DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP1: Optimizando Jambotubos

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Rodriguez, Miguel	57/19	mmiguerodriguez@gmail.com
Itzcovitz, Ryan	169/19	ryanitzcovitz@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (+54 11) 4576-3300

https://exactas.uba.ar

Índice

1. Introducción

1.1. Resumen

En el presente trabajo vamos a desarrollar distintas soluciones algorítmicas a un problema de optimización para una cadena de supermercados. Se va a explicar como funcionan cada una de estas junto con su implementación especifica para este problema, para luego hacer una experimentación que nos permita compararlas y poder sacar conclusiones sobre las distintas soluciones al problema.

1.2. Introducción teórica

La cadena de supermercados Jambo quiere construir una serie de robots con el fin de optimizar el uso de bolsas de plástico. La propuesta del supermercado fue ofrecer un servicio de empacado de productos en los denominados Jambotubos, que tienen una resistencia máxima de peso en donde la idea es apilar la mayor cantidad de productos posibles sin romperlo.

Dado un Jambotubo con resistencia R y una lista de N productos S, cada uno con un peso asociado w_i y una resistencia asociada r_i, debemos determinar la máxima cantidad de productos que pueden apilarse en un tubo sin que ninguno esté aplastado. Tanto el Jambotubo como los productos tienen un peso limite de objetos que pueden tener encima. A modo de ejemplo, un Jambotubo con $R = 50$, $N = 5$, $w = [10, 20, 30, 10, 15]$ y $r = [45, 8, 15, 2, 30]$. La solución óptima es 3, y consiste en tomar los elementos 1, 3 y 4.

Para este tipo de problemas de optimización nos interesa utilizar técnicas algorítmicas estudiadas en la materia como Fuerza Bruta, Backtracking y Programación Dinámica, que tienen la capacidad de resolver este problema.

2. Desarrollo

En esta sección vamos a explicar la funcionalidad de los distintos algoritmos implementados junto con sus complejidades.

2.1. Fuerza Bruta

El primer algoritmo de búsqueda que vamos a detallar es el de fuerza bruta. Esta técnica se destaca por realizar una búsqueda exhaustiva para enumerar todos los posibles candidatos para resolver un problema y luego verificar si dicho candidato satisface la solución.

Para la solución del Jambotubo contamos con una lista de productos P (donde cada elemento es una tupla de <peso, resistencia>) y R como la resistencia del Jambotubo. Si tomamos como ejemplo: Sea $W = 30$ y $P = [< 10, 25 >, < 25, 8 >]$ utilizando fuerza bruta obtendremos como resultado una lista de todos los candidatos como posibles Jambotubos: $[[< 10, 25 >, < 25, 8 >], [< 10, 25 >], [< 25, 8 >], []]$. Luego el conjunto de soluciones factibles es $\{[< 10, 25 >], [< 25, 8 >], []\}$ ya que el conjunto $\{[< 10, 25 >, < 25, 8 >]\}$ no es correcto debido a que el peso acumulado es mayor a lo que el Jambotubo puede soportar. Para finalizar la resolución del problema obtenemos como resultado que 1 es la mayor cantidad de

productos que podemos meter en el Jambotubo sin que ninguno quede aplastado. Notemos que no hay una única solución posible en el conjunto de soluciones encontrada ya que ambas tienen un único producto, podemos utilizar cualquiera de las dos y el resultado será el mismo ya que queremos maximizar la cantidad de productos por Jambotubo.

El algoritmo utilizado para recolectar todas las posibles soluciones es a través de una recursión en donde vamos recorriendo toda la lista de productos dividiendo en dos posibles casos: Si el producto actual es agregado al Jambotubo o no. Por lo tanto, se generará un árbol en donde las hojas serán los posibles candidatos a resolver el problema en cuestión.

En conclusión el algoritmo se encarga de generar todos los candidatos posibles en base a los valores de entrada, luego verificar quienes son solución del problema y utilizar el resultado con mayor cantidad de productos.

Algorithm 1 Algoritmo de fuerza bruta

1: $i \in \mathbb{N}$ cantidad de productos vistos
2: $k \in <\mathbb{N}, \mathbb{N}>^n$ lista de productos <peso, resistencia> agregada a la solución
3: **function** $FB(i, k)$
4: **if** $i == n$ **then**
5: **return** $esJambotuboValido(k)$? $|k|$: -1 ▷ $\mathcal{O}(|k|)$
end
6: **return** $max(FB(i+1, k), FB(i+1, k \cup productos[i]))$

La complejidad del algoritmo de fuerza bruta es de $\mathcal{O}(n \times 2^n)$. Esto se debe a que a partir de la recursión vamos a dividir el problema en 2 hasta llegar al caso base, generando una árbol de n niveles y 2^n nodos. Como en todas las hojas del árbol verificamos que sea un Jambotubo válido agregamos más complejidad al algoritmo ya que esta función tiene complejidad $\mathcal{O}(n)$. Por lo que la complejidad final será $\mathcal{O}(n \times 2^n)$.

2.2. Backtracking

La técnica de backtracking funciona con una idea parecida a la de fuerza bruta de ir generando incrementalmente soluciones candidatas, pero tiene la particularidad de decidir cuando una rama no va a llegar a un resultado válido, y dejar de ver las soluciones que esta va a generar. Este proceso se llama poda, ya que estamos cortando ramas del árbol de recursión. En nuestra implementación particular, tenemos dos tipos de podas distintas.

Poda por factibilidad Para cada nodo o solución parcial que generamos, verificamos si es un Jambotubo válido, si lo es, continuamos con esa rama, caso contrario, detenemos la búsqueda. Por ejemplo, si tuviéramos un Jambotubo con $R = 20$, y una lista de productos $P = [< 21, 10 >, < 4, 15 >, < 15, 8 >]$. Un candidato posible es el conjunto $< 21, 10 >$, pero notemos que el peso del producto supera la resistencia del Jambotubo, por lo que no es una solución válida y todas las posibles soluciones que esta rama puede generar tampoco lo serán, por ende, la podamos.

Poda por optimalidad Por cada nodo, verificamos si la mejor solución parcial va a seguir

siendo mejor que cualquier solución que pueda generar la rama actual. Por ejemplo, si nuestra mejor solución hasta el momento tiene 3 elementos, y la cantidad de productos del nodo actual tiene 1 y como máximo 2 por agregar, no nos interesa ver esa rama, por lo que la podamos.

Algorithm 2 Algoritmo de backtracking

```
1: i ∈ ℕ cantidad de productos vistos
2: p ∈ < ℕ, ℕ >^n lista de productos <peso,resistencia> agregada a la solución parcial
3: K ∈ ℕ mejor solución encontrada hasta el momento
4: function BT(i, p)
5:     if !esJambotuboValido(p) then
6:         return −1                                   ▷ Poda por factibilidad
       end
7:     if K > |p| + n − 1 then
8:         return −1                                   ▷ Poda por optimalidad
       end
9:     if i == n then
10:        if !esJambotuboValido(p) then
11:            return −1                               ▷ O(|k|)
           end
12:        K = K > |p| ? K : |p|
13:        return |p|
       end
14:    return max(BT(i + 1, p), BT(i + 1, p ∪ productos[i]))
```

La complejidad del algoritmo de backtracking es de $\mathcal{O}(n^2 \times 2^n)$. Se puede observar que es muy similar a fuerza bruta, pero tenemos el agregado que realizamos ambas podas para no tener que recorrer casos innecesarios. Esto va a hacer que aumente la complejidad del peor caso. La poda por factibilidad utiliza la función esJambotuboVálido y al ser aplicada a todos los nodos y hojas, la complejidad es de $\mathcal{O}(n^2)$. La poda por optimalidad es $\mathcal{O}(1)$ ya que solo revisamos si en esa rama podemos llegar a encontrar una mejor solución que la actual. Como solo hay que agregarle la complejidad de factibilidad a lo visto en fuerza bruta nos queda en $\mathcal{O}(n^2 \times 2^n)$.

2.3. Programación Dinámica

La programación dinámica es un método de resolución de subproblemas superpuestos para reducir el tiempo de ejecución. La idea consiste en evitar recalcular valores ya calculados anteriormente guardándolos en memoria (memoización).

En el problema del Jambotubo utilizamos una matriz de $N \times R$ donde en cada posición tenemos el mejor resultado que se puede obtener para cierta resistencia del tubo y cantidad de elementos. Para calcular el último valor de la lista se va a recurrir a los valores vistos previamente que se encuentren en la matriz y en caso de no tenerlos se van a calcular. Por ejemplo si en la posición (i, j) de la matriz el valor es distinto a -1 entonces este va a

representar a la mayor cantidad de productos que se puedan poner en un Jambotubo de resistencia j con una cantidad de productos máxima i. La cantidad de productos la longitud de la sublista de productos de entrada desde la posición 1 a la i-esima. Por ende cada matriz de memoización generada por el algoritmo sera única para cada entrada distinta.

Algorithm 3 Algoritmo de programación dinámica

```
1: n ∈ ℕ cantidad de productos
2: r ∈ ℕ resistencia del Jambotubo
3: M ∈ ℕ^{n+1×r+1} matriz de memoización
4: i ∈ ℕ cantidad de productos vistos
5: w ∈ ℕ peso acumulado de los productos agregados
6: function BT(i, p)
7:     if w > r then
8:         return −1
       end
9:     if M[i][w] == −1 then
10:        if productos[i − 1].second < (r − w) || productos[i − 1].first > w then
11:            M[i][w] = PD(i − 1, w)
           else
12:            M[i][w] = max(PD(i − 1, w), PD(i − 1, w − productos[i − 1].first) + 1)
           end
       end
13:    return M[i][w]
```

La complejidad del algoritmo es $\mathcal{O}(N \times R)$. Esto se debe a que depende de la cantidad de estados que se van a resolver y el costo de resolver cada uno. Como máximo podemos llegar a resolver $N \times R$ subproblemas, y como cada problema tiene complejidad constante, en el peor de los casos deberíamos calcular todos los estados posibles.

3. Experimentación

3.1. Instancias

Para los experimentos de complejidad de cada algoritmo en particular la manera en la que generamos las instancias fue dado un seed construir instancias de manera incremental. Esto quiere decir que la i-ésima instancia será la misma que la (i-1)-ésima, pero con un nuevo producto generado aleatoriamente (dependiendo del seed). La resistencia del Jambotubo es un valor definido según la cantidad de productos de cada instancia.

Planteamos esta forma de construir las instancias porque creemos que es la mejor forma de ver como evoluciona un algoritmo cuando tenemos instancias más grandes. Al hacer esto y no utilizar instancias aleatorias, vamos a ser capaces de relacionar los resultados incrementales de las instancias de manera más precisa.

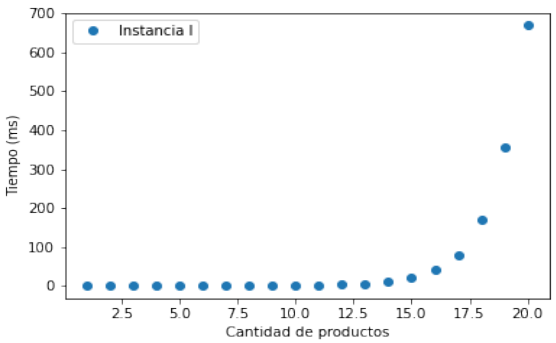
3.2. Complejidad de Fuerza Bruta

En el primer experimento vamos a analizar la complejidad del método de fuerza bruta para un conjunto de prueba. El conjunto de prueba utilizado fue generado aleatoriamente y de manera creciente como fue explicado anteriormente. Como detallamos en la sección de desarrollo, el algoritmo de fuerza bruta tiene el mismo tiempo de ejecución para el peor y mejor caso, ya que siempre vamos a generar todos los posibles candidatos y luego verificar cual es el mejor resultado que cumpla el invariante de ser un Jambotubo válido.

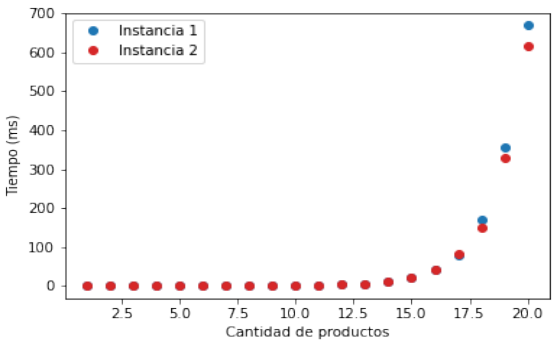
En la figura 1a ejecutamos una instancia donde todos los productos puedan entrar en el Jambotubo por lo que la solución será la cantidad de productos disponibles. Aquí observamos que aparenta ser una función con complejidad exponencial.

A continuación en la figura 1b hicimos otra ejecución, modificando la instancia a una generada aleatoriamente. La idea era observar que no importan los productos dentro de la instancia ya que el tiempo de ejecución será el mismo, por ser un método que explora todo el árbol de recursión.

La complejidad del algoritmo de fuerza bruta es $\mathcal{O}(n \times 2^n)$ tal como detallamos previamente en la sección de desarrollo. En la figura 2 podemos ver un gráfico de las instancias junto con la complejidad teórica $\mathcal{O}(n \times 2^n)$ y observamos que se asimilan con la misma.



(a) Complejidad instancia manual



(b) Complejidad instancia manual vs random

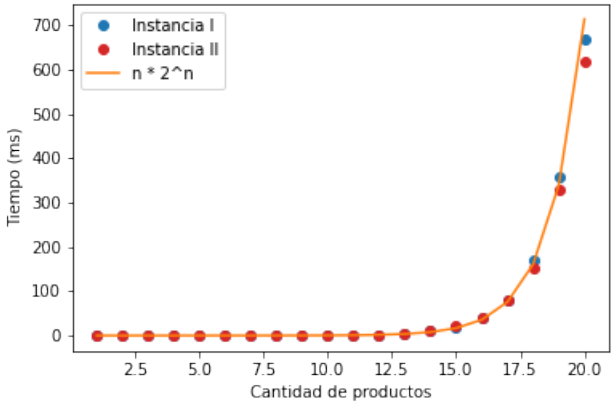
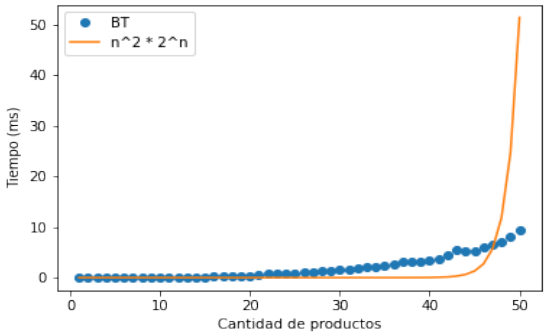


Figura 2: Complejidad esperada vs instancias

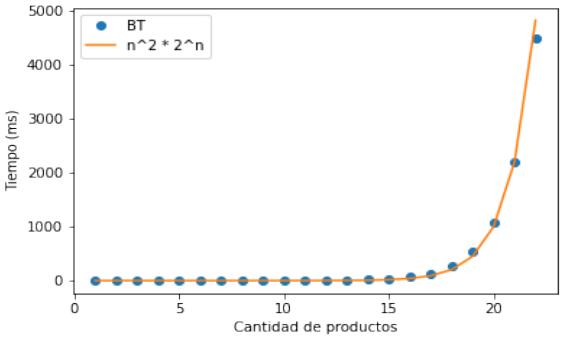
3.3. Complejidad de Backtracking

En este experimento vamos a detallar como varía la complejidad para el algoritmo de backtracking. Como explicamos, este algoritmo tiene en peor caso una complejidad de $\mathcal{O}(n^2 \times 2^n)$, pero a diferencia del algoritmo de fuerza bruta, su mejor caso no tiene la misma complejidad.

La propuesta para este experimento es generar instancias que se puedan resolver en menor tiempo aprovechando las podas, esto lo hacemos agregando productos que, cuando estamos generando soluciones posibles, superen la resistencia total del Jambotubo de manera frecuente, por lo que la poda de factibilidad pueda cortar esa rama, evitar casos que no darían posibles candidatos y optimizar el tiempo de ejecución. Este comportamiento se puede ver en la figura 3a en donde podemos observar que la complejidad es mucho mejor que la esperada en el peor caso.



(a) Complejidad instancia fácil



(b) Complejidad instancia difícil

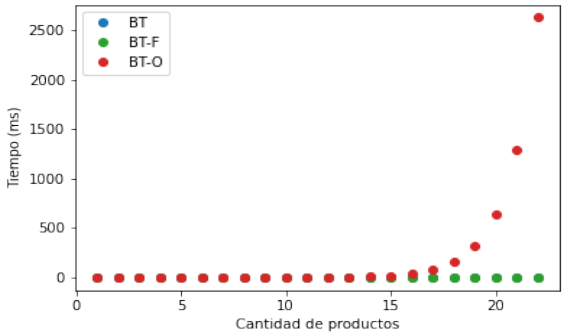
En cambio, para generar instancias que hagan que el algoritmo ejecute el peor caso, es dándole la posibilidad de siempre agregar algún producto al Jambotubo de manera tal que los productos tengan una alta resistencia y bajo peso, para poder seguir generando el árbol de candidatos al igual que lo haría el algoritmo de fuerza bruta. Esto lo podemos observar en la figura 3b en donde construimos una instancia donde la solución del problema sea la última rama del árbol, por lo que deberá recorrerlo todo hasta encontrar la solución esperada. Esto causa que el tiempo de ejecución aumente y que la complejidad se corresponda con el peor caso.

En conclusión en este experimento demostramos que la complejidad de backtracking es variable dependiendo de la instancia con la que se ejecute, a diferencia de fuerza bruta que siempre tiene la misma complejidad.

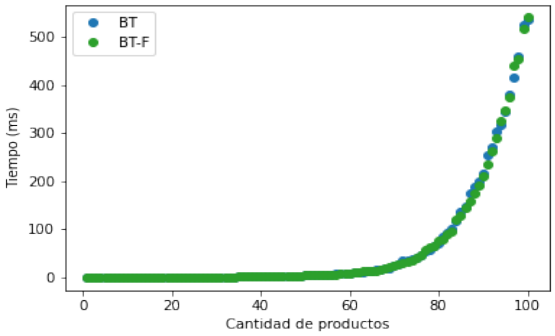
3.4. Backtracking: Experimentación de las podas

En el siguiente experimento nos interesa ver como cambian las complejidades del algoritmo de backtracking según las podas que estemos aplicando. En particular, el algoritmo utiliza ambas podas, pero podemos ejecutarlo de manera tal que utilice únicamente alguna de las dos desarrolladas (optimalidad o factibilidad).

En el problema del Jambotubo notamos que al hacer backtracking la mayor cantidad de veces que se podaba una rama era por factibilidad y esto se debe a que es mas probable cortar un candidato porque no es un Jambotubo válido a que los posibles productos restantes que se puedan agregar no superen la mejor solución obtenida previamente.



(a) Complejidad backtracking y ambas podas



(b) Complejidad backtracking vs factibilidad

A continuación observamos en la figura 4a un gráfico donde analizamos cuanto tiempo tarda backtracking con ambas podas por separado. Podemos ver que utilizando únicamente optimalidad los tiempos son similares a fuerza bruta y que factibilidad es muy similar a utilizar ambas podas.

En la figura 4b detallamos más la similitud entre backtracking con todas las podas y utilizando únicamente factibilidad. Con esto podemos concluir que en la mayoría de los casos se usa la poda por factibilidad y que la de optimalidad no aporta mucho en términos de optimización.

3.5. Complejidad de Programación Dinámica

En el siguiente experimento vamos a ejecutar diferentes instancias aleatorias y graficar el tiempo de ejecución para ver que se cumple la complejidad teórica de $\mathcal{O}(N \times R)$ ya que en el peor caso va a recorrer todos los elementos de la matriz de $N \times R$ con el fin de encontrar la mejor solución para el problema.

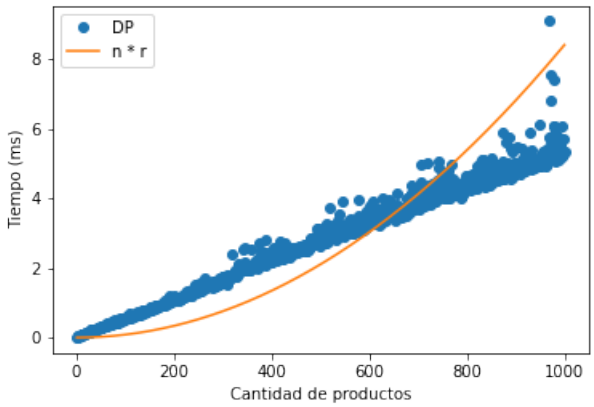


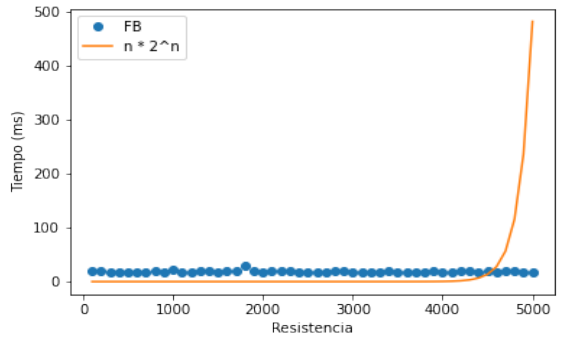
Figura 5: Complejidad de programación dinámica

La figura 5 muestra la comparación entre la complejidad esperada del algoritmo con la obtenida luego de la experimentación, observando el crecimiento del tiempo de ejecución en función de las variables N (cantidad de productos) y R (resistencia total de Jambotubo). Esta figura se corresponde con el resultado esperado.

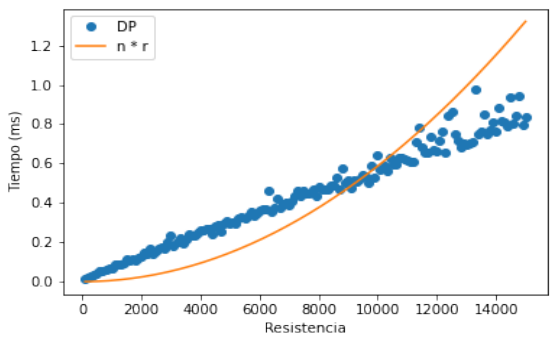
3.6. Programación Dinámica vs Fuerza Bruta

Como último experimento para presentar vamos a cambiar el enfoque que veníamos teniendo, en vez de variar el N (cantidad de productos) vamos a variar la resistencia del Jambotubo. Al observar que la complejidad de programación dinámica depende del R, queremos comparar como actúan los algoritmos de FB y PD.

Para generar las instancias lo que planteamos es utilizar un numero $N = 15$ de productos fijo e ir incrementando la resistencia R en cada nueva instancia. Nuestra hipótesis de lo que va a suceder es que la complejidad del algoritmo de fuerza bruta será siempre constante ya que siempre recorre todo el árbol de recursión (que siempre será el mismo ya que la cantidad de productos es la misma), mientras que para el algoritmo de programación dinámica la complejidad comenzará a aumentar a medida que se incremente el R hasta que llegue el punto donde sea mas lento que el de fuerza bruta..



(a) Complejidad de FB con N fijo y R variable



(b) Complejidad de PD con N fijo y R variable

En la figura 6a podemos verificar nuestra hipótesis sobre la complejidad constante de fuerza bruta para instancias con N fijo y R variable ya que no hay crecimiento en el tiempo.

En la figura 6b podemos ver como a medida que crece el R también aumenta el tiempo de ejecución, por lo que confirmamos que el algoritmo de programación dinámica depende de la resistencia.

Por ultimo, en la figura 7 observamos que luego de cierto R (aproximadamente 400000) al algoritmo de fuerza bruta le toma menos tiempo terminar que al de programación dinámica.

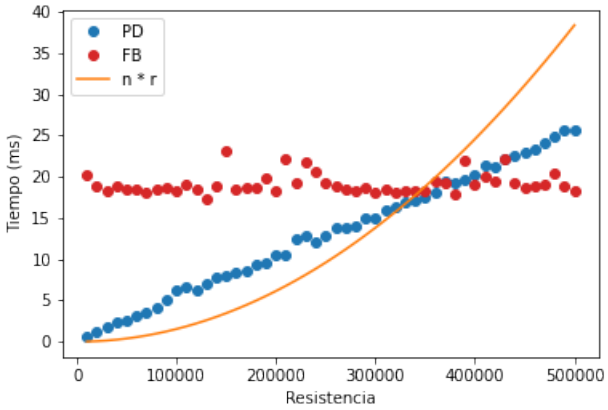


Figura 7: Comparación entre PD y FB

En conclusión, pudimos ver que al depender de la resistencia del Jambotubo la complejidad del algoritmo de programación dinámica es más lento que el de fuerza bruta para distintas instancias. En general, si tomamos $R >> N$ puede ocurrir que utilizar un algoritmo de fuerza bruta o backtracking sea lo óptimo en términos de complejidad, siempre y cuando tengamos con un N chico.

4. Conclusiones

A lo largo de los experimentos realizados pudimos analizar diferentes características de los algoritmos implementados para el problema particular del Jambotubo y sus productos. En particular, pudimos notar distintas características que tienen las podas en el algoritmo de backtracking y la eficiencia en términos de tiempos que nos da el algoritmo de programación dinámica, aunque también debemos tener en cuenta que este método puede ser superado por otros cuando se utilizan resistencias muy altas como parámetros de entrada.