Algoritmos y Estructuras de Datos III

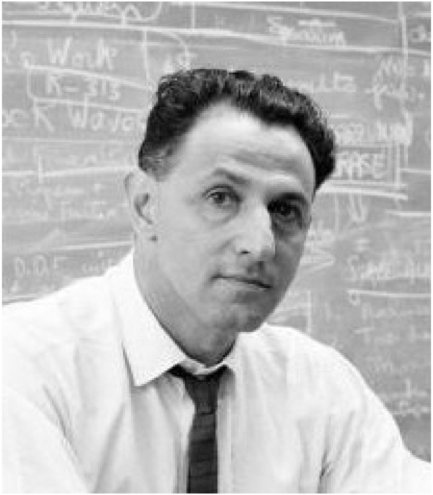
Primer cuatrimestre 2022

Técnicas de diseño de algoritmos

Técnicas de diseño de algoritmos

- *Divide and conquer* (AED2)
- Fuerza bruta y *backtracking*
- Programación dinámica
- Heurísticas y algoritmos aproximados
- Algoritmos golosos
- Algoritmos probabilísticos

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

- **Principio de optimalidad de Bellman**:
  Un problema de optimización satisface el principio de optimalidad de Bellman si en una sucesión óptima de decisiones, cada subsucesión es a su vez óptima.
- Es decir, si miramos una subsolución de la solución óptima, debe ser solución del subproblema asociado a esa subsolución.
- El principio de optimalidad es condición necesaria para poder usar programación dinámica.

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 $p_i \in \mathbb{Z}_+$ del objeto i, para $i = 1, \ldots, n$.
- Beneficio $b_i \in \mathbb{Z}_+$ del objeto i, para $i = 1, \ldots, n$.

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	...	C
0	0	0	0	0	0	...	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).
- $m(k, D) = \begin{cases} 0 & \text{si } k = 0 \\ 0 & \text{si } D = 0 \\ m(k-1, D) & \text{si } k \notin S^* \\ b_k + m(k-1, D - p_k) & \text{si } k \in S^* \end{cases}$
- Tenemos entonces que:
  1. $m(k, D) = 0$, si $k = 0$ o $D = 0$.
  2. $m(k, D) = m(k-1, D)$, si $p_k > D$.
  3. $m(k, D) = \max\{m(k-1, D), b_k + m(k-1, D - p_k)\}$, en caso contrario.

El problema de la mochila

- ¿Cuál es la complejidad computacional de este algoritmo?
    1. 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 - Multiplicación de matrices

- **Problema:** Dadas $M_1, \dots, M_n$, calcular

$$M = M_1 \times M_2 \times \dots M_n$$

  realizando la menor cantidad de multiplicaciones entre números de punto flotante.

- Por ejemplo, si $A \in \mathbb{R}^{13\times5}$, $B \in \mathbb{R}^{5\times89}$, $C \in \mathbb{R}^{89\times3}$ y $D \in \mathbb{R}^{3\times34}$, tenemos que

  1. $((AB)C)D$ requiere 10582 multiplicaciones,
  2. $(AB)(CD)$ requiere 54201 multiplicaciones,
  3. $(A(BC))D$ requiere 2856 multiplicaciones,
  4. $A((BC)D)$ requiere 4055 multiplicaciones,
  5. $A(B(CD))$ requiere 26418 multiplicaciones.

Programación dinámica - Multiplicación de matrices

- Para multiplicar todas las matrices de forma óptima, deberemos multiplicar las matrices 1 a i por un lado y las matrices $i+1$ a n por otro lado y luego multiplicar estos dos resultados, para algún $1 \leq i \leq n-1$, que es justamente lo que queremos determinar.
- Estos dos subproblemas, $M_1 \times M_2 \times \ldots M_i$ y $M_{i+1} \times M_{i+2} \times \ldots M_n$ deben estar resueltos, a su vez, de forma óptima, es decir realizando la mínima cantidad de operaciones.

Programación dinámica - Multiplicación de matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$. Suponemos que las dimensiones de las matrices están dadas por un vector $d \in \mathbb{N}^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas, para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \ldots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \ldots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- Para $s = 2, \ldots, n-1$, $i = 1, 2, \ldots, n-s$,

  $$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

La solución del problema es $m[1][n]$.

Programación dinámica - Subsecuencia común más larga

Dada una secuencia, una subsecuencia se obtiene eliminando 0 o más símbolos. Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ 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 $[9, 5, 8, 7, 1, 6]$.
- Si resolvemos este problema por fuerza bruta, listaríamos todas las subsecuencias de S_1, todas las de S_2, nos fijaríamos cuales tienen en común, y entre esas elegiríamos la más larga.

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 $(= b_s)$.
- $a_r \neq b_s$: 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.

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

$scml(A, B)$

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 $l[r][s]$

Heurísticas

- Una heurística es un procedimiento computacional que intenta obtener soluciones de buena calidad para un problema, intentando que su coportamiento sea lo más preciso posible.
- Por ejemplo, una heurística para un problema de optimización obtiene una solución con un valor que se espera sea cercano (idealmente igual) al valor óptimo.
- Decimos que A es un algoritmo ϵ-aproximado ($\epsilon > 0$) para un problema si

$$\left| \frac{x_A - x_{OPT}}{x_{OPT}} \right| \leq \epsilon.$$

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- Habitualmente, proporcionan heurísticas sencillas para problemas de optimización.
- En general permiten construir soluciones razonables (pero sub-óptimas) en tiempos eficientes.
- Sin embargo, en ocasiones nos pueden dar interesantes sorpresas!

Ejemplo: El problema de la mochila

Datos de entrada:

- Capacidad $C \in \mathbb{Z}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{N}$ de objetos.
- Peso $p_i \in \mathbb{Z}_+$ 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.

Ejemplo: El problema de la mochila

- **Algoritmo(s) goloso(s):** Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto i que ...
  - ... tenga mayor beneficio b_i.
  - ... tenga menor peso p_i.
  - ... maximice b_i/p_i.
- ¿Qué podemos decir en cuanto a la calidad de las soluciones obtenidas por estos algoritmos?
- ¿Qué podemos decir en cuanto a su complejidad?
- ¿Qué sucede si se puede poner una fracción de cada elemento en la mochila?

Ejemplo: El problema del cambio

- **Problema:** 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.
- **Algoritmo goloso:** Seleccionar la moneda de mayor valor que no exceda la cantidad restante por devolver, agregar esta moneda a la lista de la solución, y sustraer la cantidad correspondiente a la cantidad que resta por devolver (hasta que sea 0).

Ejemplo: El problema del cambio

```
darCambio(cambio)
      entrada: cambio ∈ ℕ
      salida: M conjunto de enteros

      suma ← 0
      M ← {}
      mientras suma < cambio hacer
            proxima ← masgrande(cambio,suma)
            M ← M ∪ {proxima}
            suma ← suma + proxima
      fin mientras
      retornar M
```

Ejemplo: El problema del cambio

- Este algoritmo siempre produce la mejor solución para estos valores de monedas, es decir, retorna la menor cantidad de monedas necesarias para obtener el valor *cambio*.
- Sin embargo, si también hay monedas de 12 centavos, puede ocurrir que el algoritmo no encuentre una solución óptima: si queremos devolver 21 centavos, el algoritmo retornará una solución con 6 monedas, una de 12 centavos, 1 de 5 centavos y cuatro de 1 centavos, mientras que la solución óptima es retornar dos monedas de 10 centavos y una de 1 centavo.
- El algoritmo es goloso porque en cada paso selecciona la moneda de mayor valor posible, sin preocuparse que esto puede llevar a una mala solución, y nunca modifica una decisión tomada.

Ejemplo: Tiempo de espera total en un sistema

Problema: Un servidor tiene n clientes para atender, y los puede atender en cualquier orden. Para $i = 1, \dots, n$, el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. El objetivo es determinar en qué orden se deben atender los clientes para minimizar la suma de los tiempos de espera de los clientes.

Si $I = (i_1, i_2, \dots, i_n)$ es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera es

$$
\begin{aligned}
T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \dots \\
&= \sum_{k=1}^{n} (n-k) t_{i_k}.
\end{aligned}
$$

Ejemplo: Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- Retorna una permutación $I_{\mathrm{GOL}} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$.
- ¿Cuál es la complejidad de este algoritmo?
- Este algoritmo proporciona la solución óptima!

Algoritmos probabilísticos

- Algoritmos numéricos probabilísticos: Algoritmos basados en simulaciones, que encuentran una solución aproximada para problemas numéricos.
  - A mayor tiempo de proceso, mayor precisión en la respuesta.
  - Ejemplo: Cálculo de π arrojando dardos virtuales a un círculo inscripto en un cuadrado.
- Algoritmos de Montecarlo: Algoritmos que siempre proporcionan una respuesta, pero que puede no ser la correcta. La respuesta es correcta con alta probabilidad.
  - A mayor tiempo de proceso, mayor probabilidad de dar una respuesta correcta.
  - Ejemplo: determinar la existencia en un arreglo de un elemento mayor a un valor dado.

Algoritmos probabilísticos

- Algoritmos de Las Vegas: Algoritmos que si dan una respuesta entonces es correcta, pero pueden no dar ninguna respuesta.
  - A mayor tiempo de proceso, mayor probabilidad de obtener respuesta.
  - Ejemplo: Problema de las n damas.
- Algoritmos de Sherwood: Algoritmos que "aleatorizan" un algoritmo determinístico donde hay una gran diferencia entre el peor caso y caso promedio.
  - Se espera que la aleatorización permita estar en un caso promedio con alta probabilidad, evitando así los peores casos.
  - Ejemplo: Quicksort con pivote seleccionado aleatoriamente.