Programación Dinámica

Materia: TTPS

Autor: JTP - Matías Fluxa

¿Qué es la programación dinámica?

¿Qué es la programación dinámica?

Definición de Wikipedia

- En informática, la programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.

¿Qué es la programación dinámica?

Definición de Gemini

- La programación dinámica es una **técnica** de optimización matemática y un método de diseño de algoritmos que resuelve problemas complejos dividiéndolos en subproblemas más pequeños y sencillos. La clave de esta técnica es que almacena las soluciones de estos subproblemas para no tener que volver a calcularlas más tarde.

Veamos un problema de ejemplo ...

Un entero positivo **N** se denomina **fachero** si cumple las siguientes condiciones:

- Tiene 8 dígitos
- Todos los dígitos de **N** son impares
- Todas las restas de dos dígitos consecutivos de **N** el mayor menos el menor, son iguales a 2.

Hallar la cantidad de números **facheros** que existen.

Resolución con DP

- Podemos pensar en un principio en definir

  $f(x)$: Cantidad de números facheros de x dígitos.

- Nuestro problema nos pide específicamente hallar $f(8)$.

- Si queremos seguir con la definición de **DP** que vimos anteriormente, podríamos pensar en resolver casos más chicos primero.

- Supongamos que logramos hallar $f(7)$ de alguna manera ¿podríamos obtener fácilmente $f(8)$ a partir del resultado conocido?

Resolución con DP

- Conviene cambiar la definición del **estado** de la siguiente manera

  $\mathrm{g}(x, k)$: # de números facheros de x dígitos que terminan con el dígito k.

- Notemos que $f(8) = g(8,1) + \mathrm{g}(8,3) + \mathrm{g}(8,5) + \mathrm{g}(8,7) + \mathrm{g}(8,9)$.

- Sabiendo todos los $g(7, k)$, veamos que ahora es más fácil obtener todos $g(8, k)$

Resolución con DP

$g(7,1)$

$g(7,3)$

$g(7,5)$

$g(7,7)$

$g(7,9)$

$g(8,1)$

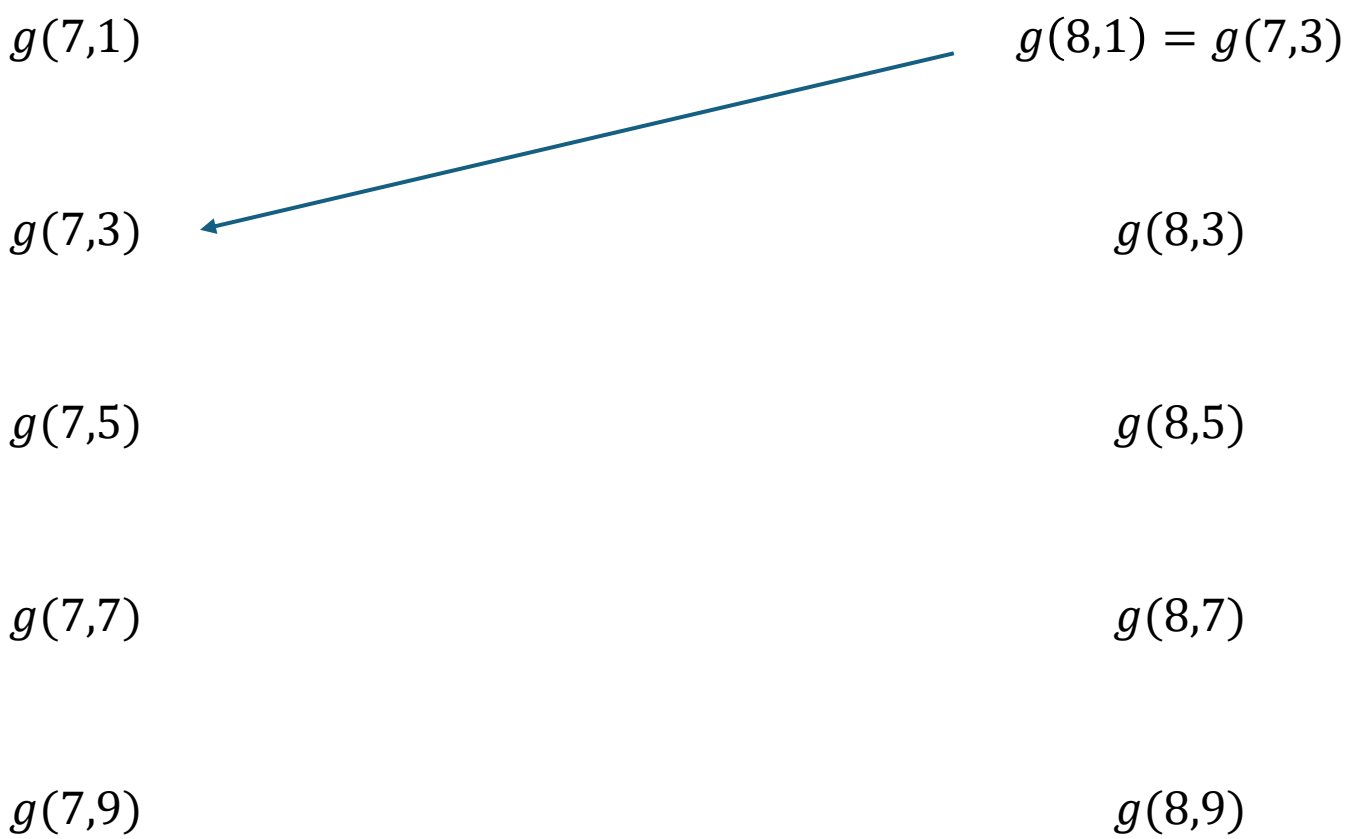
$g(8,3)$

$g(8,5)$

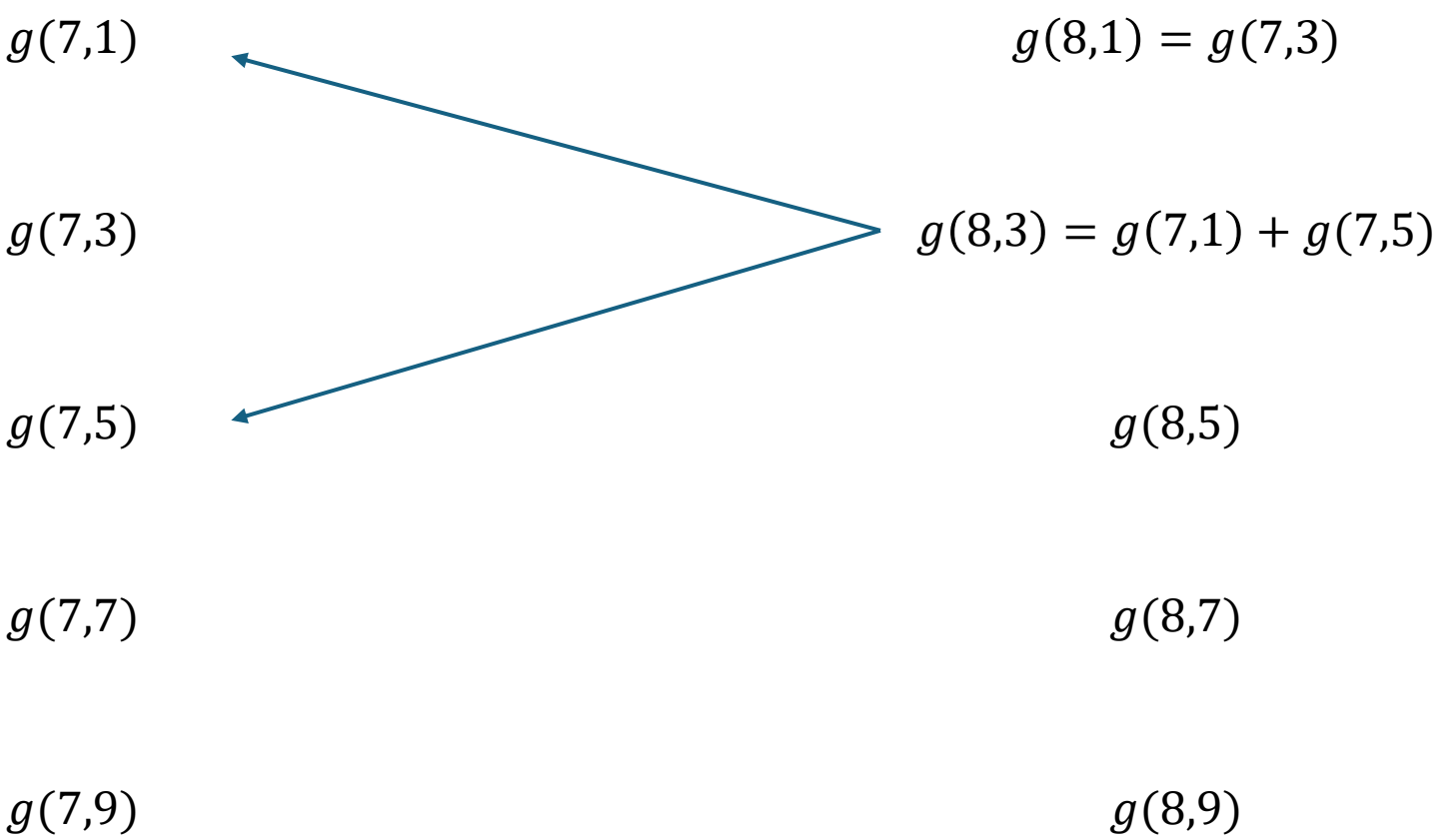
$g(8,7)$

$g(8,9)$

Resolución con DP



Resolución con DP



Resolución con DP

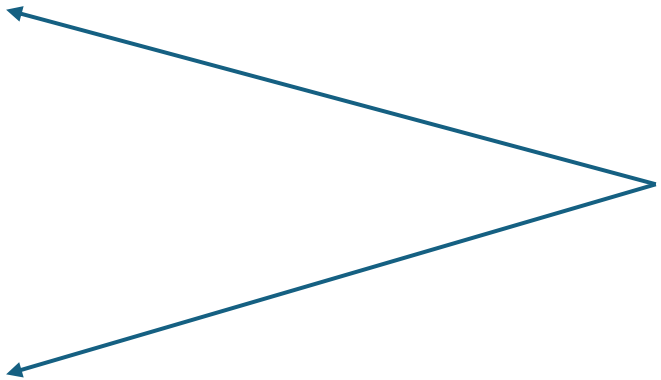
$g(7,1)$

$g(7,3)$

$g(7,5)$

$g(7,7)$

$g(7,9)$



$g(8,1) = g(7,3)$

$g(8,3) = g(7,1) + g(7,5)$

$g(8,5) = g(7,3) + g(7,7)$

$g(8,7)$

$g(8,9)$

Resolución con DP

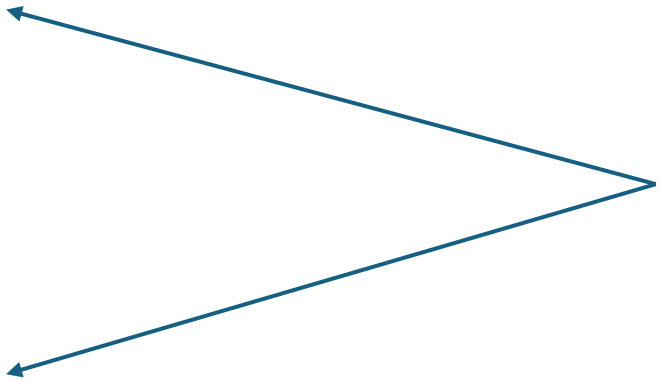
$g(7,1)$

$g(7,3)$

$g(7,5)$

$g(7,7)$

$g(7,9)$



$g(8,1) = g(7,3)$

$g(8,3) = g(7,1) + g(7,5)$

$g(8,5) = g(7,3) + g(7,7)$

$g(8,7) = g(7,5) + g(7,9)$

$g(8,9)$

Resolución con DP

$g(7,1)$

$g(7,3)$

$g(7,5)$

$g(7,7)$

$g(7,9)$

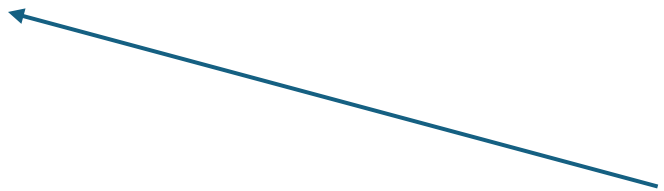
$g(8,1) = g(7,3)$

$g(8,3) = g(7,1) + g(7,5)$

$g(8,5) = g(7,3) + g(7,7)$

$g(8,7) = g(7,5) + g(7,9)$

$g(8,9) = g(7,7)$



Resolución con DP

- Como aún no sabemos la cantidad de números facheros que existen de 7 dígitos, debemos hacer el mismo proceso en el paso 6 dígitos → 7 dígitos.
- Recursivamente, vamos ir cada vez reduciendo la cantidad de dígitos, hasta llegar a un caso base o que sea trivial de resolver, que en este caso puede ser de un dígito.
- Es fácil ver que $g(1,1) = g(1,3) = g(1,5) = g(1,7) = g(1,9) = 1$

Resolución con DP

- Obtendríamos una tabla como esta:

		# Dígitos							
		1	2	3	4	5	6	7	8
Termina en ...	1	1	1	2	3	6	9	18	27
	3	1	2	3	6	9	18	27	54
	5	1	2	4	6	12	18	36	54
	7	1	2	3	6	9	18	27	54
	9	1	1	2	3	6	9	18	27
	Total	5	8	14	24	42	72	126	216

- Finalmente, la respuesta es 216 números facheros de 8 dígitos.

¿Qué es la programación dinámica?

Definición personal

- La programación dinámica es una **técnica** de resolución de problemas que consiste en definir **estados**, y obtener información de ellos a través de sus **relaciones** con otros estados más simples.

¿Cuándo aplicamos Programación Dinámica?

La programación dinámica es conveniente aplicarla cuando:

- Podemos definir un **estado** a partir de las características del problema.
- Podemos conocer fácilmente la dependencia de los estados entre sí (conocer las **relaciones**).
- Debido a que cada estado y cada relación debe ser procesada **exactamente una vez**, la cantidad de estados sumado a la cantidad de relaciones deben ser un número que se adecue a nuestras limitaciones de **tiempo y memoria.**
- Las relaciones tienden a llegar a los casos bases, es decir, no se generan “bucles” entre las relaciones. En palabras más técnicas, los estados y las relaciones forman un **Grafo Dirigido Acíclico (DAG)**.

Técnicas de implementación

Existen principalmente dos técnicas para implementar las transiciones entre estados.

- **Bottom-Up:** Consiste en partir de los casos bases, e ir “propagando” la respuesta hasta llegar al problema principal que era de interés. Sus implementaciones suelen ser **iterativas**.
- **Top-Down**: Consiste en partir desde el problema principal, e ir dividiéndolo en problemas más pequeños hasta llegar a los casos bases. Sus implementaciones suelen ser **recursivas**. Para evitar visitar múltiples veces los mismos estados, suelen implementarse con **Memoization**.

No hay un método que siempre sea mejor que el otro, su elección suele depender muchas veces del problema a resolver y cuál facilita su implementación.

Ejemplo de Bottom-Up

Vamos a hallar el n-ésimo término de Fibonacci con Bottom-Up. Recordemos que la sucesión de Fibonacci cumple que cada elemento es igual a la suma de los dos anteriores, es decir:

$$f(i) = f(i-1) + f(i-2)$$

- Para el caso Bottom-Up, arrancamos definiendo los casos bases $f(0) = 0$ y $f(1) = 1$.
- Iteramos i desde 2 a n, y en cada paso hallamos $f(i)$.
- De esta manera, vamos resolviendo de los casos más chicos, hasta llegar al problema original.

Ejemplo de Bottom-Up

Implementación en Python

```python
n = 15
fib = [0] * (n+1)
fib[0] = 0
fib[1] = 1
for i in range(2, n+1):
    fib[i] = fib[i-1] + fib[i-2]
print(fib[n])
```

✓ 0.0s

```
610
```

Ejemplo de Top-Down

Ahora, vamos a hallar el n-ésimo término de Fibonacci con Top-Down.

- Para este caso, comenzamos desde el problema principal. Sabemos que:
$$f(n) = f(n-1) + f(n-2)$$
- Como aún no sabemos ninguno de los otros dos, resolvemos recursivamente $f(n-1)$.
$$f(n-1) = f(n-2) + f(n-3)$$
- Y así recursivamente hasta llegar a los casos bases:
$$f(2) = f(1) + f(0)$$

Ejemplo de Top-Down

Implementación en Python

```python
n = 15

def fib(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)

print(fib(n))
```

✓ 0.0s

```
610
```

Ejemplo de Top-Down

Implementación en Python

WARNING

Esta implementación puede generar que la recursión visite más de una vez cada estado. Para el caso de Fibonacci, se hace exponencial el tiempo de ejecución. Por eso se implementa con Memoization.

```
n = 15

def fib(n: int) -> int:
    if n == 0: return 0
    if n == 1: return 1
    return fib(n-1) + fib(n-2)

print(fib(n))
```

✓ 0.0s

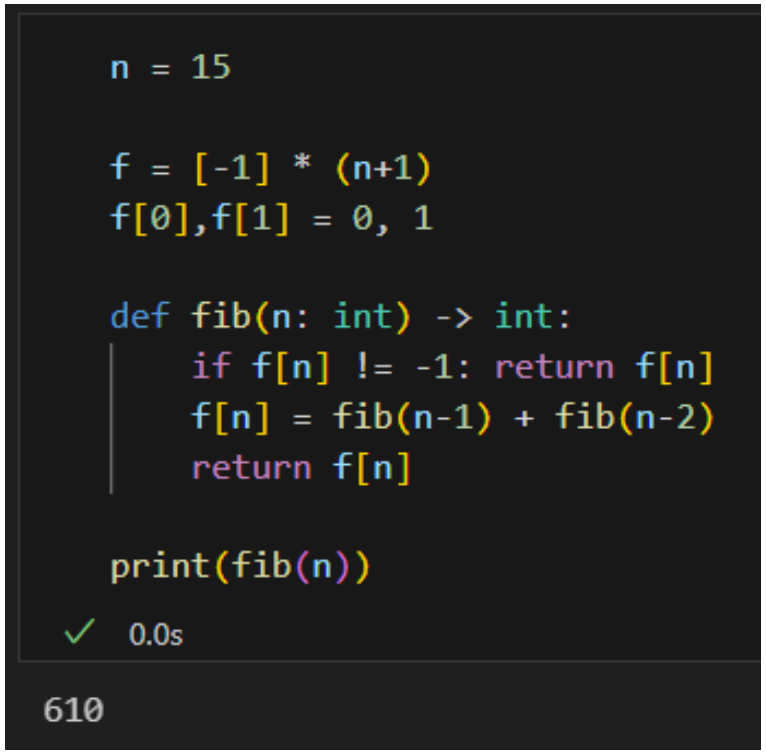
```
610
```

Ejemplo de Top-Down

Implementación en Python

Con Memoization

El concepto consiste en pagar memoria para reducir ejecuciones redundantes



Algunos problemas clásicos de DP

Longest Increasing Subsquence (LIS)

Dado un arreglo de n números, hallar la subsecuencia creciente más grande posible.

Una subsecuencia es una secuencia que se obtiene del arreglo original, eliminando algunos de sus elementos (posiblemente cero) y conservando su orden original.

Para que sea creciente, cada número de la secuencia debe ser estrictamente mayor que el anterior.

Por ejemplo:

$$A=(0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15)$$

Una de las subsecuencias crecientes más largas es:

$$A'=(0,2,6,9,11,15)$$

Longest Increasing Subsquence (LIS)

Solución

Definimos nuestro estado:
$\boldsymbol{dp}[\boldsymbol{i}]$: *La longitud de la subsecuencia más larga que termina en el i-ésimo elemento del arreglo.*

Inicialmente, $\boldsymbol{dp}[\boldsymbol{i}] = 1$ $(\forall\, i \in [1,2, \ldots . n])$

Luego las dependencias quedan:

$$\boldsymbol{dp}[\boldsymbol{i}] = \mathbf{max}(\boldsymbol{dp}[\boldsymbol{j}] + \mathbf{1})\ (\forall\, j < i : a[j] < a[i])$$

La respuesta final sería $\mathbf{max}(\boldsymbol{dp}[\boldsymbol{i}])$ $(\forall\, i \in [1,2, \ldots, n])$

Longest Increasing Subsquence (LIS)

Solución

Complejidad en memoria: $\boldsymbol{O(n)}$

Complejidad en tiempo de ejecución: $\boldsymbol{O(n^2)}$

Desafío

Implementar la solución con tiempo de ejecución $\boldsymbol{O(n \cdot log(n))}$

Longest Common Subsquence (LCS)

Dado un arreglo $\boldsymbol{a}$ de $\boldsymbol{n}$ números y un arreglo $\boldsymbol{b}$ de $\boldsymbol{m}$ números, hallar la mayor subsecuencia en común a ambos arreglos.

Por ejemplo:

$$\boldsymbol{a} = (\mathbf{3}, \mathbf{1}, \mathbf{3}, \mathbf{2}, \mathbf{7}, \mathbf{4}, \mathbf{8}, \mathbf{2})$$
$$\boldsymbol{b} = (\mathbf{6}, \mathbf{5}, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4})$$

Entonces una posible mayor subsecuencia en común es la secuencia $(1,2,4)$.

Longest Common Subsquence (LCS)

Solución

Definimos nuestro estado de la siguiente manera:

$\boldsymbol{dp[i][j]}$: La mayor subsecuencia en común considerando los primeros $\boldsymbol{i}$ elementos del arreglo $\boldsymbol{a}$ los primeros $\boldsymbol{j}$ elementos del arreglo $\boldsymbol{b}$.

Inicialmente $\boldsymbol{dp[0][0] = 0}$

Luego, para cada estado $\boldsymbol{(i, j)}$ inicializamos:

$$\boldsymbol{dp[i][j] = max(dp[i-1][j], dp[i][j-1])}$$

Si $\boldsymbol{a[i] == b[j]}$, entonces $\boldsymbol{dp[i][j] = max(dp[i][j], dp[i-1][j-1] + 1)}$

La respuesta final sería $\boldsymbol{dp[n][m]}$

Longest Increasing Subsquence (LIS)

Solución

Complejidad en memoria: $\boldsymbol{O(n^2)}$

Complejidad en tiempo de ejecución: $\boldsymbol{O\left(n^2\right)}$

Desafío

Implementar la solución con memoria $\boldsymbol{O(n)}$