PROGRAMACION DINAMICA

Manuel Ángel Rodríguez Segura

Javier Ramírez Pulido

Ejemplos y Explicaciones y acalaraciones

Introducción

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 desarrollarse recursivamente. En estos casos, la técnica conocida como Programación Dinámica (PD) proporciona algoritmos bastante eficientes. Como es el caso de los problemas de la mochila, del camino mínimo o el del viajante de comercio.

Para el primero (**el de la mochila**) 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 óptima 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 desde un vértice i a otro j en cierto grafo dirigido G, es decidiendo qué vértice en el camino debe ser el segundo, cual el tercero, cual el cuarto… hasta que alcanzáramos el vértice j. Una sucesión óptima de decisiones proporcionaría entonces el camino de longitud mínima que buscamos.

El nombre PD y la metodología que subyace en esa técnica lo ha tomado la Teoría de Algoritmos de la Teoría de Control, donde la PD es una técnica para obtener la política óptima para un problema similar, pero con n-1 etapas.

El lugar de la PD

El problema más importante en la Teoría de Control es el del **control óptimo**. Existen cuatro tipos de variables en este tipo de procesos:

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
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 óptimo de un proceso consiste en determinar la forma en que las variables producto (número 3) pueden mantenerse en los valores óptimos, 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 óptimo supone la optimización (tanto maximización como minimización) de ese índice de eficacia.

De forma general, un sistema físico de orden n-ésimo puede describirse en cualquier instante de tiempo t por medio de un conjunto finito de cantidades ($x_1(t)$, $x_2(t)$, $x_3(t)$, …, $x_n(t)$)

Estas cantidades se conocen como las **variables estado del sistema**, y constituyen las componentes del vector de estado del sistema: x(t). La relación que existe entre los cambios del parámetro tiempo y las variables de estado del sistema, se establece mediante una sencilla hipótesis, la cual establece 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 lleva asociada una fórmula matemática por medio de una ecuación diferencial vectorial:

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

con la condición inicial $x(0) = x_0$. En esta ecuación m(t) hace referencia al 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 cumplir la condición de que **g(m) ≤ 0** que refleja las restricciones que el control del sistema tenga. La función g es una función conocida de la señal de control. Así pues, el problema del diseño de un control óptimo puede plantearse de la siguiente forma: Dado un proceso que podemos controlar, determinar la ley de control, o sucesión, tal que el conjunto de índices de eficacia, previamente fijado, sea óptimo.

Es bastante frecuente que las variables de estado no sean accesibles en su plenitud para medirlas y observarlas. Entonces, la ley de control óptima 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. En definitiva, en el caso más general, tendremos que considerar la resolución de problemas de control óptima y de estimación óptima.

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. Entre los métodos más frecuentemente usados para el diseño de sistemas de control están los siguientes:

1.- El Cálculo de Variaciones

2.- El Principio de Máximo de Pontryaurin

3.- La Programación Dinámica

El propósito de todos es el mismo: encontrar la ley de control óptima tal que, dado el funcional del índice de eficacia del sistema, lo optimice.

El 1.- se adapta a la ecuación de Euler-Lagrange, el 2.- con el Principio de Hamilton y los 3.- con la teoría de Hamilton-Jacobi. En particular el 2.- emplea procedimientos más o menos directos del cálculo de variaciones, mientras que el 3.- (Programación dinámica), aún basándose en principios variacionales, usa relaciones de recurrencia.

Después de todo el coñazo de lo de arriba teórico, aquí hay una conclusión:

La PD al igual que el método de Divide y Vencerás, soluciona los problemas combinando las soluciones de subproblemas de menor tamaño, pero en contraste a él, la programación dinámica se aplica cuando los subproblemas no son independientes, esto quiere decir que los subproblemas comparten otros subproblemas. En PD cada subproblema se resuelve una sola vez y su solución se almacena en una tabla. Es por ello por lo que se aplica normalmente en problemas de optimización, pues pueden existir muchas soluciones cada una con sus valores y debemos encontrar cuál de ellos es el óptimo (ya sea mínimo o máximo).

Procesos de Decisión Multietápicos: Enfoque Funcional

En la práctica son abundantes los ejemplos de decisión multietápica. 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 euros en la compra de CTT y el resto se dedica para la compra de CCR. Lo que produce anualmente los CTT es una función de la cantidad inicial invertida, y, (1/y) e igual a g(y) el primer año. Por otro lado, lo que producen anualmente los CCR es una 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 a*y el primer año, siendo a un valor entre 0 y 1 ($0<a<1$). Por su parte, el valor del cambio de los CCR es una función igual a b*(x-y), siendo b un valor entre 0 y 1 ($0<b<1$). 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 que el problema sea más sencillo (como si lo fuera ya de primeras aes), 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 (de toda la vida como un problema de optimización). La función que recoge el beneficio durante el primer año es:

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

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

$f_1(x) = \text{Max}_{0\leq y\leq x}\{Y(x,y)\} = \text{Max}_{0\leq y\leq x}\{g(y) + h(x-y)\}$ es decir, el máximo de la función de arriba a resumidas cuentas, siendo un valor mayor o igual que 0 entre x e y.

En este caso, la recompensa máxima es función de la cantidad a invertir inicialmente. Ese máximo es trivial, pues cuando g y h se conocen, no tenemos más que derivar la función e igualar a 0 para evaluar los puntos críticos y conocer el valor que maximiza la función.

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) es la cantidad 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:

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

Y la recompensa máxima tras esos dos años es:

$f_2(x) = \text{Max}_{0\leq y\leq x,\ 0\leq y1\leq x1} \{g(y) + h(x-y) + f_1(x_1)\}$ con lo que sustituyendo tenemos para el beneficio máximo la siguiente función: $f_2(x) = \text{Max}_{0\leq y\leq x} \{g(y) + h(x-y) + f_1[ay + b(x-y)]\}$ (esta sustitución es hacer x_1 por $x_1=ay+b(x-y)$

(Todo ese rollo de $\text{Max}_{0\leq y\leq x}$ … significa que se busca un valor que maximice la función de tal forma que sea mayor o igual que 0 y que se encuentre entre x e y, al igual que con x_1 e y_1)

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 tras cuatro años, cinco, seis…

Por tanto, para un periodo de N años, la recompensa máxima vendrá dada por la función:

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

Esta es la ecuación funcional básica para este proceso de decisión N-etápico. Si te das cuenta, se comienza con $f_1(x)$ que sirve para calcular $f_2(x)$, esta a su vez sirve para sacar $f_3(x)$ y así sucesivamente. En cada etapa, se obtiene también la decisión óptima $y_j(x)$ a tomar al comienzo de la j-ésima etapa. El valor de y(x) que maximiza las funciones entre corchetes en la función de $f_N(x)$ (ay+b(x-y)) es la decisión óptima a tomar al comienzo de la operación del año N, suponiendo 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, un follón que no sabes donde te has metido.

El principio del Óptimo

Este principio nos permitirá obtener la ecuación funcional que describe un proceso de decisión multietápico como una consecuencia inmediata.

Vamos a considerar en primer lugar, **un proceso de decisión uni-etápico**. Sea x el vector de estado (es aquel que determina de manera única el estado de un sistema x(t) para cualquier tiempo $t > t_o$, una vez que se obtiene el estado en $t \sim t_o$ y se especifica la entrada u(t) para $t > t_o$) que caracteriza al sistema físico en cuestión en cualquier instante de tiempo.

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 a su vez produce una respuesta o recompensa: $R_1 = r(x^1, m_1)$. El problema consiste en elegir una decisión m_1 que maximice la respuesta (R_1). A la decisión m_1 se le llama **política uni-etápica**. En este problema está claro que el máximo de la respuesta viene dado por:

$f_1(x^1) = \text{Max } m_1 \{r(x^1, m_1)\}$ que como vemos es la decisión m_1 que maximiza la respuesta R_1

Esta decisión (m_1) que da el máximo valor de la salida, se llama **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 medio de $x^3 = g(x^2, m_2)$, esta transformación producirá una respuesta o recompensa $R_2 = r(x^1, m_1) + r(x^2, m_2)$, es decir, la suma de las dos transformaciones

En este caso el problema del diseño óptima consiste en elegir una sucesión de decisiones factibles m_1 y m_2 que maximicen la recompensa total (R_2). Este es un proceso bi-etápico en el que $r(x^1, m_1)$ es la salida que corresponde con la primera elección de una decisión, y $r(x^2, m_2)$ es la salida que corresponde con la segunda elección. Esta sucesión de decisiones $\{m_1, m_2\}$ se llama **política bi-etápica.** La máxima respuesta estará dada por:

$f_2(x^1) = \text{Max } m_1, m_2 \{r(x^1, m_1) + r(x^2, m_2)\}$ es decir el máximo de la suma de las salidas correspondientes con las dos decisiones m_1 y 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 óptima.**

La dificultad y la complejidad aumentan conforme lo hace el número de etapas del problema que se esté abordando. En general, **para un proceso N-etápico**, el problema consiste en elegir aquella política N-etápica $\{m_1, m_2 ..., mN\}$ tal que maximice la recompensa total,

$Rn = \sum j\ r(x^j, m_j),\ j \in J = \{1, 2, ..., N\}$

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 $x^3 = g(x^2, m_2)$,..., y finalmente desde x^(N-1) hasta x^N por x^N = g (x^(N-1) , m sub (N-1)). La recompensa máxima de este proceso N-etápico estará dada por:

$Fn(x^1) = \text{Max } \{mj\} \{\sum j\ r(x^j, m_j)\},\ j \in J$ es decir, el máximo de la recompensa total para un proceso N-etápico cuya fórmula está justo arriba.

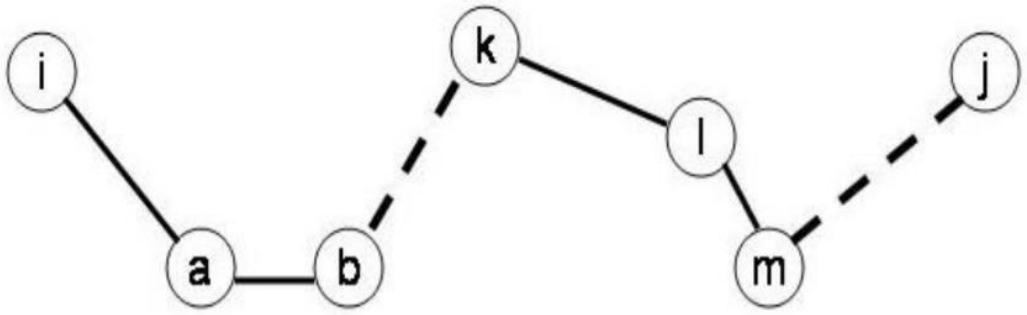
La política $\{m_j\}$ que determina $fn(x^1)$ es la **política óptima, o estrategia de control óptima.** En este caso, las m_j forman una política de control N-etápica.

Para resolver un problema de decisión óptima con un número elevado 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 del Óptimo de Bellman**, la cual establece lo siguiente:

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

En resumen, el Principio del Óptimo de Bellman lo que establece es que una política óptima solo puede estar formada por subpolíticas óptimas.

Vamos a analizar el Principio del Óptimo de una forma intuitiva. Para ello veremos la siguiente figura:



Esta imagen nos indica que la política óptima entre el estado i y el j, es la que pasa a través de los estados {a, b, k, l, m}. El Principio del Óptimo de Bellman lo que nos dice es que si por ejemplo nosotros tuviéramos que encontrar la política óptima entre el estado k y el j, ésta sería la {k, l, m, j}, o que la política óptima entre los estados b y k es la definida exactamente por el tramo discontinuo que las une.

La programación dinámica y, por tanto, el Principio Óptimo, que describe las propiedades básicas de las estrategias de control óptima, se basa en que para resolver un problema de decisión óptima especifico, el problema inicial esta encajado en una familia de problemas similares que son más fáciles de resolver.

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

$Rn = \sum j\ r\ (x_j, m_j),\ j \in J = \{1, 2, ..., N\}$

Ahora bien, de acuerdo con el Principio del Óptimo, la recompensa total del mismo proceso de decisión multietápico puede escribirse de la siguiente forma:

$Rn = r\ (x^1, m_1) + fn\text{-}1\ [g\ (x^1, m_1)]$

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

$fn(x^1) = Max\ m_1\ \{r\ (x^1, m_1) + fn\text{-}1[g\ (x^1, m_1)]\}$, a resumidas cuentas es el máximo m_1 de la recompensa máxima de arriba.

El problema es que esta última ecuación es válida solo cuando $N \geq 2$, pero para $N = 1$ la recompensa máxima viene dada por,

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

Aplicando este principio fundamental, un proceso de decisión N-etápico se reduce a una sucesión de N procesos de decisión uni-etápicos, con los que tenemos un procedimiento iterativo y sistemático para resolver eficientemente aquel problema.

El enfoque de la PD

El desarrollo de un algoritmo de PD corresponde con las siguientes etapas:

1) Caracterizar la estructura de una solución óptima

2) Definir recursivamente el valor de una solución óptima

3) Calcular el valor de una solución óptima en forma encajada de menos a más

4) Construir una solución óptima a partir de la información previamente calculada

Las etapas 1) y 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 óptima.

Vemos así que la PD, igual que la técnica Divide y Vencerás, resuelve problemas combinando las soluciones de subproblemas. De forma esquemática, los parecidos y diferencias entre estos dos enfoques pueden sintetizarse del siguiente modo:

PARECIDOS:

Programación Dinámica	**Divide y Vencerás**
Se progresa etapa por etapa con subproblemas que se diferencian en tamaño en una unidad. Se generan muchas subsucesiones de decisiones	Divide el problema en subproblemas independientes, los resuelve recursivamente y combina sus soluciones para obtener la solución total
Es aplicable cuando los sub-problemas obtenidos comparten subproblemas	Los sub-problemas no tienen ninguna característica común de tamaño, ni se sabe su número “a priori”
No se repiten cálculos porque los problemas están encajados	Como los problemas no tienen que ser independientes pueden repetirse cálculos
Siempre obtienen la solución óptima	Siempre se obtienen soluciones óptimas

DIFERENCIAS:

Programación Dinámica	**Divide y Vencerás**
Se progresa etapa por etapa con subproblemas que se diferencian entre sí en una unidad en sus tamaños	Se progresa etapa por etapa con sub-problemas que no tienen por qué coincidir en tamaño
Se generan muchas sub-sucesiones de decisiones	Solo se genera una sucesión de decisiones
Hay un gran uso de recursos (memoria)	La complejidad en tiempo suele ser baja
En cada etapa se comparan los resultados obtenidos con los precedentes: Siempre se obtiene una solución óptima	Como en cada etapa se selecciona lo mejor de lo posible sin tener en cuenta las decisiones precedentes no hay garantía de obtener el óptimo

Por último, 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 óptima, es intentarlo mediante el enfoque combinatorio, enumerativo o 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 esté usando como objetivo. La PD haciendo uso explícito del Principio del Óptimo 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 óptimas. De hecho, mientras que el número total de sucesiones de decisión diferentes es exponencial en el número 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 polinómica. Vamos a ilustrar la aplicación de la técnica de la PD con un ejemplo simple que nos servirá para describir las distintas fases de su desarrollo.

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 realizar esta subdivisión óptima se aplica la técnica de Programación Dinámica. Sea $f_n(c)$ el máximo producto que se pueda alcanzar, x el valor de la primera subdivisión, y (c-x) el valor de las (n-1) partes restantes. De acuerdo con el Principio del Óptimo ya comentado, la ecuación funcional que describe este problema de subdivisión óptima es:

$f_n(c) = \text{Max}_{0 \le x \le c} \{x f_{n-1}(c-x)\}$ Esta ecuación es válida para $n \ge 2$. Es obvio que cuando n=1,

$f_1(c) = c$ y $f_1(c-x) = c - x$

Así que para n=2

$f_2(c) = \text{Max}_{0\leq x\leq c} \{xf_1 (c-x)\} = \text{Max}_{0\leq x\leq c} \{x(c-x)\}$

y se encuentra que el valor de x (siendo x un valor entre 0 y c pudiendo ser tanto 0 como c) que maximiza ese producto es x = c/2.

Este resultado se debe a derivar e igualar a 0: $x(c-x)' = (c-x)+x(-1) = c-x-x=0 \rightarrow 2x=c \rightarrow x=c/2$

Así la política óptima, es decir, la subdivisión óptima para este proceso de decisión bietápico es la definida por $\{m_j\} = \{c/2, c/2\}$, siendo el valor máximo del producto: $f_2(c) = (c/2)^2$ (no explica el por qué de ese cuadrado, pero intuyo que al tratarse de un proceso de decisión BIetápico el valor máximo es el producto de ellos mismo, lo que es decir, al cuadrado)

Consideremos ahora el caso para 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 (la derivada e igualar a 0) se obtiene que el valor máximo para x es c/3. La subdivisión de la parte restante, 2c/3 (esto se debe a 1-(c/3)), constituye un proceso de decisión bietápico. Al igual que con el ejemplo de n=2, esta vez la subdivisión óptima para este proceso de decisión trietápico es la definida por $\{m_j\} = \{c/3, c/3, c/3\}$, siendo el valor máximo del producto: $f_3(c) = (c/3)^3$ (como es TRIetápico por eso es una 3-tupla y por tanto al cubo).

Cuando la cantidad de c se divide en cuatro partes, el producto máximo 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 valor máximo de x, c/4 (se saca derivando e igualando a 0 la función de arriba).

Ahora la subdivisión restante es de 3c/4 y constituye un proceso TRIetápico de subdivisión óptima c/4, c/4 y c/4, con lo que la subdivisión óptima de este problema TETRAetápico es

$\{m_j\} = \{c/4, c/4, c/4, c/4\}$, por lo que el valor máximo del producto es: $f_4(c) = (c/4)^4$

Todos estos análisis nos llevan a realizar una solución para este problema cuando n=k que tanto tú como yo esperamos bby, y es la siguiente:

$\{m_j\} = \{c/k, c/k, \ldots, c/k\}$ siendo el máximo valor del producto $f_k(c) = (c-k)^k$

En verdad todas las relaciones anteriores se pueden demostrar para cualquier k por medio de la inducción matemática. **De acuerdo con el Principio del Óptimo, se sigue que:**

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

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

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

Es así por lo que, por inducción matemática, la subdivisión óptima del proceso n-etápico puede obtenerse como

$\mathbf{\{m_j\} = \{c/n, c/n, \ldots, c/n\}}$

Con un valor máximo del producto $\mathbf{f_n(c) = (c/n)^n}$

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.

PROBLEMA DEL CAMINO MINIMO

G = (N,A) -> Un Grafo dirigido con un conjunto de nodos N y de arcos A. Son arcos con una longitud positiva dada por una matriz L. Se quiere determinar el camino con menor longitud entre cualquier par de nodos del grafo. Para comprobar que PD se puede aplicar, vemos sus fases:

a. **Naturaleza N-etápica de los problemas.** Para encontrar un camino mínimo desde el vértice i hasta el j se decide cual debe ser el segundo, tercero, cuarto, etc. hasta llegar a j. Una sucesión óptima de decisiones nos daría el camino mínimo que queremos.
b. **Comprobación del Principio del Óptimo.** Supongamos que $i, i_1, i_2, \ldots, i_k, j$ es el camino mínimo de i hasta j. Al decidir ir en segundo lugar a i_1, lo que toca es encontrar el camino más corto de i1 a j. Vamos a demostrar que $i_1, i_2, \ldots, i_k, j$ es el camino mínimo entre i1 y j usando la reducción a la absurda. Suponemos que no es verdad para llegar a una contradicción, por lo que imaginamos que hay un camino más corto que sea $i_1, r_1, r_2, \ldots, r_q, j.$ Entonces, el camino más corto desde i sería $i, i_1, r_1, r_2, \ldots, r, j,$ lo cual es mentira y verifica la hipótesis de partida y el Principio del Óptimo puede aplicarse a este problema.
c. **Construcción de una ecuación recurrente.** Desde el punto de vista adelantado, el Principio del Óptimo sería con un problema que se supone en un estado inicial de S_0. Hay que tomar n decisiones d_i (1 <= i <= n) y suponemos que $D_1 = \{r_1, r_2, \ldots, r_j\}$ es el conjunto de los valores que puede tomar d_1. S_i es el estado del problema después de tomar la decisión r_i (1 <= i <= j). Sea T_i una sucesión de decisiones óptimas con la que se ha llegado al estado S_i. Cuando se verifica el Principio del Óptimo, una sucesión de decisiones óptimas con respecto a S_0 es la mejor sucesión de soluciones r_iT_i con 1<=i<=j.

   A_i es el conjunto de vértices adyacentes al vértice i. Para cualquier vértice k que pertenezca a A_i ($k \in A_i$), T_k es el camino mínimo desde k hasta j. Por ello el camino más corto desde i hasta j es el más corto de los caminos $\{i, T_k / k \in A_i\}$.

Existe un vértice i que está directamente conectado con una arista a un conjunto de vértice al que llamamos A_i y cada vértice se llama k. Luego hay otro vértice j en algún punto del grafo, para el que el camino más corto desde k es el conjunto de decisiones óptimas cogidas desde este (T_k). Cada vértice vecino a i tiene su propio camino más corto hasta j (tantos T_k como k haya, y tantas k como vecinos tenga i). Por eso, el camino más corto de i a j es, de todos los caminos que hay formados por T_k + el camino desde i hasta ese k, el más corto.
El Principio del Óptimo también puede aplicarse a estados y decisiones intermedias. Imagina un camino mínimo desde i hasta j que sea *i, i_1, i_2,…, k, p_1, p_2, …, j* , los caminos de i hasta k y desde k hasta j también deben ser los más cortos posibles.

Para resolver el problema suponemos que los nodos están numerados desde 1 hasta n, N = {1,2,…,n} (N es el conjunto de nodos) y L la longitud de un arco. L(i,i) = 0, L(i,j)>=0 si i y j son distintos y L(i,j) = ∞ si no existe arco entre i y j. Para resolver el problema nos basamos en Floyd, construyendo una matriz D que da la longitud del camino mínimo entre cada par de nodos. En un principio asigna a D los valores de cada L y entonces hace n iteraciones. En cada iteración (por ejemplo la iteración número k) el algoritmo comprueba para cada par de nodos (i,j) si existe un camino que pase a través de k que sea menor que el actual que solo pasa por {1,2,…,k-1}. Tras la iteración k, D tiene la longitud de los caminos mínimos que usan como nodos intermedios los del conjunto {1,2,…,k}, por lo que después de n iteraciones ya se tiene la solución esperada. Si D es la matriz tras la iteración k:

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

Significa que, en la matriz, en la casilla que relaciona i con j, tras la iteración k, habrá un número que sea el mínimo entre el camino que había previamente en esa casilla y la suma de los valores desde i hasta k y desde k hasta j en la iteración anterior. En la matriz, en la iteración k-1, habrá un número en la posición i,k y en la posición k,j que va a contener la distancia desde i hasta k hasta j (la distancia desde i hasta j pasando por k pero expresada en dos partes) y en la casilla i,j la longitud de un camino mínimo que los une. Si ese número es más pequeño que la suma de los dos anteriores (que el camino más corto entre ellos es más pequeño que el que pasa por k) será escogido como distancia mínima, si no, será el otro valor.
Se hace uso del Principio del Óptimo para la longitud del camino más corto que pasa por k (una sola vez). Usando el algoritmo de FLOYD:

```
Begin
    For i=1 to n do        //Para recorrer las filas de la matriz D
        For j=1 to n       //Recorre las columnas de D
            D[i,j]=L[i,j]  //Inicialmente mete los valores de L[i,j] que
        representan la distancia del arco entre ambos.
```

For i=1 to n

D[i,i]=0 //Pone a 0 la diagonal porque caminos entre un nodo y él mismo puede haber varios, pero el mínimo siempre va a valer 0 pase lo que pase.

For k=1 to n do //Para cada nodo recorre toda la matriz entera en busca de caminos mínimos

For i=1 to n do //Se mueve entre las filas de la matriz

For j=1 to n do //Se mueve entre las columnas

If D[i,k] + D[k,j] < D[i,j] //Si el valor del camino pasando por k es menor que el que va directamente, entra en el if

Then D[i,j] = D[i,k]+D[k,j] //Dentro del if es donde introduce ese valor. Si no hubiese entrado, sería porque pasar por K no es más corto que el directo que había en esa posición y entonces se deja.

End

Obviamente, su eficiencia es $0(n^3)$. Usando Dijkstra para empezar en cada iteración por un nodo diferente, sería hecho n veces con una eficiencia de $O(n^2)$, por lo que sería igualmente $n*n^2$, una eficiencia de n^3, pero con Floyd es más simple.

Si queremos saber por dónde va el camino más corto y no solo su longitud, se crea una matriz P que contenga el último nodo que hizo que se modificase D(i,j). Si pone 0 es porque no se modificó nunca, por lo que ningún nodo es camino más corto que el que ya había. Ahora el algoritmo quedaría así:

Begin

For i=1 to n do //Para recorrer las filas de la matriz D

For j=1 to n do begin //Recorre las columnas de D

D[i,j]=L[i,j] //Inicialmente mete los valores de L[i,j] que representan la distancia del arco entre ambos.

P[i,j]=0 //Inicializa a 0 también cada una de las posiciones de P.

End

For i=1 to n

D[i,i]=0 //Pone a 0 la diagonal porque caminos entre un nodo y él mismo puede haber varios, pero el mínimo siempre va a valer 0 pase lo que pase.

For k=1 to n do //Para cada nodo recorre toda la matriz entera en busca de caminos mínimos

For i=1 to n do //Se mueve entre las filas de la matriz

For j=1 to n do //Se mueve entre las columnas

If D[i,k] + D[k,j] < D[i,j] then begin//Si el valor del camino pasando por k es menor que el que va directamente, entra en el if

D[i,j] = D[i,k]+D[k,j] //Dentro del if es donde introduce ese valor. Si no hubiese entrado, sería porque pasar por K no es más corto que el directo que había en esa posición y entonces se deja.

P[i,j] = k //Cambia en la posición i,j el valor que había y mete el número del nodo que modificó esta cifra, en este caso k, que es por donde pasa el camino más corto (si no, no habría hecho cambiar ese valor)

End

End

Para obtener los vértices intermedios en el camino mínimo entre i y j hacemos el Procedimiento de los Nodos:

Begin

K:=P[i,j]; //Mete en k el valor que se encuentre en la posición i,j dentro de la matriz k (recuerdo que contiene cual fue el último vértice que modificó el camino entre i,j)

If k=0 then return; //Si es 0 es porque no se modificó por lo que la distancia era la directa entre i y j, sin nodos intermediarios

Path(i,k); //Si no ha entrado en if y no hace el return, pasa por aquí y muestra el camino desde i hasta k (no lo pone en ningún lado, pero tengo la sospecha de que es recursivo, el "procedimiento nodos" puede que se llame Path también (lo que cumple con el objetivo de la función que es mostrar el camino), entonces lo que hace es mostrar el camino desde i hasta k pero al entrar en la función, verá cual fue el último nodo que modificó la distancia entre ellos y si es distinta a 0 volverá a entrar hasta que se encuentre con el camino es 0(haber llegado al origen del camino, desde i hasta i o al camino mínimo entre dos nodos que es L(i,j)) y a raíz de ahí mostrará los vértices de los caminos hasta ellos mismos)

Writeln(k); //Muestra k tras mostrar el camino de i hasta k (otra prueba de lo que pega la recursividad aquí, tras hacer todas las iteraciones recursivas posibles y mostrar los vértices desde i hasta k, ahora muestra k en la secuencia)

Path(k,j); //Igualmente muestra los vértices restantes desde k hasta j

End

Para entender el procedimiento recursivo que yo interpreto, en la carpeta PROCEDIMIENTO DE LOS NODOS pulsa la primera imagen y disfruta la experiencia.

EL PROBLEMA DE LA MOCHILA 0-1

Hay n objetos que se pueden fraccionar y una mochila. El objeto i tiene peso w_i y una fracción x_i ($0<=x_i<=1$) que produce un beneficio p_ix_i. Se quiere llenar la mochila, con capacidad M, de manera que el beneficio sea el máximo:

Máximo posible: $\sum_{1<=i<=n} p_ix_i$ (Que la suma de los beneficios de cada objeto sea el máximo que se pueda obtener)

Sabiendo que:

$\sum_{1<=i<=n} w_ix_i <= M$ (Que la suma de los pesos de cada objeto sea menor que la capacidad)

$0<=x_i<=1$ (La porción que cojas de un objeto no puede ser negativa ni superior al total que es 1)

$1<=i<=n$ (El identificador del objeto está entre 1 (porque no puede ser negativo ni 0) y n porque no puede ser el objeto 5 habiendo 4 xd)

Los pesos son naturales (1,2,3,…) y los beneficios reales no negativos (>=0). Una variante es aquella en la que x_i (la porción que coges) solo vale 0 o 1 (cogerlo entero o no cogerlo directamente). Entonces:

Máximo posible: $\sum_{1<=i<=n} p_ix_i$ (Que la suma de los beneficios de cada objeto sea el máximo que se pueda obtener)

Sabiendo que:

$\sum_{1<=i<=n} w_ix_i <= M$ (Que la suma de los pesos de cada objeto sea menor que la capacidad)

$x_i = 0, 1$ (La porción que cojas de un objeto es 0 o 1)

$1<=i<=n$ (El identificador del objeto está entre 1 (porque no puede ser negativo ni 0) y n porque no puede ser el objeto 5 habiendo 4 xd)

a. **Naturaleza N-etápica de los problemas.** La solución es el resultado de varias decisiones. Se decidiría la x_i de cada objeto (x_1, x_2, …). Una sucesión óptima de decisiones maximiza la función objetivo.

b. **Comprobación del Principio del Óptimo.** Con las restricciones anteriores se entiende que se puede llamar al problema como Mochila(1,n,M) (El primer argumento es el número del objeto del que se parte a tomar decisiones, n el número de elementos y M la capacidad). Reducimos al absurdo. $y_1, y_2, \ldots, y_n$ es una sucesión óptima de 0 y 1 para las porciones $x_1, x_2, \ldots, x_n$.

· Si $y_1=0$, entonces $y_2, \ldots, y_n$ es una sucesión óptima para el problema Mochila(2, n, M). Si no se cumple eso, $y_1, y_2, \ldots, y_n$ no puede ser sucesión óptima de Mochila(1,n,M).

· Si $y_1=1$, entonces $y_2, \ldots, y_n$ es una sucesión óptima para el problema Mochila(2, n, M-w_i). Si no lo fuera, entonces habría otra sucesión de valores 0 y 1, tal que $z_2, z_3, \ldots, z_n$ y:

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

y

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

entonces la sucesión $y_1, z_2, z_3, \ldots, z_n$ es una mejor sucesión para el problema de partida, lo que es contradictorio. Se aplica el Principio del Óptimo.

c. **Construcción de una ecuación recurrente.** Sea $g_j(y)$ el valor de una solución óptima del problema Mochila(j+1,n,y). Claramente $g_0(M)$ es una solución óptima del problema Mochila(1,n,M). Las posibles decisiones para x_i son 0 o 1 ($D_1=\{0,1\}$). Por el Principio del Óptimo se sigue que:

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

$g_1(M)$ es la mejor solución de Mochila(2,n,M). Significa que la capacidad sigue siendo M y ahora miramos desde el objeto 2 (entonces del 1 se ha cogido $x_i=0$)

$g_1(M\text{-}w_1)+p_1$ es la mejor solución de Mochila(2,n,M-w_i). Significa que la capacidad ahora ha disminuido en el peso del objeto 1 y sumamos la productividad entera (p_1) del objeto 1 (por lo que el objeto 1 se ha cogido entero)

Pues el máximo de estos, planteando coger el primer objeto o no, es la solución.

Como en el caso del camino mínimo, el Principio del Óptimo también se aplica a estados intermedios. Si $y_1, y_2, \ldots, y_n$ es una solución óptima del problema Mochila(1,n,M), entonces para cada j (1<=j<=n) se cumple que $y_1, \ldots, y_j$ e $y_{j+1}, \ldots, y_n$ deben ser soluciones óptimas de los problemas:

$$\text{Mochila } (1, j, \sum_{1 \le i \le j} w_i y_i)$$

y

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

(Ambos problemas son el problema original pero partido en dos partes, uno desde 1 hasta j y otro desde j+1 hasta n)

Por lo que podemos generalizar* a:

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

Si el aplica el Principio del Óptimo, se obtiene una recurrencia como la anterior. Los algoritmos de PD las resuelven para la solución del algoritmo que estamos considerando. Sabiendo que $g_n(y) = 0$ para todo y (porque eso significaría el máximo entre Mochila(n+1,n,y) y Mochila(n+1,n,y-w_{n+1})+p_n y como hay n objetos, vas a coger 0 del objeto n+1). A partir de esta se obtiene $g_{n+1}(y)$ usando en * i = n-1. Entonces usando $g_{n-1}(y)$ podemos obtener $g_{n-2}(y)$ y así hasta determinar $g_1(y)$ y finalmente $g_0(M)$ usando i=0 en *. Igual que se ha usado el enfoque adelantado, se podría haber hecho con el enfoque atrasado, de forma que para la decisión x_i se miraría en las decisiones óptimas para x_1, …, x_{i-1}, mirando hacia atrás en la sucesión y no hacia delante.

ALGORITMO MOCHILA1(w,p,M)

Begin

Devuelve g(n,M)

End

FUNCION g(j,c)

(No lo dice así que lo intuyo: j=número del elemento y c=capacidad)

Begin

Si j=0 entonces devuelve 0 //Si es el elemento 0 has terminado porque no corresponde a ninguno, estos empiezan en 1 (Caso Base)

Si no

Si c<w[j] //Si la capacidad es menor que lo que pesa el objeto, no lo va a coger

Entonces devuelve g(j-1,c) //No lo coge y llama a comprobar el objeto anterior recursivamente hasta llegar al 0 (Caso Base)

Si no //Si entra aquí es porque el objeto si que cabía

Si g(j-1,c) >= g(j-1, c-w[j] + p[j]) //Es una forma de implementar en código la ecuación de * que elige entre el máximo de las dos opciones, coger el elemento o no

Entonces devuelve g(j-1, c) //Muestra este si este resultaba ser más grande o igual

Si no

Devuelve g(j-1, c-w[j]) + p[j] //Muestra este si la condición anterior no se da.

End

Se ve que este esquema algorítmico no es muy eficiente porque cada problema de tamaño n lo reduce en 2 de n-1, y estos en otros dos sucesivamente, resultando en un algoritmo exponencial. Sin embargo, al final no son tantos subproblemas a resolver. Los parámetros de g pueden tomar a lo sumo n y M valores respectivamente, (como máximo la n puede ser el número de elementos, no más, e igual con M, ese valor no va a pasar de la capacidad de la mochila) por lo que solamente hay nM problemas diferentes. Para evitar repetir muchas veces los mismos cálculos con la recursividad, se deben almacenar los datos en una matriz de nxM elementos.

Ejemplo: Si hay 3 elementos, la capacidad es 15, los beneficios son (p_1, p_2, p_3) = (38, 40, 24) y los pesos son (w_1, w_2, w_3) = (9,6,5). Con la expresión de * nos quedaría una tabla así:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_1 = 9$	0	0	0	0	0	0	0	0	0	38	38	38	38	38	38	38
$p_2 = 6$	0	0	0	0	0	0	40	40	40	40	40	40	40	40	40	78
$p_3 = 5$	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	78

Que también se puede obtener con el siguiente algoritmo de O(nM). Cada fila significa el beneficio que has cogido de cada elemento sumado a lo que llevas. Por eso, en la columna 15 cabe el objeto 1 con peso 9 por lo que se coge el beneficio de 38, luego cabe también el objeto 2 con peso 6 (en total ya va peso 15) así que se coge su beneficio de 40 y ya va beneficio 78 y cuando llega el objeto 3 no cabe, por eso se coge 0 de ese objeto y no se suma nada a los 78. En el ejemplo de la columna 9 se comprueba que el objeto 1 con peso 9 cabe y se cogen esos 38 de beneficio, pero a continuación se observa que el objeto 2 con peso 6 también cabe (si no hubiese nada dentro) y su beneficio es mayor, por lo que sustituye el objeto 1 por el 2 y ya el 3 no

cabe. Última columna que explico xd: en la 11 se comprueba que cabe el 9 y se cogen esos 38, pero si sigues adelante no cabe meter el objeto 2 junto al 1, pero caber cabe solo y tiene más beneficio, por lo que se saca el anterior y se mete este. Se comprueba el objeto 3 que pesa 5 y cabe junto al 6 así que lo mete y suma su beneficio (40+24=64).

ALGORITMO MOCHILA(w,p,M)

Begin

For c = 0 hasta M hacer g[0,c] = 0 //Rellena una fila adicional con 0 encima de la fila del primer objeto (se está rellenando la fila 0 y la del primer objeto va a ser la 1)

For j = 1 hasta n hacer g[j,0] = 0 //Con esto también llena la primera columna entera de 0 porque es la columna en la que la capacidad es 0 y no cabría ningún elemento, es tontería comprobar.

For j = 1 hasta n hacer //Bucle que va a ir avanzando en las filas (una iteración por cada elemento)

For c = 1 hasta M hacer //Bucle que va a ir avanzando por las columnas (capacidad de la mochila)

If c < w[j] entonces g[j,c]=g[j-1,c] //Si la capacidad de la mochila es más pequeña que lo que pesa ese objeto, sabes ya que no cabe así que no lo metes, por lo que no modificas el beneficio que va dentro, por lo que el beneficio es el mismo que antes de compararlo (valor almacenado en la fila anterior) y por eso metes ese valor de nuevo

Else //Si cabe ese objeto

If g[j-1,c]>=g[j-1, c-w[j] + p[j]] entonces g[j,c]=g[j-1,c] //Si el beneficio que había antes es mayor o igual que el beneficio que aporta meter este objeto j, no lo mete directamente, copia el valor de encima

Else g[j,c]>=g[j-1, c-w[j] + p[j]] //Si no ocurre lo anterior es porque conviene meter ese nuevo objeto en la mochila

Fin

PROBLEMA DEL VIAJANTE DE COMERCIO

Habiendo un grafo con longitudes positivas en sus arcos, se quiere encontrar el circuito más corto posible que empiece y termine en un mismo nodo, recorriéndolos todos una sola vez con longitud mínima.

G = (N,A) es un grafo dirigido. Tomamos N={1,2,…,n} y la longitud entre los arcos es $L_{i,j}$, (L(i,i)=0, L(i,j)>=0 si i y j son diferentes y L(i,j)=∞ si no hay arco (i,j)). Se supone que el circuito empieza y termina en el 1, por lo que está formado por el arco (1,j) y un camino desde j a 1 pasando por cualquier nodo de N menos 1 y j de nuevo. Si la distancia es la mínima, ese camino de j a 1 es la solución y vale el Principio del Óptimo.

Consideramos un conjunto de nodos S (un subconjunto de N sin contar el 1) y un nodo extra i (que pertenece a N sin S, o sea, que sea del conjunto de nodos del grafo pero que no pertenezca a S) que puede valer 1 solo si S es un subconjunto de N que contiene todos los nodos de este menos el 1 (S = N – {1}). Para cada índice i se define el valor g(i,S) como la longitud del camino más corto desde el nodo i hasta 1 que pasa exactamente una vez por cada nodo de S (g(i,S) es la distancia que mide el camino que sale de i y pasando por todos los nodos de S llega a 1). La longitud de un circuito óptimo viene dada por g(1, N – {1}). Por el Principio del Óptimo vemos que:

$$g(1, N-\{1\}) = \text{Min}_{2\le j\le n} [L_{1j} + g(j, N - \{1,j\})] \qquad (5)$$

La j es cualquier nodo entre 2 (no puede pasar por 1 de nuevo) y n, y por cada j habrá un camino que sea la distancia entre 1 y j (L_{1j}) y luego el camino más corto desde j pasando por todos los nodos de N menos j y 1 de nuevo. Por eso, de todas esas opciones posibles, el circuito más corto desde 1 es el formado por el camino más corto posible de 1 a j y el camino más corto posible que parte de j y pasa por todos los restantes.

Si i no es 1 significa que S no es igual a N – {1}, el nodo 1 no estará obligado a ser cogido por lo que tampoco i pertenece a S y el conjunto S no será vacío. Así queda que:

$$g(i,S) = \text{Min}_{j\in S} [L_{ij} + g(j, S - \{j\})] \qquad (6)$$

(Es como una generalización del (5) porque es conocer el circuito mínimo que parte en i y pasa por los todos los nodos de S hasta llegar a 1 (si i=1 y S = N – {1} es el mismo caso que (5)) pero realmente es la explicación de lo que significa la parte final del (5))

Y también,

$$g(i,\varnothing) = L_{i1} \ , \ i = 2,3,\ldots n$$

(Quiere decir que, si el conjunto de nodos por el que va a pasar es vacío, no pasa por ningún nodo, por lo que la distancia entre i y 1 es la distancia directa que haya entre estos dos (L_{i1}))

Se puede aplicar el (6) en todos los conjuntos S que tengan un solo elemento que no sea el 1 (con esto sabríamos el camino más corto que pasa por un nodo), luego calculamos g en todos los S que tengan 2 nodos (ninguno sea 1), y así sucesivamente. Cuando se sabe el valor de g[j, N – {1,j}] para cada j (se conoce cuál es la distancia más corta desde cada j hasta 1), podemos usar el (5) para saber g(1, N – {1}) y resolver el problema. (Sabríamos ya de todas las j que hay, cuál tiene el camino más corto hasta 1, ahora solo queda saber cuál es el camino más corto contando además con la distancia entre el 1 y cada j).

El tiempo de este algoritmo puede calcularse de la siguiente manera:

· Calcular g(j,ø) supone consultar n-1 veces una tabla (porque el 1 no se comprueba. Lo que hace es mirar la distancia entre cada j y 1)

· Calcular todas las g(i,S) tales que 1<= #S=k <= n-2 (#S = cardinal de S (número de elementos que tiene) por lo que dice calcular la distancia mínima desde i hasta 1 pasando por todos los nodos de S siempre que S tenga 1 elemento o más y menos o los mismos elementos que n-2 (porque ni coge el 1 ni coge i)) supone realizar

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

adiciones. (n-1 = posibles valores que puede tomar la i (porque no puede valer 1), $C_{n-2,k}$ son todas las combinaciones que pueden realizarse de n-2 elementos tomados de k en k (¿?) y k los valores que puede tomar j)

· Calcular g(1, N – {1}) implica n-1 adiciones.

Conociendo estas operaciones se puede tener una referencia para calcular la eficiencia del algoritmo. De esta forma el tiempo que gasta el algoritmo en cálculos es:

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

La suma de (n-1) que tardaba en calcular cada g(j,ø) del primer · de los 3 anteriores, más (n-1) que tardaba en calcular g(1, N – {1}) del tercer · de los 3 anteriores (esas dos hacen ya 2(n-1) en la fórmula) y lo que tarda en calcular todas las g(i,S) del segundo · de los 3 anteriores.

Ya que

$$\sum_{k=1..r} k\times C_{r,k} = r2^{r-1}$$ **

(Maybe esta explicación es innecesaria y vale con tragárselo, pero si ayuda pues bueno va:

En la sumatoria, el valor que va incrementando es el de K, asi que n-1 va a valer lo mismo en cada sumando y se puede sacar como factor común quedando:

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

Ese (n-1) puede volver a sacarse como factor común:

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

Siguiendo ** :

$$\sum_{k=1..(n-2)} kxC_{n-2,k} = (n-2)2^{(n-2)-1}$$

Entonces en la expresión general queda:

$$O[(n-1)(2 + (n-2)2^{(n-2)-1})] = O[(n-1)(2+(n-2)2^{n-3})]$$

Desarrollo:

$$O[(n-1)(2+n2^{n-3} - 2^{n-2})] = O[2n - 2 + n^2 2^{n-3} - n2^{n-3} - n2^{n-2} + 2^{n-2}] =$$

$$= O[2n - 2 + (1/2^3)n^2 2^n - (1/2^3)n2^n - (1/2^2)n2^n + (1/2^2)2^n]$$

Que obviamente tiene la eficiencia del sumando más tocho que es $O(n^2 2^n)$.

Así saque un 9,5 en cálculo. Besos)

Ese tiempo es considerable pero el método directo clásico sería O(n!) que para tamaños de n pequeño parece mejor, pero cuando va creciendo n se ve que no (ejemplo en la tabla siguiente):

	Tiempo	Tiempo
N	Método directo	PD
	n!	$n^2 2^n$
5	120	800
10	3.628.800	102.400
15	$1.31x10^{12}$	7.372.800
20	$2.43x10^{18}$	419.430.400

Para el tamaño de N = 20, el método directo es una cantidad de microsegundos que supera los 77 mil años, el PD equivale a menos de 7 minutos.

Ejemplo para fijar ideas: Consideramos un grafo con la siguiente matriz de distancias:

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

(La primera fila y la primera columna son la 1 y no la 0)

g(x, ∅) = Distancias más cortas (única distancia, distancia directa) desde x a 1:

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

Usando * obtenemos ahora los caminos desde i hasta 1 que pasan por un nodo:

g(2,{3})(la distancia más corta de los caminos desde 2 hasta 1 que pasan por 3) = c_{23} (camino desde 2 hasta 3) + $g(3, \varnothing)$ (distancia desde 3 hasta 1) = 9 + 6 = 15;

g(3,{2}) = 13 + 5 = 18; g(4, {2}) = 8 + 5 = 13; g(2, {4}) = 18;

g(3, {4}) = 20; g(4,{3}) = 15

A continuación se calcula g(i,S) para conjuntos S de dos elementos (caminos desde i hasta 1 que pasan por dos nodos en vez de uno) en los que i ≠ 1 y 1 e i no pertenecen a S (porque 1 es de donde se parte y a donde se llega originalmente en el problema, por eso no se puede pasar por él (no pertenece a S) ni puede ser el que evaluamos de partida porque significaría calcular el camino desde él hasta él y lo que queremos ver son los posibles caminos intermedios. Tampoco queremos que i (el que evaluamos de partida para llegar hasta 1) esté en S porque si se parte de él, no se puede pasar por él). Por tanto, tras esas restricciones:

Salir de 2, pasar por dos nodos y llegar a 1 sin pasar por 2 ni 1 solo deja la opción de salir de 2 y pasar por 3 y 4:

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

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

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

y ya por último el camino que sale de 1 y pasa por 2,3,4 llegando a 1:

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

MULTIPLICACIÓN ENCADENADA DE MATRICES

Otro ejemplo de PD es el algoritmo para la multiplicación encadenada de matrices. Suponemos una sucesión de n matrices $\{A_1, A_2, \ldots, A_n\}$ que se quieren multiplicar de forma $A_1 \times A_2 \times \ldots \times A_n$. Parentizar es constituir un producto de matrices por una sola matriz o el producto parentizado de dos matrices cerrado por paréntesis. Como la multiplicación de matrices es asociativa, todas las parentizaciones producen el mismo resultado. Por ejemplo, el producto de $A_1xA_2xA_3Xa_4$ puede parentizarse completamente de 5 formas:

$$(A_1\times(A_2\times(A_3\times A_4))),$$
$$(A_1\times(A_2\times A_3)\times A_4),$$
$$((A_1\times A_2)\times(A_3\times A_4)),$$
$$((A_1\times(A_2\times A_3))\times A_4), \text{ y}$$
$$(((A_1\times A_2)\times A_3)\times A_4).$$

La forma en la que parenticemos puede afectar al costo de evaluar el producto. Si A es una matriz de pxq y B una de qxr, el algoritmo estándar consumirá un tiempo de pxqxr.

Para ver los diferentes costos asociados a las distintas parentizaciones de un producto de matrices se considera el siguiente ejemplo:

$\{A_1, A_2, A_3\}$ de dimensiones 10x100, 100x5 y 5x50 respectivamente. Si multiplicamos siguiendo $((A_1\ A_2)\ A_3)$ realizamos 10x100x5 = 5000 multiplicaciones para el producto de (A_1A_2) (quedando una 10x5) que suma 10x5x50 = 2500 multiplicaciones al sumar esa resultante por A_3, quedando un total de 7500 multiplicaciones.

Si en vez de ese, se hace el producto $(A_1\ (A_2\ A_3))$ realizamos 100x5x50 = 25000 multiplicaciones para (A_2A_3) (quedando una 100x50) y multiplicamos A_1 por esa resultante siendo 10x100x50 = 50000 multiplicaciones, un total de 75000 multiplicaciones, 10 veces más que la primera parentización.

Por ello, el problema de la multiplicación encadenada de matrices puede plantearse como dada una cadena de n matrices $\{A_1, A_2, \ldots, A_n\}$, donde para i = 1,2,…,n la matriz A_i tiene dimensión $p_{i-1}xp_i$, se busca parentizar completamente el producto de las matrices $A_1 \times A_2 \times \ldots \times A_n$ de tal forma que se minimice el número total de multiplicaciones escalares a hacer.

Hay dos formas de hacerlo:

1. Insertar los paréntesis de todas las formas posibles diferentes
2. Calcular para cada una el número de multiplicaciones escalares requeridas.

Vemos que la 1 no es muy eficiente. Llamemos P(n) al número de parentizaciones posibles con n matrices. Como podemos dividir una sucesión de n matrices en dos partes (las k primeras y las k+1 siguientes (creo que esto está mal porque si hay 7 por ejemplo, ¿quiere decir que puedes dividirla en las 4 (k) primeras y en las 5 (k+1)

siguientes? Eso son 9 matrices xd. Creo que sería en las k primeras y en las n-k siguientes (7 se dividen en las 4 (k) primeras y las 3 (n-k = 7-4) siguientes. Aún así, me limito a poner lo que dice porque yo que sabré)) para cualquier k = 1,2,…,n-1 y entonces parentizar las dos subsecciones resultantes independientemente, obtenemos la recurrencia:

$$P(n) = 1 \quad \text{si } n = 1$$
$$= \sum_{k=1..n-1} P(k) \times P(n-k) \quad \text{si } n \geq 2$$

(Si solo hay una matriz, P(n) (número de parentizaciones posibles) es solo 1, si hay 2 o más se puede ir partiendo por la mitad. Con dos matrices es 1 porque la suma desde 1 hasta 1 es realizar una vez P(1)xP(1) = 1x1 = 1, pero por ejemplo con n=9 sería la suma desde 1 hasta 8 de P(1)xP(7)+ P(2)xP(6)+ P(3)xP(5)+…+ P(7)xP(1), siendo P(7) la suma desde 1 hasta 6 de P(1)xP(6)+ P(2)xP(5)+…+ P(6)xP(1), siendo P(6) la suma de 1 a 5 de P(1)xP(5)+ P(2)xP(4)+…+ P(5)xP(1), siendo P(5)…, siendo P(4)…, siendo P(3)… hasta siendo P(1)=1 y ahí a resolver todas las anteriores)

La solución de esta ecuación coincide con la sucesión de los Número de Catalan:

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

Donde

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

Que es $\Omega(4^n/n^{3/2})$, por lo que el número de soluciones es exponencial en n y por tanto el método de fuerza bruta es una estrategia pobre para conocer la parentización.

Por tanto, intentamos basarnos en PD para obtener la solución óptima. La primera etapa de PD es comprobar la naturaleza n-etápica del problema, pero en este caso es evidente porque podemos asociar cada etapa al producto de una nueva matriz.

Con esto pasamos a verificar el Principio del Óptimo antes de definir una ecuación recurrente. Consideramos $A_{i..j}$ la matriz que resulta de $A_i \times A_{i+1} \times \ldots \times A_j$. Una parentización óptima del producto $A_1 \times A_2 \times \ldots \times A_n$ divide el producto entre A_k y A_{k+1} para algún entero k en el rango 1<=k<=n (A_k va a ser la matriz que resulte de multiplicar la primera parte, sea cual sea y A_{k+1} la que resulte de multiplicar el resto de matrices). Es decir, para algún valor de k primero calculamos las matrices $A_{i..k}$ y luego $A_{k+1..n}$ y tras esto, la multiplicación de ambas para tener $A_{1..n}$. El costo de esto es el costo de calcular $A_{i..k}$, más el costo de calcular $A_{k+1..n}$ y el costo de multiplicar ambas.

La clave está en que la primera subcadena $A_1 \times \ldots \times A_k$ dentro de la óptima $A_1 \times A_2 \times \ldots \times A_n$ debe ser óptima porque si hubiera una forma menos costosa de hacer $A_1 \times \ldots \times A_k$, sustituyendo esa cadena mejor dentro de $A_1 \times A_2 \times \ldots \times A_n$ la haríamos menos costosa, una contradicción. Lo mismo ocurre con la segunda cadena desde $A_{k+}1 \times \ldots \times A_n$. Así se cumple el Principio del Óptimo de Bellman. A continuación se define recursivamente el valor de una solución óptima en término de las soluciones óptimas de los

subproblemas. En este caso, los subproblemas son determinar el mínimo costo de una parentizacion de A_i x A_{i+1} x … x A_j, con 1<=i<=j<=n. m[i,j] es el mínimo número de multiplicaciones escalares necesarias para calcular la matriz $A_{i..j}$, por tanto, la forma menos costosa de calcular $A_{1..n}$ debería ser m[1,n].

Se puede definir m[i,j] recursivamente así: Si i=j, la cadena consiste de un solo elemento $A_{i..j} = A_i$, por lo que no hay que hacer multiplicación para calcularla. Así, m[i,i]=0 para i=1,2,…,n.

Para calcular m[i,j] cuando i<j, usamos la estructura de la solución óptima a partir de la primera etapa. Supongamos que la parentización óptima para A_i x … x A_j toma como punto de división A_k y A_{k+1} con i<=k<=j. Entonces m[i,j] es el costo mínimo de calcular $A_{i..k}$ y $A_{k+1..j}$ más el costo de multiplicar ambas, que supone hacer $p_{i-1}*p_k*p_j$ multiplicaciones, dando lugar a:

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

Esta ecuación da por hecho que conocemos k, pero no es así realmente. Hay solo j-i valores posibles para k (k puede ser i, i+1, … hasta j-1). Como la parentización óptima debe usar uno de estos valores, solo necesitamos saber cuál. Así nuestra recursiva queda así mejor:

$$m[i,j] = \begin{cases} \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j\} & \text{si } i < j \quad (5) \\ 0 & \text{si } i = j \end{cases}$$

Donde m[i,k] = es el costo óptimo de A_i x … x A_k; m[k+1, j] = costo óptimo de A_{k+1} x … x A_j ; $p_{i-1}*p_k*p_j$ = (A_i x … x A_k)x(A_{k+1} x … x A_j).

Definimos también s[i,j] como el valor de k que divide el producto A_i x A_{i+1} x … x A_j para la parentización óptima tal que:

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

Esto hace fácil calcular m[1,n] asociado a A_1 x A_2 x … x A_n. El algoritmo es exponencial y queda así:

MATRICES-ENCADENADAS-RECURSIVO(p,i,j**)**

//Tampoco lo dice, pero esto de suponer cosas ya es costumbre: p contiene las dimensiones de todas las matrices, i es la primera matriz de la sucesión que se quiere multiplicar y j es la última.

If i = j then return 0 //Si i = j es porque la primera matriz y la última es la misma, hace falta 0 multiplicaciones para obtenerse a ella misma. Es el caso base

m[i,j] = ∞ //Inicializa el número de multiplicaciones al máximo para hacer las comprobaciones de qué k es la que hace este número más pequeño.

for K=i to j-1 do //Prueba para todos los valores de k posibles cuál genera el menor número de multiplicaciones (recuerdo que k podía valer desde i hasta j-1)

q = Matrices-encadenadas-recursivo(p,i,k) + matrices-encadenadas-recursivo(p,k+1,j) + $p_{i-1}*p_k*p_j$ //Calcula recursivamente cuántas multiplicaciones habrá que hacer con esa k concreta

If q < m[i,j] then m[i,j]=q //Si el número de multiplicaciones es menor que el menor que ya teníamos calculado, ese será el nuevo menor número de multiplicaciones que puede hacer

Return m[i,j] //Lo que devuelve es el mínimo número de multiplicaciones que tienen que hacerse para calcular la multiplicación de las matrices desde A_i hasta A_j.

Entonces se plantea la siguiente recurrencia,

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

Se puede ver que para i = 1,2,…,n-1, cada término T(i) aparece una vez como T(k) y otra como T(n-k) (Más explicaciones que llevan 30 minutos y no sé si son útiles: Si por ejemplo ponemos 8 matrices (n=8) hará:

T(8) = 1 + (T(1) + T(7) + 1) + (T(2) + T(6) + 1) + (T(3) + T(5) + 1) + (T(4) + T(4) + 1) + (T(5) + T(3) + 1) + (T(6) + T(2) + 1) + (T(7) + T(1) + 1)

Como vemos, T(1), T(2), …, T(7) (valores que coinciden con los que dice de i = 1,2,…,n-1) han aparecido dos veces en toda la recurrencia, una como primer sumando y otra como segundo. Además, ha sumado un 1 en cada recurrencia hecha más el 1 inicial, por eso se suma 1 n veces, que es lo mismo que sumar n 1 vez. De esta forma, viendo que hace T(i) dos veces y suma n, podemos cambiar la expresión anterior por la siguiente:)

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

Como T(1)>= 1, o sea, $T(1) >= 2^0$, para n>=2 tenemos

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

Lo que demuestra que el tiempo es al menos exponencial en n. No son muchos subproblemas: uno para cada elección de i y j que cumpla 1<=i<=j<=j, es decir, en total

$C_{n,2} + n = O(n^2)$ ($C_{n,2}$ son todos los subconjuntos posibles que puedes formar con dos elementos con los valores de 1 a n ambos incluidos porque lo que está buscando son valores para i y j, por lo que tiene que probar cada i con cada j para ver esa pareja de valores qué resultados da. Más gráfico sería imaginar un for dentro de otro for que recorra todas las i posibles y todas las j posibles respectivamente y así probaría cada i con cada j y viceversa (y eso, señoras y señores, es n^2.))

Un algoritmo recursivo como hemos visto puede tratar el mismo problema muchas veces causando un solapamiento, por eso en vez de calcular (5) de forma recursiva, se hace con PD que tiene metodología tabular. El siguiente algoritmo supone que la matriz A_i tiene dimensiones $p_{i-1}*p_i$, para cualquier i = 1,2,…,n. Lo que se pasa como parámetro es una sucesión $\{p_0, p_1, \ldots, p_n\}$ de longitud n+1, es decir, leng[p] = n+1 (esta tontería sí que la aclara pero luego los algoritmos con más erratas que en la casa de los Ruiz). 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 el valor de k que produce el costo óptimo al calcular m[i,j].

ORDEN-CADENA-MATRICES(p)

n = leng[p] – 1 //Según el tamaño de p sabe cuántas matrices está multiplicando (n-1). //En este además voy a intentar poner un ejemplo en verde para hacerlo más fácil. Si p tiene 9 elementos, es porque hay 8 matrices, por lo que tomemos n=8

for i=1 to n do m[i,i]=0 //Rellena la diagonal de la tabla con 0 ya que el número de multiplicaciones que tienes que hacer para hallar una propia matriz, son 0 //En nuestra tabla aparecería el número 0 8 veces, en el número de multiplicaciones de la A_1 X A_1, A_2 x A_2 , etc

for z=2 to n do //Como las multiplicaciones de una sola matriz van a ser 0 multiplicaciones, empieza a partir de 2 (o eso deduzco) //La z va a tener valor desde 2 hasta 8. Nos ponemos en la primera iteración en la que vale 2

for i = 1 to n-z+1 do //con eso le va a dar todos los posibles valores a la i para probar todas las opciones posibles. //La i va a tener valores desde 1 hasta 7. Nos ponemos en la primera en la que vale 1

j = i + z -1 //Asigna a la j un valor que en este caso van a ser 2, 3, …, 8. Cuando la z valga 3, el rango de número que podrá valer es 3, 4, …, 8. Cuando la z valga el máximo, que será 8, el valor para la j será 8 (porque en ese caso la i como máximo valdrá n-z+1 = 8-8+1 = 1 y con i = 1, la j = 1 + 8 – 1 = 8). Es decir, en cada iteración de z, la j tendrá un valor menos disponible (porque empezará en uno más adelante) y por cada iteración de i irá avanzando hasta llegar a n (en nuestro caso 8). Todo el follón de bucles evita que la i y j valgan lo mismo y se ve lo de la metodología tabular porque cuando i vale 1, la j vale 2,3,4,5,6,7,8; cuando la i vale 2, la j vale 3,4,5,6,7,8; cuando la i

vale 3, la j vale 4,5,6,7,8; así por cada valor de 1 se está quitando muchos casos que ya ha comprobado (y que tendrías que hacer en la forma recursiva).

m[i,j] = ∞ //Pone que el número de multiplicaciones mínimo entre A_i y A_j son infinitas.

for k = i hasta j-1 do //Pone una k que va a ir desde i hasta j-1, (todos los valores que podía tomar) para comprobar en cada una de las combinaciones posibles de i y j todos los k posibles.

q=m[i,k] + m[k+1,j] + p_{i-1}*p_k*p_j //Asigna en una variable auxiliar q el número de multiplicaciones que se realizan desde la matriz A_i hasta A_j con ese número concreto de k (los acaba probando todos)

if q < m[i,j] then //condición de que ese número sea menor que el menor que teníamos calculado antes

m[i,j] = q //Si ocurre lo anterior, el nuevo mínimo es que tenemos calculado ahora mismo en q

s[i,j] = k //Y el valor de k que cumple esa condición se almacena en la tabla s

devuelve m y s //El algoritmo te devuelve el mínimo de multiplicaciones que tienes que hacer (m) y gracias a qué valor se consigue esto (k almacenado en s)

El algoritmo rellena la tabla m de manera que da solución al problema de la parentización en cadenas de marices de longitudes crecientes. 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,…,j-1, Ai..k es un producto de k-i+1 < j-i+1 matrices, y la matriz Ak+1..j es un producto de j-k < j-i+1 matrices.

El algoritmo primero pone m[i,i] = 0, para i = 1,2,…,n y durante la primera ejecución del siguiente lazo, usa (5) para calcular m[i,i+1], con i = 1,2,…,n-1 (los costos de las cadenas de longitud 2). La próxima vez que pasa el lazo calcula m[i,i+2], con i = 1,2,…,n-2 (mínimos costos para cadenas de longitud 3) y así, sucesivamente. Se ve fácilmente que es O(n3), por lo que se ve que este método es mucho más eficiente que el anterior. La siguiente etapa de PD es construir una solución óptima a partir de la información calculada. En nuestro caso usamos la tabla s[1..n,1..n] para saber la mejor forma de multiplicar las matrices. Por tanto, la forma óptima de multiplicar matrices $A_{1..n}$ es

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

La multiplicación anterior puede ser recursiva, ya que s[1,s[1,n]] determina la última multiplicación de matrices al calcular A (de una forma más entendible s[1,n] es el valor por el que partir la cadena para que la multiplicación de A_1 x…x A_n sea la que menos multiplicaciones hace. Entonces s[1,s[1,n]] es el valor por el que hay que partir la cadena de 1 a ese "k" para que la multiplicación desde A_1 x…x A_k sea la mejor. Y eso se podría volver a aplicar para hallar otra k intermedia, y otra y otra hasta que la cadena sea indivisible, ahí la recurrencia) y s[s[1,n]+1,n] (es lo mismo pero de la segunda parte de la cadena. Es valor por el que hay que partir la cadena para que la multiplicación restante desde A_{k+1} x A_n sea la mejor, y de nuevo se aplica desde esa "k" que parte la segunda cadena para sacar otra "k" por la que partir, y desde esa última "k" hasta n habrá otra "k" que parta cadena en buenas multiplicaciones y así sucesivamente hasta que no pueda haber más "k" intermedias) la última multiplicación 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, …, A_n}, la tabla s calculada por el algoritmo Orden-Cadena-matrices, y los índices i y j. El algoritmo usa MULT(A,B) de multiplicación de dos matrices.

MULTIPLICA-CADENA.MATRICES(A,s,i,j)

//Vuelve a no decir nada ☺ pero bueno, A es el conjunto de matrices que vas a multiplicar, s la tabla que contiene los números que previamente has calculado que te dice por qué número partir la cadena para que sea óptima la multiplicación dependiendo de la i de inicio y la j de fin, la i es la matriz de comiendo y la j es la de final de la cadena.

If j>i then //Si la matriz del final de la cadena es un número menor que la primera, no entra aqui

> **X = Multiplica-cadena-matrices(A,s,i,s[i,j])** //Mete en X el valor de llamar a esta misma función pero con los siguientes argumentos: La misma secuencia de matrices, con la tabla de todos los valores de s, la matriz de comienzo sigue siendo la misma y la última de la secuencia ahora es ese valor por el que se está partiendo la cadena original
>
> **Y = Multiplica-Cadena-matrices(A,s,s[i,j]+1,j)** //Mete en Y el valor de llamar a esta misma función pero con los siguientes argumentos: La misma secuencia de matrices, con la tabla de todos los valores de s, la matriz de comienzo es la siguiente al valor que parte la secuencia (la primera matriz de la segunda parte en la que se ha dividido la original) y la última de la secuencia es la misma que la de la original
>
> **Return MULT(X,Y)** //Multiplica las matrices en X e Y, porque para llegar hasta aquí tiene que haber pasado por todas las recurrencias posibles y ya llegan matrices concretas que cumplen con que son las mejores a multiplicar entre sí en cada caso

Else Return A_i //Si no se cumple el if anterior se da el caso baso, que es haber llegado a que se le pasa como matriz de inicio de secuencia y matriz final, la misma, por lo que devuelve la matriz en sí que es la que toca multiplicar. (caso base)

El problema del "play off"

Este problema no es un claro exponente de la aplicabilidad de la PD, la metodología tabular de la PD estudiada proporciona mejores resultados que los potenciales de un enfoque recursivo.

Supongamos dos equipos A y B que juegan una final en la que se quiere saber cuál será el primero en ganar n partidos, para cierto n particular, es decir, deben jugar como mucho 2n-1 partidos. El problema del play off es tal final cuando el número de partidos necesarios es n = 4 (por eso es 2n-1 porque si n=4, necesitamos 7 partidos para determinar que si un equipo ha ganado 4 partidos ya ha ganado). Se puede suponer que ambos equipos son igualmente competentes y que, por tanto, la probabilidad de que A gane algún partido concreto es ½ (es decir, hay igualdad, es un Granada-Barcelona, pueden ganar ambos al 50%).

Sea P(i,j) la probabilidad de que sea A el ganador final si A necesita i partidos para ganar y B necesita j. Antes del primer partido, la probabilidad de que gane A es P(n,n), y es obvio que

P (0, i) = 1 si $1 \leq i \leq n$ (la probabilidad de que el equipo A gane si necesita 0 partidos y B i)
P (i,0) = 0 si $1 \leq i \leq n$ (la probabilidad de que el equipo A gane si necesita i partidos y B 0)

No está definida P(0,0) (la probabilidad de que ambos equipos ganen con 0 partidos)

Así, como A puede ganar cualquier partido con probabilidad ½ y perderlo con ½ tenemos que:

$P(i,j) = [P(i-1, j) + P(i, j-1)]/2$, $i \geq 1$, $j \geq 1$ es decir,

P(i,j) = 1 si i=0 y j>0
P(i,j) = 0 si i>0 y j=0
$P(i,j) = [[P(i-1, j) + P(i, j-1)]/2$ si i>0 y j>0

Esta ultima probabilidad es la suma de la probabilidad de que el equipo A necesita i-1 partidos y el equipo B j partidos para ganar junto a la probabilidad de que el equipo A necesite i partidos y el equipo B j-1. Todo ello dividido entre 2, esto es porque cada

equipo tiene las mimas probabilidades tanto de ganar como de perder un partido bajo las mismas condiciones de partidos.

Podemos calcular P(i,j) recursivamente. Sea T(k) el tiempo necesario en el peor de los casos para calcular P(i,j) con i+j=k. Si diseñáramos un algoritmo recursivo para realizar los cálculos que queramos, tendríamos que:
T(1) = c
T(k) = 2T(k-1) + d, k>1 donde c y d son constantes.

La resolución de esa recurrencia nos dice que T(k) consume un tiempo en $O(2^k) = O(4^n)$, si i=j=n lo que demuestra que no es un método demasiado práctico para un n demasiado grande.
Intentamos sacar ventaja de la estructura encajada de los sub-problemas, es decir, de la metodología tabular subyacente en la técnica de la PD. Para ello tenemos otra forma de calcular P(i,j) y es rellenando una tabla en la que los elementos de la última fila serían todos ceros, y los de la última columna todos uno. Cualquier casilla de la tabla es la media de la casilla anterior y la que está a su derecha.
Es por ello por lo que una forma válida de rellenar la tabla es en diagonales, empezando por la esquina sureste (abajo a la derecha) y procediendo hacia arriba a la izquierda a lo largo de las diagonales, que representan casillas con valores constantes de i+j (k). Así con los datos del problema (CON N=4), partiríamos de una matriz como la siguiente:

				1	**4**
				1	**3**
				1	**2**
				1	**1**
0	0	0	0	**XXX**	**0**
4	**3**	**2**	**1**	**0**	

Las XXX es como dijimos anteriormente que la P(0,0) no estaba definida

En primer lugar, podríamos obtener que P(1,1) = (P(0,1) + P(1,0))/2 = (1+0)/2 = ½
Y sucesivamente podríamos rellenar la tabla así:

½	21/32	13/16	15/16	1	**4**
11/32	½	11/16	7/8	1	**3**
3/16	5/16	½	3/4	1	**2**
1/16	1/8	1/4	½	1	**1**
0	0	0	0	**XXX**	**0**
4	**3**	**2**	**1**	**0**	

El algoritmo escrito en código haría lo siguiente:

ALGORITMO PLAYOFF
Begin
 For s := 1 to i+j do begin

```
    P[0,s] = 1.0;
    P[s,0] = 0.0;
     For k = 1 to s-1 do
         P[k,s-k] = (P[k-1,s-k] + P[k,s-k-1])/2.0
 end;
 Return (P[i,j])
End
```

El código como vemos es muy sencillo de analizar, pues realiza un bucle de 1 a k (i+j) y calcula la probabilidad de que un equipo gane con 0 partidos y el otro con s que es 1, que ese mismo equipo gane con s partidos y el otro con 0, que es 0, y dentro de ese bucle, mediante otro for va calculando las probabilidades de que ese equipo gane jugando k partidos y el otro s-k, lo que es lo mismo que desarrollar la fórmula que hemos analizado anteriormente.
La eficiencia de este código es muy sencilla, son dos for anidados, por tanto, hablamos de $O(n^2)$, eficiencia muy inferior al método directo, de ahí los beneficios de la PD y de la subdivisión en problemas.