Lecture 16

More practice problems!

Nested brackets: Consider the following lists: [], [[]], [[[]]], [[[[]]]], etc. The first is the empty list, the second is a list containing the empty list, the third is a list containing a list that contains the empty list, etc. We say that the first list in this sequence has nesting level zero, and the second has nesting level 1, etc. Write a function which takes in the nesting level n and outputs the appropriate list.

Example solutions: With recursion:

```
def nest(n):
    if n == 0:
        return []
    return [nest(n-1)]
```

Iteratively:

```
def nest(n):
    ans = []
    for i in xrange(n):
        ans = [ans]
    return ans
```

Multidimensional arrays: In many programming languages, there exists a data type called an *array* (this data type exists in Python as well, though we haven't discussed it). A one-dimensional array is similar to a Python `list`, though you must set its size in the beginning and later cannot shrink or increase its size. An n-dimensional array A is such that $A[i]$ is an $(n-1)$-dimensional array for each i within the bounds of the size of A. In other words, it is like a nested `list` of `list`s, n deep, where each list has equal size.

Often times in this course, especially when doing memoization, we have had to create a `list` of `list`s. For example, the `mem` variable in our memoization examples usually has been a `list` of n lists, each starting off to contain m -1's. We usually create such a `list` as:

```
mem = []
for i in xrange(n):
    mem += [[-1]*m]
```

Then we can access the jth entry of the ith list as `mem`[i][j]. What if we wanted to store our data not two-dimensionally, but n-dimensionally? For example, for $n = 3$, what if we wanted a `list` of n `list` of m `list`s, each of size r, so that we could then look at values `mem`[i][j][k]? Write a function `makeArray` that takes a list `L` of the dimensions of the matrix we want, together with a value `val`, and outputs nested `list`s, where the bottom-most `list`s in the nesting have all their entries set to `val`.

For example, `makeArray`([2, 3], -1) should return [[-1, -1, -1], [-1, -1, -1]] (a `list` of 2 `list`s, each of size 3, where all the starting values are -1).

Example solution:

```
def makeArray(L, val):
    if len(L) == 1:
        return [[val]*L[0]]
    ans = []
    for i in xrange(L[0]):
        ans += [makeArray(L[1:], val)]
    return ans
```

Min cost bitonic tour: We have a bunch of points in the plane: $[[x_0, y_0],\ldots,[x_{n-1}, y_{n-1}]]$. We would like to start at the 0th point, visit all other points exactly once, then return to the 0th point. We would like to do this while spending the least amount of time traveling as possible with one constraint: we have to do our travels west to east, then head back west again. That is, we can't zig-zag (we can't go east, then west, then back east again, then back west again). x_0 is the smallest amongst all the x_i, so the 0th point is the westernmost point. Also, x_{n-1} is the largest amongst the x_i, so it is the easternmost point. This is known as the minimum cost bitonic tour problem.

We can solve this problem using dynamic programming. The optimal path goes east, possibly skipping some points along the way, lands at x_{n-1}, then heads back west again, finally ending back at x_0. Let's instead think about our optimal path just going east twice, by thinking about the return voyage in reverse. So, the first eastward path goes $x_0 \to x_{i_1} \to \ldots \to x_{i_{r_1}} \to x_{n-1}$. The return voyage, in reverse, goes $x_0 \to x_{j_1} \to \ldots \to x_{j_{r_2}} \to x_{n-1}$. We should not have any repeats amongst the x_i and x_j, and also we should have $r_1 + r_2 = n - 2$ (we should visit all the points).

Example solution:

```
import math

def distance(x, y):
    dx = x[0] - y[0]
    dy = x[1] - y[1]
    return math.sqrt(dx*dx + dy*dy)

def recurse(L, at1, at2, mem):
    if at1+1==len(L) and at2+1==len(L):
        return 0
    elif mem[at1][at2] != -1:
        return mem[at1][at]
    ans = float('infinity')
    if at2 <= at1 + 1:
        for i in xrange(at2 + 1, len(L)):
            ans = min(ans, distance(L[at1], L[i]) + recurse(L, at2, i, mem))
    else:
        ans = recurse(L, at1 + 1, at2, mem)
    mem[at1][at2] = ans
    return ans
```

```
# We assume the points are already sorted from west to east, i.e. by
# their x coordinate. We also assume all x coordinates are unique.
def bitonicTour(L):
   mem = makeArray([len(A), len(A)], -1)
   return recurse(L, 0, 0, mem)
```

The running time is $\Theta(n^2)$. There are $\Theta(n^2)$ possibilities for a, b, and only for $O(n)$ of these possibilities do we actually have to spend $\Theta(n)$ time; the rest only require $\Theta(1)$ time. The basic idea of the solution is to build both paths simultaneously from left to right. Initially both parts start at the 0th point. Then at any given stage, suppose one path ends at L[at1] so far, and the other ends at L[at2], with at1 $\leq$ at2. Since we have to visit all points, if there are points between L[at1] and L[at2], then we have no choice: the one path has to visit L[at1+1] next. Otherwise, if there are no points in between (i.e. at2 and at1 differ by at most 1), then the path that ended at L[at1] can choose where to go next, to some point after L[at2], possiby skipping some number of points. We try all possibilities and take the best choice.