El problema de la mochila fraccionaria

Para esta segunda parte de la practica sobre algoritmos ávidos volvemos a ver el problema de la mochila, pero en esta ocasión, en lugar de buscar que entre una cantidad exacta de objetos en la mochila, pero en para el problema de la mochila fraccionara podemos partir un objeto, es decir, en lugar de meter 1 podemos meter 0.2,0.5,0.6 de un mismo objeto, provocando así que en lugar de poder llevar dos objetos que cumplen con el peso limite, podríamos llevar los mismos dos o más objetos pero en esta ocasión serian fraccionados. Ahora nos preguntamos cuales son las condiciones para que la decisión del algoritmo cambie de en lugar tomar 2 objetos tomar más, la respuesta se encuentra en cómo se presentaran los datos en el algoritmo.

Para lo anterior se tomaron en cuenta las siguientes restricciones para el arreglo de ObjetosMochila:

1. Tomar un conjunto de tuplas de valores (beneficio, peso) para cada objeto sin algún tipo de orden y ver qué resultado arroja el algoritmo sin sobrepasar *M*.
2. Ordenar los elementos por ejemplo según el beneficio de mayor a menor y llevar exactamente *M* peso en la mochila (Valor Máximo).
3. Peso<*M* para maximizar el número de objetos que se llevan en la mochila.
4. Mejor Rentabilidad en donde

$$Rentabilidad = \frac{beneficio}{peso}$$

El caso 1, es donde simplemente se presenta el arreglo de la misma forma con la que se creó, para el segundo caso se acomoda de acuerdo con el beneficio de mayor a menor, el tercero se acomodó de acuerdo con el peso de menor a mayor y el ultimo se acomoda de acuerdo con la rentabilidad de cada pareja de valores, siguiendo la formula anterior.

Para poder llevar a cabo el cómodo de los datos se usó QuickSort la cual es una técnica de ordenamiento que ocupa el paradigma de vencer, el cual es un complemento perfecto para este algoritmo voraz, en caso de querer los ítems de la mochila acomodados en alguno de los casos anteriores. Podemos deducir lo anterior ya que, si se tienen muchos elementos a elegir para la mochila y queremos ordenarlos, el QuickSort nos viene de maravilla ya que su complejidad es O(nlogn) lo cual es bastante rápido comparado con otros algoritmos de ordenamiento como lo es BubbleSort.

Antes de pasar a la conclusión veamos cómo afecta el cómo se presentan los datos al algoritmo para la toma de decisión que va a hacer.

Vemos en la Imagen 1 que cada tipo de decisión es diferente y es que deseamos obtener cosas diferentes, en la primera vemos que sin ordenar nos sale eso y lo único que remos es tener el máximo beneficio posible.

Sanchez Mendez Edmundo Josué
Análisis de Algoritmos

Para el punto 2 queremos obtener el beneficio mayor por lo que obtendremos un beneficio mayor o en ocasiones igual a las demás, mencionar que ya empezamos a ver porque es el problema de la mochila fraccionaria y es que ya nos aparece una fracción en la solucionada, para el punto 3 vemos que obtenemos los primeros objetos del resultado obtenido de ordenar de menor a mayor los pesos, y el resultado nos da pesos que cumplen con M obteniendo el resultado mostrado en la imagen 1, para el punto 4 queremos obtener la mayor rentabilidad, mencionar que el orden es de mayor a menor, como podemos ver si sumamos las rentabilidades de todos los casos veremos que no hay ninguna mayor a la del caso 4.

```
******************Arreglo sin ordenar******************
ObjetoDeMochila{peso=60.0, beneficio=50.0, rentabilidad=0.8333333, solucion=1.0}
ObjetoDeMochila{peso=40.0, beneficio=40.0, rentabilidad=1.0, solucion=1.0}
******************Ordenando por beneficio de mayor a menor******************
ObjetoDeMochila{peso=30.0, beneficio=66.0, rentabilidad=2.2, solucion=1.0}
ObjetoDeMochila{peso=50.0, beneficio=60.0, rentabilidad=1.2, solucion=1.0}
ObjetoDeMochila{peso=60.0, beneficio=50.0, rentabilidad=0.8333333, solucion=0.33333334}
******************Ordenando por peso de menor a mayor******************
ObjetoDeMochila{peso=10.0, beneficio=20.0, rentabilidad=2.0, solucion=1.0}
ObjetoDeMochila{peso=20.0, beneficio=30.0, rentabilidad=1.5, solucion=1.0}
ObjetoDeMochila{peso=30.0, beneficio=66.0, rentabilidad=2.2, solucion=1.0}
ObjetoDeMochila{peso=40.0, beneficio=40.0, rentabilidad=1.0, solucion=1.0}
******************Ordenando por rentabilidad de mayor a menor******************
ObjetoDeMochila{peso=30.0, beneficio=66.0, rentabilidad=2.2, solucion=1.0}
ObjetoDeMochila{peso=10.0, beneficio=20.0, rentabilidad=2.0, solucion=1.0}
ObjetoDeMochila{peso=20.0, beneficio=30.0, rentabilidad=1.5, solucion=1.0}
ObjetoDeMochila{peso=50.0, beneficio=60.0, rentabilidad=1.2, solucion=0.8}
BUILD SUCCESSFUL (total time: 0 seconds)
```

Imagen 1. Resultados de ejecución del programa.

En conclusión, para esta práctica de forma general, vemos que los algoritmos voraces nos dan una solución rápida al problema, claro que esta puede no ser la más optima de todas, mencionar que me parece muy interesante esta forma de atacar algunos problemas, en especial me gusta el problema de los códigos de Huffman ya que su aplicación es muy importante, también nos damos cuenta de que la forma en que los datos sean presentados al algoritmo este cambiara en el resultado y esto se debe a lo mencionado anteriormente y es que es una solución rápida más puede no ser la más optima, tocando un poco el algoritmo QuickSort para ordenamiento nos damos cuenta de su velocidad, la cual es muy alta. Si pensamos aplicar el programa anterior a casos en donde hay muchos objetos a elegir y queremos tener un cierto orden la unión de QuickSort con este algoritmo ávido nos viene muy bien.

Nota: Notar que los resultados también van acordes a como se ordenaron los datos, ejemplo, para la mayor rentabilidad, esta se ordena de mayor a menor y los objetos se muestran de mayor a menor tomando en cuenta como restricción su rentabilidad, mencionar que la clase main es ProbadorDeMochila.