CS109A Notes for Lecture 3/11/96

Substrings

In ML notation, x is a *substring* of y if $y = u\hat{}x\hat{}v$ for some strings u and v.

- Similar notion for lists, i.e., $y = u@x@v$.

Example: The substrings of aba are ϵ (the empty string), a, b, ab, ba, and aba.

- Note that a substring need not be *proper* (i.e., less than the whole string).
- Special case: *prefix* of y is any substring x that begins at the beginning of y.

Example: Prefixes of aba are ϵ, a, ab, and aba.

- Special case: *suffix* of y is a substring that ends at the end of y.

Example: Suffixes of aba are ϵ, a, ba, and aba.

Subsequences

A *subsequence* of a string y is what we can obtain by striking out 0 or more of the positions of y.

Example: Subsequences of aba are ϵ, a, b, ab, ba, aa, aba.

- A *common subsequence* of x and y is a string that is a subsequence of both.
- A *longest common subsequence* (LCS) of x and y is a common subsequence of x and y that is as long as any common subsequence of these strings.

Why LCS's?

- Secret of the UNIX `diff` command (find the differences between two files).
  - □ `diff` finds a LCS of the two files and assumes the changes are "everything else."
- Generalizations important in matching of DNA sequences.

An Exponential LCS Algorithm

The following assumes two lists (not strings) and computes their LCS:

```
fun lcs(_,nil) = nil
|   lcs(nil,_) = nil
|   lcs(x::xs, y::ys) =
    if x=y then x::lcs(xs,ys)
    else let
        val l1 = lcs(xs, y::ys);
        val l2 = lcs(x::xs, ys);
    in
        if length(l1) > length(l2)
            then l1
        else l2
    end;
```

- Problem: If size n = sum of the lengths of the lists, then there are two recursive calls to `lcs` on arguments of one smaller size.
  - ☐ Leads to recurrence relation $T(n) = O(n) + 2T(n-1)$, with solution $O(2^n)$.

Dynamic Programming Solution

Recursions like this waste time because they wind up solving the same problem repeatedly.

Example: If $x = [1,2,3,4]$ and $y = [a,b,c,d]$, we call `lcs` twice on $([2,3,4],\ [b,c,d])$, four times on $([3,4],\ [c,d])$, and so on.

- *Dynamic programming* solutions tabulate the answers to subproblems, so they are available for use many times.

Example: The most common example is computing $\binom{n}{m}$ by the recursion $\binom{n}{m} = \binom{n-1}{m-1} + \binom{n-1}{m}$ vs. computing it by Pascal's triangle (see p. 172, FCS).

- For LCS, build an array L such that $L[i][j]$ is the length of the LCS for the first i positions of x and the first j positions of y.
  - ☐ Given this array, filled in, one can easily recover an LCS — see p. 324 ff, FCS.

- Fill in order of $i + j$.

Basis: $i + j = 0$. Surely $L[0][0] = 0$.

Induction:

- If either i or j is 0, then $L[i][j] = 0$.
- If neither is 0, consider a_i and b_j, the ith and jth elements of strings x and y, respectively.
  - If $a_i = b_j$, $L[i][j] = 1 + L[i-1][j-1]$.
  - Otherwise, $L[i][j]$ is the larger of $L[i][j-1]$ and $L[i-1][j]$.
- Either way, the L entries needed have already been computed.

Running Time of LCS

If n = sum of lengths of strings, time is $O(n^2)$.

- Fill $(n+1)^2$ entries, each in $O(1)$ time.