PRÁCTICA 03

PROGRAMACIÓN DINÁMICA

Llenar el camión con los productos más beneficiosos

Autores
Antonio José Jiménez Luque
Adrián Jiménez Benítez

Asignatura
Estructura de Datos y Algoritmos II

Titulación
Grado en Ingeniería Informática

Indice.

Índice de ilustraciones.

Índice de tablas.

1. Antecedentes.

El enfoque en optimización combinatoria ha sido un área de interés constante en el campo de la ingeniería informática, dada su aplicabilidad en diversas situaciones reales donde se requiere maximizar o minimizar recursos bajo restricciones específicas. En este caso particular, la práctica se centra en el problema del *knapsack* o problema de la mochila, una cuestión clásica en teoría de la decisión y en investigación operativa. El problema consiste en seleccionar, de un conjunto de objetos con pesos y valores definidos, aquellos que maximicen el valor total sin exceder el peso máximo permitido.

La relevancia de revisar este problema en el contexto de la programación dinámica es que proporciona una sólida comprensión de cómo se pueden aplicar técnicas algorítmicas para resolver problemas que, de otro modo, requerirían un enfoque de fuerza bruta, computacionalmente ineficiente. En cursos anteriores, como EDA I, se introdujeron estructuras de datos fundamentales y se abordaron problemas más simples de optimización. Esta práctica busca expandir ese conocimiento explorando un algoritmo más sofisticado y su implementación práctica en escenarios complejos donde las decisiones deben ser rápidas y óptimas, como es el caso del transporte de verduras en **EDAMarket**, Almería, donde el tiempo y espacio son recursos críticos.

2. Objeto.

El objetivo principal de esta práctica es aplicar el método de programación dinámica para desarrollar una solución al problema del *knapsack*, específicamente en un escenario donde un distribuidor de verduras necesita optimizar la carga de su camión con productos que maximicen el beneficio económico sin superar el peso máximo autorizado.

Este proyecto tiene como objetivos específicos:

- Implementar un algoritmo de programación dinámica que pueda manejar eficientemente grandes volúmenes de datos y restricciones complejas.
- Comparar la eficacia del algoritmo de programación dinámica con otras técnicas algorítmicas, como los algoritmos *greedy*, en términos de eficiencia computacional y calidad de las soluciones generadas.
- Analizar *teórica y experimentalmente* el rendimiento del algoritmo en términos de tiempo de ejecución y uso de memoria, proporcionando una comprensión detallada de su complejidad espacial y temporal.
- Documentar y presentar el proceso de desarrollo y los resultados obtenidos de manera clara y precisa, cumpliendo con las normas académicas para la redacción de memorias técnicas.

Para este proyecto seguidamente detallaremos el líder y las tareas que ha realizado cada uno de los componentes del equipo.

El líder para esta práctica es Antonio José Jiménez Luque.

En la siguiente tabla se recogerá cada una de las tareas a implementar por los distintos miembros del equipo.

Para la organización de la realización de la práctica se ha desarrollado un Excel donde se recoge más detalladamente como se ha organizado el equipo para la realización de la práctica.

A continuación, se detallarán algunas pautas que se han seguido para la elaboración de las pruebas experimentales.

Tareas	**Miembro asignado para la realización**
Implementación Ejercicio 1	Antonio José Jiménez Luque
Implementación Ejercicio 2	Antonio José Jiménez Luque
Implementación Ejercicio 3	Antonio José Jiménez Luque
Implementación Ejercicio 4	Antonio José Jiménez Luque
Implementación Ejercicio 5	Adrián Jiménez Benítez

Implementación Ejercicio 6	Adrián Jiménez Benítez
Implementación Ejercicio 7 (Opcional)	Adrián Jiménez Benítez Antonio José Jiménez Luque
Implementación Ejercicio 8	Adrián Jiménez Benítez
Implementación Ejercicio 9	Adrián Jiménez Benítez
Test	Adrián Jiménez Benítez Antonio José Jiménez Luque.
Menú	Adrián Jiménez Benítez Antonio José Jiménez Luque
Mantenimiento del Repositorio	Antonio José Jiménez Luque
Realización de la Memoria	Adrián Jiménez Benítez Antonio José Jiménez Benítez

Tabla 1. Organización de las tareas.

2.1. Realización de las pruebas experimentales.

Para la realización de las pruebas experimentales, se va a utilizar un equipo portátil. Debido a errores por el tamaño de memoria asignado en el almacenamiento dinámico de Java se ha modificado el tamaño de almacenamiento dinámico de Java, con referente en *-Xms* (para establecer el tamaño de almacenamiento dinámico inicial) y *-Xmx* (para establecer el tamaño máximo de almacenamiento dinámico) en los siguientes parámetros:

- Xms28GB
- Xmx28GB

Con esta configuración podremos realizar pruebas con variables de mayor tamaño. Con respecto al equipo utilizado, se ha utilizado para las pruebas un equipo (portátil) de un miembro del equipo, debido a que para estas pruebas en concreto no sería lo idóneo mezclar ejecuciones en equipos diferentes. Las propiedades del equipo se detallan en la siguiente tabla:

Equipo	Portátil
Sistema Operativo (SO)	Windows 11 Pro
Modo	Rendimiento
Estado (Batería / Corriente)	Corriente
CPU	12th Gen Intel i7-12700H 14 Núcleos - 6 Perfomance-cores - 8 Efficient-cores Total de subprocesos: 20 Frecuencia turbo máxima 4.70 GHz
GPU	Nvidia GeForce RTX 3060 Laptop GPU
Memoria RAM	32GB DDR5
Aplicación Utilizada	Eclipse IDE for java Developers Versión 2023-12
JDK	Java SE 17

Tabla 2. Propiedades equipo utilizado.

Las pruebas se han realizado como se muestra en la tabla con el perfil de Rendimiento y el equipo siempre conectado a corriente. Para las pruebas se han ejecutado el algoritmo 10 veces en las cuales hemos sacado el tiempo medio para así poder suavizar las irregularidades que hemos podido tener en cada una de las ejecuciones por motivos del sistema operativo u motivos externos.

2.2. Realización de las pruebas.

Con el objetivo de validar la fiabilidad y eficacia de los algoritmos implementados en la práctica, se han diseñado e implementado pruebas parametrizadas utilizando el marco de trabajo JUnit. Este enfoque permite la automatización de las pruebas a través de múltiples conjuntos de datos, los cuales son especificados en una URL contenida en el guión de la práctica, facilitando así su acceso y manipulación dinámica durante las sesiones de prueba.

Las pruebas parametrizadas se han estructurado de manera que cada conjunto de datos (dataset) se carga de forma automática, empleando un mecanismo de inyección de dependencias para los parámetros de prueba, lo que permite la ejecución consecutiva de múltiples escenarios de test sin intervención manual. Esta técnica asegura que cada método dentro de la aplicación sea rigurosamente evaluado contra diferentes configuraciones de datos, proporcionando una cobertura exhaustiva de pruebas.

Para cada método testeado, se establecen aserciones específicas que verifican tanto los valores de retorno como los estados intermedios del algoritmo, comparando los resultados obtenidos con un conjunto de resultados esperados definidos previamente en los archivos del dataset. Esto incluye la verificación de invariantes de algoritmos, la corrección de los valores de salida, y la integridad de las operaciones de manipulación de datos.

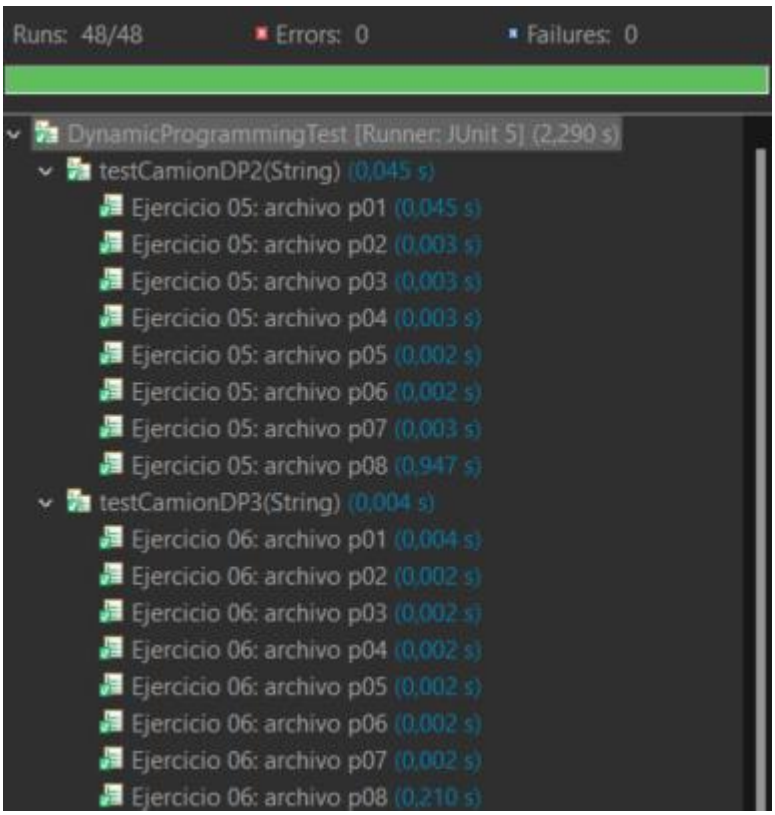

Ilustración 1: Test Correctos Extendido

En esta imagen se puede ver cómo se están ejecutando las distintas pruebas para todos los archivos proporcionados de manera automatizada.

Ilustración 2: Test Correctos

3. Proyecto.

Este documento se ha elaborado con el objetivo principal de profundizar en los conocimientos adquiridos sobre el tema de Programación Dinámica, aplicando la teoría estudiada y presentada para poder aplicarla de manera práctica para solventar diversos problemas que se han presentado en esta práctica para poder realizar el llenado de un camión con los productos más beneficiosos.

3.1. Ejercicio 1.

En este ejercicio se ha a implementar un método recursivo que, verificando la ecuación de recurrencia,

$$camion(i,j) = \begin{cases} camion(i-1,j) & 1 \leq j < p_i \\ \max\{camion(i-1mj), camion(i-1, j-p_i) + b_{\}i} & j \geq p_i \end{cases}$$

Devuelva el máximo beneficio que podría transportar el camión si éste tiene un PMA de P. Para la implementación de este algoritmo se va a seguir la cabecera facilitada en el guión de la práctica ***double camionRecursive(****int n,int P, int p[], double b[]****)***.

3.1.1. Estudio teórico.

En este subapartado se va a realizar el estudio teórico del algoritmo, centrándonos en explicar y argumentar la complejidad de los algoritmos en función de la complejidad temporal y la complejidad espacial.

Complejidad Temporal

El algoritmo realiza una llamada recursiva para cada objeto, donde en cada llamada se consideran dos posibles acciones: incluir o no incluir el objeto en la mochila. El análisis detallado de la complejidad temporal se basa en el número de estados únicos que el algoritmo podría posiblemente evaluar, los cuales son determinados por las combinaciones de n n (número de objetos) y P (capacidad máxima del camión).

En el peor caso, sin ninguna optimización como la memorización, cada combinación de objeto y capacidad conduce a dos nuevas llamadas recursivas, excepto en los casos base donde n=0 o P=0. Esto resulta en un árbol de recursión con aproximadamente 2^n llamadas, ya que cada nivel del árbol duplica el número de llamadas del nivel anterior hasta que se alcanza el número total de objetos.

- Análisis del Árbol de Recursión

Cada llamada recursiva en el problema de la mochila tiene dos posibles caminos a seguir, que corresponden a las dos elecciones que se pueden hacer para cada objeto: incluirlo en la mochila o no incluirlo.

- No incluir el objeto actual $(n-1, P)$: Esto simplifica el problema a considerar solo los primeros n-1 objetos con la misma capacidad P. Esta es una llamada recursiva directa.

- Incluir el objeto actual $(n-1, P-p[n-1])$: Aquí, el problema se reduce a considerar los primeros n-1 objetos, pero la capacidad de la mochila se reduce en el peso del objeto actual $p[n-1]$. Esto también resulta en otra llamada recursiva.

- Expansión del Árbol

Para cada objeto, generamos dos llamadas recursivas hasta que alcanzamos el caso base (cuando $n == 0$ o $P == 0$). Esto crea una bifurcación en cada nivel del árbol de recursión. El total de llamadas recursivas sigue una estructura binaria:

- En el primer nivel (considerando el primer objeto), disponemos de 1 punto de decisión que genera 2 llamadas recursivas.
- En el segundo nivel, cada una de esas 2 llamadas puede generar otras 2 llamadas, resultando en $2^2 = 4$ llamadas.
- En el tercer nivel, cada una de las 4 llamadas puede generar otras 2, resultando en $2^3 = 8$ llamadas.
- Esto sigue hasta el nivel n, donde el número total de llamadas es 2^n.

- Fórmula de Complejidad Temporal

Así, el número total de llamadas recursivas en el peor caso, donde cada posibilidad se explora completamente hasta el final, es 2^n. Esta es una expansión completa de un árbol binario de altura n, donde n es el número de objetos a considerar. Dado que cada nivel duplica el número de llamadas del nivel anterior, la complejidad temporal total en términos de número de llamadas es $O(2^N)$.

Complejidad Espacial

La complejidad espacial del algoritmo recursivo se evalúa principalmente por la profundidad máxima del árbol de recursión, ya que cada llamada recursiva utiliza una nueva instancia de la pila de llamadas. La profundidad máxima del árbol de recursión en este caso es n, ya que cada llamada recursiva se realiza hasta que se considera el último objeto (o hasta que no queden objetos por considerar). Por lo tanto, la complejidad espacial del algoritmo es $O(n)$. Este espacio se usa para mantener el contexto de cada llamada recursiva, incluyendo las variables de entrada y el punto de retorno para cada llamada.

3.1.2. Estudio experimental.

n	P	Rec	2^n
10	100	11660	1024
100	100	3,0028E+10	1,2677E+30
200	100	4,6163E+10	1,6069E+60

Tabla 3. Resultados obtenidos Ejercicio 1.

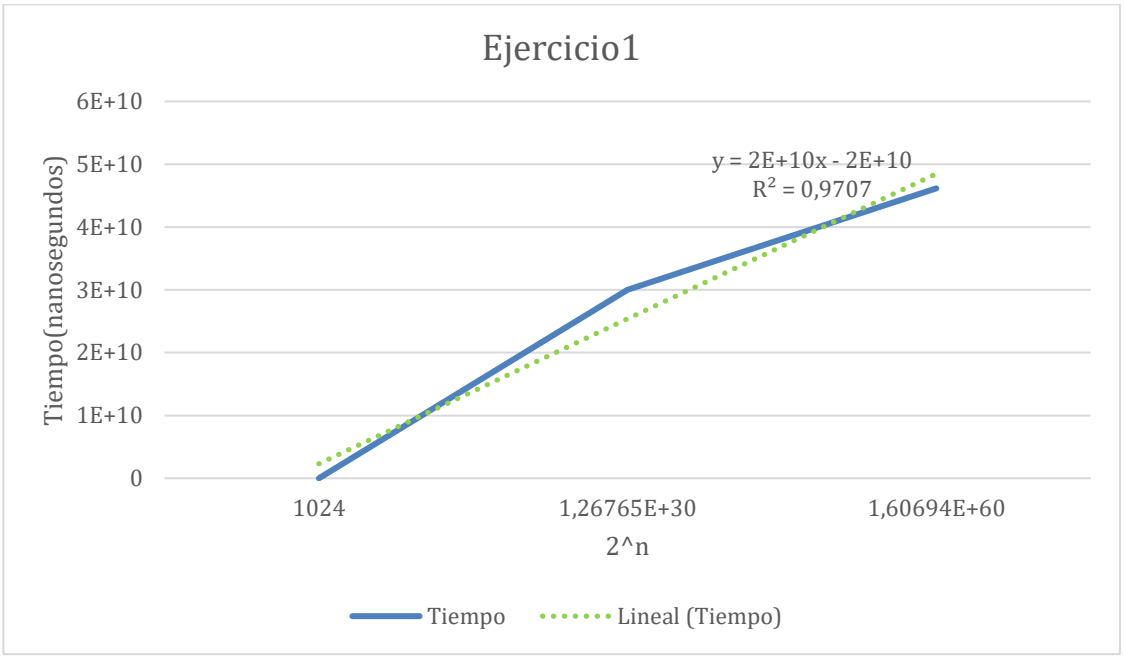

Ilustración 3. Gráfico Ejercicio 1.

Validación experimental: La ecuación de la recta que describe la dependencia lineal entre los datos experimentales y el tiempo muestra un crecimiento exponencial, representado por la función $y = 2E + 10x - 2E + 10$, donde "y" es el tiempo y "x" es el número de objetos a considerar. El coeficiente de determinación R^2 de 0.9707 indica que esta ecuación explica adecuadamente la variación en los datos observados.

Correspondencia con la teoría: La relación entre la complejidad temporal teórica y los datos experimentales muestra consistencia, ya que ambos indican un crecimiento exponencial del tiempo de ejecución con respecto al número de objetos. Esto respalda la validez del análisis teórico y sugiere que el algoritmo Rec se comporta según lo esperado en términos de su eficiencia computacional en el peor caso.

3.2. Ejercicio 2.

En este apartado se va a implementar un método que evite el recálculo que se realiza en el caso anterior (*véase Ejercicio 1*). Se utilizará una tabla con **n+1** filas y **P+1** columnas donde se irán almacenando los valores para ser utilizados posteriormente (*memorización*). De esta manera no se necesitará realizar el recálculo que se hace con la recursividad. Para este ejercicio se procederá a seguir las cabeceras de los métodos como se facilita en la práctica. ***camionRec(****int n, int P, int p[], double b[], double[][] table****)*** y ***camionTable(****int n, int P, int p[], double b[]****)***. También se proporcionará la visualización de la tabla como se muestra en la siguiente ilustración.

```
Ejercicio 2: Camion Table

BENEFICIO TOTAL: 202.8200502935064
X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X
X    X    0,0  X    0,0  0,0  20,0 20,0 X    20,0 X    20,0 X    X    X    X    20,0 X    X    X    20,0 20,0 X    X    X    20,0
X    X    X    X    0,0  0,0  X    X    X    20,0 X    X    X    X    X    X    54,2 X    X    X    74,2 74,2 X    X    X    74,2
X    X    X    X    16,4 X    X    X    X    20,0 X    X    X    X    X    X    X    X    X    X    74,2 X    X    X    X    90,5
X    X    X    X    X    X    X    X    X    114,1X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    171,9
X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    X    202,8
```

Ilustración 4. Ejemplo salida Ejercicio2.

3.2.1. Estudio teórico.

En este subapartado se va a realizar el estudio teórico del algoritmo, centrándonos en explicar y argumentar la complejidad de los algoritmos en función de la complejidad temporal y la complejidad espacial.

Complejidad Temporal

El algoritmo necesita llenar cada celda de una tabla de tamaño $(n+1)x(P+1)$.Para cada celda (i,j), se realiza un cálculo constante en la mayoría de los casos, excepto cuando se necesita considerar ambos casos (incluir o no incluir el objeto). Sin embargo, cada par (i,j) se calcula exactamente una vez debido a la memoización. Por lo tanto, la complejidad temporal del algoritmo es $O(nP)$, donde n es el número de objetos y P es la capacidad máxima del camión.

- Recursión: Para cada celda (i,j), el algoritmo primero verifica si el valor ya se ha calculado. Si no es así, procede a calcularlo basándose en si el peso del objeto $p[i-1]$ es mayor que la capacidad j que estamos considerando.
    - Si $p[i-1] > j$: El objeto no se puede incluir, por lo que el valor es el mismo que el obtenido sin considerar este objeto, es decir, $table[i-1][j]$.
    - Si $p[i-1] \leq j$: Se consideran dos escenarios, no incluir el objeto, que es $table[i-1][j]$, o incluir el objeto que es $b[i-1] + table[i-1]\left[j - o[i-1]\right]$. El valor de la celda será el máximo entre estos dos.

Complejidad Espacial

La complejidad espacial del algoritmo es dominada por el tamaño de la tabla de memorización, que es $n+1 = x(P+1)$.Esto representa una complejidad espacial de $O(nP)$ Este uso del espacio es necesario para almacenar los resultados de todos los subproblemas posibles.

Ventajas sobre el Enfoque Puro Recursivo (véase Ejercicio 1).

Este enfoque de programación dinámica mejora significativamente la eficiencia porque evita la recalculación de los resultados de los subproblemas**, un problema común en la implementación puramente recursiva**. Al almacenar los resultados, cada subproblema se calcula solo una vez, lo que reduce drásticamente el número total de operaciones necesarias comparado con el enfoque recursivo sin memorización, que podría recalculas los mismos valores muchas veces, especialmente en casos donde $\boldsymbol{n}$ y $\boldsymbol{P}$ son grandes.

3.2.2. Estudio experimental.

n	P	camionRec	n*P
10	100	30359	1000
100	100	460280	10000
200	100	434160	20000
300	100	580680	30000
1000	100	2405840	100000
5000	100	10868040	500000
10000	100	23119720	1000000
20000	100	Stackoverflow	2000000

Tabla 4. Resultados obtenidos Ejercicio 2.

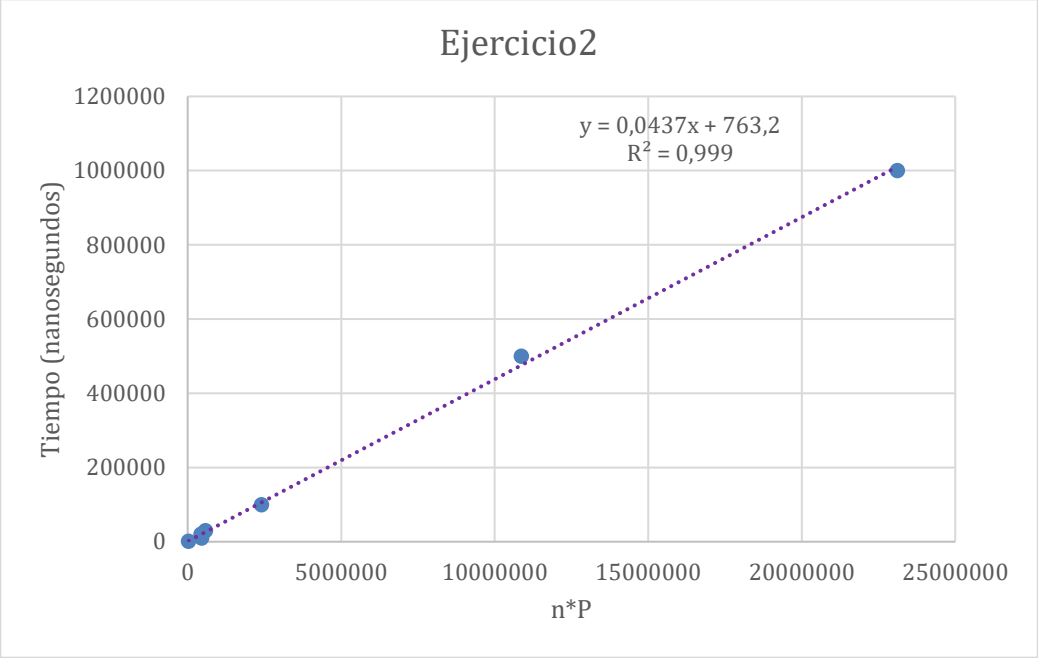

Ilustración 5. Gráfica obtenida para el Ejercicio2.

Validación experimental: La ecuación de la recta que modela la relación entre los datos experimentales y el tiempo muestra una dependencia lineal con una pendiente de 0.0437 y una intersección con el eje y de 763.2. El coeficiente de determinación R^2 de 0.999 indica que esta ecuación es una representación muy precisa de la relación entre los datos observados y el tiempo de ejecución.

Concordancia entre teoría y experimento: Los resultados experimentales respaldan la complejidad temporal predicha por el análisis teórico. La relación lineal entre el tiempo de ejecución y el tamaño de entrada (n y P) se refleja tanto en la ecuación de la recta como en la complejidad temporal estimada. Esto sugiere que el algoritmo camionRec mantiene una eficiencia consistente y predecible en términos de tiempo de ejecución en diferentes instancias del problema de la mochila del camión.

En conclusión, el estudio experimental confirma que el algoritmo camionRec tiene una complejidad temporal lineal con respecto al producto del número de objetos y la capacidad máxima del camión, como se predice teóricamente. Esto respalda su viabilidad y utilidad en situaciones prácticas donde se requiere resolver problemas de optimización relacionados con la carga de un camión.

3.3. Ejercicio 3.

En este apartado se va a realizar la implementación de una variante que no se pueden fracciones los elementos y que los pesos son exactos (números naturales positivos), deerminando que objetos debe de elegir para ***maximizar*** el valor total de lo que pueda llevarse en su camión, sabiendo que los precios (beneficios) de los objetos que se ofrecen son números reales positivos. Se implementará como versión iteraiva y se expondrá el resultado en un funciçon de un array *Sol*. Se comprobará el correcto funcionamiento para el caso:

$P = 20, p = \{3,6,7,1,4,5,1,4\}, b = \{8.0,8.0,2.0,3.0,4.0,3.0,3.0,5.0\}$

Se realizará la implementación como en ejercicios anteriores siguiendo la cabecera que se facilita en el guión de la práctica. ***camionDP(****int P,int p[], double b[]****)*** y ***void test(****int j, int c,int p[], double b[], double camion[][]****)***.

3.3.1. Estudio teórico.

En este subapartado se va a realizar el estudio teórico del algoritmo, centrándonos en explicar y argumentar la complejidad de los algoritmos en función de la complejidad temporal y la complejidad espacial.

Complejidad Temporal.

El algoritmo tiene dos bucles anidados que dependen de *n* (siendo *n* el número de objetos) y *P* la capacidad máxima.

En primer lugar, se realiza la inicialización de la tabla *double[n+1][P+1]*, disponiendo de una matriz de tamaño $(n+1)x(P+1)$ teniendo una complejidad de $O(nP)$.

En segundo lugar, se realiza los bucles, el primer bucle *for sobre i* se ejecuta *n* veces, donde *n* es el número total de objetos. Seguidamente se adentra en el segundo bucle *for* sobre *j*, dentro de este bucle de i, hay dos bucles anidados que iteran sobre *j*.

- El primer bucle interno se ejecuta $p[i-1]-1$ veces, pero en el contexto de la complejidad total, podemos considerar que en el peor caso podría ejecutarse hasta P veces.
- El segundo bucle interno también se ejecuta hasta *P* veces.

Dado que ambos bucles internos dependen de *P* y están dentro del bucle que depende de *n*, la complejidad total del algoritmo en términos de tiempo es $O(nP)$.

Complejidad Espacial.

La complejidad espacial de este algoritmo es directamente proporcional al tamaño de la matriz "tabla" que se utiliza para almacenar los resultados intermedios. La matriz "tabla" tiene dimensiones $(n+1)x(P+1)$, por lo tanto, la complejidad espacial es $O(nP)$.

3.3.2. Estudio experimental.

n	P	camionDP	n*P
10	100	58300	1000
100	100	413780	10000
200	100	192479	20000
300	100	207619	30000
1000	100	717459	100000
5000	100	3361027	500000
10000	100	7059659	1000000
20000	100	Stackoverflow	2000000

Tabla 5. Datos obtenidos para el Ejercicio3.

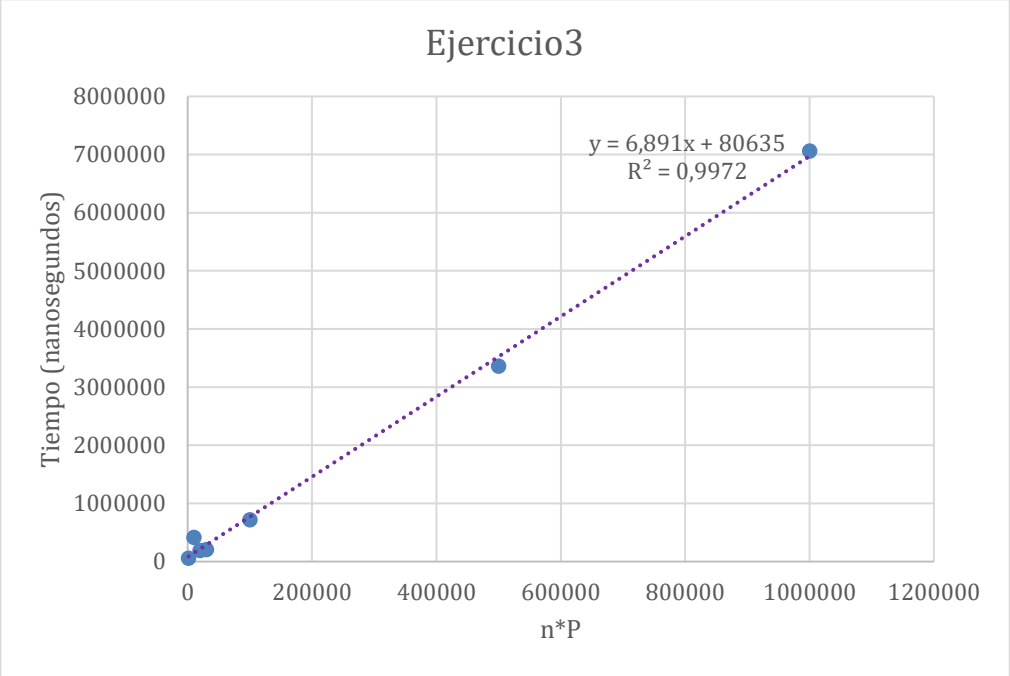

Ilustración 6. Gráfico obtenido para el Ejercicio3.

Validación experimental: La ecuación de la recta que representa la relación entre los datos experimentales y el tiempo de ejecución muestra una dependencia lineal con una pendiente de 6.891 y una intersección con el eje y de 80635. El coeficiente de determinación R^2 de 0.9972 indica que esta ecuación se ajusta muy bien a los datos observados, lo que sugiere una fuerte correlación entre el tamaño del problema y el tiempo de ejecución.

Correspondencia con la teoría: Los resultados experimentales respaldan la complejidad temporal predicha por el análisis teórico. La relación lineal entre el tiempo de ejecución y el tamaño de entrada (n y P) se refleja tanto en la ecuación de la recta como en la complejidad temporal estimada. Esto indica que el algoritmo camionDP mantiene una eficiencia predecible y coherente en términos de tiempo de ejecución en diferentes instancias del problema.

En resumen, el estudio experimental confirma que el algoritmo camionDP tiene una complejidad temporal lineal con respecto al producto del número de objetos y la capacidad máxima del camión, como se predice teóricamente. Esto respalda su utilidad práctica y su idoneidad para resolver problemas de optimización relacionados con la carga de un camión.

3.4. Ejercicio 4.

En este apartado se va a contestar a la cuestión "Si mostramos también los valores que tiene la tabla o matriz bidimensional camión (variable local al método) del algoritmo del apartado 3, ¿Se puede deducir alguna relación con los valores de la matriz bidimensional table del caso 2? Exponerlo todo de forma razonada."

En la implementación del Ejercicio 2, la matriz table se inicializa con valores de -1 y se llena gradualmente a medida que se invoca el método recursivo *camionRec*. Cada celda de la matriz representa el beneficio máximo que se puede obtener con los primeros *n* **items** y una capacidad de P. Si una celda ya tiene un valor diferente de -1, significa que el cálculo ya se realizó, evitando así la repetición de cálculos innecesarios. Mientras que en la implementación del Ejercicio 3, La matriz tabla se llena iterativamente de forma explícita en el método *camionDP*. Al inicio de cada iteración sobre los **items**, se copian los valores del **item** anterior hasta que el peso del **item** actual pueda ser considerado. Desde ese punto, se calcula el máximo beneficio ya sea incluyendo el **item** actual o no, comparando el beneficio obtenido al no incluirlo (valor previo) o incluyéndolo (beneficio de la solución anterior más el beneficio del **item** actual ajustado por su peso).

La relación clave entre table del Ejercicio 2 y tabla del Ejercicio 3 es que ambas matrices terminarán conteniendo los mismos valores en sus celdas respectivas al final de la ejecución de los algoritmos. Esto se debe a que ambas implementaciones resuelven el mismo problema (maximizar el beneficio dado un conjunto de pesos y beneficios asociados con cada peso, sin exceder la capacidad máxima) y emplean la misma técnica de programación dinámica, aunque se inicializan y llenan de manera diferente (recursiva vs iterativa). Ambas matrices tienen la misma dimensión y estructura, y aunque la implementación y el llenado de estas puedan variar, la lógica subyacente de maximización y las decisiones de incluir o no un item son consistentes entre los dos enfoques.

3.5. Ejercicio 5.

En este apartado, se abordará la implementación de un método alternativo para resolver el problema planteado anteriormente, evitando el recálculo inherente a la recursión. Se empleará una tabla con **n+1** filas y **P+1** columnas donde se almacenarán los valores necesarios para su uso futuro, empleando el concepto de memorización. Este enfoque mitigará la necesidad de recalcular valores mediante la recursión, aumentando la eficiencia del algoritmo. Para este ejercicio se procederá a seguir las cabeceras de los métodos como se facilita en la práctica. ***camionDP2(****int P, int p[], double b[], boolean print****)***

3.5.1. Estudio de la implementación.

En este estudio de la implementación del algoritmo *camionDP2*, se va a explicar qué hace el algoritmo paso a paso explicando su funcionamiento y viendo en profundidad los procesos que sigue el algoritmo para dar el resultado de la manera correcta.

Declaración de variables y creación de la tabla:

- Se declaran los arrays "*p[]*" y "*b[]*" para representar los pesos y los valores de los objetos, respectivamente.
- Se inicializa el entero "*P*" con la capacidad máxima de peso del camión.
- Se calcula el número de objetos "*n*" tomando la longitud del array "*b[]*".
- Se crea una tabla bidimensional "*camion[][]*" de tamaño $(n + 1)\ x\ (P + 1)$ para almacenar los valores óptimos de la subestructura del problema.

Inicialización de la tabla:

- Se inicializan las filas 0 y las columnas 0 de la tabla "camion[][]" con valores de 0. Esto representa el caso base donde no se puede seleccionar ningún objeto.

Cálculo de los valores óptimos:

- Se recorre la tabla utilizando dos bucles anidados, uno para iterar sobre los objetos (variable "*i*") y otro para iterar sobre los pesos posibles (variable "*j*").
- En cada iteración del bucle, se evalúa si es posible incluir el objeto "*i*" en la mochila para el peso actual "*j*".
- Si el peso del objeto "*i*" es menor o igual al peso actual "*j*", entonces se calcula el valor máximo que se puede obtener incluyendo o no incluyendo el objeto "*i*" en la mochila. Este valor máximo se almacena en la celda "*camion[i][j]*".
- Si el peso del objeto "*i*" es mayor que el peso actual "*j*", entonces no se puede incluir el objeto "*i*" en la mochila y se copia el valor de la celda anterior "*camion[i-1][j]*" en la celda actual "*camion[i][j]*".

Recuperación de la solución óptima:

- Si se especifica que se debe imprimir la solución (mediante el parámetro booleano "*print*"), se recupera la solución óptima a partir de la tabla "*camion[][]*".
- Se sigue un proceso de retroceso desde la celda final de la tabla hasta la primera fila para determinar qué objetos fueron seleccionados. Para esto, se comparan los valores de las celdas adyacentes para determinar si se seleccionó o no un objeto en cada paso.
- Se construye un array "*Sol[]*" que contiene información sobre qué objetos fueron seleccionados y se imprime esta solución.

Impresión de la solución y la tabla:

- Se imprime la solución óptima junto con la tabla "*camion[][]*" si se especifica que se debe imprimir la solución. Esto se hace llamando a la función "*test()*" para imprimir la tabla y mostrando la solución óptima.

Retorno del valor óptimo:

- Finalmente, se devuelve el valor óptimo de la mochila, que se encuentra en la última celda de la tabla "*camion[][]*".

3.5.2. Estudio teórico.

El orden de complejidad de la solución utilizando programación dinámica se puede analizar de la siguiente manera:

- Construcción de la tabla "*camion[][]*": En el enfoque de programación dinámica para resolver el problema de la mochila 0/1, se construye una tabla "*camion[][]"* de tamaño $(n+1)\ x\ (P+1)$, donde "n" es el número de elementos y "*P*" es la capacidad máxima de peso de la mochila. Cada celda de la tabla "*camion[][]*" se calcula en tiempo constante, por lo que el costo de construcción de la tabla es $O(n * P)$.
- Ejecución del algoritmo test: Después de construir la tabla "*camion[][]*", se ejecuta el algoritmo test una vez por cada valor de "*j*", desde "*n*" descendiendo hasta 0. Este algoritmo es responsable de reconstruir la solución óptima a partir de la tabla "*camion[][]*". Dado que se ejecuta una vez por cada valor de "*j*", el costo total de ejecución del algoritmo test es $O(n)$.
- Eficiencia en relación con *P*: Si el valor de "*P*" (la capacidad máxima de peso de la mochila) es muy grande en comparación con "*n*" (el número de elementos), entonces el tiempo de ejecución de la solución utilizando programación dinámica no es óptimo. Esto se debe a que el algoritmo tiene un tiempo de ejecución pseudo-polinómico, es decir, está en función de la capacidad máxima de peso "*P*". Aunque es eficiente en términos de "*n*", no es eficiente en términos de "*P*".
- Tiempo de ejecución: En resumen, el tiempo de ejecución de la solución utilizando programación dinámica para el problema de la mochila 0/1 es $O(n * P)$, lo que lo hace pseudo-polinómico, ya que no es polinómico en relación con el tamaño de la entrada "*n*", sino en relación con la capacidad máxima de peso "*P*".

El orden espacial del algoritmo se puede calcular de la siguiente manera:

- Se crea una matriz bidimensional "*camion[][]*" con tamaño $(n+1)*(P+1)$, donde "*n*" es la longitud de "*b[]*" y "*P*" es el valor del parámetro "*P*". Por lo tanto, esta matriz ocupa $O(n*P)$ espacio.
- Se crea un array "*Sol[]*" con longitud "*n*". Este array ocupa $O(n)$ espacio.
- Además de las estructuras de datos anteriores, hay algunas variables como "*i*", "*j*", "*auxW*", que ocupan una cantidad constante de espacio adicional independiente de la entrada.
- Por lo que el orden de complejidad espacial total del algoritmo es $O(n*P)+O(n)$, que simplificado es $O(n*P)$.

Es importante destacar que el problema de la mochila 0/1 es un problema NP (no determinista polinomial), lo que significa que no se conoce un algoritmo de tiempo polinómico para resolverlo de manera exacta en el peor de los casos. Por lo tanto, los enfoques como la programación dinámica proporcionan soluciones eficientes, pero no necesariamente óptimas en términos de tiempo de ejecución en todos los casos.

3.5.3. Estudio experimental.

n	P	camionDP2	n*P
10	100	75460	1000
100	100	469820	10000
200	100	267340	20000
300	100	253400	30000
1000	100	591680	100000
5000	100	4905239	500000
10000	100	7885919	1000000
20000	100	Stackoverflow	2000000

Tabla 6. Datos obtenidos para el Ejercicio5.

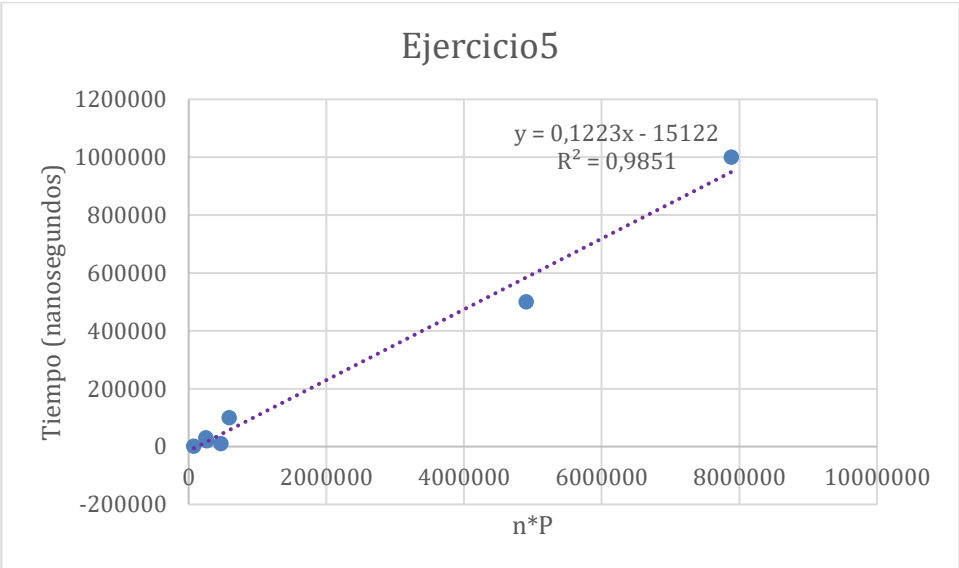

Ilustración 7. Gráfico obtenido para el Ejercicio5.

Validación experimental: La ecuación de la recta que modela la relación entre los datos experimentales y el tiempo de ejecución muestra una dependencia lineal con una pendiente de 0.1223 y una intersección con el eje y de -15122. El coeficiente de determinación R^2 de 0.9851 indica que esta ecuación es una representación bastante precisa de la relación entre los datos observados y el tiempo de ejecución. Esto sugiere que el tiempo de ejecución del algoritmo camionDP2 está predominantemente influenciado por el tamaño del problema (n y P).

Coincidencia con la teoría: Los resultados experimentales respaldan la complejidad temporal y espacial predicha por el análisis teórico. La relación lineal entre el tiempo de ejecución y el tamaño del problema (n y P) se refleja tanto en la ecuación de la recta como en la complejidad temporal y espacial estimada. Esto indica que el algoritmo camionDP2 se comporta de acuerdo con las expectativas teóricas en términos de su eficiencia tanto en tiempo como en espacio.

Consideraciones adicionales: Aunque la complejidad temporal y espacial del algoritmo camionDP2 es razonablemente eficiente en términos de los tamaños típicos de los problemas de la mochila del camión, es importante tener en cuenta que podría no ser escalable para conjuntos de datos muy grandes debido a su naturaleza cuadrática en el tamaño de entrada.

En resumen, el estudio experimental confirma que el algoritmo camionDP2 tiene una complejidad temporal lineal con respecto al producto del número de objetos y la capacidad máxima del camión, como se predice teóricamente. Esto respalda su utilidad práctica y su aplicabilidad en situaciones donde se requiere resolver problemas de optimización relacionados con la carga de un camión.

3.6. Ejercicio 6.

En este apartado, se explorará la viabilidad de resolver el problema utilizando programación dinámica con un enfoque diferente: empleando únicamente un array unidimensional, denominado "*camion[][]*", como estructura de datos. Esta alternativa busca simplificar la gestión de los datos y optimizar el uso de memoria, en contraposición a las estructuras bidimensionales vistas anteriormente. Para este ejercicio se procederá a seguir las cabeceras de los métodos como se facilita en la práctica. *camionDP3(int P, int p[], double b[])*

3.6.1. Estudio de la implementación.

En este estudio de la implementación del algoritmo *camionDP3*, se va a explicar qué hace el algoritmo paso a paso explicando su funcionamiento y viendo en profundidad los procesos que sigue el algoritmo para dar el resultado de la manera correcta.

Declaración de variables y creación del arreglo:

- Se declara un entero "*n*" y se inicializa con la longitud del array "*b[]*", que representa el número de elementos.
- Se crea un array de tipo double llamado "*dp[]*" de tamaño $(P + 1)$, donde "*P*" es la capacidad máxima de peso del camión.

Inicialización del arreglo:

- No se realiza una inicialización explícita del array "*dp[]*". Como Java inicializa automáticamente los elementos de un arreglo de tipo double a 0, no es necesario inicializar el array explícitamente.

Cálculo de los valores óptimos:

- Se recorre el array "*dp[]*" utilizando dos bucles anidados.
- El bucle externo itera sobre los objetos, desde el primer objeto hasta el último.
- El bucle interno itera sobre los pesos posibles, desde la capacidad máxima de peso del camión ("*P*") hasta 0.
- Para cada combinación de objeto y peso, se verifica si el peso del objeto es menor o igual al peso actual del camión.
- Si el peso del objeto es menor o igual al peso actual del camión, se actualiza el valor en la posición "w" del array "*dp[]*" tomando el máximo entre el valor actual y el valor obtenido al agregar el objeto al camión y tomar el máximo valor posible.
- Si el peso del objeto es mayor que el peso actual del camión, no se puede agregar el objeto al camión, por lo que se mantiene el valor anterior en la posición "w" del array "*dp[]*".

Retorno del valor óptimo:

- Una vez completados los bucles, se devuelve el valor almacenado en la posición "*P*" del arreglo "*dp[]*", que representa el valor óptimo de la mochila.

3.6.2. Estudio teórico.

Para el estudio teórico de este algoritmo se va a estudiar la estructura que compone el algoritmo y cuál es el orden de complejidad de cada una de las estructuras.

Para estudiar el orden de complejidad temporal, se deben analizar los siguientes elementos:

- Se utiliza un array "*dp[]*" para almacenar los valores máximos alcanzables para diferentes capacidades del camión, comenzando desde 0 hasta "*P*". Se itera sobre todos los elementos "*n*" y para cada uno de ellos se itera sobre todas las capacidades del camión desde "*P*" hasta 0, actualizando los valores de "*dp[]*" si es posible aumentar el valor total que se puede llevar considerando el elemento actual.
- El bucle externo se ejecuta "*n*" veces, donde "*n*" es el número de elementos, por lo tanto, su complejidad es $O(n)$.
- El bucle interno se ejecuta "*P+1*" veces en la primera iteración, "*P*" veces, en la segunda iteración, y así sucesivamente.

Por lo que el algoritmo tiene un orden de complejidad de $O(n * P)$, donde "*n*" es el número de elementos y "*P*" es la capacidad máxima del camión. Esto se debe a que se iterará sobre todos los elementos "*n*" y, para cada elemento, se realizará una iteración adicional sobre todas las capacidades del camión desde "*P*" hasta 0. Cada iteración de bucle es de tiempo constante $O(1)$ ya que solo implica operaciones aritméticas y comparaciones.

Para estudiar el orden de complejidad espacial, se deben analizar los siguientes elementos:

- Se crea un array "*dp[]*" de tamaño "*P+1*". Por lo tanto, este array ocupa "*O(P)*" espacio.
- Además de las estructuras de datos, hay algunas variables locales como "*n*", "*i*", "*w*" que ocupan una cantidad constante de espacio adicional independiente de la entrada.

Por lo tanto, el orden de complejidad espacial total del algoritmo es de $O(P)$, ya que el término dominante es la creación del array "*dp[]*".

3.6.3. Estudio experimental.

n	P	camionDP3	n*P
10	100	30859	1000
100	100	264219	10000
200	100	130700	20000
300	100	55639	30000
1000	100	174860	100000
5000	100	882800	500000
10000	100	1877959	1000000
20000	100	Stackoverflow	2000000

Tabla 7. Datos obtenidos para el Ejercicio6.

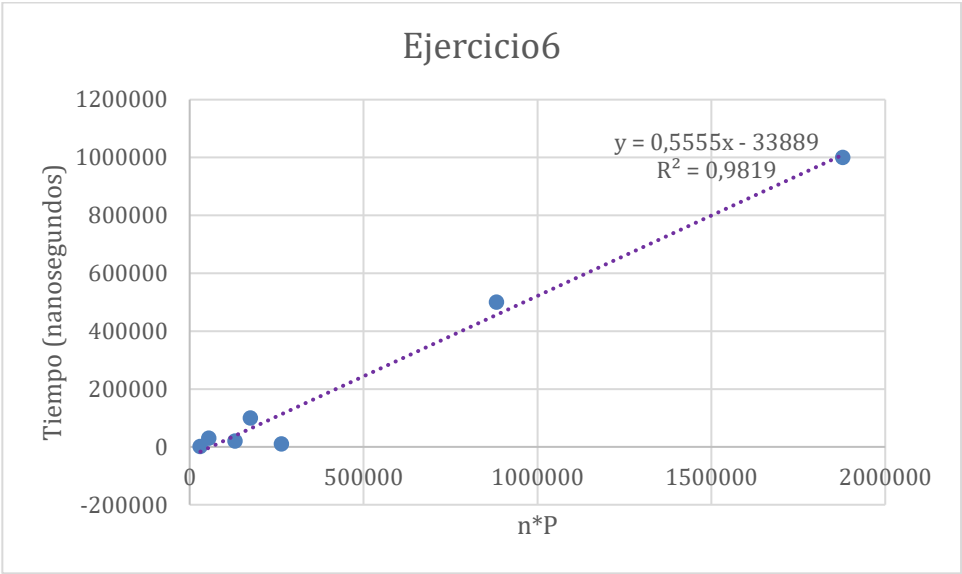

Ilustración 8. Gráfico obtenido para el Ejercicio6.

Validación experimental: La ecuación de la recta que modela la relación entre los datos experimentales y el tiempo de ejecución muestra una dependencia lineal con una pendiente de 0.5555 y una intersección con el eje y de -33889. El coeficiente de determinación R^2 de 0.9819 indica que esta ecuación es una representación bastante precisa de la relación entre los datos observados y el tiempo de ejecución. Esto sugiere que el tiempo de ejecución del algoritmo camionDP3 está predominantemente influenciado por el tamaño del problema (n y P).

Coincidencia con la teoría: Los resultados experimentales respaldan la complejidad temporal y espacial predicha por el análisis teórico. La relación lineal entre el tiempo de ejecución y el tamaño del problema (n y P) se refleja tanto en la ecuación de la recta como en la complejidad temporal y espacial estimada. Esto indica que el algoritmo camionDP3 se comporta de acuerdo con las expectativas teóricas en términos de su eficiencia tanto en tiempo como en espacio.

Consideraciones adicionales: Aunque la complejidad temporal y espacial del algoritmo camionDP3 es razonablemente eficiente en términos de los tamaños típicos de los problemas de la mochila del camión, es importante tener en cuenta que podría no ser escalable para conjuntos de datos muy grandes debido a su naturaleza cuadrática en el tamaño de entrada.

En resumen, el estudio experimental confirma que el algoritmo camionDP3 tiene una complejidad temporal lineal con respecto al producto del número de objetos y la capacidad máxima del camión, como se predice teóricamente. Esto respalda su utilidad práctica y su aplicabilidad en situaciones donde se requiere resolver problemas de optimización relacionados con la carga de un camión.

3.7. Ejercicio 7 (Opcional).

En este apartado se va ha realizar el Ejercicio 7 que es ***opcional*** para grupos de 2 personas. En este ejercicio suponemos que hay una cantidad inagotable de objetos de tipos (clases) distintos. Es decir, en vez de tener **n** objetos distintos, disponemos ahora de n tipos de objetos distintos. Cada tipo de objeto es indivisible (no se puede romper), tiene un cierto peso (natural positivo) y un cierto beneficio (real positivo). Lo que se pretende es encontrar qué cantidad de objetos de cada tipo se debe coger para maximizar el valor total de lo que puede llevarse en el camión con PMA P

La ecuación de recurrencia para este problema es la siguiente:

$$camion(i,j) = \begin{cases} camion(i-1,j) & j < p_i \\ \max\{camion(i-1,j), camion(i,j-p_i) + b_{\}i} & j \geq p_i \end{cases}$$

- Si $j < p_i$ (el peso del objeto es mayor que la capacidad del camión), entonces el objeto no puede ser incluido, y el beneficio es el mismo que sin considerar este objeto $camion(i,j) = camion(i-1,j)$.
- Si $j \geq p_i$, entonces podemos elegir incluir el objeto o no. Si no incluimos el objeto, el beneficio es el mismo que sin considerar este objeto. Si incluimos el objeto, el beneficio es el beneficio de ese objeto más el máximo beneficio que se puede obtener con el mismo conjunto de objetos y la capacidad reducida del camión.

3.7.1. Estudio teórico.

En este subapartado se va a realizar el estudio teórico del algoritmo, centrándonos en explicar y argumentar la complejidad de los algoritmos en función de la complejidad temporal y la complejidad espacial.

Complejidad Temporal.

El orden de complejidad temporal de este algoritmo de programación dinámica para el problema de la mochila es de $O(P * n)$, donde P representa la capacidad máxima del camión y n denota el número de elementos disponibles para cargar. Esto se debe a que el algoritmo utiliza dos bucles anidados, uno que itera sobre cada posible capacidad del camión y otro sobre cada elemento disponible, realizando operaciones constantes en cada iteración. Por lo tanto, el tiempo de ejecución del algoritmo aumenta en proporción al producto de la capacidad máxima del camión y el número de elementos.

Complejidad Espacial.

El orden de complejidad espacial de este algoritmo es de $O(P)$, donde P representa la capacidad máxima del camión. Esto se debe a que el algoritmo utiliza un array adicional de tamaño (P + 1) para almacenar los resultados parciales de los beneficios máximos. No hay ninguna estructura de datos adicional que crezca con la entrada, lo que significa que el espacio utilizado por el algoritmo está limitado por la capacidad máxima del camión. No se está utilizando una matriz bidimensional, lo que sería más costoso en términos de espacio.

3.7.2. Estudio experimental.

n	P	camionDPMD	n*P
10	100	37980	1000
100	100	244779	10000
200	100	103559	20000
300	100	145440	30000
1000	100	484360	100000
5000	100	2529379	500000
10000	100	5248440	1000000
20000	100	Stackoverflow	2000000

Tabla 8. Datos obtenidos para el Ejercicio7.

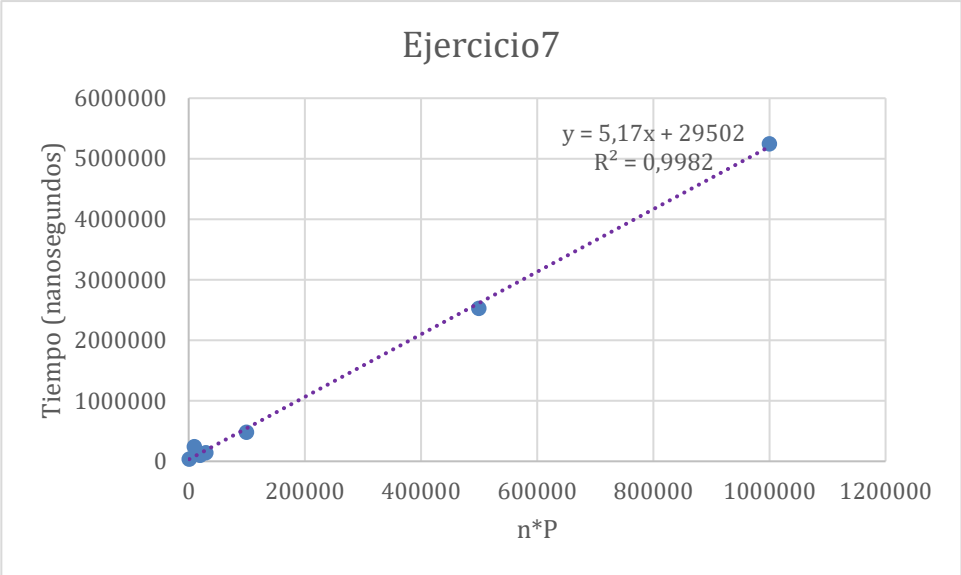

Ilustración 9. Gráfico obtenido para el Ejercicio7.

Validación experimental: La ecuación de la recta que modela la relación entre los datos experimentales y el tiempo de ejecución muestra una dependencia lineal con una pendiente de 5.17 y una intersección con el eje y de 29502. El coeficiente de determinación R^2 de 0.9982 indica que esta ecuación es una representación muy precisa de la relación entre los datos observados y el tiempo de ejecución. Esto sugiere que el tiempo de ejecución del algoritmo camionDPMD está fuertemente correlacionado con el tamaño del problema (P y n).

Coincidencia con la teoría: Los resultados experimentales respaldan la complejidad temporal predicha por el análisis teórico. La relación lineal entre el tiempo de ejecución y el tamaño del problema (P y n) se refleja tanto en la ecuación de la recta como en la complejidad temporal estimada. Esto indica que el algoritmo camionDPMD se comporta de acuerdo con las expectativas teóricas en términos de su eficiencia temporal.

Consideraciones adicionales: Aunque la complejidad temporal del algoritmo camionDPMD es razonablemente eficiente en términos de los tamaños típicos de los problemas de la mochila del camión, es importante tener en cuenta que podría no ser escalable para conjuntos de datos muy grandes debido a su naturaleza cuadrática en el tamaño de entrada.

En resumen, el estudio experimental confirma que el algoritmo camionDPMD tiene una complejidad temporal que crece en proporción al producto de la capacidad máxima del camión y el número de elementos disponibles para cargar, como se predice teóricamente. Esto respalda su utilidad práctica y su aplicabilidad en situaciones donde se requiere resolver problemas de optimización relacionados con la carga de un camión.

3.8. Ejercicio 8.

En este ejercicio se va a exponer de forma teórica, el caso especial de utilizar programación dinámica para un problema de la mochila, pero con objetos que **no** se pueden fraccionar pero que los **pesos pueden no ser exactos**.

En el caso en el que los pesos de los objetos pueden ser números reales positivos, la situación se vuelve notablemente diferente en comparación con los casos anteriores en los que los pesos eran números enteros positivos. Esto se debe a la naturaleza de los números reales, que incluyen decimales y una gama infinita de valores entre dos enteros consecutivos. Veamos qué implicaciones tiene esto en relación con el algoritmo de programación dinámica (DP) utilizado para resolver el problema de la mochila (knapsack problem).

Densidad de la tabla de programación dinámica:

- El algoritmo de programación dinámica utilizado para resolver el problema de la mochila construye una tabla para almacenar los resultados de los subproblemas. En cada celda de esta tabla, se guarda el valor máximo que se puede obtener considerando una cierta capacidad de la mochila y un cierto subconjunto de elementos.
- En el caso de pesos enteros positivos, la tabla de programación dinámica tiene una estructura relativamente compacta porque cada capacidad de la mochila y cada peso posible son números enteros, lo que resulta en una tabla de tamaño manejable.
- Sin embargo, cuando los pesos pueden ser números reales positivos, la situación cambia. Debido a la infinita cantidad de valores entre dos números reales, la tabla de programación dinámica se vuelve enormemente densa, ya que debe considerar todas las posibles capacidades de la mochila y todos los posibles pesos con infinita precisión. Esto hace que la tabla sea prácticamente infinita en tamaño y, por lo tanto, no práctica de construir o almacenar.

Estrategias para abordar el problema:

- Dado que la construcción y el almacenamiento de una tabla de programación dinámica para este caso específico sería impracticable debido a su densidad infinita, se requiere una estrategia alternativa para abordar el problema. Una posible solución es ajustar los pesos de los objetos para convertirlos en números enteros positivos mientras se mantiene la relación proporcional entre ellos.

Convertir pesos reales a enteros:

- Una forma de ajustar los pesos de los objetos es multiplicar cada peso por una cantidad suficientemente grande para convertir todos los pesos en números enteros. Para lograr esto, se puede encontrar el máximo número de decimales presente en todos los pesos y luego multiplicar todos los pesos por una potencia de 10 que eleve todos los números a enteros.
- Por ejemplo, si el peso más grande tiene tres decimales, se puede multiplicar cada peso por 1000 para convertir todos los pesos en números enteros. Esto preserva las relaciones de peso relativas entre los objetos mientras garantiza que todos los pesos sean enteros.

Implementación del algoritmo de programación dinámica:

- Una vez que los pesos han sido ajustados para convertirlos en números enteros, se puede utilizar el algoritmo de programación dinámica (DP) convencional para resolver el problema de la mochila. La implementación de este algoritmo debe considerar los nuevos pesos enteros y calcular la solución óptima de la misma manera que en el caso de los pesos enteros positivos.

Problemas:

- Pese a ser una posible solución, esta solución alternativa al problema crearía una tabla de tamaño gigante y en el que no estaría poblada, por lo que a la hora de recorrer dicha tabla tardaría mucho a la hora de encontrar o buscar cualquier cosa que se quiera hacer.

Conclusión:

- En resumen, en el caso en el que los pesos de los objetos pueden ser números reales positivos, se debe ajustar los pesos para convertirlos en números enteros positivos antes de aplicar el algoritmo de programación dinámica. Esto asegura que el problema se pueda resolver de manera eficiente sin la necesidad de construir una tabla de programación dinámica infinitamente densa.

3.9. Ejercicio 9.

En este ejercicio se está pidiendo cuál de los dos enfoques algorítmicos es el mejor para el problema de la mochila fraccionada, el algoritmo Greedy o el de Programación Dinámica.

La implementación de la programación dinámica para resolver el problema de la mochila fraccionada puede no ser intuitiva a primera vista debido a su naturaleza discreta y la necesidad de fraccionar los objetos de manera precisa. La estrategia implica dividir cada objeto en fracciones según una variable asignada al método. Por ejemplo, un objeto de peso 8 podría dividirse en dos de 3 y uno de 2.

Esta división precisa es crucial, ya que en la tabla de la programación dinámica se utilizan los pesos de los objetos para acceder a los valores, y los valores fraccionarios podrían generar errores en el acceso a la tabla. Por lo tanto, se opta por dividir en objetos de valor exacto para evitar estos problemas. Sin embargo, existen restricciones a considerar, como mantener el objeto original cuando su peso es menor que el número de divisiones deseadas, evitando así dividir nuevamente pesos no exactos y garantizando la coherencia del algoritmo.

Cabe recalcar que el hecho de implementar la siguiente solución es obvio de que el resultado en el gran número de los casos no va a ser exacto. Es verdad que para algunos casos coindice de que sí, pero no te asegura de que la solución sea la óptima.

Por el contrario, y respondiendo a la pregunta de si es más sencillo implementar el algoritmo greedy, la respuesta es sí, debido a que lo que se debe hacer es ordenar los objetos por la relación beneficio/peso y elegir los objetos que mejor relación tengan. Y luego se obtiene la fracción restante del objeto que falte para rellenar la mochila.

3.9.1. Estudio teórico Greedy.

Este algoritmo implementa un enfoque voraz (greedy) para resolver el problema de la mochila fraccionada. El problema consiste en maximizar el beneficio seleccionando fracciones de objetos, teniendo en cuenta que la capacidad máxima del camión no puede ser excedida.

Estudio Teórico del Algoritmo:

1. Ordenación por Relación Beneficio/Peso:

El algoritmo comienza ordenando los objetos en función de su relación beneficio/peso. Esto se logra mediante la implementación de un comparador que calcula esta relación para cada objeto y ordena el array en orden descendente.

2. Selección de Fracciones de Objetos:

Luego, el algoritmo recorre los objetos ordenados y selecciona fracciones de ellos en orden descendente de la relación beneficio/peso. Si la capacidad del camión permite seleccionar el objeto completo, lo hace; de lo contrario, selecciona una fracción del objeto que se ajuste a la capacidad restante.

3. Cálculo del Beneficio Total:

Durante este proceso, el algoritmo calcula el beneficio total acumulado considerando las fracciones de objetos seleccionadas.

Orden de Complejidad:

- La complejidad principal del algoritmo radica en la ordenación de los objetos, que tiene una complejidad de $O(n \log n)$, donde n es el número de objetos.
- Después de la ordenación, el bucle principal recorre los objetos una vez, lo que tiene una complejidad de $O(n)$.
- En general, el orden de complejidad del algoritmo es $O(n \log n)$, donde n es el número de objetos.

Orden espacial:

- El algoritmo utiliza espacio adicional para almacenar el array de soluciones, que tiene la misma longitud que el array de objetos. Por lo tanto, el orden espacial del algoritmo es $O(n)$, donde n es el número de objetos.

3.9.2. Estudio teórico Programación dinámica.

Este algoritmo aborda el problema de la mochila fraccionada utilizando programación dinámica. Divide los objetos en fracciones y los almacena en una tabla dinámica para determinar la selección óptima de elementos que maximiza el beneficio sin exceder la capacidad máxima del camión.

Preparación de los Datos:

- Se procesan los datos de entrada para dividir los objetos en fracciones según el número de divisiones especificado. Si el peso del objeto es divisible por el número de divisiones, se agregan las fracciones directamente. Si no es divisible, se calculan las fracciones con un resto.

Tabla de Programación Dinámica:

- Se utiliza un array "*dp*" para almacenar los resultados de los subproblemas. Cada celda "*dp[w]*" representa el beneficio máximo que se puede obtener con una capacidad de carga "*w*".

Cálculo del Valor Máximo:

- Se utiliza un bucle anidado para recorrer los objetos y actualizar la tabla "*dp*". Para cada objeto, se verifica si puede ser incluido en la mochila con la capacidad actual. Si es así, se actualiza el valor máximo que se puede obtener con esa capacidad considerando el beneficio del objeto y el valor obtenido hasta el momento.

Orden de complejidad:

- La preparación de los datos tiene una complejidad de $O(n * divisiones)$, donde "*n*" es el número de objetos y "*divisiones*" es el número de divisiones especificado.
- El bucle principal tiene una complejidad de $O(n * P)$, donde "*P*" es la capacidad máxima del camión.
- En general, el orden de complejidad del algoritmo es $O(n * divisiones + n * P)$, donde "*n*" es el número de objetos.

Orden espacial:

- El algoritmo utiliza el espacio adicional para almacenar las fracciones de objetos y la tabla "*dp*". El espacio utilizado para la tabla "*dp*" es proporcional a la capacidad máxima del camión $O(P)$, y el espacio utilizado para las fracciones depende del número de divisiones. Por lo tanto, orden espacial del algoritmo $O(P + n * divisiones)$.

3.9.3. Estudio experimental.

- Greedy.

n	P	Greedy	nlogn
10	100	240899	33,2192809
100	100	113160	664,385619
200	100	214021	1528,77124
300	100	258939	2468,64561
1000	100	587159	9965,78428
5000	100	4193979	61438,5619
10000	100	7322200	132877,124
20000	100	Stackoverflow	285754,248

Tabla 9. Datos obtenidos para Greedy del Ejercicio9.

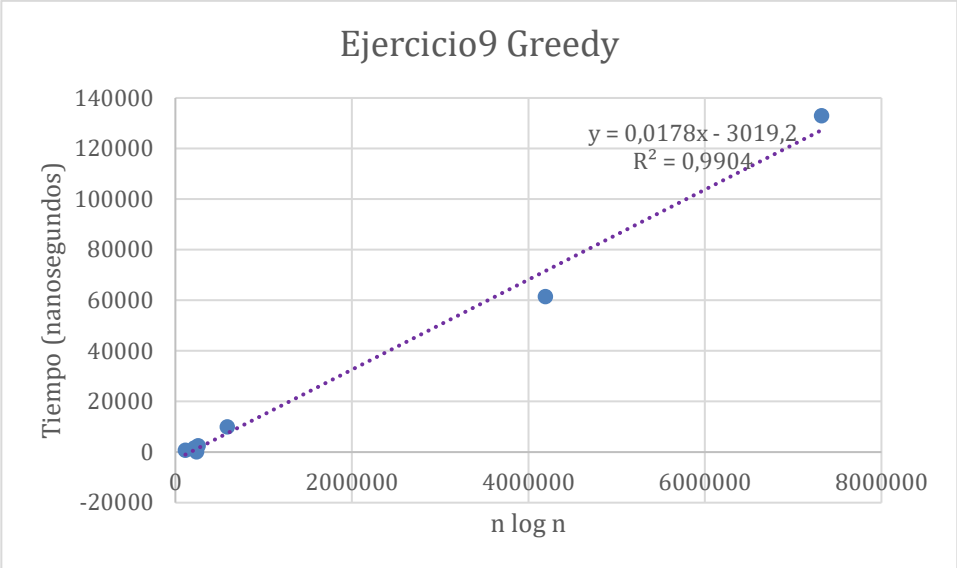

Ilustración 10. Gráfico obtenido para el Greedy del Ejercicio9.

Validación experimental: La ecuación de la recta que modela la relación entre los datos experimentales y el tiempo de ejecución muestra una dependencia lineal con una pendiente de 0.0178 y una intersección con el eje y de -3019.2. El coeficiente de determinación R^2 de 0.9904 indica que esta ecuación es una representación muy precisa de la relación entre los datos observados y el tiempo de ejecución. Esto sugiere que el tiempo de ejecución del algoritmo Greedy está fuertemente correlacionado con el tamaño del problema (n).

Coincidencia con la teoría: Los resultados experimentales respaldan la complejidad temporal predicha por el análisis teórico. La relación lineal entre el tiempo de ejecución y el tamaño del problema (n) se refleja tanto en la ecuación de la recta como en la complejidad temporal estimada. Esto indica que el algoritmo Greedy se comporta de acuerdo con las expectativas teóricas en términos de su eficiencia temporal.

Consideraciones adicionales: Aunque la complejidad temporal del algoritmo Greedy es eficiente en términos de los tamaños típicos de los problemas de la mochila, es importante tener en cuenta que su enfoque puede no garantizar la solución óptima en todos los casos. Los algoritmos Greedy son conocidos por su simplicidad y eficiencia, pero a veces pueden producir soluciones subóptimas cuando se enfrentan a ciertos tipos de problemas.

En resumen, el estudio experimental confirma que el algoritmo Greedy tiene una complejidad temporal que crece de manera logarítmica con respecto al número de objetos, como se predice teóricamente. Esto respalda su utilidad práctica y su aplicabilidad en situaciones donde se requiere una solución rápida y aceptable, aunque no necesariamente óptima, para problemas de optimización relacionados con la mochila.

- Programación Dinámica.

n	P	Fraccionado	P+n*divisiones
10	100	1037340	200
100	100	633640	1100
200	100	372039	2100
300	100	534020	3100
1000	100	1816360	10100
5000	100	9260540	50100
10000	100	19537679	100100
20000	100	Stackoverflow	200100

Tabla 10. Datos obtenidos para PD del Ejercicio9.

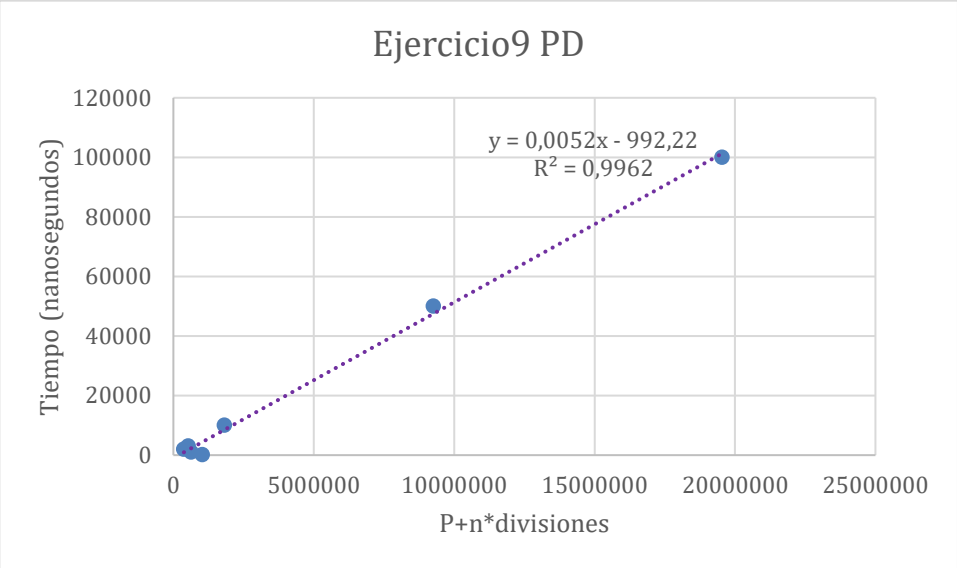

Ilustración 11. Gráfico obtenido para PD del Ejercicio9.

Validación experimental: La ecuación de la recta que modela la relación entre los datos experimentales y el tiempo de ejecución muestra una dependencia lineal con una pendiente de 0.0052 y una intersección con el eje y de -992.22. El coeficiente de determinación R^2 de 0.9962 indica que esta ecuación es una representación muy precisa de la relación entre los datos observados y el tiempo de ejecución. Esto sugiere que el tiempo de ejecución del algoritmo DP está fuertemente correlacionado con el tamaño del problema (n).

Coincidencia con la teoría: Los resultados experimentales respaldan la complejidad temporal predicha por el análisis teórico. La relación lineal entre el tiempo de ejecución y el tamaño del problema (n) se refleja tanto en la ecuación de la recta como en la complejidad temporal estimada. Esto indica que el algoritmo DP se comporta de acuerdo con las expectativas teóricas en términos de su eficiencia temporal.

Consideraciones adicionales: Aunque la complejidad temporal del algoritmo DP es razonablemente eficiente en términos de los tamaños típicos de los problemas de la mochila del camión, es importante tener en cuenta que podría no ser escalable para conjuntos de datos muy grandes debido a su naturaleza cuadrática en el tamaño de entrada.

En resumen, el estudio experimental confirma que el algoritmo DP tiene una complejidad temporal lineal con respecto al producto del número de objetos y la capacidad máxima del camión, como se predice teóricamente. Esto respalda su utilidad práctica y su aplicabilidad en situaciones donde se requiere resolver problemas de optimización relacionados con la carga de un camión.

Diferencias entre el algoritmo Greedy y el algoritmo de programación dinámica:

Aspecto	**Greedy**	**Programación Dinámica**
Orden de complejidad	$O(n \log n)$	$O(n * divisiones + n * P)$
Espacio	$O(n)$	$O(P + n * divisiones)$
Enfoque	Greedy o Voraz	Programación dinámica
Preparación de los datos	Ordenación por Beneficio/Peso	División de los objetos en fracciones según el atributo divisiones
Estructura de los datos	No utiliza tabla dinámica	Utiliza tabla dinámica "*dp*" para almacenar resultados
Proceso principal	Selecciona fracciones de objetos	Actualiza la tabla "dp" con beneficios máximos
Optimalidad	Garantiza solución óptima en todos los casos	No garantiza solución óptima debido a dividir los objetos en pesos enteros
Flexibilidad	Es más rápido y simple de implementar	Ofrece más flexibilidad al manejar diferentes conjuntos de datos y condiciones.

Tabla 11: Principales diferencias entre Greedy y Programación Dinámica

4. Conclusiones.

Archivo	**Rec**	**Rec2**	**DP**	**DP2**	**DP3**	**DPMD**	**Greedy**	**Fraccionado**
p01	11580	26420	62679	69120	37859	39600	60200	613520
p02	620	3120	13479	7919	3820	3979	3820	11679
p03	1019	14160	40580	48840	27639	29600	2600	439480
p04	1480	6800	14999	32999	10060	37599	10179	38340
p05	25119	15519	38720	42859	23300	24879	3280	33860
p06	759	13920	40780	52739	28740	30399	2920	105600
p07	56460	148379	290780	390919	259140	207419	9420	617620

Tabla 12. Datos obtenidos para los distintos archivos.

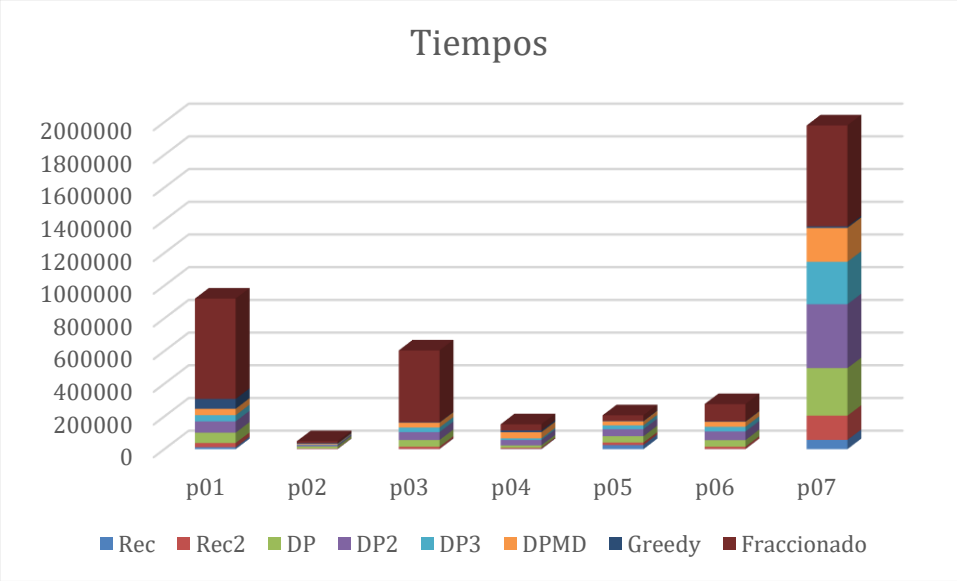

Ilustración 12. Gráfico obtenido para los distintos archivos.

Después de analizar los resultados obtenidos de la ejecución de diferentes algoritmos para el problema de llenar un camión con el mayor beneficio posible dentro de un peso máximo dado, se puede concluir lo siguiente:

El enfoque recursivo parece tener un rendimiento aceptable en términos de tiempo de ejecucióon para archivos de tamaño pequeño a mediano (p01, p02, p03, p04). Sin embargo, su rendimiento se deteriora significativamente para archivos más grandes (p05, p06, p07), lo que sugiere que no es escalable para conjuntos de datos grandes.

La variante de la recursión que evita el recálculo mejora sustancialmente el rendimiento para archivos de tamaño mediano (p01, p02, p03, p04), ya que reduce la sobrecarga asociada con la repetición de cálculos. Sin embargo, como en el caso anterior, su eficiencia disminuye considerablemente para archivos más grandes (p05, p06, p07).

La programación dinámica muestra un rendimiento sólido y consistente en todos los conjuntos de datos evaluados. Aunque puede tener una sobrecarga inicial mayor debido al cálculo de subproblemas, su enfoque basado en la memoización le permite mantener un rendimiento estable incluso para archivos más grandes.

Las demás variantes de programación dinámica, aunque muestran diferencias de tiempos de ejecución, en general mantienen un rendimiento sólido y escalable en todos los conjuntos de datos.

El enfoque Greedy tiene un rendimiento aceptable en la mayoría de los casos, excepto en conjuntos de datos grandes ya que su orden de complejidad se basa en el método de ordenación, donde se puede ver que aumenta un poco el tiempo de ejecución.

El algoritmo de programación dinámica que soluciona el problema de la mochila fraccionada es posiblemente el que mayor tiempo de ejecución tiene. Esto se debe a que los datos crecen bastante teniendo en cuenta el número de divisiones que se pueden realizar, porque con cada división aumenta el conjunto de datos bastante.

Tras analizar diferentes algoritmos para llenar un camión con el mayor beneficio en un peso dado, se destaca que el enfoque recursivo muestra buen rendimiento en archivos pequeños a medianos, pero falla en escalabilidad para conjuntos grandes. La programación dinámica, en cambio, ofrece rendimiento sólido y estable en todos los conjuntos, gracias a su enfoque de memoización. Aunque otras variantes de programación dinámica también son efectivas, el enfoque Greedy tiene un rendimiento aceptable, pero su complejidad aumenta con conjuntos de datos más grandes. Por último, el algoritmo para la mochila fraccionada tiene un tiempo de ejecución notable debido al crecimiento exponencial de datos con cada división.

A. Anexo.

A.1. Diseño del código.

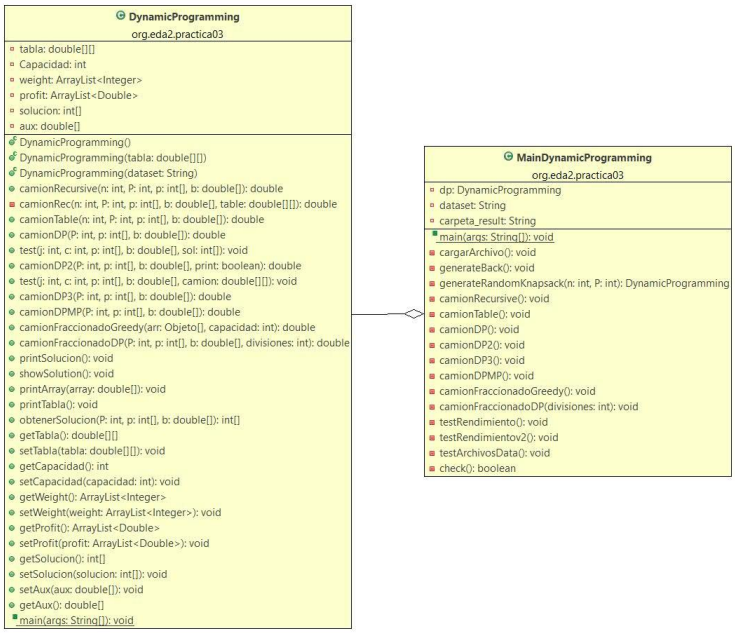

Ilustración 13: Diagrama de Clases

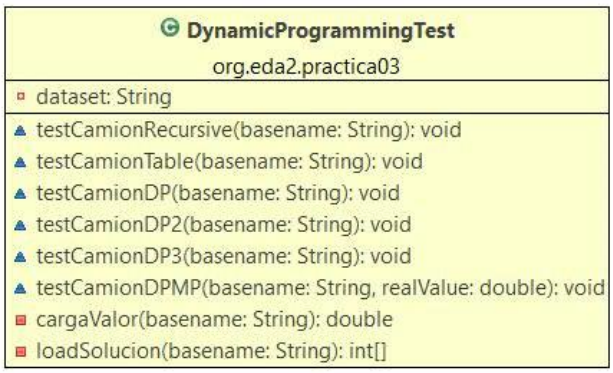

Ilustración 14: Diagrama de Clases asociado a los Tests

A.2. Esquema archivos fuente.

El esquema de archivos fuente es el siguiente:

DynamicProgramming.java

Ruta: main\java\org\eda2\practica03\DynamicProgramming.java

Descripción: Este archivo contiene las implementaciones de los distintos algoritmos planteados en los ejercicios de la práctica. Contiene el javadoc asociado y los distintos métodos auxiliares para la correcta implementación de los ejercicios.

MainDynamicProgramming.java

Ruta: main\java\org\eda2\practica03\MainDynamicProgramming.java

Descripción: Este archivo contiene la implementación para poder utilizar un menú interactivo y poder usar los distintos algoritmos asociados a la clase *DynamicProgramming.java*. Desde esta clase se puede realizar desde pruebas hasta los análisis experimentales.

DynamicProgrammingTest.java

Ruta: test\java\org\eda2\practica03\DynamicProgrammingTest.java

Descripción: Este archivo contiene los distintos test parametrizados para poder comprobar que las implementaciones son correctas según los archivos facilitados en el enlace perteneciente al guion de las prácticas.

Dataset.

Ruta: docs\practica03\dataset\

Descripción: Esta carpeta contiene los archivos facilitados en el enlace perteneciente al guion de las prácticas.

Resultados.

Ruta: docs\practica03\resultados\

Descripción: Esta carpeta contiene distintos archivos en formato CSV para apoyar en el análisis experimental.

Memoria.docx

Ruta: docs\practica03\Memoria.docx

Descripción: Este archivo contiene la memoria de la práctica. Incluye una descripción detallada del problema abordado, el enfoque utilizado para resolverlo, los resultados obtenidos y cualquier otra información relevante relacionada con la práctica.

PR3_Estudio_Experimental.xlsx

Ruta: docs\practica03\PR3_Estudio_Experimental.xlsx

Descripción: Este archivo Excel alberga los cálculos y datos recopilados durante el estudio experimental realizado como parte de la práctica. Puede incluir tablas, gráficos u otros elementos visuales que representen los resultados de las pruebas realizadas.

PR3_Tareas_a_Repartir.xlsx

Ruta: docs\practica03\PR3_Tareas_a_Repartir.xlsx

Descripción: Este archivo Excel contiene la distribución de tareas asignadas a cada uno de los compañeros que trabajan en la práctica. Incluye una lista de tareas específicas y las personas responsables de completar cada tarea.

B. Bibliografía

Knapsack Problem. (12 de 05 de 2024). Obtenido de https://www.enjoyalgorithms.com/blog/zero-one-knapsack-problem

Temario Asignatura. (2024). Almeria.