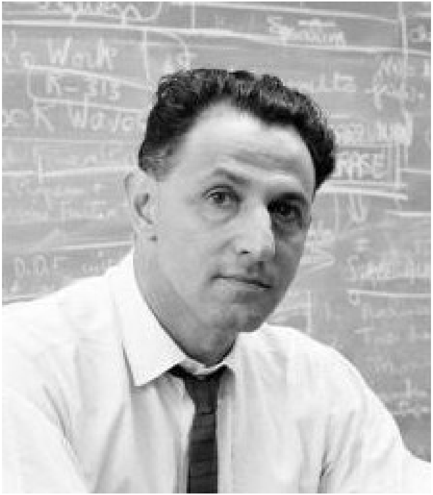
Programación dinámica

Técnicas de Diseño de Algoritmos / Algoritmos y Estructuras de Datos III

Primer cuatrimestre 2025

Programación dinámica



Richard Bellman (1920–1984)

Programación dinámica

I spent the Fall quarter [of 1950] at RAND. My first task was to find a name for multistage decision processes. (...) The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named [Charles Ewan] Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research". (...) Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.

–Richard Bellman, Eye of the Hurricane: An Autobiography (1984)

Programación dinámica

- Se divide el problema en subproblemas de tamaños menores que se resuelven recursivamente.

Programación dinámica

- Se divide el problema en subproblemas de tamaños menores que se resuelven recursivamente.
- **Ejemplo.** Cálculo de los números de Fibonacci: $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$ para $n \geq 2$.

Programación dinámica

No es buena idea implementar un algoritmo recursivo directo basado en esta fórmula (¿por qué?).

algoritmo $\mathtt{Fibonacci}(n)$

  entrada: `un entero` n

  salida: $F(n)$

  si $k = 0$ **hacer**

    retornar 0

  sino, si $k = 1$ **hacer**

    retornar 1

  si no

    $a := \mathtt{Fibonacci}(n-1)$

    $b := \mathtt{Fibonacci}(n-2)$

    retornar $a + b$

  fin si

Programación dinámica

- **Superposición de estados:** El árbol de llamadas recursivas resuelve el mismo problema varias veces.
  - Alternativamente, podemos decir que se realizan muchas veces llamadas a la función recursiva con los mismos parámetros.

Programación dinámica

- **Superposición de estados:** El árbol de llamadas recursivas resuelve el mismo problema varias veces.
  - Alternativamente, podemos decir que se realizan muchas veces llamadas a la función recursiva con los mismos parámetros.
- Un algoritmo de programación dinámica evita estas repeticiones con alguno de estos dos esquemas:

Programación dinámica

- **Superposición de estados:** El árbol de llamadas recursivas resuelve el mismo problema varias veces.
  - Alternativamente, podemos decir que se realizan muchas veces llamadas a la función recursiva con los mismos parámetros.
- Un algoritmo de programación dinámica evita estas repeticiones con alguno de estos dos esquemas:
  1. **Enfoque top-down.** Se implementa recursivamente, pero se guarda el resultado de cada llamada recursiva en una estructura de datos (memorización). Si una llamada recursiva se repite, se toma el resultado de esta estructura.
  2. **Enfoque bottom-up.** Resolvemos primero los subproblemas más pequeños y guardamos (habitualmente en una tabla) todos los resultados.

Ejemplo: Fibonacci

0	1	2	3	4	5	$\cdots$	$k-1$	k
0	1					$\cdots$		

Ejemplo: Fibonacci

0	1	2	3	4	5	$\cdots$	$k-1$	k
0	1	2				$\cdots$		

Ejemplo: Fibonacci

0	1	2	3	4	5	$\cdots$	$k-1$	k
0	1	2	3			$\cdots$		

Ejemplo: Fibonacci

0	1	2	3	4	5	$\cdots$	$k-1$	k
0	1	2	3	5		$\cdots$		

Ejemplo: Fibonacci

0	1	2	3	4	5	$\cdots$	$k-1$	k
0	1	2	3	5	8	$\cdots$		

Ejemplo: Fibonacci

algoritmo $Fibonacci(n)$
entrada: `un entero` n
salida: $F(n)$

$A[0] \leftarrow 0$
$A[1] \leftarrow 1$
para $j = 2$ **hasta** n **hacer**
$A[j] \leftarrow A[j-1] + A[j-2]$
fin para
retornar $A[n]$

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad $\Omega(F(n)) \sim \Omega(1,6^n)$.

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad $\Omega(F(n)) \sim \Omega(1,6^n)$.
- Programación dinámica (bottom-up):
  - Complejidad

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad $\Omega(F(n)) \sim \Omega(1,6^n)$.
- Programación dinámica (bottom-up):
  - Complejidad $O(n)$.

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad $\Omega(F(n)) \sim \Omega(1,6^n)$.
- Programación dinámica (bottom-up):
  - Complejidad $O(n)$. Ojo! el tamaño de entrada es $O(\log n)$

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad $\Omega(F(n)) \sim \Omega(1,6^n)$.
- Programación dinámica (bottom-up):
  - Complejidad $O(n)$. Ojo! el tamaño de entrada es $O(\log n)$
  - Espacio

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad $\Omega(F(n)) \sim \Omega(1,6^n)$.
- Programación dinámica (bottom-up):
  - Complejidad $O(n)$. Ojo! el tamaño de entrada es $O(\log n)$
  - Espacio $\Theta(1)$: sólo necesitamos almacenar los dos últimos valores calculados.

Ejemplo: Fibonacci

- Función recursiva:
  - Complejidad $\Omega(F(n)) \sim \Omega(1,6^n)$.
- Programación dinámica (bottom-up):
  - Complejidad $O(n)$. Ojo! el tamaño de entrada es $O(\log n)$
  - Espacio $\Theta(1)$: sólo necesitamos almacenar los dos últimos valores calculados. Eso no es posible con top-down

Ejemplo: El problema del cambio

- Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando monedas de 1, 5, 10 y 25 centavos. Por ejemplo, si el monto es \$0,69, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10 centavos, una de 5 centavos y cuatro de un centavo.

Ejemplo: El problema del cambio

- Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando monedas de 1, 5, 10 y 25 centavos. Por ejemplo, si el monto es \$0,69, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10 centavos, una de 5 centavos y cuatro de un centavo.

- **Problema.** Dadas las denominaciones $a_1, \ldots, a_k \in \mathbb{Z}_+$ de monedas (con $a_i > a_{i+1}$ para $i = 1, \ldots, k-1$) y un objetivo $t \in \mathbb{Z}_+$, encontrar $x_1, \ldots, x_k \in \mathbb{Z}_{\geq 0}$ tales que

$$t = \sum_{i=1}^{k} x_i \, a_i$$

minimizando $x_1 + \cdots + x_k$.

Ejemplo: El problema del cambio

- $f(s)$: Cantidad mínima de monedas para entregar s centavos, para $s = 0, \dots, t$.

$$f(s) = \begin{cases} 0 & \text{si } s = 0 \\ \min_{i: a_i \leq s} 1 + f(s - a_i) & \text{si } a_k \leq s \\ \infty & \text{ninguno de los casos anteriores} \end{cases}$$

Ejemplo: El problema del cambio

- $f(s)$: Cantidad mínima de monedas para entregar s centavos, para $s = 0, \dots, t$.

$$f(s) = \begin{cases} 0 & \text{si } s = 0 \\ \text{mín}_{i: a_i \leq s} 1 + f(s - a_i) & \text{si } a_k \leq s \\ \infty & \text{ninguno de los casos anteriores} \end{cases}$$

- **Teorema.** Si $f(s) < \infty$ entonces $f(s)$ es el valor óptimo del problema del cambio para entregar s centavos. Caso contrario no tiene solución.

Ejemplo: El problema del cambio

- $f(s)$: Cantidad mínima de monedas para entregar s centavos, para $s = 0, \ldots, t$.

$$f(s) = \begin{cases} 0 & \text{si } s = 0 \\ \min_{i:a_i \leq s} 1 + f(s - a_i) & \text{si } a_k \leq s \\ \infty & \text{ninguno de los casos anteriores} \end{cases}$$

- **Teorema.** Si $f(s) < \infty$ entonces $f(s)$ es el valor óptimo del problema del cambio para entregar s centavos. Caso contrario no tiene solución.
- ¿Cómo conviene implementar esta recursión?

Prueba del teorema

Sea $b_1, \cdots, b_p$ una solución de p monedas donde $b_j \in \{a_1, \cdots, a_k\}$ para $1 \leq j \leq p$ y $s = \sum_{i=1}^{p} b_j$. Claramente, $f(s) \leq 1 + f(s - b_1) \leq p + f(s - \sum_{i=1}^{p} b_j) = p$. Por lo tanto, si $f(s) = \infty$ entonces no hay solución. Ahora, si $f(s) = q < \infty$ entonces existen $c_1, \cdots, c_q$ de q monedas donde $c_j \in \{a_1, \cdots, a_k\}$ para $1 \leq j \leq q$ y $s = \sum_{i=1}^{q} c_j$ tal que $f(s) = \text{mín}_{i:a_i \leq s} 1 + f(s - a_i) = 1 + f(s - c_1) = \cdots = q + f(s - \sum_{i=1}^{q} c_j) = q + f(0) = q$. Es claro que q es el valor óptimo.

Ejemplo: Monedas bottom-up

constantes globales: `entero` k `y denominaciones` $a_1 \geq a_2 \geq \cdots \geq a_k$

algoritmo *Monedas*(t)
  entrada: `un entero` t
  salida: $f(t)$

  $A[0] \leftarrow 0$
  para $s = 1$ **hasta** t **hacer**
    $A[s] \leftarrow \infty$ **,** $i \leftarrow k$
    mientras $i > 0$ & $a_i \leq s$ **hacer**
      $A[s] \leftarrow \text{mín}\{A[s], A[s - a_i] + 1\}$
      $i \leftarrow i - 1$
    fin mientras
  fin para
  devolver $A[t]$

Ejemplo: Monedas top-down

constantes globales: `entero` k `y denominaciones` $a_1 \geq a_2 \geq \cdots \geq a_k$
variable global: `diccionario` A, $A(0) \leftarrow 0$

algoritmo $\mathtt{Monedas}(t)$
  entrada: `un entero` t
  salida: $f(t)$

  si `def?`(A, t) **entonces**
    devolver $A(t)$
  sino
    $a \leftarrow \infty$, $i \leftarrow k$
    mientras $i > 0$ & $a_i \leq t$ **hacer**
      $a \leftarrow \min\{a, \ \mathtt{Monedas}(t - a_i) + 1\}$
      $i \leftarrow i - 1$
    fin mientras
    $A(t) \leftarrow a$
    devolver a
  fin si

El problema de la mochila

Datos de entrada:

- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{Z}_+$ de objetos.
- Peso $p_i \in \mathbb{Z}_{>0}$ del objeto i, para $i = 1, \dots, n$.
- Beneficio $b_i \in \mathbb{Z}_+$ del objeto i, para $i = 1, \dots, n$.

Problema: Determinar qué objetos debemos incluir en la mochila sin excedernos del peso máximo C, de modo tal de maximizar el beneficio total entre los objetos seleccionados.