenderse como el resultado de una sucesión de decisiones. Tenemos que decidir los valores de  $x_i$ ,  $1 \leq i \leq n$ . Así, primero podríamos tomar una decisión sobre  $x_1$ , luego sobre  $x_2$ , etc. Una sucesión optimal de decisiones, verificando las restricciones del problema, será la que maximice la correspondiente función objetivo. Por su parte, en el problema del camino mínimo, una forma de encontrar el camino mínimo desde un vértice  $i$  a otro  $j$  en un grafo dirigido  $G$ , es decidiendo que vértice debe ser el segundo, cual el tercero, cual el cuarto, etc. hasta que alcanzáramos el vértice  $j$ . Una sucesión optimal de decisiones proporcionaría entonces el camino de longitud mínima que buscamos.

### **b) Comprobación del Principio de Optimalidad**



Concretamente, en el problema del camino mínimo, supongamos que  $i, i_1, i_2, \dots, i_k, j$  es el camino mínimo desde  $i$  hasta  $j$ . Comenzando con el vértice inicial  $i$ , se ha tomado la decisión de ir al vértice  $i_1$ . Como resultado de esta decisión, ahora el estado del problema está definido por el vértice  $i_1$ , y lo que se necesita es encontrar un camino desde  $i_1$  hasta  $j$ . Está claro que la sucesión  $i_1, i_2, \dots, i_k, j$  debe constituir un camino mínimo entre  $i_1$  y  $j$ . Si no, sea  $i_1, r_1, r_2, \dots, r_q, j$  un camino más corto entre  $i_1$  y  $j$ . Entonces  $i, i_1, r_1, r_2, \dots, r_q, j$  es un camino entre  $i$  y  $j$  que es más corto que el camino  $i, i_1, i_2, \dots, i_k, j$ . Por tanto el Principio de Optimalidad puede aplicarse a este problema.

Por otro lado, consideremos el problema de la mochila 0-1. Este problema es similar al anterior problema de la mochila que vimos en la técnica divide y vencerás, excepto que los  $x_i$  ahora están restringidos a tomar valores 0 o 1 exclusivamente. Notando Mochila  $(I, j, Y)$  al siguiente problema,

$$\begin{aligned} \text{Max: } & \sum_{1 \leq i \leq j} p_i x_i \\ \text{Sujeto a: } & \sum_{1 \leq i \leq j} w_i x_i \leq Y \\ & x_i = 0, 1; 1 \leq i \leq j \end{aligned}$$

el problema de la mochila 0-1 se representa por Mochila  $(1, n, M)$ .

Sea  $y_1, y_2, \dots, y_n$  una sucesión optimal de valores 0-1 para  $x_1, x_2, \dots, x_n$  respectivamente. Si  $y_1 = 0$ , entonces  $y_2, \dots, y_n$  debe ser una sucesión optimal para el problema Mochila  $(2, n, M)$ . Si no lo es, entonces  $y_1, y_2, \dots, y_n$  no es una sucesión optimal de Mochila  $(1, n, M)$ . Si  $y_1 = 1$ , entonces  $y_2, \dots, y_n$  debe ser una sucesión optimal para el problema Mochila  $(2, n, M - w_1)$ . Si no lo fuera, entonces habría otra sucesión 0-1,  $z_2, z_3, \dots, z_n$  tal que

$$\sum_{2 \leq i \leq n} w_i z_i \leq M - w_1$$

y

$$\sum_{2 \leq i \leq n} p_i z_i > \sum_{2 \leq i \leq n} p_i y_i$$

y por tanto la sucesión  $y_1, z_2, z_3, \dots, z_n$  es una sucesión para el problema de partida con mayor valor. De nuevo por tanto puede aplicarse el Principio de Optimalidad.

### c) Construcción de una ecuación recurrente con un enfoque adelantado

El Principio de Optimalidad, en si mismo, podría decirse que es conmutativo en el sentido de que la propiedad de una subpolítica de ser optimal, es independiente de que el estado inicial del proceso se considere a la izquierda del primer estado de esa subpolítica, o a la derecha del mismo. Esto describe en esencia lo que suele llamarse el enfoque adelantado (que de forma similar se explicaría para el retrasado) en la aplicación del principio de Optimalidad.

Consiguientemente, desde el punto de vista adelantado, el Principio de Optimalidad se aplicaría como sigue. Sea  $S_0$  el estado inicial del problema. Supongamos que hay que tomar  $n$  decisiones  $d_i$ ,  $1 \leq i \leq n$ . Sea  $D_1 = \{r_1, r_2, \dots, r_j\}$  el conjunto de los posibles valores que podría tomar  $d_1$ . Sea  $S_i$  el estado del problema tras tomar la decisión  $r_i$ ,  $1 \leq i \leq j$ . Sea  $\Gamma_i$  una sucesión de decisiones optimales con respecto al estado del problema  $S_i$ . Entonces, cuando se verifica el Principio de Optimalidad, una sucesión de decisiones optimales con respecto a  $S_0$  es la mejor de las sucesiones de decisiones  $r_i \Gamma_i$ ,  $1 \leq i \leq j$ .

En el caso del problema del camino mínimo, sea  $A_i$  el conjunto de los vértices adyacentes al vértice  $i$ . Para cada vértice  $k \in A_i$  sea  $\Gamma_k$  el camino mínimo desde  $k$  hasta  $j$ . Entonces un camino más corto desde  $i$  hasta  $j$  es el más corto de los caminos  $\{i, \Gamma_k / k \in A_i\}$

Consideremos ahora el Problema de la Mochila 0-1. Sea  $g_i(y)$  el valor de una solución óptima del problema Mochila  $(j+1, n, y)$ . Claramente  $g_0(M)$  es el valor de una solución óptima de Mochila  $(1, n, M)$ . Las posibles decisiones para  $x_1$  son 0 o 1 ( $D_1 = \{0, 1\}$ ). A partir del Principio de Optimalidad se sigue que

$$g_0(M) = \text{Max} \{g_1(M), g_1(M-w_1) + p_1\} \quad (3)$$

Hasta aquí el Principio de Optimalidad se ha establecido solo con respecto al estado y la decisión inicial, pero también puede aplicarse a estados y decisiones intermedios. Los dos siguientes ejemplos muestran como puede hacerse esto.

- 1) Caminos mínimos. Sea  $k$  un vértice intermedio en una camino mínimo desde  $i$  a  $j$ ,  $i, i_1, i_2, \dots, k, p_1, p_2, \dots, j$ . Los caminos  $i, i_1, i_2, \dots, k$ , y  $k, p_1, p_2, \dots, j$ , deben ser respectivamente los caminos mas cortos desde  $i$  a  $k$  y desde  $k$  a  $j$ .
- 2) Mochila 0-1. Sea  $y_1, y_2, \dots, y_n$  una solución óptima del problema Mochila  $(1, n, M)$ . Entonces para cada  $j$ ,  $1 \leq j \leq n$ ,  $y_1, \dots, y_j$  e  $y_{j+1}, \dots, y_n$  deben ser soluciones óptimas de los problemas

Mochila  $(1, j, \sum_{1 \leq i \leq j} w_i y_i)$  y Mochila  $(j+1, n, M - \sum_{1 \leq i \leq j} w_i y_i)$

respectivamente. Esta observación nos permite generalizar (3) a

$$g_i(y) = \text{Max} \{g_{i+1}(y), g_{i+1}(y - w_{i+1}) + p_i\} \quad (4)$$

La aplicación recursiva del Principio de Optimalidad produce una relación de recurrencia del tipo (4). Los algoritmos de Programación Dinámica resuelven estas recurrencias para obtener la solución del caso del problema que estemos considerando. La recurrencia (4) puede resolverse sabiendo que  $g_n(y) = 0$  para todo  $y$ . A partir de  $g_n(y)$  puede obtenerse  $g_{n-1}(y)$  usando la formula (4) con  $i = n-1$ . Entonces usando  $g_{n-1}(y)$  podemos obtener  $g_{n-2}(y)$ . Repitiendo esto, podremos determinar  $g_1(y)$ , y finalmente  $g_0(M)$  usando (4) con  $i = 0$ .

De la misma forma que hemos razonado la obtención de la ecuación recurrente en base al Principio de Optimalidad y al enfoque adelantado, podríamos haberlo hecho con respecto al enfoque atrasado, de modo que en este ultimo, la formulación para la decisión  $x_i$  se hace en términos de sucesiones de decisiones óptimas para  $x_1, \dots, x_{i-1}$ , mirando por tanto hacia detrás la sucesión  $x_1, \dots, x_n$ .

En lo que sigue aplicamos el enfoque de la PD al diseño de algoritmos que resuelven algunos problemas importantes, comenzando por ilustrar su aplicación sobre el siguiente sencillo y conocido ejemplo.

### El problema de la subdivisión óptima

Se quiere dividir una cantidad positiva  $c$  en  $n$  partes de forma que el producto de esas  $n$  partes sea máximo. Para determinar la subdivisión óptima aplicamos la técnica de la Programación Dinámica. Sea  $f_n(c)$  el máximo producto alcanzable,  $x$  el valor de la primera subdivisión, y  $(c-x)$  el valor de las  $(n-1)$  partes restantes. De acuerdo con el Principio de Optimalidad, la ecuación funcional que describe este problema de subdivisión óptima es,

$$f_n(c) = \text{Max}_{0 \leq x \leq c} \{x f_{n-1}(c-x)\}$$

Esta ecuación es valida para  $n \geq 2$ . Es obvio que cuando  $n = 1$ ,

$$f_1(c) = c \text{ y } f_1(c-x) = c - x$$

Así, para  $n = 2$

$$f_2(c) = \text{Max}_{0 \leq x \leq c} \{x f_1(c-x)\} = \text{Max}_{0 \leq x \leq c} \{x(c-x)\}$$

Haciendo cálculos se encuentra que el valor de  $x$  que maximiza ese producto es  $x = c/2$ , y así la política óptima, es decir, la subdivisión óptima para este proceso de decisión bi-etápico es,

$$\{m_i\} = \{c/2, c/2\}$$

siendo el máximo valor del producto

$$f_2(c) = (c/2)^2$$

Consideremos ahora el caso  $n = 3$ . El máximo producto alcanzable es,

$$f_3(c) = \text{Max}_{0 \leq x \leq c} \{xf_2(c-x)\} = \text{Max}_{0 \leq x \leq c} \{x(c-x)^2/4\}$$

Realizando cálculos se obtiene que el valor máximo de  $x$  es  $x = c/3$ .

La subdivisión de la parte restante,  $2c/3$ , constituye un proceso de decisión bi-etápico. De acuerdo con el anterior análisis, esa parte se divide en  $c/3$  y  $c/3$ , y por tanto la subdivisión óptima para el proceso tri-etápico es

$$\{m_i\} = \{c/3, c/3, c/3\}$$

siendo el valor máximo del producto

$$f_3(c) = (c/3)^3$$

Cuando la cantidad  $c$  dada se divide en cuatro partes, el máximo producto que se puede alcanzar es

$$f_4(c) = \text{Max}_{0 \leq x \leq c} \{xf_3(c-x)\} = \text{Max}_{0 \leq x \leq c} \{x(c-x)^3/27\}$$

siendo el máximo valor de  $x$ ,  $c/4$ . Ahora la subdivisión de la parte restante,  $3c/4$ , constituye un proceso tri-etápico, de subdivisión óptima  $c/4$ ,  $c/4$  y  $c/4$ , con lo que la subdivisión óptima de este problema tetra-etápico es

$$\{m_i\} = \{c/4, c/4, c/4, c/4\},$$

y el valor máximo del producto es  $f_4(c) = (c/4)^4$ .

Todo el anterior análisis nos lleva a conjeturar que la solución para este problema, cuando  $n = k$ , sería

$$\{m_i\} = \{c/k, c/k, \dots, c/k\}$$

siendo el máximo valor del producto  $f_k(c) = (c/k)^k$ .

Realmente puede demostrarse por inducción matemática que las anteriores relaciones son ciertas para cualquier  $k$ . De acuerdo con el principio de Optimalidad, se sigue que,

$$f_{k+1}(c) = \text{Max}_{0 \leq x \leq c} \{xf_k(c-x)\} = \text{Max}_{0 \leq x \leq c} \{x(c-x)^k/k\}$$

lo que por simple cálculo nos lleva a que  $x = c/(k+1)$ , siendo el máximo valor del producto,

$$f_{k+1}(c) = [c/(k+1)]^{k+1}$$

Así, por inducción matemática, la subdivisión óptima del proceso  $n$ -etápico puede obtenerse como

$$\{m_i\} = \{c/n, c/n, \dots, c/n\}$$

con un valor máximo del producto  $f_n(c) = (c/n)^n$

Evidentemente este problema es muy simple y podría resolverse fácilmente con métodos convencionales. Sin embargo hay que destacar que ilustra la idea de la resolución del mismo como un problema de decisión multi-etápico. La formulación del

problema en este sentido es lo que realmente da la clave sobre la gran utilidad del enfoque de la Programación Dinámica.

### El Problema del Camino Mínimo

Sea  $G = (N, A)$  un grafo dirigido en el que  $N$  es su conjunto de nodos y  $A$  el de sus arcos. Cada arco tiene asociada una longitud, no negativa. El problema consiste en determinar el camino de longitud mínima que una cualquier par de nodos del grafo.

Supondremos los nodos numerados de 1 a  $n$ ,  $N = \{1, 2, \dots, n\}$  y que la matriz  $L$  da la longitud de cada arco, de modo que  $L(i, i) = 0$ ,  $L(i, j) \geq 0$  si  $i$  es distinto de  $j$ , y  $L(i, j) = \infty$  si no existe el arco  $(i, j)$ . Aquí, el Principio de Optimalidad se aplica del siguiente modo: Si  $k$  es un nodo en el camino mínimo que une  $i$  con  $j$ , entonces la parte de ese camino que va de  $i$  hasta  $k$ , y la del que va de  $k$  hasta  $j$ , es también optimal.

Para resolver el problema nos basamos en el celebre Algoritmo de Floyd. De acuerdo con el, construimos una matriz  $D$  que da la longitud del camino mínimo entre cada par de nodos. El algoritmo comienza asignando a  $D$ ,  $L$  y, entonces, realiza  $n$  iteraciones. Tras la iteración  $k$ ,  $D$  da la longitud de los caminos mínimos que solo usan como nodos intermedios los del conjunto  $\{1, 2, \dots, k\}$ . Después de  $n$  iteraciones tendremos, por tanto la solución buscada. En la iteración  $k$ , el algoritmo tiene que chequear, para cada par de nodos  $(i, j)$ , si existe o no un camino que pase a través de  $k$  que sea mejor que el actual camino minimal que solo pasa a través de los nodos  $\{1, 2, \dots, k-1\}$ . Sea  $D$  la matriz después de la  $k$ -ésima iteración. El chequeo puede expresarse como,

$$D_k(i, j) = \min \{D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)\}$$

donde hacemos uso del principio de Optimalidad para calcular la longitud del camino mas corto que pasa a través de  $k$ . También hacemos uso implícitamente del hecho de que un camino minimal no pasa a través de  $k$  dos veces.

En la  $k$ -ésima iteración, los valores en la  $k$ -ésima fila y  $k$ -ésima columna de  $D$  no cambian, ya que  $D(k, k)$  siempre vale cero. Por tanto, no es necesario proteger estos valores al actualizar  $D$ . Esto nos permite trabajar siempre con una matriz  $D$  bidimensional, mientras que a primera vista parece que habríamos necesitado una matriz  $n \times n \times 2$  (o incluso  $n \times n \times n$ ). Con esto, el problema se resuelve conforme al siguiente programa,

```

Procedimiento Floyd (Var D: Array [1..n, 1..n] of real);
    L: Array [1..n, 1..n] of real;
    Var i, j, k: Integer;
begin
    For i := 1 to n do
        For j := 1 to n do
            D[i, j] := L[i, j];
    For i := 1 to n do
        D[i, i] := 0;
    For k := 1 to n do
        For i := 1 to n do
            For j := 1 to n do
                If D[i, k] + D[k, j] < D[i, j]
                Then D[i, j] := D[i, k] + D[k, j]
    End;

```

Es obvio que este algoritmo consume un tiempo  $O(n^3)$ . Para resolver el problema también podemos usar el algoritmo de Dijkstra, entonces aplicaríamos ese algoritmo  $n$  veces, cada una de ellas eligiendo un nodo diferente como origen. Si queremos usar

la versión de Dijkstra que trabaja con una matriz de distancias, el tiempo de calculo total esta en  $n \times O(n^2)$ , es decir en  $O(n^3)$ . El orden es el mismo que para el algoritmo de Floyd, pero la simplicidad de este supone que, en la practica, probablemente sea mas rápido. Por otro lado, si usamos la versión del algoritmo de Dijkstra que trabaja con una pila y, por tanto con listas de las distancias a los nodos adyacentes, el tiempo total esta en  $O((an+n^2)\log n)$ , donde  $a$  es el numero de arcos en el grafo. Si el grafo no es muy denso ( $a \ll n^2$ ) puede ser preferible usar el algoritmo de Dijkstra  $n$  veces, pero si el grafo es denso ( $a \equiv n^2$ ), es mejor usar el algoritmo de Floyd.

Pero, generalmente, queremos saber por donde va el camino mas corto, y no solo su longitud. En este caso, podemos usar una segunda matriz  $P$ , inicialmente con valores cero, en la que  $P(i,j)$  al final contendrá el valor de la ultima iteración que causo un cambio en el array  $D(i,j)$ . Si  $P(i,j) = 0$ , el camino mínimo entre  $i$  y  $j$  será directo a través del arco  $(i,j)$ . De acuerdo con esto, si el ciclo mas interior del algoritmo de Floyd lo cambiamos para que haga que

```
If  $D(i,k) + D(k,j) < D(i,j)$  Then  $D(i,j) := D(i,k) + D(k,j)$   

 $P(i,j) := k$ 
```

nos queda el siguiente algoritmo,

```
Procedure Shortest (D:Array[1..n,1..n]; L:Array [1..n,1..n]; P:Array [1..n, 1..n]);
Var i,j,k : Integer;
Begin
  For i := 1 to n do
    For j := 1 to n do begin
       $D[i,j] := L[i,j]$ ;
       $P[i,j] := 0$ 
    End;
  For i := 1 to n do
     $D[i,i] := 0$ ;
  For k := 1 to n do
    For i := 1 to n do
      For j := 1 to n do
        If  $D[i,k] + D[k,j] < D[i,j]$  then begin
           $D[i,j] := D[i,k] + D[k,j]$ ;
           $P[i,j] := k$ 
        End
      End
    End
  End;
```

Ahora, para obtener todos los vértices intermedios en el camino mínimo entre  $i$  y  $j$ , usamos el procedimiento siguiente,

```
Procedure Path (i,j: Integer);
Var K : Integer;
Begin
   $k := P[i,j]$ ;
  If  $k = 0$  then
    Return;
  Path (i,k);
  Writeln (k);
  Path (k,j)
End;
```

## El Problema del Viajante de Comercio

Dado un grafo con longitudes no negativas asociadas a sus arcos, queremos encontrar el circuito mas corto posible que comience y termine en un mismo nodo, es decir, un camino cerrado que recorra todos los nodos una y solo una vez y que tenga longitud minimal.

Sea  $G = (N, A)$  nuestro grafo dirigido. Como antes, tomaremos  $N = \{1, 2, \dots, n\}$ , y notaremos la longitud de los arcos por  $L_{ij}$ , con  $L(i, i) = 0$ ,  $L(i, j) \geq 0$  si  $i$  es distinto de  $j$ , y  $L(i, j) = \infty$  si no existe el arco  $(i, j)$ .

Suponemos, sin perdida de generalidad, que el circuito comienza y termina en el nodo 1. Por tanto, esta constituido por un arco  $(1, j)$ , seguido de un camino de  $j$  a 1 que pasa exactamente una vez a través de cada nodo de  $N - \{1, j\}$ . Si el circuito es optimal, es decir, de longitud mínima, entonces ese es el camino de  $j$  a 1 y vale el principio de Optimalidad.

Consideremos un conjunto de nodos  $S \subseteq N - \{1\}$  y un nodo mas  $i \in N - S$ , estando permitido que  $i = 1$  solo si  $S = N - \{1\}$ . Definimos el valor  $g(i, S)$  para cada índice  $i$ , como la longitud del camino mas corto desde el nodo  $i$  al nodo 1 que pasa exactamente una vez a través de cada nodo de  $S$ . Usando esta definición,  $g(1, N - \{1\})$  es la longitud de un circuito optimal.

Por el principio de Optimalidad, vemos que

$$g(1, N - \{1\}) = \min_{2 \leq j \leq n} [L_{1j} + g(j, N - \{1, j\})] \quad (5)$$

Mas generalmente, si  $i$  no es igual a 1, el conjunto  $S$  no es vacío y es distinto de  $N - \{1\}$  e  $i \notin S$ ,

$$g(i, S) = \min_{j \in S} [L_{ij} + g(j, S - \{j\})] \quad (6)$$

Además,

$$g(i, \emptyset) = L_{i1}, \quad i = 2, 3, \dots, n$$

Por tanto, los valores de  $g(i, S)$  se conocen cuando  $S$  es vacío.

Podemos aplicar (6) para calcular la función  $g$  en todos los conjuntos  $S$  que contienen exactamente un nodo (que no es el 1). Entonces, de nuevo, podemos volver a aplicar (6) para calcular  $g$  en todos los conjuntos  $S$  que contienen dos nodos (distintos del 1), y así sucesivamente. Cuando se conoce el valor de  $g[j, N - \{1, j\}]$  para todos los nodos  $j$ , excepto para el nodo 1, podemos usar (5) para calcular  $g(1, N - \{1\})$  y, definitivamente, resolver el problema

El tiempo que consumirá este algoritmo puede calcularse como sigue,

a) Calcular  $g(j, \emptyset)$  supone la realización de  $n-1$  consultas de una tabla.

b) Calcular todas las  $g(i, S)$  tales que  $1 \leq \#S = k \leq n-2$ , supone realizar,

$$(n-1) \times C_{n-2, k} \times k$$

adiciones ( $n-1$  corresponden a los posibles valores que puede tomar la variable  $i$ ,  $k$  provienen de los valores que puede tomar la variable  $j$ , y las combinaciones restantes son todos los conjuntos que podemos formar de  $n-2$  elementos tomados de  $k$  en  $k$ ).

c) Calcular  $g(1, N - \{1\})$  implica  $n-1$  adiciones.

Estas operaciones pueden usarse como referencia para calcular la eficiencia del algoritmo. Así el tiempo que se lleva el algoritmo en cálculos es,

$$O[2(n-1) + \sum_{k=1..(n-2)} (n-1) \times k \times C_{n-2,k}] = O(n^2 2^2)$$

ya que

$$\sum_{k=1..r} k \times C_{r,k} = r 2^{r-1}$$

Este tiempo es bastante considerable pero, sin embargo, es bastante mejor que uno de  $O(n!)$  que es el que proporciona el método directo clásico sin contar con el enfoque de la PD. La siguiente tabla, ilustra esto.

	Tiempo	Tiempo
N	Método directo	PD
	$n!$	$n^2 2^2$
5	120	800
10	3.628.800	102.400
15	$1.31 \times 10^{12}$	7.372.800
20	$2.43 \times 10^{18}$	419.430.400

A partir de ella vemos el enorme incremento que sufren el tiempo y el espacio necesarios conforme aumenta  $n$ . Por ejemplo,  $20^2 2^{20}$  microsegundos es menos de siete minutos, mientras que  $20!$  microsegundos supera las 77 mil años.

Por ejemplo consideremos un grafo definido por la siguiente matriz de distancias,

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

Entonces,

$$g(2, \emptyset) = c_{21} = 5; g(3, \emptyset) = c_{31} = 6 \text{ y } g(4, \emptyset) = c_{41} = 8$$

Usando (4) obtenemos

$$g(2, \{3\}) = c_{23} + g(3, \emptyset) = 15; \quad g(2, \{4\}) = 18$$

$$g(3, \{2\}) = 18; \quad g(3, \{4\}) = 20$$

$$g(4, \{2\}) = 13; \quad g(4, \{3\}) = 15$$

A continuación, calculamos  $g(i, S)$  para conjuntos  $S$  de cardinal 2,  $i \neq 1$ ,  $1 \notin S$  e  $i \notin S$ :

$$g(2, \{3, 4\}) = \text{Min} [c_{23} + g(3, \{4\}), c_{24} + g(4, \{3\})] = 25$$

$$g(3, \{2, 4\}) = \text{Min} [c_{32} + g(2, \{4\}), c_{34} + g(4, \{2\})] = 25$$

$$g(4, \{2, 3\}) = \text{Min} [c_{42} + g(2, \{3\}), c_{43} + g(3, \{2\})] = 23$$

y finalmente,

$$g(1, \{2, 3, 4\}) = \text{Min} [c_{12} + g(2, \{3, 4\}), c_{13} + g(3, \{2, 4\}), c_{21} + g(4, \{2, 3\})] \\ = \text{Min} \{35, 40, 43\} = 35$$

que es la longitud del circuito buscado.

## El Problema de la Mochila 0-1

Ya conocemos la terminología y notación de este problema. En definitiva, podemos obtener una solución para el problema de la mochila tomando una sucesión de decisiones sobre las variables  $x_1, \dots, x_n$ . Una decisión sobre la variable  $x_i$  supone decidir que valor (0 o 1) ha de tomar.

Supongamos que las decisiones sobre las  $x_i$  se hacen en el orden  $x_n, \dots, x_1$ . Tras una decisión sobre  $x_n$  nos podemos encontrar ante una de estas dos posibilidades: la capacidad restante de la mochila es  $M$ , y no se ha producido incremento de beneficio, o bien la capacidad restante es  $M - w_n$  y hemos aumentado el beneficio en  $p_n$ . Como las restantes decisiones  $x_{n-1}, \dots, x_1$  deben ser optimales con respecto al estado del problema resultante de la decisión tomada sobre  $x_n$ , ya que en otro caso,  $x_n, \dots, x_1$  no sería optimal, se verifica el Principio de Optimalidad.

Sea  $f_j(X)$  el valor de una solución optimal del problema Mochila(1,j,X). Como se verifica el Principio de Optimalidad, obtenemos,

$$f_n(M) = \text{Max} \{f_{n-1}(M), f_{n-1}(M-w_n) + p_n\}$$

que para un  $i > 0$  arbitrario, se generaliza a,

$$f_i(X) = \text{Max} \{f_{i-1}(X), f_{i-1}(X-w_i) + p_i\}$$

Esta ecuación puede resolverse para  $f_n(M)$  teniendo en cuenta que  $f_0(X) = 0$  para todo  $X$ , y que  $f_i(x) = \infty$ ,  $x < 0$ . Entonces se pueden calcular sucesivamente  $f_1, f_2, \dots, f_n$ .

Por ejemplo, consideremos el problema definido por  $n = 3$ ,  $M = 6$ ,  $(w_1, w_2, w_3) = (2, 3, 4)$ , y  $(p_1, p_2, p_3) = (1, 2, 5)$ . Cada función  $f_i$  esta perfectamente definida por los pares  $(P_j, W_j)$ , donde  $W_j$  es el valor de  $X$  en el que  $f_i$  da un salto.  $P_j = f_i(W_j)$ . Si hay  $r$  saltos entonces necesitamos conocer  $r$  pares  $(P_j, W_j)$ ,  $1 \leq j \leq r$ . Por conveniencia podemos incluir el par  $(0,0)$ . Así si suponemos que  $W_j < W_{j+1}$ ,  $0 \leq j \leq r$ , es evidente de la anterior ecuación que  $P_j < P_{j+1}$ . Además  $f_i(X) = f_i(W_i)$  para todo  $X$  tal que  $W_j \leq X \leq W_{j+1}$ ,  $0 \leq j \leq r$ .  $f_i(X) = f_i(W_r)$  para todo  $X$ ,  $X \geq W_r$ . Si  $S^{i-1}$  es el conjunto de todos los pares para  $f_{i-1}$  (incluyendo el  $(0,0)$ ) entonces se obtiene el conjunto  $S^i_1$  de todos los pares para  $g_i(X) = f_{i-1}(X - w_i) + p_i$  añadiendo a cada par en  $S^{i-1}$  el par  $(p_i, w_i)$ .

$$S^i_1 = \{(P, W) / (P - p_i, W - w_i) \in S^{i-1}\}$$

$S^i_1$  puede obtenerse ahora sin mas que mezclar juntos  $S^i_1$  y  $S^{i-1}$ . Esta mezcla se corresponde a tomar el máximo de las dos funciones  $f_{i-1}(X)$  y  $f_{i-1}(X-w_i) + p_i$ . Así si  $S^{i-1}$  y  $S^i_1$  tienen pares  $(P_j, W_j)$  y  $(P_k, W_k)$  y  $P_j \leq P_k$ , pero  $W_j \geq W_k$ , entonces el par  $(P_j, W_j)$  se elimina.

Por ejemplo en nuestro caso tendríamos:

$$S^0 = \{(0,0)\}; S^1_1 = \{(1,2)\}$$

$$S^1 = \{(0,0), (1,2)\}; S^2_1 = \{(2,3), (3,5)\}$$

$$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}; S^3_1 = \{(5,4), (6,6), (7,7), (8,9)\}$$

$$S^3 = \{(0,0), (1,2), (2,3), (5,4), (6,6), (7,7), (8,9)\},$$

donde hay que notar como hemos eliminado el par  $(3,5)$  de  $S^3$  como resultado de la anterior regla. Los detalles del algoritmo y de su aplicación los dejamos como practica. Pueden encontrarse en Horowitz-Sahni.

## Ejemplos

- 1) Se quieren asignar 12 personas a 3 tareas. La asignación de  $m$  personas a la tarea  $i$  produce  $f_i(m)$  según la siguiente tabla,



m	$f_1(m)$	$f_2(m)$	$f_3(m)$
0	0	0	0
1	3	2	2
2	5	4	3
3	6	5	5
4	7	7	6
5 o mas	7	8	6

Como debería realizarse la asignación para que lo producido sea máximo?.

Tenemos que,

$$F_3(12) = \text{Max}_{y \leq 12} \{f_3(y) + F_2(12-y)\}$$

$$F_2(n) = \text{Max}_{y \leq n} \{f_2(y) + f_1(n-y)\}$$

donde n puede valer 12, 11, 10, etc. Así, por ejemplo,

$$F_2(8) = \text{Max}_{y \leq 8} \{2+7, 4+7, 5+8, 7+7, 8+6, 7+5, \dots\}$$

Así,

N	12	11	10	9	8	7	6
$F_2(n)$	15	15	15	15	14	13	12

y,

$$F_3(12) = \text{Max} \{15, 15+2, 15+3, 15+5, 14+6, 13+6, \dots\} = 20$$

Por tanto la solución óptima se alcanza al asignar 4,5,3 o 3,5,4 o 4,4,4; produciéndose en todos los casos un beneficio de 20.

2) Supongamos ahora que queremos resolver con P.D. el siguiente problema,

$$\text{Max: } 10x_1 + 11x_2 + 12x_3$$

s.a:

$$4x_1 + 5x_2 + 6x_3 \leq 10$$

$$x_i \geq 0 \text{ y enteros}$$

Tenemos,

$$F_3(10) = \text{Max}_{0 \leq x_3 \leq 1} \{12x_3 + F_2(10-6x_3)\}$$

Hay que calcular  $F_2$  en los valores  $F_2(10-0)$  y  $F_2(10-6)$ , por lo que n valdrá 10 y 4 en la siguiente formula,

$$F_2(n) = \text{Max}_{0 \leq x_2 \leq [n/5]} \{11x_2 + F_1(n-5x_2)\}$$

Así hay que calcular  $F_1(10)$ ,  $F_1(5)$  y  $F_1(4)$ ,

$$n = 10 \Rightarrow F_1(10-5x_2) \Rightarrow F_1(10), F_1(5) \text{ y } F_1(4),$$

$$n = 4 \Rightarrow F_1(4-5x_2) \Rightarrow F_1(4)$$

Pero,

$$F_1(10) = \text{Max} \{10x_1\}_{x_1=2} = 20$$

$$F_1(5) = \text{Max} \{10x_1\}_{x_1=1} = 10$$

$$F_1(4) = \text{Max} \{10x_1\}_{x_1=1} = 10$$

Por tanto,

$$F_2(10) = \text{Max} \{20, 21, 22\} = 22 \text{ cuando } x_2 = 2 \text{ y } x_1 = 0,$$

$$F_2(4) = 10, \text{ cuando } x_2 = 0 \text{ y } x_1 = 1$$

Así, finalmente,

$$F_3(10) = 22 \text{ cuando } x_3 = 0, x_2 = 2 \text{ y } x_1 = 0$$

$$F_3(10) = 22 \text{ cuando } x_3 = 1, x_2 = 0 \text{ y } x_1 = 1$$

### El problema de la multiplicación encadenada de matrices

Otro ejemplo de problema resoluble con la técnica de la PD es un algoritmo para resolver el problema de la multiplicación encadenada de matrices. Suponemos una sucesión (cadena) de  $n$  matrices  $\{A_1, A_2, \dots, A_n\}$  que se han de multiplicar, y deseamos calcular el producto  $A_1 \times A_2 \times \dots \times A_n$ . Podemos calcular ese producto usando el algoritmo standard de multiplicación de pares de matrices como un procedimiento, una vez que hayamos parentizado esa expresión para evitar ambigüedades acerca de como han de multiplicarse las matrices.

Un producto de matrices se dice que esta completamente parentizado si esta constituido por una sola matriz, o el producto completamente parentizado de dos matrices, cerrado por paréntesis. La multiplicación de matrices es asociativa, y por tanto todas las parentizaciones producen el mismo resultado. Por ejemplo, si la cadena de matrices es la  $\{A_1, A_2, A_3, A_4\}$ , el producto  $A_1 \times A_2 \times A_3 \times A_4$  puede parentizarse completamente de cinco formas distintas:  $(A_1 \times (A_2 \times (A_3 \times A_4)))$ ,  $(A_1 \times (A_2 \times A_3) \times A_4)$ ,  $((A_1 \times A_2) \times (A_3 \times A_4))$ ,  $((A_1 \times (A_2 \times A_3)) \times A_4)$  y  $((A_1 \times A_2) \times A_3) \times A_4$ . La forma en que parenticemos un producto de matrices puede tener un fuerte impacto en el costo de evaluar el producto. Consideremos primero el costo de multiplicar dos matrices. El algoritmo standard, si  $A$  es una matriz de dimensión  $p \times q$  y  $B$  de dimensión  $q \times r$ , consumiría un tiempo proporcional a  $p \times q \times r$ .

Así, para ilustrar los diferentes costos asociados a las distintas parentizaciones de un producto de matrices, consideremos la cadena de tres matrices  $\{A_1, A_2, A_3\}$ , con dimensiones  $10 \times 100$ ,  $100 \times 5$  y  $5 \times 50$  respectivamente. Si multiplicamos conforme a la parentización  $((A_1 A_2) A_3)$ , realizamos  $10 \times 100 \times 5 = 5.000$  multiplicaciones para calcular el producto de la matriz  $(A_1 A_2)$ ,  $10 \times 5$ , mas otras  $10 \times 5 \times 50 = 2.500$  multiplicaciones para multiplicar esta matriz por  $A_3$ , realizando en total  $7.500$  multiplicaciones. Así mismo, para el producto  $(A_1 (A_2 A_3))$  realizamos  $100 \times 5 \times 50 = 25.000$  multiplicaciones para calcular  $(A_2 A_3)$ , mas  $10 \times 100 \times 50 = 50.000$  multiplicaciones para multiplicar  $A_1$  por esa matriz, teniendo que hacer definitivamente  $75.000$  multiplicaciones escalares. Por tanto el primer producto es 10 veces mas rápido.

El problema de la multiplicación encadena de matrices puede, por tanto, plantearse como sigue:

Dada una cadena de  $n$  matrices  $\{A_1, A_2, \dots, A_n\}$ , donde para  $i = 1, 2, \dots, n$  la matriz  $A_i$  tiene de dimensión  $p_{i-1} \times p_i$ , parentizar completamente el producto de las matrices  $A_1 \times A_2 \times \dots \times A_n$  de tal forma que se minimice el numero total de operaciones escalares que haya que realizar.

### Recuento del numero de parentizaciones

Antes de resolver el problema con PD, deberíamos convencernos que la enumeración de todas las parentizaciones posibles no proporciona un método eficiente. Notemos el

numero de parentizaciones alternativas de una sucesión de  $n$  matrices por  $P(n)$ . Como podemos dividir una sucesión de  $n$  matrices en dos (las  $k$  primeras y las  $k+1$  siguientes) para cualquier  $k = 1, 2, \dots, n-1$ , y entonces parentizar las dos subsucesiones resultantes independientemente, obtenemos la recurrencia:

$$\begin{aligned} P(n) &= 1 && \text{si } n = 1 \\ &= \sum_{k=1}^{n-1} P(k) \times P(n-k) && \text{si } n \geq 2 \end{aligned}$$

Puede demostrarse que la solución de esta ecuación es la sucesión de los Números de Catalan:

$$P(n) = C(n-1)$$

donde

$$C(n) = (n+1)^{-1} C_{2n,n}$$

es  $\Omega(4^n/n^{3/2})$ , por lo que el numero de soluciones es exponencial en  $n$  y, consiguientemente, el método de la fuerza bruta es una pobre estrategia para determinar la parentización óptima de una cadena de matrices.

### Estructura de una parentización óptima

La primera etapa de la técnica de la PD es la de caracterizar la estructura de una solución óptima. Para el problema de la multiplicación encadenada de matrices, podemos realizar esta primera etapa como sigue. Por conveniencia, notemos  $A_{i..j}$  la matriz que resulte de evaluar el producto

$A_i \times A_{i+1} \times \dots \times A_j$ . Una parentización óptima del producto  $A_1 \times A_2 \times \dots \times A_n$  divide el producto entre  $A_k$  y  $A_{k+1}$  para algún entero  $k$  en el rango  $1 \leq k \leq n$ . Es decir, para algún valor de  $k$ , primero calculamos las matrices  $A_{1..k}$  y  $A_{k+1..n}$  y entonces las multiplicamos para obtener el producto final  $A_{1..n}$ . El costo de esta parentización óptima es así el costo de calcular el producto  $A_{1..k}$ , más el costo de calcular el de  $A_{k+1..n}$  más el costo de multiplicar ambas.

La clave está en que la parentización de la primera subcadena  $A_1 \times \dots \times A_k$  dentro de la parentización óptima de  $A_1 \times A_2 \times \dots \times A_n$  debe ser una parentización óptima de  $A_1 \times A_2 \times \dots \times A_k$ . Por qué?. Si hubiera una forma menos costosa de parentizar  $A_1 \times A_2 \times \dots \times A_k$  entonces, sustituyendo esa parentización en la parentización óptima de  $A_1 \times A_2 \times \dots \times A_n$ , obtendríamos otra parentización de  $A_1 \times A_2 \times \dots \times A_n$  cuyo costo sería inferior a la de la óptima: una contradicción. Una observación similar es válida para la parentización de la segunda sub-cadena  $A_{k+1} \times \dots \times A_n$  en la parentización óptima de  $A_1 \times A_2 \times \dots \times A_n$ , también debe ser una parentización óptima de  $A_{k+1} \times \dots \times A_n$ .

Así, una solución óptima para un caso del problema de la multiplicación encadenada de matrices contiene las soluciones de los subproblemas del caso considerado, es decir, se está cumpliendo el Principio de Optimalidad de Bellman.

### Una solución recursiva

La segunda etapa de la técnica de la PD es definir recursivamente el valor de una solución óptima en términos de las soluciones óptimas de los subproblemas. Para nuestro problema, tomamos como subproblemas los de determinar el mínimo costo de una parentización de  $A_i \times A_{i+1} \times \dots \times A_j$ ,  $1 \leq i \leq j \leq n$ . Sea  $m[i,j]$  el mínimo número de multiplicaciones escalares necesarias para calcular la matriz  $A$ ; el costo de la forma menos costosa de calcular  $A_{1..n}$  deberá ser por tanto  $m[1,n]$ .

Se puede definir  $m[i,j]$  recursivamente como sigue. Si  $i = j$ , la cadena consiste de un solo elemento  $A_{i..j} = A_i$ , por lo que no hay que hacer ninguna multiplicación para calcular el producto. Así  $m[i,i] = 0$  para  $i = 1, 2, \dots, n$ . Para calcular  $m[i,j]$  cuando  $i < j$ , nos aprovecharemos de la estructura de una solución óptima a partir de la etapa 1. Supongamos que la parentización óptima hace el producto  $A_i \times A_{i+1} \times \dots \times A_j$ , tomando como punto de división  $A_k$  y  $A_{k+1}$ , con  $i \leq k < j$ . Entonces  $m[i,j]$  es igual al costo mínimo de calcular los subproductos  $A_{i..k}$  y  $A_{k+1..j}$  más el costo de multiplicar esas dos matrices resultantes. Como calcular la matriz producto  $A_{i..k} \cdot A_{k+1..j}$  supone hacer  $p_{i-1} \cdot p_k \cdot p_j$  multiplicaciones, obtenemos:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$$

Esta ecuación recurrente supone que conocemos el valor de  $k$ , lo que no es cierto. Hay solo  $j - i$  posibles valores para  $k$ :  $k = i, i+1, \dots, j-1$ . Como la parentización óptima debe usar uno de estos valores de  $k$ , solo necesitamos chequearlos para encontrar el mejor. Así nuestra definición recursiva del costo mínimo de la parentización del producto de  $A_i \times A_{i+1} \times \dots \times A_j$  es:

$$m[i,j] = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j\} & \text{si } i < j \end{cases} \quad (2)$$

Los valores  $m[i,j]$  dan los costos de las soluciones óptimas de los subproblemas. Ahora definimos también  $s[i,j]$  como el índice  $k$  en el que podemos dividir el producto  $A_i \times A_{i+1} \times \dots \times A_j$  para obtener una parentización óptima. Esto es,  $s[i,j]$  es aquel  $k$  tal que

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j.$$

### Calculo de los costos óptimos

Llegados a este punto, es una sencilla tarea escribir un algoritmo recursivo que, basado en la última ecuación recurrente, calcule el costo mínimo  $m[1,n]$  asociado a la multiplicación de  $A_1 \times A_2 \times \dots \times A_n$ . Sin embargo, este algoritmo va a consumir un tiempo exponencial. En efecto, el algoritmo quedaría como,

Matrices-Encadenadas-Recursivo ( $p, i, j$ )

If  $i = j$

Then Return 0

$m[i,j] = \infty$

For  $k = 1$  to  $j - i + 1$  do

$q = \text{Matrices-Encadenadas-Recursivo}(p, i, k)$

        +  $\text{Matrices-Encadenadas-Recursivo}(p, k+1, j) + p_{i-1} \cdot p_k \cdot p_j$

    If  $q < m[i,j]$

        Then  $m[i,j] = q$

Return  $m[i,j]$

Podemos plantear la siguiente recurrencia,

$$T(1) = 1$$

$$T(n) = 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \text{ para } n > 1$$

Notese que para  $i = 1, 2, \dots, n-1$ , cada término  $T(i)$  aparece una vez como  $T(k)$  y una vez como  $T(n-k)$ , por tanto la anterior ecuación puede reescribirse como

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

Ahora bien, como  $T(1) \geq 1 = 2^0$ , inductivamente para  $n \geq 2$  tenemos

$$T(n) = 2 \cdot \sum_{i=1..n-1} 2^{i-1} + n \geq 2 \cdot \sum_{i=0..n-1} 2^i + n \geq 2(2^{n-1} - 1) + n = (2^n - 2) + n \geq 2^{n-1}$$

Con lo que se demuestra que el tiempo es al menos exponencial en  $n$ .

Lo que conviene aquí observar, para seguir adelante, es que tenemos un número relativamente bajo de subproblemas: un problema para cada elección de  $i$  y  $j$  que satisfaga  $1 \leq i \leq j \leq n$ , es decir en total  $C_{n,2} + n = O(n^2)$ . Un algoritmo recursivo puede tratar el mismo subproblema muchas veces en las diferentes ramas de su árbol de recursión, y este solapamiento es una de las propiedades interesantes para la aplicabilidad de la PD.

En lugar de calcular las soluciones de la recurrencia (2) de forma recursiva, resolveremos de acuerdo con la tercera etapa de la aplicación de la técnica de la PD. El siguiente algoritmo supone que la matriz  $A_i$  tiene dimensiones  $p_{i-1} \cdot p_i$ , para cualquier  $i = 1, 2, \dots, n$ . El input es una sucesión  $\{p_0, p_1, \dots, p_n\}$  de longitud  $n+1$ , es decir  $\text{leng}[p] = n+1$ . El procedimiento usa una tabla auxiliar  $m[1..n, 1..n]$  para ordenar los  $m[i, j]$  costos y una tabla auxiliar  $s[1..n, 1..n]$  que almacena que índice de  $k$  alcanza el costo óptimo al calcular  $m[i, j]$ .

Orden-Cadena-Matrices (p)

$n = \text{leng}[p] - 1$

For  $i = 1$  to  $n$  do  $m[i, i] = 0$

For  $l = 2$  to  $n$  do

    For  $i = 1$  to  $n - l + 1$  do

$j = i + l - 1$

$m[i, j] = \infty$

        For  $k = i$  to  $j - 1$  do

$q = m[i, k] + m[k+1, j] + p_{i-1} \cdot p_k \cdot p_j$

            If  $q < m[i, j]$

                Then  $m[i, j] = q$ ;  $s[i, j] = k$

Return  $m$  y  $s$ .

El algoritmo rellena la tabla  $m$  de un modo que corresponde a la solución del problema de la parentización en cadenas de matrices de longitudes crecientes. La ecuación (2) muestra que el costo  $m[i, j]$  de calcular el producto encadenado de  $j-i+1$  matrices depende solo de los costos de calcular los productos de las cadenas con menos de  $j-i+1$  matrices. Es decir, para  $k = i, i+1, \dots, j-1$ , la matriz  $A_{i..k}$  es un producto de  $k-i+1 < j-i+1$  matrices, y la matriz  $A_{k+1..j}$  es un producto de  $j-k < j-i+1$  matrices.

El algoritmo primero toma  $m[i, i] = 0$ ,  $i = 1, 2, \dots, n$  (los mínimos costos para las cadenas de longitud 1) y entonces, durante la primera ejecución del siguiente lazo, usa la recurrencia (2) para calcular  $m[i, i+1]$ ,  $i = 1, 2, \dots, n-1$  (los mínimos costos para las cadenas de longitud  $l = 2$ ). La segunda vez que pasa el lazo, calcula  $m[i, i+2]$ ,  $i = 1, 2, \dots, n-2$  (los mínimos costos para las cadenas de longitud  $l = 3$ ), y así sucesivamente.

Una simple inspección a la estructura encajada de los lazos en el anterior algoritmo demuestra que este tiene una eficiencia de  $O(n^3)$ , ya que hay 3 lazos en  $i$ ,  $j$  y  $k$  que, en el peor caso, pueden llegar a tomar el valor  $n$ . Por tanto deducimos que este algoritmo es mucho más eficiente que el anterior método en tiempo exponencial que enumeraba todas las posibles parentizaciones, y chequeaba cada una de ellas.

## Construccion de una solucion optimal

Aunque el anterior algoritmo determina el numero optimal de multiplicaciones necesarias para calcular un producto encajado de matrices, no explica como multiplicar las matrices. La siguiente etapa de la tecnica de la PD es la de la construccion de una solucion optimal a partir de la informacion calculada.

En nuestro caso, usamos la tabla  $s[1..n, 1..n]$  para determinar la mejor forma de multiplicar las matrices. Cada elemento  $s[i, j]$  almacena el valor  $k$  tal que divide optimalmente la parentizacion de  $A_{i-1} \times A_{i+1} \times \dots \times A_j$ . Por tanto, sabemos que la multiplicacion optimal de matrices final para calcular  $A_{1..n}$  es

$$A_{1..s[1,n]} \times A_{s[1,n]+1..n}$$

La anterior multiplicacion de matrices puede efectuarse recursivamente, puesto que  $s[1, s[1, n]]$  determina la ultima multiplicacion de matrices al calcular  $A$ , y  $s[s[1, n]+1, n]$  la ultima multiplicacion de matrices al calcular  $A_{1..s[1,n]}$ . El siguiente procedimiento recursivo calcula el producto encadenado de matrices  $A_{i..j}$  dadas las matrices de la cadena  $A = \{A_1, A_2, \dots, A_n\}$ , la tabla  $s$  calculada por el algoritmo Orden-Cadena-Matrices, y los indices  $i$  y  $j$ . El algoritmo usa el procedimiento clasico MULT  $(A, B)$  de multiplicacion de dos matrices.

Multiplica-Cadena-Matrices  $(A, s, i, j)$

If  $j > i$

Then  $X = \text{Multiplica-Cadena-Matrices}(A, s, i, s[i, j])$

$Y = \text{Multiplica-Cadena-Matrices}(A, s, s[i, j]+1, j)$

    Return MULT  $(X, Y)$

Else Return  $A_i$