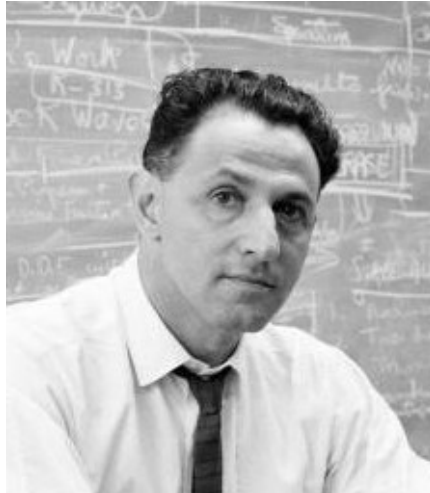


PROGRAMACIÓN DINÁMICA

Tecnología Digital V: Diseño de Algoritmos

Universidad Torcuato Di Tella



Richard Bellman (1920–1984)

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)

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

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

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

```
int combinatorio(int n, int k)
{
    if (k==0 || k==n)
    {
        return 1;
    }
    else
    {
        int a = combinatorio(n-1, k-1);
        int b = combinatorio(n-1, k);
        return a+b;
    }
}
```

- **Superposición de estados:** El árbol de llamadas recursivas resuelve el mismo problema varias veces.
 1. 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 (**memoización**). 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 todos los resultados (**tableau-filling**).

Ejemplo: Cálculo de coeficientes binomiales

	0	1	2	3	4	...	$k-1$	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							

```
int combinatorio(int n, int k)
{
    int **A = crearMatriz(n+1, k+1);

    for (int i=1; i<=n; ++i)
        A[i][0] = 1;

    for (int j=1; j<=k; ++j)
        A[j][j] = 1;

    for (int i=2; i<=n; ++i)
        for (int j=1; j<=i-1 && j<=k; ++j)
            A[i][j] = A[i-1][j-1] + A[i-1][j];

    return A[n][k];
}
```


- Función recursiva:
 - Complejidad exponencial
- Programación dinámica (bottom-up):
 - Complejidad $O(nk)$.
 - Espacio $\Theta(k)$: sólo necesitamos almacenar la fila anterior de la que estamos calculando.



Programación dinámica

$$\text{PD} = \text{Recursión con sup. de estados} + \left\{ \begin{array}{l} \text{memoización} \\ \text{tableau-filling} \end{array} \right\}$$

Ejemplo

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 $a_1, \dots, a_k \in \mathbb{Z}_+$ de monedas (con $a_i > a_{i+1}$ para $i = 1, \dots, k - 1$) y un objetivo $t \in \mathbb{Z}_+$, encontrar $x_1, \dots, x_k \in \mathbb{Z}_+$ tales que

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

minimizando $x_1 + \dots + x_k$.

- Podemos plantear una función recursiva para este problema.

Definición

Para $s = 0, \dots, t$, definimos $f(s)$ como la cantidad mínima de monedas para entregar s centavos.

$$f(s) = \begin{cases} 0 & \text{si } s = 0 \\ \min_{i: a_i \leq s} 1 + f(s - a_i) & \text{en caso contrario} \end{cases}$$

Preguntas

- ¿Podemos plantear un algoritmo recursivo directamente a partir de esta función?
- ¿Cómo conviene implementar esta recursión?

```
int cambio(int s)
{
    if (s == 0)
        return 0;

    int ret = infinito;
    for (int i=0; i<k; ++i)
    {
        if (a[i] <= s)
            ret = min(ret, 1 + cambio(s-a[i]));
    }

    return ret;
}
```

Ejemplo: El problema del cambio (top-down + memoization)

```
int cambio(int s, int* M)
{
    if (s == 0)
        return 0;

    if (M[s] >= 0) // Inicializado con -1's
        return M[s];

    int ret = infinito;
    for (int i=0; i<k; ++i)
    {
        if (a[i] <= s)
            ret = min(ret, 1 + cambio(s-a[i], M));
    }

    M[s] = ret;
    return ret;
}
```

Ejemplo: El problema del cambio (bottom-up)

```
int cambio(int t)
{
    int *M = new int[t+1];
    M[0] = 0;

    for (int s=1; s<=t; ++s)
    {
        int ret = infinito;
        for (int i=0; i<k; ++i)
        {
            if (a[i] <= s)
                ret = min(ret, 1 + M[s-a[i]]);
        }
        M[s] = ret;
    }

    return M[t];
}
```

Knapsack-01 (KP-01)

Debemos llenar una mochila eligiendo entre varios objetos posibles. Cada producto tiene un peso, una medida de comfort (beneficio) y la mochila tolera un peso máximo de carga. **Los objetos no pueden ser fraccionados, y solo se puede elegir una unidad de cada objeto.**

Una instancia del KP-01 está dada por

- $N = \{1, \dots, n\}$ el conjunto de objetos (o productos).
- $p_i \in \mathbb{Z}_+$ el peso del objeto i , para $i = 1, \dots, n$.
- $b_i \in \mathbb{Z}_+$ el beneficio del objeto i , para $i = 1, \dots, n$.
- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).

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.

Mochila($k : \mathbb{Z}, c : \mathbb{Z}$)

if $k == 0 \parallel c == 0$ **then**

▷ Estamos en una hoja

return 0

else

▷ Falta considerar más elementos

$v_{\text{without}} = \text{Mochila}(k - 1, c)$

$v_{\text{with}} = -\infty$

if $p_k \leq c$ **then**

$v_{\text{with}} = b_k + \text{Mochila}(k - 1, c - p_k);$

end if

return $\max\{v_{\text{with}}, v_{\text{without}}\}$

end if

- Adaptamos el código para que retorne la función objetivo óptima.
- Podemos ir calculando el costo como parte de la recursión.
- Iniciamos la recursión con $\text{Mochila}(N, C)$.

Instancia

- $n = 8$
- $b = (b_i) = (15, 100, 90, 60, 40, 15, 10, 1)$
- $p = (p_i) = (2, 20, 20, 30, 40, 30, 30, 10)$
- $C = 102$

Solución óptima

- $S = \{1, 2, 3, 4, 6\}$
- $z^* = 280$
- Capacidad usada: 102

Preguntas

Analizando el árbol de enumeración:

- Si definimos $f(i) = \text{máximo beneficio considerando el subconjunto de ítems } \{1, \dots, i\}$, qué podemos decir respecto a la definición recursiva en términos de $f(i - 1)$?
- Tenemos superposición de estados?
- En un determinado nivel del árbol, ¿qué nos importa en relación a la solución óptima del subproblema correspondiente?

Principio de optimalidad

Decimos que un problema exhibe una *subestructura óptima* si la solución óptima puede ser formulada a partir de las soluciones óptimas de los subproblemas. Cuando se cumple esta característica, puede ser un buen indicio para utilizar programación dinámica.

- En *coeficientes binomiales*, $\text{Comb}(n, k)$ se define en función de $\text{Comb}(n - 1, k - 1)$ y $\text{Comb}(n - 1, k)$.
- En *el problema del cambio*, $\text{CMP}(s)$ se define en función de $\text{CMP}(s - a_j)$ para ciertos valores de a_j . **A lo sumo, tantos como denominaciones.**

Para KP01, definimos $m(k, c)$ = valor óptimo del problema con los primeros $\{1, \dots, k\}$ y capacidad remanente c y tenemos la siguiente recursión:

$$m(k, c) = \max \left\{ \underbrace{m(k-1, c)}_{\text{no selec. } k}, \underbrace{b_k + m(k-1, c - p_k)}_{\text{selec. } k} \right\}, \quad k = 1, \dots, n, c > 0$$

Definimos $m(k, c)$ como una tabla de doble entrada que contiene el valor óptimo del problema con los primeros $\{1, \dots, k\}$ y capacidad remanente c . Sea $S^* \subseteq \{1, \dots, k\}$ una solución óptima para la instancia (k, c)

m	0	1	2	3	4	...	C
0	0	0	0	0	0	...	0
1	0						
2	0						
3	0						
4	0				$m(k, c)$		
⋮	⋮						
n	0						$m(n, C)$

donde

$$m(k, c) = \begin{cases} 0 & \text{si } k = 0 \\ 0 & \text{si } c = 0 \\ m(k-1, c) & \text{si } k \notin S^* \\ b_k + m(k-1, c - p_k) & \text{si } k \in S^* \end{cases}$$

- Asumimos $m(k, c) = \text{null}$.
- La función es invocada inicialmente con $\text{Mochila}(n, C)$.

$\text{Mochila}(k : \mathbb{Z}, c : \mathbb{Z}, m)$

if $k == 0 \parallel c == 0$ **then**

▷ Casos base

$m(k, c) = 0$

return 0

else

▷ Falta considerar más elementos

if $m(k, c) \neq \text{null}$ **then**

return $m(k, c)$

else

$v_{\text{without}} = \text{Mochila}(k - 1, c, m)$

$v_{\text{with}} = -\infty$

if $p_k \leq c$ **then**

$v_{\text{with}} = b_k + \text{Mochila}(k - 1, c - p_k, m);$

end if

$m(k, c) = \text{máx}\{v_{\text{with}}, v_{\text{without}}\}$

return $m(k, c)$

end if

end if

○ Sea $S^* \subseteq \{1, \dots, k\}$ una solución óptima para la instancia (k, c) .

$$\text{○ } m(k, c) = \begin{cases} 0 & \text{si } k = 0 \\ 0 & \text{si } c = 0 \\ m(k-1, c) & \text{si } k \notin S^* \\ b_k + m(k-1, c - p_k) & \text{si } k \in S^* \end{cases}$$

○ Definimos entonces:

1. $m(k, c) := 0$, si $k = 0$ o $c = 0$.
2. $m(k, c) := m(k-1, c)$, si $k > 0$ y $p_k > c$.
3. $m(k, c) := \max\{m(k-1, c), b_k + m(k-1, c - p_k)\}$, en caso contrario.

Teorema

$m(n, C)$ es el valor óptimo para esta instancia del KP01.

Analizamos qué valores necesitamos para obtener una determinada entrada de la tabla $m(k, c)$.

	...	$c - p_k$...	c	...
⋮					
$k - 1$		$m(k - 1, c - p_k)$...	$m(k - 1, c)$	
k				$m(k, c)$	
⋮					

Pregunta

En qué orden tendríamos que llenar la tabla?

```
int knapsack(int n, int C, int* p, int* b)
{
    int **m = crearMatriz(n+1, C+1);

    for (int i=0; i<=n; ++i)
        m[i][0] = 0;

    for (int c=0; c<=C; ++c)
        m[0][c] = 0;

    for (int k=1; k<=n; ++k)
        for (int c=1; c<=C; ++c)
            if (p[k] > c)
                m[k][c] = m[k-1][c];
            else
                m[k][c] = max(m[k-1][c], b[k] + m[k-1][c - p[k]]);

    return m[n][C];
}
```


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

	...	$c - p_k$...	c	...
⋮					
$k - 1$		$m(k - 1, c - p_k)$...	$m(k - 1, c)$	
k				$m(k, c)$	
⋮					

Preguntas

- Cuál es el punto de inicio?
- Qué decidimos en cada paso?
- Cuántos pasos debemos ejecutar?
- Cuál es el criterio de corte?

- Dada una secuencia A , una **subsecuencia** se obtiene eliminando cero o más símbolos de A .
 1. 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?

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

- $a_r = b_s$: La scm l entre A y B se obtiene colocando al final de la scm l entre $[a_1, \dots, a_{r-1}]$ y $[b_1, \dots, b_{s-1}]$ al elemento $a_r (= b_s)$.
- $a_r \neq b_s$: La scm l entre A y B será la más larga entre estas dos opciones:
 1. la scm l entre $[a_1, \dots, a_{r-1}]$ y $[b_1, \dots, b_s]$,
 2. la scm l entre $[a_1, \dots, a_r]$ y $[b_1, \dots, b_{s-1}]$.

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

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

Si llamamos $l[i][j]$ a la longitud de la scm1 entre $[a_1, \dots, a_i]$ y $[b_1, \dots, b_j]$, entonces:

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

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

scml(*A*, *B*)

entrada: *A*, *B* secuencias

salida: longitud de la 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 $l[r][s]$