

Estrategias algorítmicas

Tema 3(II)

Algorítmica y Modelos de Computación

Tema 3. Estrategias algorítmicas sobre estructuras de datos no lineales.

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

4. Programación dinámica.

1. Introducción. Ejemplo: Cálculo de los números de Fibonacci.
2. Método general.
 - 2.1. Funcionamiento
 - 2.2. Ejemplo: Algoritmo de Floyd.
 - 2.3. Pasos para aplicar programación dinámica.
3. Análisis de tiempos de ejecución.
4. Ejemplos de aplicación.
 - 4.1. Problema de la mochila 0-1.
 - 4.2. Problema del cambio de monedas.
 - 4.3. Problema del camino mínimo. Algoritmo de Floyd.

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 **independientes** de menor tamaño.
 - Partiendo del problema grande, descendemos hacia problemas más sencillos.
- **Métodos ascendentes (programación dinámica)**
 - Se descompone el problema en subproblemas **solapados** y se va construyendo la solución con las soluciones de esos subproblemas.
 - 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.

- **Esta técnica se aplica sobre problemas que presentan estas características:**
 - Subproblemas optimales: La solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de tamaño menor.
 - Solapamiento entre subproblemas: Al plantear la solución recursiva del problema, un mismo problema se resuelve más de una vez (1).

 - **Enfoque ascendente (bottom-up)**
 - Primero se calculan las soluciones óptimas para problemas de tamaño pequeño.
 - Con dichas soluciones, encuentra soluciones a problemas de mayor tamaño.
- (1) cuando los subproblemas obtenidos no son independientes sino que existe solapamiento entre ellos; entonces es cuando una solución recursiva no resulta eficiente por la repetición de cálculos que conlleva. En estos casos es cuando la Programación Dinámica nos puede ofrecer una solución aceptable.

4. Programación dinámica. Introducción.

☐ **Clave: Memorización**

- Almacenar las soluciones de los subproblemas en alguna estructura de datos para reutilizarlas posteriormente (se consigue un algoritmo más eficiente ya que se evita resolver el mismo subproblema una y otra vez).

☐ **Para evitar calcular lo mismo varias veces:**

- Cuando se calcula una solución, ésta se almacena.
- Antes de realizar una llamada recursiva para un subproblema Q, se comprueba si la solución ha sido obtenida previamente:
 - ☐ Si no ha sido obtenida, se hace la llamada recursiva y, antes de devolver la solución, ésta se almacena.
 - ☐ Si ya ha sido previamente calculada, se recupera la solución directamente (no hace falta calcularla).
- Usualmente, se utiliza una matriz que se rellena conforme las soluciones a los subproblemas son calculados (espacio vs. tiempo).

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 $n \leq 2$ entonces devolver 1

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

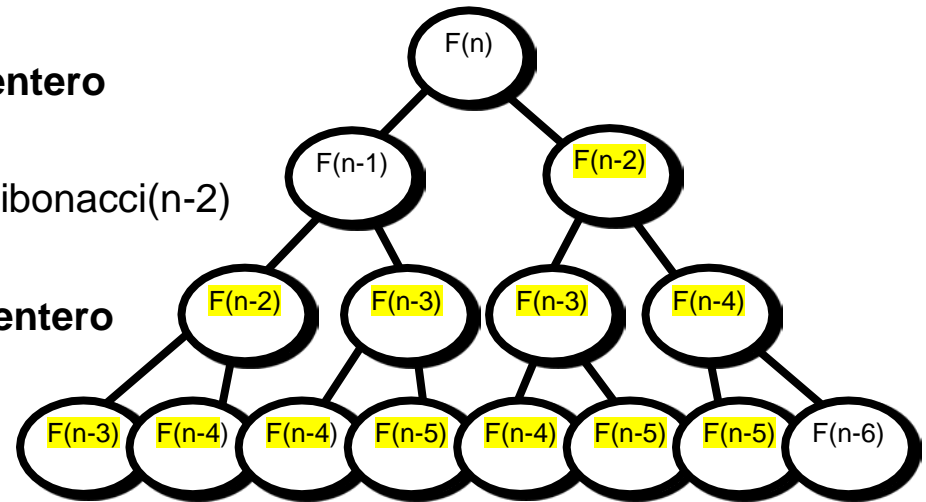
- **Con programación dinámica.**

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.

4. Programación dinámica. Método general. **Ejemplo: Algoritmo de Floyd.**

□ **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.

■ **$D_k(i, j)$:** 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. Principio de Optimalidad.

- **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.**
 - **Contraejemplo1.** 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.
 - **Contraejemplo2.** En el problema del cambio de monedas la solución óptima para 0.07€ es 0,05+0,02 y para 0,06€ es 0,05+0,01, pero la solución óptima para 0.13€ **no** es (0,05+0,02) + (0,05+0,01) sino (0,10+0,02+0,01).

4. Programación dinámica. Método general. Principio de Optimalidad.

- Para poder emplear programación dinámica, una secuencia óptima debe cumplir la condición de que cada una de sus subsecuencias también sea óptima:

Dado un problema P con n elementos, si la secuencia óptima es $e_1e_2\dots e_k\dots e_n$ entonces:

- $e_1e_2\dots e_k\dots e_n$ es solución al problema P considerando los k primeros elementos.
 - $e_{k+1}\dots e_n$ es solución al problema P considerando los elementos desde $k+1$ hasta n .
- En otras palabras, la solución óptima de cualquier instancia no trivial de un problema es una combinación de las soluciones óptimas de sus subproblemas.

4. Programación dinámica. Método general. Principio de Optimalidad.

- Supongamos que un problema se resuelve tras tomar una secuencia d_1, d_2, \dots, 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. Plantear la **solución como una sucesión de decisiones** y verificar que cumple el **principio de optimalidad** de Bellman.
2. Obtener una **descomposición recursiva** de la solución:
 - Ecuación recurrente.
 - Casos base.

Se comprueba que se cumple el principio de optimalidad de Bellman, para lo que hay que encontrar la “estructura” de la solución.

Se define recursivamente la solución óptima del problema (en función de los valores de las soluciones para subproblemas de menor tamaño).

Se identifican los subproblemas con una solución trivial (casos base).

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

3. Calcular el valor de la solución óptima mediante una tabla en donde se almacenan soluciones parciales para reutilizar los cálculos:

- **Tablas** utilizadas por el algoritmo.
- Orden y forma de **rellenarlas**.

Se calcula el valor de la solución óptima utilizando un enfoque ascendente.

Se determina el conjunto de subproblemas que hay que resolver (el tamaño de la tabla).

Se van calculando los valores de soluciones más complejas a partir de los valores previamente calculados.

4. 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. **Pasos.**

□ **Pasos** para aplicar programación dinámica:

1. Plantear la **solución como una sucesión de decisiones** y verificar que cumple el **principio de optimalidad** de Bellman.
2. Obtener una **descomposición recursiva** de la solución:
 - Ecuación recurrente.
 - Casos base.
3. Calcular el valor de la solución óptima mediante una tabla en donde se almacenan soluciones parciales para reutilizar los cálculos:
 - **Tablas** utilizadas por el algoritmo.
 - Orden y forma de **rellenarlas**.
4. 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.

5. 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 (1)
- Además, algunos de estos cálculos pueden ser innecesarios.

(1) La Programación Dinámica se base en usar estructuras (vectores o tablas) para eliminar la repetición de los cálculos, por lo que debemos tener en cuenta no sólo la **complejidad temporal** de los algoritmos estudiados, sino también en su **complejidad espacial**.
En general, los algoritmos obtenidos con esta técnica tienen complejidades (espacio y tiempo) bastante razonables, pero debemos evitar que por tratar de obtener una complejidad temporal de orden polinómico tengamos una complejidad espacial demasiado elevada, como puede ocurrir en algunos problemas.

4. Programación dinámica. Ejemplos de aplicación. **Problema de la Mochila 0-1.**

- ❑ 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.
 - **pe** = (p_1, p_2, \dots, p_n) pesos de los objetos.
 - **be** = (b_1, b_2, \dots, b_n) beneficios (valores) de los objetos.
- ❑ **Objetivo:** llenar la mochila, de capacidad **M**, de manera que se **maximice** el beneficio (valor).
- ❑ **Representación de la solución:** Una solución será de la forma **S** = (x_1, x_2, \dots, x_n) , $x_i \in \{1,0\}$, siendo cada x_i la elección o no del objeto **i**.

- ❑ **Formulación matemática:**

- **Maximizar** $\sum_{i=1}^n x_i b_i$ sujeto a la restricción $\sum_{i=1}^n x_i p_i \leq M$

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

4. Programación dinámica. Ejemplos de aplicación. **Problema de la Mochila 0-1.**

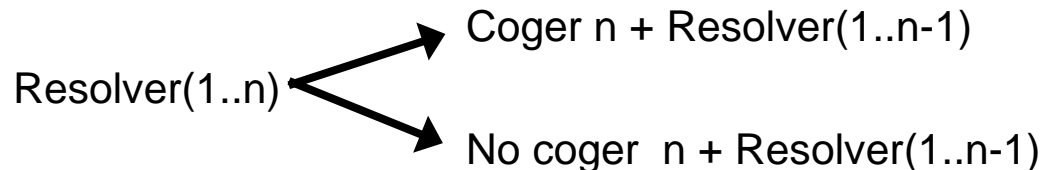
- ❑ **Ejemplo:** $n = 3$; $M = 15$; $be = (38, 40, 24)$; $pe = (9, 6, 5)$
- **Estrategia voraz**, solución que devuelve el algoritmo voraz adaptado al caso 0-1 (o se coge un objeto entero o no)
 - Tomar siempre el objeto que proporcione mayor beneficio por unidad de peso: $be/pe=(4.22,6.66,4.8)$. Se 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**.

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

- Aplicando la **programación dinámica** al problema:

Paso 1. Plantear la solución como una sucesión de decisiones y verificar Bellman.

- 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 o no coger un objeto **k**?
 - **Si se coge**: tenemos el beneficio **b_k**, pero en la mochila queda menos espacio, **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.

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

Paso 2. Obtener una descomposición recursiva (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:**
$$\text{Mochila}(k, m) = \text{Mochila}(k - 1, m)$$
 - **Si se coge:**
$$\text{Mochila}(k, m) = b_k + \text{Mochila}(k - 1, m - p_k)$$
 - **Valor óptimo:** el que dé mayor beneficio:
$$\text{Mochila}(k, m) = \max \{ \text{Mochila}(k - 1, m), b_k + \text{Mochila}(k - 1, m - p_k) \}$$
- **Casos base:**
 - Si $m=0$, no se pueden incluir objetos: **$\text{Mochila}(k, 0) = 0$**
 - Si $k=0$, tampoco se pueden incluir: **$\text{Mochila}(0, m) = 0$**
 - Si m o k son negativos, el problema es irresoluble: **$\text{Mochila}(k, m) = -\infty$**

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

Paso 2. Obtener una **descomposición recursiva** (Ecuación recurrente y caso base) (cont.)

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

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

□ ¿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 B*/  
función Mochila1(p,b:[1..n] de entero; M: natural) devuelve natural;  
  devuelve Brec(n,M)  
ffunción Mochila1  
  
/* El algoritmo recursivo Brec rellena un valor de la tabla B y lo devuelve */  
algoritmo Brec(n,M: natural) devuelve natural; /* devuelve el valor de B[n,M] */  
  /* Inicializar los casos base */  
  si M=0 entonces devuelve 0  
  sino si M<p[n] entonces devuelve B(n-1,M)  
  sino si B(n-1,M) ≥ B(n-1,M-p[n])+b[n] entonces devuelve B(n-1,M)  
  sino devuelve B(n-1,M-p[n])+b[n]  
  fsi  
falgoritmo Brec
```

□ **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**.
- La solución recursiva está generando y resolviendo el mismo problema muchas veces.

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

- La recurrencia permite escribir un **algoritmo recursivo** de forma inmediata.
- **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**.
- La solución recursiva está generando y resolviendo el mismo problema muchas veces.
- **Observación:** 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**
- Para evitar la repetición de cálculos, las soluciones de los subproblemas se deben almacenar en una tabla.
- Matriz B: $n \times M$ cuyo elemento (p, m) almacena el **beneficio** (o ganancia total) de una **solución óptima** de *Mochila*(p, m)

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

Paso 3. Definición de las tablas y cómo rellenarlas

3.1. Dimensiones y tamaño de la tabla

- Definimos la tabla **B**, para guardar los resultados de los subproblemas:
 $B[p, m] = \text{Mochila}(p, m)$ // p objetos $pe[p]$, $be[p]$ y la mochila soporta peso m
- La solución del problema original es **Mochila(n, M)**.
 - Por lo tanto, la tabla debe ser: **B: array [0..n, 0..M] de entero**
 - \Rightarrow **tamaño** de la tabla **B: (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 $-\infty$** .

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

Paso 3. Definición de las tablas y cómo rellenarlas (cont.)

3.2. Forma de rellenar las tablas:

□ Inicializar los casos base:

$B[p, 0] := 0; B[0, m] := 0$ /* inicializamos la fila 0 y columna 0 de B a 0 */

□ Para todo p desde 1 hasta n

Para todo m desde 1 hasta M , aplicar la ecuación:

$$B[p, m] := \max (B[p-1, m], be_p + B[p-1, m-pe_p])$$

□ El beneficio óptimo es $B[n, M]$

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

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

```
/* La función mochila1 devuelve el beneficio total */
función Mochila1(pe,be:[1..n] de entero; M:entero) devuelve natural;
    devuelve B(n,M)
ffunción Mochila1

/* El algoritmo B rellena un valor de la tabla y lo devuelve */
algoritmo B (n,M: natural) devuelve natural; /* devuelve el valor de B[n,M]*/
    /* Inicializar los casos base */
    para p:=0 hasta n hacer B[p,0]:=0 fpara; /* columna m=0 */
    para m:=0 hasta M hacer B[0,m]:=0 fpara; /* fila p=0 */
    /* resto de los casos B[p, m]:= max(B[p-1, m], bep + B[p-1, m-pep]) */
    para p:=1 hasta n hacer
        para m:=1 hasta M hacer
            si m<pe[p] entonces /*m-pep negativo, caso -∞, y el maximo será el otro término.*/
                B[p,m]:=B[p-1,m]
            sino
                B[p,m]:=max(B[p-1,m], B[p-1,m-pe[p]]+be[p])
        fsi
    fpara
    devuelve B[n,M]
falgoritmo B
```

Algoritmo Iterativo
versión 1

```
int Mochila1(int peso[], int valor[], int n, int M) {
    int B[n+1][M+1];
    for (int p = 0; p <= n; p++) B[p][0]=0; //columna 0
    for (int m = 0; m <= M; m++) B[0][m]=0; //fila 0
    for (int p = 1; p <= n; p++) {
        for (int m = 1; m <= M; m++) {
            if (m < peso[p-1]) B[p][m] = B[p-1][m];
            else
                B[p][m] = std::max(B[p-1][m], B[p-1][m-peso[p-1]] + valor[p-1]);
        }
    }
    return B[n][M];
}
```

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

```
/* La función mochila2 devuelve el beneficio total */
función Mochila2(pe,be:[1..n] de entero; M:entero) devuelve natural;
    devuelve B(n,M)
ffunción Mochila1

/* El algoritmo B rellena un valor de la tabla y lo devuelve */
algoritmo B (n,M: natural) devuelve natural; /* devuelve el valor de B[n,M]*/
    /* casos distintos al base B[p, m]:= max(B[p-1, m], bep + B[p-1, m-pep]) */
    para p:=0 hasta n hacer
        para m:=0 hasta M hacer
            si p=0 OR m=0 entonces /*caso base: fila 0 o columna 0 */
                B[p,m]:=0
            sino si m<pe[p] entonces /*m-pep negativo, caso -∞, y maximo será el otro término.*/
                B[p,m]:=B[p-1,m]
            sino
                B[p,m]:=max(B[p-1,m], B[p-1,m-pe[p]]+be[p])
        fsi
    fpara
    fpara
    devuelve B[n,M]
falgoritmo B

Algoritmo Iterativo
versión 2
```

```
int Mochila2(int peso[], int valor[], int n, int M) {
    int B[n+1][M+1];
    for (int p = 0; p <= n; p++) {
        for (int m = 0; m <= M; m++) {
            if (p == 0 || m == 0) B[p][m] = 0;
            else if (m < peso[p-1]) B[p][m] = B[p-1][m];
            else
                B[p][m] = std::max(B[p-1][m], B[p-1][m-peso[p-1]] + valor[p-1]);
        }
    }
    return B[n][M];
}
```

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

```
/* La función mochila3 devuelve el beneficio total */
función Mochila3(pe,be:[1..n] de entero; M:entero) devuelve natural;
    devuelve B(n,M)
ffunción Mochila1

/* El algoritmo B rellena un valor de la tabla y lo devuelve */
algoritmo B (n,M: natural) devuelve natural; /* devuelve el valor de B[n,M]*/
    /* Inicializar los casos base */
    para m:=0 hasta M hacer B[0,m]:=0 fpara; /* fila p=0 */
    /* resto de los casos B[p, m]:= max(B[p-1, m], bep + B[p-1, m-pep]) */
    para p:=1 hasta n hacer
        para m:=0 hasta pe[p]-1 hacer
            B[p,m]:=B[p-1,m]
        fpara
        para m:=pe[p] hasta M hacer
            B[p,m]:=max(B[p-1,m], B[p-1,m-pe[p]]+be[p])
        fpara
    fpara
    devuelve B[n,M]
falgoritmo B
```

Algoritmo Iterativo
versión 3

```
int Mochila3(int peso[], int valor[], int n, int M) {
    int B[n+1][M+1];
    for (int m = 0; m <= M; m++) B[0][m]=0; //fila 0
    for (int p = 1; p <= n; p++) {
        for (int m = 0; m < peso[p-1]; m++) B[p][m] = B[p-1][m];
        for (int m = peso[p-1]; m <= M; m++)
            B[p][m] = std::max(B[p-1][m], B[p-1][m-peso[p-1]] + valor[p-1]);
    }
    return B[n][M];
}
```

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

Paso 3. Ejemplo. $n= 3$, $M= 6$, $pe= (2, 3, 4)$, $be= (1, 2, 5)$

$$(B[p, m] := \max (B[p-1, m], B[p-1, m-pe_p] + be_p))$$

m

p	B	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

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

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

Paso 4. Recomponer la solución óptima a partir de la tabla

- Cálculos posibles a partir de la tabla B:
 - **beneficio** total: $B[n, M]$
 - los **objetos** metidos en la mochila:
- $B[n, M]$ almacena el beneficio óptimo, pero ¿cuál son los objetos que se cogen en esa solución?
- Obtener la tupla solución (x_1, x_2, \dots, x_n) usando **B**.
- **Proceso:** partiendo de la posición $B[n, M]$, analizar las decisiones que se tomaron para cada objeto **p**.
 - Si $B[p, m] = B[p-1, m]$, entonces la solución **no** usa el objeto **p** $\Rightarrow x_p = 0$
 - Si $B[p, m] = B[p-1, m - pe_p] + be_p$, entonces **sí** se usa el objeto **p** $\Rightarrow x_p = 1$
 - Si se cumplen ambas, entonces podemos usar el objeto **p** o no (existe más de una solución óptima).

4. Programación dinámica. Ejemplos de aplicación. Problema de la mochila 0-1.

Paso 4. Reconstruir la solución óptima (cont.). Algoritmo.

```

función Objetos(M:natural; B[0..n][0..M] de natural be,pe[1..n] de natural)
    test(n,M)
ffunciónObjetos
algoritmo test(n,M: natural)
    variables x:[1..n] de {0,1}
        m:= M
        para p:= n hasta 1 (dec 1) hacer
            si B[p, m] == B[p-1, m] entonces
                x[p]:= 0
            sino
                /* B[p, m] == B[p-1, m-pep] + bep */
                x[p]:= 1
                m:= m - pep
        fsi
    fpara
ffuncióntest

```

```

int B[n+1][M+1]; //variable global (se supone ya rellena)
void Objetos(int peso[], int valor[], int n, int M, int X[]) {
    int m = M;
    for (int p = n; p>=1; p--) {
        if (B[p][m] == B[p-1][m])
            X[p-1]=0;
        else {
            X[p-1]=1;
            m=m-peso[p-1];
        }
    }
}

```

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

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

□ **Análisis de eficiencia.**

- Cada componente de la tabla B 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 p , 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 pe_p o la capacidad M son reales, esta solución no sirve.

4. Programación dinámica. Ejemplos de aplicación. Problema de la mochila 0-1.

□ Ejercicio 1

Mochila de peso $M=15$

Número de objetos $n=3$

Valor de los objetos

$\text{int valor}[] = \{38, 40, 24\};$

Peso de los objetos

$\text{int peso}[] = \{ 9, 6, 5\};$

```
int Mochila1(int peso[], int valor[], int n, int M) {
    int B[n+1][M+1];
    for (int p = 0; p <= n; p++) B[p][0]=0; //columna 0
    for (int m = 0; m <= M; m++) B[0][m]=0; //fila 0
    for (int p = 1; p <= n; p++) {
        for (int m = 1; m <= M; m++) {
            if (m < peso[p-1]) B[p][m] = B[p-1][m];
            else
                B[p][m] = std::max(B[p-1][m], B[p-1][m-peso[p-1]] + valor[p-1]);
        }
    }
    return B[n][M];
}
```

B	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
$p_1=9$	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

$$(B[p, m] := \max (B[p-1, m], B[p-1, m-\text{peso}_p] + \text{valor}_p))$$

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

4. Programación dinámica. Ejemplos de aplicación. **Problema de la mochila 0-1.**

❑ Ejercicio 2

Mochila de peso $M=11$

Número de objetos $n=5$

Solución óptima: {3,4}

Objeto	Valor	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

```
int Mochila2(int peso[], int valor[], int n, int M) {
    int B[n+1][M+1];
    for (int p = 0; p <= n; p++) {
        for (int m = 0; m <= M; m++) {
            if (p == 0 || m == 0)        B[p][m] = 0;
            else if (m < peso[p-1])      B[p][m] = B[p-1][m];
            else
                B[p][m] = std::max(B[p-1][m], B[p-1][m-peso[p-1]] + valor[p-1]);
        }
    }
    return B[n][M];
}
```

		m											
B		0	1	2	3	4	5	6	7	8	9	10	11
p	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	1	1	1	1	1	1	1	1	1	1	1
	2	0	1	6	7	7	7	7	7	7	7	7	7
	3	0	1	6	7	7	18	19	24	25	25	25	25
	4	0	1	6	7	7	18	22	24	28	29	29	40
	5	0	1	6	7	7	18	22	28	29	34	35	40

4. Programación dinámica. Ejemplos. **Problema del cambio de monedas.**

- **Problema:** Dado un conjunto de n tipos de monedas, cada una con valor v_i , y dada una cantidad C , encontrar el número **mínimo** de monedas que tenemos que usar para obtener esa cantidad.
- **Datos del problema:**
 - C : cantidad para cambiar.
 - n : número de tipos de monedas disponibles. Supondremos una cantidad ilimitada.
 - $V = (v_1, v_2, \dots, v_n)$ valor de cada tipo de monedas.
- **Representación de la solución:**
 - Una solución será de la forma $S = (x_1, x_2, \dots, x_n)$, $x_i \geq 0$, donde x_i es el número de monedas de tipo i . Suponemos que la moneda i vale v_i .
- **Formulación matemática:** **Minimizar** $\sum_{i=1}^n x_i$, sujeto a $\sum_{i=1}^n x_i v_i = C$, $x_i \geq 0$
- **Solución:** conjunto de monedas que suman la cantidad C .
- **Ejemplo:** $C = 3.89$; $V = \{1, 2, 5, 10, 20, 50, 100, 200\}$; $X = \{0, 2, 1, 1, 1, 1, 1, 1\}$

4. Programación dinámica. Ejemplos. **Problema del cambio de monedas.**

- 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. Plantear la solución como una sucesión de decisiones y verificar Bellman

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

4. Programación dinámica. Ejemplos. **Problema del cambio de monedas.**

Paso 2. Obtener una **descomposición recursiva** (Ecuación recurrente y caso base)

- **Ecuación recurrente del problema. Cambio(i, c: entero): entero**
Problema del cambio de monedas, considerando sólo los i primeros tipos, con cantidad a devolver c . **Devuelve el número mínimo de monedas necesario.**
- **Definición de Cambio(i, c: entero): entero**
 - **Si no se coge ninguna moneda de tipo i :**
 $\text{Cambio}(i, c) = \text{Cambio}(i - 1, c)$
 - **Si se coge 1 moneda de tipo i :**
 $\text{Cambio}(i, c) = 1 + \text{Cambio}(i, c - v_i)$
 - **Valor óptimo:** el que use menos monedas:
 $\text{Cambio}(i, c) = \min \{ \text{Cambio}(i - 1, c), 1 + \text{Cambio}(i, c - v_i) \}$
- **Casos base:**
 - Si $c=0$, no usar ninguna moneda: $\text{Cambio}(i, 0) = 0$
 - En otro caso, si $c < 0$ ó $i \leq 0$, no se puede resolver el problema:
 $\text{Cambio}(i, c) = +\infty$

4. Programación dinámica. Ejemplos. **Problema del cambio de monedas.**

Paso 2. Obtener una **descomposición recursiva** (Ecuación recurrente y caso base) (cont.)

□ **Ecuación recurrente:**

$$\text{Cambio}(i, c) = \begin{cases} 0 & \text{Si } c=0 \\ +\infty & \text{Si } c < 0 \text{ ó } i \leq 0 \\ \min \{ \text{Cambio}(i-1, c), 1 + \text{Cambio}(i, c-v_i) \} & \end{cases}$$

Paso 3. Definición de las **tablas** y cómo **rellenarlas**

3.1. Dimensiones y tamaño de la tabla

□ Matriz **D** $\rightarrow D[i, c] = \text{Cambio}(i, c)$

□ **D:** array $[0..n, 0..C]$ de entero \rightarrow **tamaño** D: $(n+1) \times (C+1)$

3.2. Forma de rellenar la tabla

para $i := 0$ hasta n hacer $D[i, 0] := 0$ /* inicializamos la columna 0 a 0 */

para $c := 1$ hasta C hacer $D[0, c] := +\infty$ /* inicializamos la fila 0 a $+\infty$ */

para $i := 1$ hasta n hacer

para $c := 1$ hasta C hacer

$D[i, c] := \min(D[i-1, c], 1 + D[i, c-v_i])$

devolver $D[n, C]$

Si $c-v_i$ es negativo, entonces es el caso $+\infty$, y el mínimo será siempre el otro término.

4. Programación dinámica. Problema del cambio de monedas. Algoritmo iterativo.

```
/* La función Cambio1 devuelve el número mínimo de monedas necesario */
función Cambio1(v:[1..n] de entero; C:entero) devuelve natural;
    devuelve D(n,C)
ffunción Cambio1

algoritmo D (n,C: natural) devuelve natural; /* devuelve el valor de D[n,C]*/
    /* Inicializar caso base */
    para i:=0 hasta n hacer D[i,0]:=0 fpara; /* col c=0 */
    para c:=1 hasta C hacer D[0,c]:=+∞ fpara; /* fila i=0 */
    /* resto de los casos D[i, c]:= min(D[i-1, c], 1 + D[i, c-vi]) */
    para i:=1 hasta n hacer
        para c:=1 hasta C hacer
            si c<v[i] entonces /*c-vi es negativo, caso +∞, y el mínimo será el otro término.*/
                D[i,c]:=D[i-1,c]
            sino
                D[i,c]:=min( D[i-1,c], 1 + D[i,c-v[i]] )
        fsi
    fpara
    fpara
    devuelve D[n,C]
falgoritmo D

Algoritmo Iterativo
versión 1

int Cambio1(int valor[], int n, int C) {
    int D[n+1][C+1];
    for (int i = 0; i <= n; i++) D[i][0]=0; //columna 0 a 0
    for (int c = 1; c <= C; c++) D[0][c]=1e9; //fila 0 a infinito
    for (int i = 1; i <= n; i++) {
        for (int c = 1; c <= C; c++) {
            if (c < valor[i-1]) D[i][c] = D[i-1][c];
            else
                D[i][c] = std::min( D[i-1][c], 1 + D[i][c-valor[i-1]] );
        }
    }
    return D[n][C];
}
```

4. Programación dinámica. Problema del cambio de monedas. Algoritmo iterativo.

```
/* La función Cambio2 devuelve el número mínimo de monedas necesario */
función Cambio2(v:[1..n] de entero; C:entero) devuelve natural;
    devuelve D(n,C)
ffunción Cambio2

algoritmo D (n,C: natural) devuelve natural; /* devuelve el valor de D[n,C]*/
/* casos distintos al base D[i, c]:= min(D[i-1, c], 1 + D[i, c-vi]) */
para i:=0 hasta n hacer
    para c:=0 hasta C hacer
        si i=0 AND c<>0 entonces D[i,c]:=+∞ /*caso base: fila 0 */
        sino si c=0 entonces D[i,c]:=0 /*caso base: col 0 */
        sino si c<v[i] entonces /*c-vi negativo, caso +∞, el mínimo será el otro término.*/
            D[i,c]:=D[i-1,c]
        sino
            D[i,c]:=min( D[i-1,c], 1 + D[i,c-v[i]] )
    fsi
fpara
fpara
    devuelve D[n,C]
falgoritmo D
```

Algoritmo Iterativo
versión 2

```
int Cambio2(int valor[], int n, int C) {
    int D[n+1][C+1];
    for (int i = 0; i <= n; i++) {
        for (int c = 0; c <= C; c++) {
            if (i == 0 && c != 0) D[i][c] = 1e9; //infinito
            else if (c == 0) D[i][c] = 0;
            else if (c < valor[i-1]) D[i][c] = D[i-1][c];
            else
                D[i][c] = std::min( D[i-1][c], 1 + D[i][c-valor[i-1]] );
        }
    }
    return D[n][C];
}
```

4. Programación dinámica. Problema del cambio de monedas. Algoritmo iterativo.

```
/* La función Cambio3 devuelve el número mínimo de monedas necesario */
función Cambio3(v:[1..n] de entero; C:entero) devuelve natural;
    devuelve D(n,C)
ffunción Cambio3

algoritmo D (n,C: natural) devuelve natural; /* devuelve el valor de D[n,C]*/
    /* Inicializar caso base */
    D[0,0]:=0
    para c:=1 hasta C hacer D[0,c]:=+∞ fpara; /* fila i=0 */
    /* resto de los casos D[i, c]:= min(D[i-1, c], 1 + D[i, c-vi]) */
    para i:=1 hasta n hacer
        para c:=0 hasta v[i]-1 hacer
            D[i,c]:=D[i-1,c]
        fpara
        para c:=v[i] hasta C hacer
            D[i,c]:=min( D[i-1,c], 1 + D[i,c-v[i]] )
        fpara
    fpara
    devuelve D[n,C]
falgoritmo D
```

Algoritmo Iterativo
versión 3

```
int Cambio3(int valor[], int n, int C) {
    int D[n+1][C+1];
    D[0][0]=0;
    for (int c = 1; c <= C; c++) C[0][c]=1e9; //infinito
    for (int i = 1; i <= n; i++) {
        for (int c = 0; c < valor[i-1]; c++) D[i][c] = D[i-1][c];
        for (int c = valor[i]; c <= C; c++)
            D[i][c] = std::min( D[i-1][c], 1 + D[i][c-valor[i-1]] );
    }
    return D[n][C];
}
```

4. Programación dinámica. Ejemplos. Problema del cambio de monedas.

□ **Ejemplo.** $n=3$, $C=8$, $v=(1, 4, 6)$

$$D[i, c] = \min(D[i-1, c], 1 + D[i, c-v_i]) \quad +\infty \quad \text{Si } col < 0 \text{ ó } fila \leq 0$$

D	0	1	2	3	4	5	6	7	8
0 $c_0=0$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
1 $c_1=1$	0	1	2	3	4	5	6	7	8
2 $c_2=4$	0	1	2	3	1	2	3	4	2
3 $c_3=6$	0	1	2	3	1	2	1	2	2

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

4. Programación dinámica. Ejemplos. **Problema del cambio de monedas.**

Paso 4. Recomponer la solución óptima a partir de la tabla

- ¿Cómo calcular cuántas monedas de cada tipo deben usarse, es decir, la tupla solución (x_1, x_2, \dots, x_n) ?
- Analizar las decisiones tomadas en cada celda, empezando en **$D[n, C]$** .
- ¿Cuál fue el mínimo en cada **$D[i, c]$** ?

Si **$D[i, c]$** es igual a:

- **$D[i - 1, c]$** → No utilizar ninguna moneda más de tipo **i** . ($i=i-1$)
- **$D[i, c - v[i]] + 1$** → Usar una moneda más de tipo **i** . ($c=c-v[i]$)
- Si se cumplen ambas, entonces podemos usar una moneda más de tipo **i** o no (existe más de una solución óptima)

4. Programación dinámica. Ejemplos. Problema del cambio de monedas.

Paso 4. Recomponer la solución óptima a partir de la tabla (cont.)

□ Implementación:

función Monedas(M:natural; D[1..n][0..C] de natural v[1..n] de natural)

test(n,C)

ffunciónObjetos

algoritmo test(n,C: natural)

x: array [1..n] de entero /* x[i] = número de monedas usadas de tipo i */

x:= (0, 0, ..., 0)

i:= n

j:= C

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 - v_i

finsi

finmientras

□ **Ejemplo.** n= 3, C= 8, v= (1, 4, 6)

■ **solución óptima** es: (x₁,x₂,x₃)=(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? válida?

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 ₃ =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

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

□ Problema:

Comparar dos cadenas y encontrar la subsecuencia común de mayor longitud

Ejemplo:

Cadenas:

X = (A B C B D A B),

Y = (B D C A B A),

Subsecuencia común de mayor longitud:

LCS = (B C B A)

X = (A **B** **C** **B** D A B)

Y = (**B** D **C** A **B** **A**)

- Un algoritmo exhaustivo compararía cualquier subsecuencia X con los símbolos de Y:
 - Si $|X|=m$, $|Y|=n$, hay 2^m subsecuencias de X que debemos comparar con Y (n comparaciones) → algoritmo exhaustivo de orden **$O(n2^m)$** .
- LCS verifica el principio de optimalidad de Bellman:
 - Cualquier subsecuencia de la secuencia óptima es también secuencia óptima.

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 1. Plantear la **solución como una sucesión de decisiones** y verificar **Bellman**

- Si buscamos la LCS para pares de prefijos de X e Y, podemos calcular la solución de nuestro problema.
- Dados los prefijos X_i y Y_j , ¿cogemos o no cogemos el siguiente carácter?

Paso 2. Obtener una **descomposición recursiva** (Ecuación recurrente y caso base)

X_i Prefijo de X de longitud i.

Y_j Prefijo de Y de longitud j.

$c(i,j)$ Longitud de la LCS para X_i e Y_j .

$$c(i, j) = \begin{cases} c(i-1, j-1) + 1 & \text{si } x[i] = y[j] \\ \max\{c(i-1, j), c(i, j-1)\} & \text{en otro caso} \end{cases}$$

La longitud de la subsecuencia de mayor longitud (LCS) de X e Y será **$c(m,n)$** , con $|X|=m$, $|Y|=n$.

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 2. Obtener una **descomposición recursiva** (Ecuación recurrente y caso base) (cont.)

Definición recursiva de la solución:

- Caso base:
 - $c(0,0) = 0$ (Subcadena vacía)
 - $c(0,j) = c(i,0) = 0$ (LCS de la cadena vacía y cualquier otra cadena)
- Cálculo recursivo:
 - Primer caso ($x[i]=y[j]=s$): Se emparejan ambos símbolos y la LCS aumenta de longitud:
 $LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) + \{s\}.$
 - Segundo caso ($x[i] \neq y[j]$): No se pueden emparejar los símbolos $x[i]$ e $y[j]$, por lo que $LCS(X_i, Y_j)$ será la mejor entre $LCS(X_{i-1}, Y_j)$ y $LCS(X_i, Y_{j-1})$.

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 3. Definición de las **tablas** y cómo **rellenarlas**

3.1. Dimensiones y tamaño de la tabla

- Matriz **C** $\rightarrow C[i, j] = \text{LCD}(i, j)$
- **D**: array $[0..m, 0..n]$ de entero \rightarrow **tamaño** D: $(m+1) \times (n+1)$
- Dadas las cadenas X e Y, $|X|=m$, $|Y|=n$

3.2. Forma de rellenar la tabla

```
para i:= 0 hasta m hacer C[i, 0]:= 0      /* inicializamos la columna 0 a 0 */
para j:= 1 hasta n hacer C[0, j]:= 0      /* inicializamos la fila 0 a 0 */
para i:= 1 hasta m hacer
    para j:= 1 hasta n hacer
        si X[ i ] = Y[ j ] entonces
            C[i, j]:= C[i-1, j-1] + 1
        sino
            C[i, j]:= max ( C[i-1, j], C[i, j-1] )
    fsi
fpara
fpara
devolver C[m, n]
```

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 3. Implementación iterativa del algoritmo

```
int[][] LCS (X, Y)
{
    for (i=0; i<=X.length; i++) c[i][0]=0;           // Y0
    for (j=1; j<=Y.length; j++) c[0][j]=0;           // X0

    for (i=1; i<=X.length; i++)                       // Xi
        for (j=1; j<=Y.length; j++)                   // Yj
            if ( X[i] == Y[j] )
                c[i][j] = c[i-1][j-1] + 1;
            else
                c[i][j] = max ( c[i-1][j], c[i][j-1] );

    return c;
}
```

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 3. Implementación iterativa del algoritmo

Ejemplo:

$X = (A \ B \ C \ B),$
 $Y = (B \ D \ C \ A \ B),$

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0					
B	2	0					
C	3	0					
B	4	0					

```
for (i=0; i<=X.length; i++) c[i][0]=0;           //  $Y_0$   
for (j=1; j<=T.length; j++) c[0][j]=0;          //  $X_0$ 
```

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 3. Implementación iterativa del algoritmo

Ejemplo:

$X = (\text{A B C B}) ,$
 $Y = (\text{B D C A B}) ,$

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	0	0	0		
B	2	0					
C	3	0					
B	4	0					

```
for (i=1; i<=X.length; i++)           //  $X_i$ 
    for (j=1; j<=Y.length; j++)       //  $Y_j$ 
        if ( X[i] == Y[j] )
            c[i][j] = c[i-1][j-1] + 1;
        else
            c[i][j] = max ( c[i-1][j], c[i][j-1] );
```

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 3. Implementación iterativa del algoritmo

Ejemplo:

$X = (\text{A B C B}) ,$
 $Y = (\text{B D C A B}) ,$

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	0	0	0	1	
B	2	0					
C	3	0					
B	4	0					

```
for (i=1; i<=X.length; i++)           //  $X_i$ 
    for (j=1; j<=Y.length; j++)       //  $Y_j$ 
        if ( X[i] == Y[j] )
            c[i][j] = c[i-1][j-1] + 1;
        else
            c[i][j] = max ( c[i-1][j], c[i][j-1] );
```

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 3. Implementación iterativa del algoritmo

Ejemplo:

$X = (\text{A B C B}) ,$
 $Y = (\text{B D C A B}) ,$

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	0	0	0	1	1
B	2	0					
C	3	0					
B	4	0					

```
for (i=1; i<=X.length; i++)           //  $X_i$ 
    for (j=1; j<=Y.length; j++)       //  $Y_j$ 
        if ( X[i] == Y[j] )
            c[i][j] = c[i-1][j-1] + 1;
        else
            c[i][j] = max ( c[i-1][j], c[i][j-1] );
```

4. Programación dinámica. Ejemplos. Problema de la subsecuencia común de mayor longitud.

Paso 3. Implementación iterativa del algoritmo

Ejemplo:

$X = (A \ B \ C \ B),$
 $Y = (B \ D \ C \ A \ B),$

$LCS = (B \ C \ B)$

$X = (\ A \ \textcolor{red}{B} \ \textcolor{red}{C} \ \textcolor{red}{B} \)$
 $Y = (\ \textcolor{red}{B} \ D \ \textcolor{red}{C} \ A \ \textcolor{red}{B} \)$

		Y_j	B	D	C	A	B
		0	1	2	3	4	5
X_i	0	0	0	0	0	0	0
A	1	0	0	0	0	1	1
B	2	0	1	1	1	1	2
C	3	0	1	1	2	2	2
B	4	0	1	1	2	2	3

```
for (i=1; i<=X.length; i++)           //  $X_i$ 
    for (j=1; j<=Y.length; j++)       //  $Y_j$ 
        if ( X[i] == Y[j] )
            c[i][j] = c[i-1][j-1] + 1;
        else
            c[i][j] = max ( c[i-1][j], c[i][j-1] );
```


4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

□ **Problema:**

Caminos mínimos entre todos los pares de nodos de un grafo

Cálculo del camino de coste mínimo entre todos los pares de vértices de un grafo dirigido sin pesos negativos.

□ Posibles soluciones:

- Algoritmo Exhaustivo (de orden exponencial)
- Aplicar el algoritmo voraz de Dijkstra **para cada vértice.**
- Algoritmo de Floyd (programación dinámica).

4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

□ Algoritmo de Floyd :

Paso 1. Plantear la solución como una sucesión de decisiones y verificar Bellman

□ Principio de optimalidad:

■ Si $i_1, i_2, \dots, i_k, i_{k+1}, \dots, i_n$ es un camino de coste mínimo de i_1 a i_n , entonces:

□ i_1, i_2, \dots, i_k es un camino de coste mínimo de i_1 a i_k , y

□ i_k, i_{k+1}, \dots, 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 .

□ **Razonamiento inductivo:** para calcular los caminos mínimos pudiendo pasar por los k primeros nodos usamos los caminos mínimos pasando por los $k-1$ primeros. Camino óptimo de i a j que pasa por los k 1º vértices del grafo 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 $k-1$:

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

4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

Paso 2. Obtener una descomposición recursiva (Ecuación recurrente y caso base)

- $C(i,j)$: **coste** de la arista (i,j) o **infinito** si esa arista no existe.
- $C(i,i)=0$.
- $D_k(i,j)$: camino de coste mínimo de i a j usando sólo los k primeros vértices del grafo como puntos intermedios (no pasa por ningún vértice de índice $> k$).
- Sea $D(i,j)$ la longitud del camino de coste mínimo de i a j .
- Se cumple que:

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

$$D_0(i,j) = C(i, j) \quad 1 \leq i \leq n, 1 \leq j \leq n$$

- **ecuación recurrente** que define el método de programación dinámica.
 - $D_k(i, j)$: camino mínimo de i a j pudiendo pasar por los nodos $1, 2, \dots, k$.

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

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

4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

Paso 3. Definición de las **tablas y cómo **rellenarlas****

- Matriz **D** $\rightarrow D[i, j] = D_n(i, j)$
 - **D: array [1..n, 1..n] de nat**
 - para** $i := 1$ **hasta** n **hacer** $D[i, i] := 0$
 - para** $j := 1$ **hasta** n **hacer**
 - $D_0(i, j) = C(i, j)$ /* ∞ si no hay arco */
 - para** $k := 1$ **hasta** n **hacer**
 - para** $i := 1$ **hasta** n **hacer**
 - para** $j := 1$ **hasta** n **hacer**
 - $D_k(i, j) := \min(D_{k-1}(i, j), D_{k-1}(i, k) + D_{k-1}(k, j))$
 - devolver** D_n
- Usamos una matriz C con los pesos de las aristas, C (i,j) peso la arista (i,j).
 - **C(i,i)=0** y C(i,j)= **infinito** si esa arista no existe
 - La idea es encontrar una matriz D de n x n elementos, en la que cada elemento D(i,j) almacene el coste mínimo del camino entre los vértices i y j
 - Para calcular el camino mínimo entre los vértices i y j hay dos posibilidades:
 - no pasar por el vértice k (tendremos que calcular el mejor camino con el resto de vértices $D_{k-1}[i,j]$)
 - pasar por k (tendremos que obtener caminos que vayan de i a k y de k a j)

4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

Paso 3. Definición de las tablas y cómo rellenarlas (cont)

- El algoritmo comienza con una inicialización de D (D_0) y se genera iterativamente la secuencia de matrices D_1, D_2, \dots, D_n , cuyos elementos significan:
 - $D_0[i, j] = C[i, j]$ peso asociado a la arista del vértice i al vértice j
 - $D_1[i, j] = \min (D_0[i, j], D_0[i, 1] + D_0[1, j])$. Menor de los costes entre el anterior camino desde i hasta j y la suma de los costes de caminos desde i hasta 1 y 1 hasta j
 - $D_2[i, j] = \min (D_1[i, j], D_1[i, 2] + D_1[2, j])$. Menor de los costes entre el anterior camino desde i hasta j y la suma de los costes de caminos desde i hasta 2 y 2 hasta j
 - $D_n[i, j] = \min (D_{n-1}[i, j], D_{n-1}[i, n] + D_{n-1}[n, j])$. Menor de los costes entre el anterior camino desde i hasta j y la suma de los costes de caminos desde i hasta n y n hasta j

4. Programación dinámica. Ejemplos. Problema del camino mínimo. Algoritmo de Floyd.

```
{Pre: g es un grafo dirigido etiquetado sin pesos 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,  $D_0(i,j)=C(i,j)$   $1 \leq i \leq n$ ,  $1 \leq j \leq n$ ; 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) /*  $\infty$  si no hay arco */
    fpara;
    D[i,i]:=0 /* ó D[i,i]:="-" */
fpara;
    /* resto de los casos  $D_k(i,j):=\min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$  */
para k=1 hasta n hacer /*el índice k indica el subíndice de la matriz D que se está generando */
    para i=1 hasta n hacer /* el índice i indica la fila */
        para j=1 hasta n hacer /* el índice j indica la columna */
            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
ffunción Floyd
{Post: D=caminosMínimos(g)}
```

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

4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

□ Cálculo posible a partir de la tabla D :

■ **Caminos mínimos** entre todos los pares de nodos : $D_n(i, j)$

Paso 4. Recomponer la solución óptima a partir de la tabla

□ ¿Cómo calcular la **ruta asociada del camino mínimo** entre dos vértices, es decir, la tupla solución (x_i, \dots, 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

■ $C[i, j] = k \Rightarrow$ El nodo k es el predecesor(padre) de j y forma parte del camino mínimo entre i y j .

■ $C[i, j] = \infty \Rightarrow$ 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.

4. Programación dinámica. Ejemplos. Problema del camino mínimo. Algoritmo de Floyd.

```
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,  $D_0(i,j)=C(i,j)$  */
  /* 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
      D[i,j]:=etiqueta(g,i,j) /*  $\infty$  si no hay arco */
      C[i,j]:=i; /*  $\infty$  si no hay arco */
    fpara;
    D[i,i]:=0 /* ó  $D[i,i]:="-"$  */
    C[i,i]:=0; /* ó  $C[i,i]:="-"$  */
  fpara;
  /* resto de los casos  $D_k(i,j):=\min(D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j))$  */
  para k=1 hasta n hacer /*el índice k indica el subíndice de la matriz D que se está generando */
    para i=1 hasta n hacer /* el índice i indica la fila */
      para j=1 hasta n hacer /* el índice j indica la columna */
        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
    fsi
  devuelve D,C;
Ffunción Floyd
```

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

4. Programación dinámica. Ejemplos. Problema del camino mínimo. Algoritmo de Floyd.

Paso 4. Recomponer la solución óptima 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]  $\neq \infty$ ) AND (i  $\neq$  j) AND (C[i,j]  $\neq$  i) entonces
    x[k]:= j; /* j es el último nodo del camino de i a j */
    mientras C[i,j]  $\neq$  i hacer
      x[k]:= C[i,j]; /*C[i,j]=m, el nodo predecesor al j es el m  $\Rightarrow m \rightarrow j$  */
      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
```

4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

☐ **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 "-".

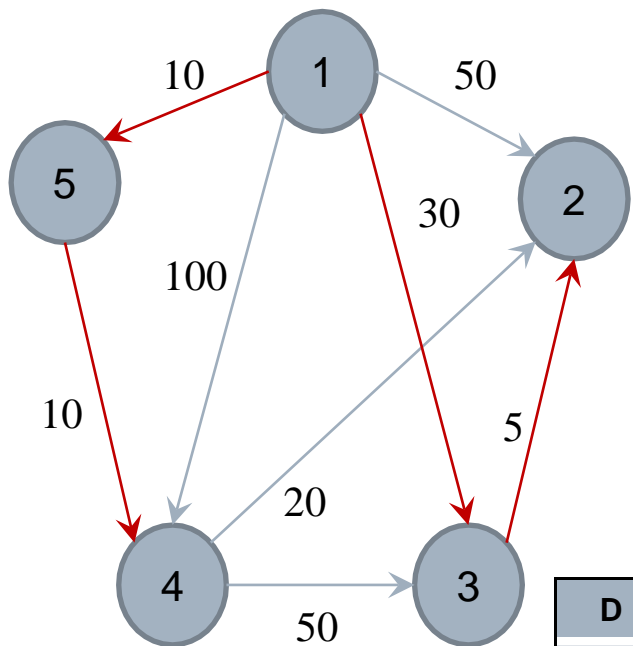
4. Programación dinámica. Ejemplos. Problema del camino mínimo. **Algoritmo de Floyd.**

□ **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 $(D[i, k] + D[k, j]) < D[i, j] \Rightarrow C[i, j] = C[k, j]$ y $D[i, j] = D[i, k] + D[k, j]$
 - 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 $D[i, j] = \infty$ 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	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: Inicialización de las matrices de costes D y de los caminos mínimos C.

4. Algoritmo de Floyd. Ejemplo.

3. Tomamos $k=1$:

3.1 Se selecciona la fila y la columna $k=1$ de la matriz D y entonces, para i y j , con $i \neq k$, $j \neq k$ e $i \neq j$, hacemos:

Si $(D[i, k] + D[k, j]) < D[i, j] \Rightarrow C[i, j] = C[k, j]$ y $D[i, j] = D[i, k] + D[k, j]$, en caso contrario, dejamos las matrices como están.

□ Tomamos $i=2$ ($i \neq k$):

■ $j=3$ ($j \neq k, j \neq i$ ($D[2,3]$)): $D[2,1] + D[1,3] = \infty + 30$, no hay que cambiar nada, no podemos llegar de 2 a 3 a través de 1.

■ $j=4$ ($j \neq k, j \neq i$ ($D[2,4]$)): $D[2,1] + D[1,4] = \infty + 100 = \infty$, no hay que cambiar nada, no podemos llegar de 2 a 4 a través de 1.

■ $j=5$ ($j \neq k, j \neq i$ ($D[2,5]$)): $D[2,1] + D[1,5] = \infty + 10 = \infty$, no hay que cambiar nada, no podemos llegar de 2 a 5 a través de 1.

□ Tomamos $i=3$ ($i \neq k$): como $D[3,1] = \infty$, entonces no habrá ningún cambio, no puede haber ningún camino desde 3 a través de 1.

□ Tomamos $i=4$ ($i \neq k$): en este caso ocurre como en el paso anterior, como $D[4,1] = \infty$, entonces no habrá ningún cambio, no puede haber ningún camino desde 4 a través de 1.

□ Tomamos $i=5$ ($i \neq k$), como $D[5,1] = \infty$, 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**: Se selecciona la fila y la columna **k=2** de la matriz **D**

□ Tomamos **i=1** ($i \neq k$):

- $j=3$ ($j \neq k, j \neq i$ ($D[1,3]$)): $D[1,2]+D[2,3]=50+\infty > D[1,3]=30$, , por tanto no hay que cambiar nada, el camino es mayor que el existente.
- $j=4$ ($j \neq k, j \neq i$ ($D[1,4]$)): $D[1,2]+D[2,4]=50+\infty > D[1,4]=100$, , por tanto no hay que cambiar nada, el camino es mayor que el existente.
- $j=5$ ($j \neq k, j \neq i$ ($D[1,5]$)): $D[1,2]+D[2,5]=50+\infty = \infty$, no hay que cambiar nada, no podemos llegar de 1 a 5 a través de 2.

□ Tomamos **i=3** ($i \neq k$):

- $j=1$ ($j \neq k, j \neq i$ ($D[3,1]$)): $D[3,2]+D[2,1]=5+\infty$ no hay que cambiar nada, no podemos llegar de 3 a 1 a través de 2.
- $j=4$ ($j \neq k, j \neq i$ ($D[3,4]$)): $D[3,2]+D[2,4]=5+\infty = \infty$, no hay que cambiar nada, no podemos llegar de 3 a 4 a través de 2.
- $j=5$ ($j \neq k, j \neq i$ ($D[3,5]$)): $D[3,2]+D[2,5]=5+\infty = \infty$, no hay que cambiar nada, no podemos llegar de 3 a 5 a través de 2.

□ Tomamos **i=4** ($i \neq k$):

- $j=1$ ($j \neq k, j \neq i$): $D[4,2]+D[2,1]=20+\infty = \infty$, no hay que cambiar nada, no podemos llegar de 4 a 1 a través de 2.
- $j=3$ ($j \neq k, j \neq i$): $D[4,2]+D[2,3]=20+\infty = \infty$, no hay que cambiar nada, no podemos llegar de 4 a 3 a través de 2.
- $j=5$ ($j \neq k, j \neq i$): $D[4,2]+D[2,5]=20+\infty = \infty$, no hay que cambiar nada, no podemos llegar de 4 a 5 a través de 2.

□ Tomamos **i=5** ($i \neq k$), en este caso ocurre como en el paso anterior, como $D[5,2]=\infty$, 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**: Se selecciona la fila y la columna **k=3** de la matriz **D**

□ Tomamos **i=1** ($i \neq k$):

- **j=2** ($j \neq k, j \neq i$): $D[1,3] + D[3,2] = 30 + 5 = 35 < D[1,2] = 50$, por tanto hacemos: **$D[1,2] = 35$ y $C[1,2] = 3$** .
- **j=4** ($j \neq k, j \neq i(D[1,4])$): $D[1,3] + D[3,4] = 30 + \infty > D[1,4] = 100$, por tanto no hay que cambiar nada, el camino es mayor que el existente.
- **j=5** ($j \neq k, j \neq i(D[1,5])$): $D[1,3] + D[3,5] = 30 + \infty > D[1,5] = 10$, por tanto no hay que cambiar nada, el camino es mayor que el existente.

□ Tomamos **i=2** ($i \neq k$): como $D[2,3] = \infty$, entonces no habrá ningún cambio, no puede haber ningún camino desde 2 a través de 3.

□ Tomamos **i=4** ($i \neq k$):

- **j=1** ($j \neq k, j \neq i$): $D[4,3] + D[3,1] = 50 + \infty = \infty$, no hay que cambiar nada, no podemos llegar de 4 a 1 a través de 3.
- **j=2** ($j \neq k, j \neq i$): $D[4,3] + D[3,2] = 50 + 5 = 55 > D[4,2]$, por tanto no hay que cambiar nada, el camino es mayor que el existente.
- **j=5** ($j \neq k, j \neq i$): $D[4,3] + D[3,5] = 50 + \infty = \infty$, no hay que cambiar nada, no podemos llegar de 4 a 5 a través de 3.

□ Tomamos **i=5** ($i \neq k$), como $D[5,3] = \infty$, 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 **$k=4$** : Se selecciona la fila y la columna **$k=4$** de la matriz D

□ Tomamos **$i=1$** ($i \neq k$):

- $j=2$ ($j \neq k, j \neq i$ ($D[1,2]$)): $D[1,4]+D[4,2]=100+20 > D[1,2]=35$, por tanto no hay que cambiar nada, el camino es mayor que el existente.
- $j=3$ ($j \neq k, j \neq i$ ($D[1,3]$)): $D[1,4]+D[4,3]=100+50 > D[1,3]=30$, por tanto no hay que cambiar nada, el camino es mayor que el existente.
- $j=5$ ($j \neq k, j \neq i$ ($D[1,5]$)): $D[1,4]+D[4,5]=100+\infty = \infty > D[1,5]=10$, no hay que cambiar nada, no podemos llegar de 1 a 5 a través de 4.

□ Tomamos **$i=2$** ($i \neq k$): como $D[2,4]=\infty$, no habrá ningún cambio.

□ Tomamos **$i=3$** ($i \neq k$): como $D[3,4]=\infty$, no habrá ningún cambio.

□ Tomamos **$i=5$** ($i \neq k$),

- $j=1$ ($j \neq k, j \neq i$ ($D[5,1]$)): $D[5,4]+D[4,1]=10+\infty$ no hay que cambiar nada, no podemos llegar de 5 a 1 a través de 4.
- $j=2$ ($j \neq k, j \neq i$ ($D[5,2]$)): $D[5,4]+D[4,2]=10+20=30 < D[5,2]=\infty$, \Rightarrow **$D[5,2]=30$ y $C[5,2]=4$** .
- $j=3$ ($j \neq k, j \neq i$ ($D[5,3]$)): $D[5,4]+D[4,3]=10+50=60 < D[5,3]=\infty$, \Rightarrow **$D[5,3]=60$ 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 **$k=5$** : Se selecciona la fila y la columna **$k=5$** de la matriz D

□ Tomamos **$i=1$** ($i \neq k$):

- $j=2$ ($j \neq k, j \neq i$): $D[1,5]+D[5,2]=10+30=80 > D[1,2]=35$, por tanto no hay que cambiar nada, el camino es mayor que el existente.
- $j=3$ ($j \neq k, j \neq i$): $D[1,5]+D[5,3]=10+35=45 > D[1,3]=30$, por tanto no hay que cambiar nada, el camino es mayor que el existente.
- **$j=4$** ($j \neq k, j \neq i(D[1,4])$): $D[1,5]+D[5,4]=10+10=20 < D[1,4]=100$, \Rightarrow **$D[1,4]=20$ y $C[1,4]=5$** .

□ Tomamos **$i=2$** ($i \neq k$): en este caso ocurre como en el paso anterior, como $D[2,5]=\infty$, entonces no puede haber ningún camino desde 2 a través de 5.

□ Tomamos **$i=3$** ($i \neq k$): en este caso ocurre como en el paso anterior, como $D[3,5]=\infty$, entonces no puede haber ningún camino desde 3 a través de 5.

□ Tomamos **$i=4$** ($i \neq k$): como $D[4,5]=\infty$, 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	-

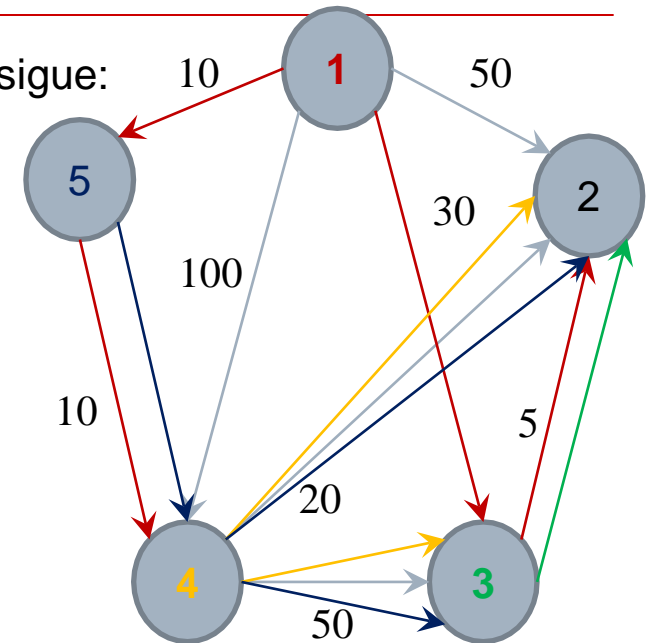
Tablas: Evolución del conjunto D y de los caminos mínimos C

4. Algoritmo de Floyd. Ejemplo.

4. **k = n** \Rightarrow **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.

\Rightarrow Aplicar el/los algoritmos vistos anteriormente para calcular:

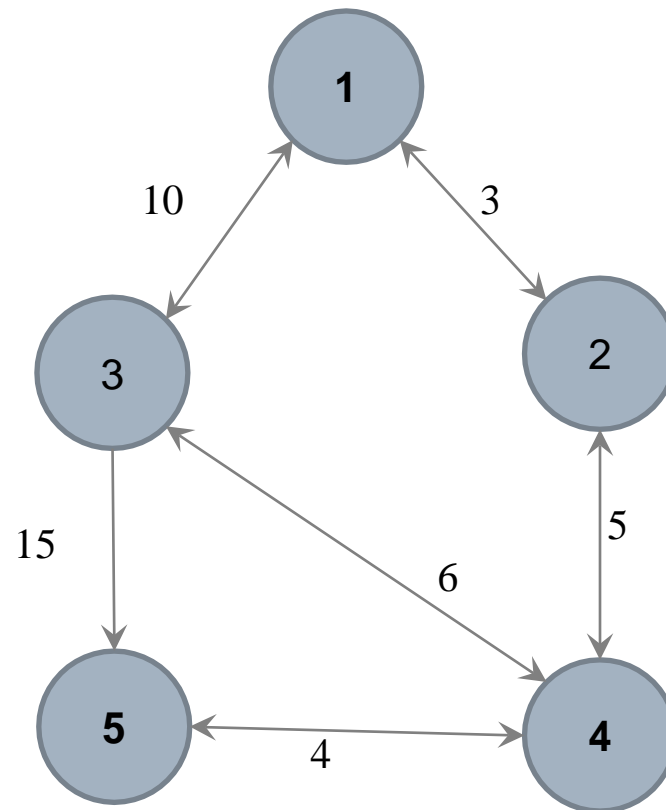
❑ **Distancia más corta:** por ej, la distancia más corta del nodo 1 al nodo 4 es $D[1,4] = 20$.

❑ **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, $5 \rightarrow 4$.
2. Consultar $C[1,5]=1$, por tanto el nodo predecesor al 5 es el 1, es decir, $1 \rightarrow 5 \rightarrow 4$, 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.