Programación dinámica

Clase 16

IIC 2133 - Sección 1

Prof. Sebastián Bugedo

Sumario

Obertura

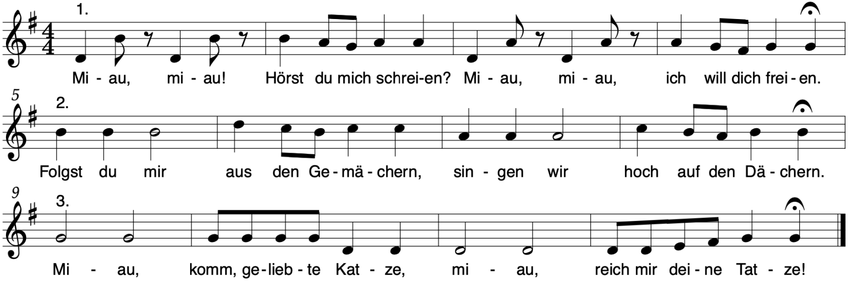
Programación dinámica

Dos aplicaciones

Epílogo

¿Cómo están?

Miau aus Frankreich



Miau, miau, hörst du mich schreien?
Miau, miau, ich will dich freien.

Folgst du mir aus den Gemächern,
singen wir hoch auf den Dächern.

Miau, komm, geliebte Katze,
miau, reich mir deine Tatze!

Tercer Acto: Los jinetes de la salvación

Estrategias de diseño de algoritmos

Playlist 3

Playlist: DatiWawos Tercer Acto

Además sigan en instagram:
@orquesta_tamen

Programación de charlas 2.0

Consideremos el problema de asignar charlas en una misma sala

- Tenemos n charlas por asignar
- La charla i tiene hora de inicio s_i y de término f_i
- Es decir, se define el intervalo de tiempo $[s_i, f_i)$

Solo se puede realizar **una charla a la vez**

ADEMÁS: si la charla i es realizada, produce una ganancia v_i

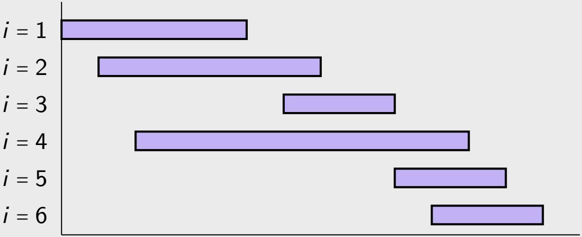
¿Qué charlas asignar de manera que maximicemos la ganancia?

Programación de charlas 2.0

Ejemplo

Sean las siguientes charlas con sus intervalos y ganancias

- $i = 1,\ [0, 5),\ v_1 = 2$
- $i = 2,\ [1, 7),\ v_2 = 4$
- $i = 3,\ [6, 9),\ v_3 = 4$
- $i = 4,\ [2, 11),\ v_4 = 7$
- $i = 5,\ [9, 12),\ v_5 = 2$
- $i = 6,\ [10, 13),\ v_6 = 1$

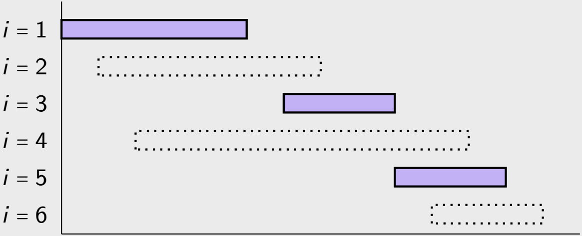


Programación de charlas 2.0

Ejemplo

El caso que sabemos resolver consideraba $v_i = c$ para cada charla

- $i = 1,\ [0, 5),\ v_1 = c$
- $i = 2,\ [1, 7),\ v_2 = c$
- $i = 3,\ [6, 9),\ v_3 = c$
- $i = 4,\ [2, 11),\ v_4 = c$
- $i = 5,\ [9, 12),\ v_5 = c$
- $i = 6,\ [10, 13),\ v_6 = c$



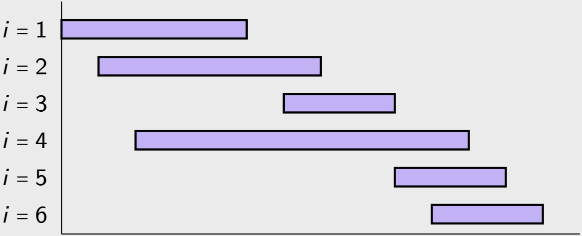
Cuando la ganancia es la misma, la estrategia codiciosa de **elegir la charla que termina antes** es óptima

Programación de charlas 2.0

Ejemplo

Sean las siguientes charlas con sus intervalos y ganancias

- $i = 1,\ [0, 5),\ v_1 = 2$
- $i = 2,\ [1, 7),\ v_2 = 4$
- $i = 3,\ [6, 9),\ v_3 = 4$
- $i = 4,\ [2, 11),\ v_4 = 7$
- $i = 5,\ [9, 12),\ v_5 = 2$
- $i = 6,\ [10, 13),\ v_6 = 1$

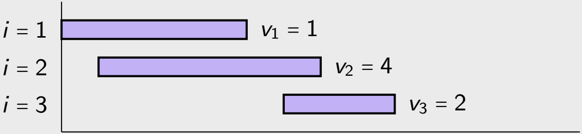


Con ganancias diferentes, el problema
no es equivalente a maximizar el número de charlas

Programación de charlas 2.0

Ejemplo

Podemos pensar en una instancia del problema de forma que la estrategia codiciosa mencionada **no funciona**



En este caso,

estrategia	charlas	ganancia
codiciosa	$\{1, 3\}$	3
óptima	$\{2\}$	4

Nuestra estrategia codiciosa no es óptima
en el caso general del problema con ganancias

Objetivos de la clase

- Comprender el paradigma de programación dinámica
- Identificar su estructura recursiva
- Aplicar la estrategia para resolver problemas concretos

Sumario

Obertura

Programación dinámica

Dos aplicaciones

Epílogo

Una nueva estrategia algorítmica

Utilizaremos una nueva estretegia: **programación dinámica**

- Generalmente usada en problemas de optimización
- Se basa en la existencia de **subproblemas** que permiten resolver el problema original
- Además, los subproblemas se **solapan**, i.e. comparten **sub-subproblemas**

La diferencia con **dividir para conquistar** es que en esta última los subproblemas son disjuntos

La clave de programación dinámica es
recordar las soluciones a los subproblemas

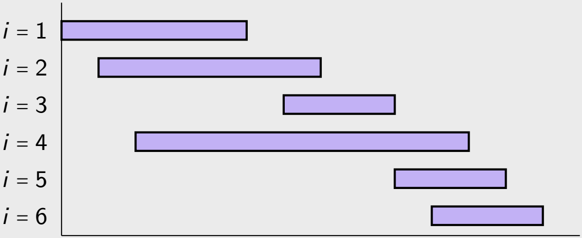
Programación de charlas 2.0

Ejemplo

Dadas las charlas $\{1, 2, \ldots, 6\}$ ordenadas por f_i, añadimos

$$b(i) := \begin{cases} j, & j \text{ es la charla que termina más tarde antes de } s_i \\ 0, & \text{no hay tal charla} \end{cases}$$

- $i = 1,\ [0, 5),\ v_1 = 2,\ b(1) = 0$
- $i = 2,\ [1, 7),\ v_2 = 4,\ b(2) = 0$
- $i = 3,\ [6, 9),\ v_3 = 4,\ b(3) = 1$
- $i = 4,\ [2, 11),\ v_4 = 7,\ b(4) = 0$
- $i = 5,\ [9, 12),\ v_5 = 2,\ b(5) = 3$
- $i = 6,\ [10, 13),\ v_6 = 1,\ b(6) = 3$



Programación de charlas 2.0

Consideremos una instancia con charlas $\{1, \ldots, n\}$ ordenadas por f_i. Supongamos que tenemos una solución óptima Ω para este problema.

Consideremos la última charla, i.e. n. Tenemos dos opciones

- Si $n \notin \Omega$, entonces Ω es solución del **subproblema** que solo considera las charlas $\{1, \ldots, n-1\}$
- Si $n \in \Omega$, entonces no hay charla r tal que $b(n) < r < n$ que esté en Ω. Además, Ω contiene una solución óptima al **subproblema** con charlas $\{1, \ldots, b(n)\}$

Para encontrar la solución a un problema, necesitamos las soluciones a problemas más pequeños

Programación de charlas 2.0

Formalicemos las ideas anteriores

- Sea Ω_j la solución al problema con charlas $\{1, \ldots, j\}$ y sea $opt(j)$ su ganancia total. **Objetivo final:** obtener Ω_n con valor $opt(n)$
- Para cada $1 \leq j \leq n$, hay dos casos
  - Si $j \in \Omega_j$, entonces $opt(j) = v_j + opt(b(j))$
  - Si $j \notin \Omega_j$, entonces $opt(j) = opt(j-1)$
- Para saber si $j \in \Omega_j$, comparamos las dos opciones

$$\boxed{opt(j) = \max\{v_j + opt(b(j)),\ opt(j-1)\}} \qquad (\bigstar)$$

Programación de charlas 2.0

La ecuación $(\bigstar)$ permite plantear el siguiente algoritmo recursivo

input : natural $0 \leq j \leq n$

output: ganancia óptima

```
Opt(j):
    if j = 0 :
        return 0
    else:
        return max{v_j + Opt(b(j)), Opt(j − 1)}
```

$\texttt{Opt}(j)$**:**

if $j = 0$ **:**

  return 0

else:

  return $\max\{v_j + \texttt{Opt}(b(j)),\ \texttt{Opt}(j-1)\}$

Notemos que `Opt` requiere

- tener ordenadas las charlas por hora de término y conocer $b(j)$
- suponer que $\texttt{Opt}(0) = 0$

¿Cuál es el problema de este algoritmo?

Programación de charlas 2.0

El problema con el algoritmo presentado es su complejidad

- Cada llamada a `Opt`, en el peor caso da origen a dos llamados `Opt`
- Complejidad $\mathcal{O}(2^n)$

Ejemplo (llamados recursivos)

`Opt`(6)

- $\texttt{Opt}(b(6)) = \texttt{Opt}(3)$
  - $\texttt{Opt}(b(3)) = \texttt{Opt}(1)$
  - $\texttt{Opt}(3-1) = \texttt{Opt}(2)$
    - $\texttt{Opt}(2-1) = \texttt{Opt}(1)$
- $\texttt{Opt}(6-1) = \texttt{Opt}(5)$
  - $\texttt{Opt}(b(5)) = \texttt{Opt}(3)$
    - $\texttt{Opt}(b(3)) = \texttt{Opt}(1)$
    - $\texttt{Opt}(3-1) = \texttt{Opt}(2)\ldots$
  - $\texttt{Opt}(5-1) = \texttt{Opt}(4)$
    - $\texttt{Opt}(4-1) = \texttt{Opt}(3)\ldots$

Programación de charlas 2.0

A pesar de hacer una cantidad exponencial de llamados, realmente se resuelven solo $n + 1$ subproblemas

$$\texttt{Opt}(0), \texttt{Opt}(1), \ldots, \texttt{Opt}(n)$$

En este problema, un mismo llamado puede aparecer varias veces en el árbol de recursión

¿Podríamos hacerlo mejor?

Programación de charlas 2.0

Agregaremos un arreglo global M donde almacenaremos cada $\texttt{Opt}(j)$ la primera vez que lo calculamos

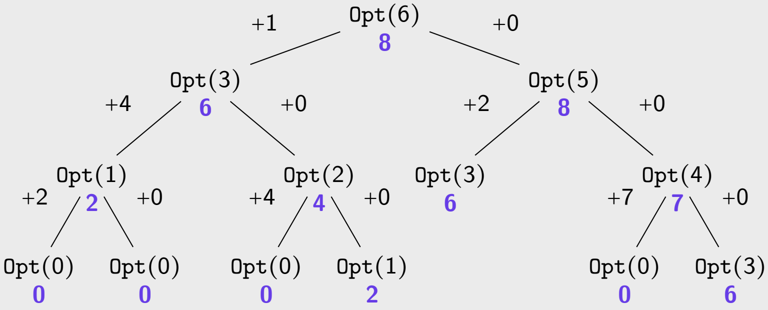
$\texttt{RecOpt}(j)$**:**

    if $j = 0$ **:**
        return 0
    else:
        if $M[j] \neq \varnothing$ **:**
            return $M[j]$
        else:
            $M[j] \leftarrow \max\{v_j + \texttt{RecOpt}(b(j)),\ \texttt{RecOpt}(j-1)\}$
            return $M[j]$

El algoritmo $\texttt{RecOpt}$ toma tiempo $\mathcal{O}(n)$

Programación de charlas 2.0

- $i = 1,\ [0,5),\ v_1 = 2,\ b(1) = 0$
- $i = 2,\ [1,7),\ v_2 = 4,\ b(2) = 0$
- $i = 3,\ [6,9),\ v_3 = 4,\ b(3) = 1$
- $i = 4,\ [2,11),\ v_4 = 7,\ b(4) = 0$
- $i = 5,\ [9,12),\ v_5 = 2,\ b(5) = 3$
- $i = 6,\ [10,13),\ v_6 = 1,\ b(6) = 3$



Programación de charlas 2.0

Ejemplo

Utilizando `RecOpt`, obtenemos la matriz de ganancias

$$M = \begin{array}{|c|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 6 & 7 & 8 & 8 \\ \hline \end{array}$$

$$\begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array}$$

Podemos **deducir la asignación** a partir de M, suponiendo que cada $v_j > 0$.

- Como $M[6] = M[6-1]$, no se incluye la charla 6
- Como $M[5] = v_5 + M[b(5)]$, se incluye la charla 5
- Como $M[3] = v_3 + M[b(3)]$, se incluye la charla 3
- Como $M[1] = v_1 + M[b(1)]$, se incluye 1

Con esto, las charlas asignadas son $\{1, 3, 5\}$

Programación de charlas 2.0

También podemos plantear una solución **iterativa** para este problema, computando los subproblemas en orden de tamaño

```
ItOpt:
    M[0] ← 0
    for j = 1, ..., n :
        M[j] ← max{v_j + M[b(j)], M[j − 1]}
```

Al terminar, `ItOpt` deja en M las ganancias óptimas

El algoritmo `ItOpt` también toma tiempo $\mathcal{O}(n)$

Programación dinámica

A partir de este ejemplo, planteamos la estrategia de **programación dinámica** para resolver un problema

1. El número de subproblemas es (idealmente) polinomial
2. La solución al problema original puede calcularse a partir de subsoluciones
3. Hay un **orden natural** de los subproblemas (*del más pequeño al más grande*) y una recurrencia *sencilla* (★)
4. Recordamos las soluciones a subproblemas

La recurrencia es la clave para plantear el algoritmo base

Sumario

Obertura

Programación dinámica

Dos aplicaciones

Epílogo

Problema de la mochila 0/1

Consideremos el problema de la mochila con n objetos **no fraccionables** (cada uno se incluye o no se incluye)

- El objeto k tiene un valor v_k y un peso w_k
- La mochila tiene capacidad de peso W tal que

$$W < \sum_k w_k$$

El **objetivo** es maximizar la suma de valores incluídos

Usaremos la variable $x_k \in \{0, 1\}$ para indicar si el objeto k se incluye o no

Resolveremos este problema con programación dinámica

Problema de la mochila 0/1

Denotaremos por $knap(p, q, \omega)$ al problema de maximizar

$$\sum_{k=p}^{q} v_k x_k$$

sujeto a

$$\begin{array}{rcl} \sum_{k=p}^{q} w_k x_k & \leq & \omega \\ x_k & \in & \{0, 1\} \end{array}$$

Nuestro problema a resolver es $knap(1, n, W)$

Problema de la mochila 0/1

Sea $\Omega = y_1, y_2, \ldots, y_n$ una elección óptima de valores binarios para las variables $x_1, x_2, \ldots, x_n$

En particular, y_1 tiene dos opciones

- Si $y_1 = 0$, entonces el objeto 1 no está en la solución. Luego, $y_2, \ldots, y_n$ debe ser solución óptima para

$$knap(2, n, W)$$

  De lo contrario, $y_2, \ldots, y_n$ no sería solución óptima de $knap(1, n, W)$

Problema de la mochila 0/1

En particular, y_1 tiene dos opciones

- Si $y_1 = 1$, entonces $y_2, \ldots, y_n$ debe ser solución óptima para

$$knap(2, n, W - w_1)$$

De lo contrario, habría otra selección $z_2, \ldots, z_n$ binaria tal que

$$\sum_{k=2}^{n} w_k z_k \leq W - w_1 \quad \text{y} \quad \sum_{k=2}^{n} v_k z_k > \sum_{k=2}^{n} v_k y_k$$

por lo que $y_1, z_2, \ldots, z_n$ sería una elección mejor para $knap(1, n, W)$.
Esto contradice que $y_1, y_2, \ldots, y_n$ es óptima

Problema de la mochila 0/1

Con este análisis de casos, planteamos nuestra estrategia recursiva de **subproblemas**

Sea $g_k(\omega)$ el valor de una solución óptima para $knap(k+1, n, \omega)$

- $g_0(W)$ es el valor óptimo de $knap(1, n, W)$
- Como hay decisión binaria para x_1,

$$g_0(W) = \max\{g_1(W),\ g_1(W - w_1) + v_1\}$$

- Podemos generalizar para un $0 \le k < n$

$$g_k(\omega) = \max\{g_{k+1}(\omega),\ g_{k+1}(\omega - w_1) + v_{k+1}\}$$

donde

$$g_n(\omega) = \begin{cases} 0, & \text{si } \omega \ge 0 \\ -\infty, & \text{si } \omega < 0 \end{cases}$$

Problema de la mochila 0/1

Ejercicio

Usando la recurrencia anterior, muestre los llamados recursivos que permiten resolver la siguiente instancia del problema de la mochila 0/1

- $n = 3,\ W = 6$
- $[w_1, w_2, w_3] = [2, 2, 3]$
- $[v_1, v_2, v_3] = [1, 2, 5]$

Problema de dar vuelto

Consideremos ahora el problema de dar S pesos de vuelto usando el menor número posible de monedas

- Suponemos que los valores de las monedas, ordenados de mayor a menor, son $\{v_1, v_2 \ldots, v_n\}$
- Tenemos una cantidad ilimitada de monedas de cada valor

Posible estrategia codiciosa:

Asignar tantas monedas *grandes* como sea posible, antes de avanzar a la siguiente moneda *grande*

Ejemplo

Si $\{v_1, v_2, v_3, v_4\} = \{10, 5, 2, 1\}$, la estrategia codiciosa **siempre** produce el menor número de monedas para un vuelto S cualquiera

Problema de dar vuelto

Ejemplo

Sin embargo, la estrategia no funciona para un conjunto de valores cualquiera. Si $\{v_1, v_2, v_3\} = \{6, 4, 1\}$ y $S = 8$, entonces la estrategia produce

$$8 = 6 + 1 + 1$$

pero el óptimo es

$$8 = 4 + 4$$

Lo atacamos con programación dinámica

Problema de dar vuelto

Dado un conjunto de valores ordenados $\{v_1, \ldots, v_n\}$, definimos $z(S, n)$ como el problema de encontrar el menor número de monedas para totalizar S

Ejercicio

Proponga una recurrencia para resolver el problema $z(S, n)$ y plantee un algoritmo a partir de ella.

Problema de dar vuelto

Ejercicio

Sea $Z(S,n)$ la solución óptima al problema $z(S,n)$. Para buscar intuición, notamos que hay dos opciones respecto a las monedas de valor v_n

- Si se incluye una moneda de valor v_n,

$$Z(S,n) = Z(S - v_n, n) + 1$$

- Si no se usan monedas de valor v_n,

$$Z(S,n) = Z(S, n-1)$$

Problema de dar vuelto

Ejercicio

Luego, generalizamos esta idea

- Para las monedas de de valor v_n,

$$Z(S,n) = \min\{Z(S - v_n, n) + 1,\ Z(S, n-1)\}$$

- Luego, generalizamos para el subconjunto de los primeros k valores de monedas

$$Z(T,k) = \min\{Z(T - v_k, n) + 1,\ Z(T, k-1)\}$$

donde $Z(T,0) = +\infty$ si $T > 0$, y $Z(0,k) = 0$

Problema de dar vuelto

Ejercicio

Con esto, podemos plantear el siguiente algoritmo iterativo

```
Change(S):
    for T = 1, ..., S :
        Z[T][0] ← +∞
    for k = 0, ..., n :
        Z[0][k] ← 0
    for k = 1, ..., n :
        for T = 1, ..., S :
            Z[T][k] ← Z[T][k − 1]
            if T − v_k ≥ 0 :
                Z[T][k] ← min{Z[T][k], Z[T − v_k, k]}
```

Sumario

Obertura

Programación dinámica

Dos aplicaciones

Epílogo

Objetivos de la clase

- ☐ Comprender el paradigma de programación dinámica
- ☐ Identificar su estructura recursiva
- ☐ Aplicar la estrategia para resolver problemas concretos