# Problema del corte de vástagos

*Pablo Pastor Martín, Isaac Aimán Salas, Javier Ramos Fernández*

## Introducción

El problema del ***corte de vástagos*** consiste en, dado un vástago de longitud *n* y una colección de precios que contiene los precios de todas las piezas cuyo tamaño es menor que *n*, se quiere determinar los cortes del vástago de modo que el beneficio obtenido por la suma de los precios de cada una de las piezas resultantes de los cortes sea máximo. Un ejemplo de este problema sería el siguiente:

| longitud $l_i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| precio $p_i$ | 2 | 6 | 9 | 10 | 13 | 15 | 18 | 20 |

En este cuadro tenemos, para diferentes longitudes de corte ($l_1$ ,..., $l_m$), sus correspondientes precios ($p_1$ ,..., $p_m$). La tabla está ordenada por la longitud de corte, por lo que $l_1$ es la longitud de corte más pequeña que podemos vender. Por ejemplo, supongamos que la longitud del vástago es 3. Por lo tanto, las diferentes posibilidades de corte son las siguientes:


Podemos observar que, en el caso de que *n* es 3, el beneficio máximo se obtiene sin hacer ningún corte. Si los valores de las longitudes de corte hubieran sido diferentes, el beneficio máximo no tendría por qué obtenerse sin realizar ningún corte.

***Propiedad de la subestructura óptima***.

Si tenemos un vástago de longitud *n* y realizamos un corte, podemos computar el precio que obtenemos de ese corte más el mejor precio de la longitud remanente del vástago. El objetivo es averiguar qué corte maximiza la suma de la primera pieza más el precio óptimo del resto del vástago. Este mejor precio de la longitud remanente del vástago es una versión más pequeña del mismo problema, de modo que una solución óptima del problema general utiliza soluciones óptimas de versiones más pequeñas del mismo problema.

# Algoritmos

## Fuerza bruta(recursivo)

El algoritmo de fuerza bruta enumera sistemáticamente todos los posibles casos para la solución del problema y elige, de entre todos esos casos, la mejor solución. Es decir, este algoritmo analiza todas la posibilidades de corte sobre un vástago y elige aquella con la que se consiga mayor beneficio.

Como podemos observar, hay $2^{n-1}$ maneras diferentes de cortar un vástago (para $n = 3$ se pueden realizar 4 cortes, por ejemplo). Esto se puede ver suponiendo que en cada incremento de longitud tenemos la decisión binaria de si hacer o no un corte (obviamente el último incremento no se incluye ya que no produce nuevas piezas). De este modo el número de permutaciones de longitudes es igual al número de patrones binarios de *n-1* cortes, de tal forma que hay $2^{n-1}$ posibilidades. Por lo tanto, para encontrar el valor óptimo, simplemente sumamos los precios de todas las piezas de cada permutación y seleccionamos el valor más alto.

Antes de centrarnos en el algoritmo recursivo, vamos a formalizar el problema. Suponiendo que tenemos una pieza de longitud *i* con un precio $p_i$, si la solución óptima realiza un corte en el vástago en *k* piezas de longitudes $i_1, i_2, ..., i_k$ de tal modo que $n = i_1, i_2, ..., i_k$, entonces el beneficio de un vástago de longitud *n* es $r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}$ . Por lo tanto, el valor óptimo se puede encontrar en términos de vástagos más pequeños observando que si realizamos un corte óptimo de longitud i (y de este modo dando lugar a una pieza de longitud n-i), entonces ambas piezas *deben ser óptimas* (y estas piezas más pequeñas serán cortadas). De otro modo podríamos realizar un corte diferente que produjera un beneficio mayor, contradiciendo la suposición de que el primer corte fue óptimo.

A partir de esto, podemos escribir el beneficio óptimo de la siguiente manera:

$$r_n = max(p_3, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

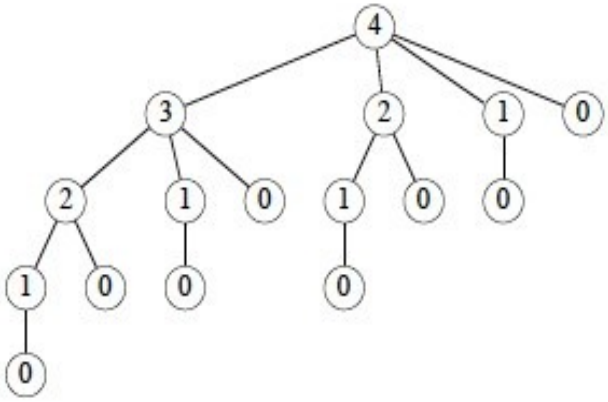
El primer término ($p_n$) corresponde a no realizar ningún corte del vástago. Los otros *n-1* términos corresponden a los beneficios máximos obtenidos tras hacer un corte inicial del vástago en dos piezas de longitudes *i* y *n-1* para *i = 1, 2, ..., n-1* y continuar haciendo cortes óptimos en esas piezas.Si suponemos que no vamos a volver a cortar la primera pieza (ya que debe haber al menos una pieza en la solución óptima) y sólo (posiblemente) cortar la segunda pieza, podemos reescribir el beneficio de la *subestructura óptima* recursivamente como:

$$rn = max_{1<=i<=n}(p_i + r_{n-i})$$

donde se repite el proceso para cada pieza $r_{n-i}$ posterior. De este modo se puede implementar esta aproximación utilizando una simple rutina recursiva.

```java
/**
 * Método recursivo que resuelve el problema
 * @param size Tamaño
 */
public int cutRod(int size) {
   if(size == 0) {
      return 0;
   }
   int max = Integer.MIN_VALUE;
   for(int i = 1; i <= size; i++) {
      int aux = getRod().getCoste(i) + cutRod(size - i);
      if(max < aux) {
         max = aux;
      }
   }
   return max;
}
```

La ejecución del anterior algoritmo para el caso n=4 se puede representar mediante el siguiente árbol de recursividad:




Al analizarlo, podemos observar que el algoritmo realiza numerosas llamadas innecesarias al ya haber sido calculados sus resultados previamente. Por este motivo, en una primera vista podemos deducir que la complejidad de este algoritmo será alta.

Para calcular la complejidad deberemos saber cuántas formas hay de cortar una barra de tamaño *n*.

A tener en cuenta:

- Una barra de tamaño *n* tiene exactamente *n-1* posibles puntos de corte
- Podemos elegir cualquier número de puntos de los anteriores, para cualquiera de estos números (k) hay $\left(\frac{n-1}{k}\right)$ formas de cortarlo.

Por tanto, el número de formas distintas en las que cortar la barra es:

$$\sum_{k=0}^{n-1}\binom{n-1}{k}=\frac{\sum_{k=0}^{n-1}(n-1)!}{k!(n-1-k)!}=(1+1)^{n-1}=2^{n-1}$$

Este número se corresponde con el número de nodos hoja del árbol.

Adicionalmente, el número de nodos no terminales del árbol para un tamaño *n* es $2^{n-1}$ (Incluyendo la raíz). Por tanto, para cada tamaño *n*¸ el trabajo será:

$$T(n)=\left(\sum_{k=0}^{n-1}\left(\frac{n-1}{k}\right)\right)+\ 2^{n-1}=\ 2^{n}$$

Se debe tener en cuenta que *T(0) = 1*.

Lo anterior es equivalente a calcular, de una forma más directa, lo siguiente:

$$T(n)=\sum_{k=0}^{n-1}T(j)=\left(\sum_{k=0}^{n-1}2^{j}\right)+1=\ 2^{n}$$

La afirmación de que $T(n) = \sum_{k=0}^{n-1} T(j) = 2^n$ puede ser demostrada por inducción:

- Caso base: $T(0) = 2^0 = 1$ , lo cual es cierto por lo que afirmamos anteriormente.
- **Hipótesis inductiva:** $T(n) = \sum_{k=0}^{n-1} T(j) = 2^n$
- Paso inductivo:

$$T(n+1) = T(n) + \sum_{k=0}^{n-1} T(k) = T(n) + T(n) = 2 * 2^n = 2^{n+1}$$

# Programación dinámica

El algoritmo recursivo que resuelve el problema de corte de vástagos se basa en resolver subproblemas más pequeños que se repiten varias veces a lo largo de su ejecución. Por lo tanto, estamos ante un algoritmo que divide un problema en subproblemas que se solapan y, así, este problema se puede resolver de forma eficiente mediante el método de *programación dinámica*.

## *Top-Down*

Si nos fijamos atentamente en el árbol de recurrencia del algoritmo recursivo, podemos observar que realiza más trabajo del necesario, ya que un mismo subproblema se resuelve varias veces. De esta forma, se puede mejorar el tiempo de ejecución del algoritmo si almacenamos las soluciones a los subproblemas en vez de volver a recalcular el mismo subproblema de forma reiterada. No obstante, este tiempo de ejecución mejora a costa de utilizar memoria adicional, por lo que debe haber un compromiso entre la memoria utilizada y el tiempo de ejecución del algoritmo.

Para implementar esta mejora del algoritmo, se puede utilizar la estrategia de procesamiento *top-down*, que consiste en resolver los subproblemas recordando las soluciones por si fueran necesarias nuevamente, combinando la recursión y la memorización. En el caso del problema del corte de vástagos se utiliza el método recursivo anterior, pero se modifica de tal forma que almacena el beneficio óptimo de las diferentes longitudes del vástago para no calcularlas de nuevo.

El algoritmo que implementa esta estrategia sigue un recorrido ***primero en profundo***, ya que se llega primero a los nodos hojas y luego se va subiendo en el árbol.

## *Bottom-Up*

Otra forma de implementar el algoritmo que resuelve este problema con la programación dinámica es la estrategia *bottom-up*. Se resuelven los problemas empezando por las longitudes pequeñas y se guardan los beneficios óptimos de estas longitudes en un array (de tamaño n+1). Después, cuando se proceda a evaluar longitudes más grandes, simplemente buscamos estos valores para determinar el beneficio óptimo para la pieza de longitud más grande que las longitudes de las que ya se ha calculado el beneficio máximo.

Esta mejora del algoritmo hace un recorrido ***topológico inverso*** puesto que se resuelven primero los nodos hijos (longitudes del vástago más pequeñas) y, a continuación, se van resolviendo los nodos predecesores (longitudes más grandes del vástago).

## *Extended-Bottom-Up*

El algoritmo anterior se puede hacer más preciso si, además de guardar los beneficios máximos de las diferentes longitudes, se almacene los puntos donde se realicen esos cortes que hagan que el beneficio sea óptimo. Para ello, se utiliza un array adicional *s* (de tamaño $n + 1$) que guarde los cortes óptimos para cada tamaño del segmento. De esta manera, para saber los cortes que propician el beneficio máximo del vástago entero, se procede hacia atrás a través de los cortes, examinando $s[i] = i - s[i]$ y empezando en $i=n$ para ver donde se realiza cada corte subsecuente hasta que $i=0$, lo que indica que cogemos la última pieza sin más cortes.

```
/*
 * @param rod cuerda que se desea cortar.
 * @param n Tamaño a cortar
 */
public static void extendedBottomUpCutRod(Rod rod, Integer n){

   for (int j = 1; j <= n; j++){
      Integer q = new Integer(Integer.MIN_VALUE);
      for (int i = 1; i <= j; i++){

         if (q < (rod.getCoste(i) + getR().get(j - i))){
            q = rod.getCoste(i) + getR().get(j - i);
            getS().set(j, i);
         }
      }
```

```
      getR().set(j, q);
    }
  }
```

Para calcular el tiempo de ejecución del algoritmo utilizando la estrategia *bottom-up*, hay que tener en cuenta que se realizan dos bucles:

- Un bucle que va desde *j=1* hasta *n* para contemplar todas las longitudes posibles del vástago.
- Otro bucle que va desde *i=1* hasta j para contemplar todos los posibles cortes de un vástago de longitud *j*.

Por tanto, el tiempo de ejecución del algoritmo es:

$$T(n)=\sum_{j=1}^{n}\sum_{i=1}^{j}c=\sum_{j=1}^{n}jc=c\sum_{j=1}^{n}j$$

La última expresión es una progresión aritmética y, por tanto, utilizamos la siguiente fórmula:

$$s_n=\frac{(a_1+a_n)n}{2}$$

Por tanto:

$$c\sum_{j=1}^{n}j=c\frac{(1+n)+n}{2}=\Theta(n^2)$$

Como podemos observar, utilizando la estrategia *botttom-up* en el algoritmo ha permitido que el tiempo de ejecución del mismo haya pasado de ser *exponencial* a *polinomial*. Por lo tanto, esta estrategia permite que el problema de corte de vástagos sea tratable para tamaño de vástagos n suficientemente grande.

# Conclusiones

El problema de corte de vástagos se puede resolver de distintas formas. Una primera aproximación es utilizar un algoritmo recursivo que aplique la fuerza bruta, donde se contemplan todas las

posibilidades de corte sobre un vástago de un tamaño determinado, siendo el tiempo de ejecución un número exponencial. De esta manera, para tamaños muy grandes el problema sería intratable.

Para resolver este problema de forma eficiente se aplica la programación dinámica puesto que los subproblemas que se generan en su resolución se solapan y no es necesario recalcular cada subproblemas de forma repetida, por lo que aplicar la técnica *Divide y Vencerás* en este problema llevaría más tiempo del necesario. A partir de esto, existen dos tipos de implementación del algoritmo que resuelve este problema : *top-down* y *bottom-up*. El primer tipo de implementación se realiza añadiendo al algoritmo recursivo una estructura para almacenar los resultados de los subproblemas (que son los vástagos de menor tamaño) y el segundo ordena los subproblemas de menor a mayor (ordena los vástagos de menor a mayor tamaño) y los resuelve en ese orden, de tal forma que los subproblemas mayores hacen uso del resultado de los subproblemas más pequeños. Con estas implementaciones el problema se consigue resolver en tiempo polinomial, de modo que hace posible su tratabilidad con tamaños de entrada muy grande y , de este modo, se puede concluir que la programación dinámica es el mejor enfoque para resolver este problema.

# Bibliografía

*Proving number of calls made in cut-rod algorithm*. (2017). *Cs.stackexchange.com*. Revisado el 19 maezo de 2017, desde http://cs.stackexchange.com/questions/29661/proving-number-of-calls-made-in-cut-rod-algorithm

Pecelli, P. (2009). *Analysis of Algorithms - Dynamic Programming for Rod Cutting*. Revisado el 19 de marzo de 2017, desde http://www.cs.uml.edu/~kdaniels/courses/ALG_503_F12/DynamicRodCutting.pdf

*Lecture 12: Dynamic Programming - Rod Cutting*. (2017). *Faculty.ycp.edu*. Revisado el de 19 marzo de 2017, desde http://faculty.ycp.edu/~dbabcock/PastCourses/cs360/lectures/lecture12.html

Kretchmar, R. (2017). *Rod Cutting: Example DP Solution* (1st ed.). Revisado el 19 de marzo de 2017, desde http://personal.denison.edu/~kretchmar/271/DPRodCuttingExample.pdf