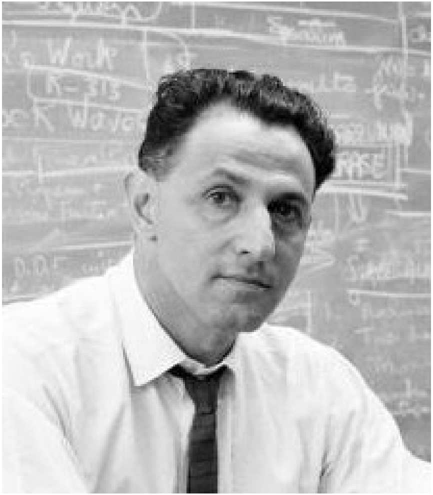
PROGRAMACIÓN DINÁMICA

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

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

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

Algoritmo recursivo basado en el teorema:

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

Tampoco es buena idea implementar ese algoritmo (¿por qué?).

- Problema: El árbol de llamadas recursivas resuelve el mismo problema varias veces.
  - En general, podemos decir que es común encontrar situaciones con algoritmos basados en recursión donde se realizan muchas veces llamadas una función recursiva f con los mismos parámetros p.
    - En el cálculo de coeficientes binomiales, ¿qué es p? ¿Y qué es f?
  - ¿Cómo podemos evitar los cálculos repetidos?
- Solución: Programación Dinámica
  - Almacenamos la respuesta la función $f(p)$ cuando la evaluamos para cada p.
  - Los valores de parámetros p se denominam estados. La evaluación de $f(p)$ varias veces con el mismo p se denomina superposiciones de estados.
  - Hay un orden parcial en los valores de p, ya que la idea es que la recursión vaya de problemas grandes a problemas menores.
  - ¿Cómo guardar los resultados ya calculados de $f(p)$?

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

```
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];
}
```

Ejemplo: Cálculo de coeficientes binomiales

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

Programación Dinámica: Dos Opciones

1. **Enfoque top-down.** Se implementa recursivamente a partir del problema más grande/inicial, 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).

Programación dinámica

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

Definición

Un problema de optimización consiste en encontrar la mejor solución dentro de un conjunto:

$$z^* = \max_{x \in S} f(x) \quad \text{o bien} \quad z^* = \min_{x \in S} f(x)$$

donde

- La función $f : S \to \mathbb{R}$ se denomina función objetivo del problema.
- El conjunto S es la región factible y los elementos $x \in S$ se llaman soluciones factibles.
- El valor $z^* \in \mathbb{R}$ es el valor óptimo del problema, y cualquier solución factible $x^* \in S$ tal que $f(x^*) = z^*$ se llama un óptimo o solución óptima del problema.

¿Cómo resolverlos con programación dinámica?

Programación dinámica: otro concepto fundamental

Prinicipio 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.

"Si el camino más corto entre A y C pasa por B, entonces el tramo entre B y C en ese camino también es el camino más corto entre B y C."

Ejemplo: Camino más corto en un grafo

- Si $v_i, v_1, v_2, \ldots, v_k, v_f$ es el camino más corto entre v_i y v_f, entonces $v_1, v_2, \ldots, v_k, v_f$ es el camino más corto entre v_2 y v_f.
  - De hecho cada subcamino de $v_i, v_1, v_2, \ldots, v_k, v_f$ es el camino más corto entre el primer y último nodo de ese subcamino!
- Definimos $z^*(v)$ como la distancia mínima entre cada nodo v y v_f.
- Definición recursiva: $z^*(v) = \min_{w \in V,\ w \neq v} \{1 + z^*(w)\}$, con $z^*(v_f) = 0$.
- $z^*(v_i)$ es el valor óptimo del problema de camino mínimo con nodo inicial v_i y nodo final v_f.
- El parámetro/estado del problema es el nodo inicial v.
- El camino mínimo $v_i, v_1, v_2, \ldots, v_k, v_f$ es la solución óptima de ese problema.

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 $t \in \mathbb{Z}_+$, resolver: $x_1, \dots, x_k \in \mathbb{Z}_+$ tales que

$$\min_{x_1,\dots,x_k \in \mathbb{Z}_+} \sum_{i=1}^{k} x_i$$

sujeto a

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

Ejemplo: El problema del cambio

Definición Recursiva

Para $s = 0, \ldots, t$, definimos $z^*(s)$ como la cantidad mínima de monedas para entregar s centavos.

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

- $z^*(s)$ es el valor óptimo del problema para cada vuelto s.
- El parámetro/estado del problema es vuelto s.
- La cantidad de monedas de cada denominación $x_1^*, x_2^*, \ldots \ldots, x_k^*$ es la solución óptima de ese problema.

Preguntas

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

Ejemplo: El problema del cambio

```
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* Z)
{
    if (s == 0)
        return 0;

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

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

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

Ejemplo: El problema del cambio (bottom-up)

```
int cambio(int t)
{
    int *Z = new int[t+1];
    Z[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 + Z[s-a[i]]);
        }
        Z[s] = ret;
    }

    return Z[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

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

Ejemplo: El problema de la mochila

Definición Recursiva

Definamos $z^*(k,c)$ como el valor óptimo del problema KP01 con los primeros $\{1,\ldots,k\}$ objetos y capacidad remanente c. Entonces:

1. $z^*(k,c) = 0$, si $k = 0$ o $c = 0$.
2. $z^*(k,c) = z^*(k-1,c)$, si $k > 0$ y $p_k > c$.
3. $z^*(k,c) = \text{máx}\{z^*(k-1,c), b_k + z^*(k-1,c-p_k)\}$, en caso contrario.

- $z^*(k,c)$ es el valor óptimo del problema KP01 con k objetos y peso máximo c.
- El parámetro/estado del problema es la tupla (k,c).
- El subconjunto $S^* \subset \{1,2,\ldots,n\}$ de objetos a incluir en la mochila es la solución óptima de ese problema.

Preguntas

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

El problema de la mochila: algoritmo recursivo

```
MOCHILA(k : ℤ, c : ℤ)
if k == 0 || c == 0 then                          ▷ Estamos en una hoja
    return 0
else                                              ▷ Falta considerar más elementos
    v_without = MOCHILA(k − 1, c)
    v_with = −∞
    if p_k ≤ c then
        v_with = b_k + MOCHILA(k − 1, c − p_k);
    end if
    return max{v_with, v_without}
end if
```

- Observen que Mochila(k,c) retorna el valor óptimo $z^*(k, c)$, no la solución óptima.
- Iniciamos la recursión con $\text{MOCHILA}(N,\ C)$.

El problema de la mochila: ejemplo

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:

- ¿Cuántas veces llamamos la función Mochila?
- ¿Tenemos superposición de estados?
- ¿Podemos mejorar la implementación con programación dinámica?

KP01: DP top-down + memoization

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

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

```
if k == 0 || c == 0 then                                   ▷ Casos base
    Z(k, c) = 0
    return 0
else                                     ▷ Falta considerar más elementos
    if Z(k, c) ≠ null then
        return Z(k, c)
    else
        v_without = MOCHILA(k − 1, c, m)
        v_with = −∞
        if p_k ≤ c then
            v_with = b_k + MOCHILA(k − 1, c − p_k, m);
        end if
        Z(k, c) = máx{v_with, v_without}
        return Z(k, c)
    end if
end if
```

KP01: DP bottom-up (table filling)

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

	$\ldots$	$c - p_k$	$\ldots$	c	$\ldots$
$\vdots$					
$k-1$		$z^*(k-1, c-p_k)$	$\ldots$	$z^*(k-1, c)$	
k				$z^*(k, c)$	
$\vdots$					

Pregunta

En qué orden tendriamos que llenar la tabla?

KP01: DP bottom-up (table filling)

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

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

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

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

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

KP01: DP bottom-up (table filling)

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.

KP01: reconstrucción de la solución

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

Teorema

Para la instancia con objectos $\{1, 2, \ldots, k\}$ y capacidad remanente c, el objeto k está en la solución óptima si y solo si $z^*(k, c) = z^*(k-1, c-p_k)$.

Preguntas

- ¿Cómo calcular la solución óptima a partir de z^*?
- ¿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 $A = [a_1, \ldots, a_r]$ 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 $(= b_s)$.
- $a_r \neq b_s$: 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.

Definición Recursiva

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

- $z^*(0,0) = 0$
- Para $j = 1, \ldots, s$, $z^*(0,j) = 0$
- Para $i = 1, \ldots, r$, $z^*(i,0) = 0$
- Para $i = 1, \ldots, r$, $j = 1, \ldots, s$
  - si $a_i = b_j$: $z^*(i,j) = z^*(i-1, j-1) + 1$
  - si $a_i \neq b_j$: $z^*(i,j) = max\{z^*(i-1,j) z^*(i,j-1)\}$

- $z^*(i,j)$ es el valor óptimo del problema scml con secuencias $[a_1., \ldots, a_i]$ y $[b_1., \ldots, b_j]$.
- El parámetro/estado del problema es la tupla (i,j).
- La scml $[a_{k_1}, \ldots, a_{k_{z^*(r,s)}}]$ es la solución óptima de ese problema.

scml(A, B)
entrada: A, B `secuencias`
salida: `longitud de a scml entre` A `y` B

$Z[0][0] \leftarrow 0$
para $i = 1$ **hasta** r **hacer** $Z[i][0] \leftarrow 0$
para $j = 1$ **hasta** s **hacer** $Z[0][j] \leftarrow 0$
para $i = 1$ **hasta** r **hacer**
  para $j = 1$ **hasta** s **hacer**
    si $A[i] = B[j]$
      $Z[i][j] \leftarrow Z[i-1][j-1] + 1$
    sino
      $Z[i][j] \leftarrow \max\{Z[i-1][j], Z[i][j-1]\}$
    fin si
  fin para
fin para
retornar $Z[r][s]$