4. Programación Dinámica

4.1 Introducción

La técnica denominada Programación Dinámica (PD), al igual que el método Divide y Vencerás, soluciona los problemas combinando las soluciones de subproblemas de menor tamaño. En divide y vencerás la partición se hace en subproblemas independientes. En contraste, programación dinámica se aplica cuando los subproblemas no son independientes, esto quiere decir que los subproblemas comparte otros subproblemas. En este contexto divide y vencerás trabajan más de lo necesario, ya que resuelven más de una vez el mismo subproblema. En programación dinámica cada subproblema se resuelve una sola vez y su solución se almacena en un tabla. Programación Dinámica se aplica normalmente en problemas de optimización. En estos problemas pueden existir muchas soluciones, cada solución tiene un valor y en este caso deseamos obtener el valor óptimo (un valor mínimo o máximo).

Ejemplo 4.1.1
Para obtener el valor de Fibonacci de un número n tenemos la siguiente formulación recursiva:

$$F(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ F(n-1)+F(n-2) & \text{si } n > 2 \end{cases}$$

Los algoritmos basados en DyV y PD son los siguientes:

Técnica DyV

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

EFICIENCIA: $\Theta(1{,}62^n)$. Se repiten muchos cálculos.

Técnica PD

```
int Fibonacci(int n){
 int T[n+1];
 T[0]=T[1]=T[2]=1;
 for (int i=3;i<=n;i++)
  T[i]=T[i-1]+T[i-2]
 return T[n];
 }
```

EFICIENCIA: $\Theta(n)$.

□

La técnica DyV se caracteriza por ser un método descendente: parte del problema original y descompone recursivamente en problemas de menor tamaño (o más sencillos).
La técnica PD se caracteriza por ser un método ascendente: resuelve primero los problemas más pequeños, guardando las soluciones en una tabla. Después se combinan para obtener la solución a problemas más grandes.

4.2 El Principio de Optimalidad de Bellman

Antes de aplicar un diseño basado en PD para resolver un problema, debemos comprobar si la solución basada en PD cumple el principio de optimalidad de Bellman (P.O.B). Si se cumple garantizamos que obtenemos una solución correcta y es posible aplicar PD.

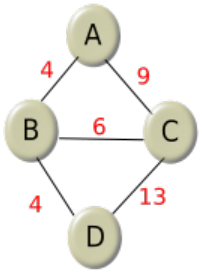
Definición 4.2.1 Principio de Optimalidad de Bellman La solución óptima de un problema se obtiene combinando soluciones óptimas de subproblemas. Esto quiere decir que cualquier subsecuencia de una secuencia óptima debe ser a su vez una secuencia óptima. ♠

Ejemplo 4.2.1
Supongamos que queremos obtener el camino más corto entre dos nodos A y B en un grafo. La forma de aplicar el principio de optimalidad de Bellman sería:
Si tengo un camino C_{AB} que es el camino óptimo para ir de A a B

$$C_{AB} = An_1n_2\cdots n_k\cdots n_tB$$

entonces $\forall k \in [1,t]$ el camino $An_1n_2\cdots n_k$ es óptimo en el sentido que es el camino más corto para ir desde A a n_k y por otro lado el camino $n_kn_{k+1}\cdots n_tB$ es el más corto para ir de n_k a B. □
El principio de Optimalidad no siempre se cumple y por lo tanto no podemos aplicar siempre un algoritmo basado en PD para resolver un problema. El siguiente ejemplo muestra una situación donde no se cumple el principio de optimalidad de Bellman.

Ejemplo 4.2.2
Encontrar el camino de mayor longitud entre dos nodos en un grafo. Para mostrar que no se cumple el principio de optimalidad de Bellman observemos el siguiente grafo:



Grafo G

El camino de mayor longitud de A a D es $A-B-C-D$ con una longitud de 23. Aplicando el principio de Optimalidad de Bellman deducimos que no se cumple. Así si $A-B-C-D$ es óptimo cualquier subcamino debe serlo para el subproblema. Supongamos el subproblema de ir de A a B si se cumpliera el principio de optimalidad de Bellman debería ser $A-B$ con longitud 4. Pero el camino $A-C-D-B$ tiene longitud 26. Por lo tanto no se cumple el P.O.B. □

4.3 Pasos para aplicar PD

Los pasos a seguir para aplicar PD son los siguientes:

1. Comprobar que se cumple el Principio de Optimalidad de Bellman. En el caso de que se cumpla
2. Obtener una ecuación recurrente del problema, compuesta por:
   - Caso base (uno o más)
   - Caso general
3. Definir la estrategia de aplicación de la fórmula
   - Tablas utilizadas para el algoritmo
   - Orden y formas de rellenarlas
4. Especificar como se recompone la solución final a partir de los valores de las tablas.

4.3.1 El problema de la Mochila

Para ver el modo de funcionamiento de la técnica PD vamos a estudiar como se soluciona el problema de la Mochila. La Mochila es la Mochila 0/1, es decir se echa el objeto entero o no so echa. No se permite partir el objeto.

Datos del problema son

- n: numero de objetos disponibles
- M: capacidad de la mochila
- $p = (p_1, p_2, \cdots, p_n)$ pesos de los objetos
- $b = (b_1, b_2, \cdots, b_n)$ beneficios de los objetos.

El objetivo es llenar la mochila con los objetos disponibles, sin superar la capacidad, de forma que se maximice el beneficio.

Obtener la ecuación recurrente. Para idear la ecuación recurrente hay que tener en cuenta dos aspectos:

- En cada paso podemos coger o no un objeto k.
  - Si se coge $\Longrightarrow$ aumentaremos el beneficio acumulado con el beneficio del objeto k y se ocupa el espacio del objeto k en la mochila, por lo tanto tenemos menos espacio en la mochila.

- Si no se coge$\Longrightarrow$ mantenemos la capacidad que nos queda libre en la mochila pero ya tenemos un objeto menos que analizar.

Los subproblemas se distinguen unos de otros en:

- Número de objetos sobre los que decidir
- Peso disponible en la mochila

Lo primero que debemos hacer es comprobar que se cumple el principio de optimalidad de Bellman, pero en este caso para asimilar la técnica vamos a dejar esta comprobación al final.
Veamos los pasos a seguir con el problema de la mochila:

Paso 1: Ecuación recurrente

$Mochila(k,m)$: devuelve el valor de beneficio total obtenido para una mochila de capacidad m teniendo que analizar k objetos.

$$Mochila(k,m) = \begin{cases} 0 & m = 0 \,\|\, k = 0 \\ -\infty & k < 0 \,\|\, m < 0 \\ max(\underbrace{Mochila(k-1,m)}_{\text{no coge el objeto } k}, \underbrace{Mochila(k-1,m-p_k)+b_k}_{\text{coge el objeto } k}) & \text{en otro caso} \end{cases}$$

Paso 2:Definición de la tabla y como rellenarla

Se define una tabla V para guardar los resultados de los subproblemas, guardando el beneficio obtenido en cada subproblema: $V[i][j] = Mochila(i,j)$, siendo el beneficio que obtiene para i objetos con una capacidad de mochila j.
La solución del problema original estará en $V[n][M] = Mochila(n,M)$. La tabla tendrá $n+1$ filas y $M+1$ columnas. La fila 0 y la columna 0 se rellenan con ceros (el primer caso base en la ecuación recurrente).
Forma de rellenar la tabla.
Inicializar

- $V[i][0] = 0 \;\; \forall \;\; i = 0 \cdots n$
- $V[0][j] = 0 \;\; \forall \;\; j = 0 \cdots M$

Para los subproblemas

Para todo $i = 1 \cdots n$
  Para todo $j = 1 \cdots M$
  if $(j - p_i > 0)$
   $V[i][j] = max(\underbrace{V[i-1][j]}_{\text{no coge el objeto } i}, \underbrace{V[i-1][j-p_i]+b_i)}_{\text{coge el objeto } i}$
  else
   $V[i][j] = V[i-1][j]$

Ejemplo 4.3.1
Supongamos que tenemos un número de objetos $n = 3$, con una mochila de capacidad $M = 6$ y el vector de pesos y beneficios son:
p=(2,3,4)
b=(1,2,5)

Tras el paso de inicialización nuestra tabla queda como sigue:

		Capacidades de las mochilas para los subproblemas						
	0	0	0	0	0	0	0	0
Objetos	1	0						
	2	0						
	3	0						

A continuación pasamos a rellenar el resto de casillas, empezando por el primer objeto. Este objeto no puede insertarse hasta que no tengamos al menos una mochila de capacidad 2.

		Capacidades de las mochilas para los subproblemas						
		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
Objetos	1	0	0	1	1	1	1	1
	2	0	0	1	2	2	3	3
	3	0	0	1	2	5	5	6

El beneficio total se puede consultar en la casilla $V[3][6]$ siendo este valor 6.

Para recuperar la solución debemos de partir de la casilla $V[3][6]$ y a continuación ver si el valor que se ha obtenido, 6 viene porque ha sido heredado de la casilla de arriba (porque tambien $V[2][6]$ tiene un valor de 6) o porque se ha obtenido como $V[2][2]+5$, que es el caso. Por lo tanto como $V[3][6]$ es distinto de $V[2][6]$ el objeto 3 se incluye en la mochila. A continuación nos situamos en la casilla $V[2][2]$ y miramos si el valor 1, tambien lo tienen $V[1][2]$ que es el caso, por lo tanto el objeto 2 no se echa. Subimos a $V[1][2]$ y como tiene un valor diferente a $V[0][2]$ entonces el objeto 1 se echa a la mochila. Ahora nos deberíamos situar en $V[0][0]$, y como $i=0$ paramos.

		Capacidades de las mochilas para los subproblemas						
		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
Objetos	1	0	0	1	1	1	1	1
	2	0	0	1	2	2	3	3
	3	0	0	1	2	5	5	6

□

Ejemplo 4.3.2

Datos del problema son

- n: 4 numero de objetos
- M: 10 capacidad de la mochila
- $p=(2,2,7,6)$ pesos de los objetos
- $b=(4,1,8,1)$ beneficios de los objetos.

La tabla solución y el camino a seguir para reconstruir la solución se da a continuación.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4	4	4	4	4
2	0	0	4	4	5	5	5	5	5	5	5
3	0	0	4	4	5	5	5	8	8	12	12
4	0	0	4	4	5	5	5	8	8	12	12

□

Paso 3:Recuperar la solución óptima

Objetivo.- Es deducir la tupla $(x_1, x_2, \cdots, x_n)$ solución a partir de la tabla V, indicando para cada x_i si adopta un valor 0 porque el objeto i no se echa o aportando un valor $x_i = 1$ porque el objeto i si se coge. Los pasos esenciales son:

Partimos de $V[n][M]$

Si $V[i][j] == V[i-1][j]$ entonces

$x_i \leftarrow 0$

else

Si $V[i][j] == V[i-1][j-p_i] + b_i$ entonces

$x_i \leftarrow 1$

Nota.- Puede que se cumpla las dos condiciones entonces tenemos las dos opciones: coger o no coger el objeto. Esto significa que existe más de una solución óptima.

Eficiencia.- La eficiencia para recuperar la solución en el peor de los casos implica que se cogen todos los objetos, y en tal caso se consume un tiempo en $\mathbf{O}(n)$. Para rellenar la tabla, que tiene $n+1$ filas y $M+1$ columnas necesita un tiempo de $\Theta(n \times M)$.

Con respecto a la eficiencia en Memoria de la técnica de PD, debe ser un parámetro a considerar, ya que construir la tabla necesaria para almacenar las soluciones de los subproblemas puede necesitar mucha memoria, y por lo tanto la implementación del algoritmo no es viable.

Paso 0:Principio de Optimalidad de Bellman

En esta sección vamos a comprobar que nuestro algoritmo cumple el P.O.B. Para ello supongamos que para nuestro problema original con una mochila de capacidad M y con n objetos tenemos una solución óptima:

$$S = (x_1, x_2, \cdots, x_n)$$

en este caso por P.O.B para todo k debe cumplirse que :

$$S_1 = (x_1, x_2, \cdots, x_k)$$
$$\text{y}$$
$$S_2 = (x_{k+1}, \cdots, x_n)$$

son soluciones óptimas al problema de poner k objetos en una mochila con espacio M_1 y al problema de poner $n-k$ objetos con una mochila con M_2 siendo $M = M_1 + M_2$.

Supongamos que existe un $k = l$ tal que cambia la elección sobre el objeto l a $\hat{x}_l$, y además es una solución óptima $(x_1, \cdots, \hat{x}_l, x_{l+1}, \cdots, x_n)$ de forma que el beneficio del subproblema es $(x_1, \cdots, \hat{x}_l)$ es mayor que el beneficio del subproblema de partida es decir:

$$B((x_1, \cdots, \hat{x}_l) > B(x_1, \cdots, x_l)$$

Si nuestro beneficio de la solución óptima de partida era:

$$B_T = \sum_{i=1}^{n} b(x_i) = \sum_{i=1}^{l-1} b(x_i) + b(x_l) + \sum_{i=l+1}^{n} b(x_i) < \sum_{i=1}^{l-1} b(x_i) + b(\hat{x}_l) + \sum_{i=l+1}^{n} b(x_i) < \hat{B}_T$$

Por lo tanto llegamos a la contradicción de que nuestra solución de partida no era óptima. Y esto no puede ser.

4.4 Problema: El cambio de Monedas

El objetivo de este problema es dada una cantidad P y un conjunto de n tipos de monedas disponibles en una cantidad ilimitada, encontrar el mínimo número de monedas que tenemos que usar para obtener esa cantidad.

Paso 1:Descomposición recurrente del problema

IDEA.-. Interpretar el problema como una toma de decisiones entre coger o no coger una moneda de tipo k. De esta forma las situaciones posibles tras decidir son:

- Si se coge la moneda de tipo k: usamos una moneda más y tenemos que devolver la cantidad menos el valor de la moneda k c_k. Esta moneda se vuelve a considerar por si se tuviera que echar otra vez.
- Si no se coge la moneda de tipo k: el número de monedas usadas son las mismas y la cantidad a devolver también es la misma. Por otro lado ya no tenemos que considerar más la moneda de tipo k.

¿Qué varía en los diferentes subproblemas?:

- Los tipos de monedas que podemos usar
- La cantidad a devolver.

Estos parámetros son los parámetros que definen las dimensiones de la tabla.

Ecuación Recurrente.-. La ecuación recurrente parte de $Cambio(k,q)$ siendo k los tipos de monedas a considerar y q la cantidad a devolver. Esta función devuelve el número mínimo de monedas a usar para devolver la cantidad q, con k tipos de monedas:

$$Cambio(k,q) = \begin{cases} min(\underbrace{Cambio(k-1,q)}_{\substack{\text{no se coge la moneda} \\ \text{de tipo } k}}, \underbrace{Cambio(k,q-c_k)+1}_{\substack{\text{se coge 1 moneda} \\ \text{de tipo } k}}) & \text{en otro caso} \\ \infty & k \leq 0 \text{ o } q < 0 \\ 0 & q = 0 \end{cases}$$

siendo c_k el valor de la moneda k.

Paso 2:Estrategia de aplicación de la recurrencia

Se define una tabla con n filas, siendo n el número de tipos de monedas. El número de columnas es $P+1$ siendo P la cantidad a devolver. La columna 0 se pone 0, ya que el número de monedas a devolver con una cantidad 0 es 0. En cada iteración se decide si se coge o no se coge la moneda de tipo i para cada una de las cantidades q con $1 \leq q \leq P$.

Ejemplo 4.4.1

Datos del Problema

- Numero de tipos de monedas. $n = 3$
- Cantidad a devolver. $P = 8$
- Valor de las monedas $c = (1,4,6)$.

Usando la recurrencia $Cambio(k,q)$ rellenamos nuestra tabla de forma $T[k][q] = Cambio(k,q)$:

		Cantidades a devolver								
		0	1	2	3	4	5	6	7	8
Tipos de Monedas	1 $c_1 = 1$	0	1	2	3	4	5	6	7	8
	2 $c_2 = 4$	0	1	2	3	1	2	3	4	2
	3 $c_3 = 6$	0	1	2	3	1	2	1	2	2

La solución está en $T[2][8]$ (si indexamos desde 0 tanto para columnas como filas). Así el número de monedas necesarias son 2.

□

Paso 3:Como obtener las monedas

A partir de la tabla debemos recuperar cuales son los tipos de monedas elegidas. Los pasos a seguir son los siguientes:

$i \leftarrow n-1$

$j \leftarrow P$

Mientras $i \geq 1$ AND j>0

Si $i-1 \geq 0$ AND $T[i][j] == T[i-1][j]$ entonces

  $i \leftarrow i-1$

else

  $x_i \leftarrow x_i + 1$

  $j \leftarrow j - c_i$

Ejemplo 4.4.2

Obtener las monedas usadas en el ejemplo 4.4.1.

		Cantidades a devolver								
		0	1	2	3	4	5	6	7	8
Tipos de Monedas	1 $c_1 = 1$	0	1	2	3	4	5	6	7	8
	2 $c_2 = 4$	0	1	2	3	1	2	3	4	2
	3 $c_3 = 6$	0	1	2	3	1	2	1	2	2

En la tabla se muestra que partiendo de $T[2][8]$ como este valor 2 es igual al que se almacena en $T[1][8]$ esto significa que la moneda 3 no se coge. En cambio la moneda 2 si se coge y además dos veces. Siendo el vector solución $S = (0,2,0)$.

□

Ejemplo 4.4.3

Datos del Problema

- Numero de tipos de monedas. $n = 3$
- Cantidad a devolver. $P = 8$
- Valor de las monedas $c = (2,4,6)$.

Este ejemplo es parecido al ejemplo 4.4.1, la única diferencia es que la moneda 1 tiene valor 2. Es un ejemplo en el que podemos obtener diferentes soluciones. Veamos la tabla:

		Cantidades a devolver								
		0	1	2	3	4	5	6	7	8
Tipos de Monedas	1 $c_1 = 2$	0	∞	1	∞	2	∞	3	∞	4
	2 $c_2 = 4$	0	∞	1	∞	1	∞	2	∞	2
	3 $c_3 = 6$	0	∞	1	∞	1	∞	2	∞	2

En este caso $T[2][8]$ podría obtenerse desde la solución que coge 6+2 o la solución que coge 4+4. Ya que $T[2][8]$ se obtiene como $T[2][2]+1$ o tambien como $T[1][8]$.

□

A continuación veremos el código $C++$ de este problema:

```
/**
  @brief Obtiene el numero de monedas minimo que suman una cantidad
  @param P: cantidad a devolver
  @param C: vector con los valores de las monedas
  @param S: vector solucion. S[i] representa el numero de monedas de valor C[i]
  @return numero de monedas
**/
int Cambio(int P, const vector<int> &C, vector<int> &S){
int n = C.size(); //tipos de monedas
Matriz T(n,P+1);//tabla con las soluciones
// Inicializacion para la cantidad 0

```

```
13  for (int i=0;i<n; i++) T[i][0]=0;
14
15  //Relleno de la tabla y la solucion
16  for (int i=0;i<n;i++)
17   for (int j=1;j<=P;j++){
18     int v1; // si no se coge
19     if (i-1<0) v1= numeric_limits<int>::max();
20     else
21       v1= T[i-1][j];
22     int v2; //si se coge
23     if (j-C[i]<0) v2=numeric_limits<int>::max();
24     else
25      v2=T[i][j-C[i]]+1;
26     //el minimo entre coger y no coger la moneda de tipo i
27     T[i][j]=min(v1,v2);
28
29   }
30
31   //Ahora obtenemos la solucion
32   int i= n-1;
33   int j= P;
34   while (i>=0 && j>0){
35     int v1;
36     if (i-1<=0)
37       v1= numeric_limits<int>::max();
38     else
39       v1= T[i-1][j];
40     int v2; //si se coge
41     if (j-C[i]<0)
42       v2=numeric_limits<int>::max();
43     else
44       v2=T[i][j-C[i]]+1;
45     if (T[i][j]==v2){
46       S[i]=S[i]+1;
47       j= j-C[i];
48     }
49     else {
50      i = i-1;
51     }
52    }
53   }
54
55
```

```
56    return T[n-1][P];
57  }
```

Eficiencia.-. El gasto en eficiencia para rellenar la tabla es $\Theta(n \times P)$. Y en el peor caso el tiempo para obtener el vector solución es $\mathbf{O}(n+P)$.

Paso 0:Principio de Optimalidad de Bellman

En esta sección vamos a comprobar que nuestro algoritmo cumple el P.O.B. Para ello supongamos que para nuestro problema original con n tipos de monedas y la cantidad a devolver P es

$$S = (x_1, x_2, \cdots, x_n)$$

en este caso por P.O.B para todo k debe cumplirse que :

$$\begin{gathered} S_1 = (x_1, x_2, \cdots, x_k) \\ \text{y} \\ S_2 = (x_{k+1}, \cdots, x_n) \end{gathered}$$

son soluciones óptimas al problema de k tipos de monedas para devolver la cantidad P_1 y al problema de escoger entre $n-k$ tipos de monedas para devolver la cantidad P_2, siendo $P_1 + P_2 = P$.
Supongamos que existe un $k = l$ tal que cambia la elección del numero de monedas que se escoge de tipo l a $\hat{x}_l$, y además es una solución óptima $(x_1, \cdots, \hat{x}_l, x_{l+1}, \cdots, x_n)$ de forma que el numero de monedas en $(x_1, \cdots, \hat{x}_l)$ es menor que el numero de monedas para el subproblema de partida es decir:

$$x_1 + \cdots + \hat{x}_l < x_1 + \cdots + x_l$$

Si el número de monedas de nuestra solución óptima era:

$$\sum_{i=1}^{l-1} x_i + x_l + \sum_{i=l+1}^{n} x_i > \sum_{i=1}^{l-1} x_i + \hat{x}_l + \sum_{i=l+1}^{n} x_i$$

Por lo tanto llegamos a la contradicción de que nuestra solución de partida no era óptima. Y esto no puede ser.

4.5 Caminos Mínimos. Algoritmo de Floyd

El objetivo de este problema es encontrar en un grafo $G = (N, A)$ el camino más corto de un nodo a cualquier otro. El algoritmo de Floyd sigue la siguiente mecánica:

- Parte de la matriz de pesos de ir de un vértice a otro (matriz de adyacencia).
- En cada iteración se fija un nodo k. Y entre cada par de nodos i, j se resuelve el problema de si es mejor pasar por k para ir desde i a j, o por el contrario es mejor dejar el camino existente entre i y j, porque es más corto.

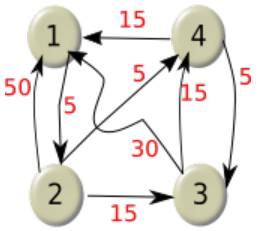
Sea C la matriz de adyacencia. $D_k(i, j)$ es la distancia del camino de i a j cuando se plantea si se pasa por k.

- Para $k = 0$ hacemos $D_k = C$.

- Para $k > 0$ $\forall i, j$ vértices

$$D_k(i,j) = min(\underbrace{D_{k-1}(i,j)}_{\text{Dejar el camino previo}}, \underbrace{D_{k-1}(i,k) + D_{k-1}(k,j)}_{\text{Pasar por el nodo } k \text{ para ir de } i \text{ a } j})$$

Ejemplo 4.5.1
Aplicar el algoritmo de Floyd al grafo siguiente:



Con matriz de adyacencia:

$$C = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

Partimos con $D_0 = C$ empezamos con $k = 1$ lo que significa si se mejora pasando por el vértice 1 para ir de un nodo a cualquier otro. Se puede comprobar que para la fila y columna 1 no va a existir ninguna modificación.

- $D_1(2,3) = min(\underbrace{D_0(2,3)}_{15}, \underbrace{D_0(2,1)}_{50} + \underbrace{D_0(1,3)}_{\infty}) = 15$
- $D_1(2,4) = min(\underbrace{D_0(2,4)}_{5}, \underbrace{D_0(2,1)}_{50} + \underbrace{D_0(1,4)}_{\infty}) = 5$
- $D_1(3,2) = min(\underbrace{D_0(3,2)}_{\infty}, \underbrace{D_0(3,1)}_{30} + \underbrace{D_0(1,2)}_{5}) = 35$
- $D_1(3,4) = min(\underbrace{D_0(3,4)}_{15}, \underbrace{D_0(3,1)}_{30} + \underbrace{D_0(1,2)}_{\infty}) = 15$
- $D_1(4,2) = min(\underbrace{D_0(4,2)}_{\infty}, \underbrace{D_0(4,1)}_{15} + \underbrace{D_0(1,2)}_{5}) = 20$
- $D_1(4,3) = min(\underbrace{D_0(4,3)}_{5}, \underbrace{D_0(4,1)}_{15} + \underbrace{D_0(1,3)}_{\infty}) = 5$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \boxed{35} & 0 & 15 \\ 15 & \boxed{20} & 5 & 0 \end{pmatrix}$$

A continuación se muestra las diferentes matrices D_k:

$$D_2 = \begin{pmatrix} 0 & 5 & \boxed{20} & \boxed{10} \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ \boxed{45} & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_4 = \begin{pmatrix} 0 & 5 & \boxed{15} & 10 \\ \boxed{20} & 0 & \boxed{10} & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Con D_4 tenemos la información de como ir de un vértice a otro de la forma más económica. Por ejemplo si queremos ir de 1 a 3, consultamos $D_3(1,3) \neq D_4(1,3)$ al ser diferente nos indica que para ir de 1 a 3 el camino será algo así $1-\cdots 4-3$. Ahora nos preguntamos como se va de 1 a 4. Para ello miramos en $D_3(1,4)$ y comprobamos que coincide con $D_2(1,4)$ pero $D_2(1,4) = 20$ y diferente de $D_1(1,4)$ por lo tanto para ir a 4 desde 1 hay que pasar por 2. Luego nuestro problema original de ir de 1 a 3 se va refinando como $1-\cdots 2-4-3$. Ahora como se llega a 2 desde 1. Para ellos comprobamos que $D_1(1,2) == D_0(1,2)$ por lo tanto nuestro camino es $1-2-4-3$ con costo $D_4(1,3) = 15$. Para no tener que hacer estas comprobaciones de consultar D en diferentes k podríamos tener una matriz que nos indique por donde debemos pasar para llegar al nodo. Esta se modifica cuando se obtiene el mínimo por el segundo término de la formulación de $D_k(i,j)$. Así en nuestro ejemplo esta matriz llamémosla P y sería para el ejemplo:

$$P = \begin{pmatrix} 0 & 1 & 4 & 2 \\ 4 & 0 & 4 & 2 \\ 3 & 1 & 0 & 3 \\ 4 & 1 & 4 & 0 \end{pmatrix}$$

Esta matriz se inicializa como $P(i,j) = i$ salvo cuando $i = j$ que se pone 0. Por ejemplo para ir de 1 a 3 consultamos $P(1,3)$ y obtenemos 4. Lo que quiere decir que hay que ir por 4. Como pasar de 1 a 4 consultamos $P(1,4) = 2$, y finalmente como ir $P(1,2) = 1$. Luego tenemos que el camino es $1-2-4-3$. □

Veamos el código C++ del algoritmo de Floyd

```
/**
 @brief Obtiene el camino minimo entre cualesquiere vertices
 @param C: matriz de adyacencia de un grafo.
 @param D: matriz de distancia minima entre dos vertices
 @param P: matriz que contiene el vertice por el que debemos pasar para
 ir desde un vertice a otro
**/
void Floyd(const Matriz &C, Matriz &D, Matriz &P){
    int n= C.numfilas();
    //Inicializacion
     D=C;
     for (int i=0;i<n;i++)
     for (int j=0;j<n;j++){
       if (i==j) P(i,j)=0;
       else P(i,j)=i;
```

```
16       }
17
18      for (int k=0;k<n;k++)//por cada vertice
19       for (int i=0;i<n;i++)
20        for (int j=0;j<n;j++){
21          if (k!=i && k!=j){
22             int v1= D(i,j);
23             int v2= D(i,k)+D(k,j);
24             if (v1>v2){
25                D(i,j)=v2;
26                P(i,j)=k;
27              }
28          }
29        }
30
31 }
```

Ahora con P podemos obtener el camino entre cualesquiera vértices de la siguiente forma:

```
1  /**
2   @brief Obtiene el camino de menor longitud entre dos vertices
3   @param origen: vertice de partida
4   @param destino: vertice donde terminara el camino
5   @param P: matriz que indica por que vertice hay que pasa para ir
6             desde un vertice a otro.
7  */
8  vector<int> ObtenCamino(int origen,int destino,const Matriz &P){
9   int d=destino;
10  vector<int> camino;
11  camino.push_back(destino);
12  while (d!=origen){
13   d=P(origen,d);
14   camino.push_back(d);
15  }
16  camino.push_back(origen);
17  //invertimos el vector.
18  reverse(camino.begin(),camino.end());
19  return camino;
20 }
```

Eficiencia.- La eficiencia de Floyd es $\Theta(n^3)$. Para obtener el camino entre dos vértices cualesquiera nos costaría en el peor de los casos $\mathbf{O}(n)$.

Ejercicio 4.1
Comprobar que el algoritmo de Floyd cumple el Principio de Optimalidad de Bellman. □

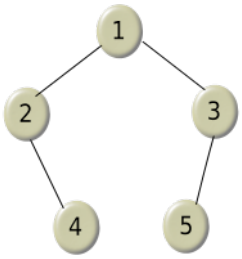
4.6 Existe Camino. Algoritmo de Warshall

Al igual que el algoritmo de Floyd en el que encuentra caminos de longitud mínima entre dos vértices en un grafo, Warshall hace algo parecido. El algoritmo de Warshall dice si existe un camino entre dos vértices. Para un grafo partimos de una matriz $L(i,j)$ con valores 0 si no existe camino entre i y j, o 1 si existe. Igual que con Floyd se construye una matriz D es una matriz con 0 y 1, de forma que si $D(i,j) = 1$ indica que existe un camino entre los vértices i y j. En el caso de que no haya camino entre i y j, el algoritmo de Warshall puede encontrar un camino, si existe, pasando por otros vértices. Por ejemplo podemos ir de i a j pasando por k, esto ocurre cuando $D(i,k) AND D(k,j) = 1$.
La ecuación de recurrencia se puede plantear como:

- Para $k = 0$ hacemos $D_k = L$.
- Para $k > 0$ $\forall i, j$ vértices

$$D_k(i,j) = \underbrace{D_{k-1}(i,j)}_{\text{Si existe camino previo}} \quad || \underbrace{(D_{k-1}(i,k) \&\& D_{k-1}(k,j))}_{\text{Pasar por el nodo } k \text{ para ir de } i \text{ a } j}$$

Ejemplo 4.6.1
Dado el siguiente grafo con la matriz L, vamos a aplicar el algoritmo de Warshall



$$L = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Partimos para $k = 0$ que $D_k = L$. Las siguientes iteraciones obtienen la siguientes matrices:

$$D_1 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & \boxed{1} & 1 & 0 \\ 1 & \boxed{1} & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \quad D_2 = \begin{bmatrix} 1 & 1 & 1 & \boxed{1} & 0 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & \boxed{1} & 1 \\ \boxed{1} & 1 & \boxed{1} & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix} \quad D_3 = \begin{bmatrix} 1 & 1 & 1 & 1 & \boxed{1} \\ 1 & 1 & 1 & 1 & \boxed{1} \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & \boxed{1} \\ \boxed{1} & \boxed{1} & 1 & \boxed{1} & 1 \end{bmatrix}$$

En este caso tenemos camino desde cualquier vértice a cualquier otro. El código C++ de este algoritmo podría ser el siguiente:

```
1  void Warshall(const Matriz<bool> &L,Matriz<bool> &D,int Matriz<int> &P){
2      int n= L.numfilas();
3      //inicializacion
4      D=L,
5
6      for (int k=0;k<n;k++)
7        for (int i=0;i<n;i++)
```

```
8           for (int j=0;j<n;j++){
9              boo aux=D(i,j);
10             D(i,j)=D(i,j) || (D(i,k) && D(k,j));
11             if (aux!=D(i,j))
12               //anotamos por que vertice debemos
13               //pasar para ir de i a j
14                P(i,j)=k;
15          }
16  }
```

□

Como se puede observar el Algoritmo de Warshall tiene un eficiencia $\Theta(n^3)$.

4.7 Problema: Intereses Bancarios

Sea $f_i(x_t)$ el interés del banco i cuando invertimos una cantidad x_t. Supongamos que tenemos una cantidad de dinero M y quiero invertir en n bancos las cantidades $x_1, x_2, \cdots, x_n$ tal que $\sum_{i=1}^{n} = M$. El objetivo es hacer la mejor inversión para obtener el mayor interés en conjunto. Es decir

$$maximizar \sum_{i=1}^{n} f_i(x_i) \text{ sujeto a } x_1 + x_2 + \cdots + x_n = M$$

Datos del Problema

- n: número de bancos en los que invertir
- $x_1, x_2, \cdots, x_n$: inversión en los N campos.
- M: cantidad a invertir.
- $I_n(M) = \sum_{i=1}^{n} f_i(x_i) = \underbrace{f_1(x_1) + f_2(x_2) + \cdots + f_n(x_n)}_{\text{Interés máximo obtenido al invertir } M \text{ en los } n \text{ bancos.}}$

Paso 0:Principio de Optimalidad de Bellman

Si $I_n(M)$ es una secuencia de decisiones (invertir una cantidad o no en el banco) y resulta ser óptima para el problema de invertir una cantidad M en n bancos, cualquiera de las subsecuencias de decisiones a de ser óptima. Así la cantidad $M - x_n$ cuando se invierte en los $n-1$ bancos debe ser óptima:

$$I_{n-1}(M - x_n) = \sum_{i=1}^{n-1} f_i(x_i) = \underbrace{f_1(x_1) + f_2(x_2) + \cdots + f_{n-1}(x_{n-1})}_{\text{Interés máximo obtenido al invertir } M - x_n \text{ en los } n-1 \text{ bancos.}}$$

Paso 1:Ecuación de recurrencia

Sea $I_n(x)$ el interés máximo obtenido cuando se invierte la cantidad x en n bancos. En el caso de que solamente tengamos un banco, $n = 1$ invertimos la cantidad x entera en el banco. En otro caso analizamos que cantidad invertir en el n-ésimo banco y que interés

produce. Este interés sumado al interés maximo que se obtiene al invertir la cantidad restante en los $n-1$ bancos debe ser máximo. Así podemos definir la ecuación como:

$$I_n(x) = \begin{cases} f_1(x) & n = 1 \\ Max_{0\leq t\leq x}(I_{n-1}(x-t)+f_n(t)) & n > 1 \end{cases}$$

Paso 2:Estrategia de la aplicación de la recurrencia

Para resolver el problema vamos a utilizar una matriz I de dimensión n con objeto de obtener $I_n(M)$ en la casilla $I(n,M)$. Así si $I(i,j)$ representa el interés de j euros cuando se dispone de i bancos. También disponemos de la matriz F, de dimensión $n \times M+1$, que indica por cada banco y una cantidad el interés que da el banco.
Se inicializa para todos los bancos con cantidad 0 a interés 0 $\Longrightarrow I(i,0) = 0$

Ejemplo 4.7.1

Supongamos que queremos invertir $M = 9$ en tres bancos $n = 3$. La matriz F, que me dice el interés que da cada banco por cantidad es la siguiente:

$$F = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1{,}2 & 1{,}4 & 4 & 4{,}5 \\ 0{,}1 & 0{,}2 & 0{,}3 & 0{,}4 & 0{,}5 & 1 & 1{,}35 & 2{,}4 & 3{,}6 \\ 0 & 0 & 0 & 0{,}45 & 0{,}45 & 1 & 1{,}25 & 3 & 8{,}1 \end{pmatrix}$$

Con esta información podemos empezar a rellenar la tabla I:

		Diferentes cantidades a invertir									
		0	1	2	3	4	5	6	7	8	9
Bancos	1	0	0	0	0	0	0	1.2	1.4	4	4.5
	2	0	0.1	0.2	0.3	0.4	0.5	1.2	1.4	4	4.5
	3	0	0.1	0.2	0.3	0.45	0.55	1.2	1.4	4	8.1

Por ejemplo para obtener el valor de $I(3,5) = 0{,}55$ se ha tenido que hacer los siguiente cálculos:

Cantidad t	Interés máximo obtenido en los bancos 1 y 2	Interés en el banco 3	Interés total
t=0	I(2,5)=0.5	0	0.5
t=1	I(2,4)=0.4	0	0.4
t=2	I(2,3)=0.3	0	0.3
t=3	I(2,2)=0.2	0	0.2
t=4	I(2,1)=0.1	0.45	0.55
t=5	I(2,0)=0	0.45	0.45

El valor de 0.55 se consigue cuando se invierte 4 en el banco 3 (0.45) y el interés máximo obtenido por los anteriores bancos es 0.1.

□

Veamos el código C++

```
1 /**
2   @brief Obtiene el interes maximo al invertir una cantidad en un conjunto de
```

```
3            bancos.
4     @param M: cantidad a invertir
5     @param F: intereses que da cada banco por cada una de las cantidades 0-M
6     @param I: maximo interes obtenido para cada uno de los subproblemas.
7     @return el interes maximo obtenido al invertir M en los n bancos.
8  */
9  int Interes(int M,const Matriz &F, Matriz &I){
10  int n=I.numfilas(); //numero de bancos
11  //Inicializacion
12  for (int i=0;i<n;i++)
13   I(i,0)=0;
14
15  //el primer banco
16  for (int j=1;j<=M;j++) I(0,j)=F(0,j);
17
18  //para los siguiente bancos
19  for (int i=1;i<n;i++)
20   for (int j=1;j<=M;j++)
21      I(i,j)=Max(I,F,i,j);
22   return I(n-1,M);
23 }
```

La función Max obtiene la mejor distribución de una cantidad en los distintos bancos para obtener el interés máximo.

```
1  int Max(const Matriz &I, const Matriz &F, int i,int j){
2   //F(i,0) sera 0
3   int max = I(i-1,j)+F(i,0);
4
5   for (int t=1;t<=j;t++)
6     max =Maximo(max,I(i-1,j-t)+F(i,t));
7   return max;
8  }
```

Eficiencia.-. En el peor de los casos para rellenar la tabla nos cuesta $\mathbf{O}(n \times M^2)$.

Ejercicio 4.2
Dar el algoritmo para en el problema del interés máximo, una vez calculada la tabla I, encontrar la inversión que se hace en cada banco. □

4.8 Problema: Subsecuencia común máxima

Dada una secuencia $X = \{x_1, x_2, \cdots, x_m\}$ decimos que $Z = \{z_1, z_2, \cdots, z_k\}$ es una subsecuencia de X (siendo $k \leq m$) si existe una secuencia creciente de índices (en X) $\{i_1, i_2, \cdots, i_k\}$ tales que para todo $j = 1, 2, \cdots, k$ tenemos que $x_{i_j} = z_j$.

Dadas dos secuencias X e Y decimos que Z es una subsecuencia común de X e Y si es subsecuencia de X y subsecuencia de Y. El objetivo en este problema es obtener la subsecuencia de longitud máxima común.

Ejemplo 4.8.1
Supongamos que X e Y contienen:

$$X = \{\ 1\quad 0\quad 0\quad 1\quad 0\quad 1\quad 0\quad 1\quad \}$$

$$Y = \{\ 0\quad 1\quad 0\quad 1\quad 1\quad 0\quad 1\quad 1\quad 0\ \}$$

Secuencias comunes podría ser las siguientes:

$$\{\ 1\quad 0\quad 1\ \}$$

$$\{\ 0\quad 1\quad 0\quad 1\ \}$$

$$\{\ 0\quad 1\quad 0\quad 1\quad 0\quad 1\ \} \Longleftarrow \text{ la de mayor longitud}$$

La subsecuencia de mayor longitud y donde coinciden con X e Y :

$$\left\{ \begin{array}{cccccc} \overbrace{0}^{} & \overbrace{1}^{} & \overbrace{0}^{} & \overbrace{1}^{} & \overbrace{0}^{} & \overbrace{1}^{} \\ X[3] & X[4] & X[5] & X[6] & X[7] & X[8] \\ Y[1] & Y[2] & Y[3] & Y[5] & Y[6] & Y[8] \end{array} \right\}$$

□

Este problema es el que nos encontramos cuando dadas dos secuencias de ADN queremos encontrar la subsecuencia común para establecer un porcentaje de parentesco.

Paso 1:Ecuación de recurrencia
Sea $L(i,j)$ la longitud de la secuencia común máxima, SCM, de las secuencias X_i e Y_j donde X_i es el prefijo de X hasta el carácter i, y Y_j el prefijo de Y hasta el carácter j. Así tenemos que $X_i = \{x_1, x_2, \cdots, x_i\}$ y $Y_j = \{y_1, y_2, \cdots, y_j\}$.
Se puede plantear la solución como un problema de decisiones en el que en cada paso indicamos si x_i e y_j forman parte de la secuencia común máxima. Formarán parte porque en particular $x_i = y_j$. Con esta idea tenemos:

- Si no se toma x_i e $y_j \rightarrow$ la SCM es la misma que teníamos
- Si se toma x_i e $y_j \rightarrow$ la SCM es la misma que teníamos mas uno más.

La recurrencia por lo tanto podemos plantearla como:

$$L(i,j) = \left\{ \begin{array}{cc} 0 & i = 0 \text{ o } j = 0 \\ max(L(i-1,j), L(i,j-1)) & \text{si } x_i \neq y_j \\ L(i-1,j-1)+1 & \text{si } x_i = y_j \end{array} \right.$$

En la tabla $L(i,j)$ además de la longitud de la subsecuencia de mayor longitud para X_i y Y_j vamos a guardar de donde procede el máximo. Así codificamos la procedencia de la siguiente forma:

- *DIA*:cuando $x_i = y_j$
- *SUP*: el máximo se obtuvo como copia de $L(i-1, j)$.
- *IZQ*: el máximo se obtuvo como copia de $L(i, j-1)$.

Ejemplo 4.8.2
Supongamos que

$$X = \{\ 1\quad 0\quad 0\quad 1\quad 0\quad 1\quad 0\quad 1\quad \}$$

$$Y = \{\ 0\quad 1\quad 0\quad 1\quad 1\quad 0\quad 1\quad 1\quad 0\ \}$$

La tabla $L(i, j)$ sería

			X							
		0	1	0	0	1	0	1	0	1
		0	0	0	0	0	0	0	0	0
	0	0	0 SUP	1 DIA	1 DIA	0 SUP	1 DIA	0 SUP	1 DIA	1 IZQ
	1	0	1 DIA	1 SUP	1 SUP	2 DIA	2 IZQ	2 DIA	2 IZQ	2 DIA
Y	0	0	1 SUP	2 DIA	2 DIA	2 SUP	3 DIA	3 IZQ	3 DIA	3 IZQ
	1	0	1 DIA	2 SUP	2 SUP	3 DIA	3 SUP	4 DIA	4 IZQ	4 DIA
	1	0	1 DIA	2 SUP	2 SUP	3 DIA	3 SUP	4 DIA	4 SUP	5 DIA
	0	0	1 SUP	2 DIA	3 DIA	3 SUP	4 DIA	4 SUP	5 DIA	5 SUP
	1	0	1 DIA	2 SUP	3 SUP	4 DIA	4 SUP	5 DIA	5 SUP	6 DIA
	1	0	1 DIA	2 SUP	3 SUP	4 DIA	4 SUP	5 DIA	5 SUP	6 DIA
	0	0	1 SUP	2 DIA	3 DIA	4 SUP	5 DIA	5 SUP	6 DIA	$\boxed{6}$ SUP

La subsecuencia común máxima tiene longitud 6, ya que es el valor que se almacena en la posición $L(9,8)$. Como se puede observar se ha añadido el valor 0 tanto a X como a Y como valores que nos permite inicializar todo el proceso. Con esta tabla podemos encontrar la subsecuencia común de longitud máxima partiendo de la casilla $L(9,8)$, cuando encontremos la etiqueta *DIA* el carácter de la fila asociada de Y o columna de X, forma parte de la subsecuencia común. En la tabla de la derecha se puede ver el camino que se ha seguido para obtener la secuencia común de mayor longitud.

		X								
		0	1	0	0	1	0	1	0	1
		0	0	0	0	0	0	0	0	0
	0	0	0 SUP	1 DIA	1 DIA	0 SUP	1 DIA	0 SUP	1 DIA	1 IZQ
	1	0	1 DIA	1 SUP	1 SUP	2 DIA	2 IZQ	2 DIA	2 IZQ	2 DIA
Y	0	0	1 SUP	2 DIA	2 DIA	2 SUP	3 DIA	3 IZQ	3 DIA	3 IZQ
	1	0	1 DIA	2 SUP	2 SUP	3 DIA	3 SUP	4 DIA	4 IZQ	4 DIA
	1	0	1 DIA	2 SUP	2 SUP	3 DIA	3 SUP	4 DIA	4 SUP	5 DIA
	0	0	1 SUP	2 DIA	3 DIA	3 SUP	4 DIA	4 SUP	5 DIA	5 SUP
	1	0	1 DIA	2 SUP	3 SUP	4 DIA	4 SUP	5 DIA	5 SUP	6 DIA
	1	0	1 DIA	2 SUP	3 SUP	4 DIA	4 SUP	5 DIA	5 SUP	6 DIA
	0	0	1 SUP	2 DIA	3 DIA	4 SUP	5 DIA	5 SUP	6 DIA	6 SUP

Como se puede observar la secuencia común de mayor longitud es 0 1 0 1 0 1.

□

El código C++ de la función que obtiene la subsecuencia común máxima sería:

```
enum DIRECCION={SUP,IZQ,DIA};
struct info{
  unsigned int valor;
  DIRECCION dir;
};
/**
@brief Obtiene la SCM de dos secuencias
@param X: primera secuencia
@param Y: segunda secuencia
@param L: matriz de info para obtener las SCM de los subproblemas.
          Tiene que tener reservada |Y| filas x|X| columnas. ES MODIFICADA
@return La longitud de la subsecuencia comun maxima
**/
int SubSecMax(const string &X, const string &Y, Matriz<info> &L){
 int n= X.size(); int m= Y.size();
 //Inicializacion
 for (int i=0;i<=m;i++)
   L(0,i).valor=0;
 for (int i=0;i<=n;i++)
   L(i,0).valor=0;
```

```
21
22   //Calculamos los subproblemas
23   for (int i=1;i<=m;i++)
24     for (int j=1;j<=n;j++){
25       if (X[j]==Y[i]){
26        L(i,j).valor=L(i-1,j-1).valor+1;
27        L(i,j).dir=DIA;
28       }
29       else
30        if (L(i-1,j).valor>L(i,j-1).valor){
31          L(i,j).valor=L(i-1,j).valor;
32          L(i,j).dir =SUP;
33        }
34        else {
35         L(i,j).valor=L(i,j-1).valor;
36         L(i,j).dir =IZQ;
37        }
38     }
39     return L(m,n);
40 }
```

Para obtener la SCM una vez calculada la matriz L se puede obtener mediante la siguiente función:

```
1  void EscribirSolucion(const string &X, const string &Y, const Matriz<info> &L,
2                        int i,int j){
3    if (i==0 || j==0) return;
4    else
5     if (L(i,j).dir==DIA){
6       EscribirSolucion(X,Y,L,i-1,j-1);
7       cout<<X[j]<<' ';//tambien vale Y[i]
8     }
9     else
10     if (L(i,j).dir==SUP)
11       EscribirSolucion(X,Y,L,i-1,j);
12     else
13       EscribirSolucion(X,Y,L,i,j-1);
14 }
```

La llamada inicial a *EscribirSolucion* sería *EscribirSolucion(X,Y,L,X.size(),Y.size())*.
Eficiencia.- Rellenar la tabla L cuesta para $n = m$ $\Theta(n^2)$. Descubrir la solución sería $\Theta(n)$

4.9 Problema: Multiplicación Encadenada de Matrices

Para multiplicar dos matrices A y B con dimensiones $p \times r$ y $r \times q$ necesitamos $p \cdot r \cdot q$ multiplicaciones escalares.

Cuando tenemos una secuencia encadenada de matrices para multiplicarlas tal como:

$$M = M_1 \times M_2 \cdots M_n$$

podemos hacerlo de diferentes formas, simplemente cambiando el orden en el que realizas las multiplicaciones.

Ejemplo 4.9.1
Dadas las matrices $A_{13\times5}$, $B_{5\times89}$, $C_{89\times3}$ y $D_{3\times34}$, queremos obtener la multiplicación encadenada $A \times B \times C \times D$. Las diferentes formas de hacerlo y el numero de multiplicaciones escalares que haríamos son:

$$\begin{array}{ll} ((A \times B) \times C) \times D & \text{10582 multiplicaciones} \\ (A \times B) \times (C \times D) & \text{54201 multiplicaciones} \\ A \times ((B \times C) \times D) & \text{4055 multiplicaciones} \\ (A \times (B \times C)) \times D & \underbrace{\text{2856 multiplicaciones}} \\ & \text{Más eficiente. 19 veces mas rápido que la parentización más lenta} \\ A \times (B \times (C \times D)) & \text{26418 multiplicaciones} \end{array}$$

□

Algoritmo basado en la Fuerza Bruta

En este algoritmo ponemos paréntesis de todas las maneras posibles y para cada posibilidad contabilizamos el número de multiplicaciones escalares que se hace. Para ver que esta forma de resolver el problema vamos a obtener la eficiencia de este algoritmo.
Sea $T(n)$ el número de formas diferentes de poner paréntesis en un producto de n matrices. Supongamos que decidimos poner una paréntesis que entre las matrices M_i y M_{i+1} de forma que quedaría:

$$(M_1 \times M_2 \cdots \times M_i) \times (M_{i+1} \times M_{i+2} \cdots \times M_n)$$

Luego el número de parentizaciones diferentes fijado un i será $T(n) = T(i) \cdot T(n-1)$. Pero hay que probar para todo los i posibles, por lo tanto

$$\underbrace{T(n)}_{\text{Números de Catalan}} = \sum_{i=0}^{n-1} T(i) \cdot T(n-1)$$

Resolviendo $T(n)$ obtenemos que en el mejor de los caso $T(n) \in \Omega\left(\frac{4^n}{n^2}\right) \Longleftarrow$ MUCHAS OPERACIONES.

Solución con PD

Sea $M(i,j)$ con $1 \leq i \leq j \leq n$ la tabla que contiene en la posición (i,j) el número de multiplicaciones escalares para multiplicar las matrices $M_i \times M_{i+1} \times \cdots \times M_j$. Nuestra solución se almacenara en $M(1,n)$ ya que indica el numero de multiplicaciones escalares de multiplicar: $M_1 \times M_2 \times M_n$.

Para poder rellenar la matriz nos hará falta un vector $d[0\cdots n]$ en el que se almacena las dimensiones de las matrices. Así para M_1 el numero de filas lo tenemos en $d[0]$ y el número de columnas en $d[1]$, para M_2, tenemos el número de filas en $d[1]$ y el número de columnas en $d[2]$, en general las filas para M_i las tenemos en $d[i-1]$ y las columnas en $d[i]$.
Cuando multiplicamos las matrices $M_i \times M_{i+1} \times M_j$ obtenemos una matriz que tiene dimensiones $d[i-1] \times d[j]$.
La estrategia al multiplicar $M_i \times M_{i+1} \times M_j$ es buscar el k tal que

$$\begin{gathered} A \times B = M_i \times M_{i+1} \times M_j \\ \text{siendo} \\ A = M_i \times M_{i+1} \cdots \times M_k \\ \text{y} \\ B = M_{k+1} \times M_{i+1} \cdots \times M_j \end{gathered}$$

necesite el menor número de multiplicaciones escalares.
Por lo tanto buscaremos el k que cumpla:

$$min_{i\leq k<j}\left(\underbrace{M(i,k)}_{\substack{\text{numero de multiplicaciones}\\ \text{escalares en}\\ M_i\times\cdots\times M_k}} + \underbrace{M(k+1,j)}_{\substack{\text{numero de multiplicaciones}\\ \text{escalares en}\\ M_{k+1}\times\cdots\times M_j}} + \underbrace{d[i-1]\times d[k]\times d[j]}_{\substack{\text{numero de multiplicaciones}\\ \text{escalares en}\\ (M_i\times\cdots\times M_k)\times(M_{k+1}\times\cdots\times M_j)}}\right)$$

Siguiendo esta estrategia podemos definir ahora la ecuación de recurrencia:

$$M(i,j) = \begin{cases} 0 & i = j \text{ diagonal 0 s=0} \\ d[i-1]\times d[i]\times d[j] & \text{si } i+1 = j \text{ diagonal 1 s=1} \\ min_{i\leq k<j}(M(i,k)+M(k+1,j)+d[i-1]\times d[k]\times d[j]) & \end{cases}$$

La ecuación de recurrencia indica que tenemos un paso de inicialización de la diagonal 0 que se inicia $M(i,i) = 0$ para todo i. La matriz se va a rellenar por diagonales, de forma que después de la inicialización se rellena las casilla en las que la diferencia entre i y j sea $s = 1$. Luego se rellena las posiciones en que la diferencia $s = i - j = 2$, y así. El significado de $s = 1$ es que estamos viendo las multiplicaciones que ocurren entre una matriz y la siguiente. Para $s = 2$ estamos contabilizando las multiplicaciones entre una matriz y las dos siguientes, y así para los sucesivos valores de s.

Ejemplo 4.9.2
Supongamos que queremos hacer la siguiente multiplicación encadenada:

$$A_{13\times 5} \times B_{5\times 89} \times C_{89\times 3} \times D_{3\times 34}$$

Con estas matrices el vector d sería $d = (13,5,89,3,34)$. En la fase de inicialización ponemos la diagonal s=0 a 0:

	A	B	C	D
A	0			
B		0		
C			0	
D				0

Diagonal 1 (s=1)

1. $A \times B \Longrightarrow M(1,2) = 5785$
2. $B \times C \Longrightarrow M(2,3) = 1335$
3. $C \times D \Longrightarrow M(3,4) = 9078$

	A	B	C	D
A	0	5785		
B		0	1335	
C			0	9078
D				0

Diagonal 2 (s=2)

1. $A \times B \times C \Longrightarrow$

$$M(1,3) = min(\overbrace{\overbrace{M(1,1)}^{0} + \overbrace{M(2,3)}^{B\times C} + \overbrace{d[0] \times d[1] \times d[3]}^{13\times 5\times 34}}^{k=1},$$

$$\overbrace{\overbrace{M(1,2)}^{5785} + \overbrace{M(3,3)}^{0} + \overbrace{d[0] \times d[2] \times d[3]}^{13\times 89\times 3}}^{k=2})$$
$$= min(1530, 9256) = 1530 \text{ con } k = 1$$

	A	B	C	D
A	0	5785	1530 k=1	
B		0	1335	1895 k=3
C			0	9078
D				0

2. $B \times C \times D \Longrightarrow$

$$M(2,4) = min(\overbrace{\overbrace{M(2,2)}^{0} + \overbrace{M(3,4)}^{C\times D} + \overbrace{d[1] \times d[2] \times d[4]}^{5\times 89\times 34}}^{k=2},$$

$$\overbrace{\overbrace{M(2,3)}^{B\times C} + M(4,4) + \overbrace{d[1] \times \quad d[3] \times d[4]}^{5\times 3\times 34}}^{k=3})$$
$$= min(24208, 1895) = 1895 \text{ con } k = 3$$

Diagonal 3 (s=3)

1. $A \times B \times C \times D \Longrightarrow$

$$M(1,4) = min(\overbrace{\overbrace{0}^{M(1,1)} + \overbrace{1895}^{M(2,4)} + \overbrace{13\times5\times34}^{d[0]\times d[1]\times d[4]}}^{k=1},$$
$$\overbrace{\overbrace{5785}^{M(1,2)} + \overbrace{9078}^{M(3,4)} + \overbrace{13\times89\times34}^{d[0]\times d[2]\times d[4]}}^{k=2},$$
$$\overbrace{\overbrace{1530}^{M(1,3)} + \overbrace{0}^{M(4,4)} + \overbrace{13\times3\times34}^{d[0]\times d[3]\times d[4]}}^{k=3})$$
$$= min(4055, 54201, 2856) = 2856 \text{ con } k = 3$$

	A	B	C	D
A	0	5785	1530 k=1	2856 k=3
B		0	1335	1895 k=3
C			0	9078
D				0

Con la tabla rellena ya sabemos que tenemos que hacer $M(1,4) = 2856$ multiplicaciones escalares. Para saber como parentizar miramos en $M(1,4)$ el valor de k que es 3 por lo tanto la primera partición es:

$$(A \times B \times C) \times D$$

A continuación vamos a $M(1,3)$ que tiene un valor de $k = 1$, por lo tanto:

$$(A \times (B \times C)) \times D$$

sería la parentización óptima.

□

El código C++ de este algoritmo podría ser el siguiente:

```
1  struct info{
2    int mesc; //numero de multiplicaciones
3    int k;
4  };
5
6  /**
7  @brief Obtiene la mejor parentizacion para multiplicar una serie de matrices
8  @param d: array con las dimensiones de las matrices
9  @param M: matriz con las soluciones para los diferentes
10           subproblemas
11 **/
12
13 void Parentesis_Mult(const vector<unsigned int> &d, Matriz<info> &M){
14  int n= M.numfilas();//numero de matrices a multiplicar
15  //Inicializacion
16  for (int i=0;i<n;i++)
```

```
17     M(i,i).mesc=0;
18
19   for (int diagonal=1;diagonal<n; diagonal++)
20    for (int i=0;i<n-diagonal;i++){
21     int k;
22     M(i,i+diagonal).mesc= Calcula_minimo(M,d,i,i+diagonal,k);
23     M(i,i+diagonal).k=k;
24    }
25
26 }
```

Como se puede observar se va rellenando la matriz por diagonales. En este caso se indexa desde 0. La función Calcula_minimo obtiene el menor numero de multiplicaciones para multiplicar una subsecuencia de matrices. La implementación de esta función sería:

```
unsigned int Calcula_minimo(const Matriz<info> &M,
                            const vector<unsigned int> &d,
                            unsigned int i, unsigned int j,
                            unsigned int & bestk){

 unsigned int min = numeric_limits<unsigned int>::max();
 for (int k=i;k<j;k++){
    int aux = M(i,k)+M(k+1,j)+d[i-1]*d[k]*d[j];
    if (aux<min){
      min = aux;
      bestk=k;
    }
 }
 return min;
}
```

Eficiencia.- La eficiencia de la función *Parentesis_Mult* tiene dos bucles anidados, y en cada iteración se llama a Calcula_minimo. Así la eficiencia la podemos poner como:

$$\sum_{diagonal=1}^{n-1}\sum_{i=0}^{n-diagonal-1}\sum_{k=i}^{i+diagonal-1}1$$

$$=\sum_{diagonal=1}^{n-1}\sum_{i=0}^{n-diagonal-1}diagonal$$

$$=\sum_{diagonal=1}^{n-1}(n-diagonal)\cdot diagonal$$

$$=n\cdot\sum_{diagonal=1}^{n-1}diagonal-\sum_{diagonal=1}^{n-1}diagonal^2$$

$$=\frac{n^3-n}{6}$$

4.10 Problema: Mochila(0 | 1) con múltiples objetos

Volvamos al problema de la Mochila, en el que se coge el objeto o no se coge (0|1). Pero en este caso en vez de existir n objetos distintos de lo que disponemos es de n tipos de objetos distintos. De cada objeto tenemos un numero de ejemplares ilimitado.
Podemos reformular este problema como una modificación de la Mochila (0|1) en el que se sustituye el requerimiento de que cada objeto se coge o no se coge ($x_i = 0$ o $x_i = 1$) por el requerimiento que $x_i \geq 0$. Igualmente deseamos maximizar la suma de los beneficios de los elementos introducidos, sujeto a la restricción de que los objetos no superen la capacidad de la mochila.
Solución.- Sea $V(i,m)$ el beneficio máximo obtenido para una mochila de capacidad m con i tipos de objetos. En cada paso decidiremos introducir un objeto o no del tipo i. De esta forma para calcular $V(i,m)$ tenemos que decidir entre:

- No coger un objeto de tipo $i \rightarrow$ En este caso nos queda la misma capacidad de la mochila y el mismo beneficio obtenido cuando se considera $i-1$ tipos de objetos,es decir el beneficio es $V(i-1,m)$.
- Coger una unidad más del objeto i Al beneficio que obtuvimos antes de añadir esta nueva unidad, $V(i,m-p_i)$, le sumamos el beneficio del objeto de tipo i, b_i.

Con estos puntos podemos ahora formular la **ecuación recurrente**:

$$V(i,m) = \begin{cases} 0 & \text{si } m = 0 \\ b_i \times \frac{m}{p_i} & \text{si } i = 1 \\ V(i-1,m) & \text{si } m < p_i \\ max(V(i-1,m), V(i,m-p_i) + b_i \text{ si } m \geq p_i & \end{cases}$$

Ejemplo 4.10.1
Supongamos que tenemos una mochila con capacidad M=6 y tenemos $n = 4$ tipos de objetos. Los vectores de pesos y beneficios de los tipos de objetos.

$$p = (2,3,1,5) \quad b = (3,1,4,30)$$

La matriz $V(i,m)$ que se obtiene siguiendo la ecuación de recurrencia anterior es:

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1 $p_1 = 2$ $b_1 = 3$	0	0	3	3	6	6	9
2 $p_2 = 3$ $b_2 = 1$	0	0	3	3	6	6	9
3 $p_3 = 1$ $b_3 = 4$	0	4	8	12	16	20	24
4 $p_4 = 5$ $b_4 = 30$	0	4	8	12	16	30	34

El camino a seguir para obtener que objeto se introduce en la mochila se puede ver en la tabla de la derecha. Como se puede comprobar se coge una unidad del objeto 3 y otra unidad del objeto 4, con un beneficio total de 34.

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1 $p_1 = 2$ $b_1 = 3$	0	0	3	3	6	6	9
2 $p_2 = 3$ $b_2 = 1$	0	0	3	3	6	6	9
3 $p_3 = 1$ $b_3 = 4$	0	4	8	12	16	20	24
4 $p_4 = 5$ $b_4 = 30$	0	4	8	12	16	30	34

□

El código C++ de este algoritmo podría ser el siguiente

```
1  struct objeto{
2    int peso;
3    int beneficio;
4  };
5  int Mochila_TiposObj(int M, Matrz & V, vector<objeto>&obj){
6   int n= obj.size();
7   //Inicializacion
8   for (int i=1;i<=n;i++)
9    V(i,0)=0;
10  for (int i=0; i<=M;i++)
11    V(0,i)=0;
12
13  //EL primer tipo de objeto
14  for (int cap=1;cap<=M;cap++)
15    V(1,cap)=obj[1].beneficio* (cap/obj[1].peso);//division entera
16
17  for (int i=2;i<=n;i++)
18   for (int cap=1;cap<=M;cap++){
19      if (cap<obj[i].peso)
20        V(i,cap)=V(i-1,cap);
21      else
22        V(i,cap)=Maximo(V(i-1,cap),V(i,cap-obj[i].peso)+obj[i].beneficio);
23    }
24   return V(n,M);
25 }
```

Eficiencia.-. La eficiencia para rellenar la tabla es $\Theta(n \times M)$.

4.11 Problema: Dos Mochilas (0|1)

Suponed que disponemos de dos mochilas con capacidades M_1 y M_2 respectivamente. En ellas queremos poner de entre los n objetos aquellos que nos den el mayor beneficio acumulado sin superar la capacidad de las dos mochilas. Para los n objetos sabemos que los pesos son $p = (p_1, p_2, \cdots, p_n)$ y el beneficio $b = (b_1, b_2, \cdots, b_n)$. Si $S = (x_1, x_2, \cdots, x_n)$ con $x_i = 0, 1$ es una solución entonces debe obtener el mayor beneficio acumulado posible, tal que: $B_T = \sum_{i=1}^{n} x_i \cdot b_i$ sujeto a $\sum_{i=1}^{n} x_i \cdot p_i \leq M_1 + M_2$

Principio de Optimalidad de Bellman.- Si $S = (x_1, x_2, \cdots, x_n)$ es una solución entonces $B_T = \sum_{i=1}^{n} x_i \cdot b_i$ es el beneficio máximo y además $\sum_{i=1}^{n} x_i \cdot p_i \leq M_1 + M_2$. Si se cumple el P.O.B entonces la subsecuencia $(x_1, x_2, \cdots, x_{n-1})$ es óptima para una capacidad $M_1 + M_2 - x_n \cdot p_n$.

Ecuación de recurrencia.- La ecuación de recurrencia la vamos a formular como $Mochila(m_1, m_2, k)$, que devuelve para dos mochilas de capacidad m_1 y m_2 con k objetos el beneficio máximo acumulado. En la formulación se plantean tres casos:

1. No coge el objeto $k \rightarrow$ En este caso las dos mochilas quedan con la misma capacidad y tenemos un objeto menos que analizar, $Mochila(m_1, m_2, k-1)$.
2. Se coge el objeto:
   a) El objeto k se pone en la mochila 1
      - La primera mochila queda con capacidad $m_1 - p_k$,
      - La segunda mochila queda igual
      - Tenemos un objeto menos que analizar
      - Aumentamos el beneficio a $Mochila(m_1 - p_k, m_2, k-1) + b_k$

   b) El objeto k se pone en la mochila 2
      - La segunda mochila queda con capacidad $m_2 - p_k$,
      - La primera mochila queda igual
      - Tenemos un objeto menos que analizar
      - Aumentamos el beneficio a $Mochila(m_1, m_2 - p_k, k-1) + b_k$

Con estas aclaraciones podemos ahora escribir matemáticamente la ecuación de recurrencia:

$$Mochila(m_1, m_2, k) = \begin{cases} -\infty & m_1 < 0 \text{ o } m_2 < 0 \\ 0 & m_1 = 0 \text{ y} m_2 = 0 \\ 0 & k \leq 0 \\ max(\underbrace{Mochila(m_1, m_2, k-1)}_{\text{No se coge}}, \underbrace{Mochila(m_1 - p_k, m_2, k-1) + b_k}_{\substack{\text{Se coge} \\ \text{y se pone} \\ \text{en la mochila 1}}}, \\ \underbrace{Mochila(m_1, m_2 - p_k, k-1) + b_k)}_{\substack{\text{Se coge} \\ \text{y se pone} \\ \text{en la mochila 2}}} \end{cases}$$

Ejemplo 4.11.1
Supongamos que el número de objetos $n = 4$ y las mochilas tienen capacidades $M_1 = 3$ y $M_2 = 2$. Los vectores de pesos y beneficios son: $p = (1,2,3,3)$, $b = (5,4,2,6)$.
Para construir la tabla V y sea legible, las columnas las indexamos por pares que representan la capacidad en la primera mochila y capacidad en la segunda mochila. Por lo tanto el número de columnas será en general $(M_1 + 1) \times (M_2 + 1)$.

	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	(3,0)	(3,1)	(3,2)
0	0	0	0	0	0	0	0	0	0	0	0	0
1 $p_1 = 1$ $b_1 = 5$	0	5	5	5	5	5	5	5	5	5	5	5
2 $p_2 = 2$ $b_2 = 4$	0	5	5	5	5	9	5	9	9	9	9	9
3 $p_3 = 3$ $b_3 = 2$	0	5	5	5	5	9	5	9	9	9	9	9
4 $p_4 = 3$ $b_4 = 6$	0	5	5	5	5	9	5	9	9	9	11	11

El beneficio total es 11. Partiendo de la casilla $V(5,(3,2))$ podemos obtener el camino para deducir que objetos debemos escoger para lograr este beneficio.

	(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)	(3,0)	(3,1)	(3,2)
0	0	0	0	0	0	0	0	0	0	0	0	0
1 $p_1 = 1$ $b_1 = 5$	0	5	5	5	5	5	5	5	5	5	5	5
2 $p_2 = 2$ $b_2 = 4$	0	5	5	5	5	9	5	9	9	9	9	9
3 $p_3 = 3$ $b_3 = 2$	0	5	5	5	5	9	5	9	9	9	9	9
4 $p_4 = 3$ $b_4 = 6$	0	5	5	5	5	9	5	9	9	9	11	11

Por lo tanto los objetos que se introducen en la mochila son el objeto 1 y el 4.

□

Ejercicio 4.3
Dar una implementación C++ del problema de S mochilas con n objetos. □

4.12 Problema: Planificación de Tareas

Supongamos que tenemos un procesador y un conjunto de tareas $\{a_1, a_2, \cdots, a_n\}$ que deben ser ejecutadas. Cada tarea a_i tiene un tiempo de procesamiento t_i y un beneficio b_i si se ejecuta, además la tarea no se puede ejecutar más allá del plazo d_i. El procesador solamente puede procesar en cada instante un trabajo, y cada tarea debe ejecutarse de forma interrumpida durante los t_i unidades de tiempo. Si la tarea a_i se ejecuta antes o igual al instante d_i se recibe el beneficio b_i. Pero si se ejecuta fuera de plazo se recibe un beneficio 0.

Paso 1:Ecuación de recurrencia

Sea $P(k,D)$ sea el beneficio máximo obtenido para planificar k tareas en un plazo máximo D. En este caso también podemos plantear la solución como un problema de decisiones en el que cada paso se decide:

- No se ejecuta la tarea en el instante D $k \Longrightarrow$ En este caso el beneficio total es el maximo entre el que se tenia sin contemplar la tarea k en el mismo plazo, es decir $P(k-1,D)$,y el que se tenia en el instante anterior contemplando la tarea k
- Se ejecuta la tarea $k \Longrightarrow$ Se suma el beneficio de la tarea k, b_k al beneficio de ejecutar $k-1$ tareas en plazo $D-t_k$, $P(k-1,D-t_k)$. Ya que los instante de tiempo $D-t_k+1, D-t_k+2, \cdots, D-t_k+t_k$ se usarán para ejecutar la tarea k. La tarea k se podrá ejecutar si $D-t_k >= 0$ y $D >= d_k$.

Con todo esto podemos plantear la ecuación de recurrencia:

$$P(k,D) = \begin{cases} -\infty & D < 0 \\ 0 & D = 0 || k = 0 \\ max(P(k-1,D), P(k,D-1)) & D > d_k \\ max(P(k-1,D), P(k-1,D-t_k)+b_k) & \text{en otro caso} \end{cases}$$

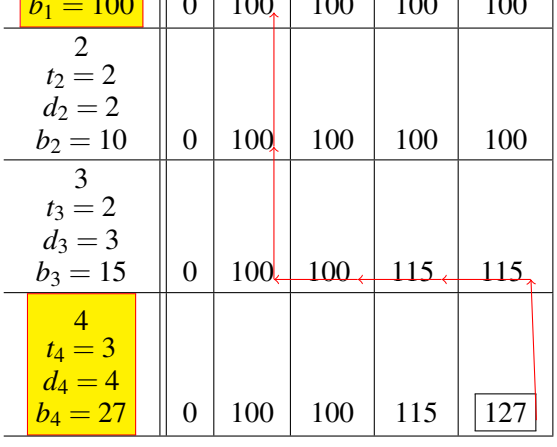
La tabla P tendrá $n+1$ filas siendo n el número de tareas y $D+1$ columnas. D se define como el máximo plazo de entre los plazos de las tareas a analizar.

Ejemplo 4.12.1
Supongamos que tenemos cuatro tareas $n = 4$ y con beneficios $b = (100, 10, 15, 27)$, plazos límite $d = (2,2,3,4)$ y tiempo de ejecución de cada tarea $t = (1,2,2,3)$. La tabla resultante sería:

	0	1	2	3	4
0	0	0	0	0	0
1 $t_1 = 1$ $d_1 = 2$ $b_1 = 100$	0	100	100	100	100
2 $t_2 = 2$ $d_2 = 2$ $b_2 = 10$	0	100	100	100	100
3 $t_3 = 2$ $d_3 = 3$ $b_3 = 15$	0	100	100	115	115
4 $t_4 = 3$ $d_4 = 4$ $b_4 = 27$	0	100	100	115	127

Como se puede observar el beneficio maximo es 127. Para ver si se coge el objeto hay que mirar que el valor del beneficio es diferente de la casilla de arriba y de la izquierda. Si es igual a la de arriba ir hacia arriba, o ir hacia la izquierda en otro caso. En la siguiente tabla se puede ver el camino para recuperar la solución:

	0	1	2	3	4
0	0	0	0	0	0
1 $t_1 = 1$ $d_1 = 2$ $b_1 = 100$	0	100	100	100	100
2 $t_2 = 2$ $d_2 = 2$ $b_2 = 10$	0	100	100	100	100
3 $t_3 = 2$ $d_3 = 3$ $b_3 = 15$	0	100	100	115	115
4 $t_4 = 3$ $d_4 = 4$ $b_4 = 27$	0	100	100	115	127



□

El código C++ podría ser el siguiente

```
1 struct tarea{
2   int beneficio;
3   int duracion;
```

```
4     int plazo;
5   };
6
7   int Planificacion_PD(const vector<tarea> & T, Matriz<int> & P){
8    int n= T.size(); //numero de tareas:
9    int D= P.numcolumnas()-1; // plazo maximo
10
11   //Inicializacion
12    for (int i=0;i<=D;i++)
13     P(0,i)=0;
14    for (int i=0;i<=n;i++)
15     P(i,0)=0;
16
17    //para cada una de las tareas
18    for (int k=1;k<=n;k++){
19     //para cada uno de los instante
20     for (int t=1;t<=D; t++){
21        if (t>T[k].plazo){
22          P(k,t)= max(P(k-1,t),P(k,t-1));
23         }
24         else {
25          int aux = t-T[k].duracion;
26          if (aux<0)
27            P(k,t)=P(k-1,t);
28          else
29            P(k,t)=max(P(k-1,t),P(k-1,aux)+T[k].beneficio);
30         }
31      }
32    }
33  }
```

La solución S será un vector con posiciones desde 0 hasta el máximo plazo D, tal que $S[i]$ especifica la tarea que se ejecuta en el instante i. Si no se ejecuta en ese instante ninguna tares $S[i] = -1$. El código para recuperar la solución sería el siguiente:

```
1   void GetSolucion_Planificacion(const Matriz<int> &P,
2              const vector<tarea> & T, vector<int> &S){
3    int D= P.numcolumnas()-1;
4    int n= P.numfilas()-1;
5    //Inicializacion
6    for (int i=0;i<=D;i++)
7     S[i]=-1; // en ese instante no se ejecuta nada
8
9    int k=n;
10   int t = D;
```

```
11   while (k>0 && t>0){
12        //hemos obtenido el maximo arriba
13       if (P(k-1,t)==P(k,t))
14          k = k-1;
15       else
16        if (P(k,t-1)==P(k,t))
17         //hemos obtenido el maximo a la izquierda
18         t = t-1;
19        else {
20           // se ejecuta la tarea k
21           for (int j=t;j>t-T[k].duracion; j--)
22              S[j]=k;
23           t = t-T[k].duracion;
24           k = k-1;
25        }
26   }
27  }
```

Eficiencia.- La función que rellena la tabla tiene eficiencia $\Theta(n \times D)$. Y la función para obtener la solución tiene eficiencia $\Theta(n)$.

4.13 Problema: El viajante de comercio

Vimos este problema en el capítulo de voraces: Un viajante de comercio tiene que recorrer un conjunto de ciudades pasando solamente una vez por cada ciudad y volviendo al punto de partida, con el objetivo de reducir el numero de kilómetros que hace. En términos generales, dado un grafo el objeto es encontrar el camino Hamiltoniano de menor coste, es decir, pasar por todo los vértices una sola vez y volver al vértice de partida, de forma que el camino obtenido sea el de menos longitud.

Planteamiento.- El problema se plantea como una toma de decisiones, en que en cada instante se decide si añadir un nuevo vértice o no. Para ello se obtiene los recorridos mínimos que incluya solamente 1 vértice, 2 vértices, 3 vértices, etc. Por lo tanto dado un grafo $G = (V,A)$ se debe obtener todas las particiones de V (conjuntos de 1 elemento, conjuntos de 2 elementos, etc).

El recorrido del viajante de comercio parte del vértice v_1. Para recorridos de tamaño 1 la solución óptima será $v_1 - v_k - v_1$ para algún vértice $v_k \in V - v_1$. De forma que si el recorrido $v_1 - v_k - v_1$ es óptimo sus partes deben ser óptimas según el P.O.B. Por ejemplo si el subcamino $v_k - v_1$ no es óptimo existe otro vértice entre medias y por lo tanto el recorrido original también debería incluir este otro vértice y por lo tanto $v_1 - v_k - v_1$ no sería óptimo.

Ecuación Recurrente.-. Sea $D(v_i, S)$ la longitud del camino mínimo que partiendo de v_i pasa por todos los nodos de S y vuelve a v_i. Con esta definición la ecuación la podemos