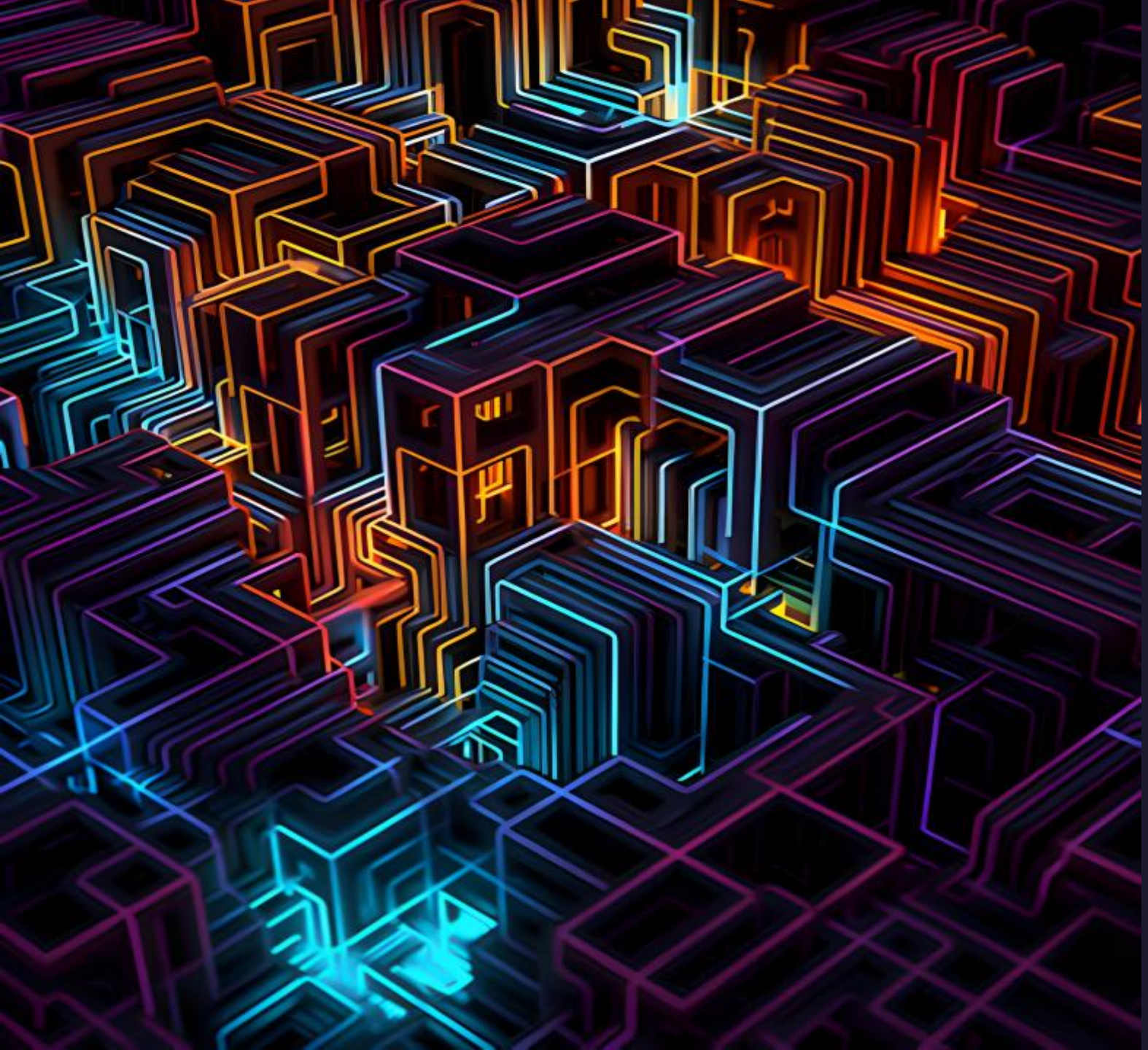


Técnicas de diseño de algoritmos: programación dinámica

Análisis y diseño de algoritmos avanzados

Dra. Valentina Narváez Terán

Tecnológico de Monterrey

Técnicas de diseño: programación dinámica

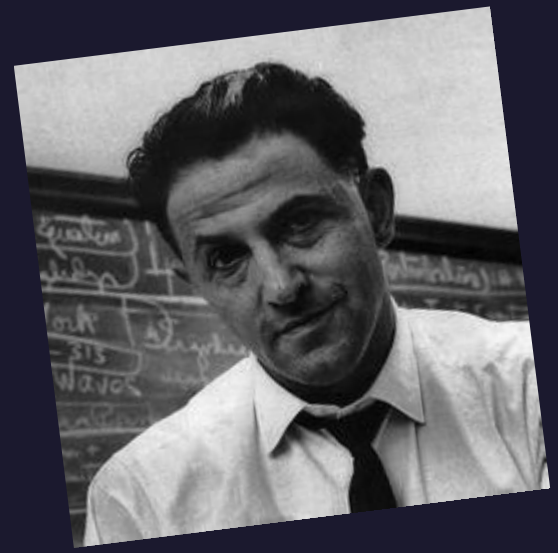
Enfoque para resolver problemas (principalmente de optimización) basado en:

1. Principio de optimalidad
   Se presenta cuando una solución optima para un problema se compone de soluciones optimas para sus subproblemas

2. Subestructuras óptimas: Partes de una solución optima, que son soluciones optimas para subproblemas

3. Relaciones de recurrencia y tablas de soluciones parciales

En este contexto:
"*programming*" != coding



Richard Bellman (1920-1984)

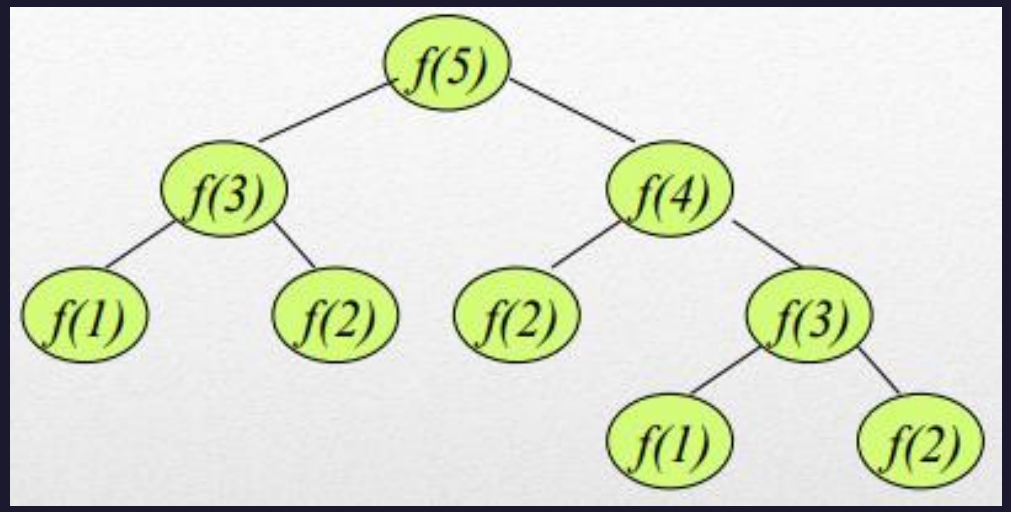
Ejemplo: números de Fibonacci

No es un problema de optimización, pero ilustra los conceptos clave de la técnica

El n-th número en la serie es:

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

```
int fibonacci (int n)
{   if (n == 0)
        {   return 0;   }
    else if (n == 1)
        {   return 1;   }
    else
        { return fibonacci(n-2)+fibonacci(n-1);}
}
```



El caso base ($n < 2$) no involucra llamadas recursivas

El caso recursivo involucra 2 por cada nueva llamada

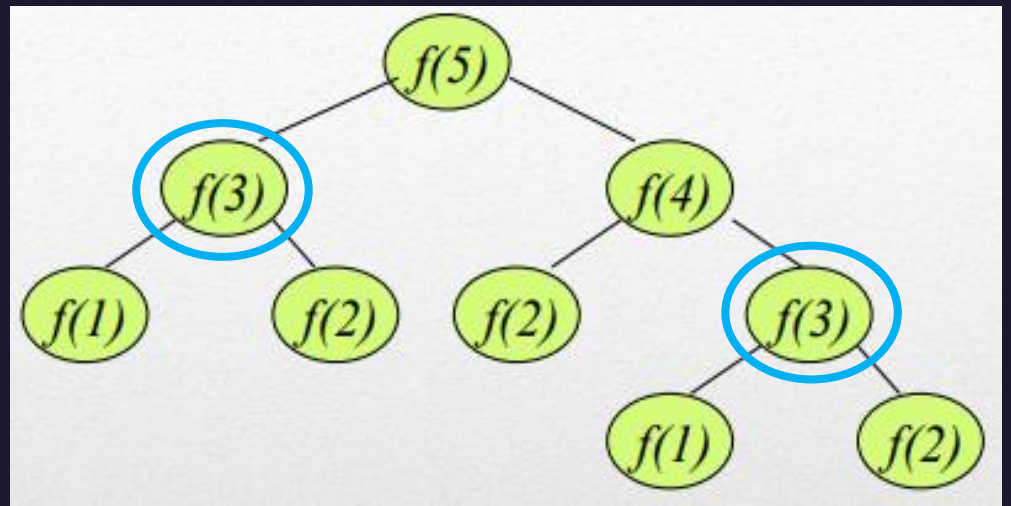
Complejidad

$O(2^n)$

Ejemplo: números de Fibonacci

Overlaping subproblems
Ocurren cuando la solución del mismo subproblema se utiliza varias veces en otros subproblemas

Por ejemplo, el cálculo de $f(3)$



Idea central en programación dinámica:
Recordar la solución a los subproblemas, en una tabla de soluciones parciales, para evitar recalcularlos

Enfoques principales:

- Memoization: enfoque top-down, recursivo, con tail-recursión y memoria auxiliar
- Tabulación: enfoque bottom-up, iterativo, con memoria auxiliar

Ejemplo: números de Fibonacci

```
#    Con memoization
def fibo_memo (n, a, b):
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fibo_memo (n-1, b, a+b )
```

Memoization:

Guardar soluciones a subproblemas en variables auxiliares, enviadas como parámetros en las llamadas recursivas (tail-recursión)

Ej. en a y b guardamos soluciones parciales, iniciando con los 2 primeros valores de la serie

Ejemplo: números de Fibonacci

Tabulación:
Es un trade-off entre espacio y tiempo

Las soluciones a problemas auxiliares se guardan en una tabla (arreglos, matrices, etc.) y el problema se resuelve iterativamente, evitando recálculos al consultar la tabla

En el caso del Fibonacci, las soluciones parciales se pueden guardar en un arreglo

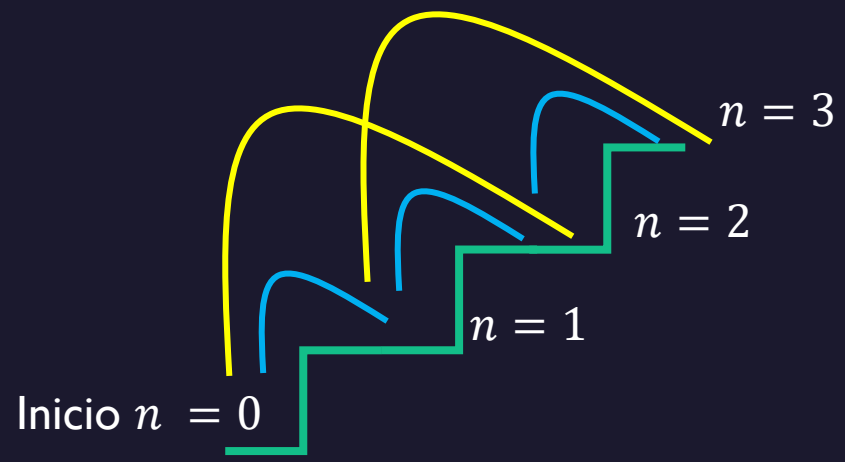
0	1	1	2	3	5	8	13
0	1	**2**	**3**	**4**	5	**6**	**7**

```python
#   Con tabulacion
def fibo_tabu(n) :
    serie = [0 for i in range(n+1)]

    if n == 0:
        return 0

    serie[1] = 1
    for i in range(2, n+1):
        serie[i] = serie[i-1] + serie[i-2]

    return serie[n]
```

Ejemplo: Stair-climbing

Tenemos una escalera de n peldaños.
Se puede subir en pasos de 1 o 2 peldaños a la vez

¿De cuantas formas diferentes se puede llegar hasta arriba?



¿Como resolverlo con programación dinámica?

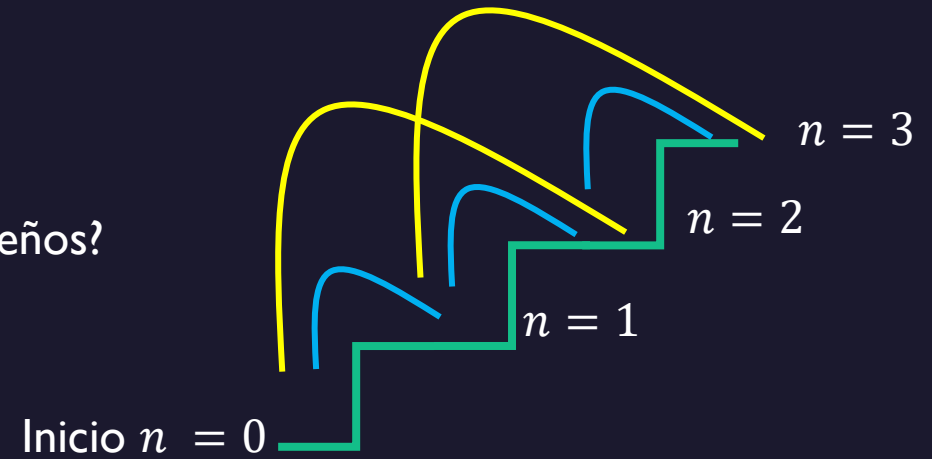
Definir el problema recursivamente, con una relación de recurrencia

¿Cuál es el caso mas pequeño?
¿Cómo un caso de tamaño n se relaciona con casos mas pequeños?

Ejemplo: Stair-climbing

¿Cuál es el caso mas pequeño?

¿Cómo un caso de tamaño n se relaciona con casos mas pequeños?



¿De cuantas formas se puede subir para $n = 1$? 1
1 paso de tamaño 1
Desde $n - 1$

¿De cuantas formas se puede subir para $n = 2$? 2
1 paso de tamaño 1
1 pasos de tamaño 2
Desde $n - 2$

¿De cuantas formas se puede subir para $n = 3$?
Las formas desde n=2 + las formas desde n-1

Otra forma de plantearlo:
¿Formas de subir desde el peldaño $n - 3$?

Ejemplo: Stair-climbing

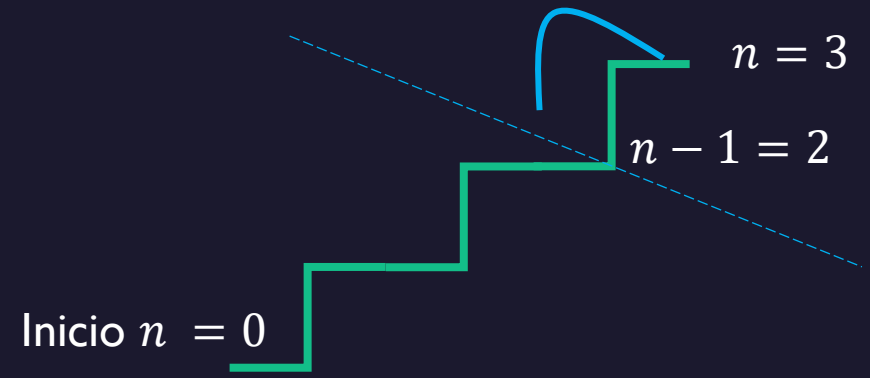
¿Y para n arbitraria?

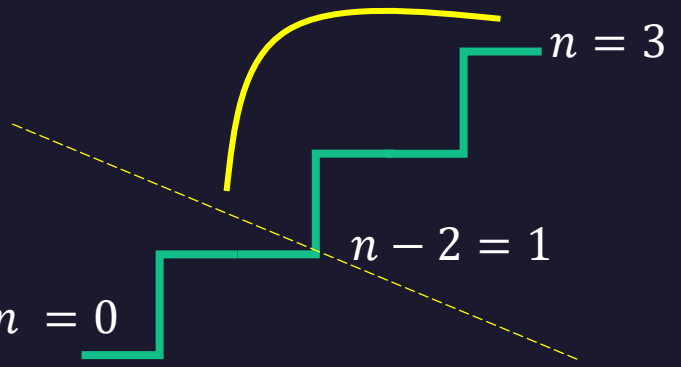
Hay dos posibilidades para el ultimo paso:
Un paso de 1, desde $n-1$
Un paso de 2, desde $n-2$

Repite recursivamente:
¿Posibilidades para el ultimo paso hasta $n-1$?
¿Posibilidades para el ultimo paso hasta $n-2$?

Entonces:

$$S(n) = \begin{cases} 0 & si\ n \leq 0 \\ n & \text{si } n = 1, n = 2 \\ S(n-1) + S(n-2) & \text{si no} \end{cases}$$





Ejemplo: Stair-climbing

```python
def stairs(n):
    if n <= 0:
        return 0
    elif n == 1 or n == 2:
        return n
    else:
        return stairs(n-1) + stairs(n-2)
```

Hasta aquí, tenemos la relación de recurrencia y podemos programarla

Pero para que sea realmente programación dinámica necesitamos memoization o tabulación

Memoization: top-down, con recursión terminal y variables acumuladoras

Tabulación: botomm-up, iterativo, con arreglo auxiliar

Luce familiar? Es (casi) como la serie Fibonacci!
Con (casi) los mismos overlaping subproblems

Entonces la implementación de programación dinámica (con memoization o tabulación) es similar

¿Pero que tal si pudiéramos dar pasos de tamaño 1, 2 y también de 3?

Ejemplo: Coin-change

Tenemos monedas de n valores diferentes

¿De cuantas formas diferentes podemos formar un valor K combinando monedas?

Ejemplo:

Para $n = 4$ denominaciones distintas

$C =$

1	2	5	10
0	1	**2**	**3**

Importante: en esta versión del problema solo importa contar las posibilidades, no listarlas

¿Si $K = 10$?

[10]

[5 5]

[2 2 1 5]
[1 1 2 1 5]
[1 1 1 1 1 5]

....

[1 1 1 1 1 2 2 1]
[1 1 1 1 1 2 1 1 1]

[1 1 1 1 1 1 1 1 1 1]

¿De cuantas formas podemos sumar K=10 con monedas de 1, 2, 5 y 10?

¿… con monedas de 1, 2 y 5?

¿… con monedas de 1 y 2?

¿… con monedas de 1?

Al elegir si una moneda de valor $C[i]$ participa, resta sumar $K - C[i]$

Ejemplo: Coin-change

Ecuaciones de Bellman: Son relaciones de recurrencia que describen el problema
Encontrarlas NO es trivial, pero una vez que se conocen, el problema es fácil

Para esta versión de coin-change, la relación de recurrencia es:

$$M(n,K) = \begin{cases} 1 & \text{si } K = 0 \\ 0 & \text{si } n \leq 0 \text{ o } K < 0 \\ M(n, K - C[n-1]) + & \\ M(n-1, K) & \text{si no} \end{cases}$$

Caso base:
-No dar ninguna moneda se considera una forma de sumar $K = 0$
-Valores de 0 o menos monedas, o K negativa no son validos

Caso recursivo

$M(n, K - C[n-1])$
Si $C[n-1]$ participa, nos resta sumar $K - C[n-1]$, con n tipos de monedas

$M(n-1, K)$
Si $C[n-1]$ no participa, nos resta sumar K, con $n-1$ tipos de monedas

Ejemplo: Coin-change

$$M(n,K) = \begin{cases} 1 & \text{si } K = 0 \\ 0 & \text{si } n \leq 0 \text{ o } K < 0 \\ M(n, K - C[n-1]) + M(n-1, K) & \text{si no} \end{cases}$$

¿Cómo guardar soluciones parciales?

Tabla de soluciones a subproblemas

Matriz T, con $n+1$ filas y $K+1$ columnas

Inicializa con $T[0][0] = 1$ para el caso base, y -1 en el resto

Celda $T[n][K]$:
La respuesta para $M(n,K)$

Ejemplo: Coin-change con memoization

```
1:  ALGORITHM      coins_memo(C, n, K, T)
2:  // Input: Monedas C, número de monedas n, valor K y matriz T, de
    tamaño n + 1 por K + 1. T inicialemente tiene T[0, 0] = 1, y −1 en todas
    las otras celdas.
3:  // Output: Número de formas T[n, K] de sumar K con n monedas
4:
5:  if K = 0 then
6:     T[n, k] = 1
7:     return T[n, k]
8:  end if
9:
10: if  n <= 0 or K < 0 then
11:    return 0
12: end if
13:
14: // Si T[n, K] ya esta calculado, no se recalcula, solo se consulta
15: // Si no esta calculado, se calcula y se guarda
16: if  T[n, K] ≠ −1 then
17:    return T[n, K]
18: else
19:    T[n, K] = coins_memo(C, n, K − C[n − 1], T) + coins_memo(C, n − 1, K, T)
20:    return T[n, K]
21: end if
```

$$M(n,K) = \begin{cases} 1 & \text{si } K = 0 \\ 0 & \text{si } n \leq 0 \text{ o } K < 0 \\ M(n, K - C[n-1]) + M(n-1, K) & \text{si no} \end{cases}$$

En el caso recursivo, se consulta la tabla T, para evitar recalcular

Ejemplo: Coin-change con tabulacion

```
1:  ALGORITHM      coins_tabu(C, n, K)
2:  // Input: Arreglo de monedas C, número de monedas n y valor K
3:  // Output: Número de formas T[i, j] de sumar K
4:
5:  T = matriz de n + 1 filas y K + 1 columnas inicializado en 0
6:
7:  T[0, 0] = 1 // Caso base
8:
9:  for i = 1 to n + 1 do
10:   for j = 0 to K + 1 do
11:     T[i, j]+ = T[i − 1, j]
12:     if j − C[i − 1] >= 0 then
13:       T[i, j]+ = T[i, j − C[i − 1]]
14:     end if
15:   end for
16: end for
17: return T[i, j]
```

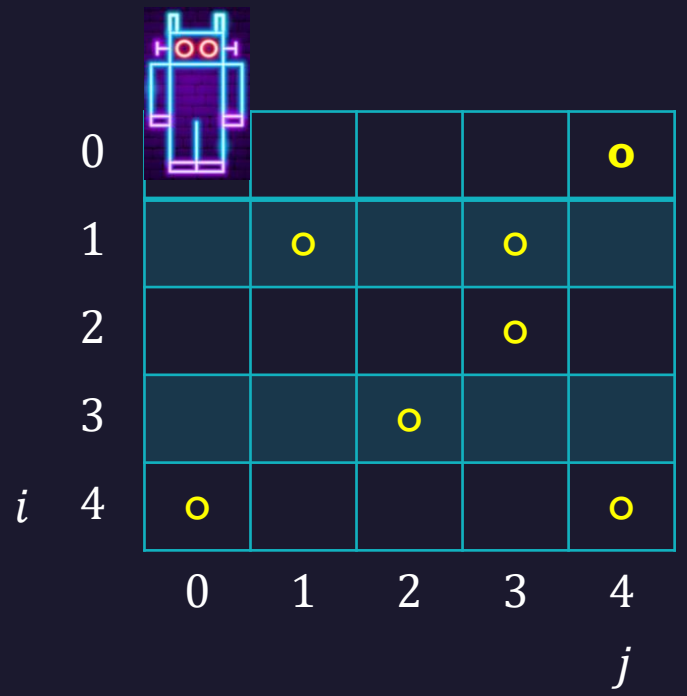
$$M(n,K) = \begin{cases} 1 & \text{si } K = 0 \\ 0 & \text{si } n \leq 0 \text{ o } K < 0 \\ M(n, K - C[n-1]) + & \\ M(n-1, K) & \text{si no} \end{cases}$$

Los ciclos iteran sobre la tabla de soluciones parciales T, hasta $T[n][K]$

Ejemplo: coin-collecting (o gold mining)

Tienes una cuadricula de tamaño n. En las celdas marcadas hay monedas de oro (representadas por unos y ceros)

Un robot minero, que inicia en $[0,0]$, debe recolectarlas… pero esta averiado!
El robot solo se puede mover a la derecha o hacia abajo, nunca a izquierda o arriba.



¿Cuál es la máxima cantidad de monedas que puede recolectar?
Asume que debe terminar el recorrido en $i, j = n$, así que la máxima cantidad es $F(i,j)$.
C es la matriz donde se encuentran las monedas.

$$F(i,j) = \begin{cases} \max\big(F(i-1,j), F(i,j-1)\big) + C[i,j] & \text{si } i,j \geq 0 \\ 0 & \text{si } i < 0 \text{ , o } j < 0 \end{cases}$$

¿Pero por qué?

Tarea + código.
Individual, día de la prox. clase