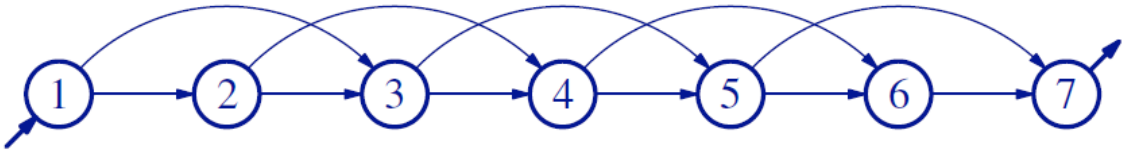
Programación dinámica:

- Obtención de la secuencia óptima.
- Reducción de la complejidad espacial

Un par de problemas sobre el río Congo

A lo largo del río Congo hay E embarcaderos a los que nombramos con los números enteros $1, 2, \ldots, E$. Es posible ir en canoa desde un embarcadero a cualquiera de los dos siguientes en la dirección de la corriente. No se puede navegar contra corriente, ni tampoco ir más allá del segundo embarcadero sin efectuar escala alguna.



Primer problema: camino de menor coste

Dada una función de ponderación que asigna un coste (positivo) a cada arco, calcular el camino de menor coste del primer al último embarcadero y su coste.

Algoritmo recursivo

A partir de la ecuación recursiva,

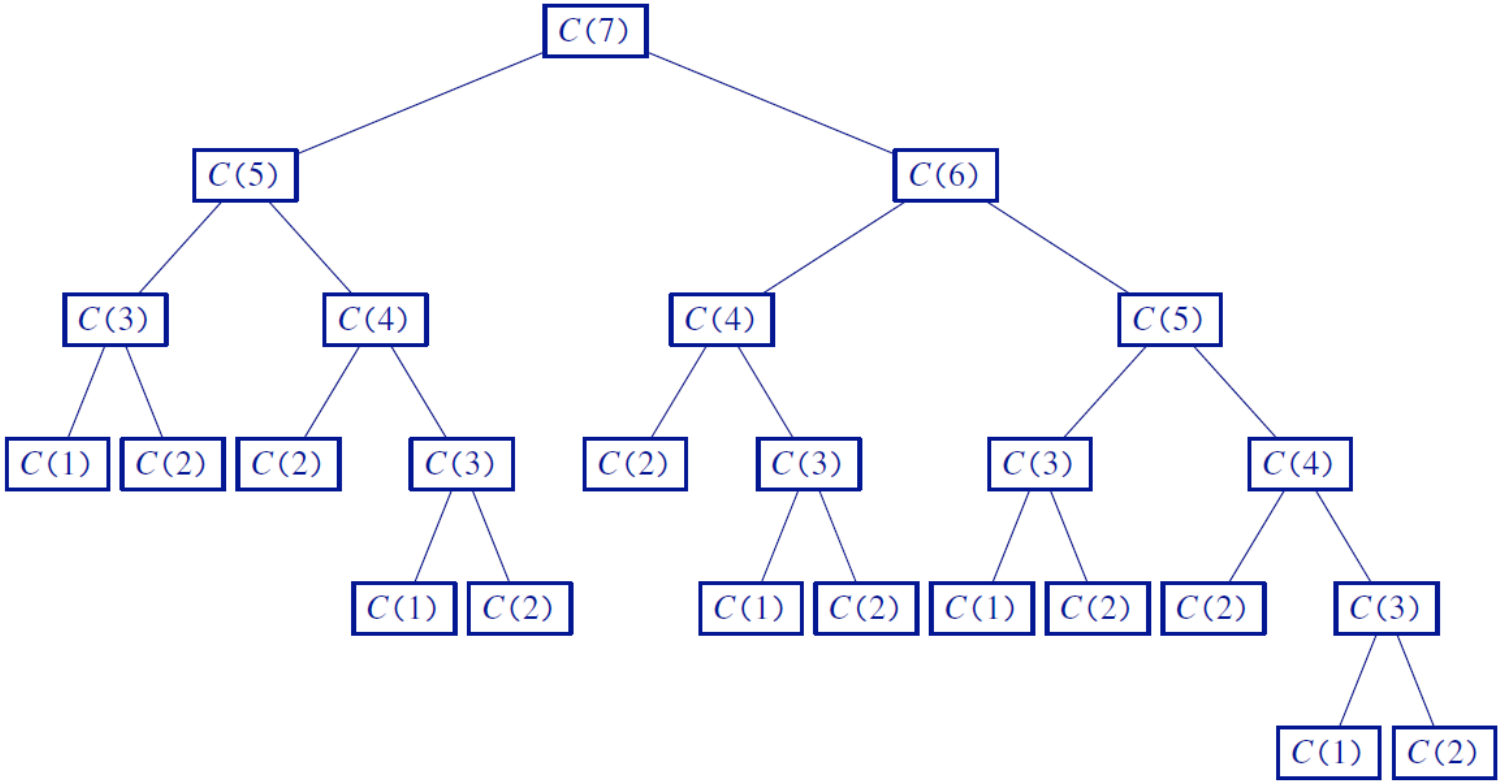
$$C(i) = \begin{cases} 0, & \text{si } i = 1; \\ c(1,2), & \text{si } i = 2; \\ \min(C(i-2)+c(i-2,i), C(i-1)+c(i-1,i)), & \text{si } i > 2, \end{cases}$$

es fácil implementar un programa Python que cálcule $C(E)$ recursivamente:

congo.py

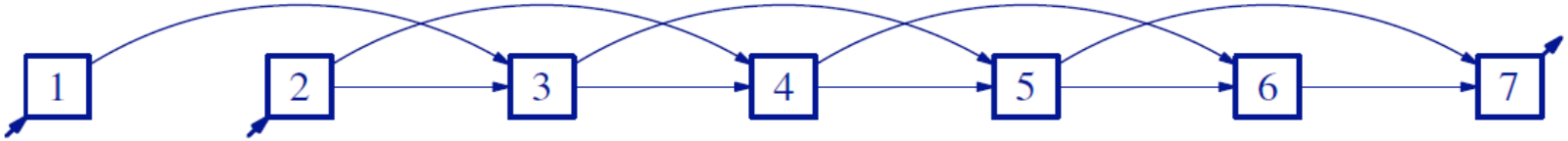
```
1 def recursive_cheapest_price(E, c):
2     def C(i):
3         if i == 1:   return 0
4         elif i == 2: return c[1,2]
5         else:        return min(C(i-2) + c[i-2,i], C(i-1) + c[i-1,i])
6     return C(E)
```

Árbol de llamadas recursivas efectuadas al calcular $C(7)$. Cada rama del árbol corresponde a un trayecto diferente. La rama de más a la izquierda, por ejemplo, corresponde al trayecto $(1,3,5,7)$, y la de más a la derecha, al camino $(1,2,3,4,5,6,7)$. El árbol permite interpretar que el algoritmo explora recursivamente todo posible trayecto y «escoge» el de menor coste.



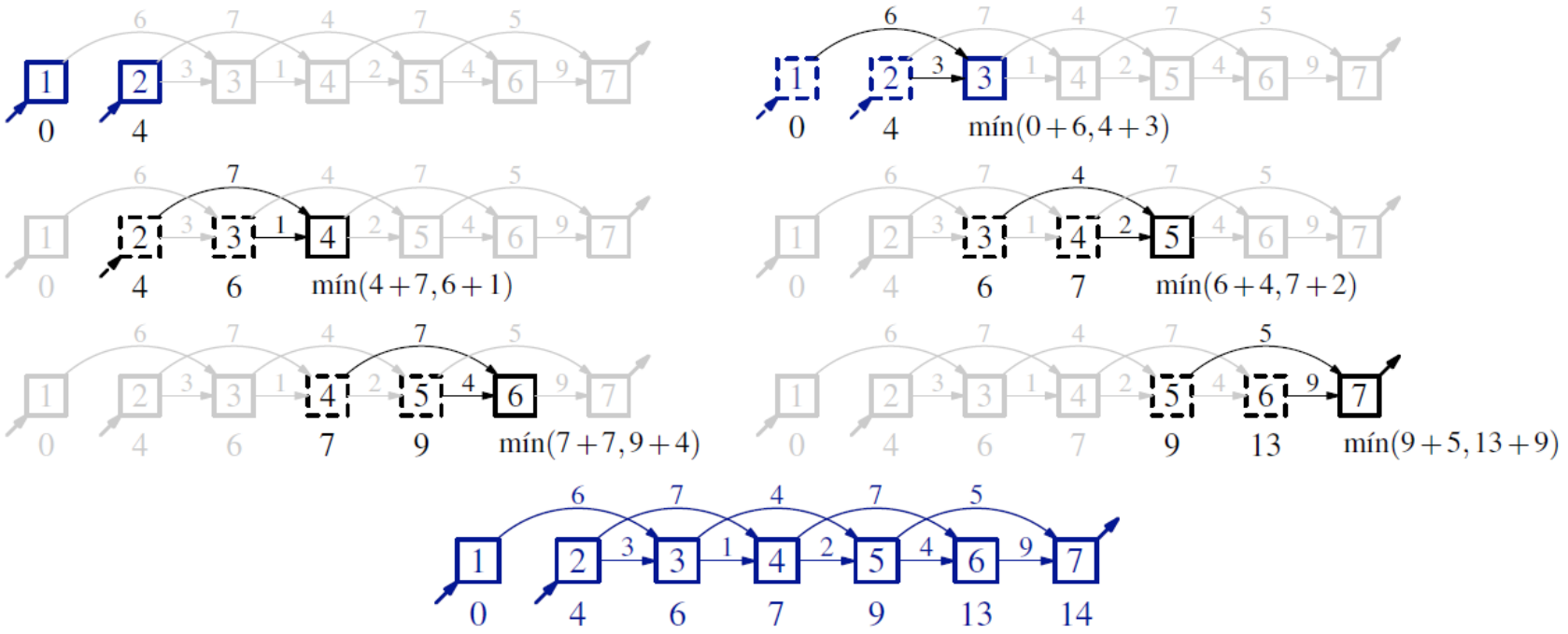
Una versión iterativa

El coste de resolver repetidamente el mismo subproblema (misma llamada recursiva) puede mejorarse resolviendo una sóla vez cada subproblema (identificado por el valor de los parámetros de la llamada recursiva), y almacenando el resultado en una tabla a la que se podrá acceder cada vez que se necesite ese valor (sin necesidad de volver a calcularlo). Para ello es necesario conocer la dependencia de cada subproblema respecto a los otros, es decir qué resultados de otros subproblemas necesita otro subproblema para ser resuelto. La figura representa los resultados almacenados de los diferentes subproblemas, y la dependencia entre ellos (que coincide con las llamadas del algoritmo recursivo cuando quiere resolver cada uno de los subproblemas).



congo.py

```
def iterative_cheapest_price1(E, c):
    C = {}
    C[1] = 0
    C[2] = c[1,2]
    for i in xrange(3, E+1):
        C[i] = min( C[i-1] + c[i-1,i], C[i-2] + c[i-2,i] )
    return C[E]
```



La secuencia de embarcaderos

Hemos resuelto eficientemente el problema de calcular el coste del camino más barato entre los embarcaderos primero y último, pero no sabemos qué secuencia de embarcaderos forma ese trayecto. Podemos aplicar una técnica de punteros hacia atrás para asociar a cada vértice i su vértice predecesor en el camino óptimo del embarcadero 1 al i.

congo.py

```
1  from offsetarray import OffsetArray
2
3  def cheapest_price(E, c):
4      C, B = OffsetArray([None] * E), OffsetArray([None] * E)
5      C[1] = 0
6      C[2], B[2] = c[1,2], 1
7      for i in xrange(3, E+1):
8          if C[i-1] + c[i-1,i] < C[i-2] + c[i-2,i]:
9              C[i] = C[i-1] + c[i-1,i]
10             B[i] = i-1
11         else:
12             C[i] = C[i-2] + c[i-2,i]
13             B[i] = i-2
14     path = []
15     i = E
16     while i != None:
17         path.append(i)
18         i = B[i]
19     path.reverse()
20     return path
```

Otra opción es recuperar el camino usando solamente la información que hay en el vector C.

congo.py

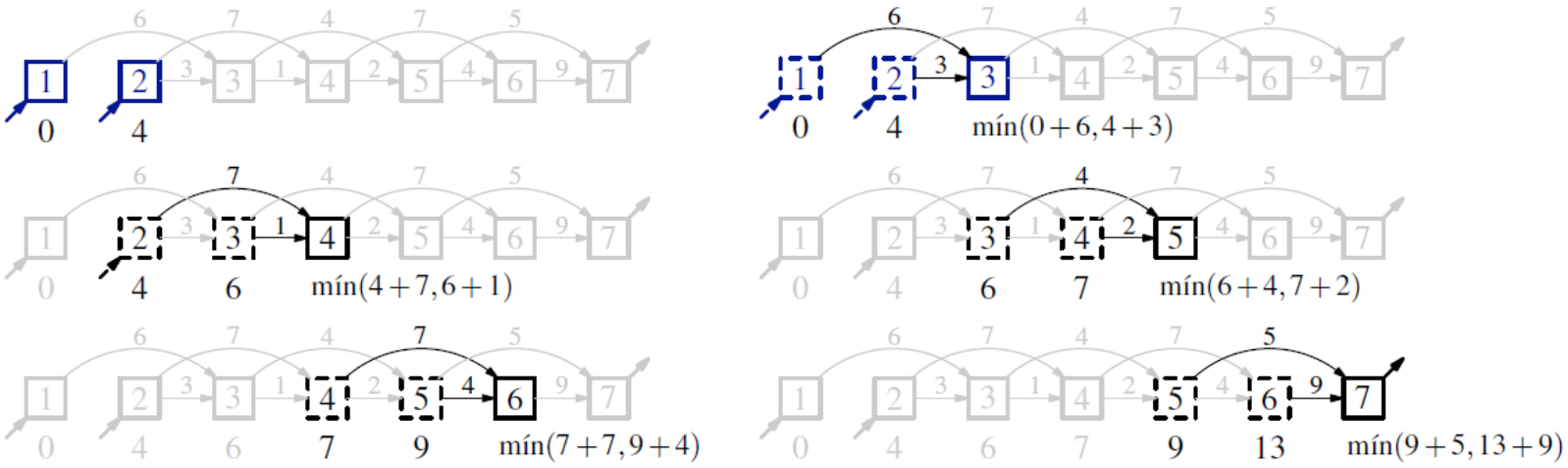
```
1  from offsetarray import OffsetArray
2
3  def cheapest_price2(E, c):
4      C = OffsetArray([None] * E)
5      C[1] = 0
6      C[2] = c[1,2]
7      for i in xrange(3, E+1):
8          C[i] = min(C[i-1] + c[i-1,i], C[i-2] + c[i-2,i] )
9      path = [E]
10     i = E
11     while i != 1:
12         if C[i] == C[i-1] + c[i-1,i]: i = i - 1
13         else: i = i - 2
14         path.append(i)
15     path.reverse()
16     return path
```

```
1  from congo import cheapest_price2
```

Reducción de complejidad espacial

Si sólo estamos interesados en el precio del trayecto más barato al viajar entre los embarcaderos 1 y E, podemos ahorrar memoria y reducir el coste espacial a $O(1)$.

La idea clave radica en considerar que cuando estamos resolviendo el cálculo del coste mínimo de un trayecto entre los embarcaderos 1 e i, sólo necesitamos conocer el precio del trayecto más barato entre los embarcaderos 1 e $i-1$ por una parte, y 1 e $i-2$ por otra. Cualquier otro precio puede ser «olvidado» sin problema alguno.



congo.py

```
def iterative_cheapest_price(E, c):
    C2 = 0
    C1 = c[1,2]
    for i in xrange(3, E+1):
        C3 = min( C1 + c[i-1,i] , C2 + c[i-2,i] )
        C2, C1 = C1, C3
    return C3
```

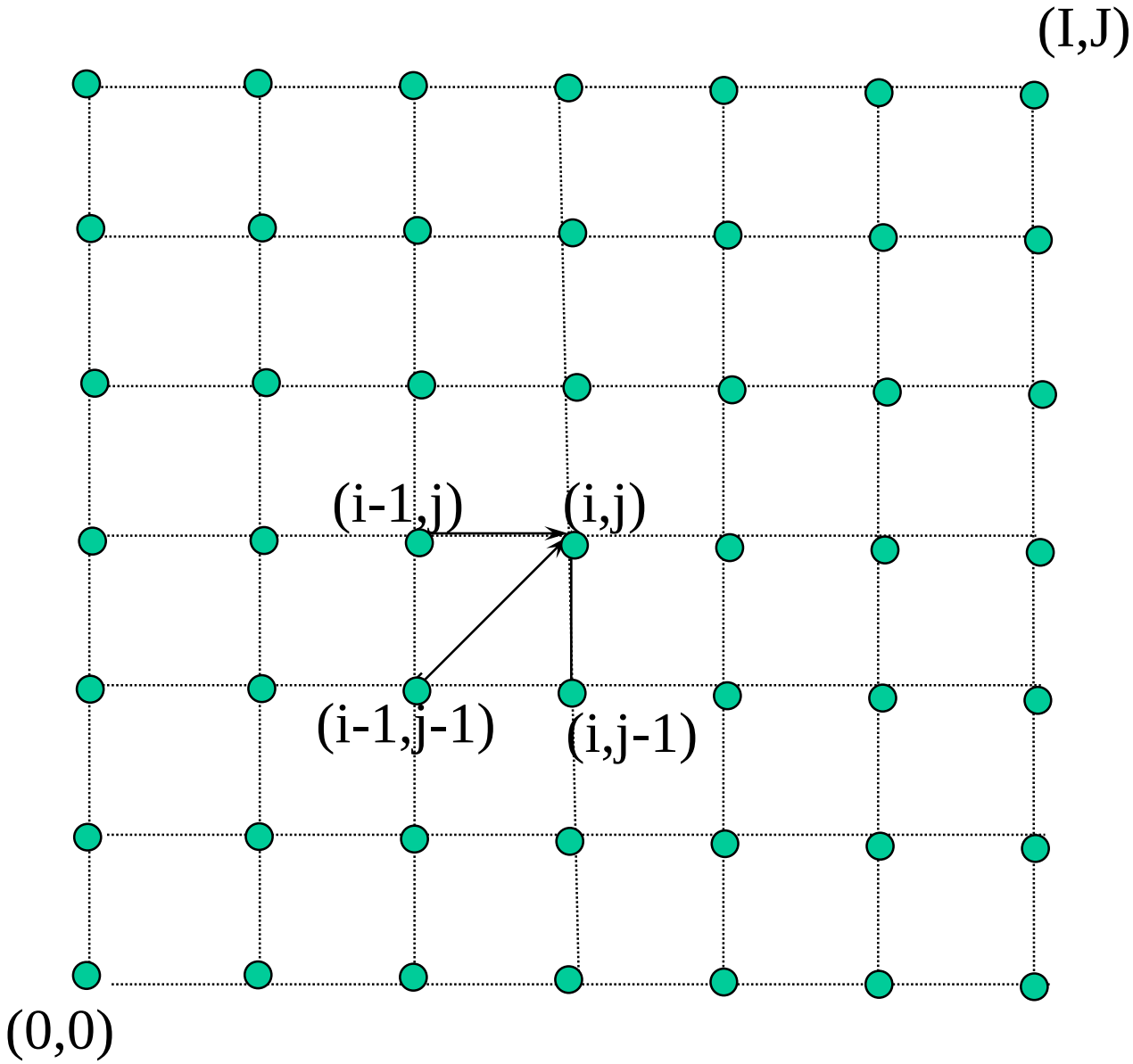
```
from congo import iterative_cheapest_price

c = {(1,2): 4, (1,3): 6,  (2,3): 3, (2,4): 7,  (3,4): 1, (3,5): 4,
     (4,5): 2, (4,6): 7,  (5,6): 4, (5,7): 5,  (6,7): 9 }
print 'Coste mínimo ir del embarcadero 1 al 7:', iterative_cheapest_price(7, c)
```

```
Coste mínimo ir del embarcadero 1 al 7: 14
```

Sólo hemos necesitado 3 variables para almacenar resultados intermedios. El coste espacial es, efectivamente $O(1)$.

Ejemplo de la cuadrícula



Función coste (I,J:**N**):**R**;
var T: matriz[1..I,1..J] de **R**;

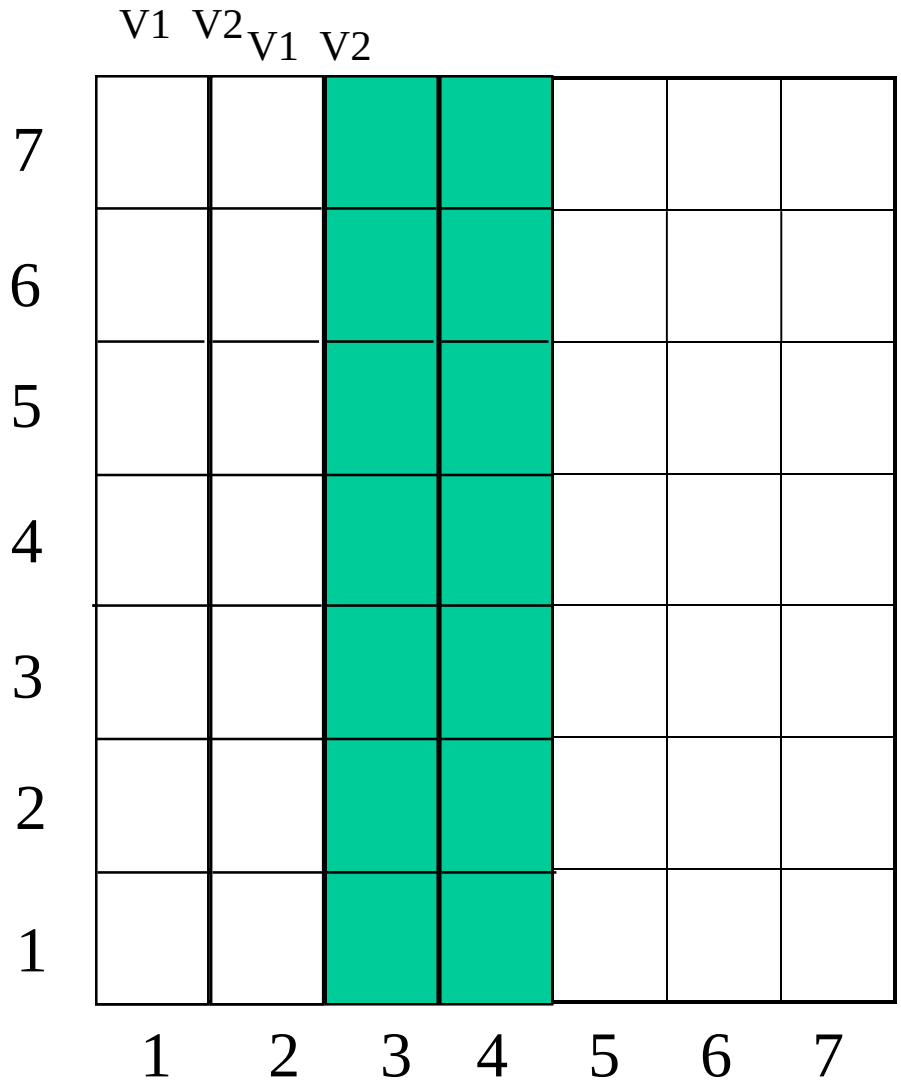
T[1,1]=0
para j=2 **hasta** J **hacer** T[1,j]=T[1,j-1]+d[(1,j),(1,j-1)] **fpara**
para i=2 **hasta** I **hacer**
  T[i,1]=T[i-1,1]+d[(i,1),(i-1,1)]
  para j=2 **hasta** J **hacer**

$$T[i,\,j]=\min\begin{cases}T[i-1,\,j]+d[(i,\,j),(i-1,\,j)],\\ T[i,\,j-1]+d[(i,\,j),(i,\,j-1)],\\ T[i-1,\,j-1]+d[(i,\,j),(i-1,\,j-1)]\end{cases}$$

  fpara
fpara
coste=T[I,J];
fin

Reducción de coste espacial

V1,V2: vector[1..J] de **R**;



Reducción del coste espacial

```
Función coste (I,J:N):R;
var V1,V2: vector[1..J] de R;

V1[1]=0
para j=2 hasta J hacer V1[j]=V1[j-1]+d[(1,j),(1,j-1)] fpara
para i=2 hasta I hacer
   V2[1]=V1[1]+d[(i,1),(i-1,1)]
   para j=2 hasta J hacer

                V2[j] = min{ V1[j] + d[(i,j),(i-1,j)],
                             V2[j-1] + d[(i,j),(i,j-1)],
                             V1[j-1] + d[(i,j),(i-1,j-1)] }
   fpara
   V1=V2;
fpara
coste=V2[J];
fin
```

Coste espacial O(J)

Observación: La reducción del coste espacial es incompatible con la recuperación de la secuencia óptima. Sólo puede obtenerse el coste óptimo.

Mochila sin fraccionamiento

Datos: Conjunto N de objetos con pesos (w_i) y valor (v_i)
Mochila de capacidad W.

Problema: Encontrar el subconjunto de objetos que caben en la mochila y que maximizan el valor total.

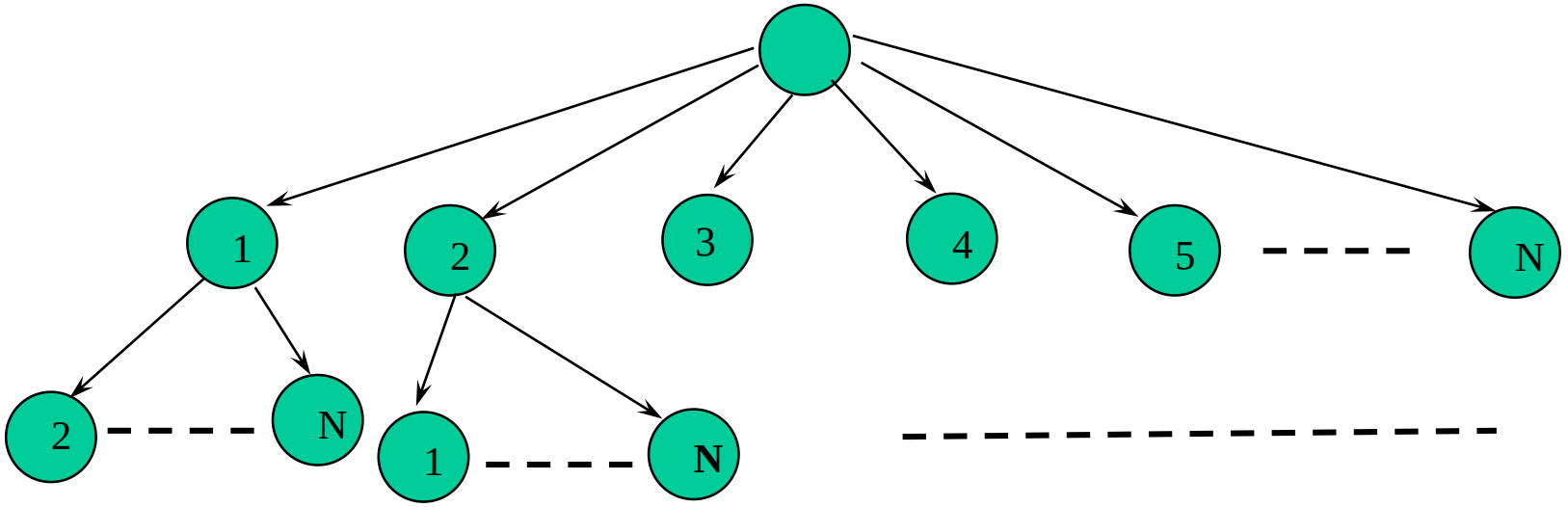
"La forma de representar la secuencia de decisiones, puede ser de gran ayuda para encontrarla".

Opción 1 (no aconsejable): Representamos la secuencia de decisiones como una secuencia de índices de los objetos que metemos en la mochila.

$$X = \left\{ (x_1, x_2, \ldots, x_n) \in [1..N]^* \;\middle|\; x_i \neq x_j, 1 \leq i \neq j \leq n; \sum_{1 \leq i \leq n} w_{x_i} \leq W \right\}$$

Ejemplos: (2,4,5,9) (1,3) (3,1,8,5.12)

Si usamos esta representación para ir obteniendo la secuencia óptima nos sale un árbol del espacio de soluciones como el siguiente.



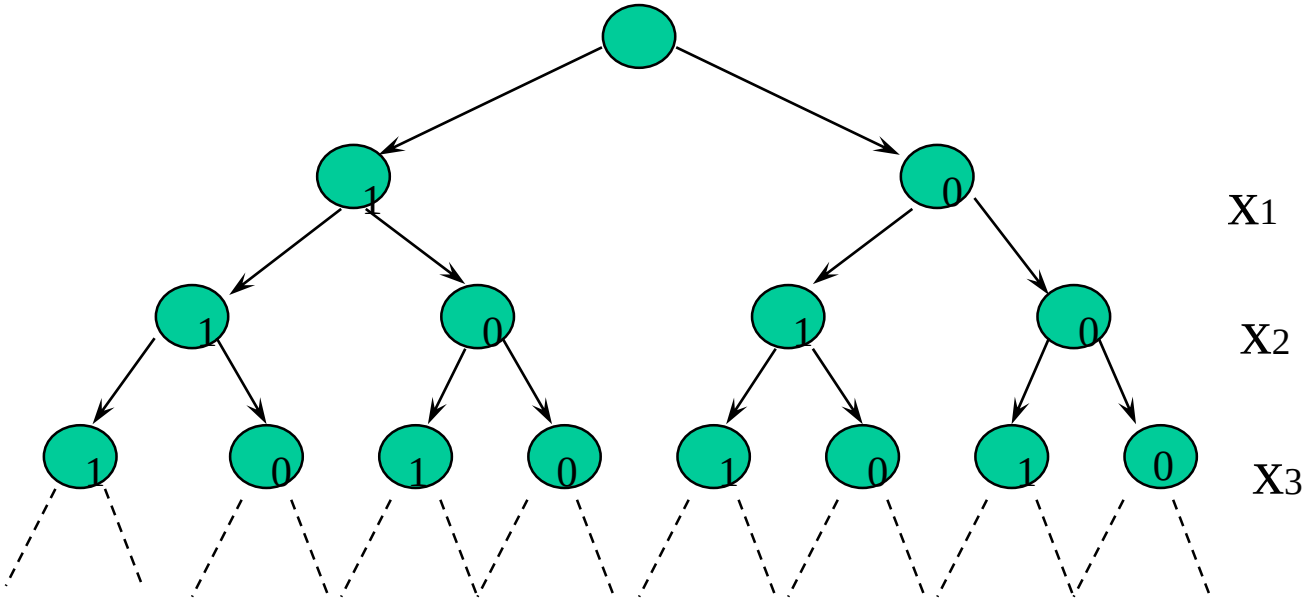
Ojo, el problema no es el formato final de la solución (siempre puede postprocesarse una solución para representarla así, el problema es basarse en esta representación para ir explorando el espacio de soluciones.

Opción 2: Representamos la secuencia de decisiones como una secuencia de N unos y ceros, que indican si el objeto correspondiente entra en la solución o no.

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \{0,1\}^N \left| \sum_{1 \le i \le N} x_i w_i \le W \right. \right\}$$

Ejemplos para N=5: (0,1,0,0,1) (1,0,0,1,1) (1,1,0,0,0)

En este caso el árbol que representa el espacio de decisiones es:



Importante: en la mayoría de problemas en que la solución es obtener un subconjunto de los datos de entrada, este tipo de exploración es la más adecuada.

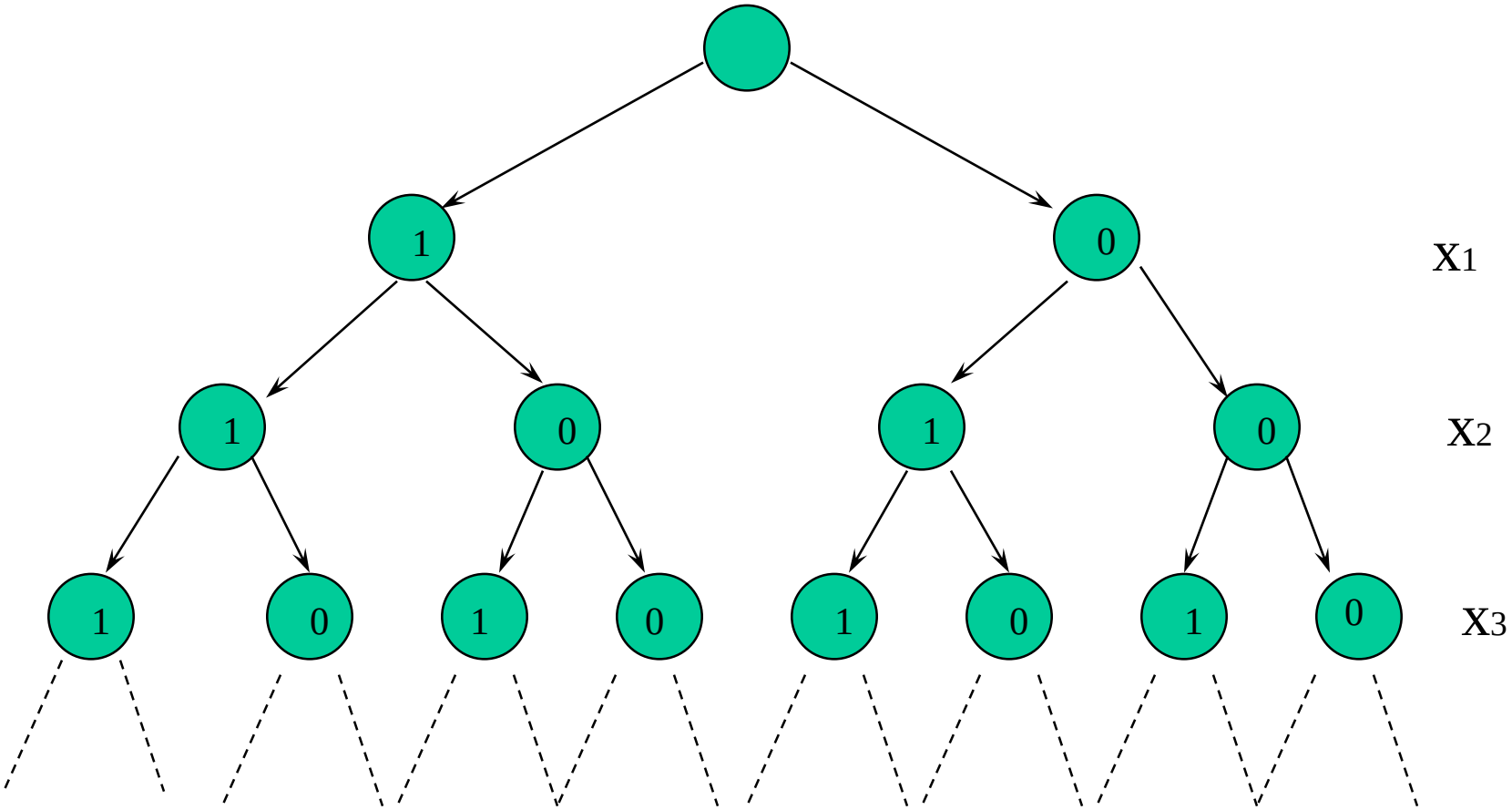
Utilizando esta representación, una solución factible es una secuencia ($x_1, x_2, \ldots, x_N$).

$$x_i = \begin{cases} 0 \\ 1 \end{cases}$$

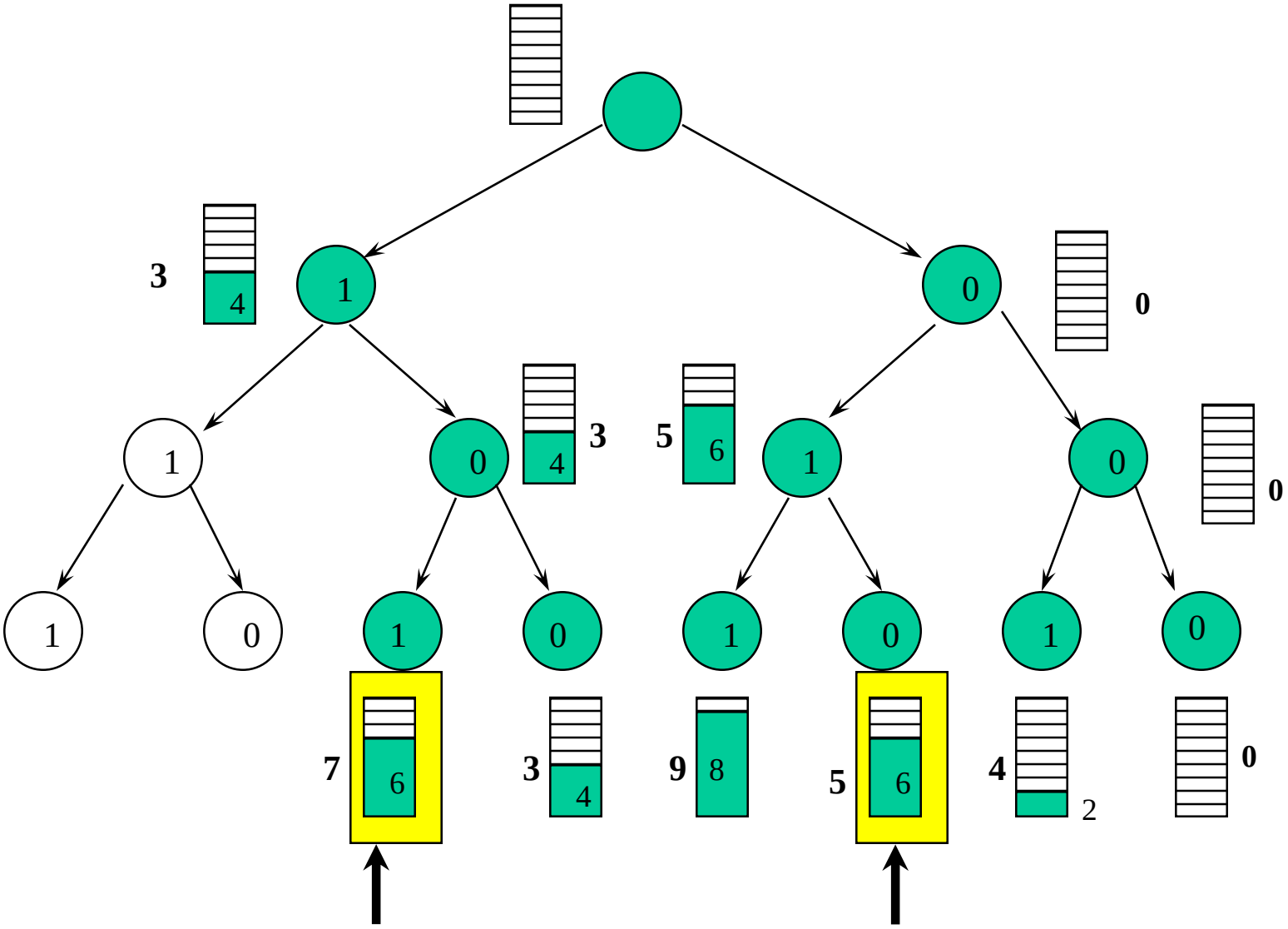
que cumple la restricción $$\sum_{i=1}^{n} w_i \cdot x_i \leq W$$

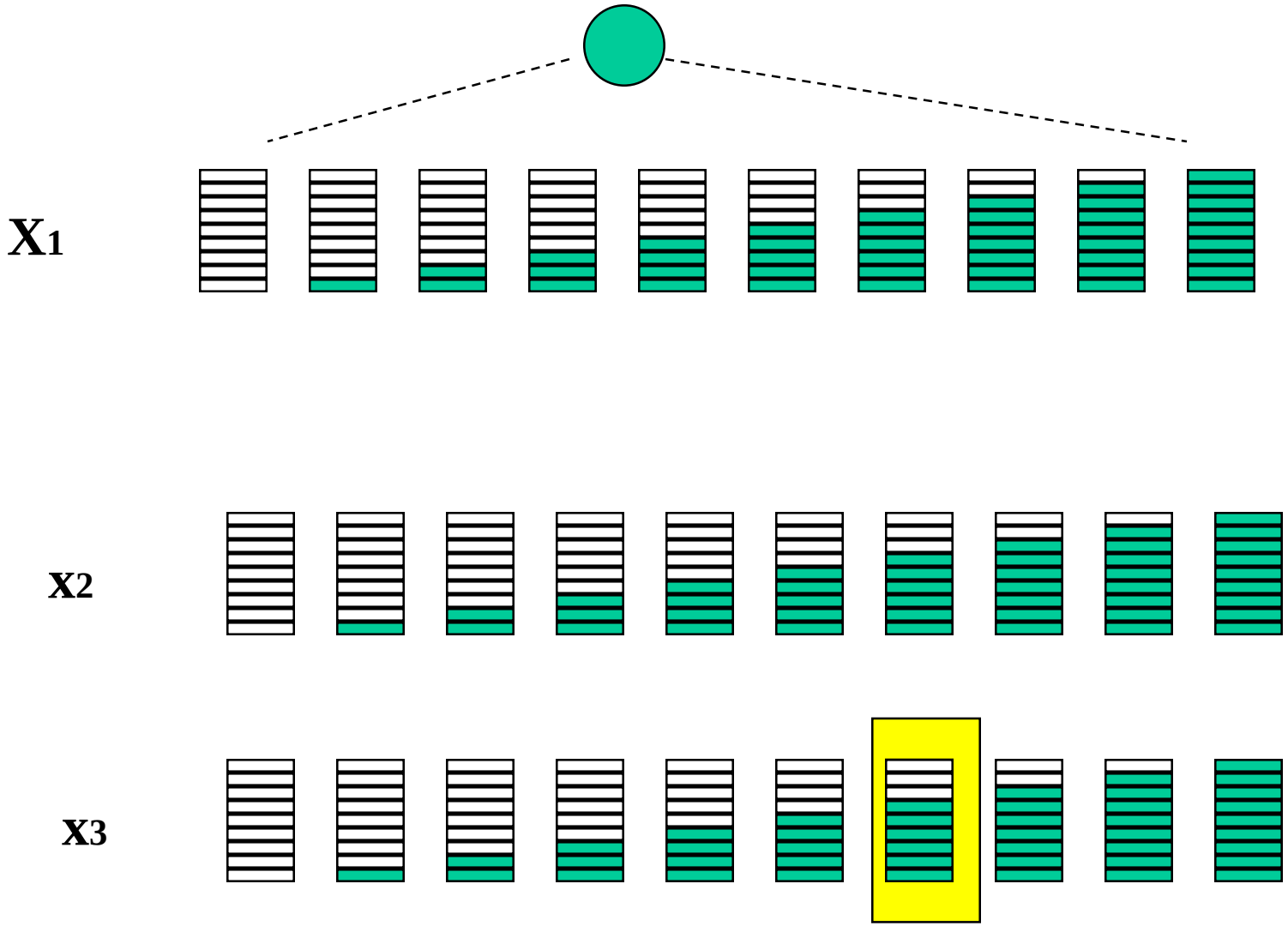
y la solución óptima es la que además maximiza la función objetivo $$\max_{\forall (x_1, x_2, \ldots, x_N)} \sum_{i=1}^{N} v_i \cdot x_i$$

Obtención de la solución recursiva

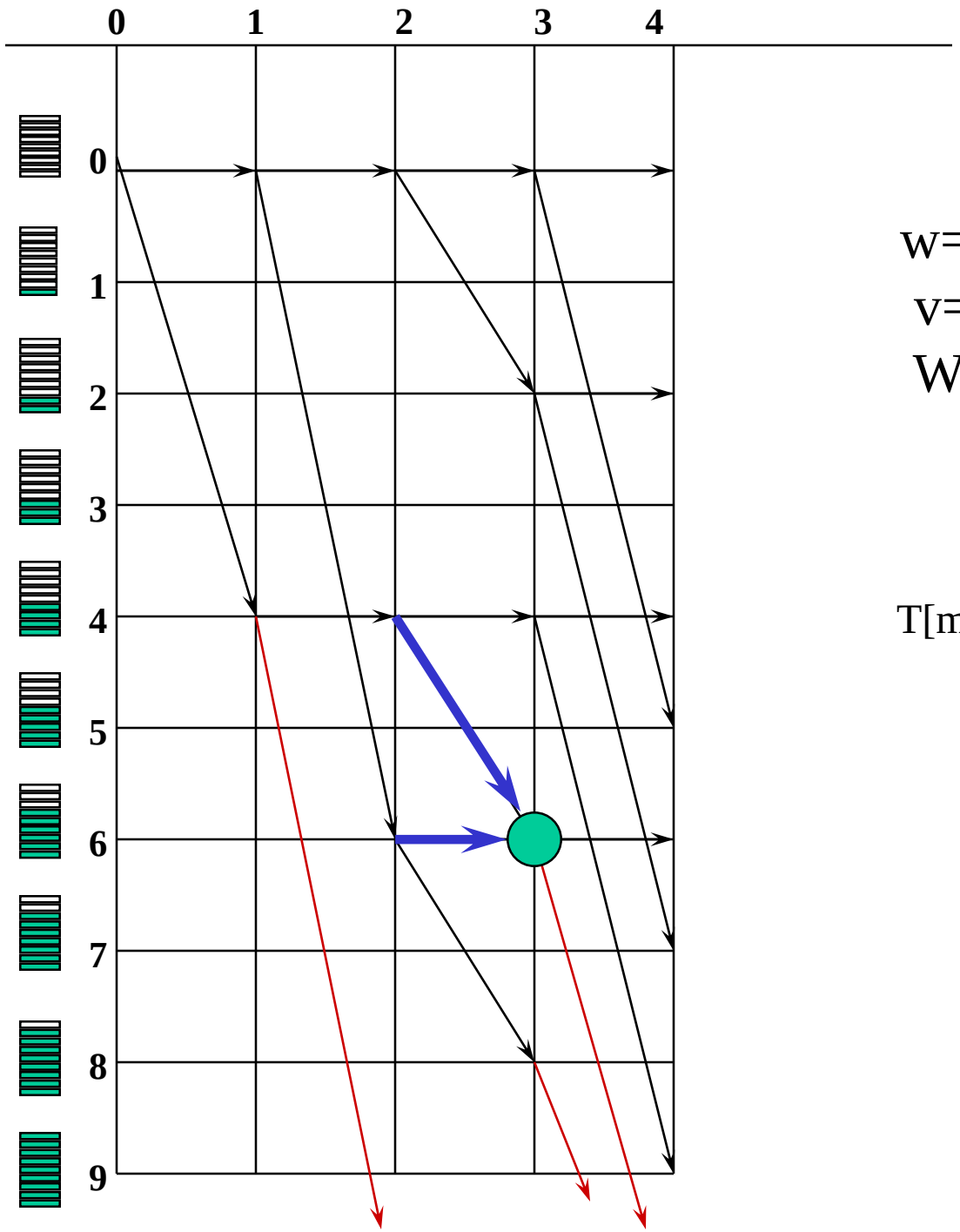


Ejemplo: w=(4,6,2,5) v=(3,5,4,6) W=9




X1
x2
x3

$$mochila(m,i) = \begin{cases} 0 & i = 0 \wedge m = 0 \\ -\infty & w < 0 \\ -\infty & i = 0 \wedge m > 0 \\ \max\begin{cases} mochila(m, i-1) + 0, \\ mochila(m - w_i, i-1) + v_i \end{cases} & i > 0 \wedge m > 0 \end{cases}$$



w=(4,6,2,5)
v=(3,5,4,6)
W=9

$T[m,i]= \min\{T[m,i-1]+0,$
$T[m-w_i,i-1]+v_i\}$

Función mochila (w,v:lista de **R**, N:**N**,W:**R**):**R**;

var T: matriz[0..M,0..N] de **R**;

T[0,0]=0;

para m=1 **hasta** W **hacer** T[m,0]=-∞;

para i=1 **hasta** N **hacer**
  para m=0 **hasta** M **hacer**
    si (m-wi)<0 **entonces** T[m,i]=T[m,i-1]
      sino si (T[m,i-1]+0)<(T[(m-wi),i-1]+vi) **entonces** T[m,i]= (T[(m-wi),i-1]+vi)
        sino T[m,i]= (T[m,i-1]+0)
      fsi
    fsi
  fpara
fpara

mochila=

fin $\max_{\forall m, 0 \le m \le W} \{T[m, N]\}$

Coste temporal O(N.W)
Coste espacial O(N.M)

Discusión: El problema de la mochila es NP-Completo (?????)