DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico I

Jambo-tubo

Algoritmos y estrcturas de datos III

Primer Cuatrimestre 2020

Integrante	LU	Correo electrónico
Ilan Olkies	250/17	ilanolkies@gmail.com
Joaquín Naftaly	816/17	joaquin.naftaly@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

Se desea desarrollar un robot con un algoritmo que maximice la cantidad de productos que se pueden apilar en un *Jambo-tubo*, cumpliendo con dos requisitos:

- La suma de los pesos de los productos en el *Jambo-tubo* no puede superar la resistencia del mismo.
- La suma de los pesos sobre cada producto no pueden exceder la resistencia del mismo

Este trabajo propone un modelo matemático para representar el problema y tres algoritmos distintos para resolverlo. Las técnicas algorítmicas utilizadas son *Fuerza Bruta*, *Backtracking* y *Programación Dinámica*. El análisis de los algoritmos esta dividido en tres secciones principales: la primera con una presentación teórica del problema a resolver, la segunda sección desarrolla cada algoritmo a evaluar y presenta demostraciones de correctitud, y la última contiene un análisis empírico que muestra cual es el mejor algoritmo.

1.1. Modelo matemático

Dada una resistencia máxima $R \in \mathbb{N}$ y una secuencia ordenada S de n productos donde $S_i = (w_i, r_i)$ con $w_i \in \mathbb{N}$ el peso y $r_i \in \mathbb{N}$ la resistencia del i-ésimo producto con $i \in [1, \ldots, n]$, se desea encontrar **la subsecuencia más larga** $S' = \{(w'_i, r'_i) \in S\}$, con largo n', manteniendo el orden de S, tal que

1. $\sum_{i=1}^{n'} w'_i \leq R$ (no se excede la resistencia máxima)
2. $\sum_{j=i+1}^{n'} w'_j <= r'_i \; \forall \; i = 1 \ldots n'$ (no se excede la resistencia de cada producto)

1.1.1. Ejemplos

- Si $S = \{(10, 14), (11, 14), (12, 13), (13, 12)\}$ y $R = 5$, entonces la solución es 0 ya que $R < w_i$ para todo $i = i \ldots n$.
- Si $S = \{(2, 5), (6, 5), (5, 2)\}$ y $R = 10$, entonces la solución óptima es 2 y es posible apilando los productos 1 y 3. Notar que, por ejemplo, apilar todos los productos no es una solución válida ya que $r_1 < w_2$.

1.1.2. Dos proposiciones importantes

Dada una secuencia de productos S y una subsecuencia S' de S con n' productos, cada producto $i = 1 \ldots n'$ tiene una resistencia restante $h_{S',i} = r_i - \sum_{j=i+1}^{n'} w_i$. La resistencia restante de la subsecuencia vacía es R.

Lema 1 *Es trival ver que si, al menos para un producto de S', la resistencia restante es negativa entonces la subsecuencia es inválida. Es decir que si $\exists i = 1 \ldots n' : h_{S',i} < 0$, entonces la subsecuencia es inválida.*

Lema 2 *Se puede demostrar por inducción que si $\hat{S}'$ es agregarle un producto al final de la subsecuencia S' de largo n', $\hat{S}' = S' \cup \{(\hat{w}, \hat{r})\}$, la mínima resistencia restante de los productos de $\hat{S}'$ es el mínimo entre la*

resistencia $\hat{r}$ y la resistencia restante mínima de los S' menos el peso $\hat{w}$. Es decir,

$\min_{i=1\ldots\hat{n'}} \{h_{\hat{S'},i}\} = \min \{\hat{r}, \min_{i=1\ldots n'} \{h_{S',i}\} - \hat{w}\}$.

2. Los algoritmos

2.1. Fuerza Bruta

La técnica consiste en generar todas las posibles instancias para el problema y determinar cual es la óptima. Aplicada al problema de los *Jambo-tubos*, consiste en enumerar todas las subsecuencias posibles, validar su factibilidad, calcular la cantidad de elementos acumulados y tomar la máxima cantidad.

Por ejemplo, se tiene una la cinta con 3 productos $\{p_1 = (2, 5), p_2 = (6, 5), p_3 = (5, 2)\}$ para poner en un *Jambo-tubo* de resistencia inicial $R = 10$, el conjunto de posibles soluciones se describe en la Figura 1.

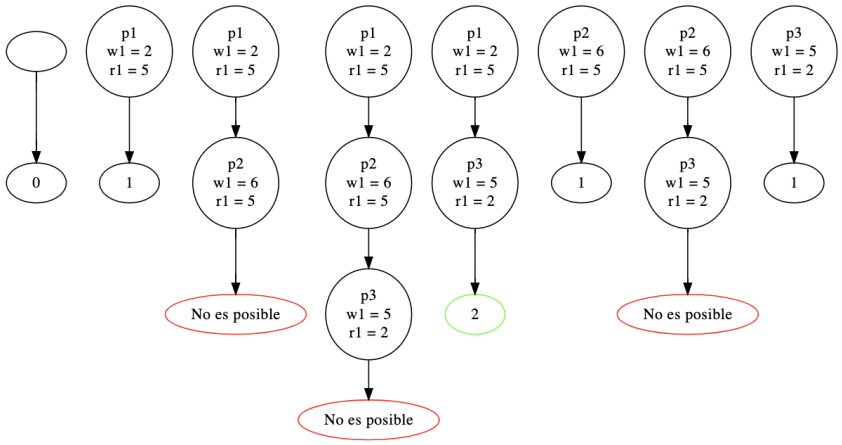


Figura 1: Enumeración de subsecuencias de $\{p_1 = (2, 5), p_2 = (6, 5), p_3 = (5, 2)\}$. La verde es la solución óptima.

El Algoritmo 1 genera todas las instancias recursivamente y se queda con la óptima.

Algorithm 1 Algoritmo de Fuerza Bruta.

```
1: function fb(h, w, r, i, c)
2:     if i = n + 1 then                                   ▷ Caso base
3:         if h ≥ 0 then return c                          ▷ Hoja válida
4:         return 0                                        ▷ Hoja inválida
5:     return máx{
6:         fb(h, w, r, i + 1, c),                          ▷ No agrego w_i
7:         fb(mín{h − w_i, r_i}, w, r, i + 1, c + 1)       ▷ Agrego w_i
8:     }
```

El i-ésimo paso inductivo se construye llamando al algoritmo con la instancia que incluye al i-ésimo producto y la instancia que no lo incluye. El máximo de esas dos llamadas es la solución óptima para el subproblema de los primeros i productos. Notar que c es la solución parcial del problema en esa rama. La correctitud del paso inductivo se puede probar basandose en el Lema 2.

En la Figura 2 se puede notar que el caso base es alcanzado cuando $i = n + 1$, es decir que esa rama del problema ya fue evaluada.

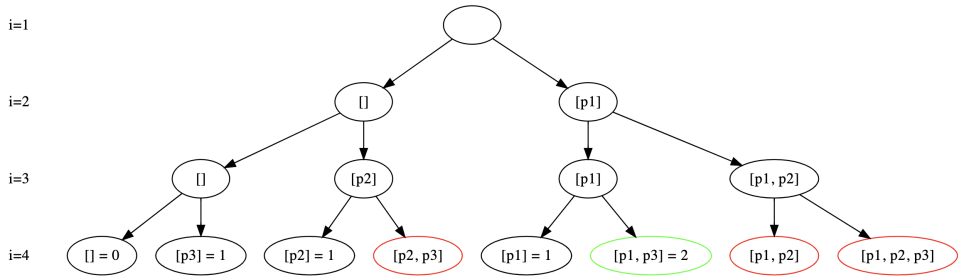


Figura 2: Árbol de recursión del Algoritmo 1 para la instancia $\{p_1 = (2, 5), p_2 = (6, 5), p_3 = (5, 2)\}$ con $R = 10$. En verde la solución óptima y en rojo las soluciones inválidas.

La solución al problema original es $fb(R, w, r, 1, 0)$. El hecho de que se generen todas las posibles soluciones asegura la **correctitud** del algoritmo ya que, de existir una solución óptima, debe estar en el conjunto generado.

Si $n \in \mathbf{N}$ es la cantidad de productos en la cinta, el árbol de recursión generado por el Algoritmo 1 es un árbol binario completo de $n + 1$ niveles y 2^n hojas (Figura 2). El tiempo de ejecución de cada nodo es constante, entonces la complejidad del algoritmo es $\Theta(2^n)$. Dado cualquier tipo de instancia, el algoritmo siempre va a generar un árbol binario completo, por eso podemos concluir que todas las instancias se van a comportar de la misma manera.

2.2. Backtracking

La técnica es similar a la de fuerza bruta, recursiva. Esta técnica intenta podar (eliminar) ramas del árbol de recursión que al momento de ejecución parcial ya se sabe que no serán la solución del problema. Existen dos tipos de poda:

- **Poda por factibilidad:** Si al agregar el i-ésimo producto, el peso supera la mínima resistencia restante de los productos ya incluidos en el tubo, entonces la instancia parcial de incluirlo no es factible y, además, ninguna instancia parcial que acumule más productos a esta lo será.
- **Poda por optimalidad:** Se almacena la solución parcial óptima encontrada k. Si al agregar el i-ésimo producto, la cantidad ya agregada c sumada a la cantidad $n - i$ de productos que potencialmente podrían ser agregados no supera a k, entonces tampoco lo superará cualquier otra instancia construida a partir de esta.

El funcionamiento de las podas para la secuencia de productos $\{p_1 = (2, 5), p_2 = (6, 5), p_3 = (5, 2), p_4 = (1, 1)\}$ y $R = 10$ se puede visualizar en la Figura 3.

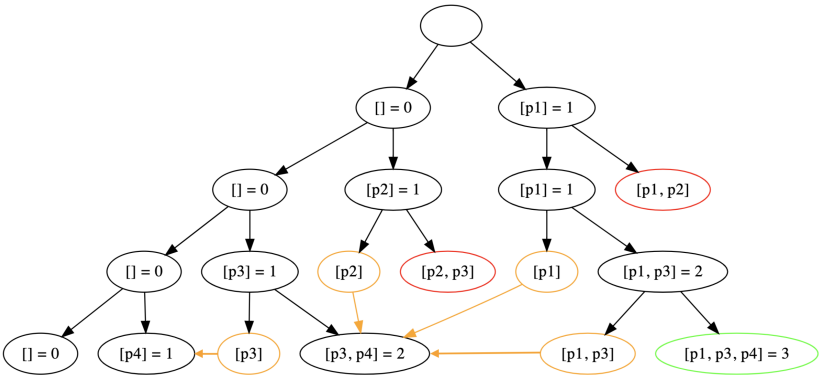


Figura 3: En rojo las podas por factibilidad y en naranja las podas por optimalidad asociadas a la instancia parcial óptima por la que fue podada.

Algorithm 2 Algoritmo de Backtracking.

```
1:  k ← 0                                                  ▷ Solución parcial óptima
2:  function bt(h, w, r, i, c)
3:      if h < 0 then return 0                             ▷ Poda por factibilidad
4:      if c + n − i < k then return 0                     ▷ Poda por optimalidad
5:      if i = n + 1 then return c                         ▷ Caso base de instancia válida
6:      if c ≥ k then k ← c                                ▷ Nueva solución parcial óptima
7:      return máx{
8:          bt(h, w, r, i + 1, c),
9:          bt(mín{h − w_i, r_i}, w, r, i + 1, c + 1)
10:     }
```

La complejidad del algoritmo en el peor caso teórico es $\mathcal{O}(2^n)$ ya que, en ese caso, no se realiza ninguna poda. Es decir que se generan las mismas instancias y el mismo árbol de recursión que en *Fuerza Bruta*. Notar que las líneas de código agregadas para las podas y la actualización del k se ejecutan en tiempo constante.

El tipo de instancias que le *molesta* a la poda de *factibilidad* son las que tienen todas las subsecuencias factibles. Por el contrario, el caso más efectivo se da cuando ninguna rama es factible, osea que $w_i > R \; \forall \; i = 1 \ldots n$. El peor caso para la poda de *optimalidad* es que ningún producto pueda ser agregado, el k no se actualiza y no genera podas. El mejor caso es encontrar el caso óptimo temprano.

2.3. Programación Dinámica

La técnica de *Programación Dinámica* consiste en dividir el problema en subproblemas más pequeños que son más fáciles de resolver. Luego, se combinan esos resultados para generar la solución del problema original. Esta técnica evita repetir llamadas recursivas almacenando los resultados que ya han sido calculados para poder usarlos después. En otras palabras, no se resuelve más de una vez el mismo subproblema.

Para aplicar el algoritmo, el problema debe cumplir con el *Principio de optimalidad de Bellman*: la so-

lución óptima para una secuencia de productos esta compuesta, por las subsoluciones óptimas de agregar o no agregar el primer producto.

Sea f la función recursiva que resuelve el problema. $f(h, i)$ es la máxima cantidad de productos $S^i = \{p_i, \ldots, p_n\} \subset S$ que se pueden ubicar en un *Jambo-tubo* de resistencia restante h. $f(R, 1)$ es la solución del problema. Formalmente,

$$f(h,i) = \begin{cases} -\infty & \text{si } h < 0, \\ 0 & \text{si } i = n+1, \\ \max\{f(h, i+1), 1 + f(\min\{h - w_i, r_i\}, i+1)\} & \text{caso contrario.} \end{cases} \tag{1}$$

Esta función representa correctamente a nuestro problema ya que:

(i) Si $h < 0$ ningún producto va a ser agregado, la respuesta es $f(h, i) = -\infty$.

(ii) Si $i = n + 1$ entonces la secuencia de productos es vacía, la respuesta es $f(h, i) = 0$.

(iii) Si $i < n$ y $h \geq 0$, $f(h, i)$ es la solución óptima de S^i y está compuesta por el máximo entre la solución óptima de agregar p_i y la de no agregarlo.

Algorithm 3 Algoritmo de Programación Dinámica.

```
1: M_ih ← ⊥ for i ∈ [1, n], h ∈ [0, R]                ▷ Matriz de n * R + 1
2: function pd(h, i)
3:     if h < 0 then return −∞                          ▷ Instancia inválida
4:     if i = n + 1 then return 0                       ▷ Caso base de instancia válida
5:     if M_ih = ⊥ then M_ih ← máx{                     ▷ Memoización
6:             pd(h, i + 1),
7:             1 + pd(mín{h − w_i, r_i}, i + 1)
8:         }
9:     return M_ih
```

Se puede observar que los posibles valores para h e i son $h = 0 \ldots R$ e $i = 1 \ldots n$, es decir $\Theta(n \times R)$ combinaciones posibles para h e i. Si se agrega una memoria en la que se guardan las llamadas ya resueltas y sus resultados, a lo sumo se resuelven $n \times R$ llamados. Ese proceso se denomina *memoización* y se puede ver en la línea 5 del Algoritmo 3. Si el problema (h, i) ya fue resuelto, se retorna el resultado previamente calculado y guardado en la memoria. En caso de que no esté definido previamente, se calcula recursivamente.

Al usar una matriz como memoria, la lectura y escritura se realiza en tiempo constante. Como se resuelven a lo sumo $n \times R$ casos y, los casos ya resueltos toman tiempo constante, entonces la complejidad del algoritmo es $\mathcal{O}(n \times R)$. Más aún, notar que la inicialización de la matriz en la línea 1 se realiza en $\Theta(n \times R)$, entonces se puede concluir que la complejidad del algoritmo es $\Theta(n \times R)$.

3. Experimentación

En esta sección se experimentará con distintas instancias representativas del problema con el objetivo de establecer comparaciones entre los algoritmos y demostraciones empíricas de la complejidad de los mismos. Los experimentos tienen un formato común: primero se plantea una breve descripción del objetivo del experimento, luego las instancias que representan al problema y por ultimo se desarrolla una discusión y exposición gráfica de los resultados. Todos los experimentos se pueden ver detallados en el repositorio del trabajo.

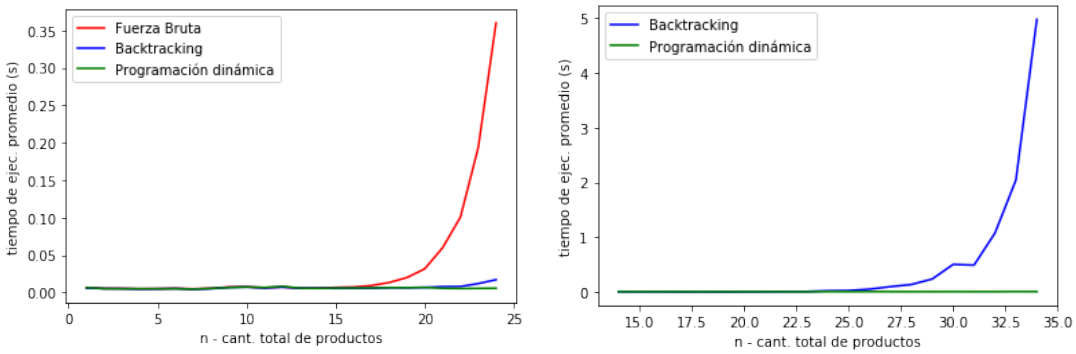
3.1. Comparación de algoritmos

El primer experimento es una simple *competencia* entre los algoritmos. Vamos a ver como se comportan para las mismas instancias aleatorias y comparar el tiempo de ejecución.

Para esto generamos dos sets de instancias con la siguiente distribución:

- R aleatorio entro 500 y 1000
- Los w_i aleatorios entre 800 y 1000
- Los r_i aleatorios entre 1 y 100

Esta distribución nos garantiza que los resultados serán cercanos a n, es decir que los algoritmos de fuerza bruta y backtracking no terminarán rápido. Además la dispersión de los w_i y r_i permiten ver casos donde se efectúen podas y aciertos en la matriz de dinámica.



(a) Comparación de tiempo de ejecución entre los 3 algoritmos

(b) Comparación de tiempo de ejecución entre *backtracking* y *programación dinámica*

Figura 4

Primero comparamos los tres algoritmos. Tomamos un conjunto de instancias de esta clase variando n entre 1 y 25, promediando 100 ejecuciones para cada n.

Notamos en la Figura 4a, como a simple vista fuerza bruta crece más rápido que los otros dos. Incluso para muestras más grandes el tiempo del algoritmo se vuelve inviable.

Ahora, para hacer una comparación mas detallada de programación dinámica y backtracking hacemos 10 repeticiones de cada n de 15 a 35. Podemos ver, en la Figura 4b, como a partir de cierto n el algoritmo de backtracking empieza a tener un peor comportamiento.

3.2. Complejidad del algoritmo de *Fuerza Bruta*

Como vimos previamente, el algoritmo de fuerza bruta simplemente lista todas las instancias posibles y retorna la mejor de ellas. El algoritmo hace la misma cantidad de operaciones para el mismo n, por eso dijimos que tenia complejidad $\Theta(2^n)$. Veamsolo:

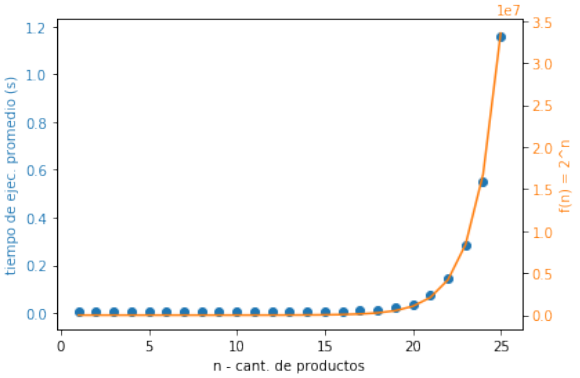


Figura 5: Comparación entre el tiempo de ejecución de fuerza bruta y una función $f(x) = 2^x$

Para este experimento usamos el mismo tipo de instancias que en el experimento anterior. Sin embargo para este caso particular, el tipo de instancias que usemos no tendría relevancia.

Podemos ver en azul el tiempo de ejecución promedio entre 100 instancias para cada n de 1 a 25. Vemos que la función $f(x) = 2^x$ en naranja, y el tiempo de ejecución son idénticos, entonces efectivamente la complejidad de nuestro algoritmo es $\Theta(2^n)$.

Sumado a esto, calculamos el coeficiente de correlación de Pearson, que nos dice cual es la correlación de las dos variables *más allá de su escala*. Cuando las variables son similares los resultados son cercanos a 1, el resultado de nuestro cálculo fue $\sim 0{,}999697682$. Esto comprueba nuestro análisis teórico de la complejidad del algoritmo.

3.3. Efectividad de las podas de backtracking

La idea de este experimento es ver en que casos son efectivas las distintas podas del algoritmo de *Backtracking*. Para esto ejecutaremos las mismas instancias usando solo la poda de factibilidad, solo la de optimalidad y ambas.

Llamamos instancias *más pesadas* a las que, para los mismos rangos de r, los productos tienen mayor peso. El contrario serían las instancias *livianas*. Como vimos previamente, el mejor caso para la poda de *factibilidad* es que no entre ningún producto y coincide con el peor caso de *optimilidad*, así que suponemos que van a funcionar mejor las familias de instancias más *pesadas* para *factibilidad*. En cambio, el peor caso para *factibilidad* es que se puedan agregar todos los productos. Podemos deducir que para las instancias más *livianas* va a funcionar mejor *optimalidad* ya que va a realizar muchas más podas.

Las instancias utilizadas en este experimento sólo varían el peso w_i. Los r_i y R se mantienen. Se generaron las instancias *pesados, no tan pesados, no tan livianos y livianos*. Nuestra deducción parece ser correcta al ver la Figura 6, en la que vemos como conjuntos mas pesados hacen que factibilidad se comporote mejor y por el contrario los más livianos ayudan a optimalidad.

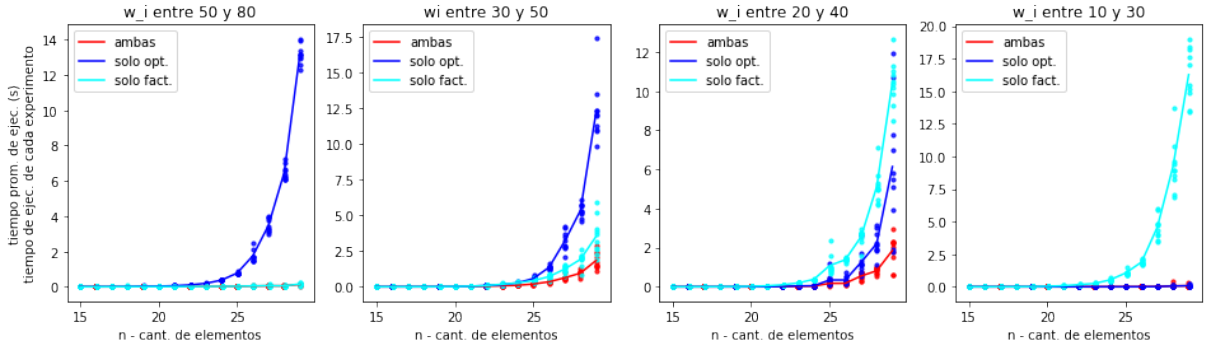


Figura 6: Podas de BT con las instancias **pesados, no tan pesados, no tan livianos y livianos** respectivamente.

3.4. Orden de productos vs Backtracking con distintas podas

Este experimento intenta mostrar como se comporta el algoritmo de *backtracking* según el ordenamiento de los productos. Para esto generamos instancias **livianas y pesadas** como en el experimento anterior pero ordenándolas por peso, de forma creciente y decrecientemente. Luego ejecutamos las 3 variantes de los algoritmos para ambos conjuntos.

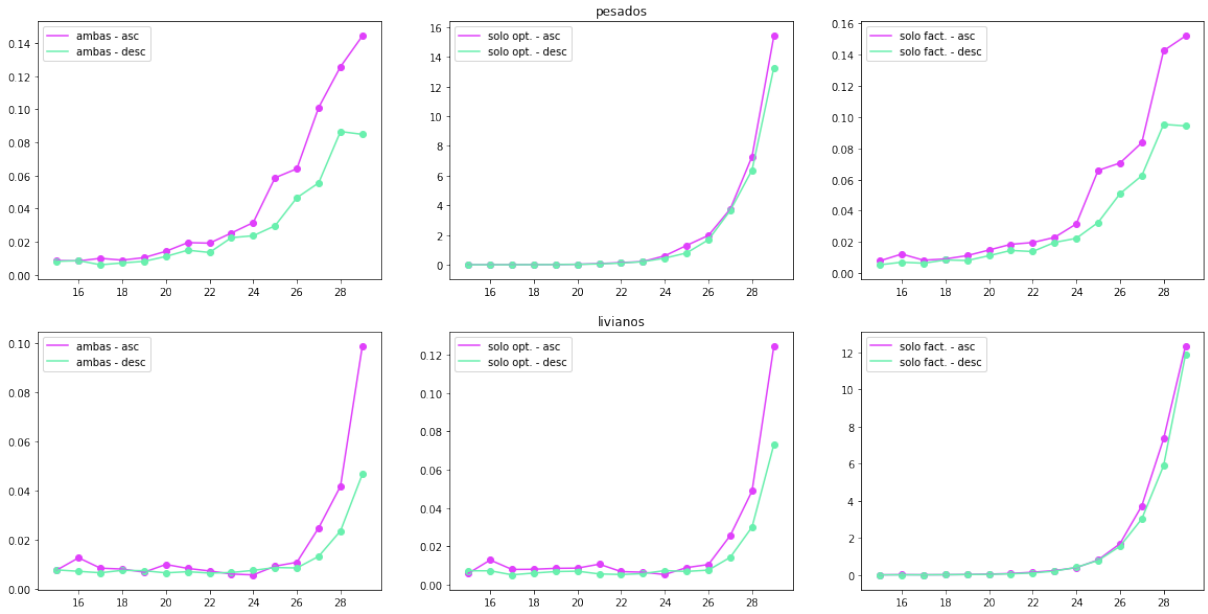


Figura 7: Tiempos de ejecución para las distintas variantes de podas con los productos ordenados creciente y decrecientemente.

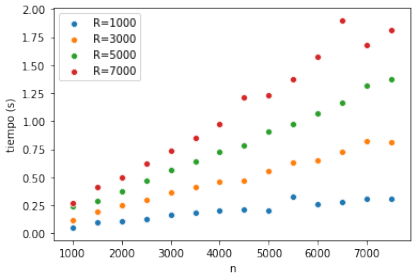
Luego de experimentar iteradas veces, incluso con conjuntos distintos, notamos que el comportamiento del algoritmo no varia mucho dependiendo del ordenamiento. En la Figura 7 parecería que el ordenamiento decreciente muestra mejores comportamientos, pero no podemos basarnos en este experimento para mostrar un caso general.

3.5. Complejidad de *Programación Dinámica*

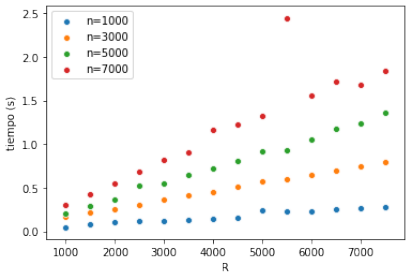
Se analiza empíricamente la complejidad del algoritmo y luego se compara con la complejidad teórica propuesta. Generaremos el *dataset* con los siguientes parámetros:

- n y R toman todos los valores múltiplos de 500 en el rango entre 1000 y 8000.
- Los w_i aleatorios entre 1 y 40
- Los r_i aleatorios entre 1 y 40

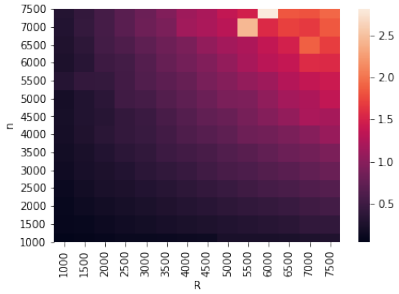
Esto nos permitirá ver como se comporta el algoritmo cuando crecen n y R.



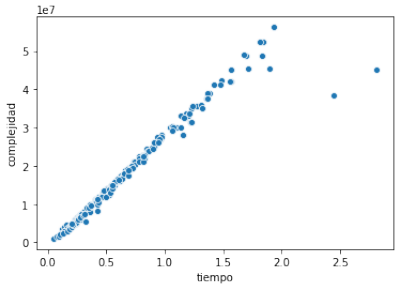
(a) Tiempo de ejec. en función de n.



(b) Tiempode ejec. en función de R.



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



(d) Correlación de tiempo de ejecución y esperado.

Figura 8

En las Figuras 8a y 8b podemos observar el tiempo de ejecución variando los parámetros n y R. Esto nos permite ver claramente que la complejidad es directamente proporcional a estos dos parámetros. En la Figura 8c, se puede ver con más detalle como el tiempo de ejecución crece linealmente para n y R con la misma proporción.

La Figura 8d nos ayuda a revalidar nuestra hipótesis. Este gráfico es particular, gráficamos el tiempo de ejecución teório $n \times R$ para cada tiempo distinto de ejecución que obtuvimos. Podemos ver claramente la correlación entre ambos. Sumado a esto, el coeficiente de correlación de Pearson para estas variables nos da $\sim 0{,}9696327$, eso quiere decir que los tiempos de ejecución resultantes son muy similares a $n \times R$.

4. Conclusiones

De los tres algoritmos que desarrollamos y experimentamos, podemos concluir que el de *Fuerza Bruta* no es eficiente ya que sin importar el tipo de instancia el tiempo de ejecución va a crecer exponencialmente con n. En cambio, el de *Backtracking* puede lograr resultados mejores gracias a las podas, llegando

inclusive a una complejidad lineal en los mejores casos. Esto va a depender fuertemente de que tipo de instancia reciba. El algoritmo de *Programación Dinámica* funciona en tiempo lineal para cualquier n y R. Evidentemente en razón del tiempo de ejecución nuestro robot debe usar el algoritmo de *Programación Dinámica*.

El llamado recursivo es igual para los tres algoritmos, entonces la cantidad de memoria ocupada por cada llamado en sí es la misma. Entonces la cantidad de memoria utilizada por los algoritmos dependerá de los otros aspectos. Por ejemplo, para introducir las mejoras del algoritmo de *Backtracking* solo necesitamos almacenar un entero. En cambio, para usar el algoritmo de *Programación Dinámica* la matriz ocupará siempre $\Theta(n \times R)$ sin importar cual sea el caso. Esto sugiere que la elección del algoritmo de programación dinámica dependa de el tamaño de R.

En conclusión, no podemos basarnos solo en el aspecto temporal de los algoritmos para evaluar cual es el más efectivo. Para casos en los que n es chico y R es grande necesitaremos usar mucha memoria para ganar una cantidad de tiempo que podría no ser significativa. Por el contrario, si las cotas de R y n son razonbales, el algoritmo de programación dinámica muestra mejores resultados para casos generales.