Transport

You have a transport plane that has to deliver items to remote locations. You would like to load all the items on the plane, but you cannot exceed the plane's weight capacity, `capacity`. Given a List of Packages `List<Package> items` of known weights and values, find the most valuable subset of the Packages that fit into the plane without exceeding the capacity. That is, the combined weight of all the packages is less than or equal to the `capacity`.

You have been provided a completed the Package class as shown below:

```
public class Package  {
   private int weight;
   private int value;

   public Package(int w, int v)
   {
      weight = w;
      value = v;
   }

   public int getWeight()  {  return weight;  }

   public int getValue()  {  return value;  }

   public int hashCode()
   {
      Integer w = new Integer(weight);
      Integer v = new Integer(value);
      return w.hashCode() + v.hashCode();
   }

   public boolean equals(Object obj)
   {
      Package p = (Package) obj;
      return value == p.getValue() && weight == p.getWeight();
   }
}
```

Your task is to complete the `getMaxCargo()` method in the `Transport` class. The method `getMaxCargo` returns the max value possible for the available `Package`s.

```
public class Transport
{
   private List<Package> items;
   private int capacity;

   public Transport(List<Package> p, int cap)
   {
      items = p;
      capacity = cap;
   }

   public int getMaxCargo()
   {
      return -1;
   }
}
```

You are correct, this is basically the famed Knapsack problem (https://en.wikipedia.org/wiki/Knapsack_problem) which is an NP problem (https://en.wikipedia.org/wiki/NP_(complexity)). But we are going to solve it anyways, because the

data will be small enough. The following is an attempt to describe the algorithm I used to solve this problem. Please note that in my solution, `getMaxCargo()` contained fewer than 30 lines of code and two short helper methods.

The general outline of my algorithm was as follows:

Assume `items` is the `List` of all possible `packages` that may be transported.

1. Create a `List<ArrayList<Packages>>, p,` to store all possible subsets of packages.
2. Loop through each `item` in `items` (all possible packages).
    a. Loop through each `package` in `p`.
        i. Make a new copy of `package`
        ii. Add `item` to the copy of `package`
        iii. Add the copy to the end of `p`.
    - Note: you need to control the loop over `p` (Package subset) to prevent the loop from processing the added subsets. I accomplished this task by getting the size of `p` (`int temp = p.size()`) between steps 1 and 2 and used `temp` to replace `p.size()` as the upper bound in the step 2 loop.
    - Added note: After thinking about it, I guess I could have looped backwards over `p`.
3. Search (loop through `p`) for the max value package subset that does not violate the weight requirement.
4. Return the value of the package with max value that does not violate the weight requirement.

- Note – You could improve the efficiency of this algorithm by checking the weight requirement before adding the new subsets in step 2ii.

Sample trace of steps 1 and step 2 from the algorithm described above:

Assume: `List<<Packages> items` contains packages a, b, c and d.

Step 1: `List<ArrayList<Packages>> p` is empty

Step 2: Loop through items

item = a
p contains: <a> - the List containing a.
item = b
p contains: <a>, <a.b>, <b> - 3 List
item = c
p contains: <a>, <a.b>, <b>, <ac>, <a.b,c>, <b,c>, <c> - 7 List
item = d
p contains: <a>, <a.b>, <b>, <ac>, <a.b,c>, <b,c>, <c>,
<a,d>, <a.b,d >, <b,d >, <ac,d >, <a.b,c,d >, <b,c,d >, <c,d >, <d> - 15 List