Estrategias algorítmicas

Tema 3(II)

Algorítmica y Modelos de Computación

1. Introducción.
2. Algoritmos divide y vencerás.
3. Algoritmos voraces.
4. Programación dinámica.
5. Algoritmos vuelta atrás (Bactracking).
6. Ramificación y poda.

4. Programación dinámica.

4. Programación dinámica. Introducción.

- La base de la **programación dinámica** es el **razonamiento inductivo**: cómo resolver un problema combinando soluciones para problemas más pequeños.
- La idea es la misma que en **divide y vencerás** pero aplicando una estrategia distinta.
- **Similitud:**
  - Descomposición recursiva del problema.
  - Se obtiene aplicando un razonamiento inductivo.
- **Diferencia:**
  - Divide y vencerás: aplicar directamente la fórmula recursiva (programa recursivo).
  - Programación dinámica: resolver primero los problemas más pequeños, guardando los resultados en una tabla (programa iterativo).

4. Programación dinámica. Introducción.

- **Métodos ascendentes y descendentes**

- **Métodos descendentes (divide y vencerás)**
  - Empezar con el problema original y descomponer recursivamente en problemas de menor tamaño.
  - Partiendo del problema grande, descendemos hacia problemas más sencillos.
- **Métodos ascendentes (programación dinámica)**
  - Resolvemos primero los problemas pequeños (guardando las soluciones en una tabla). Después los vamos combinando para resolver los problemas más grandes.
  - Partiendo de los problemas pequeños avanzamos hacia los más grandes.

4. Programación dinámica. Introducción.

- **Ejemplo. Cálculo de los números de Fibonacci.**

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

  - **Con divide y vencerás.**

    operación Fibonacci (n: entero): entero

    si n≤2 **entonces devolver** 1

    sino devolver Fibonacci(n-1) + Fibonacci(n-2)

  - **Con programación dinámica.**

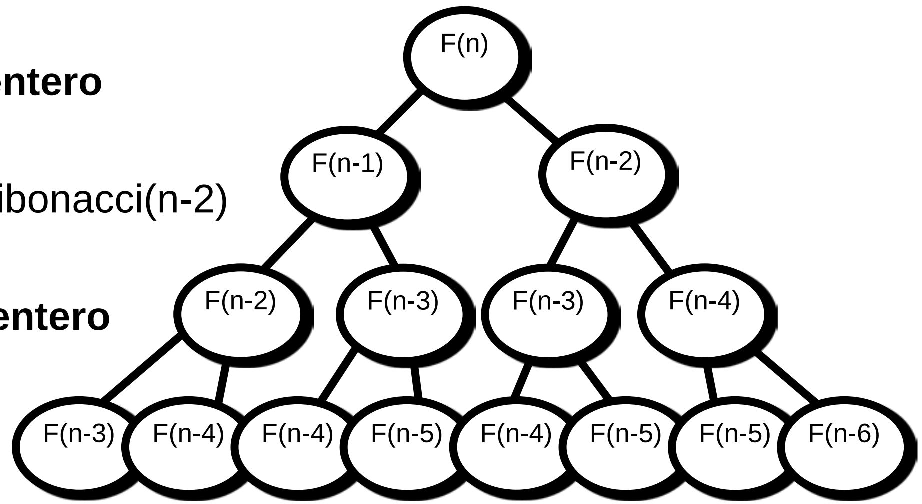
    operación Fibonacci (n: entero): entero

    T[1]:= 1; T[2]:= 1

    para i:= 3 **hasta** n **hacer**

    T[i]:= T[i-1] + T[i-2]

    devolver T[n]



- Los dos usan la misma fórmula recursiva, aunque de forma distinta. ¿Cuál es más eficiente?
  - **Con programación dinámica: $\Theta(n)$**
  - **Con divide y vencerás:**
    - **Problema:** Muchos cálculos están repetidos.
    - El tiempo de ejecución es **exponencial: $\Theta(1{,}62^n)$**

4. Programación dinámica. Método general. Funcionamiento.

- La técnica de **Programación dinámica** fue inventada como un **método general de optimización de procesos de decisión por etapas**.
- Es adecuada para resolver problemas cuya solución puede caracterizarse recursivamente (como con la técnica divide y vencerás) y en la que los subproblemas que aparecen en la recursión se solapan de algún modo, lo que significaría una repetición de cálculos inaceptable si se programara la solución recursiva de manera directa.
- La aplicación de la técnica de programación dinámica evita la repetición de cálculos mediante la memorización de la solución de cada subproblema en una tabla, de manera que no haya que calcularlo más de una vez.
- **La aplicación de la técnica** de programación dinámica tiene **dos fases** fundamentales:
  1. Definir **recursivamente la solución** del problema.
  2. Definir la **estructura de datos** para memorizar las soluciones de los subproblemas y escribir el algoritmo que va calculando los valores de esa estructura de datos siguiendo la caracterización de la solución definida en la fase 1, pero sin repetir el cálculo de soluciones de subproblemas.

- **Ejemplo. Algoritmo de Floyd**, para calcular los caminos mínimos entre cualquier par de nodos de un grafo.
  - **Razonamiento inductivo:** para calcular los caminos mínimos pudiendo pasar por los **k** primeros nodos usamos los caminos mínimos pasando por los **k-1** primeros.
  - $\mathbf{D_k(i, j)}$**:** camino mínimo de **i** a **j** pudiendo pasar por los nodos 1, 2, …, **k**.

$$D_k(i, j) = \begin{cases} C[i, j] & \text{Si } k=0 \\ \min(D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)) & \text{Si } k>0 \end{cases}$$

$D_n(i, j) \rightarrow$ caminos mínimos finales

- Aplicación de la fórmula:
  - Empezar por el problema pequeño: **k** = 0
  - Avanzar hacia problemas más grandes: **k** = 1, 2, 3, ...

4. Programación dinámica. Método general.

- Garantía para que un algoritmo de programación dinámica obtenga la solución correcta ⟹ Una descomposición es correcta si cumple el

> **Principio de optimalidad de Bellman**:
>
> La solución óptima de un problema se obtiene combinando soluciones óptimas de subproblemas.

- O bien: **cualquier subsecuencia de una secuencia óptima debe ser, a su vez, una secuencia óptima.**
  - **Ejemplo.** Si el camino mínimo de **A** a **B** pasa por **C**, entonces los trozos de camino de **A** a **C**, y de **C** a **B** deben ser también mínimos.

⟹ el principio no siempre es aplicable.

  - **Contraejemplo.** Si el camino simple más largo de **A** a **B** pasa por **C**, los trozos de **A** a **C** y de **C** a **B** no tienen por qué ser soluciones óptimas.

4. Programación dinámica. Método general.

- Supongamos que un problema se resuelve tras tomar un secuencia d_1, d_2, …, d_n de decisiones.

  - Si hay d opciones posibles para cada una de las decisiones, una técnica de fuerza bruta exploraría un total de d^n secuencias posibles de decisiones (explosión combinatoria).

  - La técnica de programación dinámica evita explorar todas las secuencias posibles por medio de la resolución de subproblemas de tamaño creciente y almacenamiento en una tabla de las soluciones óptimas de esos subproblemas para facilitar la solución de los problemas más grandes.

4. Programación dinámica. Método general. **Pasos.**

- **Pasos para aplicar programación dinámica:**

  1. Obtener una **descomposición recurrente** del problema:
     - Ecuación recurrente.
     - Casos base.
  2. Definir la **estrategia** de aplicación de la fórmula:
     - **Tablas** utilizadas por el algoritmo.
     - Orden y forma de rellenarlas.
  3. Especificar cómo se **recompone la solución** final a partir de los valores de las tablas.

  - **Punto clave:** obtener la descomposición recurrente (Requiere “creatividad”)

4. Programación dinámica. Método general.

- **Cuestiones a resolver** en el razonamiento inductivo:
  - ¿Cómo reducir un problema a subproblemas más simples?
  - ¿Qué parámetros determinan el **tamaño del problema** (es decir, cuándo el problema es “más simple”)?
- **Idea:** ver lo que ocurre al tomar una decisión concreta

  ⇒ interpretar el problema como un **proceso de toma de decisiones**.
- **Ejemplos.**
  - **Floyd.** Decisiones: Pasar o no pasar por un nodo intermedio.
  - **Mochila 0-1.** Decisiones: coger o no coger un objeto dado.

4. Programación dinámica. Análisis de tiempos de ejecución.

- La programación dinámica se basa en el uso de **tablas** donde se almacenan los resultados parciales.

- En general, el **tiempo** será de la forma:

Tamaño de la tabla * Tiempo de rellenar cada elemento de la tabla

- Un aspecto **importante** es la memoria, puede llegar a ocupar la tabla.

- Además, algunos de estos cálculos pueden ser innecesarios.

- Variante del problema de la mochila: la "**Mochila 0-1**". Como el problema de la mochila, pero los objetos no se pueden partir (se cogen enteros o nada)
  - x_i sólo toma valores **0** ó **1**, indicando que el objeto se deja fuera o se mete en la mochila.
  - Los pesos, p_i, y la capacidad son números naturales.
  - Los beneficios, b_i, son reales no negativos.
- **Datos del problema:**
  - **n**: número de objetos disponibles.
  - **M**: capacidad de la mochila.
  - **p = (p_1, p_2, ..., p_n)** pesos de los objetos.
  - **b = (b_1, b_2, ..., b_n)** beneficios de los objetos.
- **Objetivo:** llenar la mochila, de capacidad ***M***, de manera que se **maximice** el beneficio.
- **Representación de la solución:** Una solución será de la forma **S = (x_1, x_2, ..., x_n)**, $x_i \in \{1,0\}$, siendo cada $\mathbf{x_i}$ la elección o no del objeto **i**.
- **Formulación matemática:**
  - **Maximizar** $\sum_{i=1}^{n} x_i b_i$ sujeto a la restricción $\sum_{i=1}^{n} x_i p_i \le M$

con $x_i \in \{1,0\}$; M, $p_i \in \mathbf{N}$ y $b_i > 0 \in \mathbf{R}$

- **Ejemplo:** n = 3; M = 15; b = (38, 40, 24); p = (9, 6, 5)
- **Estrategia voraz**, solución que devuelve el algoritmo voraz adaptado al caso 0-1 (o se coge un objeto entero o no)
  - Tomar siempre el objeto que proporcione mayor beneficio por unidad de peso: b/p=(4.22,6.66,4.8). Se obtiene la solución:
    - (x_1,x_2,x_3)=(0,1,1), con beneficio 64
  - Sin embargo, la solución óptima es:
    - (x_1,x_2,x_3)=(1,1,0), con beneficio 78
- Por tanto, la estrategia voraz no calcula la solución óptima del problema de la mochila 0-1.
- El problema es un NP-completo clásico.
- Técnica de **programación dinámica**
  - Permite resolver el problema mediante una secuencia de decisiones (como el esquema voraz)
  - A diferencia del esquema voraz, se producen varias secuencias de decisiones y solamente al final se sabe cuál es la mejor de ellas.
  - Está basada en el **principio de optimalidad de Bellman**.

- Aplicando la programación dinámica al problema:

Paso 1. Cómo obtener la descomposición recurrente.

- Interpretar el problema como un **proceso de toma de decisiones:** coger o no coger cada objeto.
- Después de tomar una decisión particular sobre un objeto, nos queda un problema de menor tamaño (con un objeto menos).
- ¿Coger o no coger un objeto?

Coger n + Resolver(1..n-1)

Resolver(1..n)

No coger n + Resolver(1..n-1)

- ¿Coger o no coger un objeto **k**?
  - → **Si se coge:** tenemos el beneficio $\mathbf{b_k}$, pero en la mochila queda menos espacio, $\mathbf{p_k}$.
  - → **Si no se coge:** tenemos el mismo problema pero con un objeto menos por decidir.
- ¿**Qué varía en los subproblemas**?
  - Número de objetos por decidir.
  - Peso disponible en la mochila.

Paso 1. (**Continuación**. **Ecuación recurrente y caso base.)**

- **Ecuación del problema. Mochila (k, m: entero): entero.**
  Problema de la mochila 0/1, **considerando sólo los k primeros objetos** (de los **n** originales) con capacidad de mochila **m**. **Devuelve el valor de beneficio total**.

- **Definición de Mochila(k, m: entero): entero**
  - **Si no se coge el objeto k:**
    Mochila(k, m) = Mochila(k - 1, m)
  - **Si se coge:**
    Mochila(k, m) = b_k + Mochila(k - 1, m - p_k)
  - **Valor óptimo:** el que dé mayor beneficio:

    Mochila(k, m) = **max** { Mochila(k - 1, m), b_k + Mochila(k - 1, m - p_k) }

- **Casos base:**
  - Si **m**=0, no se pueden incluir objetos: **Mochila(k, 0) = 0**
  - Si **k**=0, tampoco se pueden incluir: **Mochila(0, m) = 0**
  - Si **m** o **k** son negativos, el problema es irresoluble: **Mochila(k, m) = -∞**

Paso 1. (Continuación. Ecuación recurrente y caso base.)

- **Resultado.** La siguiente ecuación obtiene la solución óptima del problema:

$$\text{Mochila}(k, m) = \begin{cases} 0 & \text{Si } \mathbf{k}=0 \text{ ó } \mathbf{m}=0 \\ -\infty & \text{Si } \mathbf{k}<0 \text{ ó } \mathbf{m}<0 \\ \max\{\text{Mochila}(k-1, m),\ b_k + \text{Mochila}(k-1, m-p_k)\} & \end{cases}$$

- ¿Cómo aplicarla de forma ascendente?
  - Usar una **tabla** para guardar resultados de los subproblemas
  - Rellenar la tabla: empezando por los casos base, avanzar a tamaños mayores.

4. Programación dinámica. **Problema de la mochila 0-1. Algoritmo recursivo.**

- La recurrencia permite escribir un algoritmo recursivo de forma inmediata.

```
/* La función mochila1 devuelve el beneficio total */
función Mochila1(p,b:[1..n] de entero; M: natural) devuelve natural;
     devuelve V(n,M)
ffunciónMochila1
/* El algoritmo recursivo Vrec rellena un valor de la tabla y lo devuelve */
algoritmo V (i,j: natural) devuelve natural;  /* devuelve el valor de V[i,j] */
   /* Inicializar los casos base */
  si j=0 entonces devuelve 0 sino
     si j<p[i] entonces devuelve V(i-1,j) sino
          si V(i-1,j)≥ V(i-1,j-p[i])+b[i] entonces  devuelve V(i-1,j) sino
              devuelve V(i-1,j-p[i])+b[i]
        fsi
     fsi
  fsi
falgoritmoV
```

- **Problema**: **ineficiencia**
- Un problema de tamaño *n* se reduce a dos subproblemas de tamaño (*n*-1).
- Cada uno de los dos subproblemas se reduce a otros dos…

  ⇒ Por tanto, se obtiene un algoritmo exponencial.
- Sin embargo, el número total de subproblemas a resolver no es tan grande. La función tiene dos parámetros: el primero puede tomar ***n*** valores distintos y el segundo, *M* valores.

  ⇒ sólo hay ***nM* problemas diferentes**
- Por tanto, la solución recursiva está generando y resolviendo el mismo problema muchas veces.

- Para evitar la repetición de cálculos, las soluciones de los subproblemas se deben almacenan en una tabla.

- Matriz V: *nxM* cuyo elemento (i,j) almacena el beneficio (o ganancia total) de una solución óptima de *Mochila*(i,j)

- Para el ejemplo anterior: n=3; M=15; (b_1,b_2,b_3)=(38,40,24); (p_1,p_2,p_3)=(9,6,5)

V	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_1=9	0	0	0	0	0	0	0	0	0	38	38	38	38	38	38	38
p_2=6	0	0	0	0	0	0	40	40	40	40	40	40	40	40	40	78
p_3=5	0	0	0	0	0	24	40	40	40	40	40	64	64	64	64	78

$$(V[i, j] := \max (V[i-1, j], V[i-1, j-p_i] + b_i))$$

Solución óptima (x_1,x_2,x_3)=(1,1,0), con beneficio 78

Paso 2. Definición de las tablas y cómo rellenarlas

2.1. Dimensiones y tamaño de la tabla

- Definimos la tabla **V**, para guardar los resultados de los subproblemas:

  V[i, j] = Mochila(i, j)

- La solución del problema original es **Mochila(n, M)**.
  - Por lo tanto, la tabla debe ser: **V: array** [0..n, 0..M] **de** entero
  - $\Rightarrow$ **tamaño** de la tabla **V: (n+1)x(M+1)**
- **Fila 0 y columna 0**: **casos base** de valor 0.
- Los **valores** que caen **fuera de la tabla** son casos base de **valor -∞**.

Paso 2. Definición de las tablas y cómo rellenarlas(cont)

2.2. Forma de rellenar las tablas:

- **Inicializar los casos base**:

  $V[i, 0] := 0; V[0, j] := 0$

- **Para** todo **i** desde 1 hasta **n**

  Para todo **j** desde 1 hasta **M**, aplicar la ecuación:

$$V[i, j] := \mathbf{max}\ (V[i-1, j], b_i + V[i-1, j-p_i])$$

Si $j-p_i$ es negativo, entonces es el caso $-\infty$, y el máximo será siempre el otro término.

- El **beneficio óptimo** es $V[n, M]$

```
/* La función mochila1 devuelve el beneficio total */
función Mochila1(p,b:[1..n] de entero; M:entero) devuelve natural;
     devuelve V(n,M)
ffunción Mochila1
/* El algoritmo V rellena un valor de la tabla y lo devuelve */
algoritmo V (i,j: natural) devuelve natural;  /* devuelve el valor de V[i,j]*/
   /* Inicializar los casos base */
  para i:=1 hasta n hacer V[i,0]:=0 fpara;
  para j:=0 hasta M hacer V[0,j]:=0 fpara;
  /* resto de los casos V[i, j]:= max(V[i-1, j], bi + V[i-1, j-pi]) */
  para i:=1 hasta n hacer
    para j:=1 hasta M hacer
      si j<p[i]  entonces /*j-pi es negativo,caso -∞, y el máximo será el otro término.*/
          V[i,j]:=V[i-1,j]
        sino
          si V[i-1,j] ≥ V[i-1,j-p[i]]+b[i] entonces
              V[i,j]:=V[i-1,j]
            sino
              V[i,j]:=V[i-1,j-p[i]]+b[i]
          fsi
      fsi
    fpara
  fpara
falgoritmoV
```

Paso 2. **Ejemplo. n**= 3, **M**= 6, **p**= (2, 3, 4), **b**= (1, 2, 5)

(V[i, j]:= max (V[i-1, j], V[i-1, j-p_i]+ b_i))

j

V	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	**6**

i

- Orden de complejidad del algoritmo : Cada componente de la tabla V se calcula en tiempo constante, luego el coste de construcción de la tabla es **O(nM).**

- Cálculos posibles a partir de la tabla V:
  - beneficio total: V[n,M]
  - los objetos metidos en la mochila:

Paso 3. Recomponer la solución óptima

- **V[n, M]** almacena el beneficio óptimo, pero ¿cuál son los objetos que se cogen en esa solución?
- Obtener la tupla solución $(\mathbf{x_1, x_2, ..., x_n})$ usando **V**.
- **Proceso:** partiendo de la posición V[n, M], analizar las decisiones que se tomaron para cada objeto **i**.
  - Si $\mathbf{V[i, j] = V[i\text{-}1, j]}$, entonces la solución **no** usa el objeto $\mathbf{i} \Rightarrow \mathbf{x_i = 0}$
  - Si $\mathbf{V[i, j] = V[i\text{-}1, j\text{-}p_i] + b_i}$, entonces **sí** se usa el objeto $\mathbf{i} \Rightarrow \mathbf{x_i = 1}$
  - Si se cumplen ambas, entonces podemos usar el objeto **i** o no (existe más de una solución óptima).

Paso 3. Recomponer la solución óptima (cont.). Algoritmo.

```
función Objetos(M:natural; V[0..n][0..M] de natural b,p[1..n] de natural)
      test(n,M)
ffunciónObjetos
algoritmo test(i,j: natural)
  variables x:[1..n] de {0,1}
     j:= M
     para i:= n hasta 1 (dec 1) hacer
          si V[i, j] == V[i-1, j] entonces
               x[i]:= 0
          sino     /* V[i, j] == V[i-1, j-pi] + bi */
               x[i]= 1
               j= j – pi
          fsi
     fpara
ffuncióntest
```

V	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	**6**

- Aplicar sobre el ejemplo anterior.

n= 3, **M**= 6, **p**= (2, 3, 4), **b**= (1, 2, 5)

- solución óptima es:
  - **(x_1,x_2,x_3)=(1,0,1)**
  - con **beneficio 6**

paso	i (Obj)	V[i, j]	V[i-1, j]	j (peso)	X (vector solución)		
					1	2	3
inicial	-	-	-	6	-	-	-
1	3	6	3	6-4=2	-	-	1
2	2	1	1	2		0	1
3	1	1	0	2-2=0	1	0	1

Análisis de eficiencia.

- Cada componente de la tabla V se calcula en tiempo constante, luego el coste de construcción de la tabla es **O(nM)**

- El algoritmo **test** se ejecuta una vez por cada valor de i, desde n descendiendo hasta 1, luego su coste es **O(n)**

- Si *M* es muy grande, entonces esta solución no es buena.

- Si los pesos p_i o la capacidad *M* son reales, esta solución no sirve.

- **Problema:** Dado un conjunto de **n** tipos de monedas, cada una con valor c_i, y dada una cantidad **P**, encontrar el número **mínimo** de monedas que tenemos que usar para obtener esa cantidad.
- **Datos del problema:**
  - **P** : cantidad para cambiar.
  - **n**: número de tipos de monedas disponibles. Supondremos una cantidad ilimitada.
  - $\mathbf{c = (c_1, c_2, \ldots, c_n)}$ valor de cada tipo de monedas.
- **Representación de la solución:**
  - Una solución será de la forma $\mathbf{S = (x_1, x_2, \ldots, x_n)}$, $x_i \geq 0$, donde $\mathbf{x_i}$ es el número de monedas de tipo **i**. Suponemos que la moneda **i** vale $\mathbf{c_i}$.
- **Formulación matemática:** **Minimizar** $\sum_{i=1}^{n} x_i$,sujeto a $\sum_{i=1}^{n} x_i c_i = P$,$x_i \geq 0$
- **Solución**: conjunto de monedas que suman la cantidad **P**.
- **Ejemplo**: P= 3.89; `C={1,2,5,10,20,50,100,200}; X={0,2,1,1,1,1,1,1}`

- El algoritmo voraz es muy eficiente, pero sólo funciona en un número limitado de casos.
- **Utilizando programación dinámica:**
  1. Definir el problema en función de problemas más pequeños
  2. Definir las tablas de subproblemas y la forma de rellenarlas
  3. Establecer cómo obtener el resultado a partir de las tablas

Paso 1. Descomposición recurrente del problema

- Interpretar como un problema de toma de decisiones.
- ¿Coger o no coger **una** moneda de tipo **k**?
  - → **Si se coge:** usamos 1 más y tenemos que devolver cantidad $\mathbf{c_k}$ menos.
  - → **Si no se coge:** tenemos el mismo problema pero descartando la moneda de tipo **k**.
- ¿Qué varía en los subproblemas?
  - Tipos de monedas a usar.
  - Cantidad por devolver.

Paso 1. (Continuación. Descomposición recurrente del problema y caso base.)

- **Ecuación recurrente del problema. Cambio(k, q: entero): entero**
  Problema del cambio de monedas, considerando sólo los **k** primeros tipos, con cantidad a devolver **q**. Devuelve el número mínimo de monedas necesario.
- **Definición de Cambio(k, q: entero): entero**
  - **Si no se coge ninguna moneda de tipo k:**
    Cambio(k, q) = Cambio(k - 1, q)
  - **Si se coge 1 moneda de tipo k:**
    Cambio(k, q) = 1 + Cambio(k, q - c_k)
  - **Valor óptimo:** el que use menos monedas:

  Cambio(k, q) = **min** { Cambio(k - 1, q), 1 + Cambio(k, q - c_k) }
- **Casos base:**
  - Si **q**=0, no usar ninguna moneda: Cambio(k, 0) = 0
  - En otro caso, si **q**<0 ó **k**≤0, no se puede resolver el problema: Cambio(q, k) = $+\infty$

Paso 1. (Continuación)

- **Ecuación recurrente:**

$$\text{Cambio}(k, q) = \begin{cases} 0 & \text{Si } \mathbf{q}=0 \\ +\infty & \text{Si } \mathbf{q}<0 \text{ ó } \mathbf{k}\leq 0 \\ \min\{\text{Cambio}(k-1, q),\ 1 + \text{Cambio}(k, q-c_k)\} & \end{cases}$$

Paso 2. Definición de las tablas y cómo rellenarlas

2.1. Dimensiones y tamaño de la tabla

2) Aplicación ascendente mediante tablas

- Matriz **D** → D[i, j] = Cambio(i, j)
- **D: array** [1..**n**, 0..**P**] **de** entero

para i:= 1 **hasta** n **hacer** D[i, 0]:= 0
para i:= 1 **hasta** n **hacer**
  para j:= 0 **hasta** P **hacer**
    D[i, j]:= min(D[i-1, j], 1+D[i, j-c_i])
devolver D[n, P]

Si j-c_i es negativo, entonces es el caso +∞, y el mínimo será siempre el otro término.

```
/* La función Cambio devuelve el número mínimo de monedas necesario */
función Cambio(c:[1..n] de entero; P:entero) devuelve natural;
     devuelve D(n,P)
ffunción Cambio
/* El algoritmo D rellena un valor de la tabla y lo devuelve */
algoritmo D(i,j: natural) devuelve natural;  /* devuelve el valor de D[i,j]*/
   /* Inicializar caso base */
  para j:=0 hasta P hacer D[0,j]:=0 fpara;
  /* resto de los casos D[i, j]:= min(D[i-1, j], 1 + D[i, j-ci]) */
  para i:=1 hasta n hacer
    para j:=0 hasta P hacer
      si j<c[i]  entonces /*j-ci es negativo,caso +∞, y el mínimo será el otro término.*/
          D[i,j]:=D[i-1,j]
        sino
          si D[i-1,j] ≤ D[i,j-c[i]]+1 entonces
              D[i,j]:=D[i-1,j]
            sino
              D[i,j]:=D[i,j-c[i]]+1
          fsi
      fsi
    fpara
  fpara
falgoritmoD
```

- **Ejemplo. n**= 3, **P**= 8, **c**= (1, 4, 6)

$\min(D[i-1, j], 1+D[i, j-c_i])$

D	0	1	2	3	4	5	6	7	8
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 $c_1=6$	0	1	2	3	1	2	1	2	**2**

- ¿Cuánto es el orden de complejidad del algoritmo?
- ¿Cómo es en comparación con el algoritmo voraz?

3) Cómo recomponer la solución a partir de la tabla

- ¿Cómo calcular cuántas monedas de cada tipo deben usarse, es decir, la tupla solución $(x_1, x_2, ..., x_n)$?
- Analizar las decisiones tomadas en cada celda, empezando en **D[n, P]**.
- ¿Cuál fue el mínimo en cada **D[i, j]**?
  - **D[i - 1, j]** → No utilizar ninguna moneda más de tipo **i**.
  - **D[i, j - C[i]] + 1** → Usar una moneda más de tipo **i**.

3) Cómo recomponer la solución a partir de la tabla (continuación)

- Implementación:

```
función Monedas(M:natural; D[1..n][0..P] de natural c[1..n] de natural)
      test(n,P)
ffunciónObjetos

algoritmo test(i,j: natural)
   x: array [1..n] de entero  /* x[i] = número de monedas usadas de tipo i */
   x:= (0, 0, ..., 0)
   i:= n
   j:= P
   mientras (i≠0) AND (j≠0) hacer
     si D[i, j] == D[i-1, j] entonces
        i:= i - 1
     sino
        x[i]:= x[i] + 1
        j:= j - ci
     finsi
   finmientras
```

D	0	1	2	3	4	5	6	7	8
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_1=6	0	1	2	3	1	2	1	2	**2**

paso	i (tipo)	D[i,j]	D[i-1,j]	j (cantidad)	X (vector solución)		
					1	2	3
inicial	3	-	-	8	0	0	0
1	**3**	2	2	8	0	0	**0**
2	**2**	2	8	8-4=4	0	**1**	0
3	**2**	1	4	4-4=0	0	**2**	0

- **Ejemplo. n**= 3, **P**= 8, **c**= (1, 4, 6)
  - solución óptima es: **(x_1,x_2,x_3)=(0,2,0)**
  - con **2 monedas** de cantidad **4** (tipo 2)

- ¿Qué pasa si hay varias soluciones óptimas?
- ¿Y si no existe ninguna solución válida?

- **Algoritmo de Floyd :**

  Caminos mínimos entre todos los pares de nodos de un grafo
- **Problema**:
  - Cálculo de los caminos de coste mínimo entre todos los pares de vértices de un grafo dirigido sin ciclos de peso negativo.
- Principio de optimalidad:
  - Si $i_1, i_2, \ldots, i_k, i_{k+1}, \ldots, i_n$ es un camino de coste mínimo de i_1 a i_n, entonces:
    - $i_1, i_2, \ldots, i_k$ es un camino de coste mínimo de i_1 a i_k, y
    - $i_k, i_{k+1}, \ldots, i_n$ es un camino de coste mínimo de i_k a i_n.
- Aplicación del principio:
  - Si k es el vértice intermedio de mayor índice en el camino óptimo de i a j, entonces el subcamino de i a k es un camino óptimo de i a k que, además, sólo pasa por vértices de índice menor que k.
  - Lo análogo ocurre con el subcamino de k a j.

- **Utilizando programación dinámica:**

Paso 1. Ecuación recurrente del problema

- ***C(i,j)*** : coste de la arista (i,j) o infinito si esa arista no existe. **C(i,i)=0**.
- $\boldsymbol{D_k(i, j)}$**:** longitud (o distancia) del camino de coste mínimo de *i* a *j* que no pasa por ningún vértice de índice mayor que *k*.
- Sea ***D*(*i,j*)** la longitud del camino de coste mínimo de *i* a *j*.
- Se cumple que:

$$D(i,j) = \min \left\{ \min_{1\le k\le n} \{ D_{k-1}(i, k) + D_{k-1}(k, j) \}\ C(i, j) \right\}$$

$$D_0(i,j) = C(i, j) \qquad 1\le i\le n, 1\le j\le n$$

- **Razonamiento inductivo:** para calcular los caminos mínimos pudiendo pasar por los **k** primeros nodos usamos los caminos mínimos pasando por los **k-1** primeros. Un camino óptimo de *i* a *j* que no pase por ningún vértice de índice mayor que *k* bien pasa por el vértice *k* ó no.
  - Si pasa por *k* entonces:

    $$D_k(i,j) = D_{k-1}(i, k) + D_{k-1}(k, j)$$

  - Si no pasa por *k* entonces ningún vértice intermedio tiene índice superior a *k*-1:

    $$D_k(i,j) = D_{k-1}(i, j)$$

- **Paso 1. Ecuación recurrente del problema** (continuación)
- **ecuación recurrente** que define el método de programación dinámica.
  - **D_k(i, j):** camino mínimo de **i** a **j** pudiendo pasar por los nodos 1, 2, …, **k**.

$$D_k(i, j) = \begin{cases} C[i, j] & \text{Si } k=0 \\ \min(D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j)) & \text{Si } k>0 \end{cases}$$

$D_n(i, j) \rightarrow$ caminos mínimos finales

Paso 2. Definición de las tablas y cómo rellenarlas

- Matriz **D** → D[i, j] = $D_n(i, j)$
- **D: array** [1..**n**, 1..**n**] **de** nat

```
para i:= 1 hasta n hacer D[i, i]:= 0
  para j:= 1 hasta n hacer
    D0(i,j)=C(i,j) /* ∞ si no hay arco */
para k:= 1 hasta n hacer
  para i:= 1 hasta n hacer
    para j:= 1 hasta n hacer
      Dk(i, j) := min(Dk-1(i, j), Dk-1(i, k) + Dk-1(k, j))
devolver Dn
```

```
{Pre: g es un grafo dirigido etiquetado sin ciclos negativos}
función Floyd(g:grafo) devuelve D[1..n,1..n] de natural
 variables D:vector[1..n,1..n] de natural; k,i,j:vértice;
     /* Inicializar los casos base, valor de la arista que une dos vértices, D0(i,j)=C(i,j) 1≤i≤n, 1≤j≤n; las
     diagonales se ponen a cero o bloquean */
 para i=1 hasta n hacer
    para j=1 hasta n hacer
      D[i,j]:=etiqueta(g,i,j)  /* ∞ si no hay arco */
    fpara;
    D[i,i]:=0  /* ó D[i,i]:=“-” */
  fpara;
     /* resto de los casos Dk(i,j):=min(Dk-1(i,j), Dk-1(i,k) + Dk-1(k,j)) */
  para k=1 hasta n hacer
    para i=1 hasta n hacer
      para j=1 hasta n hacer
        si (i≠k) AND (j≠k) AND (i≠j) entonces
          si D[i,k]+D[k,j] < D[i,j]  entonces
             D[i,j]:=D[i,k]+D[k,j]
          fsi
        fsi
      fpara
    fpara
  fpara;
  devuelve D
ffunciónFloyd
{Post: D=caminosMínimos(g)}
```

Eficiencia temporal: $\Theta(n^3)$

- Cálculo posible a partir de la tabla D:
  - Caminos mínimos entre todos los pares de nodos : $\mathbf{D_n(i, j)}$

Paso 3. Recomponer la solución a partir de la tabla

- ¿Cómo calcular la **ruta asociada del camino mínimo** entre dos vértices, es decir, la tupla solución $(x_i,.., x_j)$?
- Cálculo de las secuencias de nodos que componen los caminos mínimos
  - si el camino mínimo de *m* a *n* pasa primero por *p* y después por *q*, la secuencia de vértices que forman el camino mínimo de *p* a *q* forma parte de la secuencia de vértices que forman el camino mínimo de *m* a *n*
  - **usar** un **vector bidimensional *C*** indexado por vértices: *C*[*i*,*j*] contiene un nodo *k* que forma parte del camino mínimo entre *i* y *j*
    - $\mathbf{C[i\,,\,j]=k} \Rightarrow$ El nodo k es el predecesor(padre) de j y forma parte del camino mínimo entre *i* y *j*.
    - $\mathbf{C[i\,,\,j]}= \infty \Rightarrow$ no hay ningún camino desde *i* a *j*.
- Las matrices finales *D* y *C* contendrán toda la información necesaria para determinar la ruta más corta entre dos nodos cualquiera de la red.

```
función Floyd(g:grafo) devuelve D,C: vector[1..n,1..n] de natural
 variables D,C:vector[1..n,1..n] de natural; k,i,j:vértice;
 /* Inicializar los casos base de D, valor de la arista que une dos vértices, D0(i,j)=C(i,j) 1≤i≤n, 1≤j≤n; las
      diagonales se ponen a cero o bloquean */
 para i=1 hasta n hacer
    para j=1 hasta n hacer D[i,j]:=etiqueta(g,i,j);  /* ∞ si no hay arco */
    D[i,i]:=0;  /* ó D[i,i]:="-" */
 /* Inicializar C: C[i , j]= valor que representará el nodo predecesor a j en el camino mínimo desde i hasta j.
       Inicialmente se comienza con caminos de longitud 1, por lo que C[i , j]= i .*/
 para i=1 hasta n hacer
    para j=1 hasta n hacer C[i,j]:=i;  /* ∞ si no hay arco */
    C[i,i]:=0;  /* ó C[i,i]:="-" */
 /* resto de los casos Dk(i,j):=min(Dk-1(i,j), Dk-1(i,k) + Dk-1(k,j)) */
  para k=1 hasta n hacer
    para i=1 hasta n hacer
      para j=1 hasta n hacer
        si (i≠k) AND (j≠k) AND (i≠j) entonces
          si D[i,k]+D[k,j] < D[i,j]  entonces
             D[i,j]:=D[i,k]+D[k,j];
              C[i,j]:= C[k,j];
          fsi
        fsi
devuelve D,C;
Ffunción Floyd
{Post: D,C=caminosMínimos(g)}
```

Eficiencia temporal: $\Theta(n^3)$

Paso 3. Recomponer la solución a partir de la tabla(cont)

```
función CaminosMinimos(C[1..n][1..n])
  para i:= 1 hasta n hacer
    para j:= 1 hasta n hacer
      ruta(i,j) /*ruta asociada del camino mínimo entre el nodo i y el nodo j */
    fpara
  fpara
ffunción CaminosMinimos
algoritmo ruta(i,j: natural)
  variables x:[1..n] de natural; k:natural;
     k:= 1;
     si (C[i,j] ≠ ∞) AND (i ≠ j) AND (C[i,j] ≠ i) entonces
       x[k]:= j;  /* j es el último nodo del camino de i a j */
       mientras C[i,j] ≠ i hacer
          x[k]:= C[i,j]; /*C[i,j]=m, el nodo predecesor al j es el m ⇒ m →j */
          j:= C[i,j];    /* actualizar el nodo predecesor */
          k:= k+1
       fmientras
       x[k]:= i /* i es el primer nodo del camino de i a j */
       devolver reverse(x) /* retorna el camino desde i a j. */
     sino
     /* no hay ningún camino desde i a j. */
     fsi
ffunciónruta
```

Algoritmo de Floyd. Resumen y ejemplo de aplicación.

- El algoritmo de Floyd es más general que el de Dijkstra, ya que determina la ruta más corta entre dos nodos cualquiera de la red.

Implementación

1. **Estructuras de datos**:
   - El algoritmo representa una red de *n* nodos como una matriz cuadrada de orden *n*, la llamaremos **matriz *D***. De esta forma,
     - el valor **D[i , j]** representa el **coste de ir desde el nodo *i* al nodo *j***,
     - inicialmente en caso de no existir un arco entre ambos, el valor **D[i , j]** será infinito.
   - Definiremos otra **matriz *C***, también cuadrada de orden *n*, cuyos elementos van a ser los **nodos predecesores en el camino hacia el nodo origen**, es decir,
     - el valor **C[i , j]**= **representará el nodo predecesor a *j* en el camino mínimo desde *i* hasta *j***.
     - Inicialmente se comienza con caminos de **longitud *1***, por lo que **C[i , j]= i** .
   - Las diagonales de ambas matrices representan el coste y el nodo predecesor para ir de un nodo a si mismo, por lo que no sirven para nada, estarán bloqueadas, valor 0 o “-”.

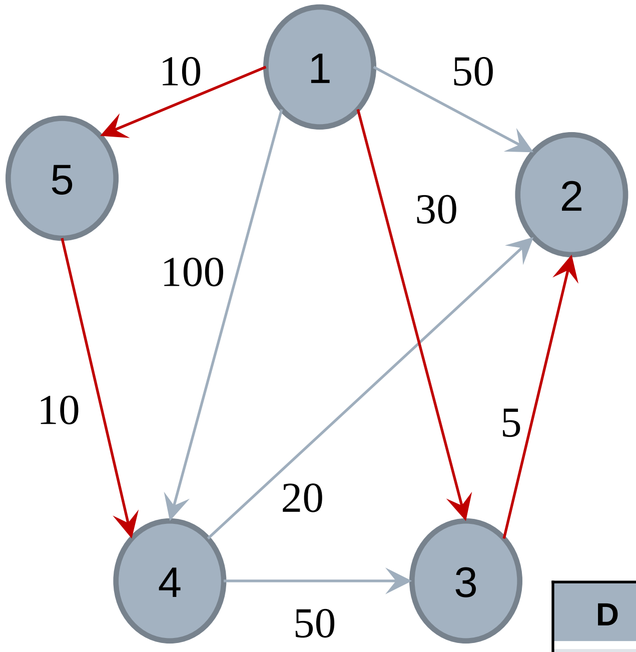
- **Implementación** (cont.)

2. Algoritmo Los pasos a dar en la aplicación del algoritmo de Floyd son los siguientes:

1. Formar las matrices iniciales *C* y *D*.
2. Se toma $k=1$.
3. Se selecciona la fila y la columna *k* de la matriz *D* y entonces, para *i* y *j*, con $i \neq k$, $j \neq k$ e $i \neq j$, hacemos:
   - Si $(\mathbf{D[i,k] + D[k,j]}) < \mathbf{D[i,j]} \Rightarrow \mathbf{C[i,j] = C[k,j]}$ y $\mathbf{D[i,j] = D[i,k] + D[k,j]}$
   - En caso contrario, dejamos las matrices como están.
4. Si $k \leq n$, aumentamos *k* en una unidad y repetimos el paso anterior(3.), en caso contrario paramos las iteraciones.

- La matriz final *D* contiene los costes óptimos para ir de un vértice a otro, mientras que la matriz *C* contiene los penúltimos vértices de los caminos óptimos que unen dos vértices, lo cual permite **reconstruir** cualquier camino óptimo para ir de un vértice a otro.
- **Ejemplo:** Aplicar el algoritmo de Floyd sobre el siguiente grafo para obtener las rutas más cortas entre cada dos nodos.

4. Algoritmo de **Floyd**. Ejemplo.



1. Formar las **matrices iniciales *D* y *C***.
   - Inicialización de la matriz ***D***.
     - **D[i , j]**= valor que representa el **coste** de ir desde el nodo ***i*** al nodo ***j***
     - inicialmente en caso de no existir un arco entre ambos, el valor **D[i , j]**= ∞ caso de no existir un arco entre **el nodo *i* y el nodo *j***
   - Inicialización de la matriz ***C***.
     - **C[i , j]**= valor que representará el **nodo predecesor** a ***j*** en el camino mínimo desde ***i*** hasta ***j***.
     - Inicialmente se comienza con caminos de **longitud *1***, por lo que **C[i , j]= i** .

D	1	2	3	4	5
1	-	50	30	100	10
2	∞	-	∞	∞	∞
3	∞	5	-	∞	∞
4	∞	20	50	-	∞
5	∞	∞	∞	10	-

C	1	2	3	4	5
1	-	1	1	1	1
2	∞	-	∞	∞	∞
3	∞	3	-	∞	∞
4	∞	4	4	-	∞
5	∞	∞	∞	5	-

Tablas: Inicialización de las matrices de costes D y de los caminos mínimos C.

4. Algoritmo de **Floyd**. Ejemplo.

2. Tomamos ***k=1***:

3.1 Se selecciona la fila y la columna ***k*** de la matriz *D* y entonces, para *i* y *j*, con *i≠k*, *j≠k* e *i≠j*, hacemos:
Si (**D[i , k] + D[k , j]**) < **D[i , j]** ⇒ **C[i , j] = C[k, j]** y **D[i , j] = D[i , k] + D[k , j] ,** en caso contrario, dejamos las matrices como están.

- Tomamos ***i=2*** *(i ≠k)*:
  - *j=3 (j≠k, j≠i (D[2,3])): D[2,1]+D[1,3]* = ∞+30, no hay que cambiar nada, no podemos llegar de *2* a *3* a través de *1*.
  - *j=4 (j≠k, j≠i (D[2,4])): D[2,1]+D[1,4] = ∞+100=∞*,no hay que cambiar nada, no podemos llegar de *2* a *4* a través de *1*.
  - *j=5 (j≠k, j≠i(D[2,5])): D[2,1]+D[1,5] =∞+10 = ∞*, no hay que cambiar nada, no podemos llegar de *2* a *5* a través de *1*.
- Tomamos ***i=3*** *(i ≠k)*: como *D[3,1]=∞*, entonces no habrá ningún cambio, no puede haber ningún camino desde *3* a través de *1*.
- Tomamos ***i=4*** *(i ≠k)*: en este caso ocurre como en el paso anterior, como *D[4,1]=∞*, entonces no habrá ningún cambio, no puede haber ningún camino desde *4* a través de *1*.
- Tomamos ***i=5*** *(i ≠k)*, como *D[5,1]=∞*, entonces no habrá ningún cambio, no puede haber ningún camino desde *5* a través de *1*.

D	1	2	3	4	5	C	1	2	3	4	5
1	-	50	30	100	10	1	-	1	1	1	1
2	∞	-	∞	∞	∞	2	∞	-	∞	∞	∞
3	∞	5	-	∞	∞	3	∞	3	-	∞	∞
4	∞	20	50	-	∞	4	∞	4	4	-	∞
5	∞	∞	∞	10	-	5	∞	∞	∞	5	-

Tablas: Evolución del conjunto D y de los caminos mínimos C

4. Algoritmo de **Floyd**. Ejemplo.

3.2. Tomamos ***k=2***:

- Tomamos ***i=1*** *(i ≠k)*:
  - *j=3 (j≠k, j≠i (D[1,3]))*: *D[1,2]+D[2,3] =50+∞ > D[1,3] = 30*, , por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=4 (j≠k, j≠i(D[1,4])): D[1,2]+D[2,4] =50+∞ > D[1,4] = 100*, , por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=5 (j≠k, j≠i(D[1,5])): D[1,2]+D[2,5] =50+∞= ∞*, no hay que cambiar nada, no podemos llegar de *1* a *5* a través de *2*.
- Tomamos ***i=3*** *(i ≠k)*:
  - *j=1 (j≠k, j≠i (D[3,1]))*: *D[3,2]+D[2,1] = 5+∞* no hay que cambiar nada, no podemos llegar de *3* a *1* a través de *2*.
  - *j=4 (j≠k, j≠i (D[3,4]))*: *D[3,2]+D[2,4] = 5+∞ = ∞*, no hay que cambiar nada, no podemos llegar de *3* a *4* a través de *2*.
  - *j=5 (j≠k, j≠i (D[3,5]))*: *D[3,2]+D[2,5] = 5+∞ = ∞*, no hay que cambiar nada, no podemos llegar de *3* a *5* a través de *2*.
- Tomamos ***i=4*** *(i ≠k)*:
  - *j=1 (j≠k, j≠i)*: *D[4,2]+D[2,1] = 20+∞ = ∞*, no hay que cambiar nada, no podemos llegar de *4* a *1* a través de *2*.
  - *j=3 (j≠k, j≠i)*: *D[4,2]+D[2,3] = 20+∞ = ∞*, no hay que cambiar nada, no podemos llegar de *4* a *3* a través de *2*.
  - *j=5 (j≠k, j≠i)*: *D[4,2]+D[2,5] = 20+∞ = ∞*, no hay que cambiar nada, no podemos llegar de *4* a *5* a través de *2*.
- Tomamos ***i=5*** *(i ≠k)*, en este caso ocurre como en el paso anterior, como *D[5,2]=∞*, entonces no habrá ningún cambio, no puede haber ningún camino desde *5* a través de *2*.

D	1	2	3	4	5	C	1	2	3	4	5
1	-	50	30	100	10	1	-	1	1	1	1
2	∞	-	∞	∞	∞	2	∞	-	∞	∞	∞
3	∞	5	-	∞	∞	3	∞	3	-	∞	∞
4	∞	20	50	-	∞	4	∞	4	4	-	∞
5	∞	∞	∞	10	-	5	∞	∞	∞	5	-

Tablas: Evolución del conjunto D y de los caminos mínimos C

4. Algoritmo de **Floyd**. Ejemplo.

3.3. Tomamos ***k=3***:

- Tomamos ***i=1*** *(i ≠k)*:
  - *j=2 (j≠k, j≠i)*: *D[1,3]+D[3,2]* =*30*+5=35 < *D[1,2] = 50*, por tanto hacemos: ***D[1,2] = 35*** y ***C[1,2] = 3***.
  - *j=4 (j≠k, j≠i(D[1,4])): D[1,3]+D[3,4] =30+∞ > D[1,4] = 100*, por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=5 (j≠k, j≠i(D[1,5])): D[1,3]+D[3,5] =30+ ∞ > D[1,5] = 10*, por tanto no hay que cambiar nada, el camino es mayor que el existente.
- Tomamos ***i=2*** *(i ≠k)*: como *D[2,3]*=∞, entonces no habrá ningún cambio, no puede haber ningún camino desde *2* a través de *3*.
- Tomamos ***i=4*** *(i ≠k)*:
  - *j=1 (j≠k, j≠i)*: *D[4,3]+D[3,1] = 50+∞* = ∞, no hay que cambiar nada, no podemos llegar de *4* a *1* a través de 3.
  - *j=2 (j≠k, j≠i)*: *D[4,3]+D[3,2] = 50+5 = 55 > D[4,2]*, por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=5 (j≠k, j≠i)*: *D[4,3]+D[3,5] = 50+∞* = ∞, no hay que cambiar nada, no podemos llegar de *4* a *5* a través de *3*.
- Tomamos ***i=5*** *(i ≠k)*, como *D[5,3]*=∞, entonces no habrá ningún cambio, no puede haber ningún camino desde *5* a través de *3*.

D	1	2	3	4	5	C	1	2	3	4	5
1	-	35	30	100	10	1	-	3	1	1	1
2	∞	-	∞	∞	∞	2	∞	-	∞	∞	∞
3	∞	5	-	∞	∞	3	∞	3	-	∞	∞
4	∞	20	50	-	∞	4	∞	4	4	-	∞
5	∞	∞	∞	10	-	5	∞	∞	∞	5	-

Tablas: Evolución del conjunto D y de los caminos mínimos C

4. Algoritmo de **Floyd**. Ejemplo.

3.4. Tomamos ***k=4***:

- Tomamos ***i=1*** *(i ≠k)*:
  - *j=2 (j≠k, j≠i (D[1,2])): D[1,4]+D[4,2] =100*+20 > *D[1,2] = 35*, por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=3 (j≠k, j≠i(D[1,3])): D[1,4]+D[4,3] =100*+50 > *D[1,3] = 30*, por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=5 (j≠k, j≠i(D[1,5])): D[1,4]+D[4,5] =100+∞= ∞ > D[1,5] = 10*, no hay que cambiar nada, no podemos llegar de *1* a *5* a través de *4*.
- Tomamos ***i=2*** *(i ≠k)*: como *D[2,4]=∞ ,* no habrá ningún cambio.
- Tomamos ***i=3****(i ≠k)*: como *D[3,4]=∞*, no habrá ningún cambio.
- Tomamos ***i=5*** *(i ≠k)*,
  - *j=1 (j≠k, j≠i (D[5,1])): D[5,4]+D[4,1] = 10+∞* no hay que cambiar nada, no podemos llegar de *5* a *1* a través de *4*.
  - *j=2 (j≠k, j≠i (D[5,2])): D[5,4]+D[4,2] = 10*+20 = ***30*** < *D[5,2]* = ∞, ⇒***D[5,2] = 30*** y ***C[5,2] = 4***.
  - *j=3 (j≠k, j≠i (D[5,3])): D[5,4]+D[4,3] = 10*+50 = ***60*** < *D[5,3]* = ∞, ⇒ ***D[5,3] = 60*** y ***C[5,3] = 4***.

D	1	2	3	4	5	C	1	2	3	4	5
1	-	**35**	30	100	10	1	-	**3**	1	1	1
2	∞	-	∞	∞	∞	2	∞	-	∞	∞	∞
3	∞	5	-	∞	∞	3	∞	3	-	∞	∞
4	∞	20	50	-	∞	4	∞	4	4	-	∞
5	∞	**30**	**60**	10	-	5	∞	**4**	**4**	5	-

Tablas: Evolución del conjunto D y de los caminos mínimos C

4. Algoritmo de **Floyd**. Ejemplo.

3.5. Tomamos ***k=5***:

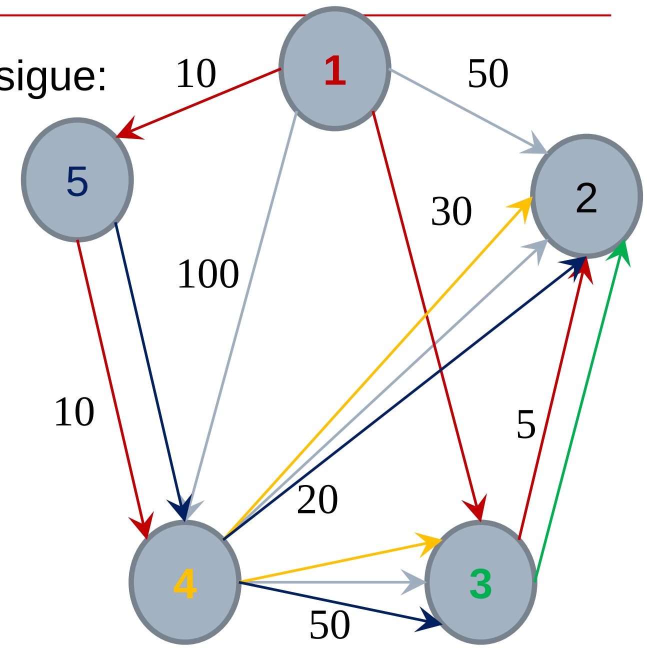
- Tomamos ***i=1*** *(i ≠k)*:
  - *j=2 (j≠k, j≠i)*: *D[1,5]+D[5,2] =10*+30=80 > *D[1,2] = 35*, por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=3 (j≠k, j≠i): D[1,5]+D[5,3] =10*+35=45 > *D[1,3] = 30*, por tanto no hay que cambiar nada, el camino es mayor que el existente.
  - *j=4 (j≠k, j≠i(D[1,4])): D[1,5]+D[5,4] =10*+10=20 < *D[1,4] = 100*, ⇒ ***D[1,4] = 20*** y ***C[1,4] = 5***.
- Tomamos ***i=2*** *(i ≠k)*: en este caso ocurre como en el paso anterior, como *D[2,5]*=∞, entonces no puede haber ningún camino desde *2* a través de *5*.
- Tomamos ***i=3****(i ≠k)*: en este caso ocurre como en el paso anterior, como *D[3,5]*=∞, entonces no puede haber ningún camino desde *3* a través de *5*.
- Tomamos ***i=4*** *(i ≠k)*: como *D[4,5]*=∞, entonces no puede haber ningún camino desde *4* a través de *5*.

D	1	2	3	4	5	C	1	2	3	4	5
1	-	**35**	30	**20**	10	1	-	**3**	1	**5**	1
2	∞	-	∞	∞	∞	2	∞	-	∞	∞	∞
3	∞	5	-	∞	∞	3	∞	3	-	∞	∞
4	∞	20	50	-	∞	4	∞	4	4	-	∞
5	∞	**30**	**60**	10	-	5	∞	**4**	**4**	5	-

Tablas: Evolución del conjunto D y de los caminos mínimos C

4. Algoritmo de **Floyd**. Ejemplo.

4. $\boxed{\textbf{k = n}}$ ⇒ **FIN** del proceso, las matrices quedan como sigue:



D	1	2	3	4	5	C	1	2	3	4	5
1	-	**35**	30	**20**	10	**1**	-	**3**	1	**5**	1
2	∞	-	∞	∞	∞	**2**	∞	-	∞	∞	∞
3	∞	5	-	∞	∞	**3**	∞	3	-	∞	∞
4	∞	**20**	**50**	-	∞	**4**	∞	**4**	**4**	-	∞
5	∞	**30**	**60**	10	-	**5**	∞	**4**	**4**	5	-

Tablas: Finales del conjunto D y de los caminos mínimos C

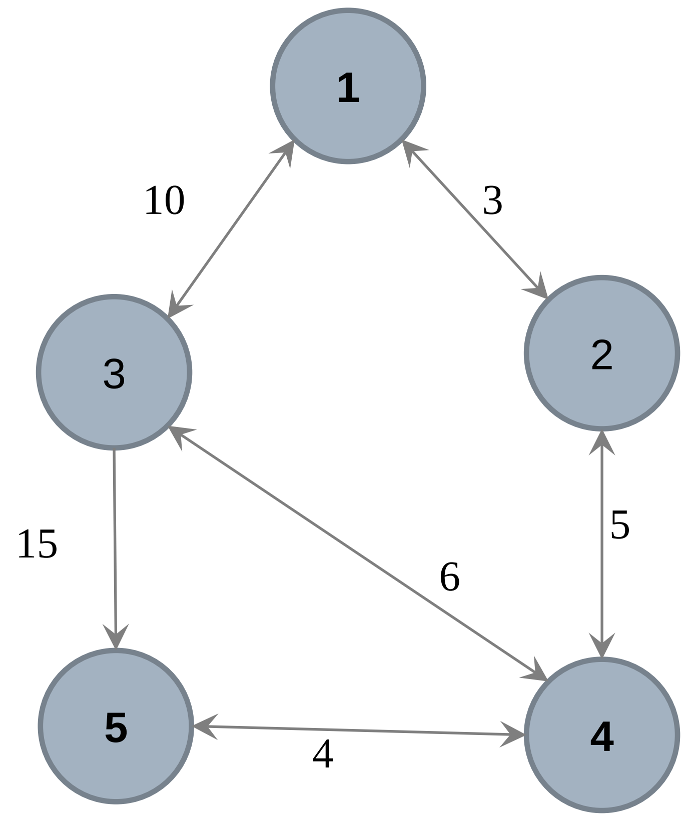
- Las matrices finales D y C contienen toda la información necesaria para determinar la ruta más corta entre dos nodos cualquiera de la red.

⇒ Aplicar el/los algoritmos vistos anteriormente para calcular:

- **Distancia más corta**: por ej, la distancia más corta del nodo *1* al nodo *4* es *D[1,4] = 20*.
- **La ruta asociada del camino mínimo**: por ej, entre el nodo *1* y el nodo *4* el procedimiento es:
  1. Consultar *C[1,4]=5*, por tanto el nodo predecesor al *4* es el *5*, es decir, *5* → *4*.
  2. Consultar *C[1,5]=1*, por tanto el nodo predecesor al *5* es el *1*, es decir, *1* → *5*→ *4*, obteniendo la ruta completa.

4. Algoritmo de **Floyd**. Ejemplo.

- **Ejercicio:** Aplicar el algoritmo de Floyd sobre el siguiente grafo para obtener las rutas más cortas entre cada dos nodos.



4. Programación dinámica. Conclusiones.

- El **razonamiento inductivo** es una herramienta muy potente en resolución de problemas.
- Aplicable no sólo en problemas de optimización.
- Para obtener la fórmula, interpretar el problema como una serie de **toma de decisiones**.
- Descomposición recursiva no necesariamente implica implementación recursiva.
- **Programación dinámica:** almacenar los resultados en una tabla, empezando por los tamaños pequeños y avanzando hacia los más grandes.