# 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:** You've already figured out the answer to this in a previous lab.

Answer: At least as much as $F_n$, the $n$th Fibonacci number. If you let $T(n)$ be the running time when trying to make change for $n$ cents, then due to the recursion we have $T(n) = T(n-1) + T(n-2) + 1$ (the "+1" to take the min of the two recursive calls). This is the same as the recurrence for computing the `fibonacci` function, except it has the +1, so it's going to be at least as big. In class we saw that $F_n \geq \sqrt{2}^n$.

**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.

**Example solution:** The most natural solution given what was in class is to change the return type in memMakeChange from an `int` to a `list`.

```
def memMakeChange(n, L, mem):
    if n==0:
        return []
    elif mem[n]!=-1:
        return mem[n]
    mem[n] = []
    for i in xrange(n):
```

```
        mem[n] += [1]
    for coin in L:
        if coin <= n:
            val = memMakeChange(n-coin, L, mem)
            if 1+len(val) < len(mem[n]):
                mem[n] = val + [coin]
    return mem[n]

# we assume L contains a 1-cent piece so that it's always possible to
# make change for n cents
def makeChange(n, L):
    mem = [-1]*(n+1)
    return memMakeChange(n, L, mem)
```

If the length of the list L is $m$, the above implementation takes time $O(n^2m)$. This is becomes in the for loop in `memMakeChange`, we are creating lists of potentially size $n$ (when creating the list `val + [coin]`, if, say, all elements in L are 1). It is possible to give an implementation taking time only $O(nm)$. The main idea is to have two `mem` lists. The first list, `mem`, is such that `mem`[n] is the smallest number of coins needed to make change for n cents. Then, `mem2`[n] is one possible coin that can be used in an optimal solution to make change for n cents.

```
def memMakeChange(n, L, mem, mem2):
    if n==0:
        return 0
    elif mem[n] != -1:
        return mem[n]
    mem[n] = n
    for coin in L:
        if coin <= n:
            val = memMakeChange(n-coin, L, mem, mem2)
            if val+1 < mem[n]:
                mem[n] = val+1
                mem2[n] = coin
    return mem[n]

# mem[n] is the minimum number of coins to make change for n
# mem2[n] is one coin you can take as part of the optimal solution for n
# we assume L contains a 1-cent piece so that it's always possible to
# make change for n cents
def makeChange(n, L):
    mem = [-1]*(n+1)
    mem2 = [-1]*(n+1)
    memMakeChange(n, L, mem, mem2)
    answer = []
    while n > 0:
        answer += [mem2[n]]
```

```
        n -= mem2[n]
    return answer
```

**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.

**Example solution:** Here is a solution running in time $\Theta(m^2)$. The basic idea is to let $f(\text{last}, \text{at})$ be the length of the longest increasing subsequence where the first number must be larger than L[last] and you're only allowed to use the numbers L[at],L[at+1],.... Then, $f$ can be computed recursively, and the answer is $f(i, i+1)$ for some $i$ (since the sequence has to start with *some* value L[i]), so the best choice over all $i$ should be returned. Memoization is used to make it faster.

```
def memlis(L, last, at, mem):
    if at==len(L):
        return 0
    elif mem[last][at]!=-1:
        return mem[last][at]
    mem[last][at] = memlis(L, last, at+1, mem)
    if L[at]>L[last]:
        mem[last][at] = max(mem[last][at], 1 + memlis(L, at, at+1, mem))
    return mem[last][at]

def lis(L):
    mem = []
    for i in xrange(len(L)):
        mem += [[-1]*len(L)]
    answer = 0
    for i in xrange(len(L)):
        answer = max(answer, 1 + memlis(L, i, i+1, mem))
    return answer
```

**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.

**Example solution:**

```
# L is given in decreasing order of base area, and memoize(L, at, last, mem)
# is the tallest stack we can make using only the boxes L[at],L[at+1],...
# where the last box we used was L[last]
def memoize(L, at, last, mem):
    if at==len(L):
        return 0
    elif mem[at][last] != -1:
        return mem[at][last]
    mem[at][last] = memoize(L, at+1, last, mem)
    if L[at][0]<L[last][0] and L[at][1]<L[last][1]:
        mem[at][last] = max(mem[at][last], L[at][2] + memoize(L, at+1, at, mem))
    return mem[at][last]

def boxes(L):
    T = []
    # put all 6 rotations of each box in a list
    for box in L:
        a1 = [box[0]*box[1], box[0], box[1], box[2]]
        a2 = [box[0]*box[1], box[1], box[0], box[2]]
        b1 = [box[0]*box[2], box[0], box[2], box[1]]
        b2 = [box[0]*box[2], box[2], box[0], box[1]]
        c1 = [box[1]*box[2], box[1], box[2], box[0]]
        c2 = [box[1]*box[2], box[2], box[1], box[0]]
        T += [a1, a2, b1, b2, c1, c2]
    # sort the list in decreasing order of base area
    T.sort()
    T.reverse()
    L = []
    for x in T:
        L += [[x[1], x[2], x[3]]]
    mem = []
    for i in xrange(len(L)):
        mem += [[-1]*len(L)]
    ans = 0
    # try all possibilities for the box at the very bottom and
    # take the best one
    for i in xrange(len(L)):
        ans = max(ans, L[i][2] + memoize(L, i+1, i, mem))
    return ans
```

**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 $m$.

**Example solution:** The basic idea is to let $f(\mathrm{n}, \mathrm{at})$ be the most value you can get in a bag of size $n$ using only the items L[at],L[at+1],.... $f$ can be computed recursively, and memoization makes it faster. Then the answer we finally want is $f(n, 0)$.

```
def memKnapsack(L, n, at, mem):
    if at==len(L):
        return 0
    elif mem[n][at]!=-1:
        return mem[n][at]
    mem[n][at] = memKnapsack(L, n, at+1, mem)
    if L[at][0]<=n:
        mem[n][at] = max(mem[n][at], L[at][1] + memKnapsack(L, n-L[at][0], at+1, mem))
    return mem[n][at]

def knapsack(n, L):
    mem = []
    for i in xrange(n+1):
        mem += [[-1]*len(L)]
    return memKnapsack(L, n, 0, mem)
```