Assignment 2 Description:

In this assignment you will:

1) test the attached simple knapsack code with some simple examples and convince your self it works. Add comments to each conditional statement. Note how in this code the list of objects begins with **None**. This is because the variable "i" in the function refers to the number of objects, not the index into the array. That is why we can terminate on i==0 meaning no objects. Don't change this scheme.

2) Expand the knapsack function to solve a different and more complex problem. In this new problem:

There are two independent knapsacks, of size K1 and K2 (ints). An object may be thrown away, put in the first knapsack or put in the second knapsack.

The function is no longer a Bool in this simple code which is solving a decision problem. Rather the new function returns a float because it is solving an optimization problem. In this problem, each object has a size and a value. So the problem is described by two arrays of size N+1. S containing integers >= 1 that represent the sizes of the objects. And V containing positive floats that represent the values of the objects.

The function should return the maximum value that can be obtained by picking a subset of objects from S. The subset of objects must fit into the knapsacks. So the sum of the sizes of objects picked from knapsack 1 must be less than or equal to K1, likewise for knapsack 2. There is no need to exactly fill the knapsacks.  The sum of the values for those objects must be the maximum obtainable (over all possible subsets of objects).

Before you panic thinking about how to write this code. Take a breath and apply the meta-algorithm to the problem. If you do that, you should arrive at a fairly simple function that requires about 8 lines of code. Test the code with some very simple examples. Note, this code does not have to determine the actual objects that are best.

3) Write a simple problem generator that takes N and aveSize and returns two arrays S and V containing N+1 entries, where the size of the objects is random and in the range from 1 to 2\*aveSize (integer). The values can be any positive float.

4) Write the Memoizing function by copying the recursive code and modifying it

5) Write the DP code by copying the recursive code and modifying it.

6) Perform the following experiment: for 10 different aveSize values, determine the execution time of the DP and the memoizing function (with fixed K1 and K2 and N, make them big enough so the code run time can be measured). Vary aveSize from 1 to 2\*N. For each aveSize, measure the time over 20 independent runs. Plot a graph of aveSize (independent variable) vs. run time (dependent variable). The graph should have two lines on it, one for DP and one for memoizing.

Submit your code, your graph and a brief explanation of why you think the graph is the way it is.