1.5em

DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Jambotubos

Algoritmos y Estructuras de Datos III
Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Matias Nahuel Castro Russo	203/19	castronahuel14@gmail.com
Maximo Yazlle	310/19	myazlle99@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

El problema de Jambotubos consiste en determinar, dados dos números naturales R y n, los cuales representan la resistencia del tubo y la cantidad de productos respectivamente, y dos listas de n valores naturales r_i y w_i (representando peso y resistencia de cada producto) donde $0 <= i < n$, cuál es la máxima cantidad de productos que puedo apilar en un tubo sin que la suma de los pesos de los productos no supere R y, que para cada producto, la suma de los pesos de los productos apilador por encima, no supere su resistencia. Sólo se puede apilar en el orden en que son recibidos.

A continuación presentamos algunos ejemplos:

1. Dados R=50 , n=5, r=[45,8,15,2,30] , w=[45,8,15,2,30], la solución en este caso es 3. Esto se logra eligiendo los productos: 1,3 y 4. De esta manera, respeto la resistencia del tubo (R), y la de cada producto. Pero agregando algún producto más la supero. Para los otros casos podemos observar que:
   - Al apilar el producto 2 o 4, no voy a poder agregar más productos, ya que el peso de cualquier otro superaría su resistencia.
   - Si comienzo apilando el producto 3, luego podemos apilar el 4 sin superar su resistencia, pero luego cualquier otro producto la superaría.
2. Dados R=25, n=5, r=[20,10,5,50,2], w=[5,21,20,22,15]. La solución es 2. Esto ocurre si apilamos los productos: 1 y 5. Con esta combinación respetamos la resistencia del tubo y de cada producto.
   - Si apilo el producto 1, sólo puedo apilar el 5 sin pasarme de R.
   - Al apilar el producto 2, 3 o 4, si apilo otro producto se pasaría de R.

El objetivo del presente trabajo es abordar el problema utilizando tres técnicas de programación distintas y evaluar la efectividad de cada una de ellas para diferentes conjuntos de instancias. En primer lugar se utiliza Fuerza Bruta que consiste en enumerar todas las posibles soluciones, de manera recursiva, buscando aquellas que son factibles. Luego, se introducen podas para reducir el número de nodos de este árbol recursivo en busca de un algoritmo más eficiente, obteniendo un algoritmo de Backtracking. Finalmente, se introduce la técnica de memoización para evitar repetir cálculos de subproblemas. Esta última técnica es conocida como Programación Dinámica (DP, por sus siglas en inglés).

El trabajo va a estar ordenado de la siguiente manera: primero en la Sección 2 se define el algoritmo recursivo de Fuerza Bruta para recorrer todo el conjunto de soluciones y se analiza su complejidad. Más tarde, en la Sección 3 se explica el algoritmo de Backtracking con un breve análisis de mejores y peores casos. Luego, se introduce el algoritmo de DP en la Sección 4 junto con la demostración correspondiente de correctitud y un análisis de complejidad. Finalmente, en la Sección 5 se presentan los experimentos con su respectiva discusión, y en la Sección 6 las conclusiones finales.

2. Fuerza Bruta

Un algoritmo de *Fuerza Bruta* enumera todo el conjunto de soluciones en búsqueda de aquellas factibles u óptimas según si el problema es de decisión u optimización. En este caso, el conjunto de soluciones está compuesto de números naturales que equivalen a la cantidad de productos que pueden ser apilados en un tubo.

La idea del Algoritmo 1 para resolver el problema de Jambotubos es ir generando las soluciones de manera recursiva, decidiendo en cada paso si uno de los productos es considerado o no, y quedándose con la mejor solución de alguna de las dos ramas. Finalmente, al identificar una solución, determina si es factible y de ser así, devuelve el cardinal de la solución.

Algorithm 1 Algoritmo de Fuerza Bruta para Jambotubos.

```
1:  solucion_actual ← 0
2:  agregados ← vector(n, false)
3:  function FB(i, k, t)
4:      if i = n then
5:          if t <= R and not rompeResistencia(n) then
6:              solucion_actual ← max(solucion_actual, k)
7:      else
8:          agregados[i] ← true
9:          FB(i + 1, k+1, t + w[i])
10:         agregados[i] ← false
11:         FB(i + 1, k, t)
12:
```

rompeResistencia es una función que dado los productos modelados por las listas r y w, y la lista $agregados$ que indica si un producto esta o no esta, valida si la resistencia de cada producto no es superada por los productos que la apilan, es decir, si se rompe algún producto.

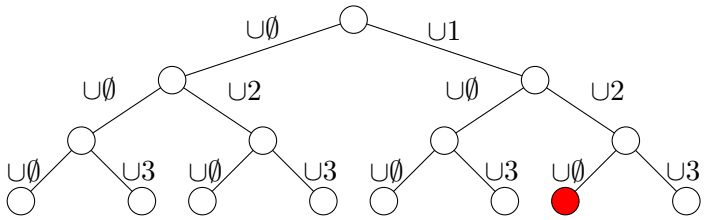


Figura 1: Ejemplo de ejecución del Algoritmo 1 para $P = \{1, 2, 3\}$ (productos) donde w = $\{5, 5, 9\}$, r = $\{20, 10, 5\}$ y $R = 10$.
En rojo la solución correcta al problema $\{1, 2\}$.

La complejidad del Algoritmo 1 para el peor caso es $O(n * 2^n)$. Esto es así porque por un lado, el árbol de recursión es un árbol binario completo de n + 1 niveles (contando la raız), dado que cada nodo se ramifica en dos hijos y en cada paso el parámetro i es incrementado en 1 hasta llegar a n. Y por otro lado, es preciso observar que la solución de cada llamado recursivo se obtiene en tiempo n ya que rompeResistencia es una función que tiene complejidad O(n). Esto se va a realizar 2^n veces por ser un árbol de recursión (2^n hojas), por lo que la complejidad queda $O(n * 2^n)$.

Podemos concluir, también, que el algoritmo se comporta de igual manera frente a todos los tipos de instancia, dado que siempre genera el mismo número de nodos. Dicho de otra forma, el conjunto de instancias de peor caso es igual al conjunto de instancias de mejor caso.

3. Backtracking

Un algoritmo de *Backtracking* se basa en una idea similar a Fuerza Bruta, pero con algunos cambios. De igual manera, se enumera todo el conjunto de soluciones formando un árbol de Backtracking (similar a Fuerza Bruta). En este caso también esta compuesto de números naturales que representa la cantidad de productos que pueden ser apilados, al igual que el de Fuerza Bruta. La diferencia se encuentra en las podas que son reglas que permiten evitar explorar partes del árbol donde sabemos que no va a existir ninguna solución de interés. En particular usamos dos clases de podas, por *factibilidad* y por *optimalidad*. La idea de este Algoritmo 2 para resolver el problema de Jambotubos, es la misma que en el Algoritmo 1 de Fuerza Bruta, solo que en cada instancia chequeamos ambas podas, si alguna se cumple, simplemente no se sigue por ese camino y se poda el árbol en ese nodo. A continuación detallamos la idea de cada poda:

Poda por factibilidad Sean r' y w' los productos de la solución parcial representada en un nodo intermedio n_0 tal que cada r'_i y w'_i representan la resistencia y peso del $i-esimo$ producto dentro del Jambotubo, y R su resistencia. De esta manera, si la suma de, todos los pesos de los productos agregados, es mayor a R, o si algún producto se rompe porque el peso de los productos que tiene apilados es mayor a su resistencia r_i, entonces no va a haber ninguna forma de extender o conservar la solución. Por lo tanto, podamos ese nodo y evitamos seguir explorando el subárbol formado por debajo de n_0. Con esta poda conseguimos reducir la cantidad de operaciones de nuestro algoritmo. Esta poda está expresada en la línea 5 del Algoritmo 2.

Poda por optimalidad Supongamos que ya se conoce una solución factible para el problema con cardinal K, donde $solucion\_actual = K$. Además, supongamos que se está en un nodo intermedio n_0 que representa a una solución parcial con k elementos, y se va a considerar al producto i. En este caso, si $k + n - i \leq solucion\_actual$, como cualquier decisión que se tome a continuación va a agregar o mantener la cantidad de productos agregados, podemos asegurar que cualquier solución factible que sea una extensión de este nodo, no nos va a ser de interés, será peor o a lo sumo igual a la solución actual. Por lo tanto, podamos esta rama y evitamos operaciones innecesarias explorando el subárbol de n_0. Esta poda está expresada en la línea 4 del Algoritmo 2.

Algorithm 2 Algoritmo de Backtracking para Jambotubos.

```
1:  solucion_actual ← 0
2:  agregados ← vector(n, false)
3:  function BT(i, k, t)
4:      if k + n − i <= solucion_actual then return
5:      if t > R or rompeResistencia(n) then return
6:      if i = n then
7:          if t <= R and not rompeResistencia(n) then
8:              solucion_actual ← max(solucion_actual, k)
9:          agregados[i] ← true
10:         BT(i + 1, k+1, t + w[i])
11:         agregados[i] ← false
12:         BT(i + 1, k, t)
13:
```

La complejidad teórica del algoritmo en el peor caso es $O(n^2 * 2^n)$. Esto es así, porque en el peor escenario no se lograría podar ninguna rama y por lo tanto se termina enumerando el árbol completo al igual que en *Fuerza Bruta*. Además, podemos ver que el código introducido en

las líneas 4 y 5 hacen que en cada iteración se pueda ejecutar la función *rompeResistencia* que tiene complejidad $O(n)$ (el resto son operaciones constantes). Esto hace que en caso de evaluar esa condición en las podas, cosa que debería suceder en el peor caso, nos termine quedando la complejidad teórica $O(n^2 * 2^n)$.

Decimos teórica ya que, para que no se realice la poda por *optimalidad*, la suma de los productos restantes a los ya apilados, debería ser mayor a la solución actual.

Por un lado, el peor caso de la poda por *optimalidad* es que no se puedan agregar ningún producto. Esto implica que no existe algún producto el cual su peso sea menor a R, ya que si existiese, podría agregarlo. Pero si todos los productos pesan más que R, quiere decir que ningún producto es factible, por lo que este seria el caso donde la poda por *factibilidad* pode siempre, ya que $t > R$. Es decir, el mejor caso para la poda por *factibilidad*.

Por otro lado, para el peor caso de *factibilidad*, buscamos que se ejecute la función *rompeResistencia*, pero que no de verdadera. Por como está implementada, para evaluar *rompeResistencia*, primero debe ser falso $t > R$, y esto implica que el peso acumulado no supere a R. Pero esto nos permite agregar productos. Más aún, para que *rompeResistencia* de falso siempre, debe permitir agregar todos los productos. Por lo tanto, el peor caso de esta poda es aquella instancia en la que todos los productos se pueden agregar. Vemos entonces que los peores casos de las podas son mutuamente excluyentes y, si uno da verdadero, va a generar que el otro de falso.

Creemos, por todo esto, que la complejidad en el peor caso práctico debe ser menor a la de Fuerza Bruta. Analizaremos mejor este caso en el *Experimento 2* de la *Sección 5*

Por otra parte, el mejor caso ocurre con la familia de instancias donde todos los pesos son mayores a R y la resistencia de todos los productos. Esto es así, porque entraría por la condición de la poda de *factibilidad* que pide $t > R$. Como la primer parte del **or** se cumple, no va a evaluar la segunda, y esto va a retornar. Cuando se ejecuta la función por primera vez, entra por el *else*, y al realizar el llamado recursivo en la línea 11, la ejecución de esa llamada va a entrar en la poda de *factibilidad*, como mencionamos antes, ya que se le agrega a t el peso del producto agregado. Haciendo esto, lo podemos acotar por una constante. Sin embargo, el llamado recursivo en la línea 12 no le agrega peso a t, por lo que no entraría en esa poda y se repite el primer escenario, el que agrega producto y hace cosas $O(1)$, y el que no agrega producto. Pero ambos lo vamos a hacer n veces, por lo que la complejidad del mejor caso termina quedando $O(n^2)$.

4. Programación Dinámica

Un algoritmo de *Programación Dinámica* se usa cuando un problema recursivo tiene superposición de subproblemas. La idea es sencilla y consiste en evitar recalcular todo el subárbol correspondiente si ya fue hecho con anterioridad. Definimos la siguiente función recursiva que resuelve el problema de Jambotubos:

$$f(i,t,rp) = \begin{cases} -1 & \text{si } t > R \vee rp < 0, \\ 0 & \text{si i == n,} \\ \text{máx}\{f(i+1,t,rp), 1 + f(i+1, t + w[i], min(r[i], rp - w[i]))\} & \text{caso contrario.} \end{cases}$$

Coloquialmente, podemos definir $f(i,t,rp)$: "máxima cantidad de productos de P=$\{P_i, \ldots, P_n\}$ con peso acumulado (t) menor que R y, la resistencia parcial (rp) mayor o igual a cero.". Podemos decir que, $f(0,0,R)$ es "máxima cantidad de productos de un subconjunto de P cuyo peso total sea menor que R y cuya resistencia parcial no sea negativa" que es precisamente la solución del problema. Veamos que la recursión es efectivamente lo que dice su definición coloquial.

Correctitud

(i) Si $t > R$ entonces el peso sumado (representado por t) es mayor a la resistencia del Jambotubo. Entonces, la respuesta es $f(i, t, rp) = -1$.

(ii) Si $rp < 0$ entonces la resistencia parcial llegó paso a ser negativa, eso quiere decir que se rompió algún producto o el Jambotubo. Entonces, la respuesta es $f(i, t, rp) = -1$.

(iii) Si $i = n$ entonces que no hay mas productos posibles para apilar tras ese nodo, es decir, se terminó de analizar esa rama del arbol. Este seria el caso base, donde sencillamente la respuesta es $f(i, t, rp) = 0$

(iv) En este caso, $i \leq n$, $t < R$, y $rp \geq 0$ entonces estamos buscando la máxima cantidad de productos del subconjunto de $P^i = \{P_i, \ldots, P_n\}$ cuyo $t < R$y su $rp > 0$. De existir un subconjunto, tiene que tener al i-ésimo producto o no tenerlo. Si no lo tiene, entonces tiene que ser un subconjunto de P^{i+1} donde $t < R$ y su $rp > 0$, por lo tanto, lo encontramos de manera recursiva $f(i + 1, t, rp)$. En cambio, si tiene al i-ésimo producto, entonces el resto de la solución debe ser $t < R$ y la resta de rp y la resistencia debe ser mayor a 0. Esto es f(i+1,t+w[i],min(r[i], rp-w[i])). Por lo tanto, la solución con más productos es $f(i, t, rp) = \text{máx}\{f(i + 1, t, rp), 1 + f(i + 1, t + w[i], min(r[i], rp - w[i]))\}$. Tener en cuenta que el término de la derecha suma 1 por haber agregado el i-ésimo producto.

Memoización Notemos que la función recursiva toma tres parámetros $i \in [0, \ldots, n]$, $r \in [0, \ldots, R]$, y $t \in N$. Notar que los casos $i = n$, $t > R$ o $rp < 0$ son casos base y se pueden resolver ad-hoc en tiempo constante. La cantidad posible de *estados* con los que se puede llamar a la función está determinada por la combinación de los parámetros rp e i. En este caso, ya que rp toma como valor R y siempre decrementa, hay $\Theta(n * R)$ combinaciones posibles de casos. Entonces, teniendo una memoria que recuerde cuando un caso ya fue resuelto y su correspondiente resultado, podemos calcular por única vez cada uno de ellos y asegurarnos de no resolver mas de $\Theta(n * R)$ casos. El *Algoritmo 3* muestra esta idea de memoización aplicada a la función recursiva en la línea 5. Este solamente se ejecuta si el estado no fue computado anteriormente.

Algorithm 3 Algoritmo de Programación Dinámica para Jambotubos.

```
M_{i,rp} ← ⊥ for i ∈ [0, n − 1], w ∈ [0, R].
function DP(i, r, rp)
    if t > R or rp < 0 then return −1
    if i = n then return 0
    if M_{i,rp} = ⊥ then M_{i,rp} ← máx{DP(i + 1, t, rp), 1 + DP(i + 1, t + w_i, min(r_i, rp − w_i))}
    return M_{i,rp}
```

La complejidad del algoritmo está determinada por la cantidad de estados que se resuelven y el costo de resolver cada uno de ellos. Ya vimos que se resuelven a lo sumo $O(n * R)$ estados distintos, y al ser todas las lineas $O(1)$, ya que todas realizan operaciones constantes, entonces cada estado se resuelve en $O(1)$. También tenemos el costo de inicializar la matriz que usamos como diccionario, que tiene un costo de $\Theta(n * R)$, y podemos implementarla de modo que se acceda y escriba en $O(1)$. Por lo tanto, la complejidad en el peor caso de nuestro algoritmo es $O(n*R)$. Como la inicialización tiene costo $\Theta(n*R)$ entonces esta va a ser su complejidad tanto en el mejor como en el peor caso.

5. Experimentos

En esta sección se presenta los experimentos realizados para evaluar los distintos métodos. Las ejecuciones fueron realizadas en una computadora con AMD Ryzen 5 @ 3.4 GHz y 32 GB de memoria RAM DDR4, y utilizando el lenguaje de programación C++.

5.1. Métodos

Las configuraciones y métodos utilizados durante la experimentación son los siguientes:

- **FB:** Algoritmo 1 de Fuerza Bruta de la *Sección 2*.
- **BT:** Algoritmo 2 de Backtracking de la *Sección 3*.
- **BT-F:** Algoritmo 2 con excepción de la línea 4, es decir, solamente aplicando podas por factibilidad.
- **BT-O:** Similar al método BT-F pero solamente aplicando podas por optimalidad, o sea, descartando la línea 6 del Algoritmo 2.
- **DP:** Algoritmo 3 de Programación Dinámica de la *Sección 4*.

5.2. Instancias

Para evaluar los algoritmos en distintos escenarios es preciso definir familias de instancias conformadas con distintas caracterısticas. Por ejemplo, el algoritmo de Backtracking como se menciona en la Sección 3 tiene familias que producen mejores y peores casos para el algoritmo. Los *datasets* definidos se enumeran a continuación.

- **res_peso_desc**: En esta familia cada instancia tiene los productos ordenados de manera descendiente de acuerdo al $\frac{resistencia}{peso}$ de cada producto.
- **res_peso_asc**: En esta familia cada instancia tiene los productos ordenados de manera ascendente de acuerdo al $\frac{resistencia}{peso}$ de cada producto.
- **bt-mejor-caso**: Cada instancia de n elementos tiene productos que rompen toda resistencia, está formada por $r = \{1, \ldots, 1\}$ y $w = \{2, \ldots, 2\}$ con $R = 1$.
- **bt-peor-caso**: Cada instancia de n elementos tiene hasta la mitad productos con resistencias que superan a todo peso, y la segunda mitad con pesos que superan a toda resistencia. Es decir, esta formada por $r = \{R, \ldots, R, 1, \ldots, 1\}$ y $w = \{R, \ldots, R\}$ con $R = n * n$.
- **bt-o-peor-caso**: Exactamente las mismas instancias que el dataset *bt-mejor-caso*. $r = \{1, \ldots, 1\}$ y $w = \{2, \ldots, 2\}$ con $R = 1$
- **bt-f-peor-caso**: Cada instancia de n elementos tiene todos productos no rompen ninguna resistencia, está formada por $r = \{1, \ldots, 1\}$ y $w = \{R, \ldots, R\}$ con $R > n$.
- **dinámica**: Esta familia de instancias tiene instancias con distintas combinaciones de valores para n y R en los intervalos $[1000, 8000]$ donde fijamos valores de n y R.

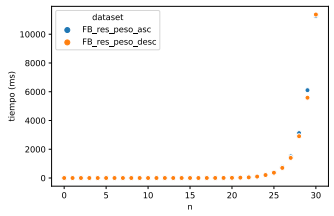
5.3. Experimento 1: Complejidad de Fuerza Bruta

En este experimento vamos a analizar la performance del método FB, la cual analizamos su complejidad previamente en la *Sección 2*, la cual teoricamente es $\theta(n*2^n)$. Para esto, analizamos distintos contextos, utilizando los datasets res_peso_desc y res_peso_asc, y graficamos el tiempo de ejecución de cada instancia en función de n. Luego tomamos una ejecución sobre el dataset res_peso_asc y contrastamos su complejidad contra una función $n * 2^n$. Por último, analizamos su coeficiente de correlación de Pearson a partir de la complejidad teórica del algoritmo y el tiempo que tarda en resolver.

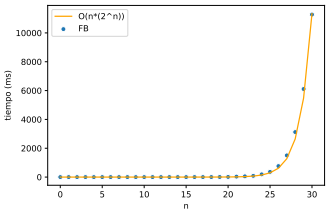
En la figura 2a se puede observar que ambas curvas se solapan en la mayoría de instancias. Por lo tanto, los tiempos de ejecución no parecen alterarse según el orden de las instancias, siguiendo la misma curva de crecimiento exponencial

En la figura 2b se puede observar el tiempo de ejecución de FB a la par de una función exponencial de $O(n * 2^n)$.

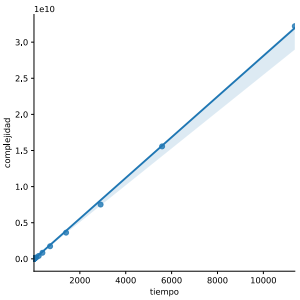
Por ultimo en la figura 2c tenemos el gráfico de correlación, donde para cada instancia se graficó su tiempo de ejecución real contra el esperado. Se puede observar que el tiempo de ejecución sigue una curva exponencial y que la correlación con la función $n^2 * 2^n$ es positiva y casi perfecta. En particular el índice de correlación de Pearson es $r \approx 0{,}999738$. Por lo tanto, podemos afirmar que el algoritmo se comporta como se describió inicialmente en las hipótesis.



(a) Tiempo de ejecución del método FB sobre entrada ordenada ascendente y descendente por resistencia/peso



(b) Tiempo de ejecución contra complejidad esperada.

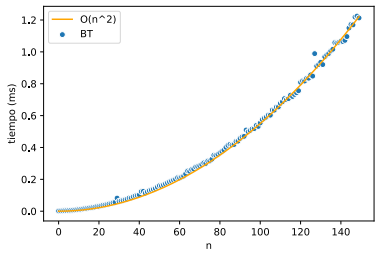


(c) Correlación entre el tiempo de ejecución y la complejidad esperada.

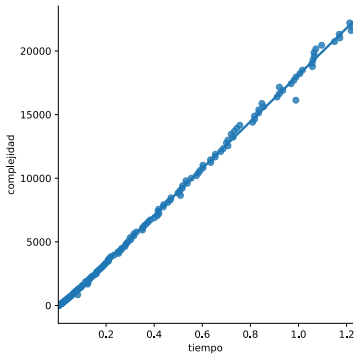
Figura 2: Análisis de complejidad del método FB.

5.4. Experimento 2: Complejidad de Backtracking

En esta experimentación vamos a contrastar las hipótesis de la *Sección 3* con respecto a las familias de instancias de mejor y peor caso para el Algoritmo 2, y su respectiva complejidad. Para esto evaluamos el método BT con respecto los datasets bt-mejor-caso y bt-peor-caso Las Figuras 3 y 4 muestran los gráficos de tiempo de ejecución de BT y de correlación para cada dataset respectivamente. En primer lugar, la hipótesis presentada sobre el mejor caso se cumple. Se puede apreciar que el tiempo de ejecución de cada instancia crece de manera cuadrática, haciendo la complejidad $O(n^2)$. Adicionalmente, se puede ver que el índice de correlación de Pearson es $r \approx 0{,}99931$, lo cual termina de confirmar nuestra primera hipótesis. Por otro lado, la complejidad en el peor caso, como planteamos en la *Sección 3*, debería ser al menos la misma del *Fuerza Bruta*, ya que el algoritmo de *Backtracking* toma el mencionado como base y le agrega podas. Pero, como enunciamos antes, esto no debería suceder en nuestro caso, ya que la implementación de las podas donde el mejor caso de una es el peor de la otra, evita que se llegue a la misma complejidad del *Algoritmo 1*. Al realizar la experimentación con entre 1000, 2000, ..., 10000 productos, vemos que crece de forma similar a una función de tipo 2^n. Esto nos dejaría una complejidad $O(2^n)$ en el peor caso. Creemos, sin embargo, que para observar mejor esto deberíamos tomar un n más grande. Esto lo podemos observar mejor en el índice de correlación de Pearson que nos dió $r \approx 0{,}96639$ en lugar de algo dentro de los $0{,}9$.

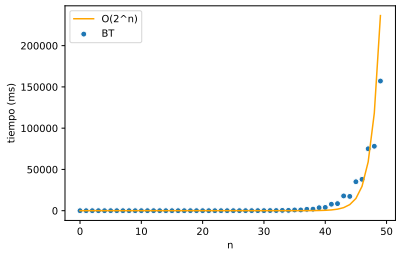


(a) Tiempo de ejecución contra complejidad esperada.

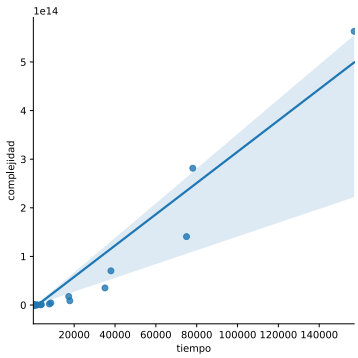


(b) Correlación entre el tiempo de ejecución y la complejidad esperada.

Figura 3: Análisis de complejidad en el mejor caso de BT.



(a) Tiempo de ejecución contra complejidad esperada.



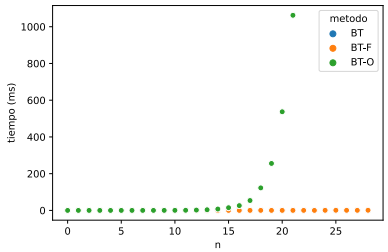
(b) Correlación entre el tiempo de ejecución y la complejidad esperada.

Figura 4: Análisis de complejidad en el peor caso de BT.

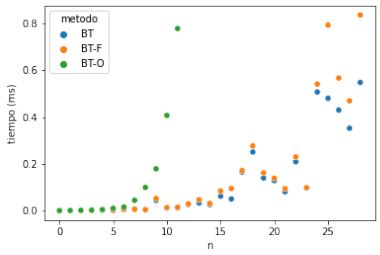
5.5. Experimento 3: Efectividad de las podas

A raíz del experimento anterior, nos surgió la duda de por qué nos queda en el mejor caso $O(n^2)$, y en el peor llegamos a $O(2^n)$. Qué factores afectan al algoritmo para que se genere esta diferencia. Una de las hipótesis es que el *Algoritmo 2* mejora su funcionamiento dependiendo de el orden de los productos teniendo en cuenta su relación $\frac{resistencia}{peso}$. Pensamos esto ya que, si ordenamos por mayor resistencia por peso podemos conseguir tener los elementos que se podarían primero por un lado, y los que se podarían últimos por otro.

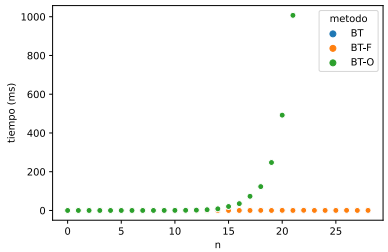
En este experimento comparamos el funcionamiento de los métodos BT, BT-F y BT-O con respecto a los datasets res_peso_asc res_peso_desc. La hipótesis es que para las instancias res_peso_asc los algoritmos van a ser más eficientes que para aquellas de res_peso_desc.



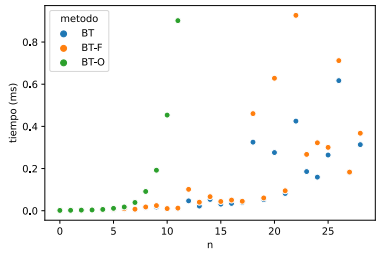
(a) Efectividad de las podas para res_peso_asc.



(b) Efectividad de las podas con zoom para res_peso_asc.



(c) Efectividad de las podas para res_peso_desc.

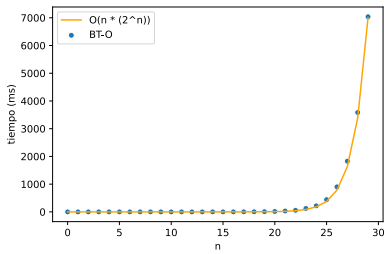


(d) Efectividad de las podas con zoom para res_peso_desc.

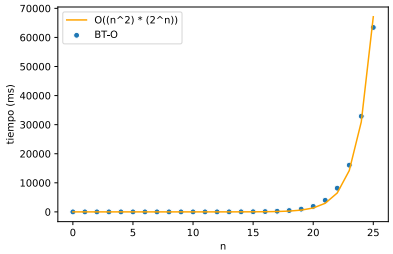
Figura 5: Comparación de efectividad en las podas.

En la *Figura 5* se muestra los resultados para el dataset res_peso_asc y res_peso_desc. Se ejecutaron instancias hasta $n = 29$ para evitar que el algoritmo demore mucho tiempo. Lo que pudimos ver con este experimento es que los tiempos de ejecución para los tres métodos se encuentran entre los observados anteriormente para los mejores y peores casos del algoritmo de BT. Por otra parte, pareciera que las podas son levemente más eficientes en el caso de las instancias con res_peso_desc. De todas formas, no vemos una diferencia significativa como para concluir nuestra hipótesis. Por esto, la rechazamos por el momento. Dado que hacer experimento con n mayores nos demanda muchísimo tiempo, quisimos probar la efectividad de las podas individualmente. Para esto, corrimos el *Algoritmo 2* utilizando únicamente una poda para concluir cuál es mejor en el peor caso y por cuánto.

Para esto utilizamos los datasets bt-o-peor-caso y bt-f-peor-caso, podemos observar los resultados en la *Figura 6*



(a) Tiempo de ejecución contra complejidad esperada BT-O.



(b) Tiempo de ejecución contra complejidad esperada BT-F.

Figura 6: Análisis de complejidad en el peor caso de BT-O y BT-F.

Como podemos observar, vemos que el peor caso con la poda de *optimalidad* es $O(n * 2^n)$ al igual que el *Algoritmo 1*. Esto es así, porque el peor caso es que no realice nunca a la po-

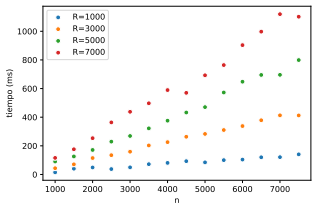
da, y como la poda contiene únicamente operaciones constantes, no le agrega complejidad al algoritmo.

Por otro lado, en el caso de la poda de *factibilidad*, observamos que el peor caso es $O(n^2 * 2^n)$. Esto es así, ya que en el peor caso, nunca realizaría la poda, pero tiene que evaluar la guarda, y al evaluar una función con complejidad $O(n)$, lo hace las veces que lo hacía en el *Algoritmo 1*, quedando complejidad dicha complejidad.

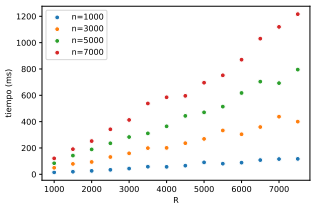
Empero, de acuerdo a los gráficos de la *Figura 5*, vemos que el algoritmo con poda de factibilidad se comporta mejor en general. Concluimos que el peor caso es peor, valga la redundancia, el de BT-F sin embargo, en la practica obtiene mejores resultados.

5.6. Experimento 4: Complejidad de Programación Dinámica

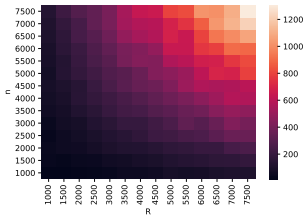
En este experimento analizamos la eficiencia del algoritmo de Programación Dinámica y la contrarrestamos contra la complejidad teórica analizada en la Sección 4. Para esto, ejecutamos las instancias del dataset dinamica sobre el metodo DP. Como queremos ver una cota de complejidad en función de dos variables n y R, recurrimos a primero analizar los tiempos de ejecución en función de n fijando R y luego viceversa, R en función de n. Esto se puede apreciar en las Figuras 7a y 7b, donde todas las lineas se ven con un crecimiento lineal. Por lo que observamos el crecimiento es similar en función de n y R. Se observa un pequeño ruido en los graficos que seguramente sea alguna interferencia del sistema operativo o cambio de contexto al momento de ejecutar el experimento. Por otro lado, observamos en la Figura 7c se grafica el crecimiento en función de ambas variables en simultaneo donde se aprecia nuevamente un crecimiento similar en función de ambas variables. Por ultimo, hicimos el gráfico de correlación, comparando para cada instancia el tiempo de ejecución contra el tiempo esperado. En la figura 7d podemos ver un índice positivo y casi perfecto, $r \approx 0{,}99439$, lo que termina de asegurar que la complejidad de este algoritmo es $\Theta(n * R)$



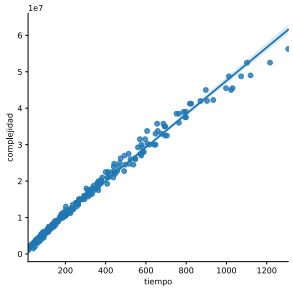
(a) Tiempo de ejecución en función de n.



(b) Tiempo de ejecución en función de R.



(c) Tiempo de ejecución en función de n y R.



(d) Correlación entre el tiempo de ejecución y la cota de complejidad temporal.

Figura 7: Resultados computacionales para el método DP sobre el dataset dinamica.