El problema de la mochila entera

El problema de la mochila tiene un parecido con el algoritmo LCS y encontrar el camino más corto en un grafo dirigido y esto es obvio ya que la forma de solucionarlos es mediante el uso de la Programación Dinámica (PD), poco a poco me doy cuenta de que tan maravillosa es la PD ya que permite reducir las complejidades de los algoritmos, en donde, la mayoría llega a tener una complejidad lineal.

Ahora hablando sobre la práctica, encontramos 5 algoritmos diferentes para poder solucionar el problema, cada uno con diferentes valores de retorno y diferentes formas de atacar el problema, pero en esencia todos son equivalentes, sin embargo, mencionemos la gran diferencia de Java y Python, las estructuras de datos y los métodos correspondientes a cada estructura que nos proporciona Python son muy amables no nos dan casi nada de trabajo, como se diría coloquialmente nos lo *dan peladito y en la boca*, en cambio Java nos hace pensar, algo que considero benéfico para nuestra formación profesional, enfocando lo anterior en los 5 algoritmos en Python no hay gran problema en la implementación, sin embargo en Java tiene su chiste, programar los primeros 3 algoritmos no tiene misterio alguno, ya que uno nos lo proporciono usted y los otros dos son muy parecidos, lo único complicado era saber cómo representar la siguiente línea:

```
t = [[0 for m in range (M+1) ] for i in range (n+1)]
```

Viendo el resultado que muestra Python, no fue nada complicado saber lo que se tenía que hacer y que significa la variable t, básicamente era un arreglo de tamaño n+1 que contiene un arreglo con M+1 ceros, es decir la traducción a Java es la siguiente:

```
int t[][] = new int[n + 1][M + 1];
    for (int i = 0; i < n + 1; i++) {
        for (int m = 0; m < M + 1; m++) {
            t[i][m] = 0;
        }
    }
```

Después de esa parte literalmente es solo copiar y pasar a la sintaxis de Java el resto del algoritmo proporcionado, ahora esos tres algoritmos enfocados al uso de la clase ObjetoDeMochila, no es nada complicada, ya que contiene lo que ocupamos en el algoritmos base, pero ahora se encuentra en una estructura, solo nos resta usar el siguiente cambio y OJO esto es aplicable para el resto de los algoritmos: Sea p y b arreglos de int y objeto un arreglo de la clase ObjetoDeMochila, entonces cuando se quiera usar p[k] usaremos objeto[k].getPeso(), en caso de

querer usar b[k] usaremos objeto[k].getBeneficio(), esto para toda k que este en el rango de valores accesibles por el arreglo.

Ahora hablemos de los últimos 2 algoritmos, en mi opinión los más interesantes y satisfactorios de hacer.

Hablemos sobre la tercera versión del algoritmo, las primera líneas son iguales al resto, pero llega lo interesante:

```
            ant = act[:]
    x =[]
    m=M
    for i in range (n , 0, -1):
            x. insert (0, d[i][m])
            if d[i][m ]==1:
                    m = m-p[i-1]
return x, act[M]
```

En ese momento me pregunte, ¿Cómo rayos hago eso en Java?, lo que hice fue sencillo, paso a Java lo que conozco de primera mano y el resto lo investigo o mediante prueba y error, nos topamos con un act[:], ¿Qué significa el uso de "[:]" en Python? Según la documentación oficial el uso de [:] en una lista nos devuelve una copia superficial de la lista, entonces pensé ¿tenemos que hacer ant=act en Java no es así? La respuesta fue un no, los resultados no coincidían, en ese momento recordé el método clone() que nos proporciona Java, el método clone() crea una nueva instancia de la clase de objeto actual e inicializa todos sus campos con exactamente el contenido de los campos correspondientes de este objeto y se solucionó el problema, el último punto interesante fue resolver x=[], en java no podemos hacer eso, pero viendo el código nos damos cuenta de que el tamaño de x debe de ser n, listo solucionamos una parte, ahora el método x.insert(0,d[i][m]), por las ocasiones en que ocupe Python sé que lo que hace ese método es meter el valor del parámetro de la derecha en índice indicado por el parámetro de la izquierda y recorrer los demás valores, ejemplo: Tenemos los siguientes valores en x:

2	3	4	5	6	7	8	9	10

Si hacemos x.insert(0,d[i][m]) entonces x tendría el siguiente orden:

0	2	3	4	5	6	7	8	9	10

Al analizar el cómo hacerlo en Java llegue al siguiente código:

```
int[] x = new int[n];
    for (int i = n; i > 0; i -= 1) {
        int[] xTemp = new int[n];
        for (int j = 0; j < n - 1; j++) {
            xTemp[j + 1] = x[j];
        }
```

```
            x = xTemp.clone();
            x[0] = d[i][m];
            if (d[i][m] == 1) {
                m = m - p[i - 1];
            }
        }
```

Su funcionamiento es básico, crea un arreglo temporal con los valores recorridos para después clonar ese contenido en la x original y finalmente meter cómodamente el valor deseado en índice 0.

El retorno original es un arreglo y un int, para no complicarnos con ese retorno se imprime el arreglo y se devuelve el int, al final del día obtenemos el mismo resultado que en Python y hacer eso nos permite poder acomodarlos como la salida que muestra Python, todo esto en el caso de datos primitivos, para el caso de usar objetos se usa lo que se mencionó anteriormente, tenemos que aplicar los cambios correspondientes y el resultado será el mismo.

Ahora vayamos con el ultimo, este me costó trabajo, una anécdota pequeña: después de dormir desperté con la respuesta en mente de cómo hacer este algoritmo. La pregunta era la siguiente: ¿Cómo rayos hago eso, quitando el for evidentemente?

```
ant = [[0, []] for m in range (M +1)]
for i in range (1, n+1):
        act = [[0, [0 for j in range (i)]]]
```

Examinando en Python nos decía que eras listas, claro que Python como siempre combinando todo para nuestra comodidad, pero esa lista ¿Que contiene?, bueno ant es una lista que contiene un arreglo de longitud M+1 conformado por un 0 y un arreglo vacío del tipo int, se concluye ese de acuerdo al uso, después para act nos damos cuenta de que es un arreglo que contiene un 0 y un arreglo de tamaño i lleno con ceros. Después del sueño la respuesta era clara, hacer una estructura de datos, lo que en POO se conoce como clase, una estructura de datos compuesta por un valor entero y un ArrayList de enteros, MaximoBeneficio y Solución respectivamente, el uso del ArrayList es porque para ant necesitamos un arreglo de un tamaño que se desconoce (aunque se puede deducir) y que tenga la facilidad de modificarse sin usar muchas operaciones y ArrayList es la opción. Entonces tenemos que la traducción para ant a Java es la siguiente:

```
ArrayList<EstructuraDeDatos> ant = new ArrayList<EstructuraDeDatos>();
ArrayList<Integer> Arrayvacio = new ArrayList<Integer>();
for (int i = 0; i < M + 1; i++) {
```

```
        ant.add(new EstructuraDeDatos(0, Arrayvacio));
    }
```

Y para act:

```
ArrayList<EstructuraDeDatos> act = new ArrayList<EstructuraDeDatos>();
        act.add(new EstructuraDeDatos(0, Arrayvacio));
```

Aunque en act si se nos dice el tamaño que se debe de usar para el arreglo que contiene los ceros lo omitiremos, tiene un inconveniente que hay que arreglar, la solución se presenta más adelante.

Después de eso no hay misterio alguno que resolver, solo usar correctamente la estructura, usemos la siguiente transformación de sintaxis de Python a Java usando la estructura de datos (clase) definida:

ant [k]][0] es equivalente a usar ant.get(k).getMaximoBeneficio(), ya que queremos obtener el valor entero, en donde k es un entero y representa el índice del arreglo de donde queremos obtener ese dato.

Para: act.append ([b[i-1] + ant [m-p[i -1]][0] , ant [m-p[i-1]][1][:] + [1]]) es equivalente a hacer lo siguiente:

```
ArraysolucionTemp = (ArrayList<Integer>) ant.get(m - p[i - 1]).getSolucion().clone();
ArraysolucionTemp.add(1);
act.add(new EstructuraDeDatos(b[i - 1] + ant.get(m - p[i-1]).getMaximoBeneficio(),
ArraysolucionTemp));
```

En donde ArraysolucionTemp esta previamente inicializado, esto se hace ya que nos encontramos con un “[:]” que ya solucionamos anteriormente y el “+[1]” es ingresar un 1 al final del arreglo, para eso se usa el ArraysolucionTemp.add(1), al final ingresamos un objeto EstructuraDeDatos con los valores que se nos piden en el algoritmo.

Para: act.append ([ant[m][0], ant[m][1][:] + [0]]) es equivalente a hacer lo siguiente:

```
ArraysolucionTemp = (ArrayList<Integer>) ant.get(m).getSolucion().clone();
ArraysolucionTemp.add(0);
act.add(new EstructuraDeDatos(ant.get(m).getMaximoBeneficio(),
ArraysolucionTemp));
```

En donde ArraysolucionTemp esta previamente inicializado, esto se hace ya que nos encontramos con un “[:]” que ya solucionamos anteriormente y el “+[0]” es ingresar un 0 al final del arreglo, para eso se usa el ArraysolucionTemp.add(0), al

final ingresamos un objeto EstructuraDeDatos con los valores que se nos piden en el algoritmo, idéntico al punto anterior.

Y listo ya tenemos el ultimo algoritmo en Java, pero mencionamos que había un detalle en usar ArrayList para act, bueno el detalle es muy simple y su solución es igual de simple, al no tener un arreglo previamente llenado con una determinada cantidad de ceros al final del día puede o no que nos falte un cero al inicio, en caso de que no haga falta un cero la estructura se pasa en el estado en el que se encuentra, si no, hay que usar el siguiente código:

```
if (ant.get(M).Solucion.size() < n) {
        ant.get(M).Solucion.add(0);
        for (int i = n-1; i > 0; i--) {
            ant.get(M).Solucion.set(i,ant.get(M).Solucion.get(i - 1));
        }
        ant.get(M).Solucion.set(0, 0);
    }
```

Lo que hacemos en el código es sencillo metemos un 0 al final para después quitarlo debido al desplazamiento que se hace y al final metemos el 0 para completar la solución.

Para el caso de usar objetos se usa lo que se mencionó casi al inicio del documento, tenemos que aplicar los cambios correspondientes y el resultado será el mismo.

Para este algoritmo en su forma primitiva y con objeto regresa un objeto EstructuraDeDatos, el cual contiene el máximo beneficio y la solución, para lograr que el resultado en pantalla sea similar a el resultado en Python se creó un método llamado CrearStringSolucion el cual nos devuelve un string que contiene el formato con los datos para el arreglo que tiene la solución, solo resta en el main completarlo con el máximo beneficio y listo acabamos la práctica.

En conclusión, el problema de la mochila tiene varias soluciones que nos conducen al mismo resultado, unas nos proporciona únicamente el máximo beneficio y en otras nos regresa el máximo beneficio y un arreglo de solución, que básicamente son los objetos que se va a tomar para la mochila, vemos lo que se mencionó al principio del reporte la programación dinámica es muy importante y potente, ademas, de que aun a pesar de cambiar la forma de ataque al problema se obtiene el mismo resultado y en algunos casos hasta, nos da aún más información como en los últimos dos algoritmos de esta práctica.

Nota 1: Los valores de prueba son los mismos que tiene el documento de la práctica, en caso de querer probar otros valores se tendrá que modificar directamente en la clase ProbadorDeMochila

Nota 2: El código para el algoritmo discreto recursivo, fallaba en algunos casos como el siguiente: p = {1,3,4,5}, b = {1,4,5,7} y M=7; no nos daba el mismo resultado que los otros algoritmos ver Imagen 1

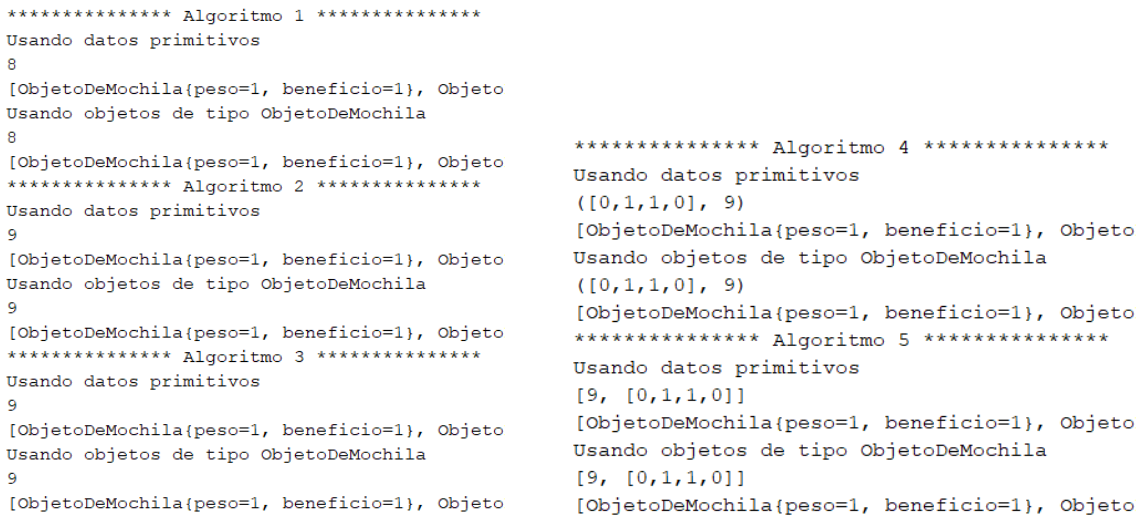

Imagen 1. Resultado del ejemplo, algoritmo 1 con error

El error es muy sencillo, solo faltaba agregar un símbolo “=” en la condición:

if (objetos[i - 1].getPeso() < M)

Ahora poniendo el símbolo “=”

if (objetos[i - 1].getPeso() <= M)

Con ello obtenemos que todos los resultados coincidan ver Imagen 2.

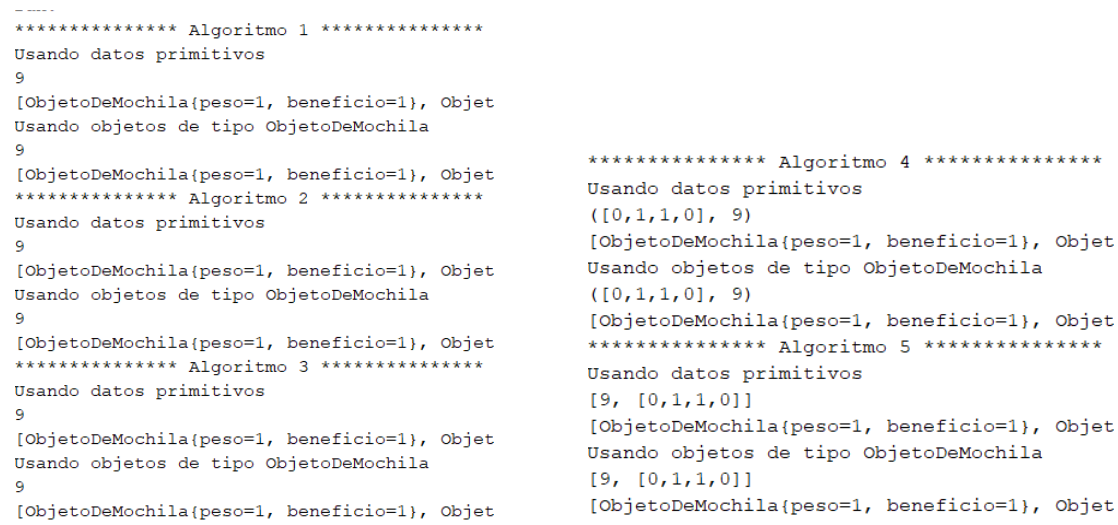

Imagen 2. Resultado del ejemplo, algoritmo 1 sin error