UNIVERSIDAD DE ALMERÍA

Proyecto 3 Programación Dinámica

Llenar el camión con los productos más beneficiosos

Estructura de datos y algoritmos II

equipo-ualeda2-2024-simon-antonio

Venzal Sánchez, Simón María
Guerrero Crespo-López, Antonio

Índice

Proyecto 3 Programación Dinámica

1.- Objetivo.

1.1 Introducción teórica a la programación dinámica.

La programación dinámica o PD utiliza un método ascendente (bottom-up) para la resolución de problemas, a diferencia de divide y vencerás que es descendente, en este enfoque se descompone el problema en subproblemas solapados y se va construyendo la solución con las soluciones de estos subproblemas.

Otra similitud con DyV es el razonamiento inductivo o la descomposición recursiva del problema, no obstante a la hora de resolver el problema DyV primero aplica la forma recursiva, mientras que la programación dinámica resuelve primero los subproblemas (una única vez) y guarda los resultados en una tabla (algoritmo iterativo).

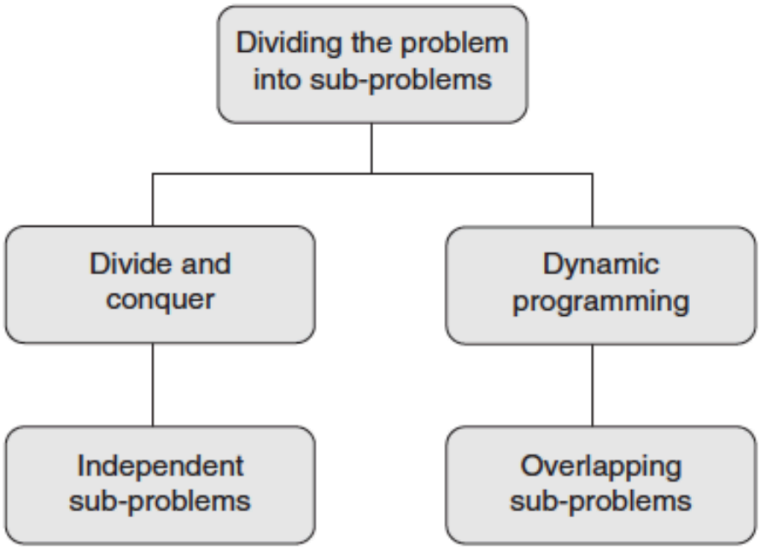

Imagen 1: Diagrama sobre las diferencias de enfoques en DyV y Programación dinámica.
Fuente: Apuntes de clase Estructura de datos y algoritmos 2 (Tema 4 Dynamic Programing), Universidad de Almería.

Para la PD no hay un esquema algorítmico único, aunque todos se fundamentan en el principio de optimalidad de Bellman, este concepto fundamental para la PD es una técnica de optimización matemática la cual establece que:

“Una política óptima tiene la propiedad de que, independientemente del estado inicial y la decisión inicial, las decisiones restantes deben constituir una política óptima con respecto al estado resultante de la primera decisión” (Bellman, 1957, Cap. III.3.)

Es decir, dada una secuencia óptima de decisiones que resuelve un problema, cualquier subsecuencia de decisiones que tenga el mismo estado final también debe ser óptima para el subproblema correspondiente.

Este principio es la base de la ecuación de Bellman, donde se busca descomponer un problema de optimización dinámica en subproblemas más simples, permitiendo como se resolver problemas de una manera más eficiente, en el que por ejemplo como en esta práctica en la que se pretende maximizar n objetos (verduras), cada una con un peso natural pi (pi > 0) y con un beneficio (que se puede obtener al venderlo) de un valor real bi (bi > 0) y que el camión tiene un PMA (Peso Máximo Autorizado) de P.

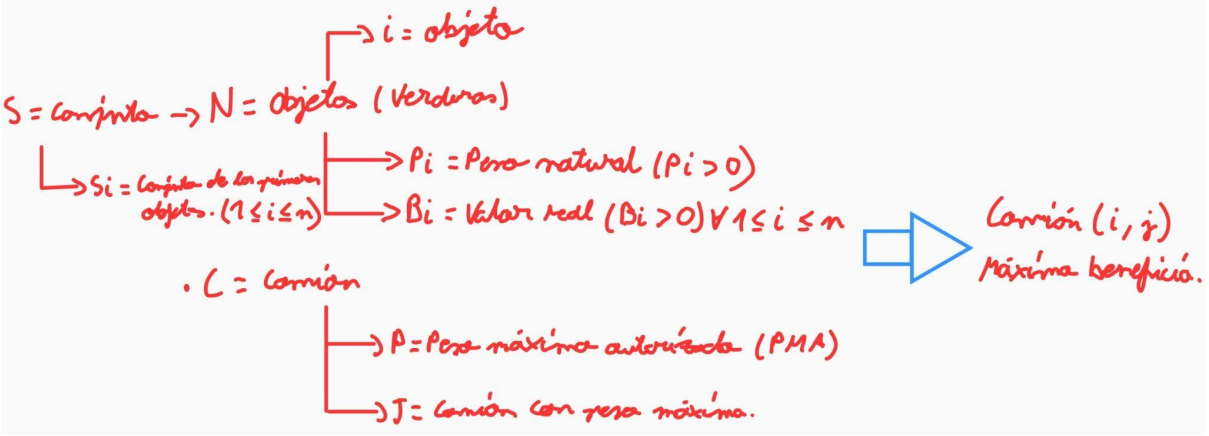

Imagen 2: Diagrama resumen de variables requeridas para el desarrollo de la práctica 3 de EDA".
Fuente: Elaboración propia.

1.2 División del trabajo.

La división del trabajo realizada ha sido muy similar a los anteriores proyectos, en la que ambos hemos hecho de todo, al ser únicamente dos personas realizamos un contacto diario entre nosotros sobre los problemas, avances y corrección de errores que vamos realizando, donde trabajamos tanto de manera presencial como online.

2.- Antecedentes.

2.1 Motivación.

Para aplicar la programación dinámica de manera general se realizan los siguientes pasos:

1. Se comprueba que se cumple el principio de optimalidad de Bellman, para lo que hay que encontrar la estructura de la solución.

2. Se define recursivamente la solución óptima del problema (en función de los valores de las soluciones para subproblemas de menor tamaño).

3. Se calcula el valor de la solución óptima utilizando un enfoque bottom-up.

    - Determinando el conjunto de subproblemas a resolver (tamaño de la tabla).
    - Identificando los subproblemas que son un caso base.
    - Calculando los valores de soluciones más complejas a partir de los valores previamente calculados (ecuación recurrente).

4. Fin de proceso determinando la solución óptima a partir de los datos almacenados en la tabla.

Estos pasos y necesidad previa de estructuración del enfoque a la hora de resolver el problema proporcionan una visión y metodología de trabajo diferente con la que pensar de otra manera cómo resolver problemas complejos de programación

2.2 Alternativas planteadas.

Dado la carga de trabajo que tenemos tanto Antonio como yo (Simón), el espacio entre entregas de las prácticas de la asignatura y que el propio guión de la práctica propone cabeceras para la implementación, se consideró tomar un enfoque de trabajo en el que se resolvieran los problemas propuestos de una manera ágil y directa, dado que el tiempo era muy ajustado se prefirió resolver y más adelante mejorar o plantear otras opciones.

Por ejemplo el primer ejercicio se podría resolver haciendo uso de DyV, mientras que en el segundo directamente se plantó con enfoque con PD, también se indica usar un enfoque Greedy en el noveno ejercicio, luego como se ha comentado no nos hemos planteado muchas alternativas dado el camino por el que nos lleva la práctica en sí y las recomendaciones que proporciona.

2.3 Solución final.

Las diversas soluciones finales alcanzadas buscaban un código eficiente y muy legible ya que al existir numerosos ejercicios se ha buscado una practicidad total y de poderse reusar y replantear código ya funcional.

3.- Trabajo a desarrollar.

3.1 Estudio de la implementación.

3.1.1 Camión Recursivo.

Para resolver el problema del camión recursivo se emplean dos métodos, camionRecursive() y camionRecursive(int n, int P, int[] p, double[] b).

El primer método, **camionRecursive()**. Empieza obteniendo los datos necesarios como el número total de objetos, la capacidad máxima de la mochila, los pesos y los beneficios de los objetos. Luego llama al segundo método, que es recursivo, pasando estos datos como argumentos. Este método devuelve un beneficio máximo que se almacena en una instancia de Resultado, a la cual se le asigna este valor de beneficio máximo antes del return.

El segundo método, **camionRecursive(int n, int P, int[] p, double[] b),** es donde se realiza el cálculo recursivo para determinar el beneficio máximo que se puede obtener. Comprueba si se han considerado todos los objetos o si la mochila ya está llena, casos en los cuales retorna cero porque no se puede obtener más beneficio. Si el objeto actual es demasiado pesado para la capacidad restante de la mochila, simplemente lo omite y hace una llamada recursiva sin incluir este objeto. Si el objeto puede ser incluido, entonces calcula el beneficio

de dos escenarios: uno donde el objeto se incluye y otro donde no se incluye. Obtiene el máximo de estos dos beneficios como el beneficio resultante para esa instancia particular de la llamada recursiva. De esta forma, el método explora todas las combinaciones posibles de incluir o no cada objeto, asegurando que se encuentre el máximo beneficio total para la capacidad dada de la mochila.

Pseudocódigo camionRecursive
Función camionRecursive() devuelve Resultado n = obtenerN() P = obtenerP() pesos = obtenerPesos() beneficios = obtenerBeneficios() maxBeneficio = camionRecursive(n, P, pesos, beneficios) resultado = nuevo Resultado() resultado.setBeneficioTotal(maxBeneficio) devolver resultado Función camionRecursive(n, P, p, b) devuelve número real si n == 0 o P == 0 entonces devolver 0 si p[n - 1] > P entonces devolver camionRecursive(n - 1, P, p, b) devolver máximo(camionRecursive(n - 1, P - p[n - 1], p, b) + b[n - 1], camionRecursive(n - 1, P, p, b))

Tabla 1: Pseudocódigo camionRecursive.

3.1.2 Camión Tabla.

Para resolver este problema se utiliza programación dinámica para evitar el recálculo de estados ya evaluados.

El primer método, **camionTable()**. Inicialmente, se obtienen valores como el número total de objetos (n), la capacidad máxima de la mochila (P), los pesos de los objetos (p) y sus beneficios (b). Se crea una tabla tabla de tamaño (n+1) x (P+1) y se inicializa con valores de -1 para indicar que aún no se ha calculado el beneficio para esa combinación de objeto y capacidad de la mochila.

El método llama a **camionRec(n, P, p, b)**, que es un método recursivo que utiliza esta tabla para almacenar los beneficios máximos calculados, evitando así la necesidad de recalculaciones innecesarias. El beneficio máximo obtenido de esta función se asigna a una instancia de Resultado, para devolverlo al final.

El método **camionRec(int n, int P, int[] p, double[] b)**. Verifica si ya se alcanzó el caso base de la recursión, es decir, cuando no quedan objetos por considerar (n == 0) o no hay capacidad en la mochila (P == 0), retornando 0 en esos casos. Si ya se ha calculado previamente el beneficio para la combinación de objeto y capacidad (esto es, tabla[n][P] es distinto de -1), simplemente devuelve el valor almacenado. Si el objeto actual no cabe en la mochila, llama recursivamente a la función sin incluir el objeto. Si el objeto cabe, compara el beneficio de incluir el objeto actual en la mochila más el mejor beneficio sin ese peso, contra

el beneficio de no incluir el objeto. El mayor de estos dos valores se almacena en tabla[n][P] y se devuelve como el beneficio óptimo para esa instancia.
Al no recalcular continuamente las mismas operaciones ya hechas anteriormente, el algoritmo mejora muchísimo su eficiencia, y si en lugar de usar una tabla para guardar esos datos se emplea un mapa se puede ganar aún más eficiencia.

Pseudocódigo camionTable

```
Función camionTable() devuelve Resultado
    n = obtenerN()
    P = obtenerP()
    p = obtenerPesos()
    b = obtenerBeneficios()
    tabla = nueva matriz de números reales con dimensiones (n + 1) x (P + 1)

    para i desde 0 hasta n hacer
        para j desde 0 hasta P hacer
            tabla[i][j] = -1
        fin para
    fin para

    maxBeneficio = camionRec(n, P, p, b)
    resultado = nuevo Resultado()
    resultado.setBeneficioTotal(maxBeneficio)
    devolver resultado

Función camionRec(n, P, p, b) devuelve número real
    si n == 0 o P == 0 entonces
        devolver 0
    si tabla[n][P] != -1 entonces
        devolver tabla[n][P]
    si p[n - 1] > P entonces
        tabla[n][P] = camionRec(n - 1, P, p, b)
    sino
        conObjeto = camionRec(n - 1, P - p[n - 1], p, b) + b[n - 1]
        sinObjeto = camionRec(n - 1, P, p, b)
        tabla[n][P] = máximo(conObjeto, sinObjeto)
    fin si
    devolver tabla[n][P]
```

Tabla 2: Pseudocódigo camionTable.

3.1.3 Camión Programación dinámica.

Consiste en el problema de la mochila mediante la técnica de programación dinámica para optimizar el beneficio total sin exceder la capacidad máxima permitida.
El método **camionDP()** comienza por inicializar la tabla de programación dinámica, tabla, que tiene dimensiones (n+1) x (P+1), donde n es el número de objetos disponibles y P es la capacidad máxima de la mochila. Cada celda de esta tabla representa el máximo beneficio obtenido con un número determinado de objetos y hasta una capacidad específica de la mochila. Las celdas se inicializan en -1 para indicar que aún no se ha calculado el beneficio para esa configuración de objeto-capacidad.

Utilizando un doble bucle, se llena la tabla de la siguiente manera:

- Bucle externo: Itera sobre cada objeto disponible.
- Bucle interno: Itera sobre cada posible capacidad desde 1 hasta P.

Si la capacidad actual es menor que el peso del objeto en consideración, se copia el valor de la celda anterior (mismo objeto, capacidad anterior) porque el objeto actual no puede ser incluido.
Si la capacidad es suficiente para incluir el objeto, se calcula el beneficio de incluirlo sumando su valor al máximo beneficio obtenido con la capacidad restante después de incluir el objeto. Este valor se compara con el beneficio de no incluir el objeto (valor en la celda anterior). La tabla en la posición actual se actualiza con el máximo de estos dos valores.

Una vez completada la tabla, el valor en tabla[n][P] representa el máximo beneficio que se puede obtener con todos los objetos considerados y la capacidad máxima de la mochila.
Determinación de los objetos incluidos:

Para identificar qué objetos forman parte de la solución óptima, se utiliza el método **getItemSolRecursiva()** que llama a **test()**. Este método recursivo explora la tabla de programación dinámica de abajo hacia arriba y de derecha a izquierda, marcando los objetos que deben incluirse para alcanzar el máximo beneficio:

Comienza en tabla[n][P] y revisa si el beneficio en esta celda provino de incluir el objeto n. Si es así, marca el objeto como incluido (ajustando el valor en el arreglo de soluciones) y reduce la capacidad en el peso de ese objeto para continuar el chequeo.
Si el beneficio máximo no provino de incluir el objeto actual, simplemente se mueve a la celda anterior para continuar la verificación.

El resultado, que incluye el beneficio total y el peso total de los objetos seleccionados, se guarda en una instancia de la clase Resultado. Esta instancia se retorna, proporcionando el peso y beneficio total además de los objetos seleccionados.

Pseudocódigo camionDP

```
Función camionDP() devuelve Resultado
    P = obtenerP()
    p = obtenerPesos()
    b = obtenerBeneficios()
    n = longitud(b)
    tabla = nueva matriz de números reales con dimensiones (n + 1) x (P + 1)

    para i desde 1 hasta n hacer
        para j desde 1 hasta (p[i - 1] - 1) hacer
            tabla[i][j] = tabla[i - 1][j]
        fin para
        para j desde p[i - 1] hasta P hacer
            b1 = tabla[i - 1][j - p[i - 1]] + b[i - 1]
            b2 = tabla[i - 1][j]
            tabla[i][j] = máximo(b1, b2)
```

```
        fin para
    fin para

    maxBeneficio = tabla[n][P]
    resultado = nuevo Resultado()
    resultado.setBeneficioTotal(maxBeneficio)
    resultado.setPesoTotal(calcularPesoTotal(getItemSolRecursiva(), p))
    devolver resultado

Función getItemSolRecursiva() devuelve arreglo de enteros
    n = obtenerN()
    P = obtenerP()
    p = obtenerPesos()
    b = obtenerBeneficios()
    sol = nuevo arreglo de enteros con longitud n
    test(n, P, p, b, sol)
    devolver sol

Subrutina test(j, c, p, b, sol)
    si j > 0 entonces
        si c < p[j - 1] entonces
            test(j - 1, c, p, b, sol)
        sino
            si tabla[j - 1][c - p[j - 1]] + b[j - 1] > tabla[j - 1][c] entonces
                test(j - 1, c - p[j - 1], p, b, sol)
                sol[j - 1] = 1
            sino
                test(j - 1, c, p, b, sol)
            fin si
        fin si
    fin si

Función calcularPesoTotal(sol, pesos) devuelve número real
    pesoTotal = 0
    para i desde 0 hasta longitud(sol) - 1 hacer
        si sol[i] == 1 entonces
            pesoTotal += pesos[i]
        fin si
    fin para
    devolver pesoTotal
```

Tabla 3: Pseudocódigo camionDP.

3.1.4 Camión Programación dinámica 2.

Este ejercicio trata el problema de la mochila usando programación dinámica iterativa, optimizando el almacenamiento y recuperación de los objetos que maximizan el beneficio sin exceder la capacidad máxima de la mochila.

El método **camionDP2()** comienza obteniendo la capacidad máxima de la mochila (P), los pesos (p) y los beneficios (b) de los objetos. Se inicializa una tabla de tamaño (n+1) x (P+1) para ayudar en el cálculo del beneficio máximo, donde n es el número total de objetos. Esta

tabla almacena los beneficios máximos alcanzados para cada combinación de número de objetos y capacidades de mochila para evitar el recálculo de estados.

El llenado de la tabla se realiza mediante dos bucles anidados. El bucle externo itera sobre cada objeto y el bucle interno sobre todas las capacidades posibles desde 1 hasta P. Para cada capacidad, si el objeto actual puede ser incluido, se calcula el beneficio de incluir este objeto sumando su beneficio al máximo beneficio obtenido con la capacidad restante después de incluir el objeto. Este nuevo beneficio se compara con el beneficio obtenido sin incluir el objeto. La celda se actualiza con el máximo de estos dos beneficios.

Una vez que la tabla está completamente llena, el beneficio máximo se obtiene de tabla[n][P]. Para determinar qué objetos forman parte de la solución óptima, se utiliza el método **getItemSolIterativa()**, que recorre la tabla de programación dinámica de manera iterativa para identificar los objetos incluidos.
Se comienza en la última celda tabla[n][P] y se mueve hacia atrás para determinar si el beneficio en una celda particular provino de incluir el objeto correspondiente. Si fue así, el objeto se marca como incluido y se ajusta la capacidad a la que correspondería haber excluido el peso del objeto.

El resultado guarda el beneficio total, el peso total y los objetos seleccionados en una instancia de la clase Resultado para después retornarla con el formato adecuado.

Pseudocódigo camionDP2

```
Función camionDP2() devuelve Resultado
    P = obtenerP()
    p = obtenerPesos()
    b = obtenerBeneficios()
    n = longitud(b)
    tabla = nueva matriz de números reales con dimensiones (n + 1) x (P + 1)

    para i desde 1 hasta n hacer
        para j desde 1 hasta P hacer
            tabla[i][j] = tabla[i - 1][j]
            si p[i - 1] <= j entonces
                b1 = tabla[i - 1][j - p[i - 1]] + b[i - 1]
                b2 = tabla[i][j]
                tabla[i][j] = máximo(b1, b2)
            fin si
        fin para
    fin para

    maxBeneficio = tabla[n][P]
    resultado = nuevo Resultado()
    resultado.setBeneficioTotal(maxBeneficio)
    sol = getItemSolIterativa()
    resultado.setPesoTotal(calcularPesoTotal(sol, p))
    para i desde 0 hasta longitud(sol) - 1 hacer
        si sol[i] == 1 entonces
            resultado.agregarObjeto(nuevo Objeto("Objeto " + (i + 1), p[i], b[i]))
        fin si
```

```
    fin para
    devolver resultado

Función getItemSolIterativa() devuelve arreglo de enteros
    sol = nuevo arreglo de enteros con longitud obtenerN()
    j = longitud(tabla[0]) - 1
    para i desde longitud(tabla) - 1 hasta 1 hacer
        si tabla[i][j] != tabla[i - 1][j] entonces
            sol[i - 1] = 1
            j -= getPesos()[i - 1]
        fin si
    fin para
    devolver sol
```

Tabla 4: Pseudocódigo camionDP2.

3.1.5 Camión Programación dinámica 3.

En este caso se implementa otra variante del problema de la mochila usando programación dinámica, pero en esta ocasión utiliza un array para optimizar el cálculo del beneficio máximo sin exceder la capacidad máxima permitida de la mochila. Esto hace que esta variante sea más eficiente en términos de uso de memoria.

El método **camionDP3()** inicia extrayendo la capacidad máxima de la mochila (P), los pesos (p) y los beneficios (b) de los objetos. Después, llama al método camionDP3() que toma estos valores como parámetros para calcular el beneficio máximo.

Dentro del método, se declara un array con tamaño P+1, donde cada posición j del arreglo representará el máximo beneficio obtenible con una mochila de capacidad j. Este array se inicializa con ceros, representando que inicialmente no se ha obtenido ningún beneficio para ninguna capacidad.

El proceso de llenado del array utiliza un bucle que itera sobre cada objeto y un bucle anidado que itera sobre las capacidades de la mochila desde la capacidad máxima P, hacia abajo hasta el peso del objeto actual. Esto se hace para que cada capacidad solo considere el objeto actual una vez y evita sobrescribir los resultados de una iteración anterior que aún podrían ser necesarios.

Cada vez que se evalúa si un objeto puede ser incluido en la mochila (es decir, cuando la capacidad j actual es mayor o igual al peso del objeto p[i-1]), se realiza una comparación entre el beneficio máximo existente en esa capacidad (array[j]) y el nuevo beneficio posible si se incluyera el objeto. Este nuevo beneficio se calcula sumando el beneficio del objeto (b[i-1]) al beneficio máximo obtenible con la capacidad restante después de incluir el objeto (array[j - p[i-1]]). Si este nuevo

cálculo resulta ser mayor que el beneficio almacenado, entonces se actualiza array[j] con este nuevo máximo.

Finalmente, el beneficio máximo que se puede obtener con todos los objetos y la capacidad total de la mochila se encuentra en array[P]. Este valor se asigna a una nueva instancia de la clase Resultado, estableciendo el beneficio total alcanzado. Este resultado se retorna, proporcionando así el beneficio máximo posible junto con el peso.

Pseudocódigo camionDP3

```
Función camionDP3() devuelve Resultado
    P = obtenerP()
    p = obtenerPesos()
    b = obtenerBeneficios()
    maxBeneficio = camionDP3(P, p, b)
    resultado = nuevo Resultado()
    resultado.setBeneficioTotal(maxBeneficio)
    devolver resultado

Función camionDP3(P, p, b) devuelve número real
    n = longitud(b)
    array = nuevo arreglo de números reales con longitud (P + 1)

    para i desde 1 hasta n hacer
        para j desde P hasta p[i - 1] hacer
            array[j] = máximo(array[j], array[j - p[i - 1]] + b[i - 1])
        fin para
    fin para
    devolver array[P]
```

Tabla 5: Pseudocódigo camionDP3.

3.1.6 Camión Voraz.

El método **camionGreedy()** implementa el algoritmo voraz (greedy) para resolver el problema de la mochila, buscando maximizar el beneficio total sin exceder la capacidad máxima de la mochila, conocida como P. Este enfoque selecciona objetos basándose en su relación beneficio/peso.

El método inicia estableciendo un peso acumulado inicial (pesoFinal) de cero y crea un array (used) para marcar los objetos que ya han sido seleccionados y evitar su reconsideración. También se inicializa una instancia de Resultado para almacenar el beneficio y el peso total acumulados, así como los objetos específicos que se van seleccionando.
El bucle principal del método se ejecuta mientras el pesoFinal sea menor que la capacidad máxima P. Dentro de este bucle ocurre lo siguiente:

Se define una variable maxBeneficioCoste para identificar el mayor ratio de beneficio a peso de los objetos aún no seleccionados.Se itera sobre el arreglo de objetos. Para cada objeto

no seleccionado, se calcula su ratio de beneficio a peso. Si este ratio es mayor que el maxBeneficioCoste registrado, se actualiza este máximo y se guarda el índice del objeto como item. Si después de revisar todos los objetos, item sigue siendo -1, significa que no hay más objetos que puedan ser seleccionados sin repetir, y sale del bucle.

Si se encuentra un objeto con el máximo ratio (item diferente de -1), se marca como usado. Luego, se verifica si agregar este objeto excede la capacidad de la mochila. Si no la excede ocurre lo siguiente:
Se suma el peso del objeto al pesoFinal. El objeto se añade a la lista de objetos en la instancia resultado y se actualizan los totales de peso y beneficio en resultado.

El proceso se repite hasta que no se pueden agregar más objetos sin superar la capacidad P o hasta que todos los objetos han sido considerados.

Al final del método, se retorna la instancia resultado, que contiene el beneficio total alcanzado, el peso total y la lista de objetos que optimizan el valor de la mochila bajo el enfoque greedy.

Pseudocódigo camionGreedy

```
Función camionGreedy() devuelve Resultado
    finalWeight = 0
    usado = nuevo arreglo de booleanos con longitud obtenerN()
    resultado = nuevo Resultado()
    resultado.setPesoTotal(0)
    resultado.setBeneficioTotal(0)

    mientras finalWeight < P hacer
        maxProfitCost = 0
        item = -1
        para i desde 0 hasta longitud(objetos) - 1 hacer
            si no usado[i] entonces
                rel = objetos[i].getBeneficio() / objetos[i].getPeso()
                si rel > maxProfitCost entonces
                    maxProfitCost = rel
                    item = i
                fin si
            fin si
        fin para
        si item == -1 entonces
            salir del bucle
        fin si

        usado[item] = verdadero
        si finalWeight + objetos[item].getPeso() <= P entonces
            finalWeight += objetos[item].getPeso()
            resultado.getObjetos().add(nuevo Objeto(objetos[item].getNombre(),
objetos[item].getPeso(), objetos[item].getBeneficio()))
            resultado.setPesoTotal(resultado.getPesoTotal() + objetos[item].getPeso())
            resultado.setBeneficioTotal(resultado.getBeneficioTotal() +
objetos[item].getBeneficio())
```

```
    sino
        salir del bucle
    fin si
fin mientras

devolver resultado
```

Tabla 6: Pseudocódigo camionGreedy.

3.2 Estudio teórico.

Dadas las dos tecnicas algoritmicas en esta práctica como son PD y Greedy a modo de resumen general tenemos:

1. **Programación Dinámica (PD):**

- Características: Divide el problema en subproblemas más pequeños y resuelve cada subproblema una sola vez, a menudo se utiliza para problemas de optimización.

- Ventajas: Garantiza una solución óptima, puede manejar problemas con restricciones.

- Desventajas: Requiere más espacio de memoria debido a la tabla de almacenamiento, también puede ser más lento que el enfoque greedy para algunos problemas.

- Ejemplo: Resolución del problema de la mochila.

2. **Greedy:**

- Características: Toma decisiones basadas en el momento concreto en lugar de considerar todo el conjunto de posibilidades, no garantiza una solución óptima general.

- Ventajas: Más eficiente en términos de tiempo de ejecución, útil para problemas de selección o asignación.

- Desventajas: No siempre encuentra la solución óptima.

- Ejemplo: Árbol de expansión mínimo haciendo uso del algoritmo de Kruskal.

La complejidad esperada en las diferentes versiones puede variar pero de manera general se tiene:

- Algoritmos recursivos: Ofrece peores rendimientos debido a que recalcula varias veces el mismo problema.

- Programación dinámica: Los tiempos están determinados por el tamaño de la tabla a rellenar, con una complejidad de orden $O(n * P)$.

- Algoritmos Greedy: La complejidad de los algoritmos greedy puede variar según el problema específico, por ejemplo en el cambio de monedas la complejidad puede ser baja, sin embargo, no hay una regla general para la complejidad de los algoritmos greedy.

Para la PD que es en la que se centra la práctica, el tiempo de ejecución dependerá de las características concretas del problema que se vaya a resolver.
En general, será de la forma:

Tamaño de la tabla x Tiempo de rellenar cada elemento de la tabla

La programación dinámica (PD) se basa en el uso de tablas para almacenar los resultados parciales, aunque algunos de estos cálculos parciales pueden ser innecesarios, en general, los algoritmos obtenidos mediante esta técnica logran tener complejidades (espaciales y temporales) bastante razonables, de igual manera se debe de no intentar obtener una complejidad temporal de orden polinómico que conduzca a una complejidad espacial excesivamente alta.

3.2.1 Camión Recursivo.

Este método, que utiliza un enfoque recursivo, implementa un algoritmo de programación dinámica recursiva para resolver el problema de la mochila.

El objetivo del problema es determinar la máxima cantidad de beneficio que se puede obtener al llenar una mochila con capacidad P con elementos de pesos p y beneficios b.
En este contexto, se busca calcular el máximo beneficio posible que se puede obtener al llenar una mochila con capacidad limitada.

En cuanto a la complejidad se tiene:

- **camionRecursive():** De complejidad temporal O(1), ya que solo realiza operaciones constantes y llama a la función camionRecursive(int n, int P, int[] p, double[] b).

- **camionRecursive(int n, int P, int[] p, double[] b):** Esta función tiene una complejidad de tiempo de **O(2^n)** en el peor de los casos, dado que para cada elemento, se tiene dos opciones: incluirlo en el camión o no incluirlo, luego se explora todas las posibles combinaciones de elementos.

3.2.2 Camión Tabla.

Este método utiliza un enfoque de programación dinámica con una tabla de memorización para calcular el máximo beneficio posible que se puede obtener al llenar una mochila con capacidad limitada.

Inicializa una tabla de memorización y luego llama al método auxiliar *camionRec* para realizar el cálculo, para la complejidad se obtiene:

- **camionTable():** Da una complejidad temporal de **O(n*P)**, donde n es el número de elementos y P es la capacidad del camión, ya inicializa una tabla de tamaño n x P con -1, lo que toma tiempo O(n*P), y luego llama a la función camionRec(n, P, p, b).

- **camionRec(int n, int P, int[] p, double[] b):** También tiene una complejidad temporal de **O(n*P)** ya que utiliza la técnica de memorización para almacenar los resultados de los subproblemas ya calculados en la tabla, por lo tanto, cada subproblema se calcula solo una vez y se almacena en la tabla para su uso futuro, evitando la necesidad de calcular el mismo subproblema varias veces, lo que reduce la complejidad de tiempo de exponencial a polinomial.

Por lo tanto, la complejidad temporal total del algoritmo es **O(n*P)**, lo que es mucho más eficiente que la versión recursiva sin memorización.

3.2.3 Camión Programación dinámica.

Este método, que utiliza un enfoque de PD, calcula el máximo beneficio posible que se puede obtener al llenar una mochila con capacidad limitada.

Resuelve el problema de la mochila construyendo una tabla de manera iterativa para almacenar y actualizar los resultados de subproblemas.

El análisis de complejidad resultante en este caso es:

- **camionDP():** Esta función tiene una complejidad temporal de **O(n*P)**, donde n es el número de elementos y P es la capacidad del camión ya que inicializa una tabla de tamaño n x P con -1, y luego llena la tabla con los beneficios máximos posibles para cada subproblema, cada celda de la tabla se llena en tiempo constante, por lo que la complejidad de tiempo total es O(n*P).

- **getItemSolRecursiva()** y **test(int j, int c, int[] p, double[] b, int[] sol):** Estas funciones tienen una complejidad de tiempo de O(n), ya que recorren los elementos una vez para determinar cuáles se incluyen en la solución óptima.

- **calcularPesoTotal(int[] sol, int[] pesos):** Esta función también tiene una complejidad temporal de O(n), ya que recorre la solución para calcular el peso total.

Por lo tanto, la complejidad temporal total del algoritmo es **O(n*P)**.

3.2.4 Ejercicio 4, Relación entre el caso 2 y 3.

Para los ejercicios 2 y 3, el camión tiene una capacidad de carga de cero, esto significa que no es posible cargar ningún elemento en él y por consiguiente:

- En el caso 3, los valores de la última columna de la tabla bidimensional deben ser cero, reflejando la incapacidad de cargar cualquier elemento.

- Estos valores son consistentes con los de la última columna de la matriz bidimensional del caso 2.

Para determinar los valores de las demás celdas de la tabla bidimensional en el caso 3, se aplica un procedimiento análogo al del caso 2 y es seleccionar el valor máximo entre dos opciones:

- El valor de la celda superior, que refleja la capacidad actual del camión sin incluir el elemento nuevo.

- El valor que resulta de restar el peso del elemento nuevo a la capacidad del camión, más el beneficio acumulado hasta ese momento, lo que representa la inclusión del elemento en la carga.

Esto es idéntico al algoritmo con memorización empleado en el caso 2, en ambos escenarios, se evalúa si conviene o no incluir el elemento en cuestión, basándose en el espacio disponible en el camión y el beneficio total acumulado.

A pesar de que la organización y el cálculo de los valores difieren entre el caso 2 y el caso 3, ambos enfoques llevan a la misma solución, la verdadera diferencia radica en la presentación y el procesamiento de los datos, no obstante, ambos métodos llegan a la búsqueda de la solución más eficiente para el dilema del camión con capacidad limitada.

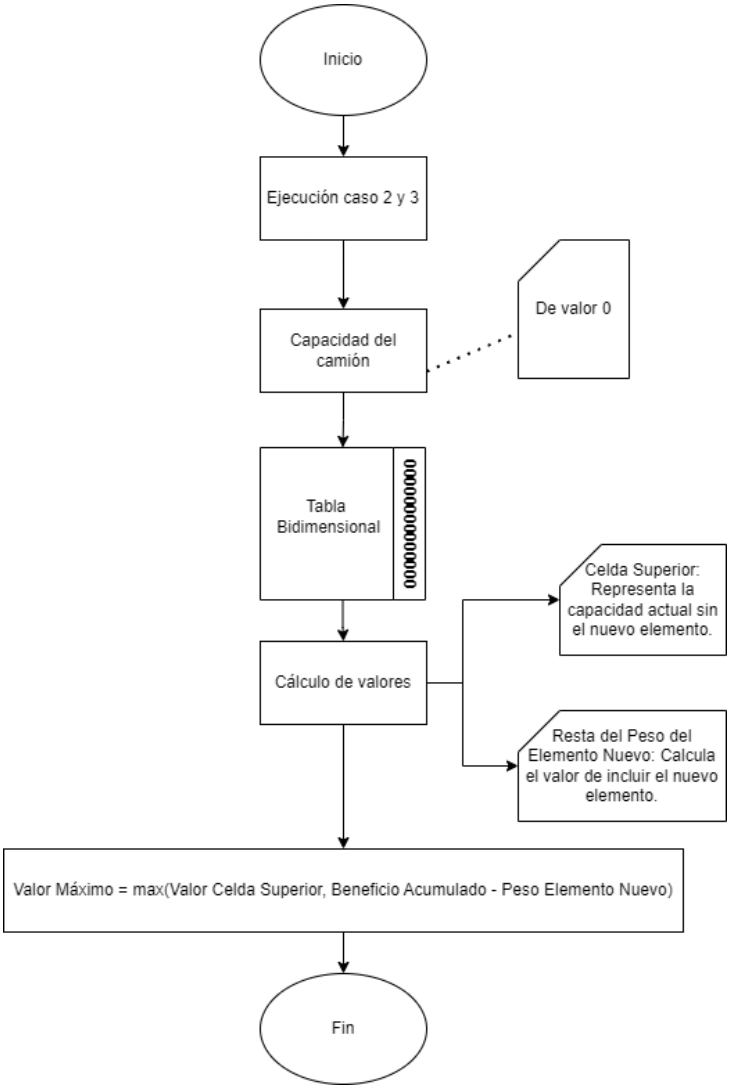

Imagen 3: Diagrama explicativo ejercicio 4.

3.2.5 Camión Programación dinámica 2.

Este método calcula el máximo beneficio posible que se puede obtener al llenar una mochila con capacidad limitada, utilizando un enfoque de PD.

Para resolver el problema de la mochila, se construye una tabla de manera iterativa para almacenar y actualizar los resultados de subproblemas.

- **camionDP2():** Esta función tiene una complejidad temporal de O(n*P),en ella se inicializa una tabla de tamaño n x P con -1, y luego llena la tabla con los beneficios máximos posibles para cada subproblema, cada celda de la tabla se llena en tiempo constante, por lo que la complejidad temporal total es **O(n*P)**.

- **getItemSolIterativa():** Esta función tiene una complejidad temporal de **O(n)**, ya que recorre los elementos una vez para determinar cuáles se incluyen en la solución óptima.

Como resultado la complejidad temporal total es **O(n*P)**.

3.2.6 Camión Programación dinámica 3.

En este se calcula el máximo beneficio posible que se puede obtener al llenar una mochila con capacidad limitada, utilizando PD.

Para resolver el problema de la mochila, es usado un array unidimensional, en este array se almacena el máximo beneficio alcanzado hasta el momento para cada capacidad de la mochila.

- **camionDP3():** Tiene una complejidad temporal de O(n*P), inicializa un array de tamaño P + 1 con 0, y luego llena el array con los beneficios máximos posibles para cada subproblema, cada celda del array se llena en tiempo constante, por lo que la complejidad temporal total es **O(n*P)**.

La complejidad temporal total del algoritmo es **O(n*P)**.

3.2.7 Ejercicio 8 Objetos no fraccionables con pesos decimales en PD.

En la programación dinámica, como en el problema de la mochila, se parte de la premisa de que los pesos son enteros, esto simplifica la representación de los pesos en una tabla donde cada columna representa un posible peso entero para el camión.

En caso de que los pesos sean decimales, la tabla requeriría un número exponencialmente mayor de columnas, volviéndola impracticable.

Para adaptar pesos decimales, una estrategia consiste en escalar los pesos multiplicándolos por 10^m donde ‘m’ es el número máximo de decimales.

Por ejemplo, si m=2 y tenemos un peso de 1.23, al multiplicarlo por 100 resulta en 123, lo que incrementa el número de columnas de la tabla a P×10m que de manera previsible consumirá una cantidad elevada de memoria.

Otra técnica implica el uso del Máximo Común Divisor (MCD). Se calcula el MCD de todos los pesos y se divide cada peso por este número para reducir el tamaño de la tabla, sin embargo, encontrar un MCD común para muchos objetos con pesos variados es poco probable.

Finalmente, redondear los pesos es una alternativa viable cuando el número de decimales es elevado, llevando a una solución aproximada debido a la pérdida de precisión al no considerar los decimales exactos.

3.2.8 Camión Voraz.

Este método resuelve el problema de la mochila utilizando un enfoque codicioso (greedy), con el objetivo de maximizar el beneficio por unidad de peso.

Determina qué elementos se incluyen en la mochila y devuelve una solución que indica la fracción de cada elemento que se incluye en la mochila para alcanzar el máximo beneficio posible.

Para el problema de la mochila 0/1 (o en este caso, el problema del camión) se obtiene:

- **camionGreedy():** Una complejidad temporal de **O(n^2)**, donde n es el número de elementos, en el peor de los casos para cada elemento, se recorre todos los elementos restantes para encontrar el que tiene la mayor relación beneficio/peso.

  Por lo tanto, la complejidad temporal total es O(n^2).

Es importante tener en cuenta que aunque este algoritmo es más eficiente en términos de tiempo que las versiones de programación dinámica, aunque no garantiza que se encuentre la solución óptima, ya que el algoritmo voraz selecciona la opción óptima en cada paso sin considerar las consecuencias futuras, lo que puede llevar a soluciones subóptimas en algunos casos.

3.3 Estudio experimental.

El programa de inicio solicita seleccionar una de las 9 opciones, para ejecutar los algoritmos propuestos es necesario generar una carga para el camión, esta puede ser generada de manera aleatoria en función del número de objetos deseado y el PMA.

```
LLENAR EL CAMIÓN CON LOS PRODUCTOS MÁS BENEFICIOSOS
-----------------------------------------------------
1. Generador de contenido
2. Ejercicio 1 camionRecursive
3. Ejercicio 2 camionTable
4. Ejercicio 3 camionDP
5. Ejercicio 5 camionDP2
6. Ejercicio 6 camionDP3
7. Ejercicio 9 camionGreedy
8. Test de rendimineto
9. Finalizar programa
-----------------------------------------------------
```

Imagen 4: Salida por pantalla LlenadoCamionMain menú de opciones.

```
Seleccione una opción: 1
1. Generador de carga aleatorio.
2. Generador de carga manual.
Seleccione una opción: 1
Generando carga aleatoria...
Introduzca el número de objetos: 10
Introduzca el peso maximo autorizad del camion: 50
CAMION (50)
Objeto 1        p=6,0        b=35,0
Objeto 2        p=49,0       b=50,0
Objeto 3        p=22,0       b=28,0
Objeto 4        p=15,0       b=65,0
Objeto 5        p=44,0       b=47,0
Objeto 6        p=12,0       b=69,0
Objeto 7        p=39,0       b=98,0
Objeto 8        p=18,0       b=46,0
Objeto 9        p=1,0        b=98,0
Objeto 10       p=30,0       b=33,0

Carga generada correctamente
```

Imagen 5: Salida por pantalla LlenadoCamionMain generador de cargas aleatorias.

Para una carga manual el programa solicitará el número de objetos disponibles para cargar, el PMA del camión, el nombre de los objetos, su peso y beneficio.

```
1. Generador de carga aleatorio.
2. Generador de carga manual.
Seleccione una opción: 2
Generando carga manual...
Introduzca el número de objetos: 3
Introduzca el peso maximo autorizado del camion: 40
Introduzca el nombre del objeto 1: a
Introduzca el peso entero del objeto 1: 25
Introduzca el beneficio del objeto 1: 34
Introduzca el nombre del objeto 2: b
Introduzca el peso entero del objeto 2: 15
Introduzca el beneficio del objeto 2: 20
Introduzca el nombre del objeto 3: c
Introduzca el peso entero del objeto 3: 20
Introduzca el beneficio del objeto 3: 26
CAMION (40)
a               p=25,0       b=34,0
b               p=15,0       b=20,0
c               p=20,0       b=26,0

Carga generada correctamente
```

Imagen 6: Salida por pantalla LlenadoCamionMain generador de cargas manual.

Cabe destacar que el programa cuenta con control de errores para que por ejemplo no se pueda seleccionar una opción no disponible o se introduzcan valores no permitidos.

```
LLENAR EL CAMIÓN CON LOS PRODUCTOS MÁS BENEFICIOSOS
-----------------------------------------------------
1. Generador de contenido.
2. Ejercicio 1 camionRecursive.
3. Ejercicio 2 camionTable.
4. Ejercicio 3 camionDP.
5. Ejercicio 5 camionDP2.
6. Ejercicio 6 camionDP3.
7. Ejercicio 9 camionGreedy.
8. Test de rendimineto.
9. Finalizar programa.
-----------------------------------------------------

Seleccione una opción: 2

Error: Genera primero una carga.

LLENAR EL CAMIÓN CON LOS PRODUCTOS MÁS BENEFICIOSOS
-----------------------------------------------------
1. Generador de contenido.
2. Ejercicio 1 camionRecursive.
3. Ejercicio 2 camionTable.
4. Ejercicio 3 camionDP.
5. Ejercicio 5 camionDP2.
6. Ejercicio 6 camionDP3.
7. Ejercicio 9 camionGreedy.
8. Test de rendimineto.
9. Finalizar programa.
-----------------------------------------------------

Seleccione una opción: -124124

Error: Debe introducir una opcion entre 1 y 9
Seleccione una opción: hola

Error: Debe introducir valor numerico
Seleccione una opción: 1
1. Generador de carga aleatorio.
2. Generador de carga manual.
Seleccione una opción: 213

Error: Debe introducir una opcion entre 1 y 2
1. Generador de carga aleatorio.
2. Generador de carga manual.
Seleccione una opción: 1
Generando carga aleatoria...
Introduzca el número de objetos: eda

Error: Debe introducir valor numerico
Introduzca el número de objetos: -12

Error: Debe introducir una opcion entre 1 y 2147483647
Introduzca el número de objetos: 2
Introduzca el peso maximo autorizad del camion: 3
CAMION (3)
Objeto 1        p=0,0        b=22,0
Objeto 2        p=1,0        b=84,0

Carga generada correctamente
```

Imagen 7: Salida por pantalla LlenadoCamionMain excepciones.

Se probarán todos los algoritmos con la misma carga generada de manera aleatoria, esta no será muy grande para facilitar su comprobación.

```
LLENAR EL CAMIÓN CON LOS PRODUCTOS MÁS BENEFICIOSOS
-----------------------------------------------------
1. Generador de contenido.
2. Ejercicio 1 camionRecursive.
3. Ejercicio 2 camionTable.
4. Ejercicio 3 camionDP.
5. Ejercicio 5 camionDP2.
6. Ejercicio 6 camionDP3.
7. Ejercicio 9 camionGreedy.
8. Test de rendimineto.
9. Finalizar programa.
-----------------------------------------------------

Seleccione una opción: 1
1. Generador de carga aleatorio.
2. Generador de carga manual.
Seleccione una opción: 1
Generando carga aleatoria...
Introduzca el número de objetos: 4
Introduzca el peso maximo autorizado del camion: 10
CAMION (10)
Objeto 1        p=9,0        b=28,0
Objeto 2        p=9,0        b=37,0
Objeto 3        p=3,0        b=0,0
Objeto 4        p=4,0        b=45,0

Carga generada correctamente
```

Imagen 8: Salida por pantalla LlenadoCamionMain carga seleccionada para las pruebas.

3.3.1 Camión Recursivo.

```
LLENAR EL CAMIÓN CON LOS PRODUCTOS MÁS BENEFICIOSOS
-----------------------------------------------------
1. Generador de contenido.
2. Ejercicio 1 camionRecursive.
3. Ejercicio 2 camionTable.
4. Ejercicio 3 camionDP.
5. Ejercicio 5 camionDP2.
6. Ejercicio 6 camionDP3.
7. Ejercicio 9 camionGreedy.
8. Test de rendimineto.
9. Finalizar programa.
-----------------------------------------------------

Seleccione una opción: 2
Beneficio final: Resultado [beneficioTotal=45.0]
```

Imagen 9: Salida por pantalla camionRecursive.

3.3.2 Camión Tabla.

```
Seleccione una opción: 3
X    X    X    X    X    X    X    X    X    X    X
X    0,0  X    0,0  X    X    0,0  0,0  X    X    28,0
X    X    X    0,0  X    X    0,0  0,0  X    X    37,0
X    X    X    X    X    X    0,0  X    X    X    37,0
X    X    X    X    X    X    X    X    X    X    45,0
Capacidad del camion = 10
Objeto 1        p=9,0        b=28,0
Objeto 4        p=4,0        b=45,0
Beneficio Total = 73.0
Peso Total =  13
```

Imagen 10: Salida por pantalla camionTable.

3.3.3 Camión Programación dinámica.

```
Seleccione una opción: 4
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  28,0 28,0
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  37,0 37,0
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  37,0 37,0
0,0  0,0  0,0  0,0  45,0 45,0 45,0 45,0 45,0 45,0 45,0
Capacidad del camion = 10
Objeto 4        p=4,0        b=45,0
Beneficio Total = 45.0
Peso Total =  4
```

Imagen 11: Salida por pantalla camionDP.

3.3.4 Camión Programación dinámica 2.

```
Seleccione una opción: 5
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  28,0 28,0
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  37,0 37,0
0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  0,0  37,0 37,0
0,0  0,0  0,0  0,0  45,0 45,0 45,0 45,0 45,0 45,0 45,0
Capacidad del camion = 10
Objeto 4        p=4,0        b=45,0
Beneficio Total = 45.0
Peso Total =  4
```

Imagen 12: Salida por pantalla camionDP2.

3.3.5 Camión Programación dinámica 3.

```
Seleccione una opción: 6
Resultado [beneficioTotal=45.0]
0.0, 0.0, 0.0, 0.0, 45.0, 45.0, 45.0, 45.0, 45.0, 45.0, 45.0
Beneficio final: 45.0
```

Imagen 13: Salida por pantalla camionDP3.

3.3.6 Camión Voraz.

```
Seleccione una opción: 7
Capacidad del camion = 10
Objeto 4 p=4.0 b=45.0
Beneficio Total = 45.0
Peso Total = 4.0
```

Imagen 14: Salida por pantalla camionGreedy.

3.3.7 Test de rendimiento.

Las entradas para esta muestra serán pequeñas para mejorar su comprensión.

3.3.7.1 test01NConstantePVariable.

```
Seleccione una opción: 8
1. N constante P variable.
2. N variable P constante.
Seleccione una opción: 1
Rendimiento para camion de 10 objetos
P       cRec    cTable  DP      DP2     DP3     Greedy
1       734500  18100   19300   169600  11200   44900
2       22800   9500    13400   54100   7700    34500
3       14000   11200   15200   32600   7300    26100
4       8400    11900   14100   30400   8100    19600
5       8600    12100   14800   33200   7700    25400
6       7800    13000   16700   31300   7900    23400
7       8300    12600   16600   34000   8700    24100
8       7900    11600   17000   35100   8600    19500
9       7200    25700   14900   30900   9800    18100
10      11300   13300   12400   25500   7000    12600
11      19400   11900   12500   19800   5900    8800
12      5800    15400   14600   27800   7300    11700
13      6300    17900   14500   29600   7600    14200
```

Imagen 15: Salida por pantalla test01NConstantePVariable.

3.3.7.2 test02NVariablePConstante.

```
Seleccione una opción: 8
1. N constante P variable.
2. N variable P constante.
Seleccione una opción: 2
Rendimiento para camion de capacidad 10
N       cRec    cTable  DP      DP2     DP3     Greedy
1       441500  13900   12100   130300  7200    9200
2       3900    6100    7000    13000   3900    6000
3       5700    7600    8800    14300   5000    7600
4       5200    6500    9000    16700   5000    10600
5       6400    8900    11500   20100   8000    8800
6       8500    7900    12900   23500   13500   11800
7       12100   20900   15400   29500   9200    20300
8       9200    10500   13700   28900   7700    15700
9       11300   13600   15800   22800   8800    15200
10      44100   17800   17800   42300   10500   22200
11      18500   22400   15900   34600   8800    12400
12      29800   18000   14500   34700   8800    16500
13      10800   12300   14600   22000   7200    9900
```

Imagen 16: Salida por pantalla test02NVariablePConstante.

4.- Anexo 1.

4.1 Diseño del código.

En este diagrama ilustrativo del procedimiento que sigue en línea general el programa se puede ver cómo interactuar con él para realizar las diferentes funciones y así aplicar los algoritmos realizados junto con los test de rendimiento.

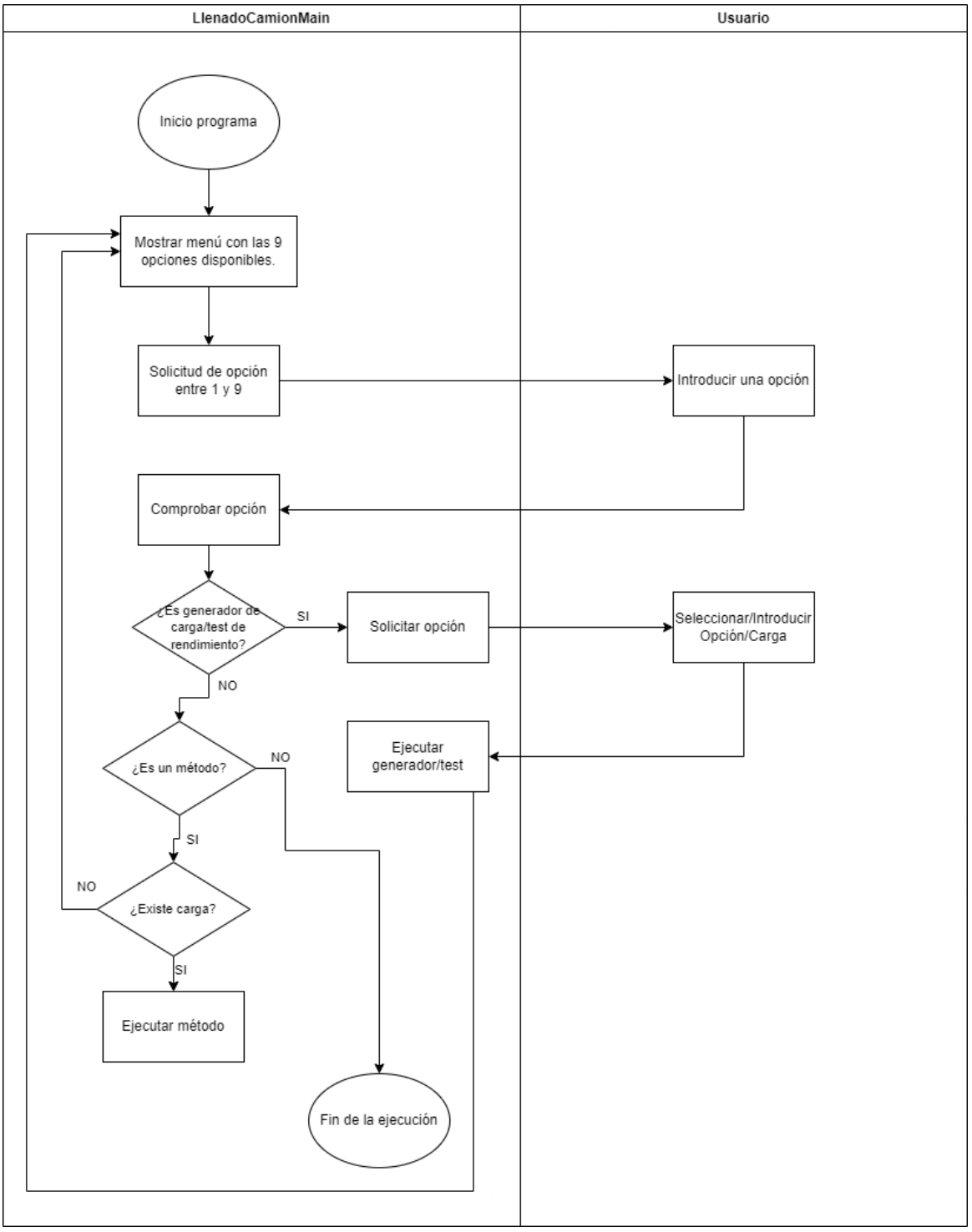

Imagen 17: Diagrama de funcionamiento general de la práctica03 implementada.

4.2 Diagramas de clase.

Haciendo uso de la herramienta PlantUML en eclipse, se han generado los diagramas de clases de la implementación realizada, en el que de manera global del proyecto obtenemos el siguiente:

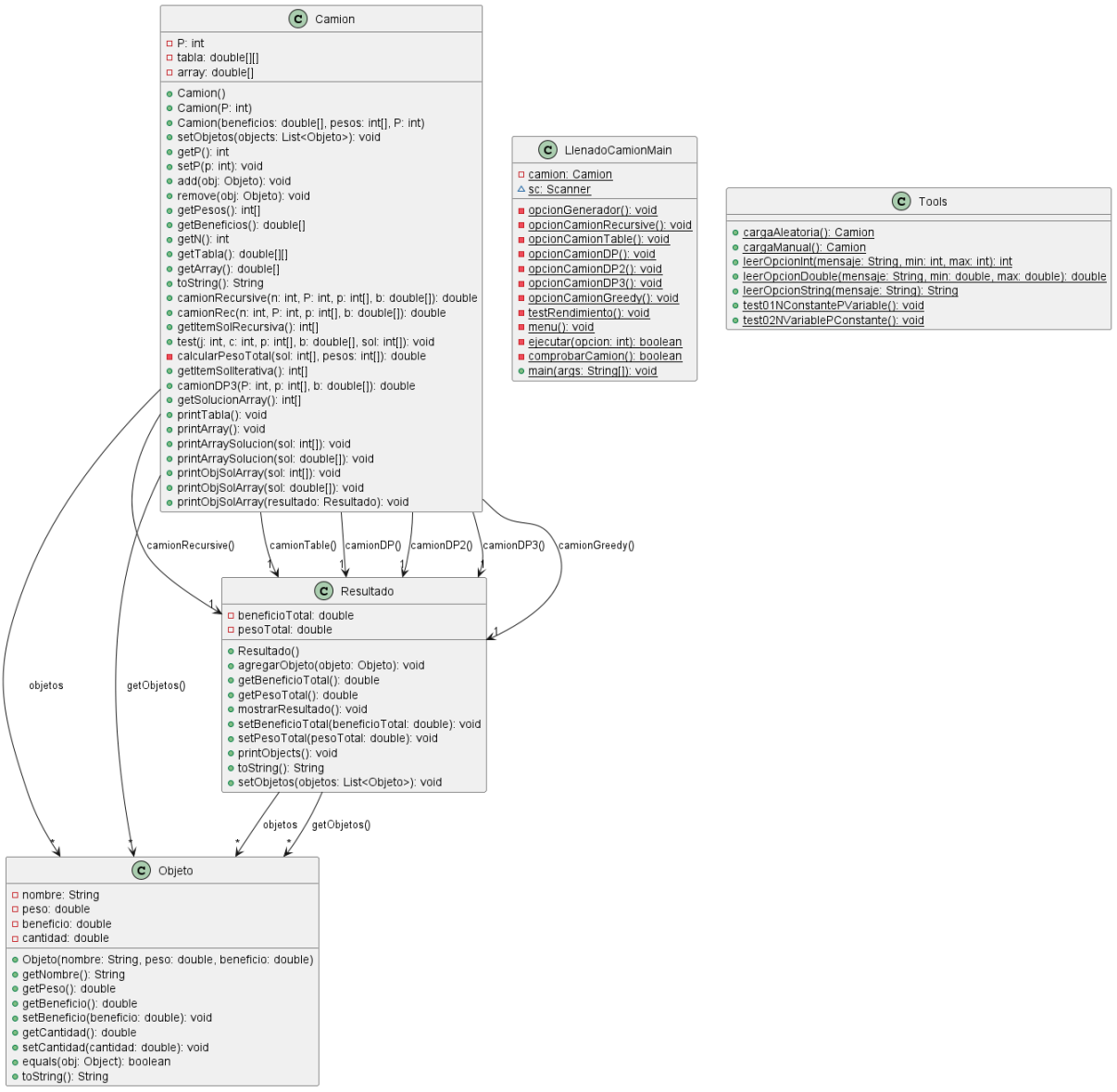

Imagen 18: Diagrama de clase general del proyecto.

4.2.1 Clase LlenadoCamionMain.

Atributos:

- El atributo estático de clase llamado camion almacena una instancia de la clase Camion que será usado para todos los métodos propuestos en esta práctica.

- El atributo *sc* es usado para el método Scanner de java y así poder introducir datos por pantalla.

Métodos destacables:

- Todos los métodos que comienzan por *opcion* llaman a sus respectivos métodos para así ser usados.

- testRendimiento: Permite ejecutar los test de rendimiento creados permitiendo seleccionar entre dos opciones.

- menu: Menú gráfico que indica al usuario las opciones disponibles creadas.

- ejecutar: Llama a los métodos solicitados por la práctica

- comprobarCamion: Comprueba que existe una carga actual para el camión y así ejecutar los algoritmos.

- main: Método de inicio de donde parte el programa y permite seleccionar una de las opciones del menú.

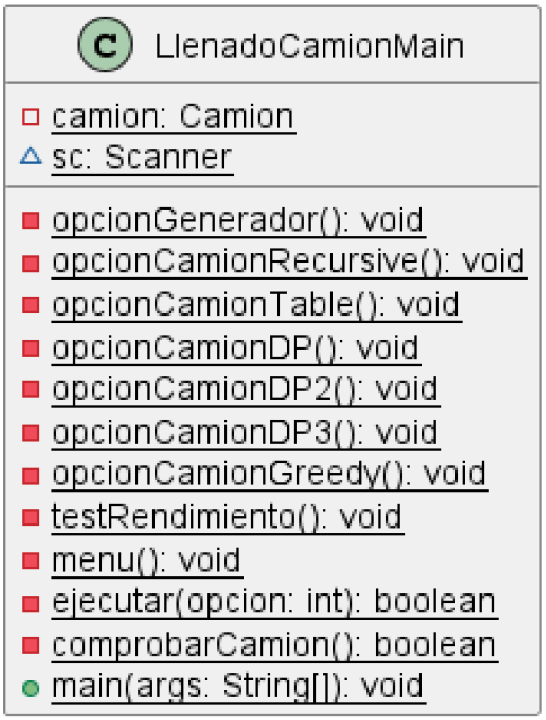

Imagen 19: Diagrama de la clase LlenadoCamionMain.

4.2.2 Clase Objeto.

Atributos: Tanto nombre, peso, beneficio y cantidad almacenan los datos que serán usados por el constructor.

Métodos destacables: Clase que implementa el constructor dados unos atributos, los getters y setters usados, equals para comparar dos objetos y toString para convertir a cadena de caracteres.

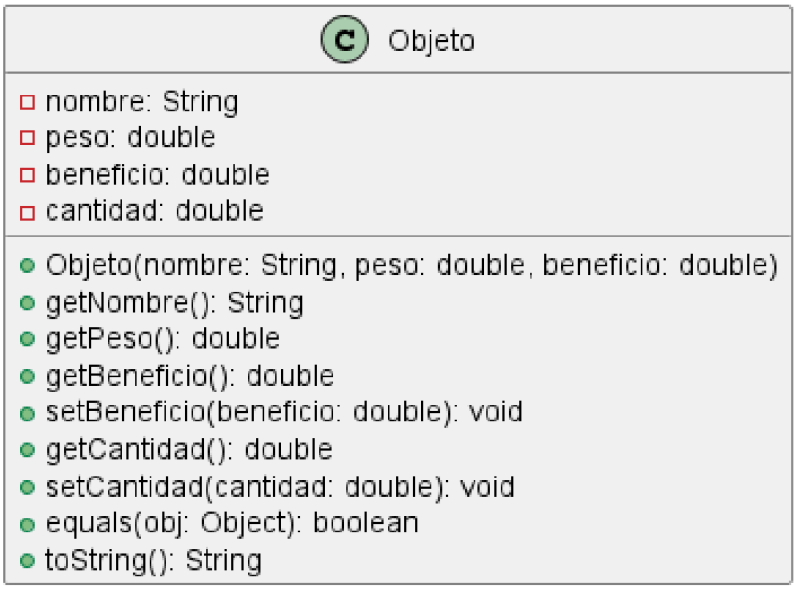

Imagen 20: Diagrama de laclase Objeto.

4.2.3 Clase Tools.

Esta clase es usada para aplicar diferentes herramientas que pueden ser reusadas en otras prácticas como es el caso de *leerOpcionInt* que ha sido rescatado de la práctica anterior con alguna modificación ya que en este caso verifica si los números son enteros y han sido también creados métodos para las cadenas de texto y los dobles.

Destacar finalmente que han sido guardados aquí los métodos para realizar test de rendimiento mejorando así la legibilidad del código y su modificación para otros test.

Métodos destacables:

- cargaAleatoria: Genera de manera sintética y rápida una carga de objetos para el camión, solicitando el número de objetos y el peso máximo autorizado.

- cargaManual: Mismo funcionamiento que el método anterior para generar una carga pero en este caso de manera manual, solicitando el nombre del objeto, el peso y el beneficio.

- leerOpcionInt: Función que lee la entrada introducida por el usuario y de no ser un número entero comprendido entre dos valores lanza un error.

- leerOpcionDouble: Función que lee la entrada introducida por el usuario y de no ser un string con decimales comprendido entre dos valores lanza un error.

- leerOpcionString: Función que lee la entrada introducida por el usuario y de no ser un string lanza un error.

- test01NConstantePVariable: Test de rendimiento para diferentes algoritmos de llenado de camiones con una cantidad constante de objetos (n) y una capacidad de camión variable (P) que ejecuta y cronometra los algoritmos especificados.

- test02NVariablePConstante: Test de rendimiento para diferentes algoritmos de llenado de camiones con una cantidad variable de objetos (n) y una capacidad de camión constante (P) que ejecuta y cronometra los algoritmos especificados.

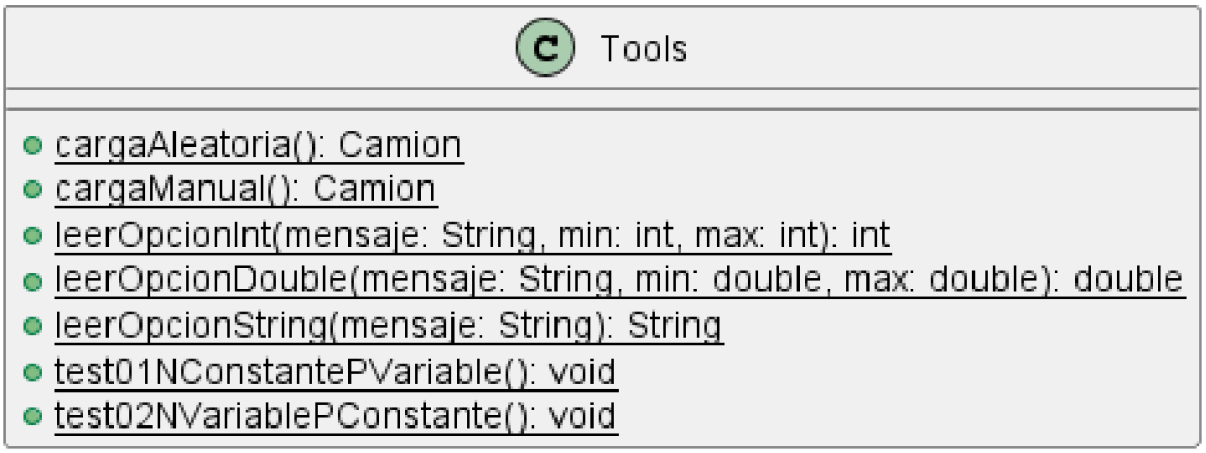

Imagen 21: Diagrama de la clase Tools.

4.2.4 Clase Resultado.

Esta clase es usada para mostrar el resultado de algunos métodos haciendo uso de getters y setters

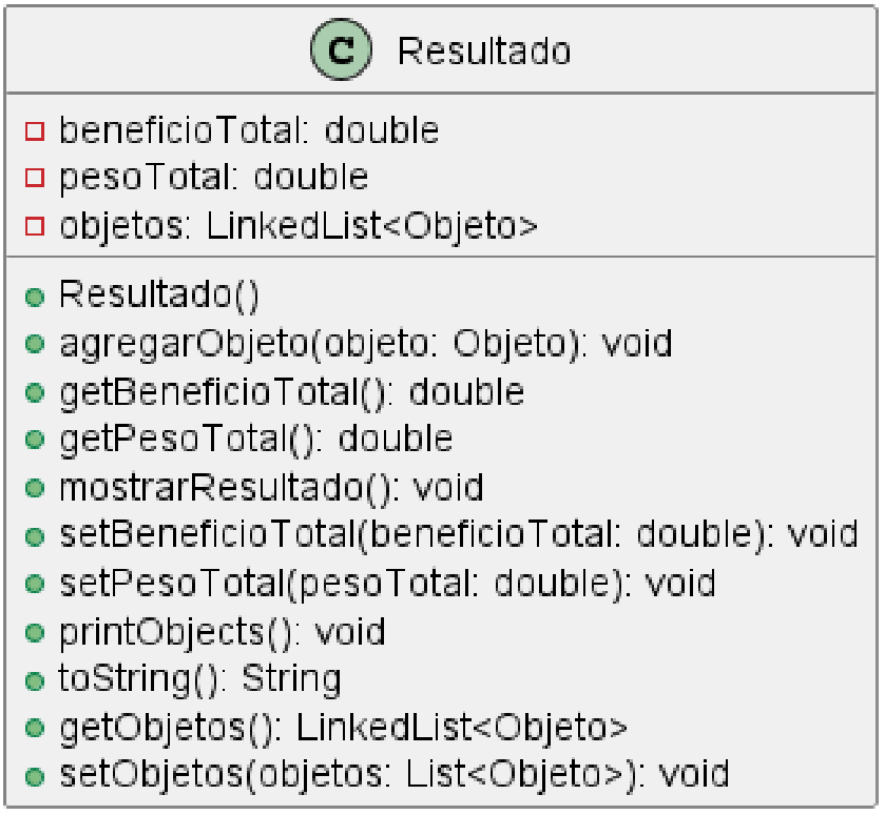

Imagen 22: Diagrama de la clase Resultado.

4.2.5 Clase Camion.

En esta clase se encuentran los algoritmos implementados en esta práctica haciendo uso de las cabeceras propuestas para mejorar su comprensión.

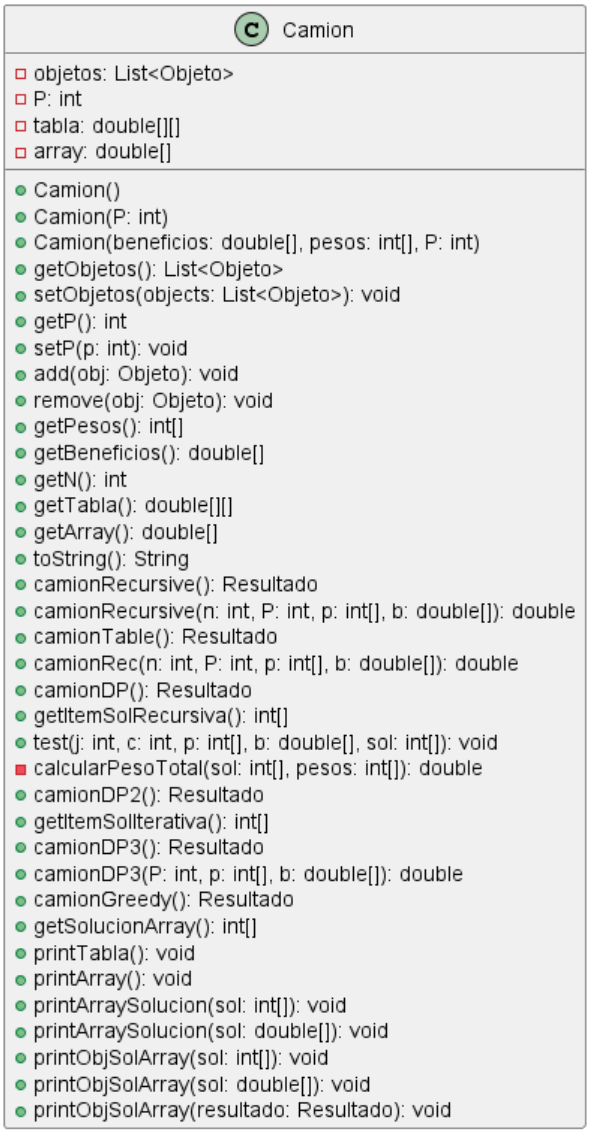

Imagen 23: Diagrama de la clase Camion.

4.3 Listado de archivos fuente.

La estructura usada ha sido muy similares a las anteriores en la que haciendo uso de la nomenclatura indicada se ha guardado en el paquete *org..eda2.practica03* el proyecto en código Java creado, en este caso no han sido creadas pruebas unitarias dado el tiempo que teníamos para el desarrollo de esta práctica, y finalmente en la carpeta *docs* se encuentran los documentos apartados como resolución de esta práctica.

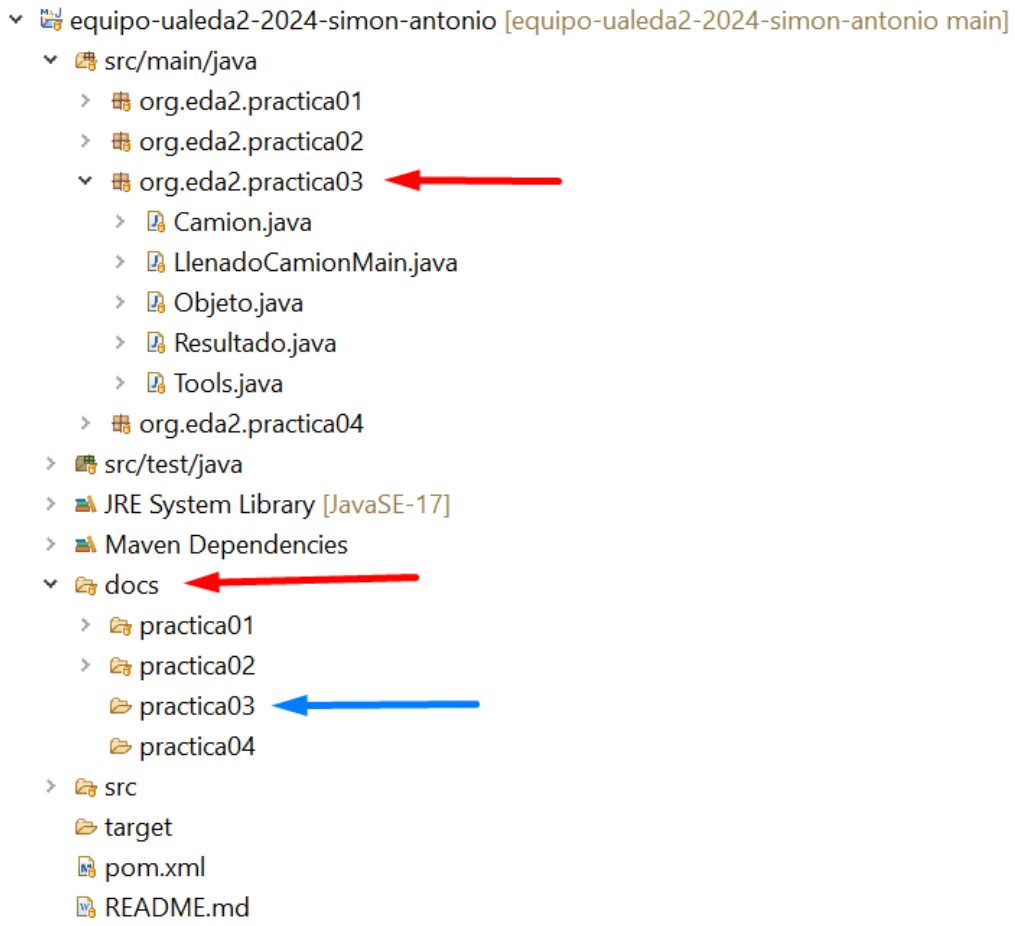

Imagen 24: Listado archivos fuente en eclipse para la práctica 3.

5.- Anexo 2.

5.1 Cálculos realizados.

Para el estudio experimental se han implementado 2 test de rendimiento, con lo que para realizar una prueba se mide el rendimiento de los algoritmos de la mochila diseñados, el tiempo ha sido tomado en nanosegundos (ns), los datos tomados son con 100 menos el recursivo:

5.1.1 test01NConstantePVariable.

Número de objetos constantes y capacidad del camión variable, este es interesante ya que se aprecia cómo al aumentar el peso del camión el tiempo de procesamiento aumenta debido a la capacidad y las combinaciones posibles, las fluctuaciones son dadas en los momentos que posee mayor número de combinaciones.

5.1.1.1 Camión Recursivo.

Dado que este método es recursivo se realiza su prueba por separado para poder introducir datos mayores en los demás, datos con 30 objetos.

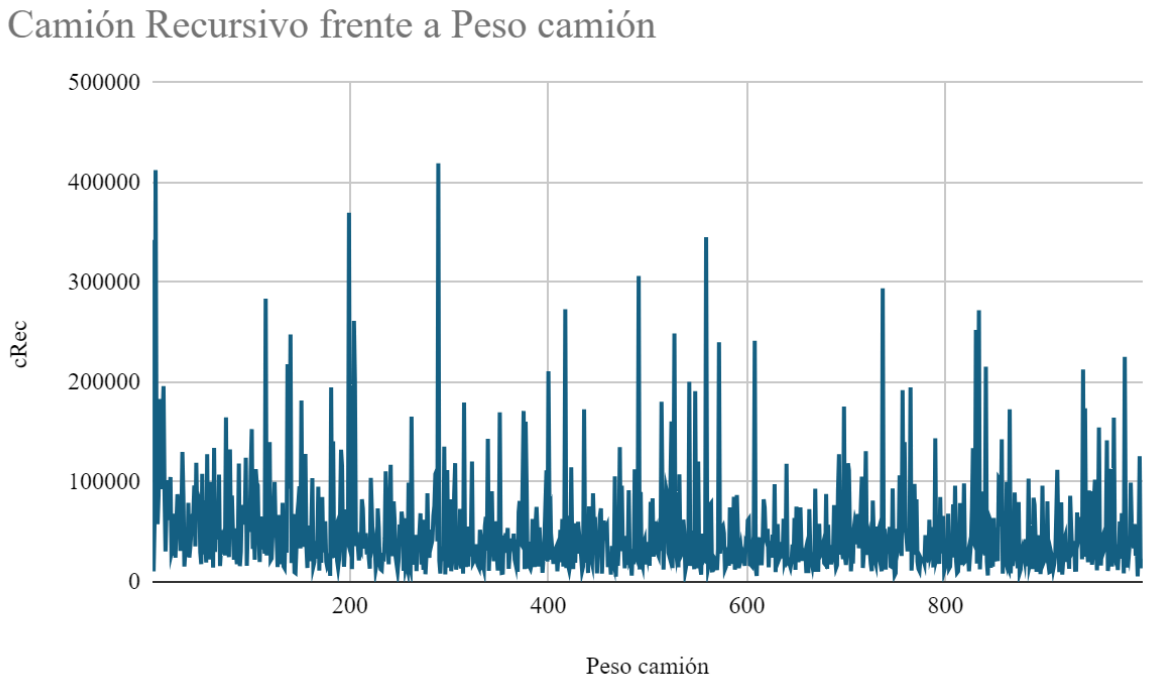

Imagen 25: Gráfico Camión Recursivo test01.

5.1.1.2 Camión Tabla.

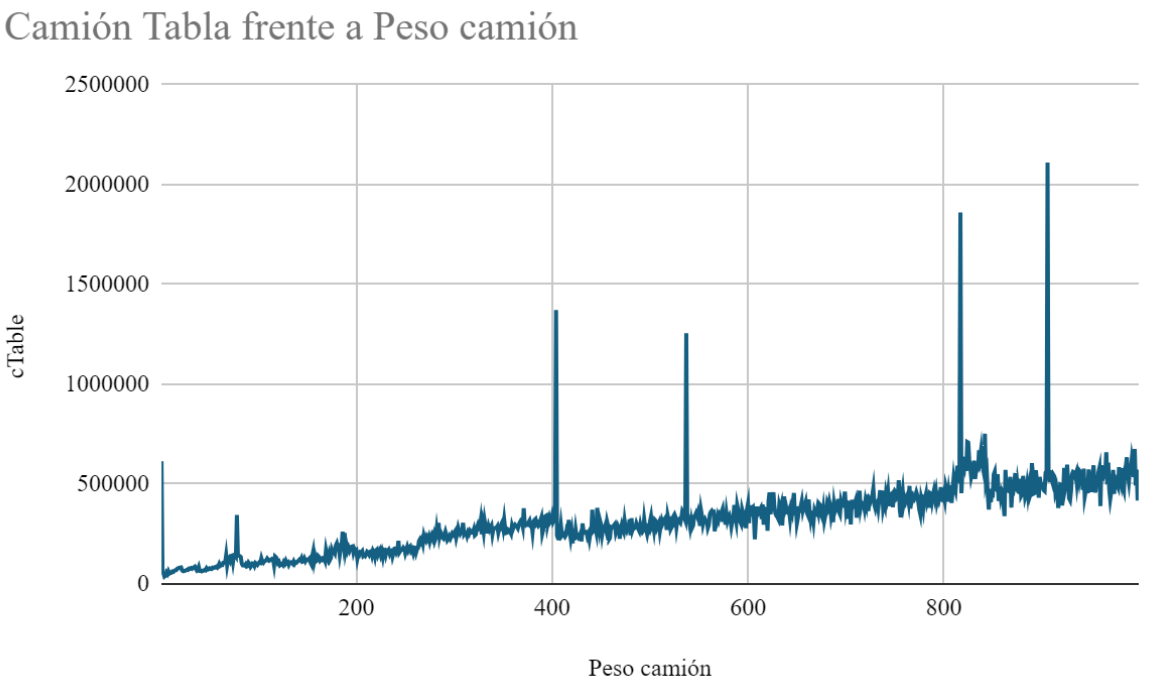

Imagen 26: Gráfico Camión Tabla test01.

5.1.1.3 Camión Programación dinámica.

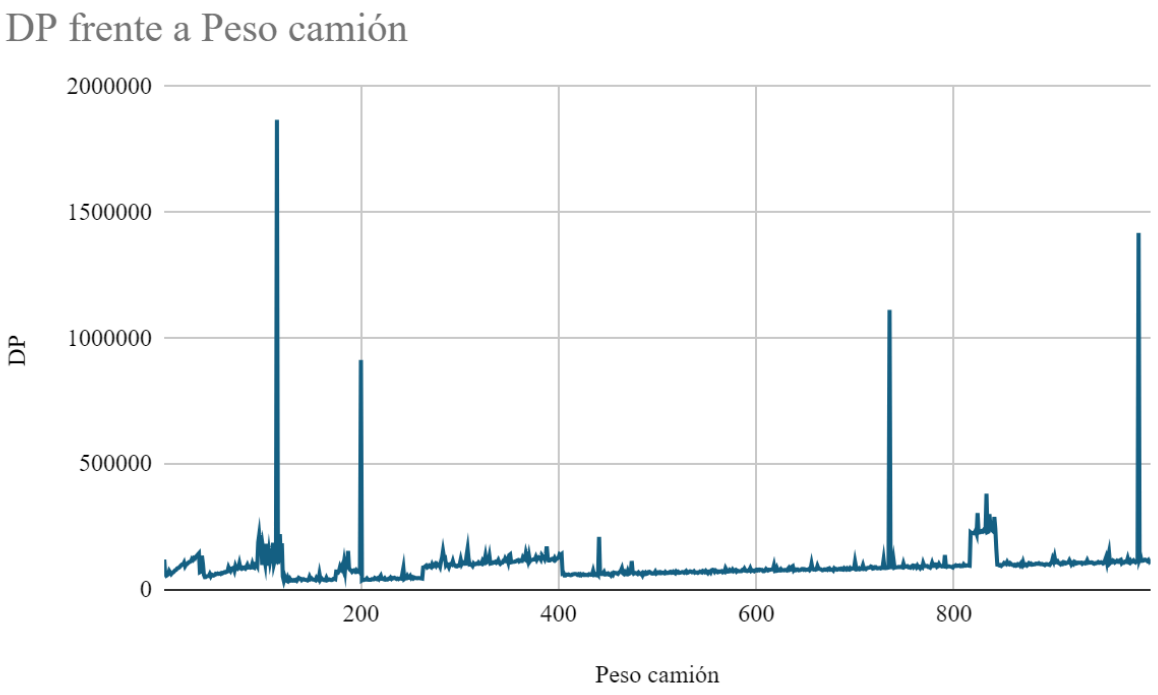

Imagen 27: Gráfico Programación dinámica test01.

5.1.1.4 Camión Programación dinámica 2.

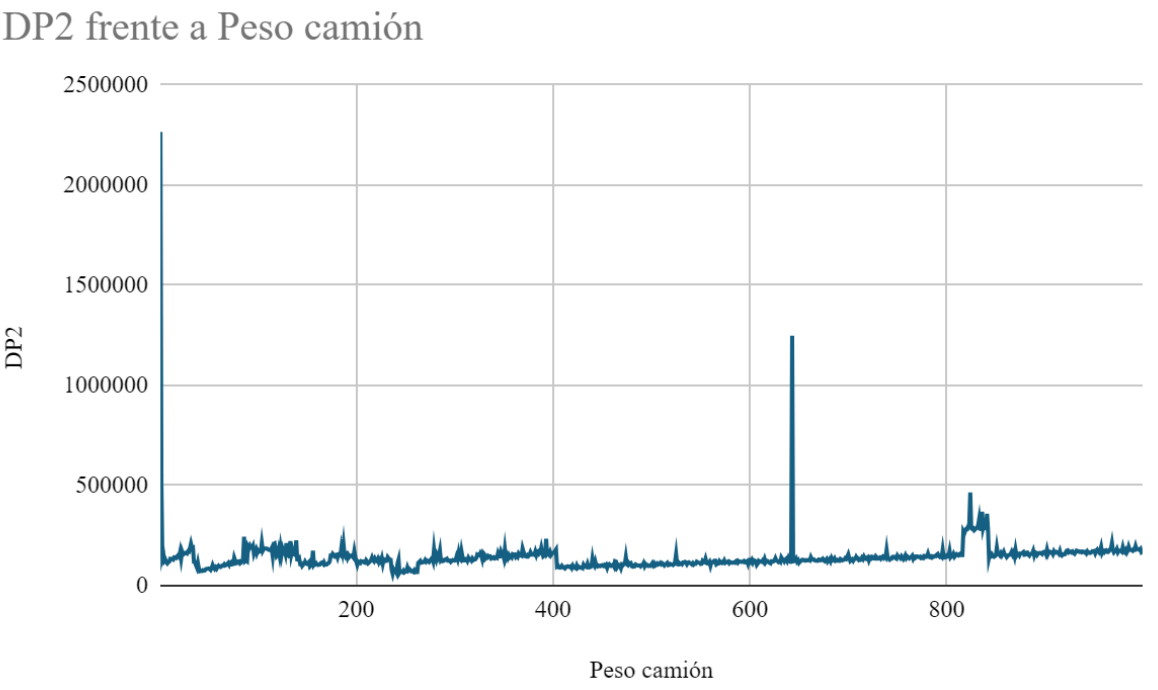

Imagen 28: Gráfico Programación dinámica 2 test01.

5.1.1.5 Camión Programación dinámica 3.

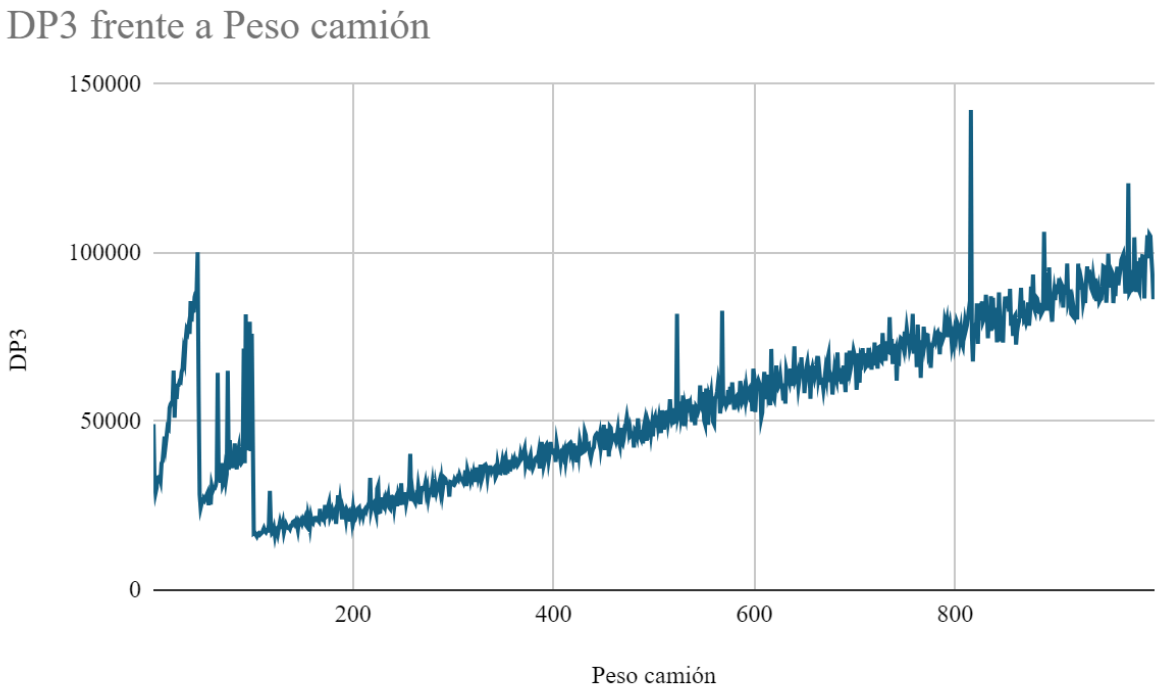

Imagen 29: Gráfico Programación dinámica 3 test01.

5.1.1.6 Camión Voraz.

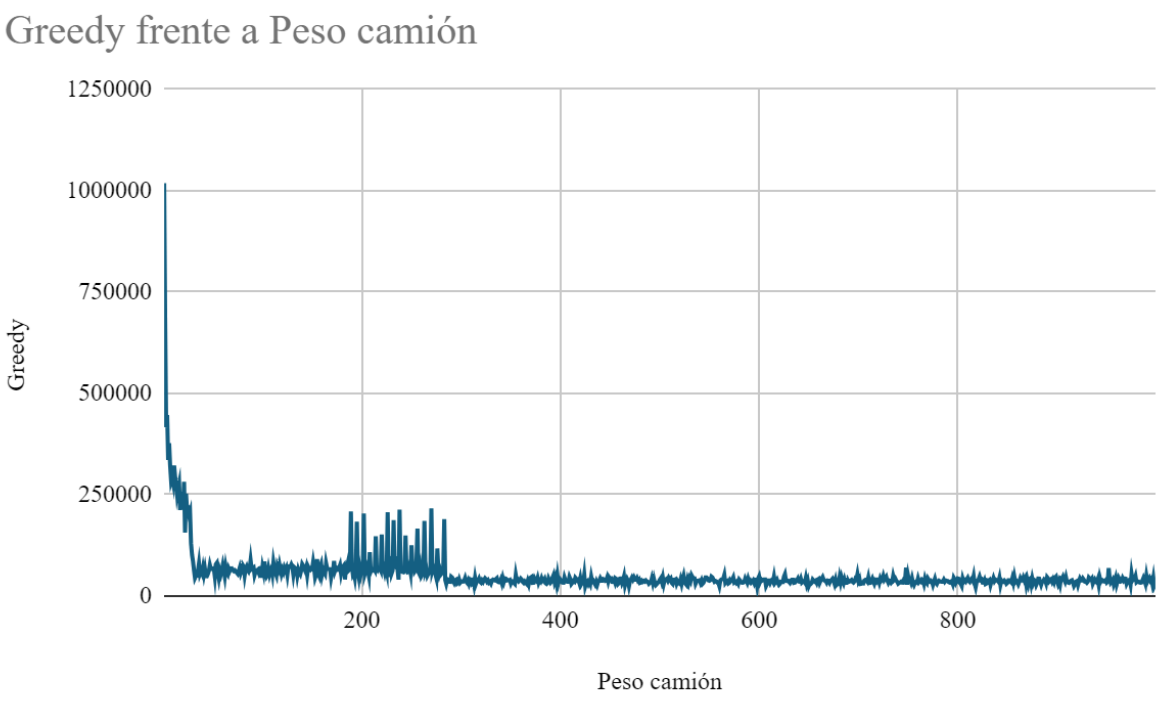

Imagen 30: Gráfico Camión Voraz test01.

5.1.2 test02NVariablePConstante.

Capacidad constante del camión y variando el número de objetos, en este test se han tomado menos valores para mejorar la representación de los datos, la capacidad del camión es de 10.

En el caso recursivo y en los demás se aprecia cómo sigue la línea de tendencia haciendo variar el número de objetos manteniendo una capacidad estable en el camión.

5.1.2.1 Camión Recursivo.

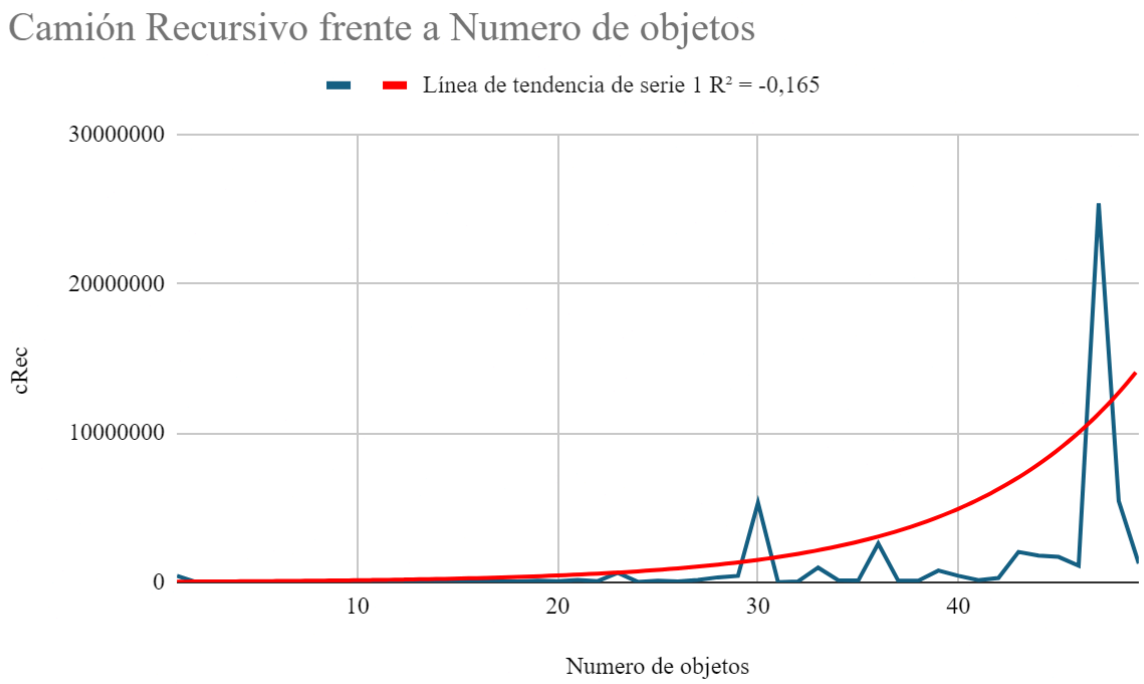

Imagen 31: Gráfico Camión Recursivo test02.

5.1.2.2 Camión Tabla.

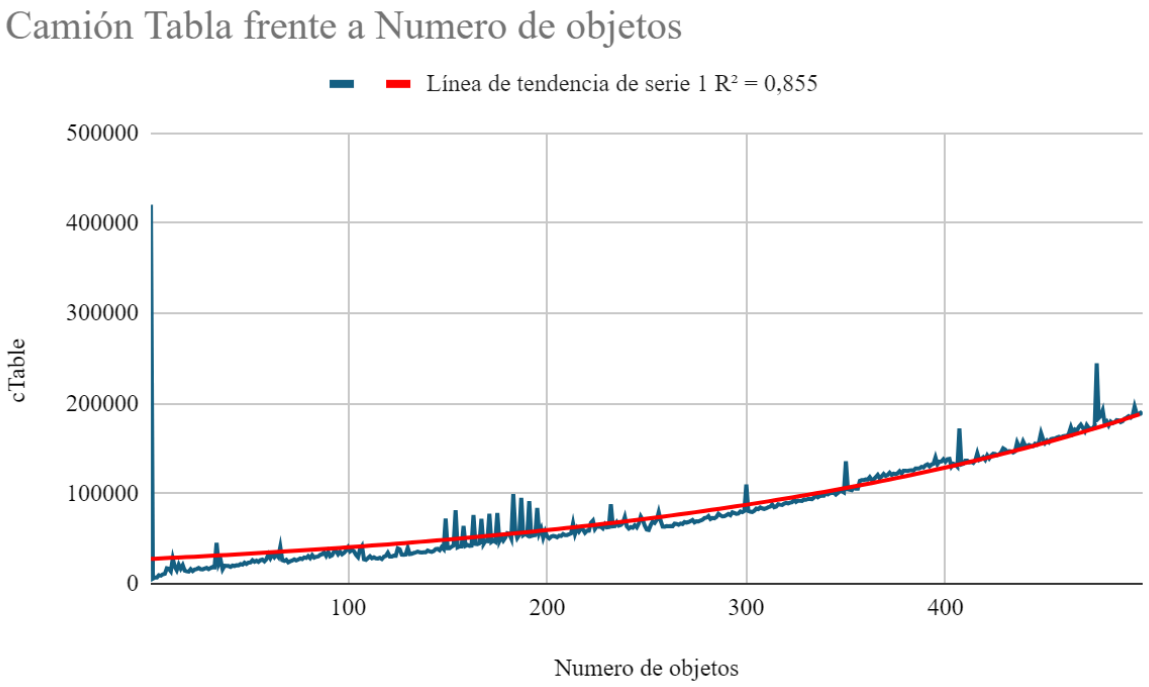

Imagen 32: Gráfico Camión Tabla test02.

5.1.2.3 Camión Programación dinámica.

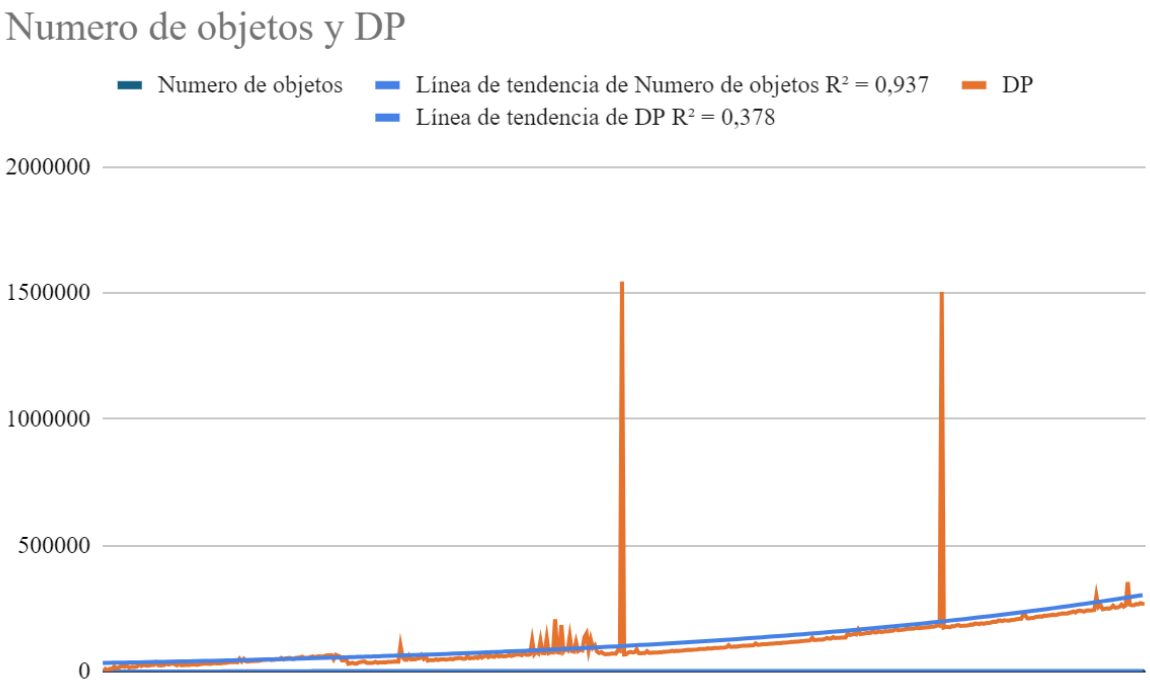

Imagen 33: Gráfico Programación dinámica test02.

5.1.2.4 Camión Programación dinámica 2.

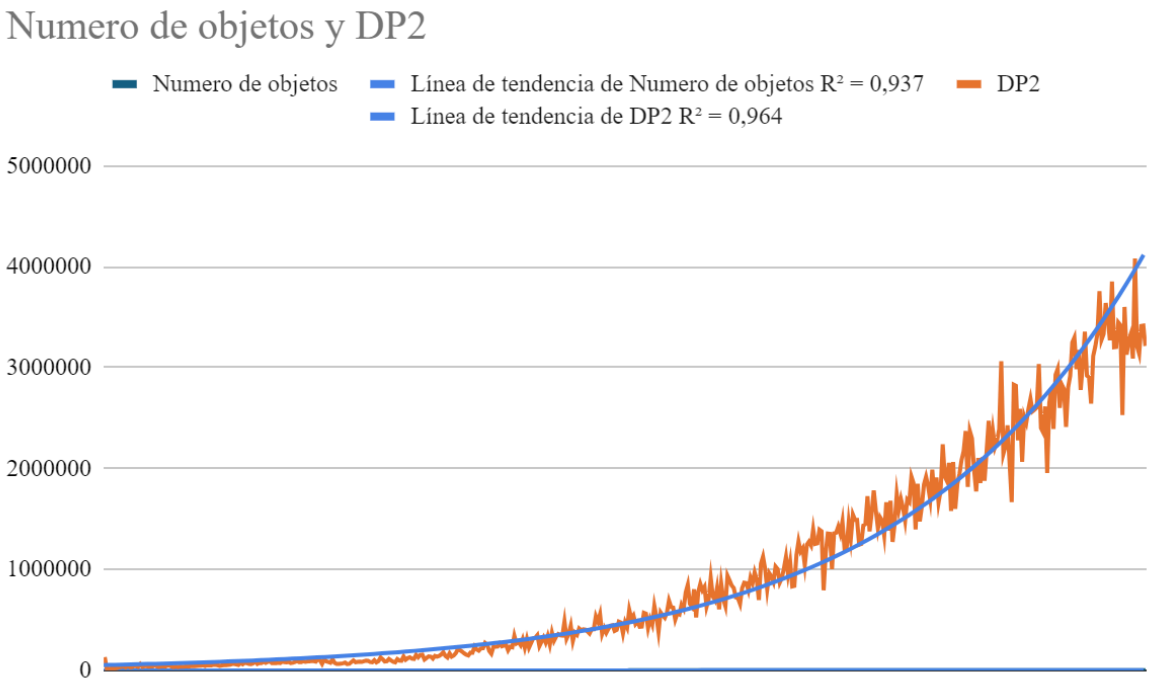

Imagen 34: Gráfico Programación dinámica 2 test02.

5.1.2.5 Camión Programación dinámica 3.

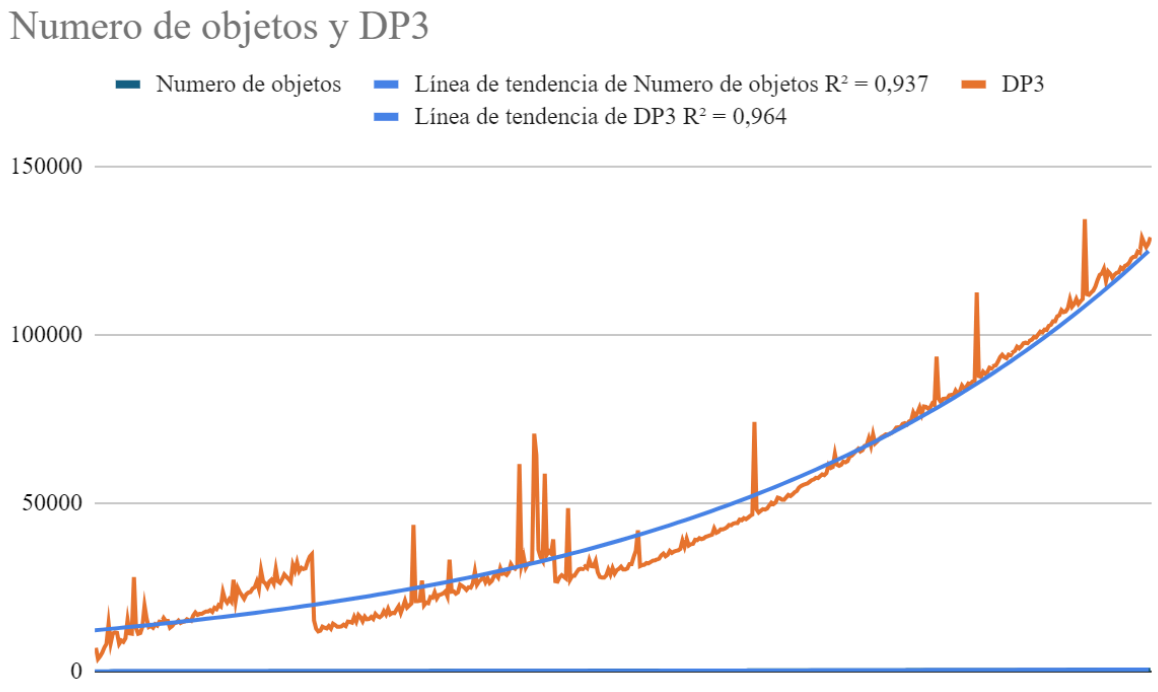

Imagen 35: Gráfico Programación dinámica 3 test02.

5.1.2.6 Camión Voraz.

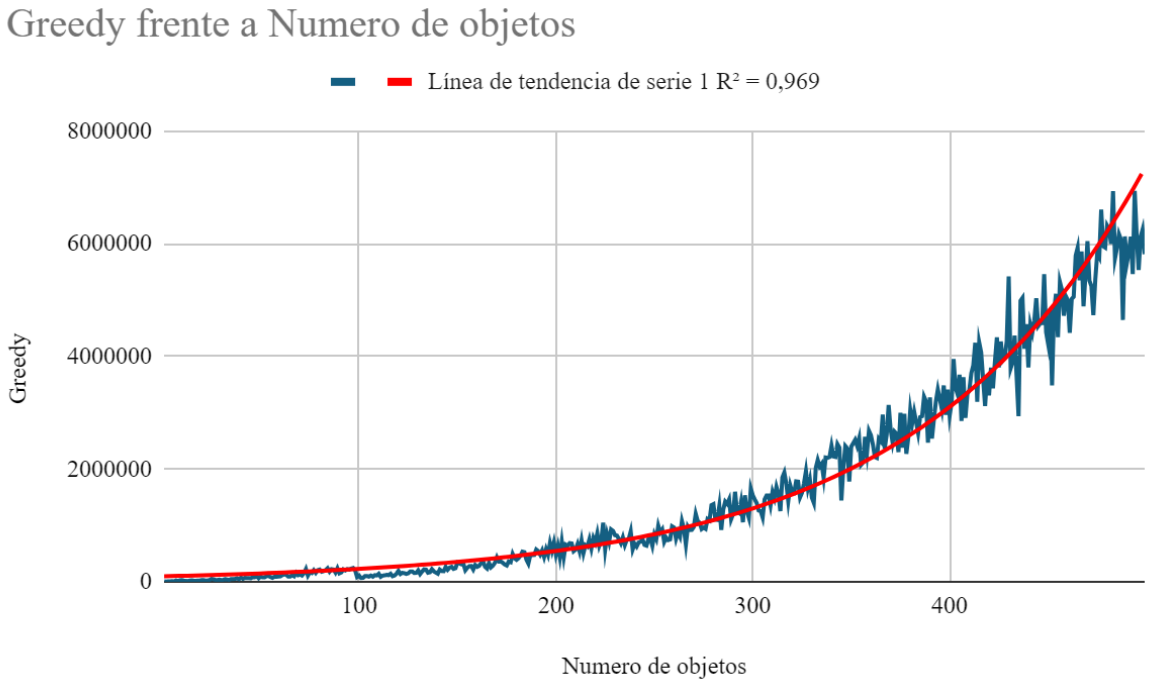

Imagen 36: Gráfico Camión Voraz test02.

5.2 Conclusiones finales.

En esta práctica han sido implementados numerosos métodos para la comprensión de la programación dinámica, para ello nos hemos propuesto mejorar la modularidad y la legibilidad del código para así como equipo poder trabajar ambos en un mismo problema y comprobar su funcionamiento rápidamente, cabe destacar que nos hubiera gustado desarrollar el ejercicio 7 propuesto pero dado el tiempo del que disponíamos consideramos que de manera global hemos comprendido y entrado de lleno en esta metodología de resolución de problemas con PD.

Se ha aprendido que de manera general para la PD los pasos para diseñarlo son:

1. Planteamiento de la solución como una sucesión de decisiones y verificación de que cumple el principio de optimalidad.

2. Definición recursiva de la solución.

3. Cálculo del valor de la solución óptima mediante una tabla que almacena soluciones a problemas parciales.

4. Construcción de la solución óptima utilizando la información contenida en la tabla.

6.- Bibliografía.

- UNE 157001:2014 Criterios generales para la elaboración formal. . . (s. f.). https://www.une.org/encuentra-tu-norma/busca-tu-norma/norma?c=N0052985
- Scribbr. (s. f.). Guía rápida de cómo citar en APA según su 7a edición. https://www.scribbr.es/category/normas-apa/
- Apuntes de clase Estructura de datos y algoritmos 2, Universidad de Almería.
- Apuntes de clase Lógica y algoritmica, Universidad de Almería.
- colaboradores de Wikipedia. (2024, 19 abril). Ecuación de Bellman. Wikipedia, la Enciclopedia Libre. https://es.wikipedia.org/wiki/Ecuaci%C3%B3n_de_Bellman
- Kariuki, C. (2023, 3 enero). Programación dinámica: qué es, cómo funciona y recursos de aprendizaje. Geekflare. https://geekflare.com/es/dynamic-programming/
- inDrive.Tech. (2023, 4 septiembre). Comprender la programación dinámica para poder utilizarla de forma eficaz. HackerNoon. https://hackernoon.com/es/comprender-la-programacion-dinamica-para-que-puedas-usarla-eficazmente
- Gautam, S. (s. f.). 0-1 Knapsack problem. https://www.enjoyalgorithms.com/blog/zero-one-knapsack-problem