"Análisis de Algoritmos"
José Helo Guzmán

Contenido

▬▬▬▬ 1. Introducción.

Vamos a resolver el problema del “knapsack” utilizando una técnica conocida como “dynamic programming”, en español programación dinámica.

La programación dinámica es una técnica que busca resolver un problema al dividirlo en etapas de decisión. Cada decisión dependerá de la decisión anterior.

Esta técnica tiene una parte de arte y ciencia, pues para encontrar la fórmula de recurrencia matemática se debe conocer bien la descripción del problema.

▬▬▬▬ 2. The Knapsack Problem.

Volvamos a describir el problema del Knapsack.

Suponga que tenemos una lista con los siguientes artículos. En la lista se menciona el nombre, la ganancia de vender el artículo y su peso. Se utiliza una lista con la siguiente descripción:

[nombre, [ganancia_$, peso_kilos]]

Vamos a suponer adicionalmente, que los pesos de los artículos son valores enteros.

Tenemos entonces:

```
objetos:
[
[ guitarra, [1500, 1]],
[ stereo,   [3000, 4]],
[ laptop,   [2000, 3]],
[ iphone,   [2000, 1]]
];
```

Supongamos que tenemos una mochila que puede llevar 4 kilos de peso.

El objetivo es escoger de esta lista de 4 artículos, aquellos que van a producir el máximo valor al sumar sus ganancias.

▬▬▬ 3. Solución con Programación Dinámica.

Primero creamos una tabla con los posibles pesos y los artículos. Por lo tanto el tamaño de la tabla es (num_artículos * peso). En este caso se han indicado los índices de la fila, y los índices de la columna corresponden al peso.

i		1k	2k	3k	4k
1	guitarra	--	--	--	--
2	stereo	--	--	--	--
3	laptop	--	--	--	--
4	iphone	--	--	--	--

Vamos a ir llenado esta matriz por filas. Una vez que se hayan llenado todas la fila, vamos a tener la solución del problema.

>>> Llenando la fila 1.

La guitarra cabe en una mochila de 1k, de 2k, de 3k y de 4k.
Se anota la ganancia en la fila.
[guitarra, [1500, 1]]

i		1k	2k	3k	4k
1	guitarra	1500 g	1500 g	1500 g	1500 g
2	stereo	--	--	--	--
3	laptop	--	--	--	--
4	iphone	--	--	--	--

>>> Llenando la fila 2.

[guitarra, [1500, 1]]
[stereo, [3000, 4]]

i		1k	2k	3k	4k
1	guitarra	1500 g	1500 g	1500 g	1500 g
2	stereo	1500 g	1500 g	1500 g	3000 s
3	laptop	--	--	--	--
4	iphone	--	--	--	--

>>> Llenando la fila 3.

Hagamos lo mismo con la fila del laptop.
[guitarra, [1500, 1]]
[stereo, [3000, 4]]
[laptop, [2000, 3]]

i		1k	2k	3k	4k
1	guitarra	1500 g	1500 g	1500 g	1500 g
2	stereo	1500 g	1500 g	1500 g	3000 s

i		1k	2k	3k	4k
3	laptop	1500 g	1500 g	2000 l	3500 ◀◀ lg
4	iphone	--	--	--	--

Observemos la decisión tomada en el campo marcado.

Se puede poner un stereo de 4k, y ganar $3000.

o

Se puede poner un laptop de 3k, ganar $2000, queda 1k
y ahí poner una guitarra de 1k, ganar $1500, en total $2000 + $1500 = $3500

Por esta razón se calcularon los valores de mochilas más pequeñas.

Matemáticamente la fórmula que se utiliza es:

$$celda[i][j] = \max \begin{cases} valorEn\ celda[i-1][j] \\ valorDelObjetoActual \\ + valorEn\ celda[i-1][j-pesoObjetoActual] \end{cases}$$

Puede revisar que hemos usado esta fórmula en todos los casos anteriores.

>>> Llenando la fila 4.

```
[ guitarra, [1500, 1]]
[ stereo,   [3000, 4]]
[ laptop,   [2000, 3]]
[ iphone,   [2000, 1]]
```

i		1k	2k	3k	4k
1	guitarra	1500 g	1500 g	1500 g	1500 g
2	stereo	1500 g	1500 g	1500 g	3000 s
3	laptop	1500 g	1500 g	2000 l	3500 lg
4	iphone	2000 i	3500 ig	3500 ig	4000 il

●● Algunas observaciones.

• Conforme aumenta el tamaño de las filas, aumenta el valor de la ganancia.

• No es importante el orden de los objetos en las filas, siempre se llega a la misma respuesta.

• Si tuviera un collar que pesa 0.50k y una ganancia $2500. Se podría resolver el problema, pero tendríamos que usar columnas, o pesos con incrementos de 0.50. Esto produciría una matriz de un gran tamaño.

▬▬▬ 4. Solución Recursiva y Memorización.

El knapsack se puede resolver de forma recursiva.

```
/*********************************
*  Funciones Auxiliarea
*
*/

nombre(obj):=first(obj);

ganancia(obj):=first(second(obj));

peso(obj):= second(second(obj));



/*********************************
*  Función recursiva
*
*/

ks(c,obj):=block
(
   ksAux( length(obj),
          c,
          reverse(obj)
         )
);



ksAux(n,c,obj):=block
(  [res,temp1,temp2],

   if (n<=0) or (c<=0) then
   (
      res:0
   )
   elseif peso(obj[n]) > c then
   (
      res: ksAux(n-1,c,obj)
```

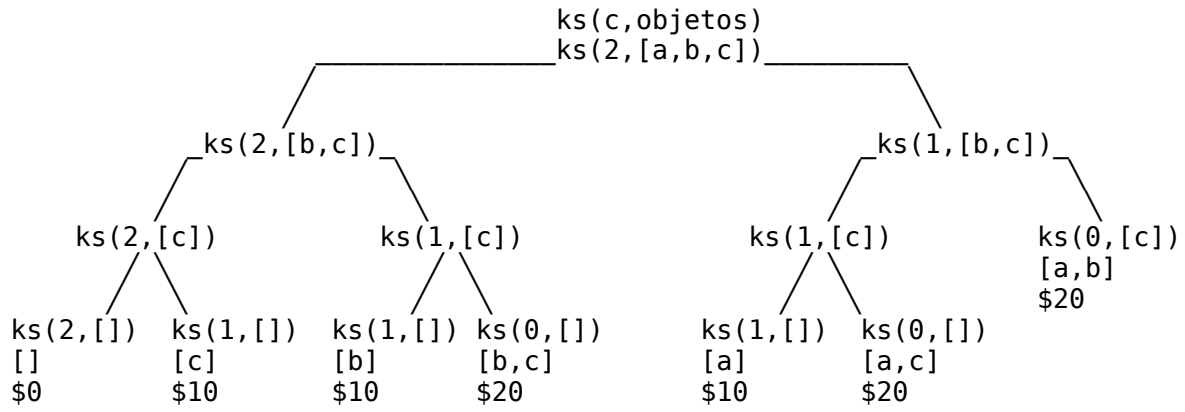
```
   )
   else
   (
      temp1: ksAux(n-1,c,obj),
      temp2: ganancia(obj[n]) + ksAux(n-1,c-peso(obj[n]),obj),
      res: max(temp1,temp2)
   ),
   return(res)
);
```

Desafortunadamente este mecanismo de solución tiene una complejidad de $O(2^n)$. Veamos un ejemplo sencillo del porque ocurre esto.

●● Un Ejemplo de Arbol de Soluciones.

Veamos un ejemplo recursivo de un knapsack,
cuya mochila tiene 2k,
Y tiene tres objetos

```
objetos:
[
  [a,  [10, 1]],
  [b,  [10, 1]],
  [c,  [10, 1]]
]
```



Para resolver este problema se tiene que hacer un proceso de “memoization”, el cual ocupa la misma cantidad de memoria que el proceso de programación dinámica.

▬▬▬ 5. Optimizando un Itinerario de Viaje.

Vamos de vacaciones para Londres. Tenemos 4 días para pasear y conocer lugares. Hay muchas cosas que usted desea hacer, pero no se pueden visitar todos los lugares ni hacer todas las cosas que uno desea en tan poco tiempo.

Hemos hecho una lista de los lugares que nos gustaría visitar, del tiempo que se necesita para ver cada lugar y del nivel de satisfacción que nos produce.

Atracción	Rating	Tiempo
westminsterAbbey	70	1 día
globeTheater	60	1 día
nationalGallery	90	2 días
britishMuseum	90	4 días
stPaulsCathedral	80	1 día

Con base en esta información, se desea escoger aquellas atracciones que tengan el mayor rating posible dentro del número de días disponibles.

●● Modelaje de los Datos.

Vamos a representar los lugares como una lista:

```
objetos:
[
[ westminsterAbbey, [70, 1]],
[ globeTheater,     [60, 1]],
[ nationalGallery,  [90, 2]],
[ britishMuseum,    [90, 4]],
[ stPaulsCathedral, [80, 1]]
];
```

Y resolveremos este problema mediante programación dinámica. Es fácil descubrir que es un problema de knapsack. En lugar de ganancias tenemos ratings, y en lugar del peso de la mochila tenemos los días que se necesitan para visitar cada lugar.

●● Tabla de Soluciones.

```
objetos:
[
[ westminsterAbbey, [70, 1]],
[ globeTheater,     [60, 1]],
[ nationalGallery,  [90, 2]],
[ britishMuseum,    [90, 4]],
[ stPaulsCathedral, [80, 1]]
```

];

Aquí pueden ver la tabla de soluciones que se produce:

i	1d	2d	3d	4d
westminsterAbbey	70 w	70 w	70 w	70 w
globeTheater	70 w	130 wg	130 wg	130 wg
nationalGallery	70 w	130 wg	160 wn	220 wgn
britishMuseum	70 w	130 wg	160 wn	220 wgn
stPaulsCathedral	80 s	150 ws	210 wgs	240 wns

Por lo tanto la respuesta final está dada por:

westminsterAbbey 1d, 70

natinalGallery 2d, 90

stPaulsCathedral 1d, 80

totalDías = 1d + 2d + 1d = 4d

totalRating = 70 + 90 + 80 = 240

▬▬▬ 6. Análisis de Complejidad.

El algoritmo original recursivo, sin memorización tiene un tiempo de ejecución de $O(2^n)$.

El utilizar el algoritmo de programación dinámica, el tiempo que se requiere para solucionar el problema, es equivalente al tiempo requerido para llenar la matriz.

Si tenemos “n” objetos, y el peso de la mochila es un valor “w”, el tamaño de la matriz será de: n*w, por lo tanto el tiempo de ejecución está dado por:

O(n*w)

▬▬▬ 7. Resumen.

• La programación dinámica es útil cuando se está tratando de optimizar un problema sujeto a restricciones.

• Se puede utilizar cuando el problema se puede descomponer en problemas más pequeños.

• Todo problema de programación lineal necesita un matriz de cálculos, ya sea implícita o explícita.

• Los valores en las celdas son usualmente lo que se está tratando de maximizar o minimizar.

• Cada celda representa un subproblema.

• Para diferentes problemas de programación lineal, se deben utilizar diferentes fórmulas para calcular la matriz.

▬▬▬ 8. Ejercicios.

●● Ejercicio 1.

Se tiene este nuevo knapsack con una mochila de 4kilos.

```
objetos:
[
[ guitarra, [1500, 1]],
[ stereo,   [3000, 4]],
[ laptop,   [2000, 3]],
[ iphone,   [2000, 1]],
[ mxp,      [1000, 1]]
];
```

Cuál es la solución de este problema?

●● Ejercicio 2.

Para esta lista de objetos, puede crear el árbol que ejecuta la solución recursiva.

```
[objetos:
[ [c,  [10, 1]],
  [b,  [10, 1]],
  [a,  [10, 1]]
]
```

●● Ejercicio 3.

Hay alguna manera que la solución recursiva indique los objetos que van en la mochila?
Puede usar una instrucción print, o una lista que guarde los objetos.

●● Ejercicio 4.

Supongamos que vamos de campamento. Tenemos una mochila en la que caben 6 kilos. Y se desean llevar algunos de los siguientes objetos. Para cada objeto se muestra su valor y su peso.

- agua, $10, 3k
- libro, $ 3, 1k
- comida, $ 9, 2k
- jacket, $ 5, 2k
- cámara, $ 6, 1k

Cuál es el conjunto óptimo de objetos que se deben llevar en la mochila para maximizar su valor?