PROGRAMACIÓN DINÁMICA

1. Introducción
2. Esquema recursivo.
3. Esquema iterativo.
4. Ejemplos.

Bibliografía: (Horowitz,97) (Brassard,97) (Cormen,91)

1. Introducción

Problemas de optimización.

- Función objetivo.

- Restricciones

Ejemplos: Cambio de monedas, Mochila, Camino mínimo en un grafo.

Solución: Secuencia de decisiones

Algoritmos voraces: Se obtiene la secuencia de decisiones (la tupla solución) de modo iterativo, de modo que en cada paso se toma una decisión que SEGURO que pertenece a la solución óptima. Su coste es proporcional al número de decisiones más el coste de tomar cada decisión, es por tanto lineal con la talla de la tupla solución

Cambio de monedas/billetes: Dado un conjunto de monedas, ¿cómo devolver la cantidad C con el menor número de monedas/billetes?
Solución: Escoger secuencialmente la moneda de más valor que no supera la cantidad restante a devolver.
Ejemplo: para devolver 23 euros usamos 1 billete de 20, una moneda de 2 y una moneda de 1.

¿Qué ocurre cuando es imposible tomar una decisión “definitiva” con una visión local del problema?

Ejemplo: Supongamos que el conjunto de monedas es de 1 euro, 10 euros y 17 euros. Si queremos devolver la cantidad de 20 euros, el algoritmo anterior nos proporcionaría la solución (1 billete de 17, 1 moneda de 1, 1 moneda de 1, una moneda de 1), o sea 4 monedas. Hay una solución mejor que es devolver 2 billetes de 10 euros.

Ejemplo: Encontrar el mejor camino en un grafo entre dos vértices.

Si nos situamos en el vértice de partica y tenemos dos arcos, no podemos decidir que el de menor coste es mejor, ya que luego puede empeorar.

Forma de abordar esos problemas:
En principio habrá que explorar todo el espacio de decisiones (que es exponencial) y escoger la mejor.

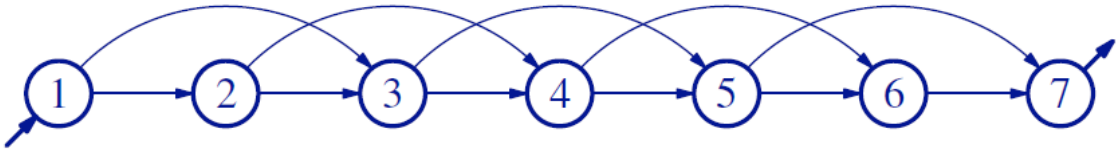
Programación dinámica: Son unos algoritmos que, aunque obtienen la solución óptima a partir de una búsqueda en el espacio de soluciones, consiguen reducir el número de cálculos a partir de la identificación de situaciones equivalentes, de modo que finalmente se consiguen costes polinómicos.

Vuelta atrás: Son unos algoritmos que consisten en una exploración sistemática del espacio de soluciones (son exponenciales)

Ramificación y poda: Consisten en ir explorando todo el espacio de soluciones, pero descartando secuencias incompletas de soluciones que se sabe que no pueden conducir a la solución óptima. En el peor caso son exponenciales, pero el comportamiento promedio puede ser razonable.

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.

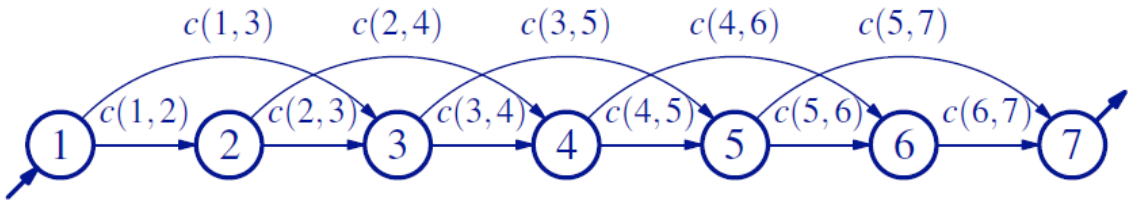


Primer problema: camino de menor coste

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.

El problema del trayecto más barato en el río Congo

El coste que presenta ir en canoa del embarcadero i al $i+1$, para $1 \leq i \leq E-1$, se representa con $c(i, i+1)$, y el coste que presenta ir del embarcadero i al $i+2$, para $1 \leq i \leq E-2$, se representa con $c(i, i+2)$. Naturalmente, los costes son cantidades no negativas. Queremos ir del primer embarcadero al último. ¿Cuál es el camino de menor coste? ¿Qué coste presenta dicho camino?



Formalización

Problema de optimización: determinar el valor mínimo de un función objetivo (coste) que asigna un valor real a cada una de las soluciones factibles (secuencias válidas de embarcaderos).

Solución factible (un trayecto válido entre los nodos inicial y final): secuencia de embarcaderos $(e_1, e_2, \ldots, e_n)$ tal que

- empieza en el primer embarcadero: $e_1 = 1$;
- finaliza en el último embarcadero: $e_n = E$; y
- desde un embarcadero e_i se puede ir al embarcadero $e_i + 1$ o al $e_i + 2$, es decir, $1 \leq e_i - e_{i-1} \leq 2$ para $1 < i \leq n$.

Conjunto de soluciones factibles:

$$X = \{(e_1, e_2, \ldots, e_n) \in [1..E]^+ \mid e_1 = 1; \quad e_n = E; \quad 1 \leq e_i - e_{i-1} \leq 2, 1 < i \leq n\}.$$

Función objetivo: el coste de un trayecto es la suma del coste de transitar directamente entre cada par de embarcaderos contiguos en el trayecto:

$$C((e_1, e_2, \ldots, e_n)) = \sum_{1<i\leq n} c(e_{i-1}, e_i).$$

Solución óptima: estamos interesados en encontrar el trayecto de menor coste (o uno cualquiera de ellos si hay más de uno con coste mínimo):

$$(\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_n) = \arg \min_{(e_1,e_2,\ldots,e_n)\in X} C((e_1, e_2, \ldots, e_n))$$

Además, deseamos conocer el coste de dicho trayecto:

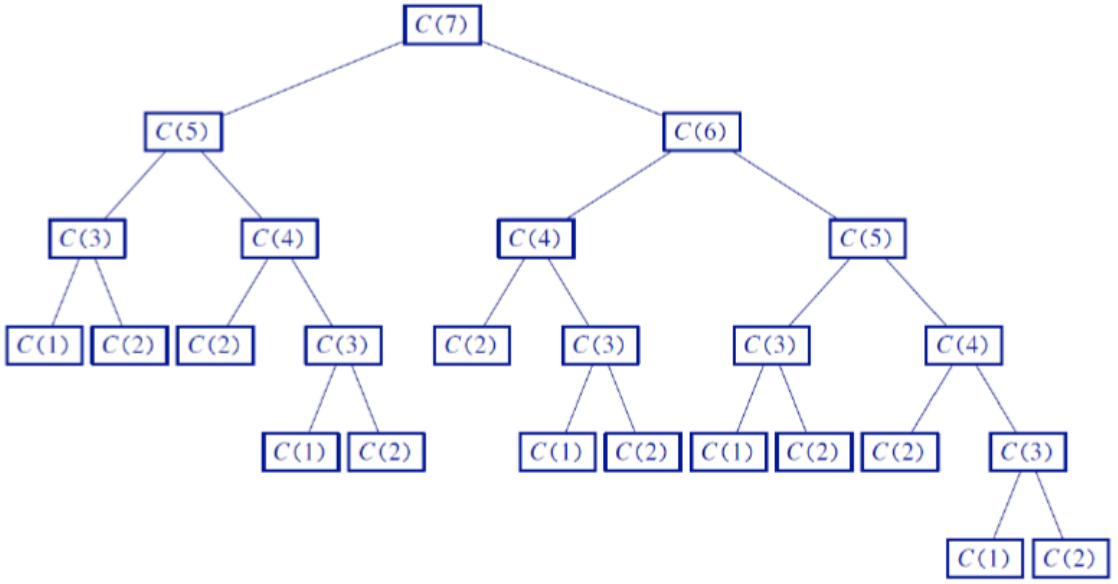
$$\min_{(e_1,e_2,\ldots,e_n)\in X} C((e_1, e_2, \ldots, e_n))$$

Nos podríamos plantear, entonces, una **exploración exhaustiva** de todo el conjunto de soluciones factibles:

$$X = \{(e_1, e_2, \ldots, e_n) \in [1..E]^+ \mid e_1 = 1; \quad e_n = E; \quad 1 \leq e_i - e_{i-1} \leq 2, 1 < i \leq n\}.$$

Es decir, enumerar todos los trayectos válidos, calcular el precio de cada uno de ellos y quedarnos con el más barato. El coste temporal de este algoritmo será $O(2^E)$.

¿Podemos hacerlo mejor?



Mejor camino que llega hasta E.

Principio de optimalidad: Cualquier subsecuencia de una secuencia óptima también es óptima para el subproblema que resuelve.

$$C(E) = \min_{j \in \{E-1, E-2\}} (C(j) + c(j, E)),$$

$$C(E) = \min \left\{ \begin{array}{l} C(E-2) + c(E-2, E), \\ C(E-1) + c(E-1, E) \end{array} \right\}.$$

El razonamiento seguido es igualmente válido si queremos calcular el coste del camino más barato entre el primer embarcadero y un embarcadero i cualquiera tal que $2 < i \leq E$, así que esta ecuación también es válida:

$$C(i) = \min \left\{ \begin{array}{l} C(i-2) + c(i-2,i), \\ C(i-1) + c(i-1,i) \end{array} \right\}.$$

Embarcaderos 1 y 2 son especiales: $C(1) = 0$ y $C(2) = C(1) + c(1,2) = c(1,2)$.

Podemos agrupar las diferentes ecuaciones alcanzadas en ésta:

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

donde, recordemos, $C(i)$ es coste del camino óptimo (el más barato) que parte del embarcadero 1 y llega al embarcadero i-ésimo. El coste del trayecto con que un turista viaja del primer al último embarcadero de la forma más económica es $C(E)$.

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:

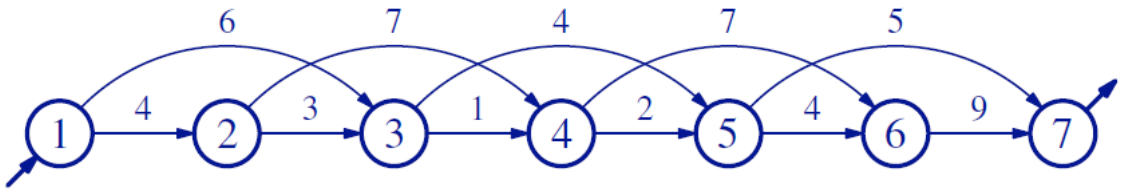
congo.py

```
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)
```

El valor de $C(E)$ se obtiene efectuando la llamada *recursive_cheapest_price*(E, c)

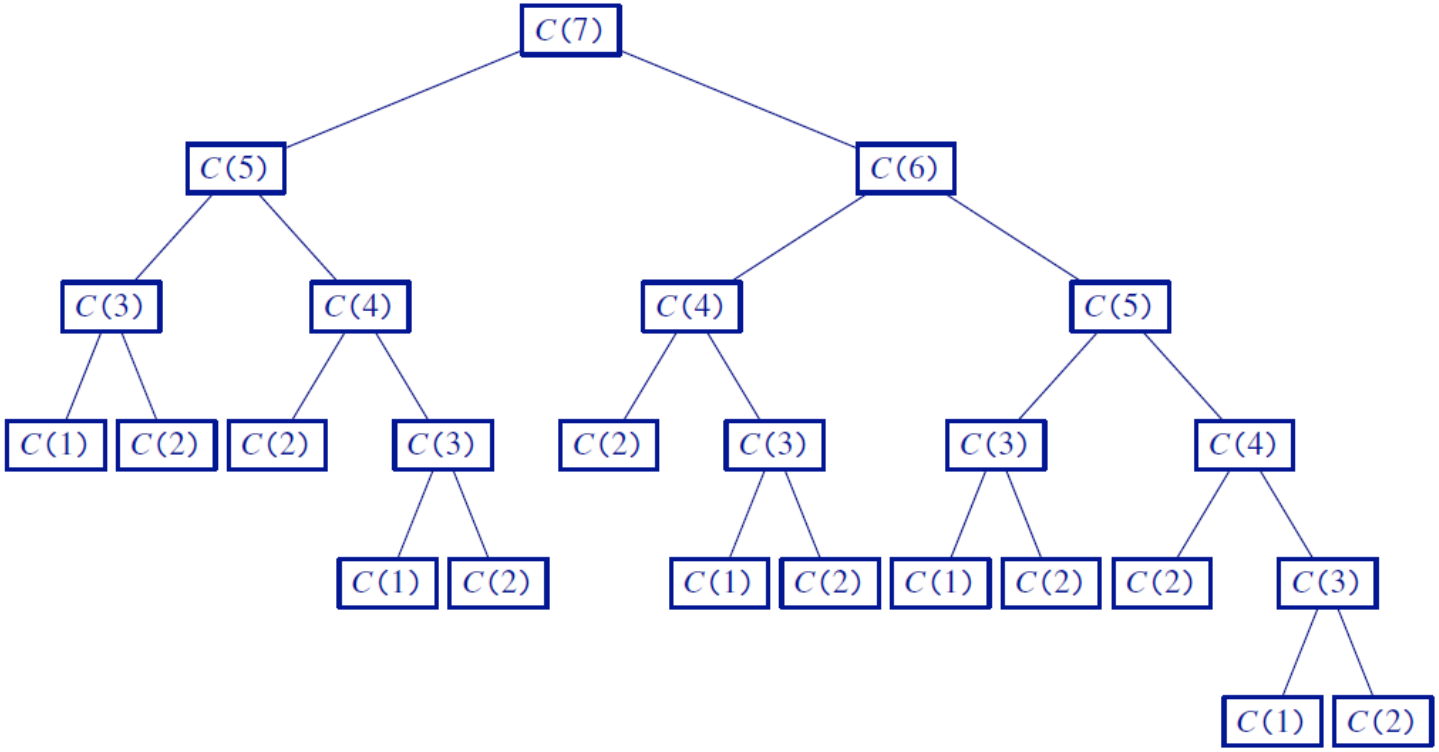
```
from congo import recursive_cheapest_price

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 del embarcadero 1 al 7:', recursive_cheapest_price(7, c)
```

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

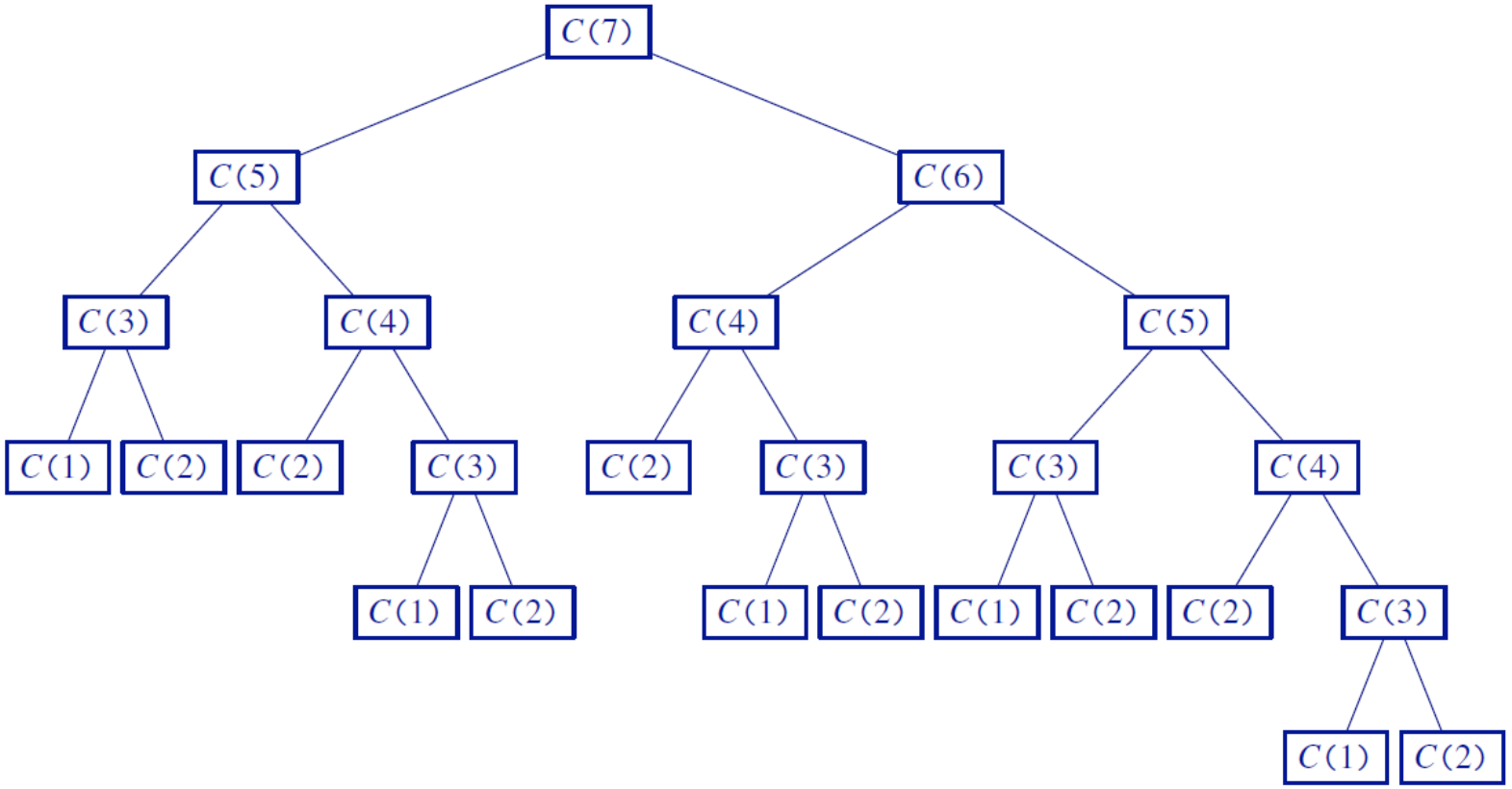


congo.py

```
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)
```

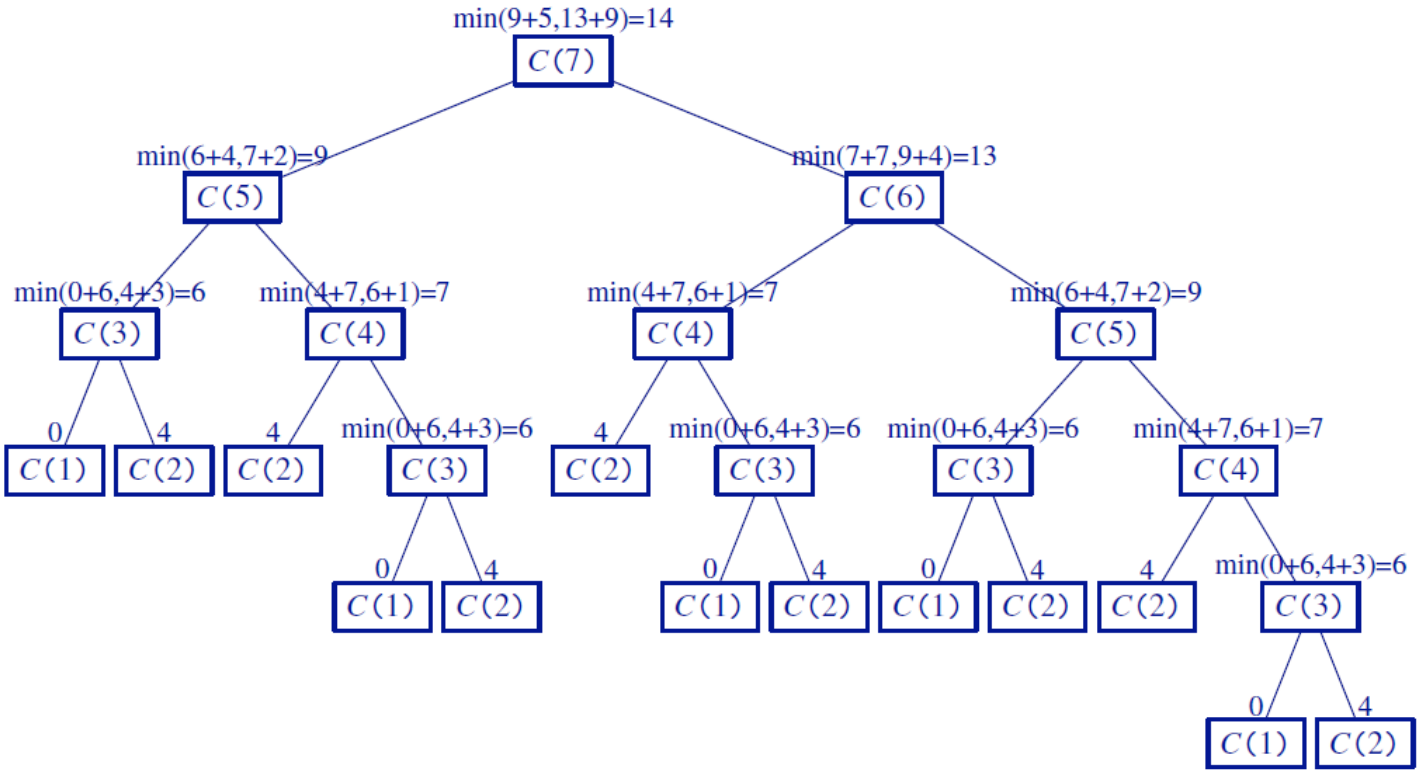


Árbol de llamadas recursivas efectuadas al calcular $C(7)$. Cada rama del árbol corresponde a un trayecto diferente. La rama de más a la izquierda, por ejemplo, corresponde al trayecto $(1,3,5,7)$, y la de más a la derecha, al camino $(1,2,3,4,5,6,7)$. El árbol permite interpretar que el algoritmo explora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



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)
```



¿Coste?

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)
```

Estudiar el número de llamadas a la función C cuando calculamos $C(E)$:

$$N(E) = \begin{cases} 1, & \text{si } i \leq 2; \\ N(E-2)+N(E-1)+1, & \text{si } i > 2. \end{cases}$$

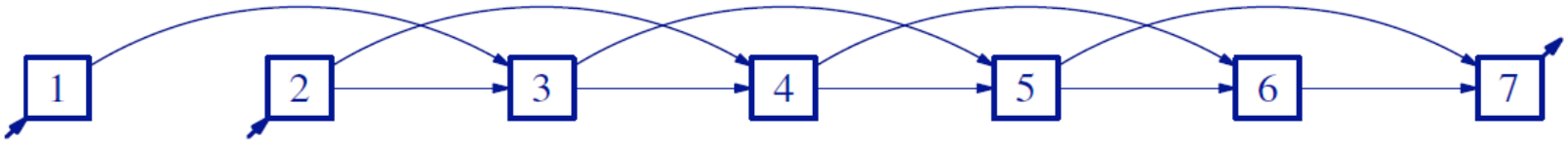
El número de llamadas crece exponencialmente con E.

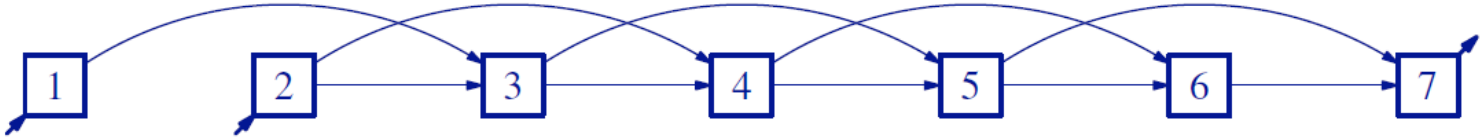
Sin embargo, no deberían ser necesarias más de E llamadas recursivas (no hay más que E costes diferentes que calcular, uno por embarcadero).

Problema: llamadas repetidas.

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).





Si efectuamos un recorrido de los vértices del grafo de dependencias en un orden topológico, al calcular cada valor de $R[i]$ tendremos ya calculados los valores que necesitamos, que son $R[i-1]$ y $R[i-2]$. Un recorrido por valor creciente del índice es un orden topológico en el grafo de dependencias:

- Calculamos primero $R[1]$ como 0.
- Calculamos entonces $R[2]$ como $c[1,2]$.
- Calculamos $R[3]$ como el mínimo entre $R[2]$ más $c[2,3]$ y $R[1]$ más $c[1,3]$, pues ya conocemos el valor de $R[2]$ y de $R[1]$.
- Calculamos $R[4]$ como el mínimo entre $R[3]$ más $c[3,4]$ y $R[2]$ más $c[2,4]$, pues ya conocemos el valor de $R[3]$ y de $R[2]$.
- ...

congo.py

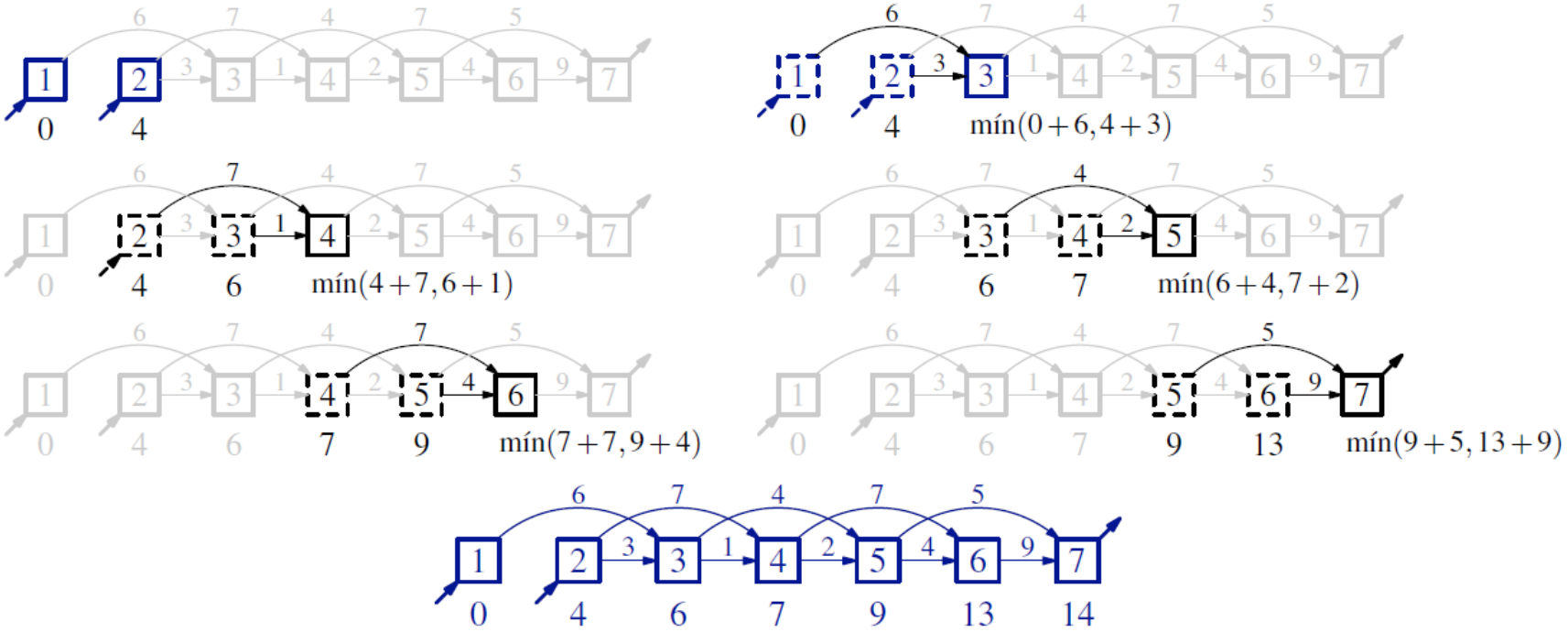
```
def iterative_cheapest_price1(E, c):
    C = {}
    C[1] = 0
    C[2] = c[1,2]
    for i in xrange(3, E+1):
        C[i] = min( C[i-1] + c[i-1,i], C[i-2] + c[i-2,i] )
    return C[E]
```

```
from congo import iterative_cheapest_price1

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_price1(7, c)
```

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

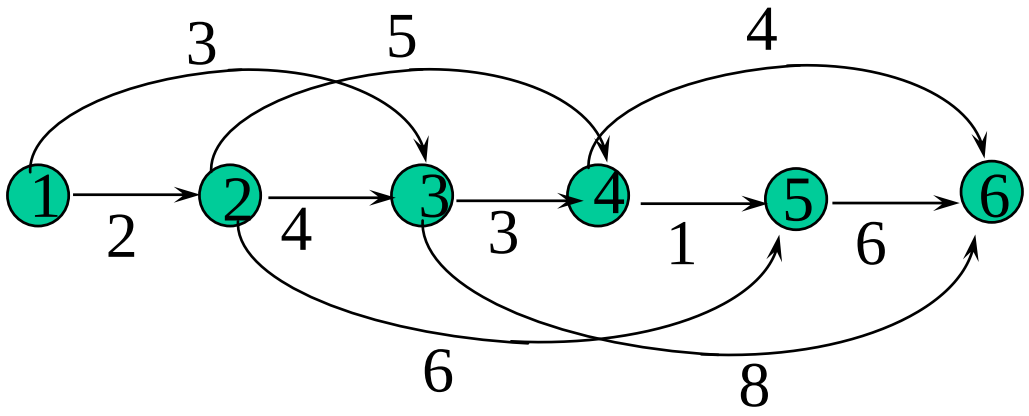
congo.py

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



Solución1: **Recursividad hacia atrás**

¿Cuál es la mejor forma de llegar ***hasta*** *i?*



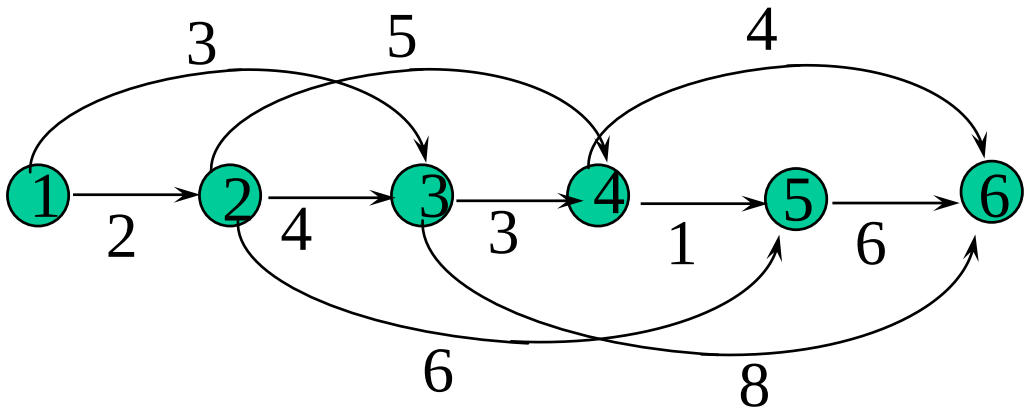
$$coste(i) = \begin{cases} 0 & i = 1 \\ \min\limits_{\forall j \in pred(i)} \{coste(j) + c(j,i)\} & 1 < i \leq F \end{cases}$$

F=6

Solución: Coste(6)

Solución2: Recursividad hacia adelante

¿Cuál es la mejor forma de llegar ***desde*** *i?*



$$\text{coste}(i) = \begin{cases} 0 & i = F \\ \min\limits_{\forall j \in suc(i)} \{\text{coste}(j) + c(i,j)\} & 1 \le i < F \end{cases}$$

F=6

Solución: Coste(1)

Esquema recursivo.

Sea un problema cuyo conjunto de soluciones es X, y la función objetivo $F:X \rightarrow \mathbf{R}^+$

Hay que encontrar la secuencia $X^* \in X$, tal que F(X*) sea óptima.

$$X^* = \arg \max_{X' \in X} F(X')$$

Esquema recursivo:

esquema PDR (A:λ): **R**

si contorno(A) entonces PDR=trivial(A)

sino $\text{PDR} = \underset{x \in \text{alternativas(A)}}{opt} \{\text{calculo}(\text{PDR}(x), \text{coste}(A,x))\}$

fsi

fPDR

Esquema iterativo

Esquema PDI(A: λ): **R**

/* Sean p_i los "nombres" nombres de los subproblemas ordenados $p_i \leq p_{i+1}$*/

/* Sea n el número de subproblemas */

var T: Tabla[**p**] de **R** ; /*Tabla para almacenar resultados de subproblemas */

p= Problema_inicial(A);

Mientras no_final (p) hacer /* mientras no se han resuelto todos los subproblemas*/

    resul=Resolver(p)

    Almacenar (T[p],resul);

    p=siguiente_subproblema(A,p)

fmientras

Devuelve Extraer_result(T)

fPDI

Esquema iterativo (más concreto)

Esquema PDI(A: λ): **R**

/* Sean p_i los "nombres" nombres de los subproblemas ordenados $p_i \leq p_{i+1}$ */

/* Sea n el número de subproblemas */

var T: Tabla[**p**] de **R** ; i:**N**; fvar

para i=1 hasta n hacer

si $p_i \in$ contorno(A) entonces $T[p_i]$=trivial(p_i)

sino

$$T[p_i] = \underset{x \in \text{alternativa}(p_i)}{opt} \{\text{calculo}(T[p_i], \text{coste}(x, p_i))\}$$

fsi

fpara

PDI=$T[p_n]$;

fPDI