Lab 6

Exercise 1: Consider the recursive `makeChange` implementation in the lecture notes, without memoization:

```
# if we can't make change for n cents using L, returns -2
def makeChange(n, L):
    if n==0:
        return 0
    # if possible to make change for n, the answer is definitely less
    # than n + 1
    answer = n + 1
    for x in L:
        if x <= n:
            val = makeChange(n-x, L)
            if val != -2:
                answer = min(answer, 1 + val)
    if answer == n + 1:
        answer = -2
    return answer
```

What's the running time of this function in terms of n when L is the list [1,2]?
Hint: Relate it to something you've seen already.

Exercise 2: Modify the memoized `makeChange` implementation so that it doesn't just tell you the minimum number of coins that are needed to make change, but instead it returns a list of *which* coins you should use to make change using the fewest number of coins.

Exercise 3: Write a function `lis`(L) which takes as input a list of integers L and outputs the length of the longest increasing subsequence (lis) of L. A subsequence of L is a sublist of L that does not have to be contiguous. For example, [1, 5, 9] is a subsequence of the list L = [1,2,3,4,5,6,7,8,9] since 1,5,9 appear in the list L in the same order (though just not in a row). 9,5,1 is not a subsequence of L since it does not appear in L in that order.

Your implementation should run in time $O(m^2)$ where the size of L is m. **Bonus (try at the end):** See if you can find a way to solve the problem in $O(m \log m)$ time.

Exercise 4: (taken from http://people.csail.mit.edu/bdean/6.046/dp/)

You are given a list of boxes, each having some height, width, and depth. You want to stack boxes to make the tallest stack possible, where you can only put one box on top of the other if its base has strictly smaller length and width than the box immediately underneath it. Given a list L of boxes of the form L = [[width1, height1, depth1], [width2, height2, depth2], ..., [widthm, heightm, depthm]], implement a function `boxes`(L) which returns the height of the tallest stack you can possibly make using the given boxes. You do not have to use all the boxes given to you, and you can use the same box multiple times.

Exercise 5: In the *knapsack* problem we assume we have a bag that can hold some n liters of volume, and we want to pack the bag with the most valuable combination of items possible out of some given list of items. Implement a function `knapsack`(n, L) which takes as input this bag volume n, and a list of items L, and returns the maximum value you can pack in the bag. L is a list of lists, where each list in L is of size 2 containing the volume of the item as its 0th element, and its volume as its 1st element. For example, L = [[7, 10], [5, 6], [4, 5]] represents a list of 3 items. The first item takes 7 liters and is worth 10 dollars, the second item takes 5 liters and is worth 6 dollars, and the last item takes 4 liters and is worth 5 dollars. `knapsack`(10, [[7, 10], [5, 6], [4, 5]]) should return 11 since the best thing to do is to take the and and third item (which both fit, since their total volume is $5 + 4 = 9$ liters, and we can fit 10 liters).

Your implementation should run in time $O(nm)$ where the size of L is m.