Programación Dinámica – Parte II

Materia: TTPS

Autor: JTP - Matías Fluxa

Knapsack Problem

Knapsack Problem

Enunciado

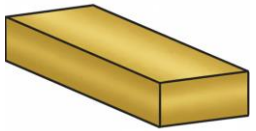
Dado un conjunto de n artículos, donde cada artículo i tiene un **peso** w_i y un **valor** v_i, y dada una **capacidad máxima** W para una mochila.

Determinar qué subconjunto de artículos se debe incluir en la mochila para que:

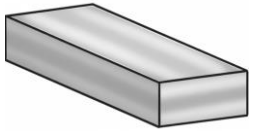
- La **suma total de los pesos** de los artículos seleccionados **no exceda** la capacidad máxima W.
- La **suma total de los valores** de los artículos seleccionados sea la **máxima posible**.

Knapsack Problem

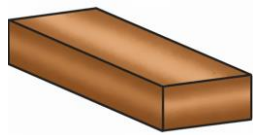
$P_1 = 2\ kg\ - V_1 = 600$



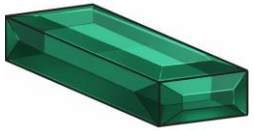
$P_2\ 4\ kg\ - V_2 = 1000$



$P_3 = 5\ kg\ - V_3 = 1300$



$P_4 = 3\ kg\ - V_4 = 500$



$P_5 = 1\ kg\ - V_5 = 250$

Capacidad $W = 10\ kg$

Knapsack Problem

Ordenamos los ítems por “Densidad”

Item	Peso [kg]	Valor [$]	Densidad [$/kg]
1.Oro	2	600	300
3. Bronce	5	1300	260
2. Plata	4	1000	250
5. Vidrio	1	250	250
4. Esmeralda	3	500	167

Knapsack Problem

Ordenamos los ítems por "Densidad"

Item	Peso [kg]	Valor [$]	Densidad [$/kg]
1.Oro	2	600	300
3. Bronce	5	1300	260
2. Plata	4	1000	250
5. Vidrio	1	250	250
4. Esmeralda	3	500	167

Si realizamos un método greedy eligiendo del objeto más al menos denso, nos queda la siguiente configuración:
Oro + Bronce + Vidrio – Peso Total: 8kg – Valor: 2.150

¡La cual no es la más óptima!

Knapsack Problem

Definimos nuestro estado.
$\boldsymbol{dp[i][x]}$: Máximo valor posible utilizando los primeros i items y utilizando exactamente x kilos en la mochila.

Supongamos que ya conocemos las soluciones óptimas para los primeros $i - 1$ items.
Si procesamos el i-ésimo item obtenemos:

$$dp[i][x] = \max(dp[i-1][x], dp[i-1][x - P_i] + V_i)$$

Knapsack Problem

$$dp[i][x] = \max(dp[i-1][x], dp[i-1][x-P_i] + V_i)$$

Es claro que tenemos dos posibilidades para nuestro objeto i, lo usamos o no lo usamos, y debemos elegir cuál es mejor.

Transición 1: $dp[i][x] = dp[i-1][x]$
En este caso no usamos el i-ésimo objeto, así que nos quedamos con la mejor configuración para $i-1$ objetos.

Transición 2: $dp[i][x] = dp[i-1][x-P_i] + V_i$
Consideramos utilizar el i-ésimo objeto, antes debíamos tener una configuración de $i-1$ objetos que pesen $x-P_i$ y sumamos su valor V_i por haberlo guardado.

Resolución con DP

- Cabe mencionar que no importa el orden en que se procesa los objetos. Cualquier permutación permitirá encontrar la solución correcta.
- La complejidad del algoritmo es $O(N \cdot W)$ donde N es la cantidad de ítems y W la capacidad máxima de la mochila. También existen soluciones con complejidad $O\left(N \cdot \sum_{i=1}^{N} V_i\right)$.
- Puede implementarse utilizando solo $O(W)$ de memoria.

Resolución con DP

Paso a paso la resolución de los estados. Representamos con -1 los estados no posibles.

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Resolución con DP

Paso a paso la resolución de los estados. Representamos con -1 los estados no posibles.

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
i=1	0	-1	600	-1	-1	-1	-1	-1	-1	-1	-1

Resolución con DP

Paso a paso la resolución de los estados. Representamos con -1 los estados no posibles.

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
i=1	0	-1	600	-1	-1	-1	-1	-1	-1	-1	-1
i=2	0	-1	600	-1	1000	-1	1600	-1	-1	-1	-1

Resolución con DP

Paso a paso la resolución de los estados. Representamos con -1 los estados no posibles.

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
i=1	0	-1	600	-1	-1	-1	-1	-1	-1	-1	-1
i=2	0	-1	600	-1	1000	-1	1600	-1	-1	-1	-1
i=3	0	-1	600	-1	1000	1300	1600	1900	-1	2300	-1

Resolución con DP

Paso a paso la resolución de los estados. Representamos con -1 los estados no posibles.

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
i=1	0	-1	600	-1	-1	-1	-1	-1	-1	-1	-1
i=2	0	-1	600	-1	1000	-1	1600	-1	-1	-1	-1
i=3	0	-1	600	-1	1000	1300	1600	1900	-1	2300	-1
i=4	0	-1	600	500	1000	1300	1600	1900	1800	2300	2400

Resolución con DP

Paso a paso la resolución de los estados. Representamos con -1 los estados no posibles.

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
i=1	0	-1	600	-1	-1	-1	-1	-1	-1	-1	-1
i=2	0	-1	600	-1	1000	-1	1600	-1	-1	-1	-1
i=3	0	-1	600	-1	1000	1300	1600	1900	-1	2300	-1
i=4	0	-1	600	500	1000	1300	1600	1900	1800	2300	2400
i=5	0	250	600	850	1000	1300	1600	1900	2150	2300	2550

¿Dónde practicar más problemas de DP?

La programación dinámica es una técnica que es muy evaluada en muchas entrevistas laborales. Dominar bien la técnica puede ser muchas veces determinante para obtener un puesto laboral.

La variedad de problemas de programación dinámica es infinita, por lo que es una herramienta que hay que comprender bien y no memorizar cada uno de los ejemplos.

La mejor forma de practicar es haciendo muchos ejercicios, comprendiendo bien su lógica y reteniéndola para futuros problemas.

¿Dónde practicar más problemas de DP?

- Leetcode: https://leetcode.com/problem-list/dynamic-programming/
  Leetcode es una gran página para lo que buscan practicar para entrevistas laborales. Abarca muchos tutoriales de varias técnica evaluadas en entrevista.

- Guía de DP de atcoder: https://atcoder.jp/contests/dp/tasks
  Con problemas desde la A hasta la Z, comprende un gran rango de problemas de programación dinámica y sus distintas ramas de aplicación.

- Guía de DP de CSES: https://cses.fi/problemset/
  Esta página finlandesa también abarca problemas típicos de varias ramas. Muchos de los problemas de la práctica son tomados de esta página.

Curiosidades

En la final mundial de programación ICPC han tomado un problema que se resuelve con programación dinámica.

Era uno de los problemas más sencillo de la prueba pero aún así, un problema lindo y desafiante. Solamente un equipo de 139 no pudo resolverlo.

El primer equipo en resolverlo tardó 14 minutos!!

https://worldfinals.icpc.global/problems/2025/finals/problems/D-buggyrover.pdf

No era el único problema de la prueba que utilizaba DP ...