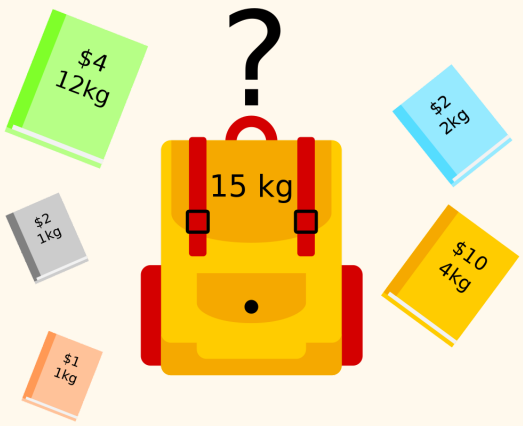
ANÁLISIS Y DISEÑO DE ALGORITMOS CC371-A
Prof. Jaime Osorio Ubaldo

PROGRAMACIÓN DINÁMICA
EL PROBLEMA DE LA MOCHILA

—

Por: Guillermo Ronie Salcedo Alvarez



DISEÑO APLICANDO PROGRAMACIÓN DINÁMICA

El problema de la mochila (*knapsack problem*) es un famoso problema que consiste en encontrar la mejor manera de llevar una mochila con un cierto conjunto de objetos, de tal modo que se maximice el valor total de los objetos llevados, pero sin exceder la capacidad de la mochila.

El primer enfoque de resolución que se le podría dar a este problema sería a través de un método recursivo conocido como *backtracking* en el cual buscamos todas las posibles soluciones con la capacidad de retroceso o *backtrack*, cuando una solución parcial no puede completarse hasta una solución válida. Entonces, para el problema de la mochila, se buscarían todas los subconjuntos y de ellos se elige la mejor opción; no obstante, de resolverse de este modo, estaríamos ante una complejidad $O(2^n)$.

Ante dicho inconveniente, buscamos una manera más óptima de resolver nuestro problema y será por medio de **programación dinámica**. En donde construiremos una tabla para almacenar aquellos valores que se vuelven a calcular.

Empezaremos con definir que nuestra mochila tiene una capacidad máxima W, y tenemos n objetos enumerados del 1 al n, cada uno con un peso w_i y valores v_i, buscaremos:

- Maximizar $\sum_{i=1}^{n} v_i x_i$
- Y que se verifique $\sum_{i=1}^{n} w_i x_i \leq W$ tal que $x_i \epsilon \{0, 1\}$

Donde x_i nos indicará si se toma el elemento i o no.

Luego, para el pseudocódigo tendríamos dos funciones, la primera, que servirá para inicializar nuestra tabla y llamar a la siguiente función, y esta, que resolverá el problema, más realizará el proceso de *memorización*.

```
probMochilaRec(int W, vector objetos, int i, table dp)

	Si i < 0 o W == 0
		Devuelve 0
	Si dp[i][W] != -1
		Devuelve dp[i][w]

	Si objetos.peso[i] > W
		dp[i][W] = probMochilaRec(W, objetos, i-1, dp)
		Devuelve dp[i][W]
	Sino
		dp[i][W] = max(objetos.valor[i] +
				probMochilaRec(W - objetos.peso[i], objetos, i-1, dp),
				probMochilaRec(W, objetos, i-1, dp))
		Devuelve dp[i][W]

fin-probMochilaRec



probMochila(int W, vector objetos, int n)

	Crear tabla de n x W+1
	Inicializar valores con -1
	Devuelve probMochilaRec(W, objetos, n-1, dp)

fin-probMochila
```

Entonces tendríamos que la resolución al problema por medio de programación dinámica, será a través de la tabla dp. Empezaremos por llamar a la función probMochila en donde se creará e inicializará nuestra tabla, e inmediatamente llamará a probMochilaRec, en donde se encuentra la resolución al problema de manera recursiva.

La función probMochilaRec es recursiva, por lo que empezaremos definiendo nuestros casos base, donde se verifica que el índice del objeto a evaluar no sea menor que cero o que la capacidad de la mochila no sea cero. Además, si tenemos que existe un elemento en la tabla para el valor (i, W), lo devolveremos.

Por último, si el peso del objeto que estamos evaluando pasa de la capacidad de la mochila, entonces guardaremos en la tabla el resultado de no tomar dicho objeto y lo devolveremos. De lo contrario, en la tabla guardaremos el mayor valor entre tomar dicho valor o no, y se devuelve dicho valor.

EJEMPLO DE EJECUCIÓN

Procederemos a resolver nuestro problema paso a paso con el pseudocódigo dado. Sea nuestra capacidad W = 8, y tenemos n = 4 objetos, wi = {2, 3, 4, 5} y vi = {1, 2, 5, 6}. Guardaremos estos últimos valores en un vector con la estructura de dicho objeto. Y procedemos:

```
probMochila(W, obj, n)

       table dp inicializada con -1.

       probMochilaRec(8, obj, 3, dp)

              obj.peso[3] = 5 < W = 8
                     dp[3][8] = max(obj.valor[3] = 6 + probMochilaRec(3, obj, 2, dp),
                                    probMochilaRec(8, obj, 2, dp))
```

- `probMochilaRec(3, obj, 2, dp)`

```
              obj.peso[2] = 4 > 3
                     dp[2][3] = probMochilaRec(3, obj, 1, dp)
```

- probMochilaRec(3, obj, 1, dp)
  obj.peso[1] = 3 == 3
  dp[1][3] = max(obj.valor[1] = 2 + probMochilaRec(0, obj, 0, dp),
  probMochilaRec(3, obj, 0, dp))

- probMochilaRec(0, obj, 0, dp)
  W == 0
  Devuelve 0

- probMochilaRec(3, obj, 0, dp)
  obj.peso[0] = 2 < 3
  dp[0][3] = max(obj.valor[0] = 1 + probMochilaRec(1, obj, -1, dp),
  probMochilaRec(3, obj, -1, dp))

- probMochilaRec(1, obj, -1, dp)
  i = -1 < 0
  Devuelve 0

- probMochilaRec(3, obj, -1, dp)
  i = -1 < 0
  Devuelve 0

- **probMochilaRec(3, obj, 0, dp)**
  obj.peso[0] = 2 < 3
  dp[0][3] = max(1, 0) = 1

- **probMochilaRec(3, obj, 1, dp)**
  obj.peso[1] = 3 == 3
  dp[1][3] = max(2, 1) = 2

- **probMochilaRec(3, obj, 2, dp)**
  obj.peso[2] = 4 > 3
  dp[2][3] = 2

- probMochilaRec(8, obj, 2, dp)
  obj.peso[2] = 4 < 8
  dp[2][8] = max(obj.valor[2] = 5 + probMochilaRec(4, obj, 1, dp),
  probMochilaRec(8, obj, 1, dp))
- probMochilaRec(4, obj, 1, dp)
  obj.peso[1] = 3 < 4
  dp[1][4] = max(obj.valor[1] = 2 + probMochilaRec(1, obj, 0, dp),
  probMochilaRec(4, obj, 0, dp))
- probMochilaRec(1, obj, 0, dp)
  obj.peso[0] = 2 > 1
  dp[0][1] = probMochilaRec(1, obj, -1, dp)

- probMochilaRec(1, obj, -1, dp)
  i = -1 < 0
  Devuelve 0

- **probMochilaRec(1, obj, 0, dp)**
  obj.peso[0] = 2 > 1
  dp[0][1] = 0

- probMochilaRec(4, obj, 0, dp)
  obj.peso[0] = 2 < 4
  dp[0][4] = max(obj.valor[0] = 1 + probMochilaRec(2, obj, -1, dp),
  probMochilaRec(4, obj, -1, dp))

- probMochilaRec(2, obj, -1, dp)
  i = -1 < 0
  Devuelve 0

- probMochila(4, obj, -1, dp)
  i = -1 < 0
  Devuelve 0

- **probMochilaRec(4, obj, 0, dp)**
  obj.peso[0] = 2 < 4
  dp[0][4] = max(1, 0) = 1

- **probMochilaRec(4, obj, 1, dp)**
  obj.peso[1] = 3 < 4
  dp[1][4] = max(2, 1) = 2

- probMochila(8, obj, 1, dp)
  obj.peso[1] = 4 < 8
  dp[1][8] = max(obj.valor[1] = 2 + probMochilaRec(4, obj, 0, dp),
  probMochilaRec(8, obj, 0, dp))

- **probMochila(4, obj, 0, dp)**
  dp[0][4] != -1
  dp[0][4] = 1

- probMochilaRec(8, obj, 0, dp)
  obj.peso[0] = 2 < 8
  dp[0][8] = max(obj.valor[0] = 1 + probMochilaRec(6, obj, -1, dp),
  probMochilaRec(8, obj, -1, dp))

- probMochilaRec(6, obj, -1, dp)
  i = -1 < 0
  Devuelve 0

- probMochila(8, obj, -1, dp)
  i = -1 < 0
  Devuelve 0

- **probMochilaRec(8, obj, 0, dp)**
  obj.peso[0] = 2 < 8
  dp[0][8] = max(1, 0) = 1

- **probMochila(8, obj, 1, dp)**
  obj.peso[1] = 4 < 8
  dp[1][8] = max(3, 1) = 3

- **probMochilaRec(8, obj, 2, dp)**
  obj.peso[2] = 4 < 8
  dp[2][8] = max(7, 3) = 7

- **probMochilaRec(8, obj, 3, dp)**
  obj.peso[3] = 5 < W = 8
  dp[3][8] = max(8, 7) = 8

Se obtiene el valor máximo obtenido para llevar en la mochila igual a 8.

COMPLEJIDAD DEL ALGORITMO

A continuación se presenta el desarrollo de cómo obtener la complejidad Big O del algoritmo de *programación dinámica* del problema de la mochila. Para ello, partiremos por obtener el costo temporal de la misma.

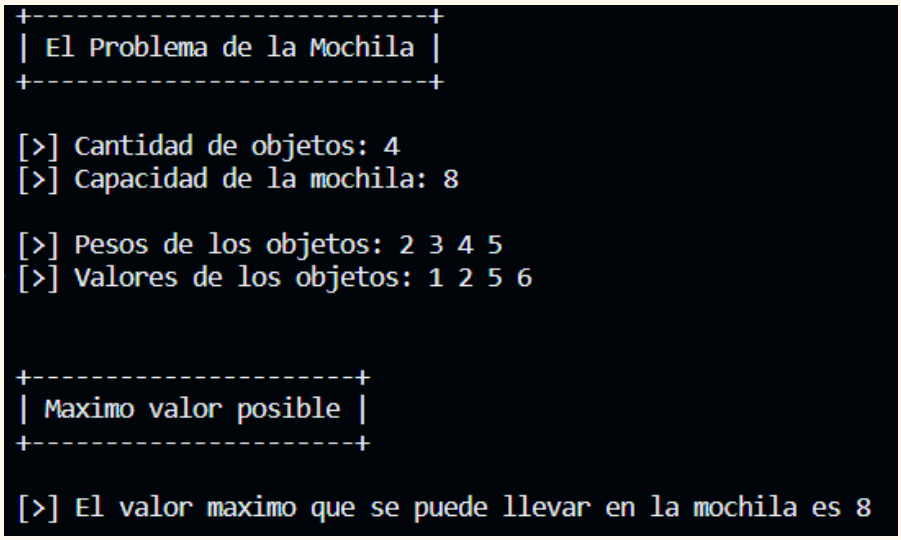
Notemos que el programa iniciará con la inicialización de la tabla, y dado que se iterará a través de cada elemento en una tabla de n x W+1. Entonces la complejidad de esta será O(nW). Inmediatamente se llama a la función recursiva, donde los costos temporales de comparaciones serán O(1). Luego, veremos que el problema iterará, en el peor de los casos, sobre cada i y cada W, equivalente a O(nW). Para terminar en el *else* de nuestro pseudocódigo vemos que se llama dos veces a la función recursiva, esto quiere decir que tendríamos 2 x O(nW).

Una vez sumemos todos estos costos temporales, nos daremos cuenta que la complejidad predominante será O(nW), con lo que queda que nuestro algoritmo es de dicha complejidad.

PROGRAMA DEL ALGORITMO

La implementación del algoritmo en programación dinámica para el problema de la mochila se realizó en el lenguaje de programación C++. En él, se nos solicitará la cantidad de objetos, la capacidad de la mochila, y los pesos y valores respectivos para cada uno de los objetos. Inmediatamente se ejecuta la función que resolverá el problema y se imprimirá la respuesta en consola, el máximo valor que se puede conseguir.

Fig. 1: Resultados en consola para un ejemplo dado



UNA NUEVA ALTERNATIVA

Otro enfoque que se le puede dar al problema es a través de la resolución de todo el problema con ayuda de una tabla, para este caso, será de tamaño (n+1) x (W+1). Esta implementación consiste en que para nuestra tabla dp, el elemento `dp[i, w]` representará el máximo valor que se puede obtener hasta un peso *w*, tomando *i* objetos.

Para ello, se definirá `dp[i][w]` como:

- dp[0][w] = 0
- Si `wi > w: dp[i][w] = dp[i-1][w]`
- Si `wi <= w: dp[i][w] = max(m[i-1][w], m[i -1][w-wi] + vi)`

Finalmente, el valor máximo que puede llevar la mochila será `dp[i][W]`. El algoritmo consistirá en ir iterando para cada elemento de la tabla y realizar el cálculo respecto. Tendríamos dos iteraciones, uno de `i=1` hasta n y otro de `j=1` hasta W. Por lo que estaríamos frente a una complejidad O(nW), al igual que nuestra implementación anterior.

A continuación se presenta el pseudocódigo con dicho enfoque:

```
probMochila(int W, vector objetos)

	Inicializar tabla dp (n+1) x (W+1) con 0

	Para i = 1 hasta n:
		Para j = 1 hasta n:
			Si objetos[i-1].peso <= j
				dp[i][j] = max(dp[i-1][j],
					objetos[i].valor + dp[i-1][j-objetos[i].peso])
	Devuelve dp[n][W]

fin-probMochila
```

Referencias

- Wikipedia contributors. (2023). Knapsack problem. *Wikipedia*. https://en.wikipedia.org/wiki/Knapsack_problem.