Addis Ababa University, Amist Kilo July 22, 2011

Algorithms and Programming for High Schoolers

Lecture 14

Review:

Today we'll review previous material by doing a few example problems.

Longest Common Subsequence: Given a `str` $s = s[0]s[1]\ldots s[n-1]$, a *subsequence* of s is a `str` $s[i_1]s[i_2]\ldots s[i_r]$ with $0 \le i_1 < i_2 \ldots < i_r \le n-1$. 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)
```

Ternary Search: The last example we're covering today is ternary search. Binary search is useful for the following problem: we are given a sorted list L[0] $\leq$ L[1] $\leq \ldots \leq$ L[n-1], and we would like to find out which i has L[i] == x for some input x. One way is with a `for` loop:

```
# L is sorted
def findX(L, x):
    for i in xrange(len(L)):
        if L[i] == x:
            return i
    return -1
```

However, in the worst case the above code takes $\Theta(n)$ time. It is much better to find x by binary search as in Lecture 3, where we check the middle element to see whether it is too small or too big, then recursively check the half where x might possibly lie.

What if L is not increasing, but rather is decreasing until some unknown point, then increasing? To find an x in such a list we could first find the position j where L switches from being decreasing to increasing, then binary search in L[:j] and L[j:] separately to try to find x. So, how do we find this position j where L switches behavior? To do this, we could use a an algorithm known as *ternary search*.

The idea behind ternary search is as follows. First, we have to assume that L never has the same value twice at two different positions for this algorithm to work. Now, for a `list` L of length n set `posA` $= n/3$ and `posB` $= 2n/3$. We look at L[posA] and L[posB]. There are two cases: either L[posA] ¡ L[posB], or the other way around. In the first case, it can be that both posA and posB are to the right of the switching point j, or posA is to the left of it and posB is to the right of it. However, if L[posA] ¡ L[posB], it *cannot* be the case that both are to the left of the switching point. Thus, we can eliminate L[posB:] from consideration. Similarly in the case L[posA] ¿ L[posB], it cannot be the case that both posA and posB are to the right of the switching point, so we can eliminate L[:posA+1]. In either case we eliminate 1/3rd of the possible entries and are thus left with only $2n/3$ possibilities. The running time is thus the smallest k such that $(2/3)^k \cdot n \leq 1$, so it is $\Theta(\log_{3/2} n) = \Theta(\log_2 n)$ (recall that $\log_a n = (1/\log_b a) \cdot \log_b n$).

```
# find the switching point from decreasing to increasing
# in the list L[from:to+1]
def recurse(L, from, to):
    if from == to:
        return from
    # n items remaining
    n = to - from + 1
    posA = from + n/3
    posB = from + 2*n/3
    if L[posA] < L[posB]:
        return recurse(L, from, posB - 1)
    else:
        return recurse(L, posA + 1, to)

# find the switching point from decreasing to increasing
# in the list L
def ternarySearch(L, x):
    return recurse(L, 0, len(L) - 1)
```