PRÁCTICA FINAL INDIVIDUAL

(Teoría Avanzada de la Computación)

0-1 Knapsack

(Problema de la mochila)

Fecha	29/05/2020
Autor	Ángel Luis Alonso Blázquez
NIA	100363923
Grupo	83

ÍNDICE

1. Introducción

En el presente documento abordaremos un problema de optimización NP en su variante 0-1 (con todos los inputs enteros). Esta variable tiene un algoritmo que lo resuelve de form óptima en tiempo pseudo-p, convirtiendo el problema en un problema NP-completo débil.

Un problema NP es un problema que no se puede resolver en un tiempo polinómico por una máquina de Turing determinista. Un problema es NP-completo si ese problema está contenido en el conjunto NP y todo problema NP puede ser reducible a este problema. Al haber un algoritmo que lo resuelve en tiempo pseudo-p, este problema se convierte automáticamente en un problema NP-completo débil, como hemos comentado antes.

2. Descripción

2.1. Problema abordado

El **problema de la mochila** o **knapsack problem** es un problema NP-completo de optimización combinatoria, donde dada una cantidad de elementos diferentes, el peso de cada uno de ellos y el valor de los mismos, tratamos de maximizar el valor de los objetos que se incluyen en una 'mochila' que soporta *x* peso.

2.2. Aplicación escogida

La variante escogida de este problema ha sido **bounded 0-1 knapsack problem**. Esta variante limita la cantidad de **copias de cada elemento** a **1** y define los **pesos** de los **objetos** en **enteros estrictamente positivos**.

Restricciones: nº de items del mismo tipo en la mochila = {0,1}
pesos de cada item = integer & >0
pesoX1 + pesoX2 + ... + pesoXN <= W

Un conjunto de datos válido sería: *3 items: {X1, X2, X3}* *(n = 3)*
item = [valor, peso] → *X1 = [60, 1]* *X2 = [100, 2]* *X3 = [120, 3]*
Peso máx de la mochila (W) = 5
Solución: *220* *{X1=0, X2=1, X3=1}*

2.3. Algoritmos desarrollados/escogidos

2.3.1. Recursive Implementation

El primer algoritmo **escogido** ha sido uno recursivo. Este algoritmo considera todas las combinaciones posibles de solución y devuelve la que maximiza el valor. Es un algoritmo **completo** (ya que en caso de existir solución, la encuentra) y **admisible** (encuentra siempre la solución óptima.
Ejecutando el ejemplo propuesto en el apartado anterior, el árbol resultante sería el siguiente:

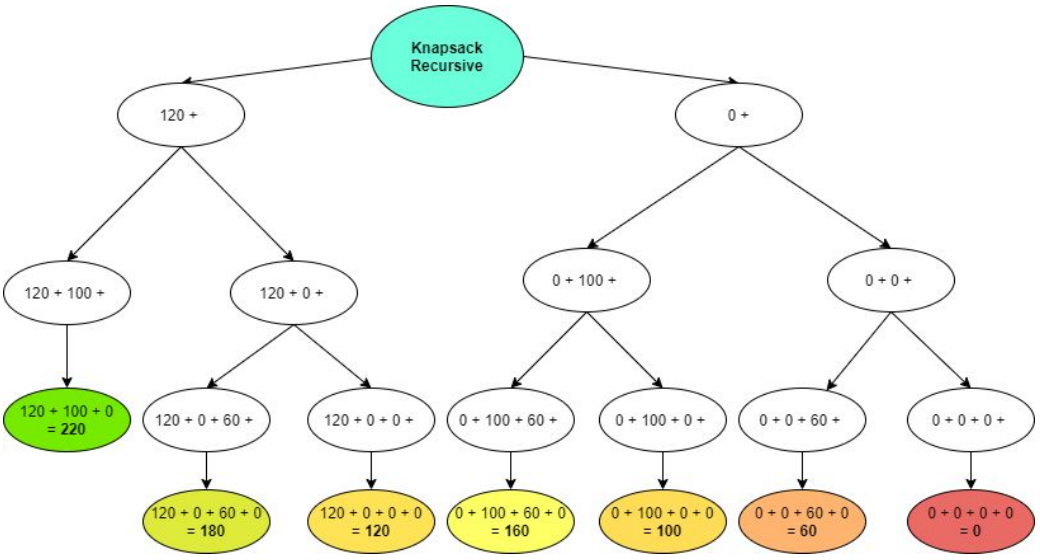


Figura 1. Recursion tree (img > 2_3_1-RecursionTree.png).

Esta implementación en concreto nos ahorra cada uno de los estados repetidos de la combinatoria natural; ej: se hace solamente {X3, X2}, y no {X2, X3} debido a que el código va desde el último item al primero (si se puede) y no vuelve a hacerlo en el otro sentido.

2.3.2. Dynamic Programming Implementation

El siguiente algoritmo **escogido** es programación dinámica. Este algoritmo es **completo**, **admisible** y considera los mismos casos que los mencionados en el algoritmo recursivo pero evita explorar todas las secuencias posibles por medio de la resolución de subproblemas de un tamaño que va en aumento. Está basado en el principio de optimalidad de Bellman:

"Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve."

Se crea un array bidimensional que se rellena de manera ascendente. Para el ejemplo propuesto en el apartado 2.2 la tabla resultante sería la siguiente, encontrándose el resultado en la posición [n][W]:

```
[0,  0,   0,   0,   0,   0]
[0, 60,  60,  60,  60,  60]
[0, 60, 100, 160, 160, 160]
[0, 60, 100, 160, 180, 220]
```

2.3.3. Brute Force Implementation

El algoritmo **desarrollado** ha sido puramente fuerza bruta. Este algoritmo es **completo** y **admisible** ya que contempla todas las combinaciones no repetidas.

El **funcionamiento** es el siguiente: primeramente se calcula cada una de las combinaciones (sin repetición) de los elementos totales. Estas son devueltas en forma de *iterador*, que no penaliza la memoria como sí fuera el caso de una *lista*. Para el ejemplo propuesto en el apartado 2.2 quedarían unas tuplas así: *(0,) (1,) (2,) (0, 1) (0, 2) (1, 2) (0, 1, 2)*. Se van sumando los valores (val[]) que tiene cada elemento de la combinación y se **poda** la iteración si la suma de un objeto de la combinación sobrepasa el peso máximo permitido (W).

3. Experimentación

3.1. Descripción del planteamiento experimental

El planteamiento ha consistido en probar básicamente 3 cosas: comprobar el buen funcionamiento de los 2 algoritmos implementados (recursivo y programación dinámica) y de la implementación creada (fuerza bruta), un test con un peso límite de la mochila (W) entre 20 y 50 y otro test aumentando al doble este peso. Con estos dos últimos tests podemos ver la correspondencia en el aumento del orden de valor del límite de pesos y/o ítems a analizar con el coste experimental de la solución y en el punto 4.2 relacionarlo con el coste hallado analíticamente.

La implementación y ejecución de los algoritmos y los tests a analizar se encuentra en el archivo '**Code.ipynb**', el cual está más desarrollado en profundidad en el anexo.

3.2. Resultados

En el **test #1** se ha tenido en cuenta un vector de valores (val) enteros estrictamente positivos inicializados aleatoriamente entre 10 y 1000, un vector de pesos (wt) enteros estrictamente positivos inicializados aleatoriamente entre 10 y 100 y un peso límite (W_t1) inicializado entre 20 y 50. En el **test #2**, la única variante ha sido el valor del peso límite (W_t2), la cual se ha aumentado de orden de 20-50 a 50-100. Hay un **caso especial**, que es la explotación del peor caso del algoritmo recursivo (RW), inicializando cantidades que hacen que sus podas nunca se cumplan: el vector de pesos (wt_rw) se inicializa todo a 1, mientras que la variable de peso máximo (W_rw) es igual al mayor número de ítems que se van a evaluar (highest_test_rw). Se ha realizado un tercer test (**test #3**) para observar el comportamiento de la ejecución recursiva.

La **ejecución de los algoritmos** (y el máximo de ítems a evaluar) han sido **limitados** tanto inferiormente como superiormente. Para todos los algoritmos, **inferiormente** empiezan siempre en 8 items. Esto se ha calculado tras estudiar varias ejecuciones y ver donde empezaban los casos de tiempos significativos. **Superiormente** hay varias diferencias: las versiones con alto coste computacional (fuerza bruta y recursivo peor caso) han sido limitadas a 30 ítems a analizar como máximo, ya que con ejemplos de 8 a 30 tenemos más que suficiente para evaluarlos (fuerza bruta con 30 ítems tarda en torno a media hora en ejecutarse). Para programación lineal, que tiene una complejidad pseudo-p, y para recursividad normal, que tiene un comportamiento peculiar que evaluaremos a continuación, se han limitado a 350 ítems en el *test #1* (highest_test_t1) y a 150 ítems en el *test #2* (highest_test_t2).

El coste computacional experimental se evaluará en el punto 4.2 junto al coste computacional analítico hallado en el punto 4.1.

Todos los gráficos mostrados en el presente documento se encuentran en el archivo '***Experimental_vs_Analítico.xlsx***'.

3.2.1. Test #1

Todos los resultados están en milisegundos (ms). En esta sección se comentarán los aspectos más importantes de la ejecución experimental del test #1. Los puntos que se quieran comparar con el test #2 se mostrarán en el punto 3.2.3.

Figura 2. Gráfico de 8 a 30 ítems comparando R, RW, DP y BF (en milisegundos).

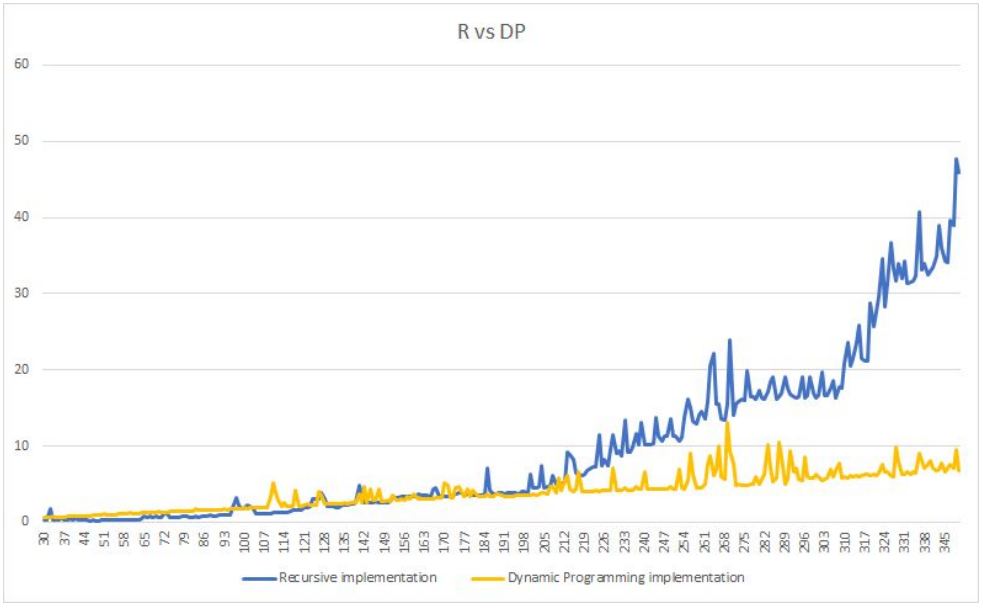


Figura 3. Gráfico de 30 a 350 ítems comparando R con DP (en milisegundos).

Al ejecutar el test con las limitaciones en los algoritmos de fuerza bruta y recursivo (en el peor caso): ítems de **8 a 30**, podemos observar el carácter exponencial de estas dos ejecuciones. La ejecución del **peor caso** del algoritmo **recursivo** ajusta a una función exponencial ($t(N) = 0{,}1584e^{0{,}6782N}$) con un R^2 de 0'9982. La ejecución por **fuerza bruta** ajusta a una exponencial también ($t(N) = 0{,}3932e^{0{,}6562}$) con un R^2 de 0'9973. Los tiempos de programación dinámica y del recursivo se mantienen mucho más bajos, por lo que tendremos que extender la gráfica.

Para los ítems de **30 a 350** ya podemos conseguir unos resultados más concluyentes acerca de estas dos ejecuciones. **Programación dinámica** parece tener un crecimiento **lineal** (que comprobaremos en el punto 4.2.2. Los tiempos de ejecución del caso **recursivo** son algo más curiosos y los analizaremos en detalle.

RW es el peor caso de la ejecución recursiva, por tanto es **cota** superior. podemos ver que con el test realizado, la ejecución del algoritmo no llega ni de lejos a esta cota:

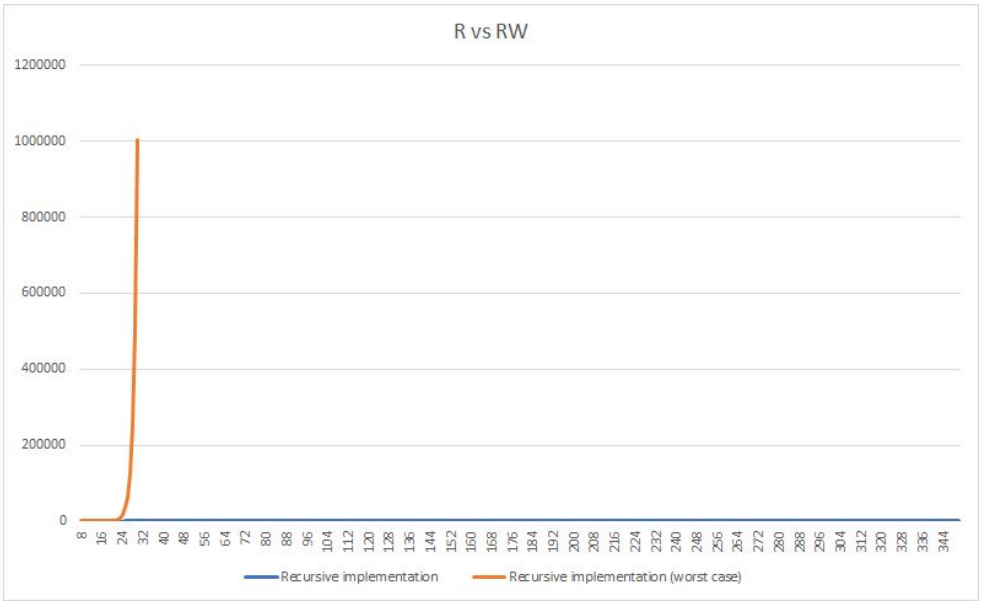


Figura 4. Gráfico de 8 a 350 ítems comparando R con RW (en milisegundos).

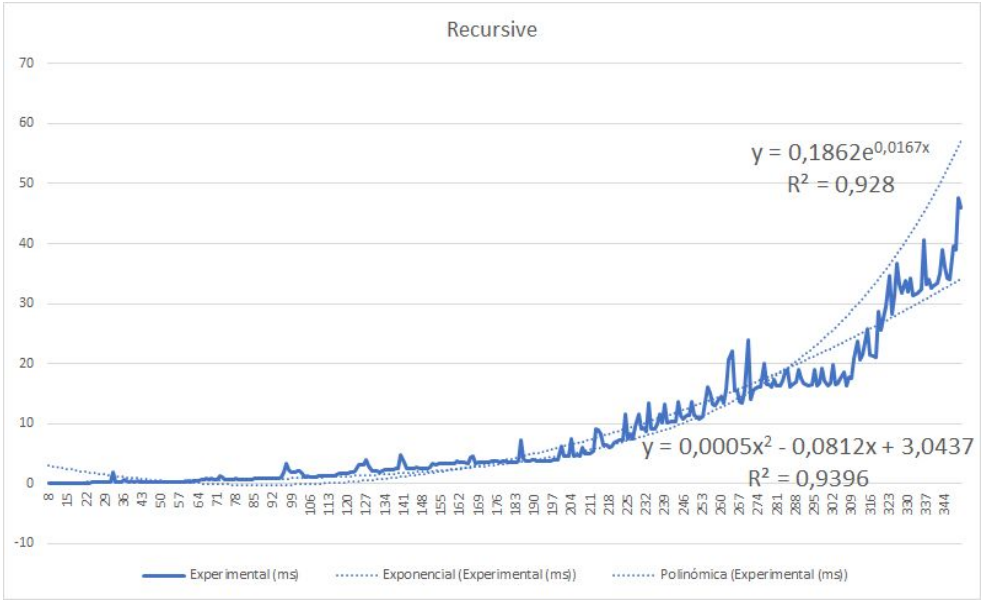


Figura 5. Gráfico de 8 a 350 ítems de ejecución de R (en milisegundos).

Esta ejecución ajusta de primeras incluso más a una función de tiempo **polinómico** que a una de tiempo **exponencial**. Esto es debido a que el peso límite (W) es bajo (entre 20 y 50 para este test), y muchas veces actúa la **poda** de ramas. Para comprobar que este comportamiento no se debe a falta de ejecuciones, se estudiará más a fondo con una ejecución más grande en el **test #3** (punto 3.2.4).

Podemos observar en esta gráfica también que hay momentos en el que los **tiempos** de ejecución se mantienen medianamente **constantes** (de 274 a 309). Esto podría ser debido a que, aunque se añaden mayor número de ítems, la **poda** corta esas ramas de manera muy **temprana**. Los motivos de la poda deben ser por el valor de los pesos de los ítems nuevos, que seguramente superen el peso límite de la mochila (W) por sí mismos o sumándolos en las diferentes combinaciones.

3.2.2. Test #2

Todos los resultados están en milisegundos (ms). En esta sección se comentarán los aspectos más importantes de la ejecución experimental del test #2. Los puntos que se quieran comparar con el test #1 se mostrarán en el punto 3.2.3.

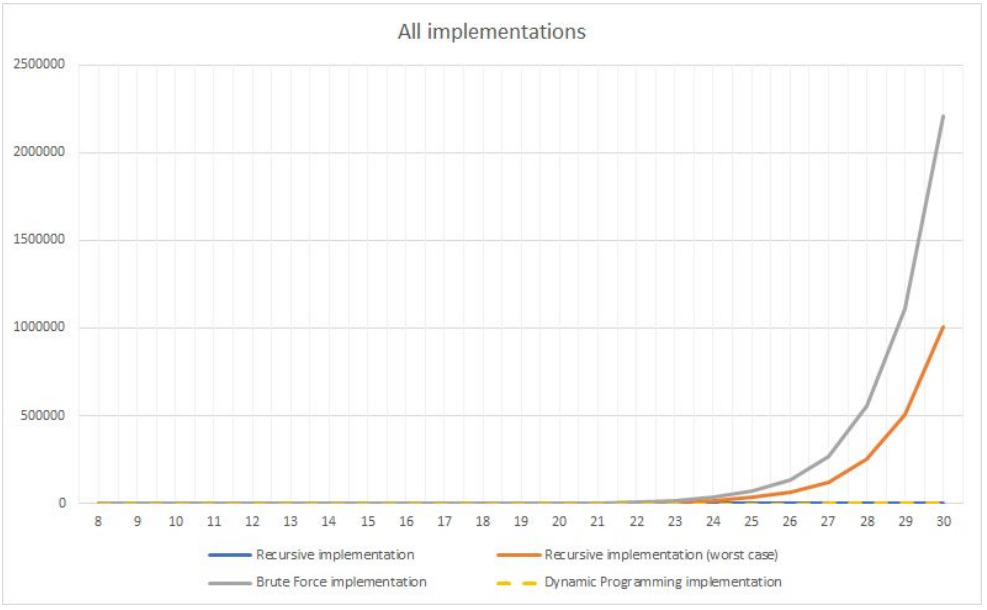


Figura 6. Gráfico de 8 a 30 ítems comparando R, RW, DP y BF (en milisegundos).

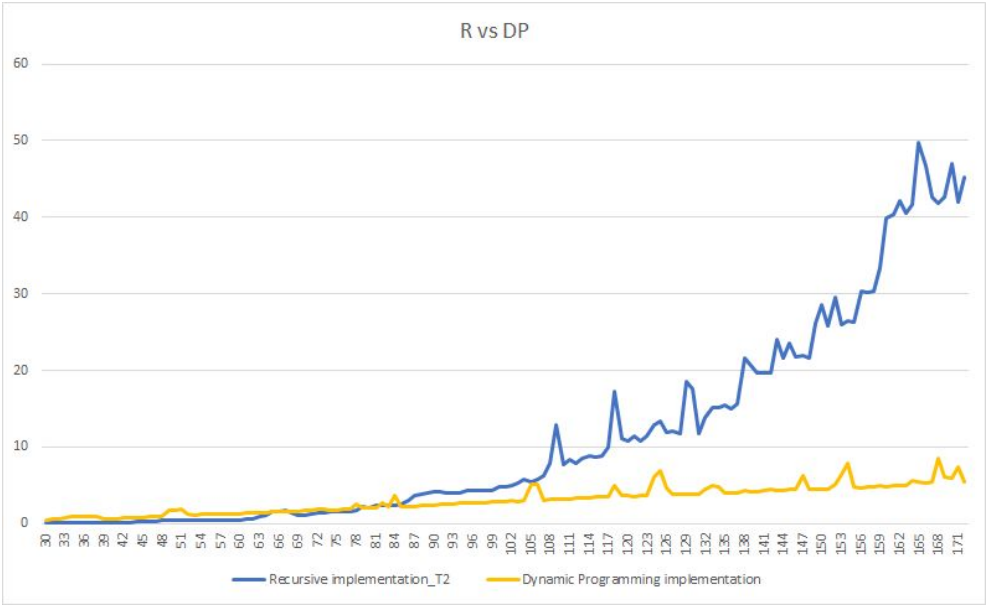


Figura 7. Gráfico de 30 a 350 ítems comparando R con DP (en milisegundos).

En este test, al **aumentar** el valor del peso límite (**W**) de la mochila, se **aumenta** el **tiempo** de **ejecución** de los algoritmos (excepto rw que tiene sus propios datos). El algoritmo de **fuerza bruta** llega a un tiempo de ejecución de 2211 segundos (~37 minutos) con 30 ítems, que comparado con el test #1, que son 1800 segundos (~30 minutos), es superior.

Observando de cerca el algoritmo de **programación dinámica**, este aumenta sus tiempos debido a que la tabla que se crea es de tamaño N*W, por lo tanto, al aumentar W, la tabla es más grande en el test #2 y el tiempo de ejecución a su vez es mayor.

Ej: test #1 DP (150 ítems): 5,49 ms | test #2 DP (150 ítems): 2,88 ms.

Al doblarse el valor de W, se ve una relación en los tiempos experimentales de ejecución del mismo orden.

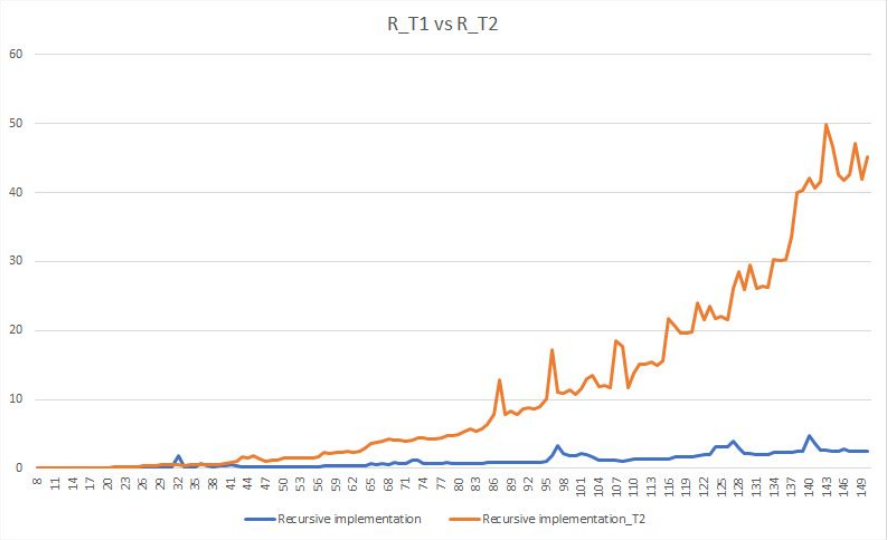


Figura 8. Gráfico de 30 a 350 ítems comparando R de Test #1 con R de Test #2 (en milisegundos).

La ejecución **recursiva** también **aumenta** sus **tiempos** (**Figura 8**) y además se 'despega' antes de los tiempos de programación dinámica (**Figura 7**), ya que a poda se ejecuta en nodos más profundos, por lo que poda bastante menos que en el test #1 y tiende más al peor caso del algoritmo (rw = complejidad exponencial). La ejecución sigue muy por **debajo** de la **cota superior**:

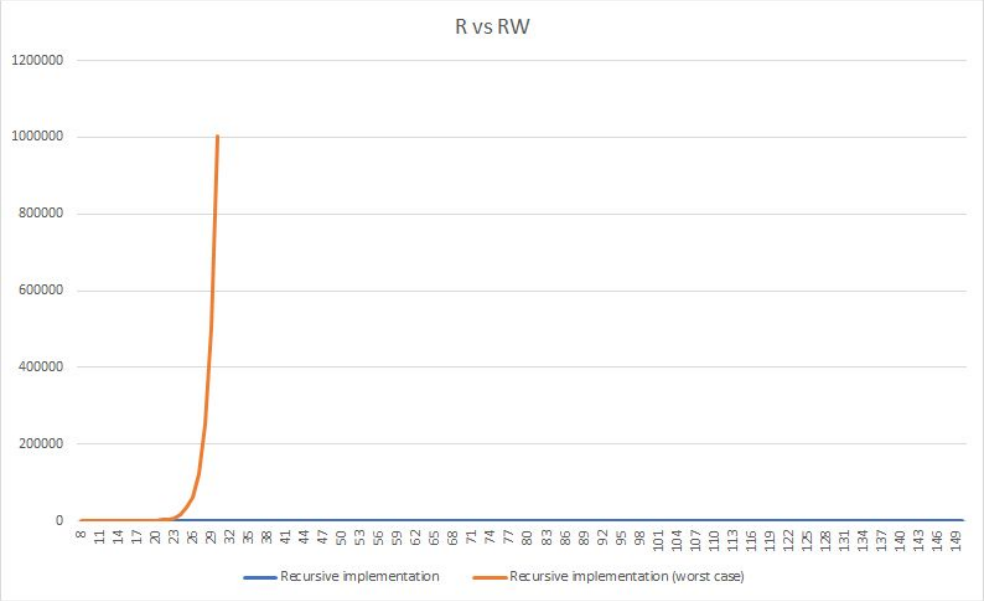


Figura 9. Gráfico de 8 a 350 ítems comparando R con RW (en milisegundos).

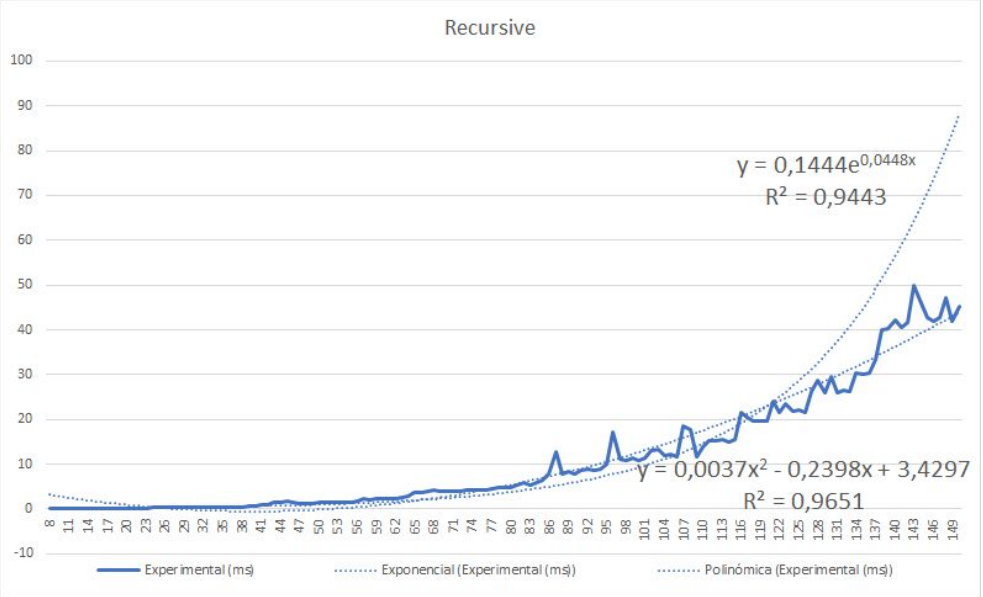


Figura 10. Gráfico de 8 a 350 ítems de ejecución de R (en milisegundos).

Además, la similitud con una gráfica de complejidad O(n^2) aumenta a 0'965 (también es verdad que el número de ítems a analizar a decrecido). Este comportamiento se estudia más a fondo en el **test #3** (punto 3.2.4).

3.2.3. Comparación entre tests #1 y #2

Todos los resultados están en milisegundos (ms). Comparación de todas las ejecuciones realizadas en los test #1 y #2.

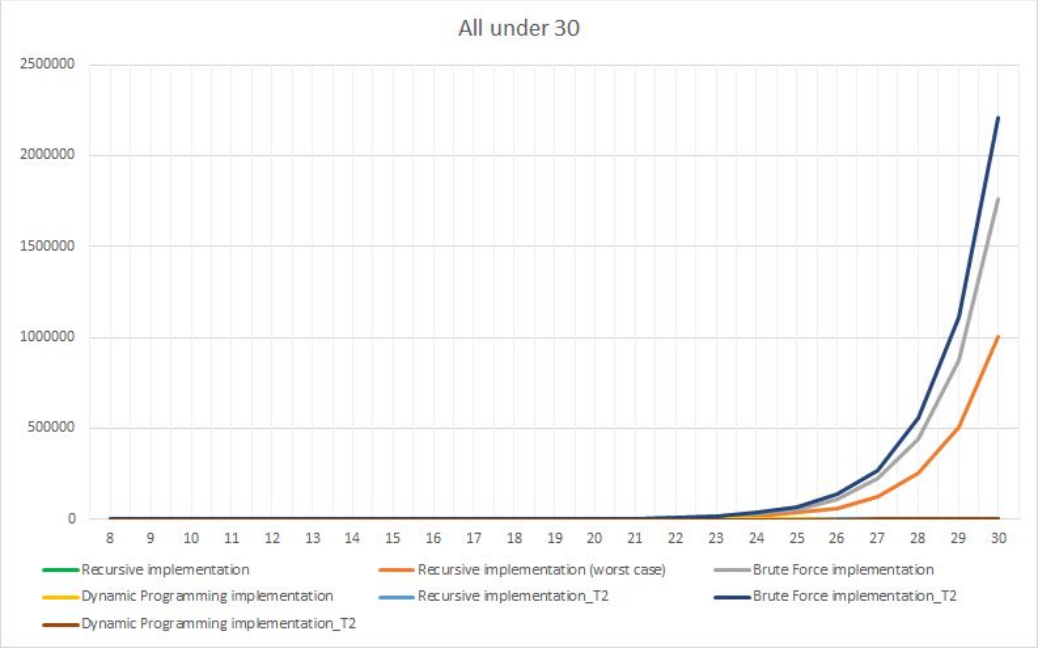


Figura 11. Gráfico de 8 a 30 ítems comparando R, RW, DP y BF de Test #1 y Test #2 (en milisegundos).

Podemos observar que el **ejecución más lenta** es la de **fuerza bruta** del *Test #2* (donde se aumenta el valor de W). Esto ocurre ya que las **podas** del mismo son más profundas que el *Test #1* y hacen que el algoritmo realice más operaciones.

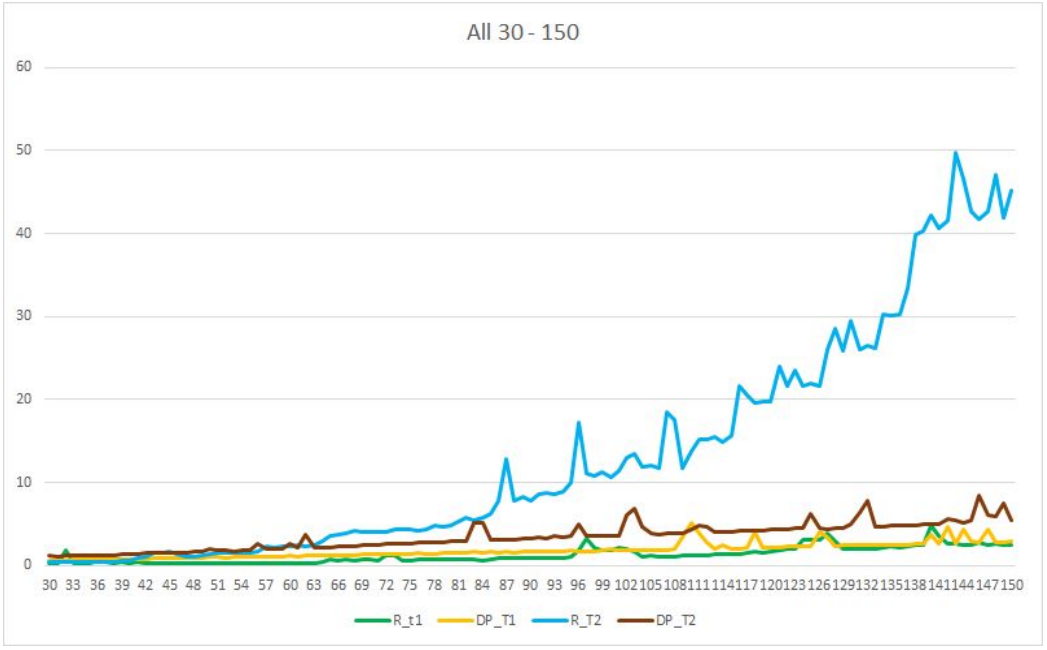


Figura 12. Gráfico de 30 a 150 ítems comparando R y DP de Test #1 y Test #2 (en milisegundos).

Apreciamos en la gráfica que con el aumento de W (Test #2), la ejecución **recursiva** ya **no** es tán **rápida** durante tanto tiempo como en la primera ejecución, donde se cumplían ciertas características (W muy pequeño en comparación a pesos de wt). Las ejecuciones **DP** continúan de manera lineal, notándose el pequeño 'gap' que hay entre cada una debido al incremento del tiempo de ejecución (O(N)=(N*W)) por el aumento de W (de 20-50 a 50-100).

3.2.4. Test #3

Se ha intentado llegar a los 1000 ítems, pero Google Colaboratory excede la máxima profundidad (error). Se ha limitado a 900 ítems para poder observar el comportamiento de la ejecución recursiva. Se ha usado un vector de valores (val) enteros estrictamente positivos inicializados aleatoriamente entre 10 y 1000, un vector de pesos (wt) enteros estrictamente positivos inicializados aleatoriamente entre 10 y 100 y un peso límite (W_t3) inicializado entre 20 y 50. El número de ítems evaluados han sido de 8 a 900:

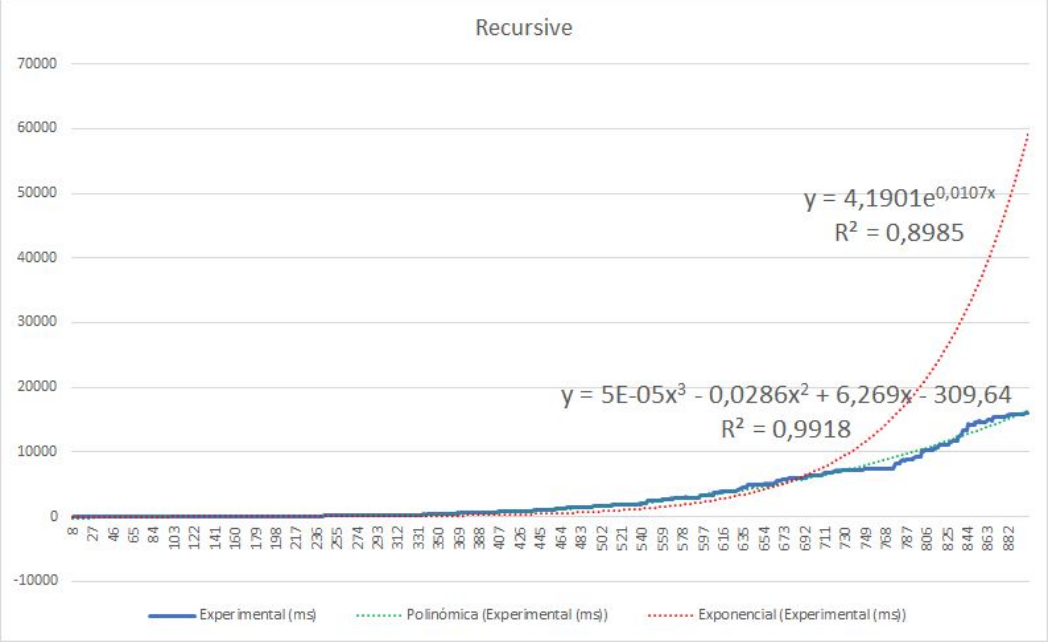


Figura 13. Gráfico de 8 a 900 ítems de ejecución de R (en milisegundos).

Podemos ver que el comportamiento polinómico sigue existiendo y se mantiene (en este caso con una función cuadrática el ajuste es de R^2 = 0'97). La **tendencia polinómica** (línea verde) mostrada en la **gráfica** es **cúbica**, que se ajusta con un R^2 = 0'99.

Esto nos quiere decir que para casos **muy particulares**, como por ejemplo en este, en el que el peso límite de la mochila es muy pequeño en comparación a los pesos (caben como máximo 5 ítems en la mochila si W = 50 y hay cinco pesos que sean igual a 10), se puede obtener una **complejidad** de **ejecución polinómica** debido a las **podas** (las cuales son muy severas en este caso). En cuanto **aumentamos** el peso límite de la mochila (**W**), como en el test #2, observamos un **incremento** de tiempos **no polinómico**.

4. Análisis

4.1. Coste computacional analítico

4.1.1. Recursive implementation

Para realizar el análisis de la complejidad vamos a tener en cuenta el **peor caso**: cuando la poda `if (wt[n-1] > W)` nunca se cumple.

La complejidad de la función es la siguiente: 2 comprobaciones en el primer if; 1 acceso y 1 comprobación en el segundo if; y 1 acceso a 'val', 1 comparación en el 'max' y 2 llamadas recursivas. Esto da lugar a 6 accesos de memoria y 2 llamadas recursivas:

T(n) = 6 + 2 * T(n - 1) si n >= 1 siendo T(0) = 3

T(n) - 2 * T(n - 1) = 6 → No es homogénea → T(n - 1) - 2 * T(n - 2) = 6
T(n) - 2 * T(n - 1) = T(n - 1) - 2 * T(n - 2)

T(n) - 3 * T(n - 1) + 2 * T(n - 2) = 0 → Homogénea
$x^2 - 3x + 2 = 0$ → r1=2 ; r2=1 → (x - 2)*(x - 1) = 0

T(n) = b1* 2^n + b2* 1^n si n >= 1 T(0) = 3
T(1) = b1* 2^1 + b2* 1^1 = 12 | b1 = 9 ; b2 = -6
T(2) = b1* 2^2 + b2* 1^2 = 30 | **T(n) = 9 * 2^n - 6** para n>=0

T(R) = 9 * 2^n - 6 ~ O(R) = 2^n | **S(R) ~ 1**

4.1.2. Dynamic Programming implementation

Al comienzo del algoritmo se **inicializa** toda la matriz **K** a 0, que es la que almacena los valores de los diferentes cálculos que se harán seguidamente. En programación dinámica se necesita un tamaño de matriz de **número de elementos + 1** por **rango de 0 a total de límite + 1**. En este algoritmo, la **memoria necesaria**, S(DP), es **(N+1)*(W+1)**. La complejidad temporal de estos bucles anidados es la siguiente por cada uno: T(cond) + T(i=0) + Iter (T(cond) + T(i++) + T(S); T(cond) contiene una suma por lo que su valor total es 2. Dentro del bucle inferior hay tan solo una asignación a 0: T(U) = 1. El resultado es: **T(matrizK)** = 3 + IterN (3 + (3 + IterW (3 + 1)) =
= **3 + N * (6 + 4 W)**

El **otro** conjunto de **bucles anidados** tiene el mismo recorrido, tan solo cambia lo que hay dentro: T(tablaK) = 3 + N * (6 + W (3 + T(R))). Dentro contiene un **if** con dos condiciones y una asignación dentro; un **elseif** que contiene una asignación y una función max entre dos valores (uno de ellos una suma); y un **else** con una asignación. T(R) = T(cond1) + max (T(elseif), T(else)). T(cond1) evalúa 2 comparaciones, por lo tanto: T(R) = 2 + 13 = 15.

El máximo de entre elseif y else es **elseif**, por lo que calcularemos este último: T(elseif) = T(cond2) + T(O). La condición contiene una resta, un acceso a vector y la comparación (consideramos a 'w' como un int, no como un acceso) → T(cond2) = 3. En cuanto a T(O), se hace una comparación de máximos (1) entre una suma (1) entre los accesos a vectores (3), las restas dentro de las llamadas a vectores (4) y 1 acceso a vector. T(O) = 10 → T(elseif) =13.

Como resultado, tenemos que **T(tablaK) =** 3 + N * (6 + 18 W) = **3 (1 + N * (2 + 6 W))**. Por último, el último return cuesta 1. La complejidad total de la implementación en programación lineal es la siguiente:

T(DP) = 7 + 2 N * (6 + 11 W) ~ O(DP) = N*W | S(DP) ~ N*W

4.1.3. Brute Force implementation

En el principio de la implementación, se inicializa (1) a 0 la variable que se va a retornar al final de la función. A continuación se entra en un conjunto de bucles anidados del estilo:

T(comb) = T(cond1) + T(i++) + IterComb(T(cond1) + T(i++) + T(tupla)) = 2 + IterComb(...)
T(tupla) = T(CalcComb) + T(cond2) + T("i++") + IterTupla(T(cond2) + T("i++") + 2 + T(inside))

Siendo ***comb*** el bucle que calcula las posibles combinaciones (sin repetir) de los diferentes elementos y ***tupla*** el bucle que examina cada una de estas combinaciones generadas. Consideramos que **CalcComb** solo se ejecuta una vez por cada iteración del bucle combinaciones debido a que en este bucle se calculan las combinaciones posibles cuando le hacen falta (debido a que es de tipo iterator) y seguidamente se pasan los resultados al bucle *tupla*. Estos resultados se pasan, como hemos dicho, como iterables, teniendo un impacto en la memoria que vamos a usar, ya que los iterables se calculan 'al vuelo' (cuando se necesitan), por lo tanto no se necesita guardar la lista entera de todas las combinaciones en memoria. Esta ejecución devuelve $C_{n,x} = \frac{n!}{x!(n-x)!}$ *(x=i+1)* tuplas que guardar en cada ejecución. Como máximo puede almacenar en **memoria S(BF)** = $C_{n,z}$ (**z=n/2** para **n par** | **z=(n+1)/2** para **n impar**) elementos. Si se almacenara todo en una lista, el espacio en memoria auxiliar que necesitaríamos sería de **S'(BF)** = $\sum_{i=1}^{n} C_{n,i}$.

Consideraremos que el número de accesos a CalcComb, analizando el código base de python [14], en la función *combinations*, es de al menos $C_{n,i+1}$ y en la función *chain* igual, ya que realiza una transformación de formato por cada combinación.

Volviendo a los bucles,
T(inside) = T(cond3) + T("j++") + IterI(T(cond3) + T("j++") + T(S)) = 3 + IterI(3 + T(S))
T(S) = 2 (acceso a vector y suma) * 2 + T(cond4) + 1 (asignación) = 6
T(inside) = 3 + IterI(9) = 3 + 9 * len(comb_tuple)

El bucle *tupla* se ejecuta todas las combinaciones de 1 a N del conjunto N de elementos $(C_{n,i+1})$: **T(tupla)** = T(CalcComb) + 3 + IterTupla(8 + 9 * len(comb_tuple)) =

$$= 2\,C_{n,i+1} + 3 + C_{n,i+1} * (8 + 9 * (i+1))$$

El bucle *comb* se ejecuta N veces, siendo:

T(comb) = 2 + IterComb(2 + 2 $C_{n,i+1}$ + 3 + $C_{n,i+1}$ *(8 + 9 * (i+1))) =

$$= \sum_{i=0}^{N} [2 + N * (5 + 2\,C_{N,i+1} + C_{N,i+1} * (17 + 9\,i)\,)\,] =$$

$$= 2 + N*(N+1) \sum_{i=0}^{N} [5 + C_{N,i+1} * (19 + 9\,i)\,] =$$

$$= 2 + 5\,N*(N+1) + N*(N+1) \sum_{i=0}^{N} [\,C_{N,i+1} * (19 + 9\,i)\,] = \qquad (i + 1 = j) \rightarrow$$

$= 2 + 5\,N*(N+1) + N*(N+1) \sum_{j=1}^{N} [\, C_{N,j} *(10 + 9\,j)\,] =$

$= 2 + 5\,N*(N+1) + N*(N+1) \sum_{j=0}^{N} [\, C_{N,j} *(10 + 9\,j)\,] - 10 =$

$= -8 + 5\,N*(N+1) + N*(N+1) * 2^N * \sum_{j=0}^{N} [10 + 9\,j\,] =$

$= -8 + 5\,N*(N+1) + N*(N+1) * 2^N * \sum_{j=0}^{N} [10 + 9\,j\,] =$

$= -8 + 5\,N*(N+1) + 10 * N*(N+1) * 2^N + N*(N+1) * 2^N * \sum_{j=0}^{N} [9\,j\,] =$

$= -8 + N*(N+1)*[5 + 2^N * (10 + \sum_{j=0}^{N} [9\,j\,])\,] =$

$= -8 + N*(N+1)*[5 + 2^N * (10 + \sum_{j=1}^{N} [9\,j\,] + 0)\,] =$ deshacer c.v. (j = i+1) →

$= -8 + N*(N+1)*[5 + 2^N * (10 + \sum_{i=0}^{N} [9\,(i+1)\,])\,] =$

$= -8 + N*(N+1)*[5 + 2^N * (10 + 9 + \sum_{i=0}^{N} [9\,i\,])\,] =$

$= -8 + N*(N+1)*[5 + 2^N * (19 + 9 * \frac{N*(N+1)}{2})\,] =$

$= -8 + N*(N+1)*[5 + 2^N * (19 + 9 * N*(N+1))\,]$

T(BF) = -7 + N*(N + 1)*[5 + 2^N * (19 + 9 * N*(N + 1))] ~ O(BF) = 2^N

S(BF) ~ $C_{N,z}$ (**z=N/2** para **N par** | **z=(N+1)/2** para **N impar**)

4.2. Relación con resultados experimentales

Seguidamente, se presentan gráficos de cada implementación en los que se muestra una línea con los valores experimentales, otra con la función analítica corregida (se ha multiplicado por una constante para ajustar el orden de tiempo) y una última línea de puntitos donde se muestra la línea de tendencia. Todo está en ms.

4.2.1. Recursive implementation

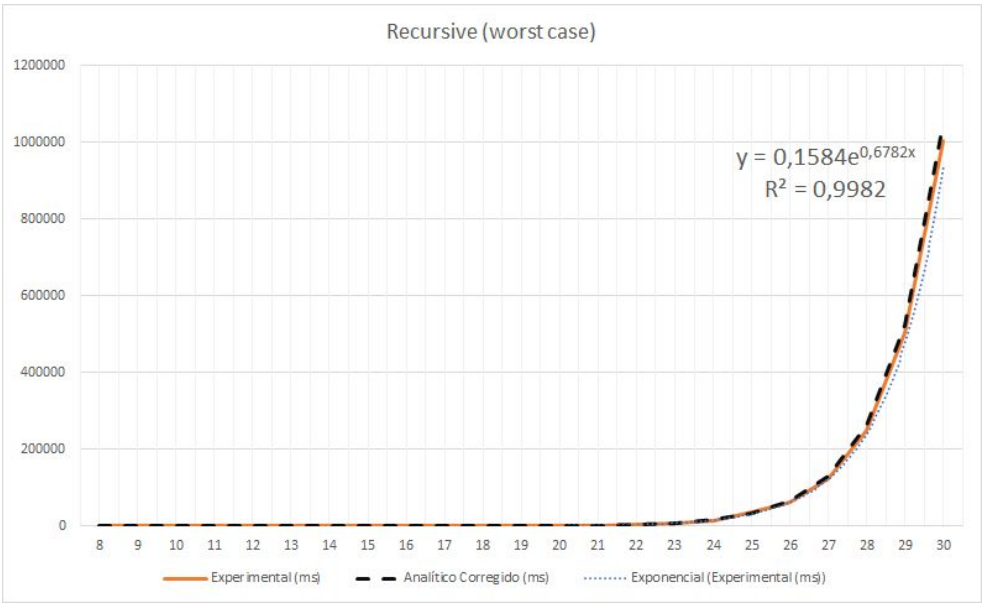


Figura 14. Gráfico de 8 a 30 ítems de ejecución de RW (en milisegundos).

Ajuste de analítico: $\frac{T(R)}{9*2^{10}}$

En esta gráfica de **8 a 30 ítems** podemos en la línea naranja el valor experimental que toma la gráfica, con una línea discontinua la función analítica y de forma tenue la línea de tendencia de los valores experimentales. Como podemos apreciar, el **analítico** calculado se **ajusta** bastante **bien** a los valores experimentales, y a su vez, la **línea de tendencia** exponencial se ajusta con un R^2 de 0'998 (casi perfecta) a los datos experimentales.

Este caso, que es el peor caso posible de tiempo (no se explotan las podas del algoritmo) en la implementación recursiva, se puede considerar la **cota superior** de la misma (**9** * 2^n **- 6**). Todas las demás ejecuciones tendrán el mismo coste o menor, pero nunca mayor. Por ser el peor caso de la ejecución se ha usado T(R) en vez de O(R), para mostrar la cota superior.

4.2.2. Dynamic Programming implementation

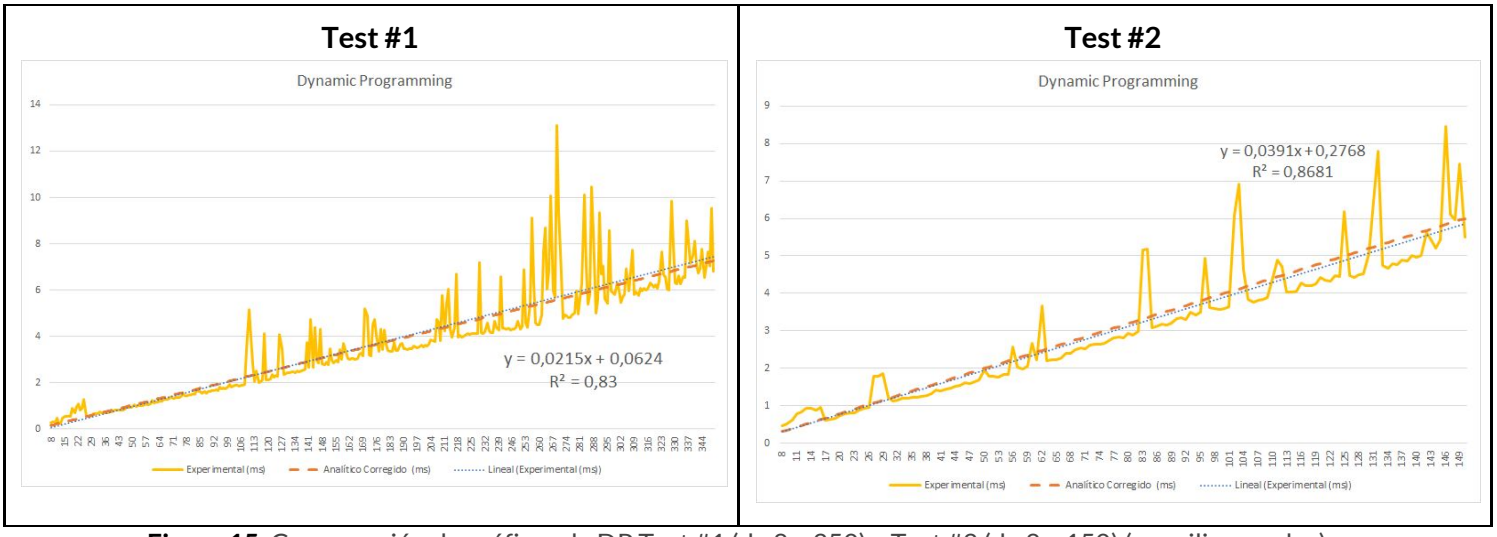


Figura 15. Comparación de gráficas de DP Test #1 (de 8 a 350) y Test #2 (de 8 a 150) (en milisegundos).

Ajuste de analítico: $\frac{O(DP)}{100*13}$

Para la versión en programación dinámica (de **8 a 350 ítems** en el **test #1** y de **8 a 150 ítems** en el **test #2**) hemos usado **O(DP)** en vez de T(DP) ya que nos interesa mostrar que los datos experimentales siguen una distribución, a primera vista, lineal. Realmente esta forma de enfocar el problema nos permite resolver el problema en **tiempo pseudo-polinómico**. Esto es así porque el tiempo de ejecución depende tanto del número de inputs (N) como de lo grande que sea el peso máximo permitido (W) → O(DP): **N*W**.

En un algoritmo de programación dinámica, se crea una matriz bidimensional de longitud 0,1,...,W por 0,1,...,N, entonces, contra más crezca una de las dos variables, más tardará la ejecución. Si reducimos el orden del vector de pesos (wt) y de (W), dividiéndolo por un mismo número, podemos lograr contrarrestar el incremento de tiempo creado por el aumento de ítems a valorar (N). El único problema es que los valores tienen que seguir manteniéndose como integers estrictamente positivos.

4.2.3. Brute Force implementation

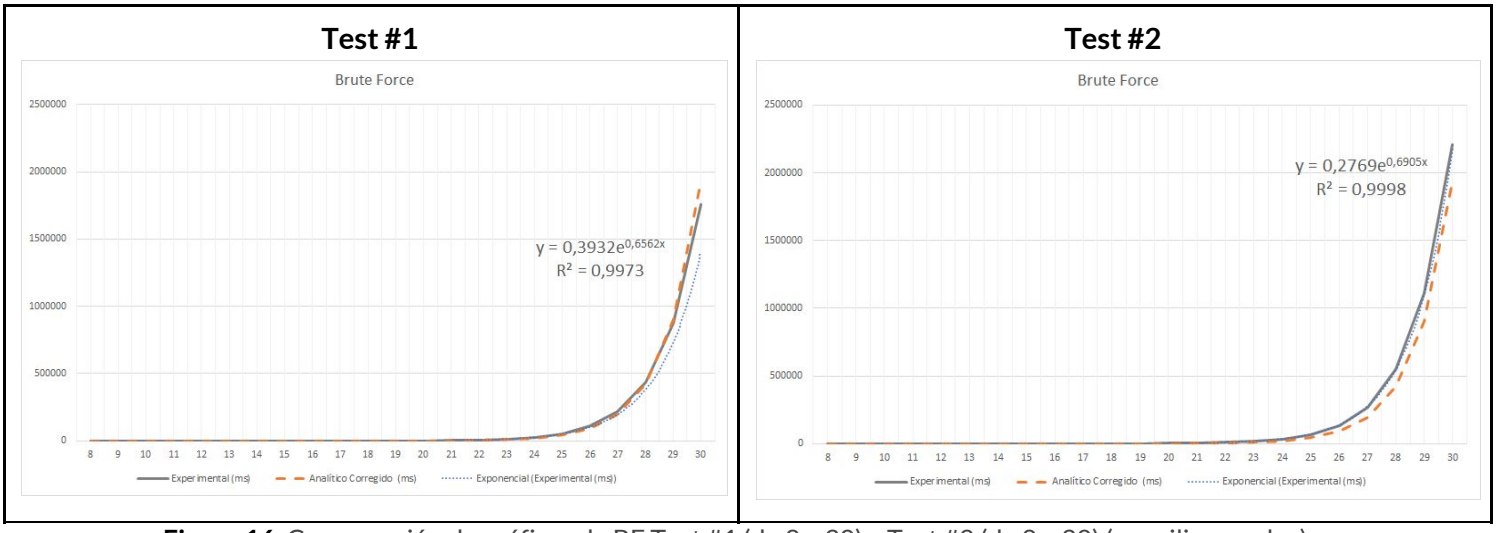


Figura 16. Comparación de gráficas de BF Test #1 (de 8 a 30) y Test #2 (de 8 a 30) (en milisegundos).

Ajuste de analítico: $\frac{T(BF)}{100*6*2^{14}}$

Para el algoritmo auto-diseñado que resuelve el problema por fuerza bruta (de **8 a 30 ítems** en este caso), usamos **T(BF)** en busca de una **cota superior**. Observando las gráficas, apreciamos que los datos experimentales sobrepasan en tiempo (sobretodo en el test #2) a la supuesta cota superior T(BF). Esto puede ser debido a que en el cálculo del analítico realizado en el punto 4.1.3, hemos aproximado el valor de la función *itertools.combinations* e *itertools.chain* a la baja, cogiendo el valor de complejidad que al mínimamente cumplía ($C_{n,i+1}$ para cada una de las dos funciones).

Las **podas** del algoritmo no tienen casi incidencia, ya que la complejidad se halla en calcular las tuplas de las diferentes combinaciones posibles, por lo que, aunque ayudan, no son tan decisivas como en la implementación recursiva para casos específicos.

Por otra parte, podemos ver que el la ejecución **experimental** cumple casi con total afinidad la línea de tendencia **exponencial** (R^2: 0,9998 y 0,9973), que es igual a O(n) (2^n).

5. Conclusiones

Como se ha comentado en el punto 4.2.2, en un algoritmo de **programación dinámica**, se crea una matriz bidimensional de longitud 0,1,...,W por 0,1,...,N. Entonces, contra más crezca una de las dos variables, más tardará la ejecución. Si reducimos el orden del vector de pesos (wt) y de (W), dividiéndolo por un mismo número, podemos lograr contrarrestar el incremento de tiempo creado por el aumento de ítems a valorar (N). El único problema es que los valores tienen que seguir manteniéndose como integers estrictamente positivos. Redondeando correctamente estos valores, podríamos conseguir una **solución aproximada** de nuestro problema reduciendo el tiempo de ejecución.

Como se ha profundizado en el punto 3.2.4, se puede conseguir una ejecución **polinómica** del algoritmo **recursivo** en condiciones muy particulares. Estas condiciones son cuando las **podas** son **severas** debido a que el peso límite de la mochila (**W**) es **pequeño** en comparación a los pesos inicializados para los diferentes ítems (wt).

Con los resultados obtenidos, podríamos mejorar el tiempo de ejecución medio del algoritmo de **fuerza bruta**. Podríamos crear una versión modificada de esta implementación que en vez de calcular una combinación y directamente actuar sobre ella, calcular las diferentes combinaciones no repetidas (almacenándolas en memoria) y luego ejecutar el bucle anidado. Cuando se detecte una combinación en la que la suma supera el peso límite (W), todas las combinaciones que contengan esta subcombinación se borran de memoria para así no calcularlas innecesariamente y **podar** estas. Esta solución sacaría la **solución óptima** ya que no afecta a la completitud del algoritmo pero por contraparte, **afectaría drásticamente** al uso de **memoria**, pasando de $C_{N,z}$ (z=N/2 para N par | z=(N+1)/2 para N impar) a $\sum_{i=1}^{N} C_{N,i}$ (uno por cada combinación almacenada). Además de sumar la complejidad de buscar y eliminar las combinaciones que contienen esta subcombinación rechazada.

Hay otras formas de abordar el problema de manera que se consiga una **solución** en **tiempo polinómico**, pero estos algoritmos no son óptimos (no encuentran la mejor solución). Uno de ellos es el GA (Greedy algorithm). Este algoritmo tiene una complejidad temporal (O grande) de N*log(N). Observando los resultados de este paper [5] (página 11), podemos comprobar que la solución que genera este algoritmo encuentra una solución que se acerca bastante a la óptima en un tiempo razonable. Además, el uso de memoria es igual a N (bastante bajo).

Las **aplicaciones** posibles para este problema [7] son: la optimización de la asignación de energía a electrodomésticos, 'Large-scale multi-period precedence constrained knapsack problem: A mining application' [8], resolver el problema de planificación de producción, gestión de asignación de energía (y demás aplicaciones de distribución de energía), etc.

6. Referencias

1. Introduction to 0-1 Knapsack Problem - enlace
2. The 0-1 Knapsack Problem and a dynamic programming solution to this problem - enlace
3. 0-1 Knapsack Problem | DP-10 - enlace
4. Python Program for 0-1 Knapsack Problem **(Recursive and DP implementations)** - enlace
5. Different Approaches to Solve the 0/1 Knapsack Problem - enlace
6. Comparison and Analysis of Algorithms for the 0/1 Knapsack Problem - enlace
7. Some applications of Knapsack problem - enlace
8. Large-scale multi-period precedence constrained knapsack problem: A mining application - enlace
9. Programación dinámica - enlace
10. Numpy: The fundamental package for scientific computing with Python - enlace
11. Matprolib: Python plotting - enlace
12. Pandas: Python Data Analysis Library - enlace
13. Análisis de la complejidad de los algoritmos - enlace
14. Itertools — Functions creating iterators for efficient looping - enlace

7. Anexos

7.1. Anexo 1: Archivos y tutorial de uso

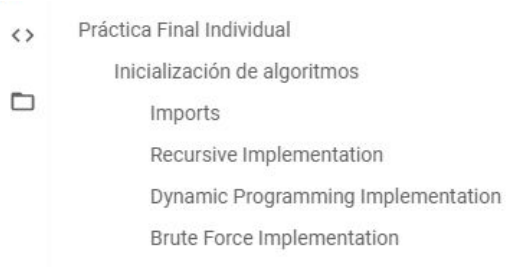
La entrega contiene los diferentes archivos:

- **execution_time**: carpeta en la que se encuentran los diferentes archivos .csv de los datos de la ejecución de los los tests realizados.
- **img**: aquí están las imágenes que se han usado en la memoria (excepto los gráficos).
- **Code.ipynb**: cuadernillo donde se implementan los algoritmos y se ejecutan las pruebas. Enlace del cuadernillo en Google Colab (solo accesible con un correo de la UC3M).
- **Experimental_vs_Analítico.xlsx**: archivo excel donde se encuentran las gráficas implementadas en la memoria con todos sus datos transformados.
- **memoria.pdf**: memoria explicando el problema, las implementaciones, costes y resultados.

7.1.1. Code.ipynb

7.1.1.1. Inicialización de los algoritmos

En esta parte se añaden los imports necesarios y las 3 diferentes implementaciones que usamos: recursiva, programación dinámica y fuerza bruta.



En la implementación de fuerza bruta se puede quitar unos comentarios que hay en el código para conseguir que el algoritmo te devuelva la tupla que maximiza los valores:

Descomentamos *best_tuple* del inicio de la función.

```
max_value = 0
# best_tuple = ()
```

Comentamos `max_value = max(current_value, max_value)`, `return max_value` y descomentamos el código comentado que aparece en la imagen:

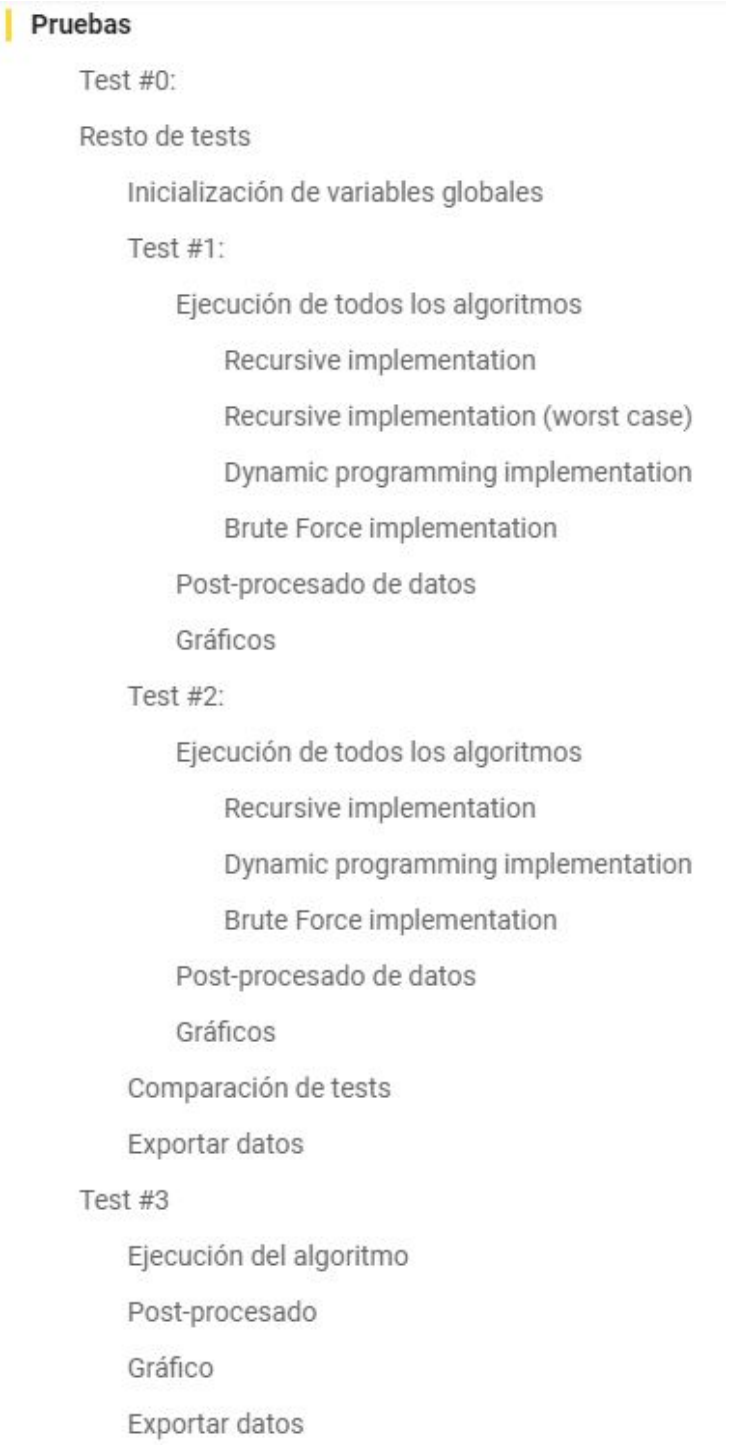
```
  max_value = max(current_value, max_value)

  '''
  # Tuple return modify (you have to comment the previous line & uncomment 'best_tuple' inicialize)
  if (max_value < current_value):
    max_value = current_value
    best_tuple = comb_tuple

print("Combination: ")
print(best_tuple)
  '''

return max_value
```

7.1.1.2. Pruebas



El *Test #0* es una ejecución básica de las implementaciones donde se concentra todo.

El *resto de tests* tienen una estructura diferente. Al comienzo se inicializan unas variables que afectan a los tests *#1* y *#2*, que hacen más fácil el mantenimiento del código.

Después se ejecutan los diferentes algoritmos de los tests, se procesan (en función de la cantidad de tests que hemos hecho, y se saca unos gráficos orientativos acerca de la ejecución.

Después se sacan gráficos con todos los datos de los tests *#1* y *#2* para ver la diferencia entre ejecuciones.

Existe un apartado de *Exportar datos*, donde se pasan todos los datos a diferentes archivos .csv para su posterior análisis en una herramienta externa que nos permita más precisión.

El *Test #3* tiene los mismos apartados que los tests *#1* y *#2*.

Se ha decidido no pasar el cuadernillo a pdf debido a que no se ven las ejecuciones de los diferentes módulos.

7.1.1.3. Uso

El uso es muy sencillo, se conecta a Google Colab o se inicia un entorno Jupyter y se carga en función de los módulos que se quieran ejecutar y sus dependencias.