Lecture 11

Memoization: more worked examples.

Party Fun:

(Problem source: Swiss Olympiad in Informatics 2004, see `http://www.spoj.pl/problems/PARTY/`). In this problem there are n parties. The ith party has some entrance fee c_i and gives you some amount of fun f_i. Given that you only have D dollars, how can you choose which parties to attend so as to maximize your fun? This can be solved recursively.

```
# return the maximum fun that can be had from parties L[at:] given
# that we only have D dollars
def recurse(L, D, at):
    if at == len(L):
        return 0
    # don't attend party L[at]
    ans = recurse(L, D, at + 1)
    if D >= L[at][0]:
        # if we have enough money, try attending party L[at]
        ans = max(ans, L[at][1] + recurse(L, D - L[at][0], at + 1))
    return ans

# L is a list [[c_0,f_0], [c_1,f_1], ..., [c_{n-1}, f_{n-1}]]
def maximizeFun(L, D):
    return recurse(L, D, 0)
```

The running time of the above implementation is $\Theta(2^n)$. This can be seen by drawing the recursion tree: if we have enough dollars D to start off with, then at each level we branch in two ways (either attend the party or don't).

We can obtain running time $\Theta(nD)$ by using memoization.

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

# we assume all fun amounts are positive; we wouldn't bother attending
# a party that doesn't have positive fun
```

```
def maximizeFun(L, D):
    mem = []
    for i in xrange(len(L)):
        mem += [[-1]*(D+1)]
    return recurse(L, D, 0, mem)
```

Parenthesizing expressions:

(Problem source: Ulm Algorithm Course SoSe 2005, see `http://www.spoj.pl/problems/LISA/`). You are given an arithmetic expression with digits separated by '*' and '+'. How can you parenthesize the expression so as to maximize its value? For example, with the expression

$$1 + 2 * 3 + 4 * 5,$$

the best way of parenthesizing it is $(1 + 2) * (3 + 4) * 5$, giving 105. For example, parenthesizing it as $1 + (2 * 3) + (4 * 5)$ would only give 27.

Suppose the length of the expression is n. We will show how to solve the task in time $\Theta(n^3)$ using recursion and memoization. First, a recursive solution:

```
def maxValue(expr):
    if len(expr) == 1:
        return int(expr)
    else:
        ans = 0
        for op in xrange(1, len(expr), 2):
            # this loops over indices 1, 3, 5, ...
            # the "2" tells it to step by 2 each time
            if expr[op] == '+':
                ans = max(ans, maxValue(expr[:op]) + maxValue(expr[op+1:]))
            else:
                ans = max(ans, maxValue(expr[:op]) * maxValue(expr[op+1:]))
        return ans
```

Now let's speed it up with memoization.

```
# return the max value that can be obtained by parenthesizing the
# expression expr[a:b+1]
def f(expr, a, b, mem):
    if a == b:
        return int(expr[a])
    elif mem[a][b] != -1:
        return mem[a][b]
    else:
        ans = 0
        for op in xrange(a + 1, b + 1, 2):
            if expr[op] == '+':
                ans = max(ans, f(expr(a, op-1, mem)) + f(expr, op+1, b, mem))
            else:
                ans = max(ans, f(expr(a, op-1, mem)) * f(expr, op+1, b, mem))
        mem[a][b] = ans
        return ans

def maxValue(expr):
    mem = []
    for i in xrange(len(expr)):
        mem += [[-1]*len(expr)]
    return f(expr, 0, len(expr)-1, mem)
```

There are n possibilities for each of a, b, and for each possibility we do a while look taking $O(n)$ time. The running time is thus $O(n^3)$ (and in fact it is $\Theta(n^3)$).

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.