3

PROGRAMACIÓN DINÁMICA

PROBLEMAS DE OPTIMIZACIÓN

Una **aplicación típica de Programación Dinámica** es la resolución de problemas de optimización.

Estos problemas pueden tener muchas soluciones factibles (que cumplen unas ciertas restricciones), pero se trata de encontrar **LA SOLUCIÓN ÓPTIMA**.

Cada solución tiene un valor asociado, y lo que se desea encontrar es la solución con el valor óptimo (máximo o mínimo)

En este caso tendremos:

- **FUNCIÓN OBJETIVO**
- **RESTRICCIONES**

PROBLEMAS DE OPTIMIZACIÓN

Otra característica de este tipo de problemas es que la solución se puede expresar como una **SECUENCIA DE DECISIONES**

$$<X_1, X_2, \ldots, X_n>$$

PROBLEMAS DE OPTIMIZACIÓN

Además, para garantizar que la aplicación de Programación Dinámica a un problema de optimización produce una solución óptima, dicho problema debe satisfacer ciertas condiciones que se expresan bajo el denominado **"PRINCIPIO DE OPTIMALIDAD DE BELLMAN"**:

"Dada una secuencia óptima de decisiones,
toda subsecuencia de ella es, a su vez, óptima"

PRINCIPIO DE OPTIMALIDAD

Ejemplo.-

Supongamos que el camino más corto entre Gijón y Santander pasa por Llanes, entonces la parte del camino que va desde Gijón a Llanes también debe ser el camino más corto posible, al igual que la parte del camino que va de Llanes a Santander.

PRINCIPIO DE OPTIMALIDAD

Demostración.-

Sea la solución óptima al problema de la ruta más corta entre Gijón y Santander:

$$\underbrace{\text{Gijón} \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow \text{Llanes}}_{Y} \rightarrow \underbrace{P_i \rightarrow P_{i+1} \rightarrow \ldots \rightarrow \text{Santander}}_{Z}$$

Cuya distancia total es **X**, esto es,

$$X = Y + Z$$

PRINCIPIO DE OPTIMALIDAD

$\underbrace{\text{Gijón} \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow \text{Llanes}}_{Y} \rightarrow \underbrace{P_i \rightarrow P_{i+1} \rightarrow \ldots \rightarrow \text{Santander}}_{Z}$

Supongamos ahora que **Y NO** es la distancia del camino más corto que va de Gijón a Llanes, sino que hay otro camino de Gijón a Llanes con una distancia menor

$\underbrace{\text{Gijón} \rightarrow P'_1 \rightarrow P'_2 \rightarrow \ldots \rightarrow \text{Llanes}}_{Y'}$

En conclusión, **Y' < Y**

PRINCIPIO DE OPTIMALIDAD

Lo que implica que

$$\text{Gijón} \underbrace{\rightarrow P'_1 \rightarrow P'_2 \rightarrow \ldots \rightarrow \text{Llanes}}_{Y'} \underbrace{\rightarrow P_i \rightarrow P_{i+1} \rightarrow \ldots \rightarrow \text{Santander}}_{Z}$$

X' = Y' + Z

Mejoraría a

$$\text{Gijón} \underbrace{\rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow \text{Llanes}}_{Y} \underbrace{\rightarrow P_i \rightarrow P_{i+1} \rightarrow \ldots \rightarrow \text{Santander}}_{Z}$$

X = Y + Z

PRINCIPIO DE OPTIMALIDAD

Ya que

$$X' = Y' + Z$$

es menor que

$$X = Y + Z$$

lo cual contradice la hipótesis de partida, pues ésta era la solución óptima de dicho problema. Se cumple pues, el principio de optimalidad.

Aún cuando este principio pueda parecer evidente, no es aplicable a todos los problemas.

RESOLUCIÓN PROBLEMAS DE OPTIMIZACIÓN

Pasos a seguir.-

- Solución como secuencia de decisiones
- Función objetivo y restricciones
- Demostración principio de optimalidad
- Definición recursiva
- Árbol de llamadas
- Repetición Subproblemas
- Orden de los subproblemas
- Estructura de Almacenamiento
- Proceso de rellenado
- Búsqueda de la solución óptima
- Algoritmo

EJEMPLO 1: MOCHILA 0/1

Tenemos n objetos y cada objeto i, con $0 \leq i \leq n$, tiene un peso asociado (P_i) y un beneficio (B_i), donde Bi y Pi son mayores que 0.

Además, tenemos una mochila con una capacidad C que cumple

$$\sum_{i=1}^{n} P_i > C$$

¿Qué objetos debemos introducir en la mochila de tal forma que maximicemos el beneficio que transporta y no superemos la capacidad C de la mochila?

MOCHILA 0/1: SOLUCIÓN COMO SECUENCIA DE DECISIONES

$$<X_1, X_2, \ldots X_n>$$

donde X_i es la decisión con respecto al objeto i:

X_i=1 indica que el objeto i se introduce en la mochila y

X_i=0 indica que el objeto i no se introduce en la mochila.

MOCHILA 0/1: FUNCIÓN OBJETIVO Y RESTRICCIONES

$$\boldsymbol{maximizar} \sum_{i=1}^{n} X_i\, B_i$$

sujeto a

$$\sum_{i=1}^{n} X_i\, P_i \leq C$$

MOCHILA 0/1: PRINCIPIO DE OPTIMALIDAD

PRINCIPIO DE OPTIMALIDAD

"En una secuencia óptima de decisiones, toda subsecuencia ha de ser óptima"

Demostración.-

Sea $<X_1, X_2, \ldots, X_n>$ la solución óptima para el problema **(1, n, C)**, esto es, para el problema con n objetos (del 1 al n) y una mochila de capacidad C.

Su valor asociado es

$$\sum_{i=1}^{n} X_i \, B_i$$

sujeto a

$$\sum_{i=1}^{n} X_i \, P_i \leq C$$

MOCHILA 0/1: PRINCIPIO DE OPTIMALIDAD

Vamos a prescindir de la última decisión, es decir, X_n.

La subsecuencia que queda, esto es, $< X_1, X_2, \ldots, X_{n-1} >$, es la solución óptima para el problema asociado, es decir, para el problema **(1, n-1, C-$X_n P_n$)**.

Su valor asociado es

$$\sum_{i=1}^{n-1} X_i\, B_i$$

sujeto a

$$\sum_{i=1}^{n-1} X_i\, P_i \leq C - X_n P_n$$

MOCHILA 0/1: PRINCIPIO DE OPTIMALIDAD

Supongamos que $< X_1, X_2, \ldots, X_{n-1} >$ **NO** es la solución óptima del problema $(1, n-1, C-X_nP_n)$ sino que hay otra tupla, siendo ésta, $< Y_1, Y_2, \ldots, Y_{n-1} >$, que mejora el resultado para el problema $(1, n-1, C-X_nP_n)$.

Su valor asociado es

$$\sum_{i=1}^{n-1} Y_i B_i$$

sujeto a

$$\sum_{i=1}^{n-1} Y_i P_i \leq C - X_n P_n$$

MOCHILA 0/1: PRINCIPIO DE OPTIMALIDAD

Dado que $< Y_1, Y_2, \ldots, Y_{n-1} >$ mejora el resultado de $< X_1, X_2, \ldots, X_{n-1} >$, eso implica lo siguiente

$$\sum_{i=1}^{n-1} X_i\, B_i < \sum_{i=1}^{n-1} Y_i B_i$$

MOCHILA 0/1: PRINCIPIO DE OPTIMALIDAD

Pero entonces sumando X_n B_n a ambos lados de la desigualdad, se cumpliría que:

$$\sum_{i=1}^{n-1} X_i\, B_i + X_n B_n < \sum_{i=1}^{n-1} Y_i B_i + X_n B_n$$

lo cual significa que la tupla $< Y_1, Y_2, \ldots, Y_{n-1}, X_n >$ mejoraría el resultado de la tupla $< X_1, X_2, \ldots, X_{n-1}, X_n >$ para el problema (1, n, C), lo cual contradice la hipótesis de partida, pues ésta era la solución óptima de dicho problema.

Se cumple, pues, el principio de optimalidad.

MOCHILA 0/1: DEFINICIÓN RECURSIVA

BACKWARD (HACIA ATRÁS).- ¿ $< X_1, X_2, \ldots, X_{n-1}, X_n >$?

¿ (1,n,C) ?

MOCHILA 0/1: DEFINICIÓN RECURSIVA

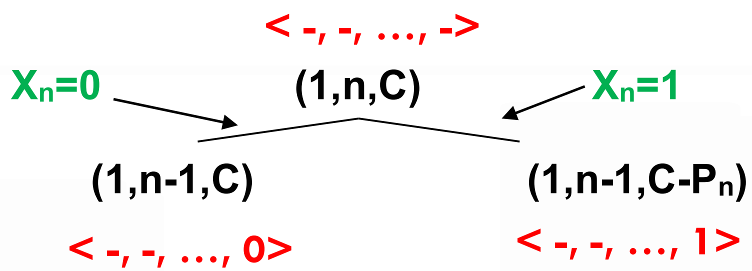
BACKWARD (HACIA ATRÁS).- ¿ < X1, X2, ..., Xn-1, Xn > ?

X_n=0 (1,n,C) X_n=1

(1,n-1,C) (1,n-1,C-P_n)

MOCHILA 0/1: DEFINICIÓN RECURSIVA

BACKWARD (HACIA ATRÁS).- ¿ < X1, X2, …, Xn-1, Xn > ?



MOCHILA 0/1: DEFINICIÓN RECURSIVA

BACKWARD (HACIA ATRÁS)

$X_n=0$ (1,n,C) $X_n=1$

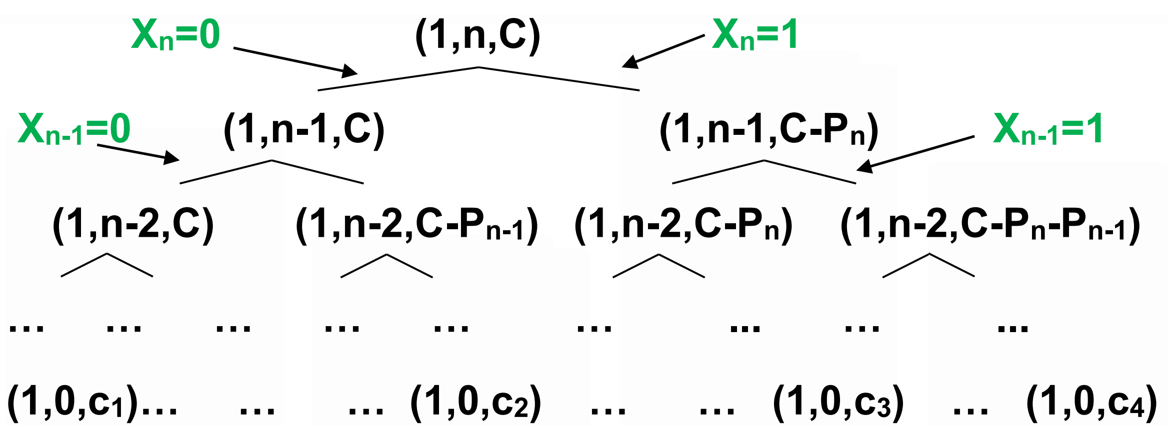
$X_{n-1}=0$ (1,n-1,C) $X_{n-1}=1$ $X_{n-1}=0$ (1,n-1,C-P_n) $X_{n-1}=1$

(1,n-2,C) (1,n-2,C-P_{n-1}) (1,n-2,C-P_n) (1,n-2,C-P_n-P_{n-1})

… … … … … … … …

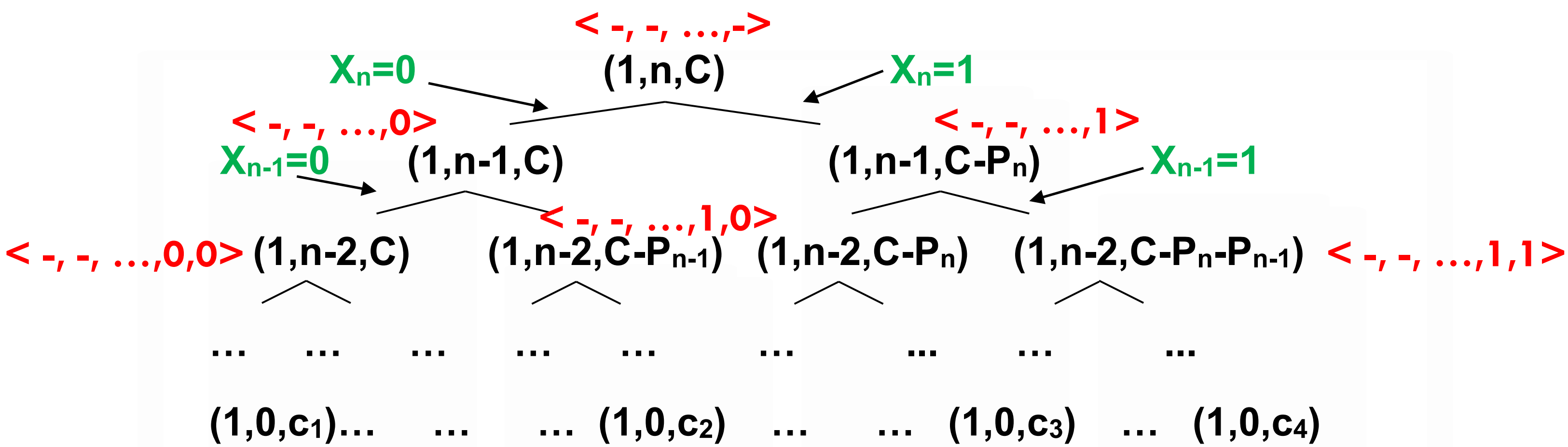
MOCHILA 0/1: DEFINICIÓN RECURSIVA

BACKWARD (HACIA ATRÁS)



MOCHILA 0/1: DEFINICIÓN RECURSIVA

BACKWARD (HACIA ATRÁS)



MOCHILA 0/1: DEFINICIÓN RECURSIVA

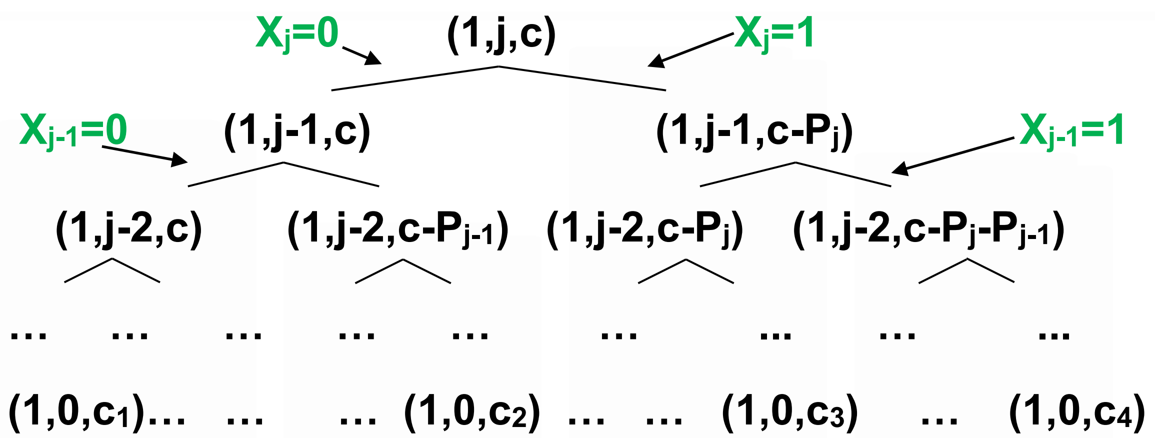
Representaremos el problema a través de la terna **(1,n,C)**.-

manejaremos subproblemas de la forma **(1, j, c)**, consistente en resolver óptimamente el subproblema formado por los objetos del **1** al **j** y con una mochila de capacidad **c** (**BACKWARD o HACIA ATRÁS**)

$$< X_1, \ldots, X_{j-1}, X_j >$$

MOCHILA 0/1: DEFINICIÓN RECURSIVA

BACKWARD (HACIA ATRÁS)



MOCHILA 0/1: DEFINICIÓN RECURSIVA

BACKWARD (HACIA ATRÁS).- Dado que el parámetro 1 va a permanecer invariable a lo largo de todo el proceso recursivo, podemos prescindir del mismo en la representación del problema con el fin de simplificar la notación, esto es,

Mochila(1, j, c) pasa a ser Mochila(j, c)

$$\boldsymbol{Mochila(j,c)} = \begin{cases} \mathbf{0} & \boldsymbol{si\ j = 0} \\ \boldsymbol{Mochila(j-1,c)} & \boldsymbol{si\ j > 0\ y\ c - P_j < 0} \\ \max\{\boldsymbol{Mochila(j-1,c), Mochila(j-1, c-P_j) + B_j}\} & \boldsymbol{si\ j > 0\ y\ c - P_j \geq 0} \end{cases}$$

Llamada inicial: Mochila(n,C)

MOCHILA 0/1: DEFINICIÓN RECURSIVA (otra forma de expresarla)

BACKWARD (HACIA ATRÁS).- Dado que el parámetro 1 va a permanecer invariable a lo largo de todo el proceso recursivo, podemos prescindir del mismo en la representación del problema con el fin de simplificar la notación, esto es,

Mochila(1, j, c) pasa a ser Mochila(j, c)

$$\boldsymbol{Mochila(j,c)} = \begin{cases} \mathbf{0} & \boldsymbol{si\ j = 0} \\ \boldsymbol{Mochila(j-1,c)} & \boldsymbol{si\ j > 0\ y\ c - P_j < 0} \\ \boldsymbol{max}_{x_j \in \{0,1\}} \{\boldsymbol{Mochila(j-1, c - x_j P_j) + x_j B_j}\} & \boldsymbol{si\ j > 0\ y\ c - P_j \geq 0} \end{cases}$$

Llamada inicial: Mochila(n,C)

MOCHILA 0/1: REPETICIÓN DE SUBPROBLEMAS

Desde la perspectiva recursiva el problema está resuelto.

Pero debemos tener en cuenta que la complejidad resultante es de orden exponencial, pues la ecuación recurrente tiene la forma

$$T(n)=2T(n-1)+c_1,$$

lo cual da lugar a un orden $O(2^n)$.

Esta complejidad es debida a la reiteración en la solución de subproblemas.

MOCHILA 0/1: ESTRUCTURAS DE ALMACENAMIENTO

Dos matrices-

- con n+1 filas (para representar el objeto en curso de decisión) y
- con C+1 columnas (para representar todos los posibles estados c de la mochila)

Bmax[0..n][0..C] → en la que se almacena el Beneficio máximo

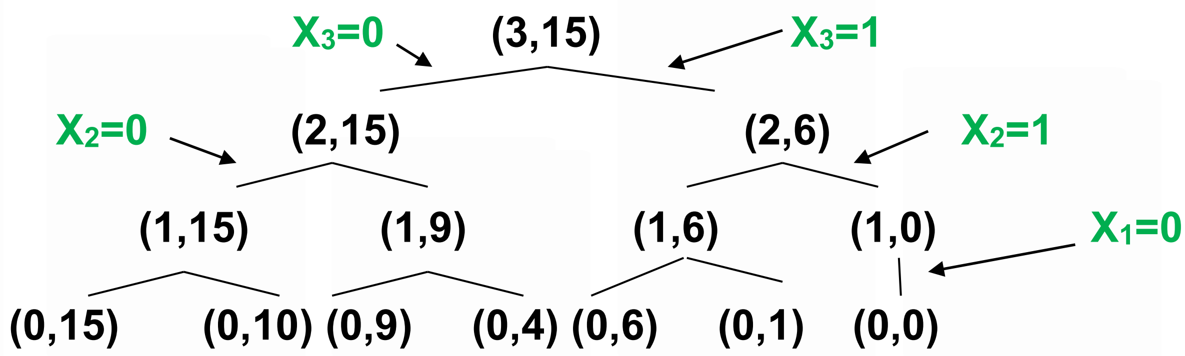
Dec[0..n][0..C] → en la que se guarda la decisión que proporciona el beneficio máximo

Mochila(j, c) → **Bmax[j][c]**

Mochila(j, c) → **Dec[j][c]**

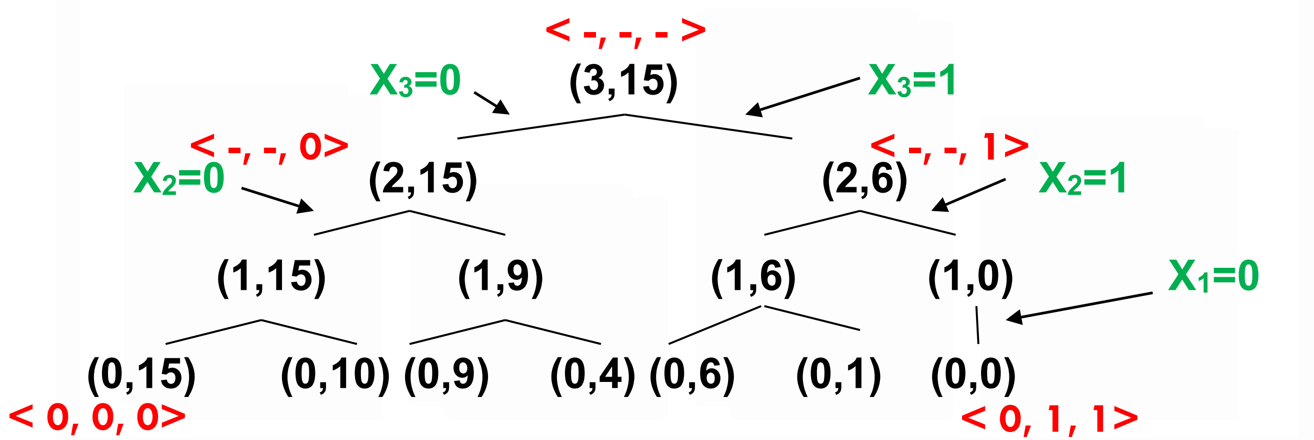
MOCHILA 0/1: FORMA DE RELLENAR-ORDEN SUBPROBLEMAS

Sean n=3, C=15 , B[1..3] = {24, 40, 38} y P[1..3] = {5, 6, 9}



MOCHILA 0/1: FORMA DE RELLENAR-ORDEN SUBPROBLEMAS

Sean n=3, C=15 , B[1..3] = {24, 40, 38} y P[1..3] = {5, 6, 9}



MOCHILA 0/1: Bmax y Dec

1° /* PROBLEMAS TRIVIALES */

		c															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
j	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1																
	2																
	3																

MOCHILA 0/1: Bmax

2º /* RESTO DE PROBLEMAS EN ORDEN ASCENDENTE */

j \ c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	24	24	24	24	24	24	24	24	24	24	24
2	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	64
3	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	78

Bmax [3][15] = max { Bmax [2][15], Bmax [2][6]+38 }=
= max {64, 40+38} = max {64, 78} = **78**

Dec[3][15]=**1**

MOCHILA 0/1: Dec

j	c: 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Dec[3][15]=1

MOCHILA 0/1: MOSTRAR SOLUCIÓN

Consiste de dos partes.-

- **1ª parte: Valor que optimiza la función objetivo**

Beneficio máximo → **Bmax[3][15]**

en general **Bmax[n][C]**

j \ c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	24	24	24	24	24	24	24	24	24	24	24
2	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	64
3	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	78

MOCHILA 0/1: MOSTRAR SOLUCIÓN

- **2ª parte: Secuencia óptima de decisiones.**

Se realiza un recorrido por la estructura que almacena las decisiones (Dec).

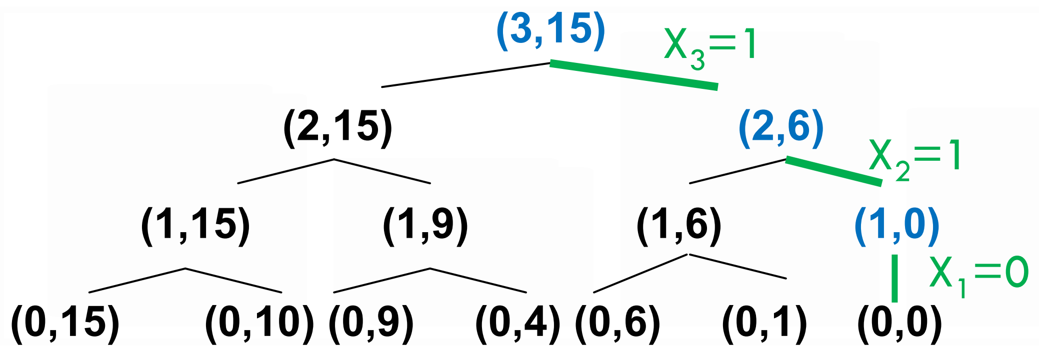
n=3, C=15 , B[1..3] = {24, 40, 38} y P[1..3] = {5, 6, 9}

c

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

$< X_1, X_2, X_3 > = < 0, 1, 1 >$

MOCHILA 0/1: SOLUCIÓN ÓPTIMA VISTA EN EL ÁRBOL



MOCHILA 0/1: ALGORITMO

Función Mochila(P[1..n], B[1..n]:vector de enteros; n, C:entero)

retorna (p:entero, f[1..n]:vector de enteros)

/* **Necesita:** el número de objetos, los pesos y valores de los n objetos y la capacidad de la mochila.

Produce: El valor de la solución óptima y un vector con la decisión tomada para cada objeto. */

```
var
    /* almacenará el beneficio máximo de cada subproblema */
    Bmax[0..n][0..C]: matriz de enteros;

    /* registrará la decisión óptima para cada subproblema */
    Dec[0..n][0..C] : matriz de enteros;
    j, c : entero;

    /* Secuencia de decisiones óptima */
    X[1..n]: vector de enteros;
fvar
```

MOCHILA 0/1: ALGORITMO

/* inicializar la estructura de almacenamiento con los resultados de los problemas triviales */

```
para c=0 hasta C hacer
    Bmax [0][c]=0;
    Dec [0][c]=0;
fpara
```

MOCHILA 0/1: ALGORITMO


```
/* rellenar el resto de estructura de almacenamiento con resultados de problemas no triviales, sentido ascendente */
para j=1 hasta n hacer
    para c=0 hasta C hacer
        si ( c < P[ j ] ) entonces
                    Bmax [ j ][ c ] = Bmax [ j-1 ][ c ];
                    Dec [ j ][ c ] = 0;
        si no
            si ( Bmax [ j-1 ][ c ] ≥ Bmax [ j-1 ][ c-P[ j ] ] + B[ j ] )  entonces
                                Bmax [ j ][ c ] = Bmax [ j-1 ][ c ];
                                Dec [ j ][ c ] = 0;
            si no
              Bmax [ j ][ c ] = Bmax[ j-1 ][ c - P[ j ] ] + B[ j ];
              Dec [ j ][ c ] = 1;
            fsi
        fsi
    fpara
fpara
```

MOCHILA 0/1: ALGORITMO


```
/* solución */

j = n;
c = C;
mientras ( j > 0 ) hacer
    X[ j ] = Dec [ j ][ c ];
    c = c - X[ j ] * P[ j ];
    j = j - 1;
fmientras
retorna ( Bmax[ n ][ C ], X );
ffunción
```

TAREA PROPUESTA

Tras haber resuelto el problema de la mochila manejando subproblemas de la forma **(1, j, c)**, consistente en resolver óptimamente el subproblema formado por los objetos del 1 al j y con una mochila de capacidad **c** (**BACKWARD o HACIA ATRÁS**).-

$$< X_1, \ldots, X_{j-1}, X_j >$$

La tarea consiste en resolver el problema de la mochila manejando los subproblemas del tipo **(j, n, c)**, consistente en resolver óptimamente el subproblema formado por los objetos del j al n y con una mochila de capacidad **c** (**FORWARD o HACIA ADELANTE**).-

$$< X_j, X_{j+1}, \ldots X_n >$$

EJEMPLO 2: DESCOMPONER UN NÚMERO EN SUMANDOS

Dados dos enteros N y M con $N \geq M > 0$, se pide descomponer el número N en M sumandos (enteros positivos) de forma que los M sumandos sumen N y su producto sea máximo.

SOLUCIÓN COMO SECUENCIA DE DECISIONES.-

$$< S_1, S_2 , \ldots, S_M >$$

donde $\mathbf{S_i}$ es el valor correspondiente al sumando i-ésimo. Dicho valor es entero positivo.

DESCOMPONER UN NÚMERO EN SUMANDOS

FUNCIÓN OBJETIVO Y RESTRICCIONES.-

La secuencia óptima de decisiones será aquella $< S_1, S_2, \ldots, S_M >$ tal que maximice la siguiente función

$$\prod_{i=1}^{M} S_i$$

sujeta a la siguiente restricción

$$\sum_{i=1}^{M} S_i = N$$

DESCOMPONER N EN SUMANDOS

PRINCIPIO DE OPTIMALIDAD

"En una secuencia óptima de decisiones, toda subsecuencia ha de ser óptima"

Demostración.-

Sea $<S_1, S_2, \ldots, S_M>$ la solución óptima para el problema **(1, M, N)**, esto es, para el problema de descomponer N en M sumandos (del 1 al M).

Su valor asociado es

$$\prod_{i=1}^{M} S_i$$

sujeto a

$$\sum_{i=1}^{M} S_i = N$$

DESCOMPONER N EN SUMANDOS: PRINCIPIO DE OPTIMALIDAD

Vamos a prescindir de la última decisión, es decir, S_M

La subsecuencia que queda, esto es, $<S_1, S_2, \ldots, S_{M-1}>$, es la solución óptima para el problema asociado, es decir, para el problema $(1, M-1, N-S_M)$.

Su valor asociado es

$$\prod_{i=1}^{M-1} S_i$$

sujeto a

$$\sum_{i=1}^{M-1} S_i = N - S_M$$

DESCOMPONER N EN SUMANDOS: PRINCIPIO DE OPTIMALIDAD

Supongamos que $< S_1, S_2, \ldots, S_{M-1} >$ **NO** es la solución óptima del problema $(1, M-1, N-S_M)$ sino que hay otra solución, siendo ésta, $< Y_1, Y_2, \ldots, Y_{M-1} >$ que mejora el resultado para el problema $(1, M-1, N-S_M)$

Su valor asociado es

$$\prod_{i=1}^{M-1} Y_i$$

sujeto a

$$\sum_{i=1}^{M-1} Y_i = N - S_M$$

DESCOMPONER N EN SUMANDOS: PRINCIPIO DE OPTIMALIDAD

Dado que $< Y_1, Y_2, \ldots, Y_{M-1} >$ mejora el resultado de $< S_1, S_2, \ldots, S_{M-1} >$, eso implica lo siguiente

$$\prod_{i=1}^{M-1} S_i < \prod_{i=1}^{M-1} Y_i$$

DESCOMPONER N EN SUMANDOS: PRINCIPIO DE OPTIMALIDAD

Pero entonces, multiplicando S_M a ambos lados de la desigualdad, se cumpliría que:

$$\prod_{i=1}^{M-1} S_i * S_M < \prod_{i=1}^{M-1} Y_i * S_M$$

lo cual significa que $< Y_1, Y_2, \ldots, Y_{M-1}, S_M >$ mejoraría el resultado de $< S_1, S_2, \ldots, S_{M-1}, S_M >$ para el problema (1, M, N), lo cual contradice la hipótesis de partida, pues ésta última era la solución óptima de dicho problema.

Se cumple, pues, el principio de optimalidad.

DESCOMPONER UN NÚMERO EN SUMANDOS

DEFINICIÓN RECURSIVA.-

Tal y como hemos visto en la demostración del principio de optimalidad, representaremos el problema a resolver por la terna

(1, M, N)

esto es, el problema que debe determinar el valor de los sumandos del 1 al M, los cuales tienen que sumar N y su producto ha de ser máximo.

DESCOMPONER UN NÚMERO EN SUMANDOS

Existen dos posibilidades:

- manejar subproblemas del tipo **(j, M, n)**, consistente en resolver óptimamente el subproblema formado por los sumandos del j al M y con un valor a obtener como suma de n.

$$< s_j, s_{j+1}, \ldots, s_M >$$

- manejar subproblemas de la forma **(1, j, n)**, consistente en resolver óptimamente el subproblema formado por los sumandos del 1 al j y con un valor a obtener como suma de n.

$$< s_1, \ldots, s_{j-1}, s_j >$$

DESCOMPONER UN NÚMERO EN SUMANDOS

Vamos a optar por considerar los problemas del segundo tipo, es decir,

(1, j, n)

Aunque, al igual que ocurría en el problema de la mochila, el parámetro **1** va a permanecer invariable a lo largo de todo el proceso recursivo, por lo que podemos prescindir del mismo en la representación del problema con el fin de simplificar la notación.

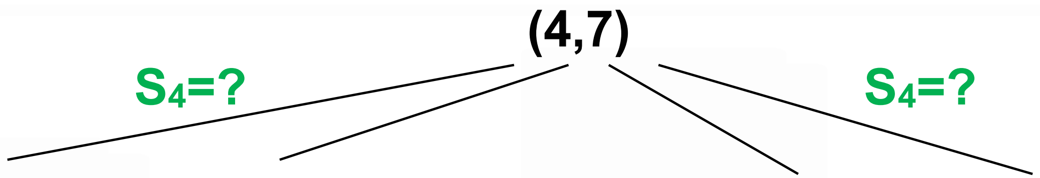
Llamaremos **Sumandos(j, n)** a la solución óptima de este subproblema.

Por tanto, **Sumandos(j, n)** representa el producto máximo de los j sumandos (del 1 al j) que sumados dan n. La llamada inicial a la función corresponde a j=M y n=N.

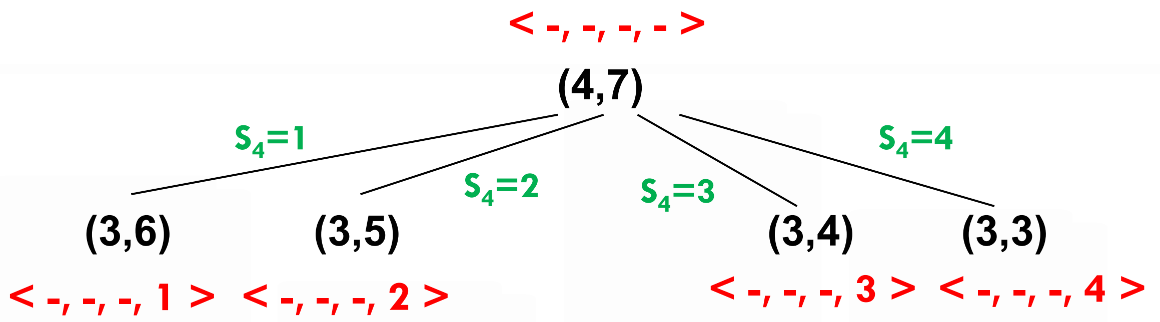
DESCOMPONER UN NÚMERO EN SUMANDOS: N=7 y M=4

¿ Sumandos(1, 4, 7) ?

¿< S_1, S_2, S_3, S_4 >?



DESCOMPONER UN NÚMERO EN SUMANDOS



Sumandos(4,7) = máximo { Sumandos(3,6) * 1, Sumandos(3,5) * 2,
Sumandos(3,4) * 3, Sumandos(3,3) * 4 }

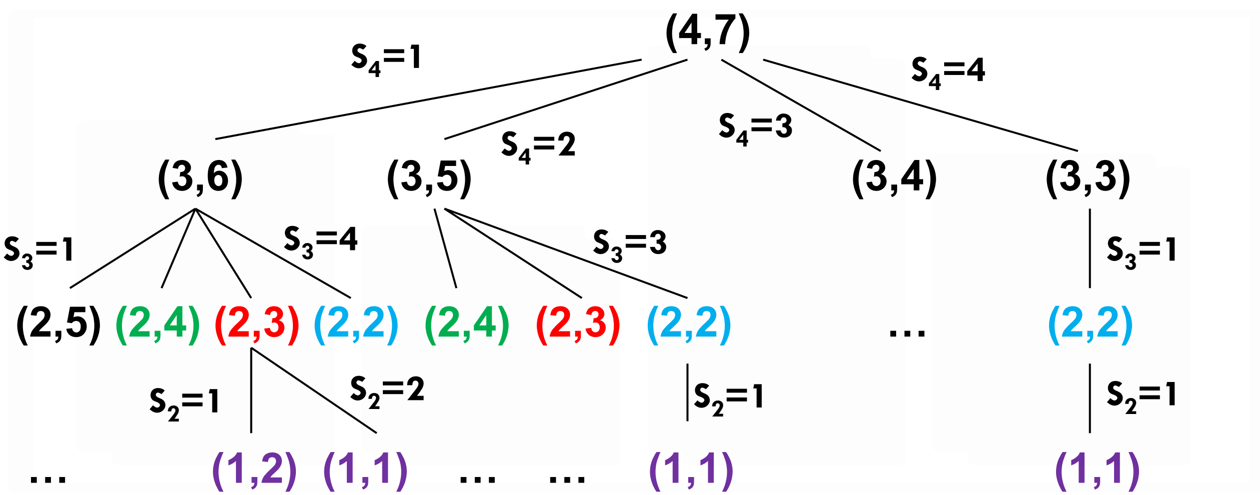
En general,

$$Sumandos(j,n) = \begin{cases} n & si\ j = 1 \\ \max_{1 \leq S_j \leq n-j+1} \{Sumandos(j-1, n-S_j) * S_j\} & si\ j > 1 \end{cases}$$

DESCOMPONER UN NÚMERO EN SUMANDOS

ÁRBOL DE LLAMADAS para el ejemplo M=4 y N=7.-

En él se han desarrollado sólo algunos subproblemas, pero aún así, se puede ver la repetición en el cálculo de subproblemas. En color morado se indican los subproblemas triviales.



DESCOMPONER UN NÚMERO EN SUMANDOS

ESTRUCTURA DE ALMACENAMIENTO.- Dos estructuras bidimensionales

- con M filas (para representar el sumando en curso) y
- con N columnas (para representar todos los posibles estados n del número a conseguir)

Pmax[1..M][1..N] → almacena el producto máximo

Dec[1..M][1..N] → guarda la decisión que proporciona el producto máximo

Sumandos(j, n) → **Pmax[j][n]**

Sumandos(j, n) → **Dec[j][n]**

DESCOMPONER UN NÚMERO EN SUMANDOS

RELLENADO DE LA ESTRUCTURA.-

1° /* problemas triviales */

Pmax	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2							
3							
4							

Dec	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2							
3							
4							

DESCOMPONER UN NÚMERO EN SUMANDOS

RELLENADO DE LA ESTRUCTURA.-

2° /* resto de los subproblemas */

Pmax [4][7] = max { Pmax [3][6]*1, Pmax [3][5]*2, Pmax [3][4]*3, Pmax [3][3]*4 }=

= max {8*1, 4*2, 2*3, 1*4} = **8**

Dec[4][7]=**1** ó Dec[4][7]=**2**

Pmax	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2		1	2	4	6	9	12
3			1	2	4	8	12
4				1	2	4	8

DESCOMPONER UN NÚMERO EN SUMANDOS

RELLENADO DE LA ESTRUCTURA.-

2° /* resto de los subproblemas */

Pmax [4][7] = max { Pmax [3][6]*1, Pmax [3][5]*2, Pmax [3][4]*3, Pmax [3][3]*4 }=

= max {8*1, 4*2, 2*3, 1*4} = **8**

Dec[4][7]=**1** ó Dec[4][7]=**2**

Dec	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2		1	2	2	3	3	4
3			1	2	2	2	3
4				1	2	2	2

DESCOMPONER UN NÚMERO EN SUMANDOS

MOSTRAR SOLUCIÓN.-

1ª parte: Valor que optimiza la función objetivo.

Producto máximo → **Pmax[4][7]**

en general **Pmax[M][N]**

Pmax	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2		1	2	4	6	9	12
3			1	2	4	8	12
4				1	2	4	8

DESCOMPONER UN NÚMERO EN SUMANDOS

MOSTRAR SOLUCIÓN.-

2ª parte: Secuencia óptima de decisiones.

Se realiza un recorrido por la estructura que almacena las decisiones (Dec). Este recorrido está guiado por la definición recursiva.

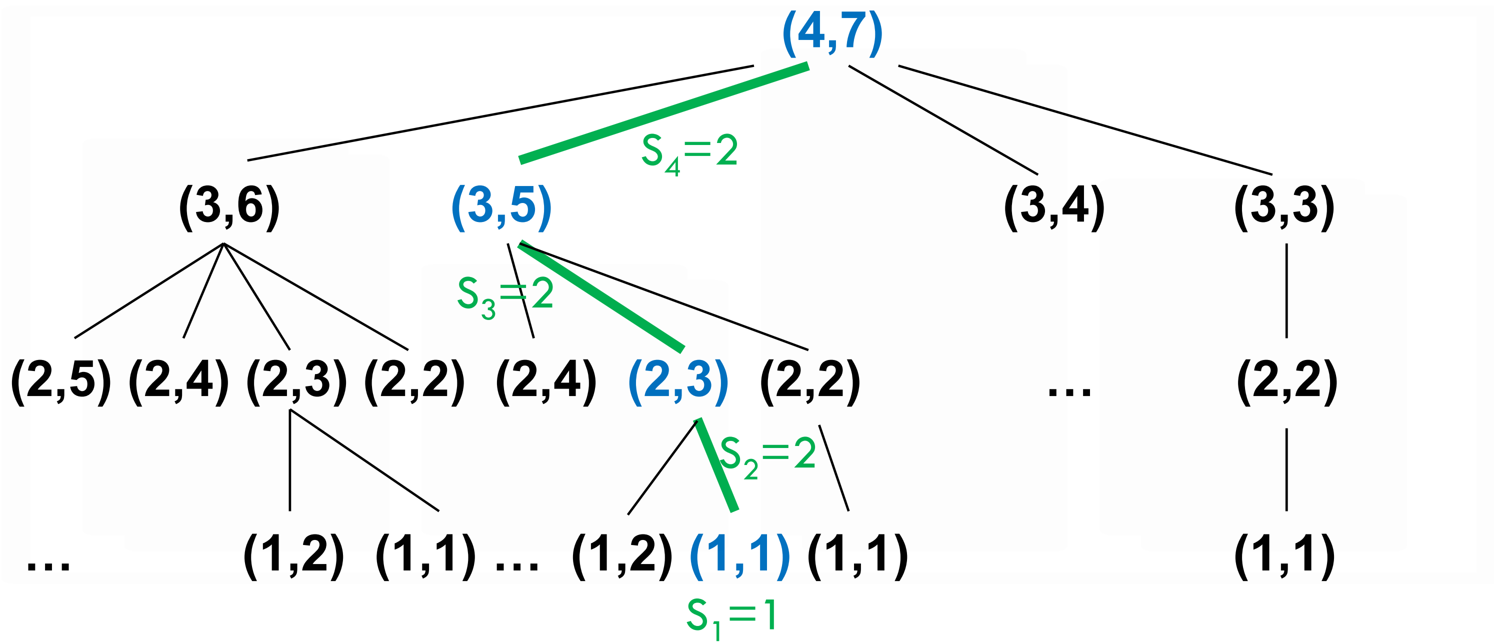
$$< S_1, S_2, S_3, S_4 > = < 1, 2, 2, 2 >$$

Dec	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2		1	2	2	3	3	4
3			1	2	2	2	3
4				1	2	2	2

$$Sumandos(j,n) = \begin{cases} n & si\ j = 1 \\ \max_{1 \le S_j \le n-j+1} \{Sumandos(j-1, n-S_j) * S_j\} & si\ j > 1 \end{cases}$$

DESCOMPONER UN NÚMERO EN SUMANDOS

Secuencia de decisiones óptima vista en el árbol



DESCOMPONER UN NÚMERO EN SUMANDOS: ALGORITMO

Función Sumandos (N, M:entero) retorna (p:entero, S[1..M]:vector de enteros)

/* **Necesita:** número a descomponer (N) y número de sumandos (M) en los que hay que descomponer N.

Produce: El producto de los sumandos de la solución óptima y un vector con el valor de cada sumando.*/

```
var
    /* almacenará el producto máximo de cada subproblema */
    Pmax[1..M][1..N]: matriz de enteros;
    /* registrará la decisión óptima para cada subproblema */
    Dec[1..M][1..N]: matriz de enteros;
    j, Sj, n, p:entero;
    /* Secuencia de decisiones óptima */
    S[1..M]: vector de enteros;
fvar
```

DESCOMPONER UN NÚMERO EN SUMANDOS: ALGORITMO

/* inicializar la estructura de almacenamiento con los resultados de los problemas triviales */

```
para n=1 hasta N hacer
    Pmax[1][n] = n;
    Dec[1][n] = n;
fpara
```

DESCOMPONER UN NÚMERO EN SUMANDOS: ALGORITMO


```
/* rellenar el resto de la estructura de almacenamiento con los resultados de los problemas
no triviales, en sentido ascendente */
para j = 2 hasta M hacer
        para n = j hasta N hacer
                Pmax[ j ][ n ]=1;
                para Sj = 1 hasta n – j + 1 hacer
                        p = Pmax[ j -1 ][ n – Sj ] * Sj;
                        si p ≥ Pmax[ j ][ n ] entonces
                                Pmax[ j ][ n ] = p;
                                Dec[ j ][ n ] = Sj;
                        fsi
                fpara
        fpara
fpara
```

DESCOMPONER UN NÚMERO EN SUMANDOS: ALGORITMO


```
/* solución */
j = M;
n = N;
mientras (j > 0) hacer
    S[ j ]=Dec[ j ][ n ];
    n = n - S[ j ];
    j = j - 1;
fmientras
retorna (Pmax[M][N], S);
ffunción
```

EJEMPLO 3: EMBARCADEROS

A lo largo de la ribera de un río hay n embarcaderos. Se tiene una matriz C de tamaño n x n, de manera que C[i][j] indica el coste de ir directamente del embarcadero i al embarcadero j ($1 \leq i \leq j \leq n$).

Se asume que el coste de ir de un embarcadero a él mismo es cero, o sea,

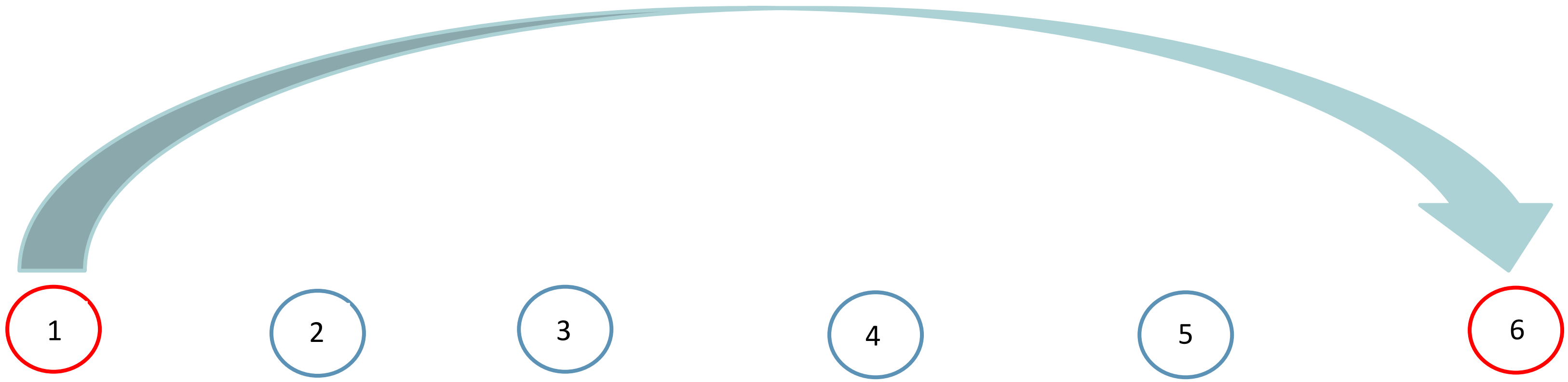
$$C[\,i\,][\,i\,] = 0 \ \forall i: 1..n$$

La única limitación que existe en este problema es que desde un embarcadero no se puede volver a ninguno de los anteriores.

Se pretende saber cuál es el coste mínimo para ir desde el primer embarcadero al último y cuál es el trayecto que proporciona este coste.

EMBARCADEROS: EJEMPLO n=6

¿Embarcaderos(1,6)?



Posibles soluciones factibles:

1,6

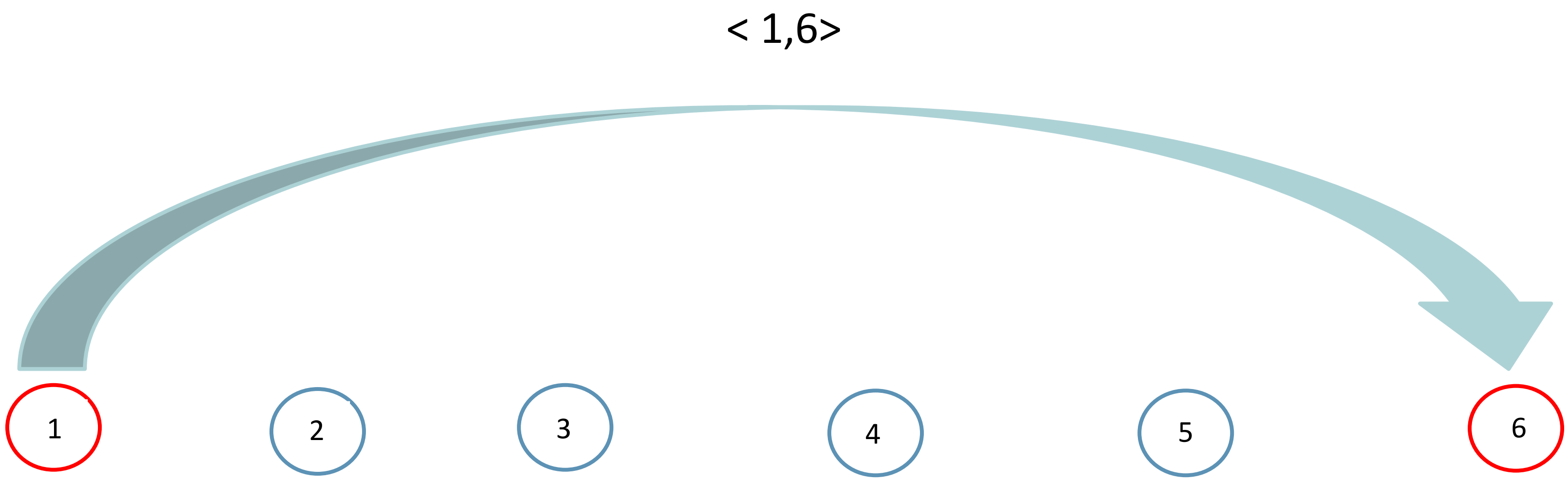
1,2,6 ó 1,3,6 ó 1,4,6 ó 1,5,6

1,2,5,6 ó 1,3,5,6 ó 1,4,5,6 …

…

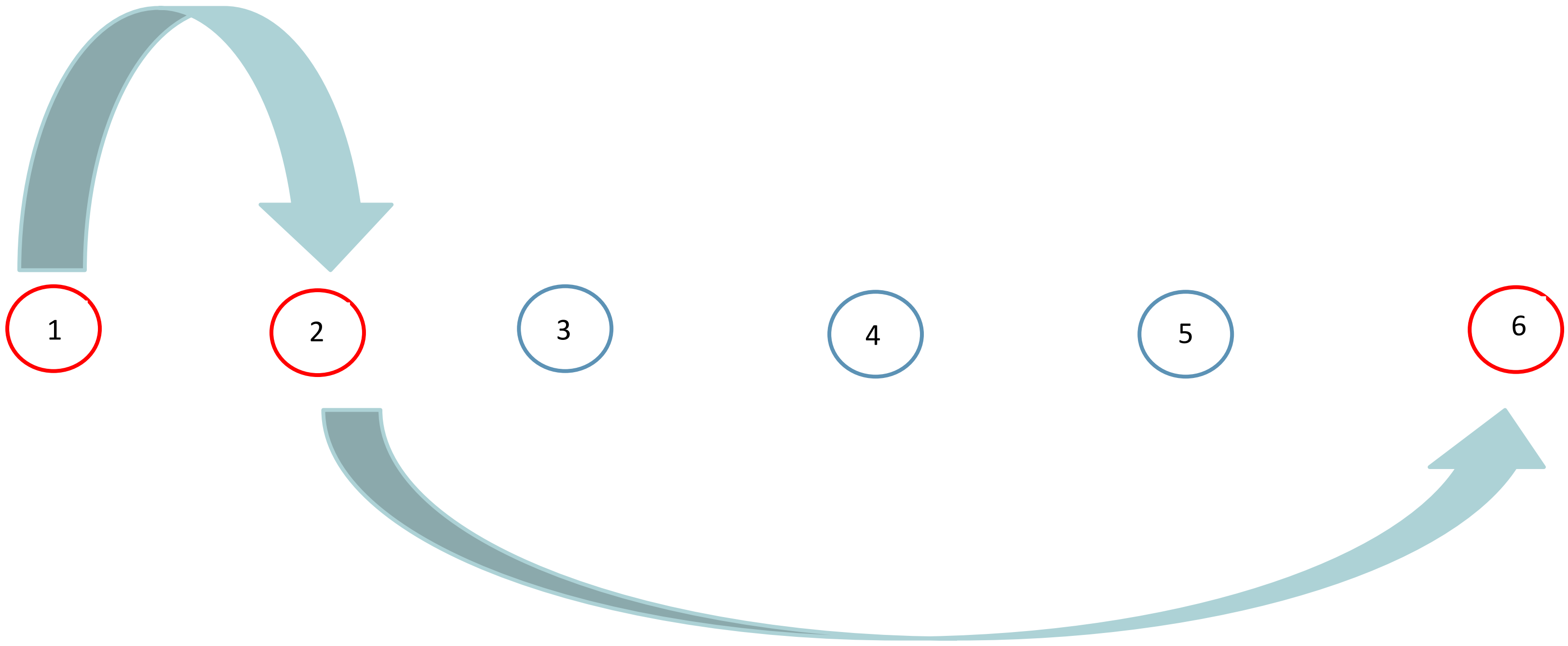
1,2,3,4,5,6

EMBARCADEROS: EJEMPLO n=6



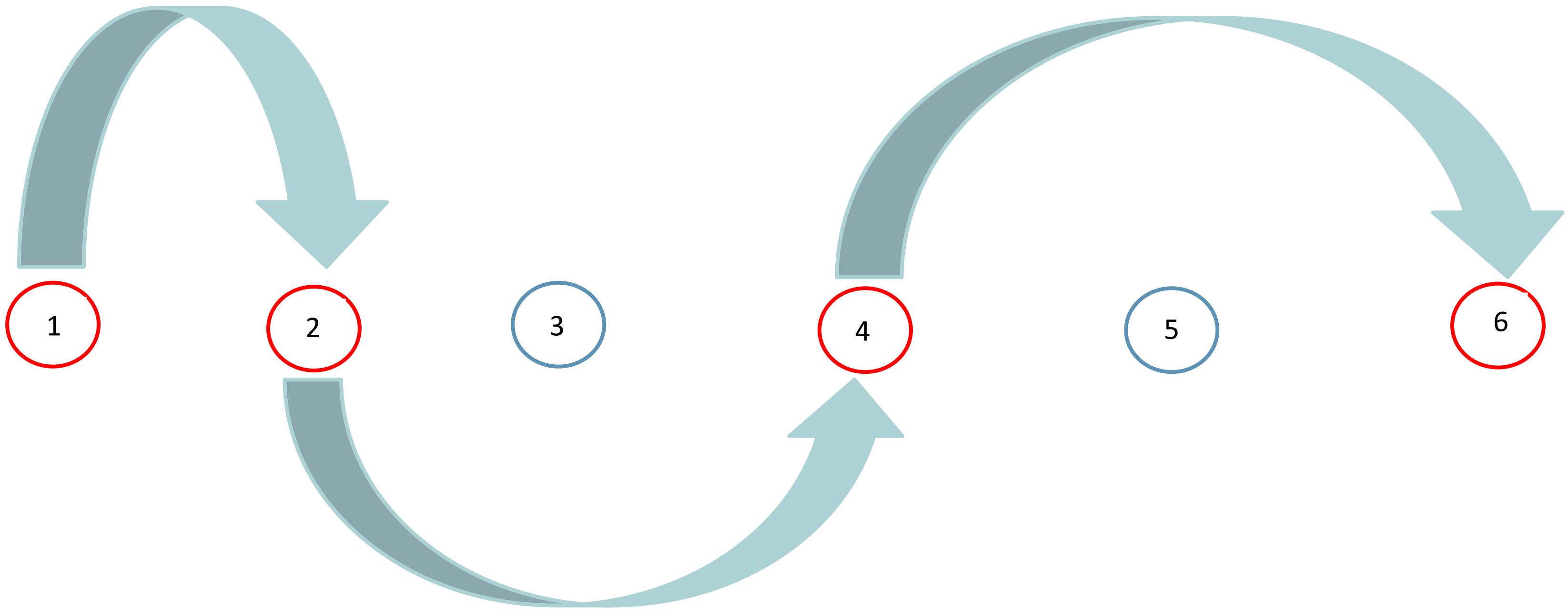
EMBARCADEROS: EJEMPLO n=6

< 1, 2, 6>



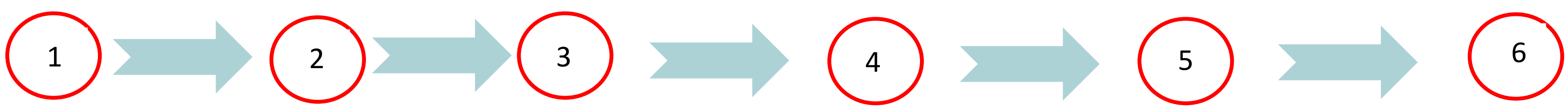
EMBARCADEROS: EJEMPLO n=6

< 1, 2, 4, 6>



EMBARCADEROS: EJEMPLO n=6

< 1, 2, 3, 4, 5, 6>



SOLUCIÓN EXPRESADA COMO SECUENCIA DE DECISIONES

$$\langle x_1, x_2, \ldots, x_s \rangle \quad \text{con } 1 \leq s \leq n\text{-}1$$

en cada una de las decisiones optamos por uno de los embarcaderos situados más adelante en el río con respecto al embarcadero actual.

x_1 representa el embarcadero al que viajamos tras el embarcadero 1, por tanto

$$x_1 \in \{ j \,/\, 1 < j \leq n \} = \{ 2, 3, \ldots, n \}$$

EMBARCADEROS: SECUENCIA DE DECISIONES

$$<x_1, x_2, \ldots, x_s>$$

x_2 por su parte, representa el embarcadero al que viajamos desde x_1, así

$x_2 \in \{ j \,/\, x_1 < j \leq n \} = \{ x_1+1, x_1+2, \ldots, n \}$

y así sucesivamente hasta alcanzar el embarcadero de destino (x_s=n), o lo que es lo mismo, hasta que el conjunto de alternativas sea vacío.

EMBARCADEROS

FUNCIÓN OBJETIVO Y RESTRICCIONES

Se trata de minimizar por tanto el coste

$$C[1][x_1]+C[x_1][x_2]+\ldots+C[x_{s-1}][x_s]$$

o lo que es lo mismo

$$C[1][x_1]+C[x_1][x_2]+\ldots+C[x_{s-1}][n]$$

sujeto a

$$1 < x_1 < x_2 < \ldots < x_s$$

EMBARCADEROS

PRINCIPIO DE OPTIMALIDAD

Sea $<x_1, x_2, \ldots, x_s>$ la solución óptima para el problema de los embarcaderos del 1 al n. Dicho problema lo vamos a representar como (1, n). Y su valor asociado es:

$$C[1][x_1]+C[x_1][x_2]+\ldots+C[x_{s-1}][n]$$

Dicha secuencia cumple que $1 < x_1 < x_2 < \ldots < x_s$

Si prescindimos de la primera decisión, x_1, la subsecuencia que nos queda es

$<x_2, \ldots, x_s>$, la cual es la solución óptima al problema de los embarcaderos del x_1 al n.

Dicho problema lo representaremos como (x_1, n).

EMBARCADEROS: PRINCIPIO DE OPTIMALIDAD

Supongamos que existiese otra secuencia de decisiones que proporcionase un coste menor para el problema (x_1, n), siendo ésta:

$$<y_1, \ldots, y_m> \text{ sujeta a } x_1 < y_1 < y_2 < \ldots < y_{m-1} < y_m$$

donde $y_m=n$.

Entonces ocurrirá que

$$C[x_1][y_1]+ \ldots + C[y_{m-1}][n] < C[x_1][x_2]+ \ldots + C[x_{s-1}][n]$$

EMBARCADEROS: PRINCIPIO DE OPTIMALIDAD

Pero sumando a ambos lados el coste $C[1][x_1]$ nos encontraríamos con que

$$\mathbf{C[1][x_1] +} C[x_1][y_1] + \ldots + C[y_{m-1}][n] < \mathbf{C[1][x_1] +} C[x_1][x_2] + \ldots + C[x_{s-1}][n]$$

lo cual es contradictorio, porque entonces la secuencia $<x_1, y_1, \ldots, y_m>$ resolvería el problema (1, n) con un coste inferior a la secuencia $<x_1, x_2, \ldots, x_s>$, pues ésta última era la solución óptima de dicho problema.

EMBARCADEROS: EJEMPLO n=6, PRIMERA DECISIÓN

¿Embarcaderos(1,6)?

1ª decisión (¿ x_1?): ¿ir al embarcadero 2? ¿ir al embarcadero 3? … ¿ir al embarcadero 6?

Analizar todas las posibilidades y retener la de coste mínimo:

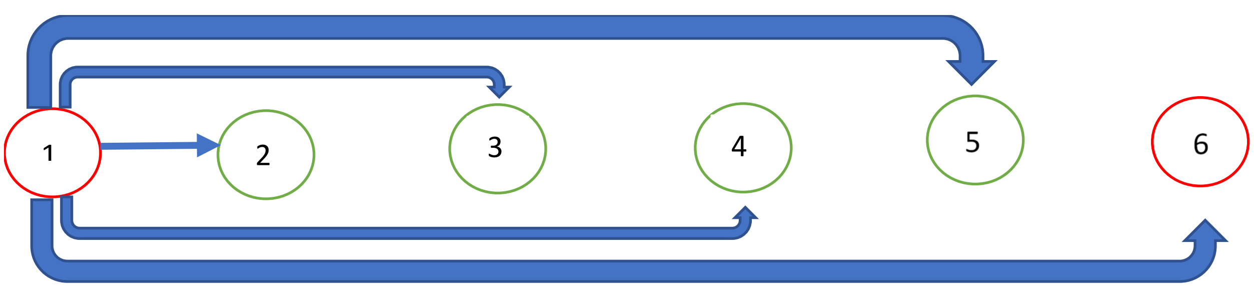
C[1][2] + Embarcaderos(2,6)

C[1][3] + Embarcaderos(3,6)

C[1][4] + Embarcaderos(4,6)

C[1][5] + Embarcaderos(5,6)

C[1][6] + Embarcaderos(6,6)



EMBARCADEROS

DEFINICIÓN RECURSIVA

Representaremos el problema a través de dos parámetros **(1, n)**, que corresponde al coste mínimo para ir del embarcadero 1 al embarcadero n.

De forma genérica, vamos a manejar los subproblemas **(j, n)** consistentes en resolver óptimamente el problema de los embarcaderos desde el embarcadero **j** al embarcadero **n.**

Dado que el parámetro **n** va a permanecer invariable a lo largo de toda la secuencia de decisiones, podemos prescindir del mismo en la representación del problema con el fin de simplificar la notación.

EMBARCADEROS

DEFINICIÓN RECURSIVA

Plantearemos la siguiente ecuación recursiva

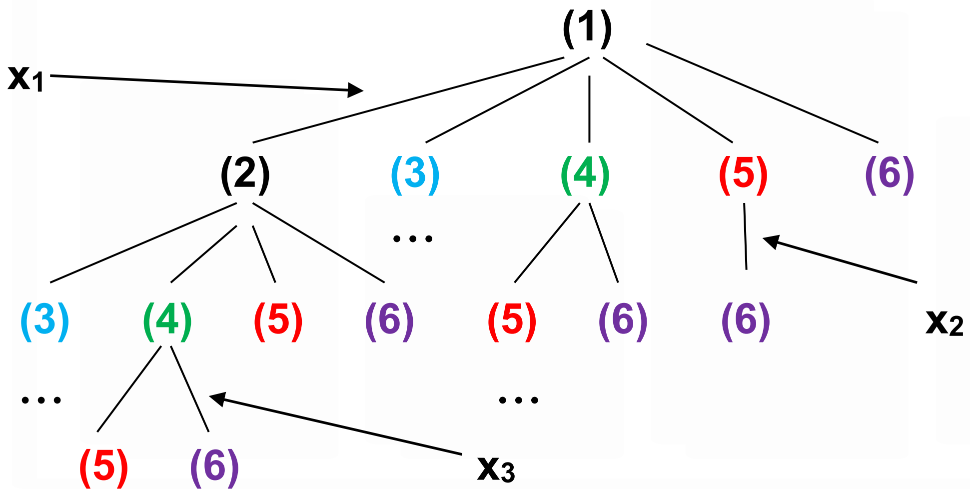
$$Embarcaderos(j) = \begin{cases} 0 & si\ j = n \\ min_{j<k\leq n}\ \{C[j][k] + Embarcaderos(k)\} & si\ j < n \end{cases}$$

donde Embarcaderos(j) devolverá el coste mínimo para ir desde el embarcadero j al embarcadero n.

Llamada inicial a la función: Embarcaderos(1)

ÁRBOL DE LLAMADAS Y REPETICIÓN DE SUBPROBLEMAS

n=6



EMBARCADEROS

ORDEN DE LOS SUBPROBLEMAS

1° TRIVIALES (j = n), en nuestro caso j = 6.

2° el subproblema Embarcaderos(5) ya que requiere el subproblema Embarcaderos(6) y éste ya está resuelto.

3° el subproblema Embarcaderos(4) ya que requiere de Embarcaderos(5) y de Embarcaderos(6) y éstos ya están resueltos.

…

Finalmente resolveremos Embarcaderos(1) ya que necesita de todos los anteriores: Embarcaderos(2), Embarcaderos(3), … Embarcaderos(6) y éstos ya están resueltos.

En términos generales, los subproblemas se resuelven en el siguiente orden: Embarcaderos(n), Embarcaderos(n-1), …, Embarcaderos(1).

EMBARCADEROS

ESTRUCTURAS DE ALMACENAMIENTO

Utilizaremos estructuras unidimensionales, pues hay un único parámetro en la ecuación recursiva. Los vectores tendrán n posiciones, tantas como embarcaderos desde los que se puede viajar a n.

Costemin[1..n] → almacena el coste mínimo

Dec[1..n] → guarda la decisión que proporciona el coste mínimo

Embarcaderos(j) → **Costemin[j]**

Embarcaderos(j) → **Dec[j]**

EMBARCADEROS

Ejemplo: n=6 y la matriz C de costes es la siguiente

C	1	2	3	4	5	6
1	0	10	6	15	23	30
2		0	5	1	11	18
3			0	8	12	25
4				0	6	3
5					0	7
6						0

EMBARCADEROS

RELLENADO DE LAS ESTRUCTURAS

Primero rellenamos el caso base (Embarcaderos(6)=0) que corresponde a:

Costemin

1	2	3	4	5	6
					0

Dec

1	2	3	4	5	6
					6

EMBARCADEROS: RELLENADO DE LAS ESTRUCTURAS

Seguidamente rellenamos el resto de la estructura desde la posición n-1 a la 1.

Luego, calculamos

Costemin[5] = $\min_{5 < k \leq 6}$ { C[5][k] + Costemin[k]} =min {C[5][6]+Costemin[6] } =

= min { 7+0 } = **7**   Dec[5]=**6**

Costemin

1	2	3	4	5	6
				7	0

Dec

1	2	3	4	5	6
				6	6

EMBARCADEROS

Costemin[4] = $\min_{4 < k \leq 6}$ { C[4][k]+Costemin[k] } = min { C[4][5]+Costemin[5], C[4][6]+Costemin[6] } = min{ 6+7,3+0 } = **3** Dec[4]=**6**

Costemin	1	2	3	4	5	6
				3	7	0

Dec	1	2	3	4	5	6
				6	6	6

EMBARCADEROS

Costemin[3] = $\min_{3 < k \leq 6}$ { C[3][k]+Costemin[k] } = min {C[3][4]+Costemin[4], C[3][5]+Costemin[5], C[3][6]+Costemin[6] } = min { 8+3,12+7,25+0 } = **11**

Dec[3]=**4**

Costemin	1	2	3	4	5	6
			11	3	7	0

Dec	1	2	3	4	5	6
			4	6	6	6

EMBARCADEROS

$\text{Costemin}[2] = \min_{2 < k \leq 6} \{ C[2][k] + \text{Costemin}[k] \} = \min \{ C[2][3] + \text{Costemin}[3],$

$C[2][4] + \text{Costemin}[4], C[2][5] + \text{Costemin}[5], C[2][6] + \text{Costemin}[6] \} =$

$= \min \{ 5+11, 11+3, 11+7, 18+0 \} = 4$ Dec[2]=4

Costemin

1	2	3	4	5	6
	4	11	3	7	0

Dec

1	2	3	4	5	6
	4	4	6	6	6

EMBARCADEROS

Y por último,

Costemin[1] = min { C[1][2]+Costemin[2], C[1][3]+Costemin[3], C[1][4]+Costemin[4], C[1][5]+Costemin[5]+, C[1][6]+Costemin[6] } = min { 10+4, 6+11, 15+3, 23+7, 30+0 }=**14** Dec[1]=**2**

Costemin	1	2	3	4	5	6
	14	4	11	3	7	0

Dec	1	2	3	4	5	6
	2	4	4	6	6	6

EMBARCADEROS

CÓMO OBTENER LA SOLUCIÓN

1ª parte: Valor que optimiza la función objetivo.

	1	2	3	4	5	6
Costemin	14	4	11	3	7	0

El coste óptimo es Costemin[1]=**14**

EMBARCADEROS: CÓMO OBTENER LA SOLUCIÓN

Secuencia de decisiones óptima (itinerario óptimo).-

Dec

1	2	3	4	5	6
2	4	4	6	6	6

Dec[1] = 2 → x_1

Dec[2] = 4 → x_2

Dec[4] = 6 → x_3

EMBARCADEROS

El viaje consistirá en ir desde el embarcadero 1 hasta el embarcadero 2, desde el 2 hasta el 4, y finalmente, desde el 4 hasta el 6.

$$\langle x_1, x_2, \ldots, x_s \rangle$$

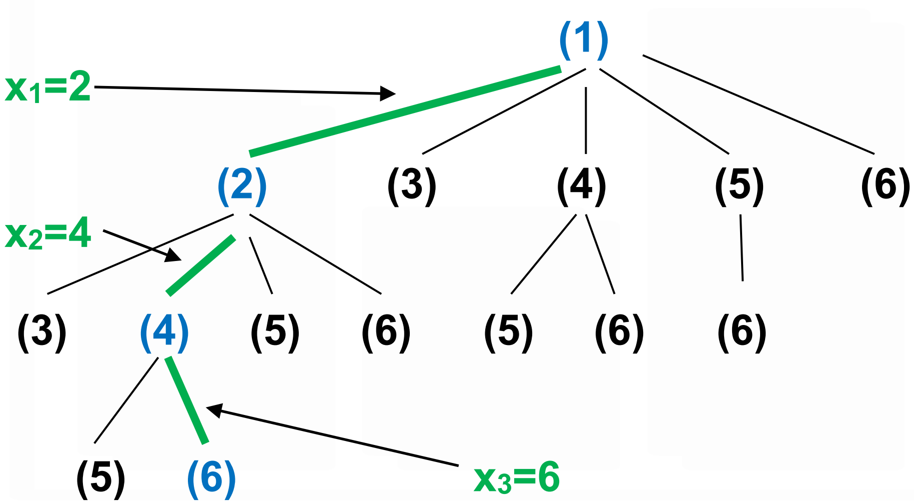
s=3

$$\langle x_1, x_2, x_3 \rangle$$

$$\langle 2, 4, 6 \rangle$$

EMBARCADEROS: SOLUCIÓN ÓPTIMA EN EL ÁRBOL

Secuencia óptima.- $\langle x_1, x_2, x_3 \rangle = \langle 2, 4, 6 \rangle$



EMBARCADEROS: ALGORITMO


```
Función Embarcaderos(C[1..n][1..n]:matriz de enteros; n:entero)
                                retorna (p:entero, f[1..n]:vector de enteros)
var
  Costemin[1..n] : vector de enteros;
  Dec[1..n] : vector de enteros; j , k : entero;
  X[1..n]: vector de enteros;
fvar

/* inicializar la estructura de almacenamiento con los resultados de los problemas
   triviales */
   Costemin [n]=0;
   Dec[n]=n;
```

EMBARCADEROS: ALGORITMO

/* rellenar el resto de la estructura de almacenamiento con los resultados de los problemas no triviales, en sentido ascendente, de problemas más sencillos a más complejos */

```
para j = n-1 hasta 1 hacer
    Costemin[ j ]=∞
    para k = j +1 hasta n hacer
        si Costemin[ j ] > C[ j ][ k ] + Costemin[ k ] entonces
            Costemin[ j ] = C[ j ][ k ] + Costemin[ k ];
            Dec[ j ] = k;
        fsi
    fpara
fpara
```

EMBARCADEROS: ALGORITMO


```
/* mostrar solución */

X[1]=Dec[1];
 j = 1;
mientras X[ j ] ≠ n hacer
     j = j + 1;
     X[ j ] = Dec[X[ j - 1]];
fmientras

retorna (Costemin[1], X)
```

APLICACIONES DE PROGRAMACIÓN DINÁMICA: EJEMPLOS

- Camino más corto que une cada par de vértices de un grafo:
  - Algoritmo de Floyd_Warshall (no hay pesos negativos)
  - Algoritmo de Bellman-Ford (hay pesos negativos)
- Distancia de edición (también conocida como distancia de Levenshtein) mide la diferencia entre dos cadenas s y t como el número mínimo de operaciones de edición que hay que realizar para convertir una cadena en otra)
  - Correctores ortográficos,
  - Detección de plagios,
  - …
- Distancia entre dos series temporales: $O(n^2)$
  - Algoritmo DTW permite medir la similitud entre dos secuencias que pueden variar en el tiempo o en el espacio.
- Análisis sintáctico: $O(n^3)$
  - Algoritmo Cocke-Younger-Kasami (Algoritmo CKY),
  - Algoritmo de Early

APLICACIONES DE PROGRAMACIÓN DINÁMICA: EJEMPLOS

- Algoritmo de Viterbi:
  - decodificación de señales,
  - procesamiento de lenguaje natural,
  - bioinformática
  - …
- Multiplicación encadenada de matrices
- Problema de subsecuencia común más larga (LCS problem) en el que se trata de encontrar la subsecuencia más larga que es común en un conjunto de secuencias. El problema de LCS es uno de los problemas clásicos de las ciencias computacionales y es la base de programas que comparan datos como la utilidad diff.