PROBLEM A: 0-1 KNAPSACK (REVISITED)

Camera
Weight: 1 kg
Value: 1000$

Laptop
Weight: 3 kg
Value: 2000$

Necklace
Weight: 4 kg
Value: 4000$

Knapsack
Capacity: 7 kg
Max value: ???

Vase
Weight: 5 kg
Value: 4500$

Supplemental materials:

- "knapsacktests.zip" is already provided before midterm.
- dfsKnapsack.py : an exhaustive depth-first-search program for 0/1 Knapsack
- KnapsackBound.py to be used as heuristic function for 0/1 Knapsack

Given a maximum weight that you can carry in a knapsack and items, each with a weight and a value, find a set of items you can carry in the knapsack so as to maximize the total value.

1. Consider the DP/memoized solultion, a state of this problem can be defined with two components.
   I. the current item being considered (implied that a known subset of items have already been considered, and the rest have not)
   II. the currently available capacity of the knapsack.

   Given that there are N items and max capacity is M, what is the size of the memoization or dynamic programming table ? _____

   The running time for filling in the table depends on both N and M. When M is relatively very large, the running time of the solution can be dominated by the capacity of the knapsack, even when there are only a few items.

2. Study the depth-first-search brute-force solution for 0-1 knapsack in Python (dfsKnapsack.py).
    - In this version, a state is defined by only the current item being considered (implying items with smaller indices have been considered).
    - Also pass the following arguments with each recursive call, in order to facilitate quick knapsack's property recalculation: the current weights of the knapsack, and the current sum of values in the knapsack
    - The algorithm updates a global variable "maxV" which is, at any time, the maximum total value in the knapsack that DFS has found so far. The update is required only when all items have been considered.

      Notice, however, that when all items have been considered, the total weight of some set of selected items exceeds the maximum capacity of the knapsack. And it is not valid to consider this set for updating maxV.

3. Add a part of the code so as to count the total number of recursions. Print the number of recursive calls as an output of the program. Try the program with the previous set of test cases.

4. Notice that even in a state where only some items have been considered, the total weight in the knapsack already exceed the maximum capacity. Is it worthwhile to DFS further from this state ?

5. Modify the program to prevent DFS call when the total weight in the knapsack exceeds maximum capacity. **This technique is called "pruning"**. Then, check whether the number of total recursive calls and the running time is reduced by pruning the DFS.

6. Given a state, think of a way to estimate the maximum value that can be produced out of performing DFS from this state? This estimate must be overly optimistic, that is – it is always better than the actual maximum value that can be obtained from starting at this state.

    Hint

    A better than the best realistic value is acceptable, for example, ∞ is a correct estimate, but is not useful. A way to get a better than the best realistic value is to assume that certain rule is relaxed (and thus, the certain condition may be relaxed).

    In this case, what if each item does not have to be taken for its entirety, for example, *what if each item is a kind of liquid*? Then we can take only some fraction of it, and the taken value is proportional to the fraction of the item's total value.

7. The “optimistic” estimate obtained from question 6, combined with the value of items that is already in the knapsack at state S, gives the maximum total value by going through state S.

   If this maximum total value is less than the current value of “maxV”, would it be worthwhile to DFS on state S ?

8. Study the provided KnapsackBound.py as **“bounding function”** that take state S and returns the estimate as determined in question 6. *What is the condition for this function to work correctly?*

9. Modify the program from question 5 so that the program incorporates the use of bounding function in determining whether to continue DFS from each state.

10. Then, check whether the number of total recursive calls and the running time is reduced by utilizing the bounding function to prevent unnecessary DFS.

11. As a further improvement over step 9, if an item can fit in the knapsack, there will be two options to DFS i.e. pick this item or skip this item. By computing “bound” for each option, how should the program decide which option should be searched first?

12. Apply the concept obtained from question 11 above to develop the **“Branch and Bound”** version of the solution.

In summary, Branch and Bound technique is, at a DFS state, to order the DFS options such the more prospective search is chosen before the less prospective one.