Lecture 10

Memoization: worked examples.

Longest Common Subsequence: Given a `str` $s = s[0]s[1]\dots s[n-1]$, a *subsequence* of s is a `str` $s[i_1]s[i_2]\dots s[i_r]$ with $0 \le i_1 < i_2 \dots < i_r \le n-1$. That is, you form a subsequence by going from left to right along the `str` s and taking some subset of letters (including possibly no letters, in which case you'd get the empty `str`).

Now, the algorithmic problem we will consider is the following: computing the length of the longest common subsequence (LCS) of two strings s and t. The idea is to use recursion, which we can speed up using memoization. When trying to form a LCS for s,t, let us consider the first letters of both s and t. We have three options. Either both the first letter of s and the first letter of t are in the LCS (in which case these letters must be equal), or one of them *isn't* in the LCS. This is of course assuming s and t both have a first letter; if one of them is the empty string, then their LCS is the empty string, which has length 0. Thus we have the following:

$$\mathrm{LCS}(s,t) = \begin{cases} 0 & \text{if } \mathrm{len}(s) = 0 \text{ or } \mathrm{len}(t) = 0 \\ \max\{1 + \mathrm{LCS}(s[1:], t[1:]), \mathrm{LCS}(s[1:], t), \mathrm{LCS}(s, t[1:])\} & \text{if } s[0] == t[0] \\ \max\{\mathrm{LCS}(s[1:], t), \mathrm{LCS}(s, t[1:])\} & \text{otherwise} \end{cases}.$$

(In fact it's not too hard to show that if $s[0] == t[0]$ then the best option is always to take both $s[0]$ and $t[0]$ in the LCS, and thus the the middle case can just have one option instead of three.)

The above can then be implemented in code recursively:

```
def lcs(s, t):
    if len(s)==0 or len(t)==0:
        return 0
    ans = lcs(s[1:], t)
    ans = max(ans, lcs(s, t[1:])
    if s[0] == t[0]:
        ans = max(ans, 1 + lcs(s[1:], t[1:]))
    return ans
```

The above code though is quite slow and can be sped up via memoization.

```
# compute the LCS between s[atS:] and t[atT:]
def recurse(s, t, atS, atT, mem):
    if atS==len(s) or atT==len(t):
        return 0
    elif mem[atS][atT] != -1:
        return mem[atS][atT]
    ans = lcs(s, t, atS + 1, atT, mem)
    ans = max(ans, lcs(s, t, atS, atT + 1, mem)
```

```
    if s[atS] == t[atT]:
        ans = max(ans, 1 + lcs(s, t, atS + 1, atT + 1, mem))
    mem[atS][atT] = ans
    return ans

def lcs(s, t):
    mem = []
    for i in xrange(len(s)):
        mem += [[-1]*len(t)]
    return recurse(s, t, 0, 0, mem)
```

The running time of the memoized version is $\Theta(\text{len}(s) \cdot \text{len}(t))$.

Edit Distance: Have you ever misspelled a word when searching on Google, and Google asks you "Did you mean to search for ...?", with the correct spelling given? Or have you ever used a spellchecker in Microsoft Word or some other word processor which gives you suggestions for how to correct your misspellings? For these and other applications, it is useful to have an algorithm for computing the *edit distance* between two strings. The edit distance between strings s and t is the minimum number of "changes" that one can make to s to turn it into t. The allowed changes are one of the following: (1) insert any character at any position in s, (2) delete any character from s, and (3) change any character in s to a different character. For example, consider the misspelling "fantom" of "phantom". Three changes can be made to fantom to turn it into phanton: delete the "f", then insert the "p" and "h". However, a cheaper way is to replace the "f" with an "h" then insert the "p" in the beginning, and in fact this is the cheapest way, so the edit distance of those two strings is 2.

How does one compute the edit distance between two strings s and t? Again this can be done using recursion and memoization. If s is the empty string, we must insert $\text{len}(t)$ letters into s to change it into t. If t is the empty string, we must delete $\text{len}(s)$ letters to turn s into the empty string. Otherwise, we have a few options. If $s[0] == t[0]$, we have the option of just trying to turn $s[1:]$ into $t[1:]$. Otherwise, we can insert $t[0]$ right before s then try to turn s into $t[1:]$. We can also delete $s[0]$ then try to turn $s[1:]$ into t. Finally, we can also change $s[0]$ into $t[0]$ then try to change $s[1:]$ into $t[1:]$. Let ED represent edit distance. Then,

$$\text{ED}(s,t) = \begin{cases} \text{len}(t) & \text{if } \text{len}(s) = 0 \\ \text{len}(s) & \text{if } \text{len}(t) = 0 \\ \min\{\text{ED}(s[1:],t[1:]), 1+\text{ED}(s[1:],t), 1+\text{ED}(s,t[1:])\} & \text{if } s[0] == t[0] \\ \max\{1+\text{ED}(s[1:],t[1:]), 1+\text{ED}(s[1:],t), 1+\text{ED}(s,t[1:])\} & \text{otherwise} \end{cases}.$$

In code:

```
def ED(s, t):
    if len(s)==0:
        return len(t)
    elif len(t)==0:
```

```
        return len(s)
    elif s[0] == t[0]:
        return min(ED(s[1:],t[1:]), 1 + min(ED(s[1:], t), ED(s, t[1:])))
    else:
        return 1 + min(ED(s[1:],t[1:]), min(ED(s[1:], t), ED(s, t[1:])))
```

The above implementation can be sped up using memoization, as follows.

```
# return the edit distance between s[atS:] and t[atT:]
def recurse(s, t, atS, atT, mem):
    if atS == len(s):
        return len(t)
    elif atT == len(t):
        return len(s)
    ans = 1 + min(recurse(s, t, atS + 1, atT, mem), recurse(s, t, atS, atT + 1, mem))
    if s[0] == t[0]:
        ans = min(ans, recurse(s, t, atS + 1, atT + 1, mem))
    else:
        ans = min(ans, 1 + recurse(s, t, atS + 1, atT + 1, mem))
    mem[atS][atT] = ans
    return ans

def ED(s, t):
    mem = []
    for i in xrange(len(s)):
        mem += [[-1]*len(t)]
    return recurse(s, t, 0, 0, mem)
```

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 $\Theta(nh^2)$.