Programación Dinámica

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.

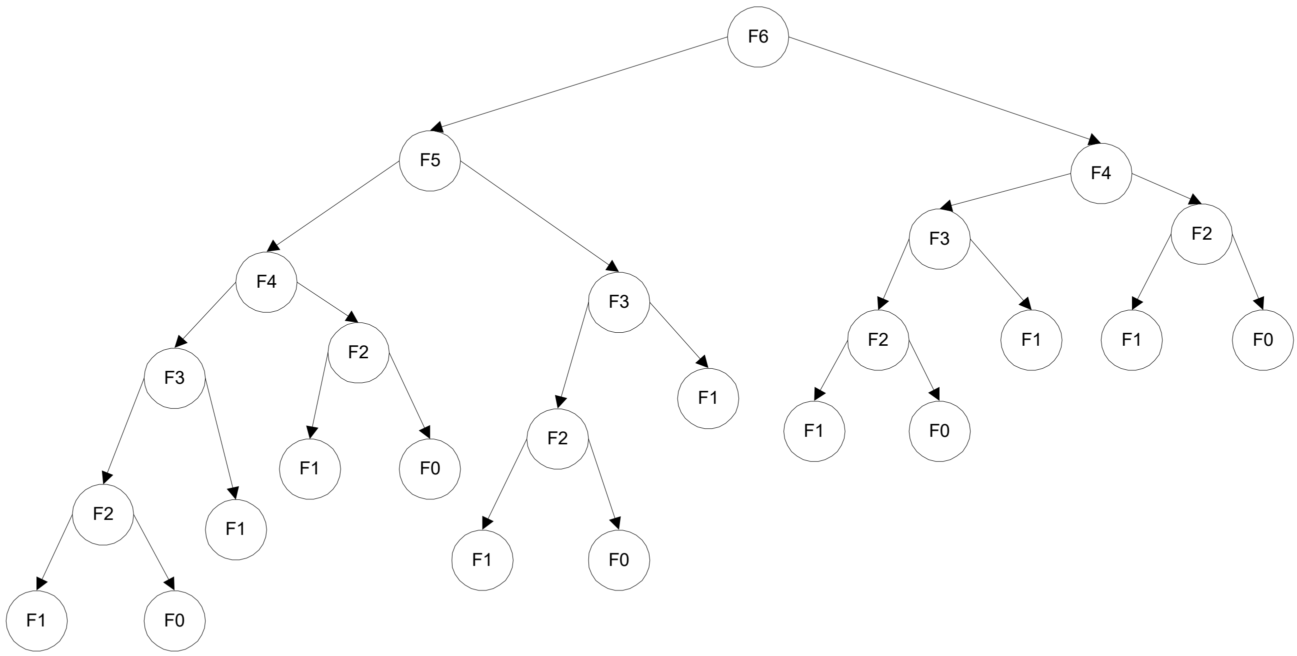
3) Especificar cómo se **recompone la solución** final a partir de los valores de las tablas.
- **Punto clave:** obtener la descomposición recurrente.
- Requiere mucha "creatividad"...

Problema de Fibonacci

Podemos dar una nueva versión de la función que calcula el n-ésimo término de la sucesión de Fibonacci.

$$F(n) = \begin{cases} 1 & \text{Si } n = 1 \text{ ó } n = 0 \\ F(n-1) + F(n-2) & \text{Si } n > 1 \end{cases}$$

El motivo de que el algoritmo recursivo sea tan lento es que el algoritmo recursivo del Fibonacci realiza muchos cálculos redundantes en forma explosiva.



- Si el algoritmo recursivo fuera capaz de **mantener en un vector los cálculos previamente realizados** y no hacer llamadas recursivas para un subproblema ya resuelto, se evitaría esta explosión de cálculos y sería más eficiente.

```
Método Cvector.Fibonacci( n )
        Si (n = 0 ó n = 1) entonces
                Respuesta  ← 1
        Sino
                Ultimo  ← 1
                Penúltimo  ← 1
                Para  k  desde  2  hasta  n   hacer
                        Respuesta  ← Ultimo + Penúltimo
                        Penúltimo  ← Ultimo
                        Ultimo  ←  Respuesta
                fPara
        FSi
        Retornar Respuesta
FMétodo
```

```
Métodos Cvector.Fibonacci( n )
        Si (n = 0 ó n = 1) entonces
                Vector[0]  ← 1
                Vector[1]  ← 1
        Sino
                Para  i  desde  2  hasta  n  hacer
                        Vector[ n ]  ← Vector [n –1] + Vector [n –2]
                fPara
        FSi
        Retornar Vector[n]
FMétodo
```

- El calculo de F(i) se consigue en tiempo **O(n).**

El Problema del cambio de monedas

- **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 dar el vuelto es

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

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 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.

- **Ecuación 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): enteroSi 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) ← +infinito
- Ecuación recurrente:

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

2) Aplicación ascendente mediante tablas

Matriz D → D[i, j] ← Cambio(i, j)
D: array [1..n, 0..P] de entero
para i:= 1, ..., n hacer
D[i, 0] ← 0
para i:= 1, ..., n hacer
para j:= 1, ..., P hacer
D[i, j] ← min(D[i-1, j], 1+D[i, j-c_i])
devolver D[n, P]

Ejemplo. N = 3, P= 8, c= (1, 4, 6)

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_1=4	0	1	2	3	1	2	3	4	2
3 C_1=6	0	1	2	3	1	2	1	2	2

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.

- Implementación:
  - x: array [1..n] de entero
  - x[i] = número de monedas usadas de tipo i
- Diseño del algoritmos

```
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 – c_i
        finsi
finmientras
```

V= 1	V =4	V = 6
0	2	0

El Problema de la Mochila

- 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.**1) Descomposición recurrente del problema**

- ¿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 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.

- **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 del mayor 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
  - ¿Y si m o k son negativos?Si m o k son negativos, el problema es irresoluble: Mochila(k, m) = -infinito

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

$$\mathrm{Mochila}(k, m) = \begin{cases} 0 & \text{si } k = 0 \text{ ó } m = 0 \\ -\infty & \text{si } q < 0 \text{ ó } k < 0 \\ \min\{\ \mathrm{Mochila}(k-1, m), b_k + \mathrm{Mochila}(k-1, m-p_k)\ \} & \text{si } q >= 0 \text{ ó } k > 0 \end{cases}$$

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

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

- Fila 0 y columna 0: casos base de valor 0.
- Los valores que caen fuera de la tabla son casos base de valor -□

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

- El beneficio óptimo es V[n, M]

Ojo: si j-p_i es negativo, entonces es el caso -¥, y el máximo será siempre el otro término.

•Ejemplo. n= 3, M= 6, p= (2, 3, 4), b= (4, 2, 5)

D	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	4
2	0	0	4	4	4	6	6
3	0	0	4	4	5	6	9

- ¿Cuánto es el orden de complejidad del algoritmo?**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 (x_1, x_2, ..., x_n) usando V.
- Idea: partiendo de la posición V[n, M], analizar las decisiones que se tomaron para cada objeto i.
  - Si V[i, j] = V[i-1, j], entonces la solución no usa el objeto i

        - x_i:= 0
    - Si V[i, j] = V[i-1, j-p_i] + b_i, entonces sí se usa el objeto i
        - x_i:= 1

- Si se cumplen ambas, entonces podemos usar el objeto i o no (existe más de una solución óptima).

```
j:= P
i := n
mientras i <> 0 y j <> 0 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
        finsi
        i := i - 1
finpara
```

P = 2	P =3	P = 4
1	0	1