4 Programación dinámica

ALGORÍTMICA Y COMPLEJIDAD

Camilo Palazuelos Calderón

Curso 2023-2024

Qué es la programación dinámica (PD)

- La PD es una estrategia que se basa en el uso de un array de soluciones de subproblemas
  - Suele utilizarse cuando el problema no puede dividirse en subproblemas no solapados

- Tanto el diseño recursivo como el iterativo son maneras naturales de pensar en PD
  - Se expresa la solución *recursivamente*
  - Se rellena el array de soluciones hasta encontrar la del problema original

- Ejemplo: Computar el n-ésimo término de la sucesión de Fibonacci
  - ¡El coste temporal del algoritmo directo (FIB1) es $O(\varphi^n)$!
  - Queremos un algoritmo eficiente

```
FIB1(n):
    if n ≤ 1
        return n
    return FIB1(n – 1) + FIB1(n – 2)
```

```
FIB2(n):
    if n ≤ 1
        return n
    if F[n] = NULL
        F[n] ← FIB2(n – 1) + FIB2(n – 2)
    return F[n]

FIB3(n):
    F[0] ← 0
    F[1] ← 1
    for i ← 2 to n
        F[i] ← F[i – 1] + F[i – 2]
    return F[n]
```

Memoización

- La memoización consiste en almacenar el resultado de la llamada recursiva para su uso posterior
  - Es la base del enfoque *de arriba abajo* (*top-down*; véase FIB2)
- Observando cómo FIB2 rellena $F[0..n]$, podemos diseñar un algoritmo iterativo que lo rellene deliberadamente
  - Este es el enfoque *de abajo arriba* (*bottom-up*; véase FIB3)
- Ambos algoritmos realizan $O(n)$ sumas
  - El n-ésimo número de Fibonacci tiene $O(n)$ dígitos
  - El coste temporal de FIB2 y FIB3 es, por tanto, $O(n^2)$

Particiones, I

- Algunas definiciones
  - $p(n)$: número de formas en que se puede escribir n como suma de enteros positivos sin importar el orden
  - $p_k(n)$: número de formas en que se puede escribir n como suma de como mucho k enteros positivos sin importar el orden
  - $p(n) = p_n(n)$
  - $p_k(n) = p_{k-1}(n) + p_k(n-k)$ si $n \geq 0$ y $k > 0$, $p_0(0) = 1$ y $p_k(n) = 0$ en otro caso

n \ k	0	1	2	3	4	5
0	1	1	1	1	1	1
1		1	1	1	1	1
2		1	2	2	2	2
3		1	2	3	3	3
4		1	3	4	5	5
5		1	3	5	6	7

- Pasos para llegar a la solución
  - Identificar, en el array $P[0..n, 0..n]$, la *casilla solución* (en el ejemplo, $P[5, 5]$)
  - Definir los *casos base* (en el ejemplo, $n = 0$ y $k = 1$)
  - *Top-down:* de la casilla solución a los casos base recursivamente
  - *Bottom-up:* de los casos base a la casilla solución iterativamente

Particiones, II

- Enfoque de arriba abajo (*top-down*)

```
TopDown(n):
    return TopDown(n, n)

TopDown(n, k):
    if n = 0 or k = 1
        return 1
    else if n > 0 and k > 1
        ▷ Memoización
        if P[n, k] = Null
            P[n, k] ← TopDown(n, k – 1)
                    + TopDown(n – k, k)
        return P[n, k]
    else
        return 0
```

- Enfoque de abajo arriba (*bottom-up*)

```
BottomUp(n):
    ▷ Rellenado de los casos base
    P[0, 0] ← 1
    for i ← 1 to n
        P[0, i] ← 1
        P[i, 1] ← 1
    ▷ Rellenado del resto del array
    for i ← 1 to n
        for j ← 2 to n
            P[i, j] ← P[i, j – 1]
            ▷ Se comprueba que el segundo término
              no se sale del array
            if i ≥ j
                P[i, j] ← P[i, j] + P[i – j, j]
    return P[n, n]
```

Particiones, III

- Tanto TOPDOWN como BOTTOMUP realizan $O(n^2)$ sumas
  - $p(n)$ tiene $O(n^{1/2})$ dígitos, por lo que el coste temporal es $O(n^{2.5})$

- El coste espacial de ambos algoritmos es $O(n^2)$

- Sin embargo, tal y como hemos implementado los algoritmos, el *top-down* será más rápido
  - Es complicado reproducir, en el *bottom-up*, el *patrón recursivo* de rellenado del array
  - Aunque asintóticamente sean equivalentes, esto hace que las complejidades temporal y espacial del algoritmo memoizado sean difíciles de igualar por el enfoque de abajo arriba