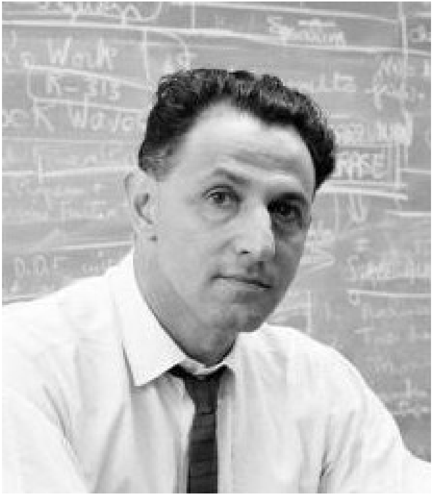
Técnicas de Diseño de Algoritmos (Ex Algoritmos y Estructuras de Datos III)

Segundo cuatrimestre 2025

Programación dinámica

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

- Al igual que *divide and conquer*, se divide el problema en subproblemas de tamaños menores que se resuelven recursivamente.

- **Ejemplo.** Cálculo de coeficientes binomiales. Si $n \geq 0$ y $0 \leq k \leq n$, definimos

$$\binom{n}{k} \;=\; \frac{n!}{k!(n-k)!}$$

- No es buena idea computar esta definición (¿por qué?).

- **Teorema.** Si $n \geq 0$ y $0 \leq k \leq n$, entonces

$$\binom{n}{k} \;=\; \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \end{cases}$$

Programación dinámica

Tampoco es buena idea implementar un algoritmo recursivo directo basado en esta fórmula (¿por qué?).

algoritmo $\mathtt{combinatorio}(n,k)$

entrada: dos enteros n y k

salida: $\binom{n}{k}$

si $k = 0$ **o** $k = n$ **hacer**

  retornar 1

si no

  $a := \mathtt{combinatorio}(n-1,\ k-1)$

  $b := \mathtt{combinatorio}(n-1,\ k)$

  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.
- 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: Cálculo de coeficientes binomiales

	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Ejemplo: Cálculo de coeficientes binomiales

```
algoritmo combinatorio(n,k)
    entrada: dos enteros n y k
    salida: (n k)

    para i = 1 hasta n hacer
        A[i][0] ← 1
    fin para
    para j = 0 hasta k hacer
        A[j][j] ← 1
    fin para
    para i = 2 hasta n hacer
        para j = 1 hasta min(i − 1,k) hacer
            A[i][j] ← A[i − 1][j − 1] + A[i − 1][j]
        fin para
    fin para
    retornar A[n][k]
```

Ejemplo: Cálculo de coeficientes binomiales

- Por definición:
  - Complejidad $O(n)$. Ojo! el tamaño de entrada es $O(\log n)$!
  - Inconvenientes inestabilidad numérica y/o manejo de enteros muy grandes.
- Función recursiva:
  - Complejidad $\Omega(\binom{n}{k})$.
- Programación dinámica (bottom-up):
  - Complejidad $O(nk)$.
  - Espacio $\Theta(k)$: sólo necesitamos almacenar la fila anterior de la que estamos calculando. No es posible con top-down
  - Se podría mejorar un poco sin cambiar la complejidad aprovechando que $\binom{n}{k} = \binom{n}{n-k}$.

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 $x_1, \ldots, x_k \in \mathbb{Z}_+$ 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 $s = 0, \ldots, t$.

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

- **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.

- ¿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.

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.

El problema de la mochila

- Definimos $m(k, D)$ = valor óptimo del problema con los primeros k objetos y una mochila de capacidad D.
- Podemos representar los valores de este parámetro en una tabla de dos dimensiones:

m	0	1	2	3	4	$\dots$	C
0	0	0	0	0	0	$\dots$	0
1	0						
2	0						
3	0						
4	0				$m(k, D)$		
$\vdots$	$\vdots$						
n	0						$m(n, C)$

El problema de la mochila

- Sea $S^* \subseteq \{1, \ldots, k\}$ una solución óptima para la instancia (k, D).
- $m(k, D) = \begin{cases} 0 & \text{si } k = 0 \\ 0 & \text{si } D = 0 \\ m(k-1, D) & \text{si } k \notin S^* \\ b_k + m(k-1, D - p_k) & \text{si } k \in S^* \end{cases}$
- Definimos entonces:
  1. $m(k, D) := 0$, si $k = 0$.
  2. $m(k, D) := m(k-1, D)$, si $k > 0$ y $p_k > D$.
  3. $m(k, D) := \max\{m(k-1, D), b_k + m(k-1, D - p_k)\}$, en caso contrario.
- **Teorema.** $m(n, C)$ es el valor óptimo para esta instancia del problema de la mochila.

El problema de la mochila

- Sea $S^* \subseteq \{1, \ldots, k\}$ una solución óptima para la instancia (k, D).
- $m(k, D) = \begin{cases} 0 & \text{si } k = 0 \\ 0 & \text{si } D = 0 \\ m(k-1, D) & \text{si } k \notin S^* \\ b_k + m(k-1, D - p_k) & \text{si } k \in S^* \end{cases}$
- Definimos entonces:
  1. $m(k, D) := 0$, si $k = 0$.
  2. $m(k, D) := m(k-1, D)$, si $k > 0$ y $p_k > D$.
  3. $m(k, D) := \text{máx}\{\underbrace{m(k-1, D)}_{(1)}, \underbrace{b_k + m(k-1, D - p_k)}_{(2)}\}$, ...
- **Teorema.** $m(n, C)$ es el valor óptimo para esta instancia del problema de la mochila.

Prueba del teorema

Primero ver que $m(k, 0) = 0$ es valor óptimo ya que no se puede guardar ningún objecto en una mochila con capacidad 0. Probar para capacidad > 0 por inducción en k. Caso base $k = 0$, no hay ningún objeto para considerar entonces $m(0, D) = 0$ es el valor óptimo. Supongamos que vale para instancias de primeros $k' - 1$ $(k' \geq 1)$ objetos. Ahora consideramos las instancias de primeros k' objetos. Si $p_{k'} > D$ entonces el objeto k'-ésimo no cabe en la mochila y en este caso es equivalente considerar la instancia de los primeros $k' - 1$ objetos con la mochila de capacidad D y por inducción el valor óptimo es $m(k' - 1, D) = m(k', D)$. En caso que $p_{k'} \leq D$, hay dos posibilidades: (i) descartamos el k'-ésimo objeto entonces es equivalente al caso anterior o (ii) incluimos el k'-ésimo objeto en la mochila asegurando al menos el beneficio $b_{k'}$ y considerar a continuación los primeros $k' - 1$ objetos en una mochila con capacidad remanente de $D - p_{k'}$ para sumar como beneficio adicional su valor óptimo $m(k' - 1, D - p_{k'})$ (por inducción). Consecuentemente, el valor óptimo de la instancia es $m(k', D) = \max\{m(k' - 1, D), b_{k'} + m(k' - 1, D - p_{k'})\}$.

El problema de la mochila

- ¿Cuál es la complejidad computacional de este algoritmo?
  - Supongamos que la tabla se representa con una matriz en memoria, de modo tal que cada acceso y modificación es $O(1)$.
- Si debemos completar $(n+1)(C+1)$ entradas de la matriz, y cada entrada se completa en $O(1)$, entonces la complejidad del procedimiento completo es $O(nC)$ (?).
- **Algoritmo pseudopolinomial:** Su tiempo de ejecución está acotado por un polinomio en los valores numéricos del input, en lugar de un polinomio en la longitud del input.

El problema de la mochila

- El cálculo de $m(k, D)$ proporciona el valor óptimo, pero no la solución óptima.
- Si necesitamos el conjunto de objetos que realiza el valor óptimo, debemos reconstruir la solución.

	$\dots$	$D - p_k$	$\dots$	D	$\dots$
$\vdots$					
$k-1$		$m(k-1, D-p_k)$	$\dots$	$m(k-1, D)$	
k				$m(k, D)$	
$\vdots$					

Programación dinámica - Subsecuencia común más larga

- Dada una secuencia A, una subsecuencia se obtiene eliminando cero o más símbolos de A.
  - Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $A = [4, 7, 8, 2, 5, 3]$, pero $[2, 7]$ no lo es.
- **Problema.** Encontrar la subsecuencia común mas larga (scml) de dos secuencias dadas.
- Es decir, dadas dos secuencias A y B, queremos encontrar la mayor secuencia que es tanto subsecuencia de A como de B.
- Por ejemplo, si $A = [9, 5, 2, 8, 7, 3, 1, 6, 4]$ y $B = [2, 9, 3, 5, 8, 7, 4, 1, 6]$ las scml es $[9, 5, 8, 7, 1, 6]$.
- ¿Cómo es un algoritmo de fuerza bruta para este problema?

Programación dinámica - Subsecuencia común más larga

Dadas las dos secuencias $A = [a_1, \ldots, a_r]$ y $B = [b_1, \ldots, b_s]$, consideremos dos casos:

- $a_r = b_s$: La scml entre A y B se obtiene colocando al final de la scml entre $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_{s-1}]$ al elemento a_r $(= b_s)$. ¡Habría que probar esta afirmación!
- $a_r \neq b_s$: La scml entre A y B será la más larga entre estas dos opciones:
  1. la scml entre $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_s]$,
  2. la scml entre $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$.

  Es decir, calculamos el problema aplicado a $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_s]$ y, por otro lado, el problema aplicado a $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$, y nos quedamos con la más larga de ambas. ¡También probar la correctitud!

Programación dinámica - Subsecuencia común más larga

Esta forma recursiva de resolver el problema ya nos conduce al algoritmo.

Si llamamos $l[i][j]$ a la longitud de la scml entre $[a_1., \ldots, a_i]$ y $[b_1., \ldots, b_j]$, entonces:

- $l[0][0] = 0$
- Para $j = 1, \ldots, s$, $l[0][j] = 0$
- Para $i = 1, \ldots, r$, $l[i][0] = 0$
- Para $i = 1, \ldots, r$, $j = 1, \ldots, s$
  - si $a_i = b_j$: $l[i][j] = l[i-1][j-1] + 1$
  - si $a_i \neq b_j$: $l[i][j] = \text{máx}\{l[i-1][j], l[i][j-1]\}$

Y la solución del problema será $l[r][s]$.

Programación dinámica - Subsecuencia común más larga

$\mathtt{scml}(A,B)$

entrada: A, B `secuencias`

salida: `longitud de a scml entre` A `y` B

$l[0][0] \leftarrow 0$

para $i = 1$ **hasta** r **hacer** $l[i][0] \leftarrow 0$

para $j = 1$ **hasta** s **hacer** $l[0][j] \leftarrow 0$

para $i = 1$ **hasta** r **hacer**

  para $j = 1$ **hasta** s **hacer**

    si $A[i] = B[j]$

      $l[i][j] \leftarrow l[i-1][j-1] + 1$

    sino

      $l[i][j] \leftarrow \max\{l[i-1][j], l[i][j-1]\}$

    fin si

  fin para

fin para

retornar $l[r][s]$