**Universidad Tecnológica Nacional**

**Facultad Regional de Resistencia**

****

**Complejidad y Técnicas de Algoritmos**

**Profesores:**

* **Acuña, Cesar.**
* **Tortosa, Nicolás.**

**Alumnos:**

* **Encina, Axel.**
* **Mass, Matías.**
* **Sosa, Diego.**

**Grupo:**

* **Número de grupo: 2 - PyLove.**

**Año:2021**

**Indice:**

[**Introducción:**](#_o2jiktmx9ani) **3**

[**Rod Cutting Problem - Problema de la Varilla**](#_hzxg9p6rfzp5) **4**

[**Ávidos o Voraces (Greedy)**](#_uxorjrqp4ylq) **5**

[**Programación Dinámica**](#_2aks5dclbbo5) **7**

[Ecuación de recurrencia](#_9luscf680c16) 7

[Tabla](#_a5m7x5nmgnj7) 7

[Código en Python](#_u8jmqwht52l1) 8

[Complejidad:](#_tt1s6fc0c8lg) 8

[**Backtracking (Vuelta Atrás)**](#_75f5192jb8dv) **9**

Árbol 9

[Código en Python](#_fxanj71a4kfn) 10

[Complejidad:](#_k96qvhh8tos) 12

**Comparación 13**

[**Conclusión**](#_5gdkcc9uwixo) **14**

[**Referencias**](#_g8jy83jdjn8) **15**

# **Introducción:**

En la siguiente entrega, se pondrá a prueba lo aprendido durante la cursada trabajando en un problema en cuestión, en este caso el Problema de Varilla (Rod Cutting) el cual buscará ser resuelto mediante tres técnicas diferentes de resolución de paradigmas algorítmicos los cuales son Ávidos (Greedy), Programación Dinámica y Backtraking (Vuelta Atrás).

En primer lugar se mostrará una presentación del escenario, su metodología, con ejemplos del mismo y seguido a este sus formas de resolverlo junto con los planteos y resultados obtenidos. Comenzaremos por la resolución mediante Ávidos utilizando un contraejemplo para su explicación, seguido de Programación Dinámica junto con la ecuación de recurrencia, la tabla y su código correspondiente y por último Backtraking con su árbol binario, nuestro planteo para su solución y su respectivo código. En cada caso se mostrará la complejidad de los algoritmos que conlleva cada técnica.

Cabe destacar que el lenguaje de programación que utilizamos en cada caso es Python.

Se anexará dentro del documento el hipervínculo al árbol de Backtraking, para una mejor visualización del mismo.

# **Rod Cutting Problem - Problema de la Varilla**

El problema de la varilla consta vale la redundancia de una varilla de n centímetros de largo y una variedad de precios arbitrarios (no infiere que porque tenga más tamaño deba tener mayor costo) de todas las piezas de tamaño menores que n, siendo n valores enteros mayores a 1. El problema consiste en determinar el mayor valor que se puede obtener cortando la varilla y vendiendo las piezas.

Por ejemplo, si la longitud de la varilla es 8 y los valores de diferentes piezas son:

| Longitudes | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Precios | 1 | 5 | 8 | 9 | 10 | 13 | 15 | 19 |

Ejemplos de algunos cortes posibles:

Si no corto la vara entonces: 8 → 19$

Longitudes: 7 + 1 → 15 + 1 = 16$

Longitudes: 6 + 2 → 13 + 5 = 18$

Longitudes: 4 + 4 → 9 + 9 = 18$

Longitudes: 4 + 4 → 9 + 9 = 18$

Longitudes: 3 + 3 + 2 → 8 + 8 + 5 = 21$ → **MÁXIMO**

Longitudes: 2 + 2 + 2 + 2 → 5 + 5 + 5 + 5 = 20$

Longitudes: 1 + 1 + 1 + 2 + 3 → 1 + 1 + 1 + 5 + 8 = 16$

Longitudes: 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 → 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 8$

Entonces el valor máximo que se puede obtener para una varilla de 8 cm es $21 (cortando en tres piezas, dos de longitud 3 y una de longitud 2).

# **Ávidos o Voraces (Greedy)**

Con este método de resolución de algoritmos, sucede de forma análoga que con el famoso Problema del Cambio (monedas). Pongamos en contexto:

Si tengo un conjunto M = {25, 10, 5, 1} que representa mis monedas en cuanto a valor, busco dar cambio a un valor n dando la menor cantidad de monedas para alcanzarlo; entonces mediante cada paso de ejecución utilizando ávidos voy a elegir un tipo de moneda, en la que según mi condición sea la más óptima para el subproblema más pequeño que planteo (problemas locales).

Si el vuelto a considerar es de 38 centavos, entonces:

38 = 25 + 10 + 1 + 1 + 1 → 5 Monedas

Por lo que utilizando Algoritmos Ávidos es eficiente y no causa “problemas”. Pero si mis valores de monedas fueran M = {25, 19, 10, 5, 1} y quisiera devolver otra vez 38 centavos:

38 = 25 + 10 + 1 + 1 + 1 → 5 Monedas → Con Greedy

Pero la solución más óptima general en realidad es:

38 = 19 + 19 → 2 Monedas

Por lo que se puede ver que es que si el conjunto que representa mis valores no sigue un patrón, no puede realizarse con algoritmos ávidos.

Volviendo al Rod Cutting (Problema de la Varilla) sucede lo mismo y es que si las ganancias no son lineales (considerando que como mínimo no se duplican unas de las otras) no puede resolverse con ávidos. Por ejemplo: si los precios para la vara considerando de 1cm a 6cm fueran P = {1, 5, 8, 8, 10, 15} sería irresoluble mediante paradigmas de solución tipo Greedy (donde $1 corresponde a 1cm, $5 a 2cm y así sucesivamente)

El algoritmo ávido lo que hará es elegir la primera pieza (primer recorte) como si fuera la más “brillante y atractiva”, y después se bloquea para el resto del problema, con una longitud restante muy subóptima de acuerdo con la tabla de precios. Siendo esta arbitraria por lo que en el mayor de los casos es no lineal, y en el hipotético caso de que si fuese lineal estaríamos empleando como si fuera un tratamiento de cambio en monedas y considerando el escenario en cuestión sería muy absurdo.

En retrospectiva lo que sucede es que el máximo local que considera el algoritmo no va a ser un máximo global lo que deriva a un resultado no óptimo.

**Hay que tener en cuenta que en el problema de la varilla el objetivo de la optimización no es encontrar la mejor relación precio - longitud para una sola pieza, sino que es encontrar el mejor precio total para la cantidad de varilla que realmente tenemos.**

# **Programación Dinámica**

Para este método planteamos la solución como una definición recursiva, pero obviamente resolvemos los subproblemas una sola vez, guardando los resultados parciales en una tabla de dos dimensiones. Comenzamos planteando la tabla, junto con su ecuación de recurrencia y construimos la solución óptima haciendo uso de la información guardada en la tabla.

## **Ecuación de recurrencia**

****

Dónde:

PrecioMax: Tabla de dos dimensiones.

precios: Arreglo de los precios para cada tamaño de varilla.

F: Filas (representan los costos).

C: columna (tamaños de varitas).

Donde la función mayor devuelve el precio mayor.

## **Tabla**

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 = $1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2 = $5 | 0 | 1 | 5 | 6 | 10 | 11 | 15 | 16 | 20 |
| 3 = $8 | 0 | 1 | 5 | 8 | 10 | 13 | 16 | 18 | 21 |
| 4 = $9 | 0 | 1 | 5 | 8 | 10 | 13 | 16 | 18 | 21 |
| 5 =$10 | 0 | 1 | 5 | 8 | 10 | 13 | 16 | 18 | 21 |
| 6 = $13 | 0 | 1 | 5 | 8 | 10 | 13 | 16 | 18 | 21 |
| 7 = $15 | 0 | 1 | 5 | 8 | 10 | 13 | 16 | 18 | 21 |
| 8 = $19 | 0 | 1 | 5 | 8 | 10 | 13 | 16 | 18 | 21 |

Tabla de programación dinámica para Cutting Rod con longitud(n) = 8

Una vez obtenida la tabla, planteamos la solución en el lenguaje.

## **Código en Python**

| import time  def rod\_cutting\_problem(precios, n):  tabla = [0 for i in range(n+1)]  for i in range(1, n + 1):  for j in range(1, i + 1):  tabla[i] = max(tabla[i], precios[j - 1] + tabla[i - j])  return tabla[n]  precios = [1, 5, 8, 9, 10, 13, 15, 19]  n = 8  inicio = time.time()  precioMax = rod\_cutting\_problem(precios, n)  fin = time.time()  print(f"La ganancia es {precioMax}")  print("Tiempo en segundos: ", fin - inicio) |
| --- |

Vemos que el resultado obtenido es el correcto ya que para para n = 8 cm el valor máximo que se puede obtener es $21 (cortando en tres piezas, dos de longitud 3 y una de longitud 2).

Longitud 8: 3 + 3 + 2 → 8 + 8 + 5 = 21$

Este código puede ser utilizado para cualquier n > 0 siempre y cuando la lista de precios tenga el mismo tamaño que n, si quisiera un n = 11 debería darle un valor de costo a una vara de 9, 10 y 11 cm también.

## La Complejidad de este algoritmo es deO(N2) porque cuenta con dos bucles que rellenan la matriz de dos dimensiones (i \* j)

# **Backtracking (Vuelta Atrás)**

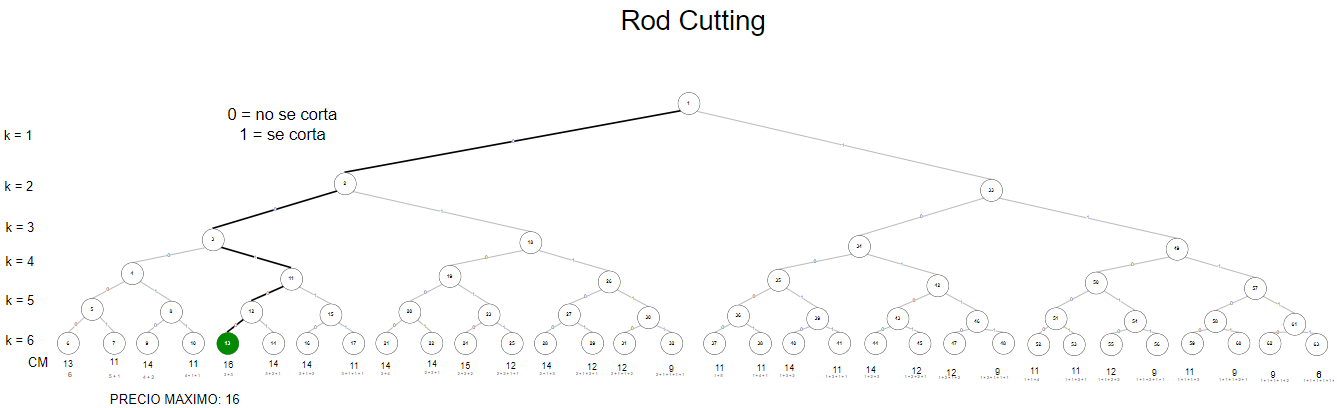
Para este método realizamos un árbol binario donde el planteo fue el siguiente:

Tomamos 6 posibles cortes con sus respectivos precios en cada tamaño en centímetros

Piezas = {1, 2, 3, 4, 5, 6}

Precios = [1, 5, 8, 9, 10, 13]

Archivo draw.io del Árbol: <https://app.diagrams.net/#G1Ft7nGkI2dRR2Q550-juSa0m3ZCPhv6sx>



En el Nodo Raíz (N°1) partimos de tener la Varilla de 6cm donde si voy hacia la derecha (1) lo que hago es realizar un corte en el extremo y en cambio si me dirijo hacia las hojas por la izquierda (0) no produzco ningún corte y avanzo en la varilla.

A medida que vamos bajando en el árbol va aumentando k (que representa un lugar de corte), y siempre el camino de la derecha representa una separación en la varilla, los cual va a generar un nuevo fragmento donde el tamaño de este depende de donde se realizó el corte, y si existe otro corte con un k mayor o también si llegue al final de la varilla.

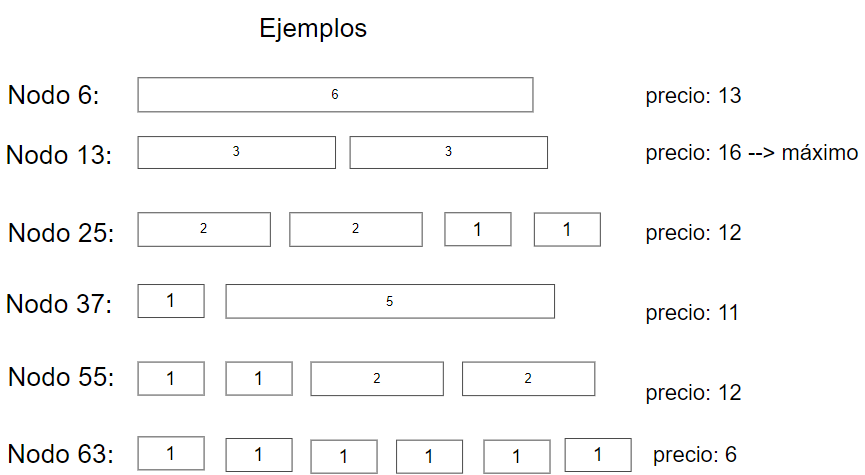
Por ejemplo si tuviera cortes en k = 2 y k = 5 entonces mi vara resultante sería la siguiente:



Y calculando según la lista de precios:

Longitudes: 2 + 3 + 1 → 5 + 8 + 1 = 14$

Entonces mediante esta metodología los hijos son prolongaciones posibles al añadir nuevas etapas y lo que se busca es el nodo hoja que sume el resultado mayor. A continuación algunos ejemplos:

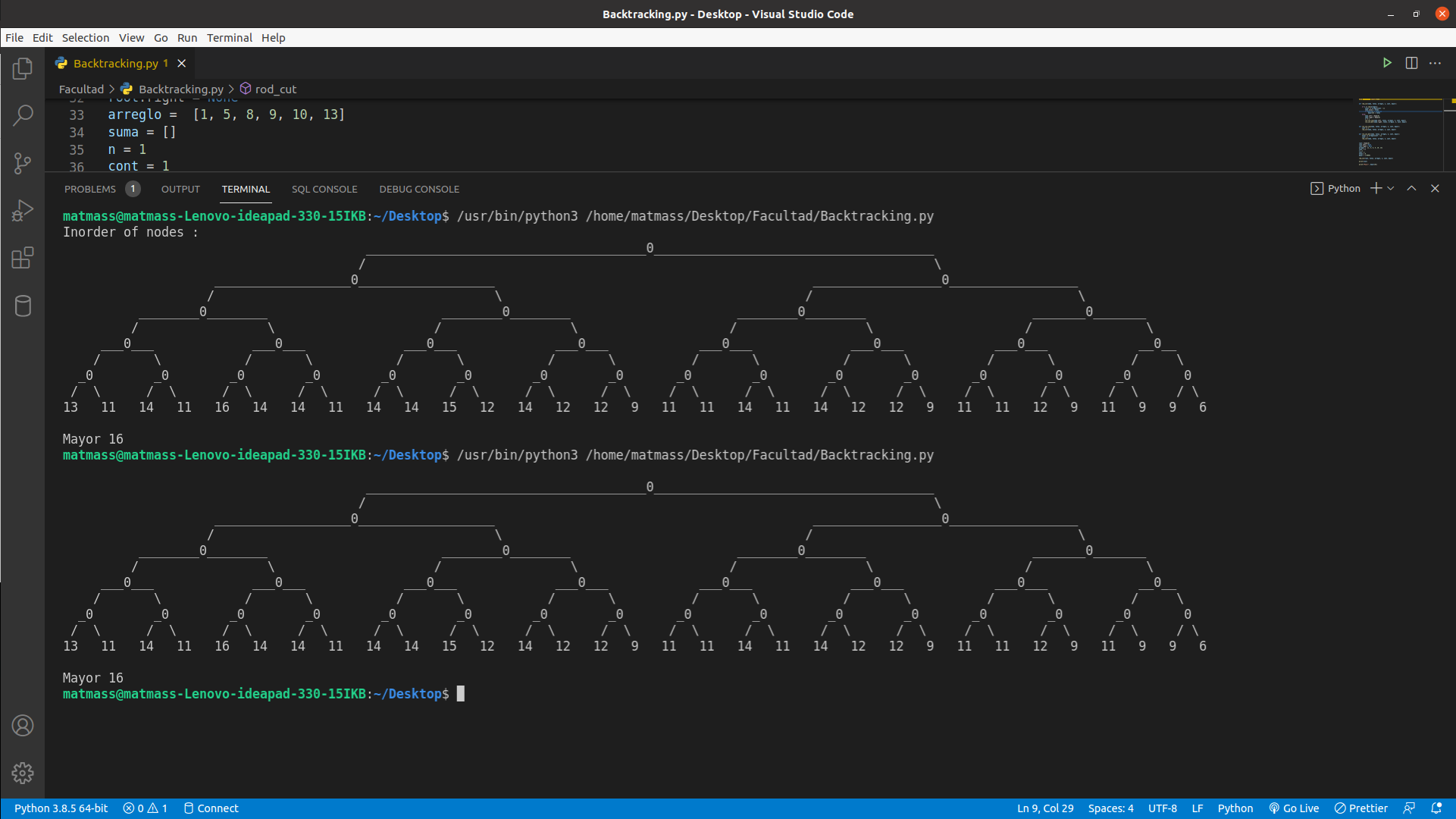
****

Entonces el valor máximo que se puede obtener es $16 para una varilla de 6 cm (cortando en dos piezas, las dos de longitud 3 cm).

## **Código en Python**

| import time import sys from binarytree import Node   def rod\_cut(nodo, total, arreglo, n, cont, mayor): """Función para calcular el máximo ganancias (en dinero) dependiendo del tamaño de la varilla y si se corta o no """  if n == len(arreglo):  *# Cuando lleva al último nivel, guardamos los cortes de las varillas con su precio máximo en el nodo hoja*  total += arreglo[cont - 1]  nodo.value = total  if mayor[0] < total:  *# Guardamos en la lista el número de la ganancia máxima*  mayor[0] = total  else:  *# Si no es el último nivel, creamos los dos nodos izq y derecha, y vamos expandiendo el árbol por izq y der*  nodo.left = Node(0)  nodo.right = Node(0)  n += 1 *# Contador para los niveles del árbol (k = 1, k = 2, etc)*  rot\_cut\_izq(nodo.left, total, arreglo, n, cont, mayor)  rot\_cut\_der(nodo.right, total, arreglo, n, cont, mayor)   def rot\_cut\_izq(nodo, total, arreglo, n, cont, mayor):  """Función para ir expandiendo el árbol por la izquierda"""  cont += 1 *#Guarda el tamaño de la varilla sin cortar*  rod\_cut(nodo, total, arreglo, n, cont, mayor)   def rot\_cut\_der(nodo, total, arreglo, n, cont, mayor):  """Función para ir expandiendo el árbol por la derecha"""  total += arreglo[cont - 1] *# Cada vez que se corta, vamos sumando el precio máximo de la longitud de la varilla*  cont = 1 *# Se reinicia el contador cuando se corta la varilla*  rod\_cut(nodo, total, arreglo, n, cont, mayor)   root = Node(0) # Creamos el nodo raiz root.left = None root.right = None arreglo = [1, 5, 8, 9, 10, 13]  n = 1 cont = 1 total = 0 mayor = [-sys.maxsize - 1] *# Menor valor de un entero en python*  inicio = time.time() rod\_cut(root, total, arreglo, n, cont, mayor) fin = time.time()  *# print(root) Imprime el árbol en forma gráfica*  print("Mayor", mayor[0]) print("Tiempo en segundos - Backtracking: ", fin - inicio) |
| --- |

Nota: Debe estar instalado el módulo binarytree.

****

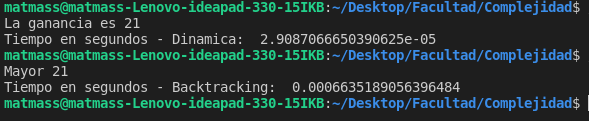
La complejidad de este algoritmo es O(2n) por el hecho de estar trabajando con estructuras recursivas que se solapan entre sí.

Ejemplo: Con una vara de 8 cm hay 128 maneras distintas de cortar la varilla, si fuera de 20 cm existirían 1048576 de maneras de cortarla.

**Comparación:**

| Programación dinámica | Backtracking | Ávidos o voraces |
| --- | --- | --- |
| Produce una solución prácticamente instantánea gracias a la matriz que almacena los resultados parciales de las llamadas recursivas. El orden de complejidad es O(n2), que se ve que es mucho mejor comparado con la técnica de Backtracking, es decir, con los recursos que utiliza. | La ejecución de la solución es más lenta, ya que tiene una complejidad O(2n), si se fuera a emplear con un n de tamaño 100, llevaría mucho más tiempo comparando con programación dinámica. | No se puede hacer. |

Como comparación final, se puede ver en la imagen la cantidad de tiempo en segundos empleando ambas técnicas:



Backtraking→ 6.63 \* 10 ^-4 s

Dinámica→ 2.9\*10 ^-5 s

Para n = 8 mediante programación dinámica el resultado es 22 veces más rápido que utilizando Backtraking. Es de grato entendimiento que si n fuese un número más grande esa distancia de tiempo iría aumentando exponencialmente.

# **Conclusión:**

En este documento se logró presentar como resolver un problema de sencillo entendimiento, pero no así de resolución aplicando diferentes paradigmas de solución. Se logró poner en juego lo aprendido durante la cursada, además de aplicar nuestros conocimientos sobre la programación en especial en un lenguaje como Python. Se pudo demostrar la complejidad con la que cuenta cada método, y además aplicando el uso de un contraejemplo para explicar el por qué no se puede aplicar la metodología de algoritmos Greedys.

A lo largo del documento, se encuentran ejemplos más que claros e ilustrativos de cómo es el problema, particionando en pasos para diferentes valores y cómo trabaja la solución para cada uno de estos valores.

Al final también se añadió un cuadro comparativo con una comparación en segundos denotando la eficiencia que resalta utilizar programación Dinámica por sobre Backtraking.

# **Referencias:**

Apuntes de la cátedra: Backtraking - Extra, Cap 4, 5 y Cap 6.

Radford - EDU: <https://www.radford.edu/~nokie/classes/360/dp-rod-cutting.html>

Stanford - EDU: <https://web.stanford.edu/class/archive/cs/cs161/cs161.1168/lecture12.pdf>

Codesdope - COM: https://www.codesdope.com/course/algorithms-rod-cutting/

Elvex - UGR: <https://elvex.ugr.es/decsai/algorithms/slides/5%20Backtracking.pdf>