Apuntes de Teórico

PROGRAMACIÓN 3

Programación Dinámica

Versión 1.1

Índice

Introducción

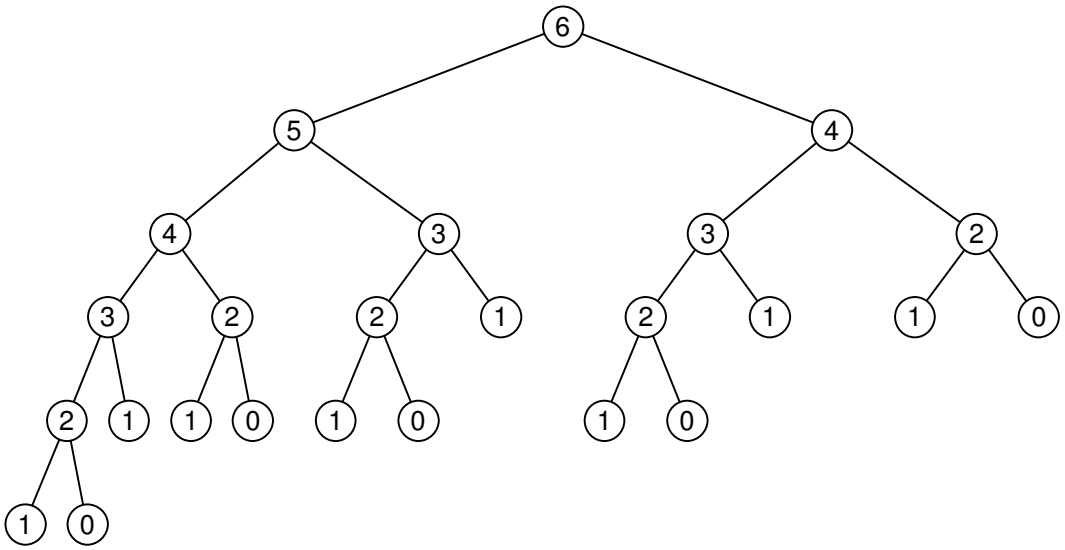
En el curso se vio como método de resolución de problemas la técnica Divide & Conquer, la cual consiste en dividir el problema en subproblemas, resolver estos problemas en forma independiente, eventualmente aplicando a cada uno de ellos la misma técnica y finalmente combinar las soluciones de los subproblemas para obtener la solución al problema original.
Al resolver cada subproblema en forma independiente es posible que un mismo subproblema se resuelva varias veces, por ejemplo si se considera el cálculo de los números de Fibonacci, los cuales se definen como:

$$F(0) = F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

El algoritmo para calcular $F(n)$ utilizando la técnica Divide & Conquer surge naturalmente de la definición:

```
int F (int n)
{
      if (n == 0)
            return 1;
      else
            if (n == 1)
                  return 1;
            else
                  return F(n-1) + F(n-2);
}
```

Para resolver $F(6)$ el algoritmo realiza los siguientes cálculos:



donde, por ejemplo:

Fibonacci(4) se resuelve 2 veces.
Fibonacci(3) se resuelve 3 veces.
Fibonacci(2) se resuelve 5 veces.

Sin embargo es posible obtener una ventaja de la repetición de cálculos, guardando los resultados intermedios para utilizarlos cuando sea necesario. Los resultados intermedios pueden ser guardados en una tabla en este caso.

```
int F(int n)
{
      int Fib[n+1];
      Fib[0] = 1;
      Fib[1] = 1;
      for(int i = 2; i <= n; i++)
      {
            Fib[i] = Fib[i-1] + Fib[i-2];
      }
      return Fib[n];
}
```

Se puede observar que además de evitar la repetición de cálculos se obtiene una mejora en el orden del algoritmo que pasa a ser lineal.
Este último algoritmo fue escrito utilizando la técnica de *Programación Dinámica* que tiene entre sus objetivos evitar la repetición de cálculos, usando habitualmente tablas donde se almacenan las soluciones a los subproblemas a medida que estos son resueltos y usando dichas soluciones cuando sea necesario.

Una de las principales aplicaciones de *Programación Dinámica* es la resolución de problemas de optimización, la solución a los cuales puede expresarse como una secuencia de decisiones.

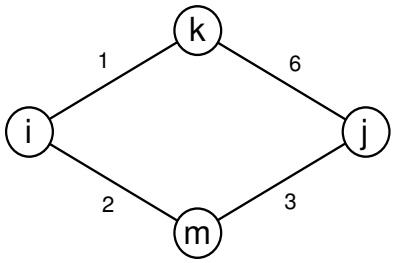
Anteriormente se estudió el método Greedy y se lo aplicó a problemas tales como:

a) Problema de la mochila.
b) Caminos de mínimo costo desde un vértice a todos los demás vértices (Dijkstra).

Para problemas similares a los anteriores la solución óptima puede hallarse tomando una decisión a la vez y sin cometer errores (decisión localmente óptima). Sin embargo existen otros problemas de optimización para los cuales no es posible hallar la solución óptima a partir de una secuencia de decisiones localmente óptimas, un ejemplo es el siguiente:

> Dado un grafo G = (V, E) y 2 vértices i, j ∈ V, hallar el camino de menor costo de i a j.

Si A_i el conjunto de vértices adyacentes al vértice i, la decisión a tomar es cual de los vértices de A_i será el segundo vértice del camino, sin embargo no hay forma de tomar una decisión (elegir un vértice) en este paso que garantice que futuras decisiones conduzcan a la solución óptima, como se observa en el siguiente caso:



Una posibilidad para resolver este tipo de problemas para los cuales no es posible tomar una decisión única en un paso, es generar todas las posibles secuencias de decisión y luego tomar la mejor (*Backtracking*) lo cual conduce generalmente a algoritmos de ordenes muy altos (exponenciales o factoriales).

Programación Dinámica reduce drásticamente la cantidad de secuencias de decisión generadas evitando generar aquellas que no pueden ser óptimas. La forma de lograr esto (una secuencia óptima de decisiones) es haciendo uso del *Principio de Optimalidad*.

En conclusión el método de *Programación Dinámica* es aplicable a la resolución de problemas de optimización basándose en la aplicación del *Principio de Optimalidad* y almacenando resultados intermedios en tablas.

Principio de optimalidad

El *Principio de Optimalidad* es una propiedad que deben cumplir las soluciones óptimas de un problema si es que es posible usar *Programación Dinámica* para resolverlo.
Este principio establece que:

Sea $d_1, d_2, \ldots, d_n$ una secuencia de decisiones que conduce a la solución óptima.
Una secuencia óptima de decisiones como la anterior cumple la propiedad de que cualquiera sea el estado inicial del problema y la primer decisión tomada, las restantes decisiones deben constituir una secuencia óptima de decisiones con respecto al estado resultante del problema luego de la primer decisión.

Para aplicar *Programación Dinámica* para resolver un problema se debe verificar que el principio de optimalidad es aplicable.

Ejemplo: Camino de menor costo

Consideremos nuevamente el problema de encontrar un camino de menor costo entre dos vértices dados i y j.

Sea $i, i_1, i_2, \ldots, i_k, j$ el camino de menor costo de i a j.

Comenzando en el vértice i se tomó como primera decisión ir al vértice i_1, luego de esta decisión el estado del problema está definido por el vértice i_1 y encontrar un camino de menor costo de i_1 a j.

Claramente el camino $i_1, i_2, \ldots, i_k, j$ debe ser el camino de menor costo de i_1 a j, porque si no lo fuera existiría otro camino $i_1, r_1, r_2, \ldots, r_q, j$ de costo menor que el anterior y por lo tanto el camino $i, i_1, r_1, r_2, \ldots, r_q, j$ sería un camino de costo menor que $i, i_1, i_2, \ldots, i_k, j$ lo cual es absurdo ya que este último es la solución optima, en consecuencia el *Principio de Optimalidad* es aplicable a este problema.

Ejemplo: problema de la mochila

Consideremos una variante del problema de la mochila en la cual no se admite fraccionar los objetos, el planteo del problema es:

$$\text{Maximizar } \sum_{i=1}^{n} p_i x_i$$

$$\text{sujeto a } \sum_{i=1}^{n} w_i x_i \leq M$$

$$\text{con } x_i \in \{0,1\} \quad \forall i \quad 1 \leq i \leq n$$

A los efectos de representar el problema de colocar los objetos del 1 al n-ésimo en una mochila de capacidad M, se utilizará la siguiente notación: Knap(1, n, M)

Sea $y_1, y_2, \ldots, y_n$ una solución óptima al problema Knap(1, n, M).

La primer decisión es el valor de y_1:

Si $y_1 = 0 \Rightarrow$ $y_2, \ldots, y_n$ debe ser una solución óptima al problema Knap(2, n, M), si no lo fuera entonces $y_1, y_2, \ldots, y_n$ no sería una solución óptima a Knap(1, n, M)

Si $y_1 = 1 \Rightarrow$ $y_2, \ldots, y_n$ debe ser una solución óptima al problema Knap(2, n, $M - w_1$), porque si no lo es, existe otra solución $z_2, \ldots, z_n$ mejor (con mayor ganancia) por lo tanto $y_1, z_2, \ldots, z_n$ es una solución mejor que la secuencia $y_1, y_2, \ldots, y_n$ lo cual es absurdo porque $y_1, y_2, \ldots, y_n$ es una solución óptima.

Por lo tanto el *Principio de Optimalidad* es aplicable a este problema.

Aplicación del Principio de Optimalidad

En esta sección se verá como utilizar el *Principo de Optimalidad* para hallar la solución óptima de un problema. Para lo cual se considerará la siguiente formulación del mismo:

Sea S_0 el estado inicial de problema.
Se deben tomar n decisiones d_i con $1 \leq i \leq n$ para obtener la solución óptima.
Sea $D_1 = \{r_1, r_2, \ldots, r_k\}$ los posibles valores de la solución d_1.
Sea S_j el estado del problema luego de haber elegido el valor r_j, $1 \leq j \leq k$ para la decisión d_1.
Sea T_j una secuencia óptima de decisiones respecto al estado S_j.

Entonces cuando el *Principio de Optimalidad* es aplicable, una secuencia óptima de decisiones con respecto al estado inicial S_0 del problema es la mejor de las secuencias de decisión r_jT_j con $1 \leq j \leq k$.

Ejemplo: problema de la mochila

Sea $g_j(Y)$ el valor de la ganancia de la solución óptima al problema Knap($j + 1$, n, Y).

Claramente $g_0(M)$ es el valor de la ganancia de la solución óptima al problema Knap(1,n,M).

Los posibles valores para la primer decisión son $x_1 = 0$ ó $x_1 = 1$ ($D_1 = \{0, 1\}$).
Según el principio de optimalidad:

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

donde:

- $g_1(M)$: ganancia de la solución óptima del problema Knap(2, n, M) ($x_1 = 0$)
- $g_1(M - w_1)$:ganancia de la solución óptima del problema Knap(2, n, $M - w_1$) ($x_1 = 1$)
- p_1: ganancia del objeto 1

El *Principio de Optimalidad* es aplicable a cualquier estado intermedio del problema de la misma manera que es aplicable al estado inicial:

Si $y_1, y_2, \ldots, y_n$ es la solución óptima al problema Knap(1, n, M) entonces para cada j, con $1 \leq j \leq n$

$y_1, \ldots, y_j$ es la solución óptima al problema Knap$\left(1, j, \sum_{i=1}^{j} w_i y_i\right)$

y

$y_{j+1}, \ldots, y_n$ es la solución óptima al problema Knap($j+1, n, M - \sum_{i=1}^{j} w_i y_i$)

Lo anterior significa que en una secuencia óptima de decisiones cualquier subsecuencia debe ser una solución óptima. Esto permite generalizar la relación $g_0(M)$ a:

$$g_j(Y) = \text{máximo}(g_{j+1}(Y), g_{j+1}(Y - w_{j+1}) + p_{j+1}) \quad \forall Y \quad 0 \le Y \le M, \quad 0 \le j < n$$

donde

$g_{j+1}(Y)$: ganancia cuando $x_{j+1} = 0$
$g_{j+1}(Y - w_{j+1})$: ganancia cuando $x_{j+1} = 1$
p_{j+1} : ganancia del objeto $j + 1$

Además: $g_n(Y) = 0 \quad \forall Y \quad 0 \le Y \le M$

La aplicación recursiva del principio de optimalidad conduce a una relación de recurrencia como la anterior. Los algoritmos basados en *Programación Dinámica* resuelven dicha recurrencia para obtener la solución al problema. En general los algoritmos basados en *Programación Dinámica* son iterativos para evitar la repetición de cálculos.

Se empieza calculando $g_n(Y) = 0 \quad \forall Y \quad 0 \le Y \le M$ y a partir de estos valores se puede obtener $g_{n-1}(Y)$, luego $g_{n-2}(Y)$ y de esta manera se llega a $g_0(M)$ que es la ganancia de la solución óptima.

El algoritmo es el siguiente:

```
int FKnap(int CantObj, int *W, int *P, int M)
{
      int g[CantObj + 1][M + 1];
      for(int c = 0; c <= M; c++)
            g[CantObj][c] = 0;
      for(int j = CantObj - 1; j >= 0; j--)
            for(int c = 0; c <= M; c++)
                  if (W[j] <= c)
                        g[j][c] = max(g[j+1][c], g[j+1][c-W[j]] + P[j]);
                  else
                        g[j][c] = g[j+1][c];
      return g[0][M];
}
```

Se mostrará cómo funciona detalladamente el algoritmo para el siguiente ejemplo:

$CantObj = 3$

$M = 6$

$w = \{2,3,4\}$

$P = \{1,2,5\}$

n \ M	0	1	2	3	4	5	6
0							
1							
2							
3	0	0	0	0	0	0	0

<u>$j = CantObj - 1$ ($j = 2$)</u>

el objeto es más grande y no cabe

$$\begin{cases} c = 0: & G[2,0] = G[3,0] \\ c = 1: & G[2,1] = G[3,1] \\ c = 2: & G[2,2] = G[3,2] \\ c = 3: & G[2,3] = G[3,3] \end{cases}$$

$$c = 4: \quad G[2,4] = \max(G[3,4], G[3,0] + p[2]) = 5$$

$$c = 5: \quad G[2,5] = \max(G[3,5], G[3,1] + p[2]) = 5$$

$$c = 6: \quad G[2,6] = \max(G[3,6], G[3,2] + p[2]) = 5$$

n \ M	0	1	2	3	4	5	6
0							
1							
2	0	0	0	0	5	5	5
3	0	0	0	0	0	0	0

<u>$j = CantObj - 2$ ($j = 1$)</u>

el objeto es más grande y no cabe

$$\begin{cases} c = 0: & G[1,0] = G[2,0] \\ c = 1: & G[1,1] = G[2,1] \\ c = 2: & G[1,2] = G[2,2] \end{cases}$$

$$c = 3: \quad G[1,3] = \max(G[2,3], G[2,0] + p[1]) = 2$$
$$c = 4: \quad G[1,4] = \max(G[2,4], G[2,1] + p[1]) = 5$$
$$c = 5: \quad G[1,5] = \max(G[2,5], G[2,2] + p[1]) = 5$$
$$c = 6: \quad G[1,6] = \max(G[2,6], G[2,3] + p[1]) = 5$$

n \ M	0	1	2	3	4	5	6
0							
1	0	0	0	2	5	5	5
2	0	0	0	0	5	5	5
3	0	0	0	0	0	0	0

j = CantObj – 3 (*j* = 0)

el objeto es más grande y no cabe

$$\begin{cases} c = 0: \quad G[0,0] = G[1,0] \\ c = 1: \quad G[0,1] = G[1,1] \end{cases}$$
$$c = 2: \quad G[0,2] = \max(G[1,2], G[1,0] + p[0]) = 1$$
$$c = 3: \quad G[0,3] = \max(G[1,3], G[1,1] + p[0]) = 2$$
$$c = 4: \quad G[0,4] = \max(G[1,4], G[1,2] + p[0]) = 5$$
$$c = 5: \quad G[0,5] = \max(G[1,5], G[1,3] + p[0]) = 5$$
$$c = 6: \quad G[0,6] = \max(G[1,6], G[1,4] + p[0]) = 6$$

n \ M	0	1	2	3	4	5	6
0	0	0	1	2	5	5	6
1	0	0	0	2	5	5	5
2	0	0	0	0	5	5	5
3	0	0	0	0	0	0	0

$$\Rightarrow G[0, M] = G[0,6] = 6$$

Solución óptima: *Ganancia* = 6

Cuando se formula la relación de recurrencia que debe ser resuelta se pueden utilizar dos enfoques: “hacia adelante” o “hacia atrás”.

Enfoque hacia delante: la decisión d_i es tomada en base a la secuencia óptima de decisiones $d_{i+1}, \ldots, d_n$.

Enfoque hacia atrás: la decisión d_i es tomada en base a la secuencia óptima de decisiones $d_1, \ldots, d_{i-1}$.

Notar que la recurrencia hallada en el ejemplo de la mochila corresponde al enfoque hacia adelante.

A continuación se verá como formular la recurrencia con el enfoque hacia atrás.

Sea $f_j(Y)$ el valor de la solución óptima para Knap(1, j, Y)

Claramente $f_n(M)$ es la solución óptima al problema Knap(1, n, M)

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

donde:

$f_{n-1}(M)$ es la solución óptima a Knap(1, n-1, M) (no se lleva el objeto n-ésimo)
$f_{n-1}(M-w_n)$ es la solución óptima a Knap(1, n-1, M-w_n) (se lleva el objeto n-ésimo)

Generalizando:

$$\begin{cases} f_i(Y) = \max\{ \quad f_{i-1}(Y),\ f_{i-1}(Y - w_i) + p_i\} \forall\, Y\ 0 \le Y \le M\ ,\ 0 < i \le n \\ f_0(Y) = 0 \quad \forall\, Y\ ,\ 0 \le Y \le M \end{cases}$$

Partiendo de $f_0(Y)$ se calcula $f_1(Y)$, luego $f_2(Y)$ y se continúa hasta llegar a $f_n(M)$ que es el valor buscado.

```
int BKnap(int CantObj, int* w, int* p, int M)
{
      int f[CantObj+1][M+1];
      for(int c = 0; c <= M; c++)
            f[0][c] = 0;
      for(int i = 1; i <= CantObj; i++)
            for(int c = 0; c <= M; c++)
                  if (w[i-1] <= c)
                        f[i][c] = max(f[i-1][c], f[i-1][c-w[i-1]]+p[i-1]);
                  else
                        f[i][c] = f[i-1][c];
      return f[CantObj][M];
}
```

Se mostrará cómo funciona detalladamente el algoritmo para el mismo ejemplo anterior:

$CantObj = 3$
$M = 6$
$w = \{2,3,4\}$
$P = \{1,2,5\}$

n \ M	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1							
2							
3							

<u>$i = 1$</u>

$c = 0: \quad f[1,0] = f[0,0]$
$c = 1: \quad f[1,1] = f[0,1]$
$c = 2: \quad f[1,2] = \max(f[0,2], f[0,0] + p[1]) = 1$
$c = 3: \quad f[1,3] = \max(f[0,3], f[0,1] + p[1]) = 1$
$c = 4: \quad f[1,4] = \max(f[0,4], f[0,2] + p[1]) = 1$
$c = 5: \quad f[1,5] = \max(f[0,5], f[0,3] + p[1]) = 1$
$c = 6: \quad f[1,6] = \max(f[0,6], f[0,4] + p[1]) = 1$

n \ M	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2							
3							

<u>$i = 2$</u>

$c = 0:\quad f[2,0] = f[1,0]$

$c = 1:\quad f[2,1] = f[1,1]$

$c = 2:\quad f[2,2] = f[1,2]$

$c = 3:\quad f[2,3] = \max(f[1,3], f[1,0] + p[2]) = 2$

$c = 4:\quad f[2,4] = \max(f[1,4], f[1,1] + p[2]) = 2$

$c = 5:\quad f[2,5] = \max(f[1,5], f[1,2] + p[2]) = 3$

$c = 6:\quad f[2,6] = \max(f[1,6], f[1,3] + p[2]) = 3$

n \ M	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3							

<u>$i = 3$</u>

$c = 0:\quad f[3,0] = f[2,0]$

$c = 1:\quad f[3,1] = f[2,1]$

$c = 2:\quad f[3,2] = f[2,2]$

$c = 3:\quad f[3,3] = f[2,3]$

$c = 4:\quad f[3,4] = \max(f[2,4], f[2,0] + p[3]) = 5$

$c = 5:\quad f[3,5] = \max(f[2,5], f[2,1] + p[3]) = 5$

$c = 6:\quad f[3,6] = \max(f[2,6], f[2,2] + p[3]) = 6$

n \ M	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	6

$\Rightarrow f[n, M] = f[3,6] = 6$

Solución óptima: *Ganancia* = 6

Ejemplo:Producto de n matrices

Consideremos el problema de realizar el producto de *n* matrices $M = M_0 \times M_1 \times ... \times M_{n-1}$ donde cada matriz M_i tiene r_i filas y r_{i+1} columnas.

Como el producto de matrices cumple la propiedad asociativa existen distintas maneras de parentizar la expresión y el orden en el cual se realizan las multiplicaciones va a determinar el número total de multiplicaciones de elementos que son necesarias.

Por ejemplo:

$M = M_0 \times M_1 \times M_2 \times M_3$

$M_0 \rightarrow 10 \times 20$

$M_1 \rightarrow 20 \times 50$

$M_2 \rightarrow 50 \times 1$

$M_3 \rightarrow 1 \times 100$

Una posible forma de evaluar *M* es realizar las siguientes multiplicaciones:

$$M = (M_0 \times (M_1 \times (M_2 \times M_3)))$$

$$\left.\begin{array}{l} M_2 \times M_3 \rightarrow 50 \times 1 \times 100 = 5000 \\ M_1 \times (M_2 \times M_3) \rightarrow 20 \times 50 \times 100 = 100000 \\ M_0 \times (M_1 \times (M_2 \times M_3)) \rightarrow 10 \times 20 \times 100 \rightarrow 20000 \end{array}\right\} 125000 \textit{ multiplicaciones}$$

Otra forma es:

$$M = ((M_0 \times (M_1 \times M_2)) \times M_3)$$

$$\left.\begin{array}{l} M_1 \times M_2 \rightarrow 20 \times 50 \times 1 = 1000 \\ M_0 \times (M_1 \times M_2) \rightarrow 10 \times 20 \times 1 = 200 \\ (M_0 \times (M_1 \times M_2)) \times M_3 \rightarrow 10 \times 1 \times 100 \rightarrow 1000 \end{array}\right\} 2200 \textit{ multiplicaciones}$$

Se quiere determinar la cantidad mínima de multiplicaciones necesarias para realizar el producto de *n* matrices., el *Principio de Optimalidad* es aplicable?

Sea d_0, d_1, …, d_q la secuencia de decisiones óptima para determinar la cantidad mínima de multiplicaciones de elementos necesaria para evaluar el producto $M = M_0 \times M_1 \times ... \times M_{n-1}$ donde d_i indica que subsecuencia de matrices deben ser multiplicadas entre sí.

En el ejemplo anterior sería:

$$M_0 \times ((M_1 \times M_2) \times M_3)$$

|----------| d_0

|----------| d_1

|----------| d_2

Si la decisión d_q (última decisión) implica que el proceso de multiplicación culmina con la multiplicación de las matrices resultantes de multiplicar las subcadenas de *0* a *k* y de *k*+1 a *n-1*

$$\underbrace{M_0 \times M_1 \times ... \times M_k}_{M'} \times \underbrace{M_{k+1} \times ... \times M_{n-1}}_{M''}$$

$\Rightarrow$

las decisiones d_0, …, d_{q-1} deben realizar la multiplicación de cada subcadena de forma óptima, porque sino existe otra secuencia de decisiones d'_0, …, d'_{q-1} que implica una menor cantidad de multiplicaciones $\Rightarrow$ d'_0, …, d'_{q-1}, d_q tendría menor cantidad de multiplicaciones que d_0, …, d_{q-1} lo cual es absurdo porque esta última es la solución óptima.

Como notación se utilizará

Costo(i, j) : solución óptima (mínimo número de multiplicaciones) para resolver el producto $M_i \times M_{i+1} \times ... \times M_j$

$\Rightarrow$ el valor buscado es Costo(0, *n-1*)

Considerando la demostración de la aplicabilidad del *Principio de Optimalidad* surge que:

$$\begin{cases} Costo(i, j) = \min(Costo(i,k), Costo(k+1, j) + r_i \times r_{k+1} \times r_{j+1}) & 0 \le i \le k < j \le n-1 \\ Costo(i, j) = 0 \quad si\ i = j \end{cases}$$

En la primera ecuación:

- el término *Costo*(*i*,*k*) es la mínima cantidad de multiplicaciones para el producto $M' = M_i \times ... \times M_k$

- el término *Costo*(*k*+1, *j*) es la mínima cantidad de multiplicaciones para el producto $M'' = M_{k+1} \times ... \times M_j$

- el término $r_i \times r_{k+1} \times r_{j+1}$ es la cantidad de multiplicaciones que se requieren para calcular el producto $M' \times M''$, observar que *M'* es de dimensión r_i x r_{k+1} y *M''* es de dimensión r_{k+1} x r_{j+1}.

Como la función *Costo* depende de dos variables se utilizará como estructura para almacenar los resultados intermedios una matriz, además como $i \leq j$ solo se utilizará la diagonal principal y el triángulo superior.

Lo primero que se calculará es la diagonal, aplicando el paso base de la recurrencia. Luego el algoritmo procederá por diagonales hasta las casilla (0, *n-1*).

```
void MultMatrices(int r[n+1], int costo[n][n], int plan[n][n])
{
      for(int i = 0; i < n; i++)
            costo[i][i] = 0;
      for(int diag = 1; diag < n; diag++)
            for(int col = diag; col < n; col++)
            {
                  int fil = col - diag;
                  int min = MAX_INT;
                  int ind;
                  for(int k = fil; k < col; k++)
                  {
                        int c = costo[fil][k] + costo[k+1][col] +
                              r[fil]*r[k+1]*r[col+1];
                        if (c < min)
                        {
                              min = c;
                              ind = k;
                        }
                  }
                  costo[fil][col] = min;
                  plan[fil][col] = ind;
            }
}
```

Caminos de menor costo entre todo par de vértices

Sea $G = (V, E)$ un grafo en el cuál cada arista tiene asociado un costo, se quiere calcular el costo del camino de mínimo costo entre todo par de vértices $(i, j) / i, j \in V$

Sea C la matriz que contiene el costo de cada arista, en la cual se tienen los valores:

$C(i, i) = 0$
$C(i, j)$ = costo de la arista si $i \neq j$
$C(i, j) = \infty$ si $(i, j) \notin E$

Los costos de las aristas pueden ser negativos, pero el grafo no puede tener ciclos de costo negativo.

Para resolver este problema mediante *Programación Dinámica* se debe verificar que el *Principio de Optimalidad* es aplicable al problema:

> Si k es un vértice intermedio que pertenece al camino de menor costo de i a $j \Rightarrow$ los subcaminos de i a k y de k a j deben ser los de menor costo, si no fuera así, el camino de i a j no sería óptimo.

Por lo tanto el principio de optimalidad es aplicable.

La idea es obtener una matriz D que contenga el costo del camino de menor costo entre todo par de vértices.

Inicialmente $D=C$.

La matriz D contendrá luego de k iteraciones el costo de los caminos de menor costo entre todo par de vértices que únicamente estén formados por los vértices $0..k$ como vértices intermedios, por lo tanto luego de n iteraciones contendrá el resultado buscado:

> el costo del camino de menor costo entre todo par de vértices que pasan por cualquiera de los n vértices del grafo como vértices intermedios.

Se utilizará la siguiente notación:

$D^k(i, j)$: costo del camino de menor costo entre los vértices i y j utilizando únicamente los vértices $0..k$ como vértices intermedios.

El algoritmo deberá chequear en la k-ésima iteración si existe un camino mejor, entre todo par de vértices i, j, utilizando el vértice k, que el camino de la iteración anterior que utiliza

los vértices *0..k*-1. O sea que se está intentando hallar $D^k(i, j)$ a partir de $D^{k-1}(i, j)$, para lo cual se deben considerar los siguientes casos:

Caso 1:

El camino de menor costo entre el vértice *i* y el *j* utilizando los vértices 0..*k* como vértices intermedios no pasa por el vértice *k*

$\Rightarrow D^k(i, j) = D^{k-1}(i, j)$

Caso 2:

El camino de menor costo del vértice *i* al *j* utilizando los vértices *0..k* como vértices intermedios pasa por el vértice *k*

$\Rightarrow$ el camino puede ser dividido en dos subcaminos:
el subcamino de *i* a *k* y el subcamino de *k* a *j*

$\Rightarrow D^k(i, j) = D^{k-1}(i, k) + D^{k-1}(k, j)$ (en forma implícita se esta diciendo que el camino óptimo no pasa 2 veces por el mimo vértice)

De los casos anteriores se concluye que:

$$\begin{cases} D^k(i, j) = \min(D^{k-1}(i, j), D^{k-1}(i,k) + D^{k-1}(k, j)) \forall k \in 1..n-1 \\ D^0(i, j) = C(i, j) \end{cases}$$

Se ha encontrado una relación de recurrencia que permite, a partir de los problemas más simples (D^0, D^1,…) hallar la solución al problema planteado (D^n).

El algoritmo que implemente la relación de recurrencia y permite calcular D^n es el siguiente y recibe el nombre de *Floyd*.

```
void Floyd (int C[n][n], int D[n][n])
{
      for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
                  D[i][j] = C[i][j];
      for(int k = 0; k < n; k++)
            for(int i = 0; i < n; i++)
                  for(int j = 0; j < n; i++)
                        if (D[i][k] + D[k][j] < D[i][j])
                              D[i][j] = D[i][k] + D[k][j];
}
```

Si se quiere conocer también el camino de menor costo entre todo par de vértices y no solamente su costo, se puede utilizar otra matriz *P*, la cual se inicializaría en 0 y el algoritmo sería:

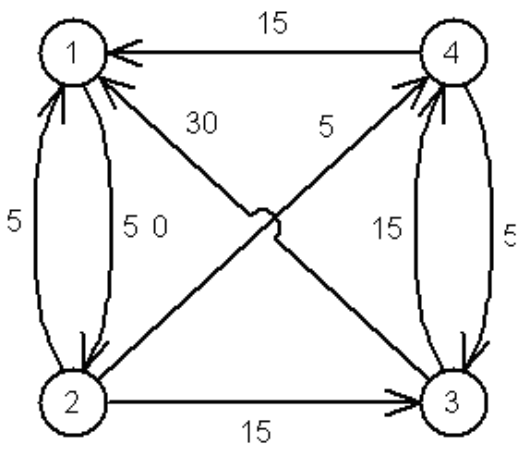
```
void Floyd (int C[n][n], int D[n][n])
{
      for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++)
            {
                  D[i][j] = C[i][j];
                  P[i][j] = 0;
            }
      for(int k = 0; k < n; k++)
            for(int i = 0; i < n; i++)
                  for(int j = 0; j < n; i++)
                        if (D[i][k] + D[k][j] < D[i][j])
                        {
                              P[i][j] = k;
                              D[i][j] = D[i][k] + D[k][j];
                        }
}
```

Cuando finaliza el algoritmo, $P(i, j)$ contiene el vértice intermedio que forma el camino de menor costo. Para encontrar el camino de menor costo de i a j se hace lo siguiente:

Si $P(i, j) = 0 \Rightarrow$ el camino de menor costo de i a j es la arista (i, j) .

Si $P(i, j) \neq 0 \Rightarrow$ sea $k = P(i, j) \Rightarrow$ el camino de menor costo de i a j pasa por el vértice k Mirando recursivamente en $P(i, k)$ y $P(k, j)$ se pueden encontrar todos los vértices intermedios del camino.

Sea el siguiente grafo:



$$D^0 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$ (Matriz de costos *C*)

$$D^1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Costos de los caminos de menor costo de *i* a *j* pasando por el vértice 1 como único vértice intermedio.

$$D^2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Costos de los caminos de menor costo de *i* a *j* pasando por los vértices 1 y 2 como vértices intermedios.

$$D^3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Costos de los caminos de menor costo de *i* a *j* pasando por los vértices1,2 y 3 como vértices intermedios.

$$D^4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Costos de los caminos de menor costo de *i* a *j* pasando por los vértices 1,2, 3 y 4 como vértices intermedios.

La matriz P con los vértices que forman el camino de menor costo queda:

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$