# Programación dinámica

Algoritmia y optimización
Grado en Ingeniería en Inteligencia Artificial

# Programación dinámica

## La sucesión de Fibonacci

La sucesión de Fibonacci viene definida tal que:

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

# Programación dinámica

## La sucesión de Fibonacci

```
función fibonacci(n)
    si n <= 1
        devuelve n
    si no
        devuelve fibonacci(n-1) + fibonacci(n-2)
```

$$F(n) \in \mathcal{O}(2^n)$$

¿Dónde está el problema?

# Esquema general

# Programación dinámica

## Esquema general

- **Idea general**: descomposición del problema en subproblemas solapados.
- La eficiencia se consigue almacenando los resultados de los subproblemas, de manera que sólo se resuelven una única vez.

# Programación dinámica

## Principio de optimalidad

- Para que se pueda aplicar programación dinámica es esencial que se cumpla el **principio de optimalidad**.
- Un problema tiene una **subestructura óptima** si una solución **óptima** puede construirse **eficientemente** a partir de las **soluciones óptimas de sus subproblemas**.

# Programación dinámica

## Esquema general

Tipos de programación dinámica:

- **Memoización** (programación dinámica recursiva): Se almacenan los resultados de las llamadas recursivas realizadas.
- **Tabulación** (programación dinámica iterativa): Los resultados de subproblemas más pequeños se almacenan en una estructura, utilizándose para resolver los problemas más grandes

# Programación dinámica

## El número de Fibonacci (con memoización)

```
función fib(n, memo)
    si n <= 1
        devuelve n
    si n en memo
        devuelve memo[n]
    si no
        memo[n] := fib(n-1, memo) + fib(n-2, memo)
        devuelve memo[n]
```

$$T(n) \in \mathcal{O}(n)$$

# Programación dinámica

## El número de Fibonacci (con tabulación)

```
función fib(n, memo)
    si n <= 1
        devuelve n
    tabla[0] := 0
    tabla[1] := 1
    para i desde 2 hasta n:
        tabla[i] := tabla[i-1] + tabla[i-2]
    devuelve tabla[n]
```

$$T(n) \in \mathcal{O}(n)$$

# Programación dinámica

## Comparación con *Divide y vencerás*

| Divide y vencerás | Programación dinámica |
|---|---|
| Divide el problema en **subproblemas independientes** | Divide el problema en **subproblemas independientes** |
| Los subproblemas se resuelven **por separado** | Los subproblemas se resuelven **una vez y se almacenan** |
| **Combina** los resultados de los subproblemas **después de resolverlos** | **Reutiliza** subproblemas **previamente resueltos** para construir la solución |

# El problema de la mochila

# Programación dinámica

## El problema de la mochila

- El **problema de la mochila** (*knapsack problem*) es un problema clásico en teoría de algoritmos y computación.
- Consiste en **seleccionar** un subconjunto de **elementos**, cada uno con un **peso** y un **valor**, de manera que se **maximice el valor total sin exceder el peso máximo** permitido.
- Muchos problemas **reales** pueden reducirse a **una instancia** del problema de la mochila.

# Programación dinámica

## El problema de la mochila

- Existen varias versiones del problema de la mochila.
  - Problema de la mochila *discreta* (**programación dinámica**): los pesos son enteros.
  - Problema de la mochila *continua* (**algoritmo voraz**): los objetos se pueden fraccionar.
  - Problema de la mochila *general* (**algoritmos de vuelta atrás**): los pesos no se pueden fraccionar

# Programación dinámica

## El problema de la mochila

**Instancia**

Valores $(v_1, v_2, \ldots, v_n)$

Pesos $(w_1, w_2, \ldots, w_n)$

Peso máximo $W$

**Problema (versión discreta)**

$$\arg \max_{\mathbf{x} \in \{0,1\}^n} \sum_{i=1}^{n} v_i x_i$$

$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \leq W \qquad w_i \in \mathbb{N}$$

# Programación dinámica

## El problema de la mochila (discreta)

Instancia:

- **Valores**: v = (6, 6, 2, 1)
- **Pesos**: w = (3, 2, 1, 1)
- **Capacidad**: W = 5

**Valor máximo:** 12

**Solución:** x = (1, 1, 0, 0)

# Programación dinámica

## El problema de la mochila (discreta)

¿Cómo se puede resolver recursivamente?

- La mejor opción entre coger el objeto *i*-ésimo o no cogerlo cuando aún hay capacidad *W*.

$$K(i,W) = \begin{cases} 0 & \text{si } i = 0 \text{ o } W = 0 \\ K(i-1,W) & \text{si } w_i > W \\ \max(K(i-1,W), v_i + K(i-1,W-w_i)) & \text{si } w_i \leq W \end{cases}$$

# Programación dinámica

## El problema de la mochila (discreta)

### Solución recursiva

```
función knapsack(i, W, v, w)
    si i = 0
        devuelve 0
    si w[i] > W
        devuelve knapsack(i-1, W, v, w)
    si no
        devuelve máx( knapsack(i-1, W, v, w),
                      v[i] + knapsack(i-1, W-w[i], v, w) )
```

¿Qué complejidad tiene este algoritmo?

# Programación dinámica

## El problema de la mochila (discreta)

¿Cómo convertimos a Programación dinámica?

```
función knapsack(i, W, v, w)
    si i = 0
        devuelve 0
    si w[i] > W
        devuelve knapsack(i-1, W, v, w)
    si no
        devuelve máx( knapsack(i-1, W, v, w),
                      v[i] + knapsack(i-1, W-w[i], v, w) )
```

¿Qué complejidad tienen estas soluciones?

# La distancia de edición

# Programación dinámica

## La distancia de edición

- La distancia de edición (o *distancia de Levenshtein*) es una métrica que indica cuán similares son dos cadenas de texto.
- La distancia se define como el número mínimo de operaciones necesarias para transformar una cadena en otra.
  - Las operaciones permitidas son la inserción, eliminación y sustitución de caracteres.
- Esta métrica es fundamental en diversas aplicaciones de la inteligencia artificial

# Programación dinámica

## La distancia de edición

Consideremos el ejemplo **d(“casa”, “costa”)**:

- Sustituimos 'o' por la primera 'a': “cosa”
- Insertamos 't' después de 's’: “costa”
- Distancia total: 2

# Programación dinámica

## Distancia de edición: planteamiento recursivo

Dada la cadena origen y destino, se comparan los últimos caracteres:

- Si es el mismo, la solución es la misma que para sus subcadenas.
- Si es diferente:
  - Consideramos la operación de menor coste entre:
    - Insertar el carácter de la cadena destino.
    - Sustituir el carácter de la cadena origen por el de la cadena destino.
    - Eliminar el carácter de la cadena origen.
  - Y llamar recursivamente a las subcadenas correspondientes.

# Programación dinámica

## Distancia de edición: planteamiento recursivo

Dada la cadena s1 (de tamaño i) y la cadena s2 (de tamaño j):

$$d(i,j) = \begin{cases} j & \text{si } i = 0 \\ i & \text{si } j = 0 \\ d(i-1, j-1) & \text{si } s1[i-1] = s2[j-1] \\ 1 + \min\{d(i-1,j), d(i,j-1), d(i-1,j-1)\} & \text{si } s1[i-1] \neq s2[j-1] \end{cases}$$

# Programación dinámica

## Distancia de edición: solución recursiva

```
función distancia_edición(s1, s2)

    si s1[i-1] = s2[j-1]
        devuelve distancia_edición(s1[:i-1],s2[:j-1])
    si no
        devuelve 1 + min( distancia_edición([s1[:i-1],s2)
                          distancia_edición(s1,s2[:j-1]),
                          distancia_edición(s1[:i-1]s2[:j-1]) )
```

# Programación dinámica

## Distancia de edición

```
función distancia_edición(s1, s2)
    n,m := |s1|,|s2|
    DP := matriz de tamaño (n+1, m+1)
    DP[i][0] := i para i desde 0 hasta n
    DP[0][j] := j para j desde 0 hasta m

    para i desde 1 hasta n:
        para j desde 1 hasta m:
            si s1[i-1] = s2[j-1]
              DP[i][j] := DP[i-1][j-1]
            si no
              DP[i][j] := 1 + min( DP[i-1][j], DP[i][j-1], DP[i-1][j-1] )

    devuelve DP[n][m]
```

# Recuperación de soluciones

# Programación dinámica

## Recuperación de soluciones

- En los ejemplos anteriores hemos calculado los valores óptimos:
  - Máximo valor de la mochila
  - Distancia mínima de edición
- No proporcionamos las soluciones (qué objetos, qué operaciones de edición...)
- ¿Cómo podemos calcular las soluciones (eficientemente)?

# Programación dinámica

## Recuperación de soluciones

**d(“casa”,“costa”)**

| | - | c | o | s | t | a |
|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 |
| c | 1 | 0 | 1 | 2 | 3 | 4 |
| a | 2 | 1 | 1 | 2 | 3 | 3 |
| s | 3 | 2 | 2 | 1 | 2 | 3 |
| a | 4 | 3 | 3 | 2 | 2 | 2 |

# Programación dinámica

## Recuperación de soluciones

**d(“casa”,“costa”)**

| | - | c | o | s | t | a |
|---|---|---|---|---|---|---|
| - | 0 | 1 | 2 | 3 | 4 | 5 |
| c | 1 | 0 | 1 | 2 | 3 | 4 |
| a | 2 | 1 | 1 | 2 | 3 | 3 |
| s | 3 | 2 | 2 | 1 | 2 | 3 |
| a | 4 | 3 | 3 | 2 | 2 | 2 |

**Orden inverso**:

- cost**a** ← cas**a** (‘a’ con ‘a’)
- cost ← cas (insertar ‘t’)
- cos ← cas (‘s’ con ‘s’)
- c**o** ← c**a** (‘a’ por ‘o’)
- **c** ← **c** (‘c’ con ‘c’)
- ←

# Ejercicios

# Programación dinámica

## Ejercicio: búsqueda de soluciones

- Dada una matriz PD[·][·] utilizada para calcular la distancia de edición entre dos cadenas, escribe el algoritmo que recupera las operaciones de edición a realizar.
- Dada una matriz PD[·] utilizada para calcular el valor óptimo que cabe en una mochila, escribe el algoritmo que recupera qué objetos se han añadido.

# Programación dinámica

## Ejercicio: venta de oro

- Una empresa compra piezas de oro de **n** onzas y las trocea en piezas de **i** onzas (*i = 1, 2, ..., n*) que luego vende.
- El corte le sale gratis.
- El precio de venta de una pieza de **i** onzas es **v**$_{i}$
- ¿Cual es la forma óptima de trocear una pieza de **n** onzas para maximizar el precio de venta acumulado?

# Programación dinámica

## Ejercicio: cesta de la compra

- Queremos minimizar el coste de comprar N productos, que pueden adquirirse en cualquiera de K supermercados de la ciudad.
- Se conoce el coste $c_k$ de desplazarse hasta el sitio $s_k$ y los precios $p_{k1}, p_{k2}, \ldots, p_{kN}$ de cada producto en $s_k$.
- El gasto $c_k$ sólo se suma al coste total (una sola vez) si se adquiere, al menos, un producto en $s_k$.
- ¿Cómo se calcula el coste mínimo de la compra?