Addis Ababa University, Amist Kilo July 25, 2011

Algorithms and Programming for High Schoolers

Lecture 15

Review:

Today we'll do more recursion/memoization review by doing more example problems.

Egg Dropping: Suppose we live in a building with H floors, and we are trying to figure out the highest floor from which we can drop an egg such that the egg won't break. We make a few assumptions:

- All eggs behave the same way. That is, if one egg would or wouldn't drop at any given floor, then neither would any other.
- If an egg would break being dropped from some floor, then it would also break being dropped from a higher floor.
- If an egg wouldn't break being dropped from some floor, then it also wouldn't break being dropped from a lower floor.
- If an egg breaks from a fall, we can't reuse it. If it doesn't break, we can reuse it.

Now, suppose we have N eggs and would like to know: what is the minimum number of egg drop tests we have to do to find the highest floor from which it is safe to drop eggs?

Well, if we have 0 eggs and $H > 0$, the task is impossible; we'll represent this by the answer ∞. If $H = 0$, then the answer is 0 tests. If $N = 1$, then we have no other option than to start at the lowest floor and try floor by floor going upwards, thus requiring H tests. Otherwise, we can choose a floor x to drop the egg from. Let $f(N, H)$ be the worst-case minimum number of tests required when we have N eggs and H consecutive floors left to test. Then, if our first egg drop is at floor x, either it will break and we have to do $f(N-1, x-1)$ more tests, or it won't and we'll have to do $f(N, H-x)$ more tests. Thus, in the worst case, we will have to do $\max\{f(N-1, H-1), f(N, H-x)\}$ extra tests. We can choose x to minimize this expression. Thus, in total we have:

$$f(n,h) = \begin{cases} 0 & \text{if } h = 0 \\ \infty & \text{if } n = 0 \text{ and } h > 0 \\ h & \text{if } n = 1 \\ 1 + \min_{x=1,\ldots,h}\{\max\{f(n-1, x-1), f(n, h-x)\}\} & \text{otherwise} \end{cases}.$$

We can implement this easily using recursion:

```
def f(n, h):
    if h == 0:
        return 0
    elif n == 0:
        return float('infinity')
    elif n == 1:
        return h
    else:
        ans = float('infinity')
        for x in xrange(1, h+1):
            ans = min(ans, 1 + max(f(n-1, x-1), f(n, h-x)))
        return ans
```

As usual, the above can be sped up using memoization.

```
def recurse(n, h, mem):
    if h == 0:
        return 0
    elif n == 0:
        return float('infinity')
    elif n == 1:
        return h
    elif mem[n][h] != -1:
        return mem[n][h]
    else:
        ans = float('infinity')
        for x in xrange(1, h+1):
            ans = min(ans, 1 + max(f(n-1, x-1), f(n, h-x)))
        mem[n][h] = ans
        return ans

def f(n, h):
    mem = []
    for i in xrange(n+1):
        mem += [[-1]*(h+1)]
    return recurse(n, h, mem)
```

The running time of the memoized version is $O(nh^2)$ (nh possibilities for the input, and $O(h)$ work in the `for` loop for each input if not already in memory). In fact it's even $\Theta(nh^2)$.

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 $\Theta(n^3)$).

Rainbow ride:

(Problem source: Kurukshetra 09 OPC, see `http://www.spoj.pl/problems/RAINBOW/`). A large family of n people is going on vacation, and they are deciding who should go on a certain roller coaster ride. Each person in the family likes some other people in the family, and a person will only go on the ride if everyone he likes and everyone who likes him also goes on the ride. Each person has a weight, and the roller coaster can only support a total weight of W. Note: if two people don't like each other, they're still allowed to go on the roller coaster together. How do we choose who to go on the ride so that the maximum number of people participate?

How do we solve this problem? First we build a graph where family members are vertices, and there is an edge (u, v) between two family members if either u likes v or v likes u. Then if one family member goes on the ride, everyone from his connected component must go on the ride with him. So, essentially we must choose which connected components go on the ride so that the total weight of all connected components we choose is at most W. We can first find the connected components and their total sizes and weights by either DFS or BFS. Then this becomes exactly the knapsack problem from Exercise 5, Lab 6. We have some number m of items (each item is a connected component of the family graph), with item i having value v_i (the number of people in that component) and weight w_i (the total weight of the component). We must choose which items to put in our knapsack of weight W (i.e. the roller coaster ride) so as to maximize the total value. This can be solved in time $\Theta(mW)$ using memoization; see the Lab 6 solutions for details.