Solutions to Math 416 Homework due November 15, Chapters 15–16

November 15, 2004

1 Problem 15.4-5

Given a sequence X of length n, let Y be a sorted sequence of the elements in X. Let Z be an LCS of X and Y, which can be found in time $O(n^2)$ by dynamic programming. Then Z is a longest monotonically increasing subsequence of X.

To see this, assume the entries of X are unique (the general case is similar). It is straightforward to check that any common subsequence of X and Y is a monotonically increasing subsequence of X and, conversely, any monotonic subsequence corresponds to an LCS. (If X has duplicates, it helps to use a *stable* sort, in which two records with equal keys retain their relative order in the sorted output.)

The following is only slightly different from unwinding the above reduction. Define a table with cells $c[i, j]$, for $0 \le i, j \le n$. The cell $c[i, j]$ stores the longest monotonically increasing subsequence of the prefix $X_i = \langle x_1, x_2, \ldots, x_i \rangle$ such that each element of the sequence is at most x_j. Fill in the appropriate boundaries of the table and fill in the innards of the table as follows:

```
for( $i \leftarrow 0$; $i \le n$; $i++$)
    for( $j \leftarrow 0$; $j \le n$; $j++$)
        if( $x_i > x_j$)
            $c[i, j] \leftarrow c[i-1, j]$
        else
            $c[i, j] \leftarrow \max(c[i-1, j], c[i-1, i] + 1)$
```

The idea is if $x_i > x_j$, then x_i cannot be used in a sequence where every item is at most x_j. So we use the longest sequence up to $i - 1$. Otherwise, x_i is allowed, but it might still be optimal to omit it. If we don't use x_i, then, as before, we want a sequence on X_{i-1} bounded by x_j. If we do use x_i, then we want a sequence on X_{i-1} bounded by x_i, which we extend by a single element (x_i).

One can arrive at this by trying to form a monotonic subsequence of X_i from subsequences Z of X_{i-1}, and realizing that there's a condition on Z and x_i needed to make this work, namely, all entries in Z must be at most x_i. The setup of dynamic programming means that we'll need to attend to this condition while considering Z, before seeing x_i. One way to do this is by anticipating every possible x_i as a bound for the sequence Z.

2 Problem 15-3

Part a.

To transform x into y, we can either transform a prefix of x into y without the kill operation and then use a final kill, or we can transform all of x into y with no final kill. So we'll compute the cost of transforming each prefix of x into y. To do that, we'll also compute, for each i and j, the cost of transforming $x[1..i]$ to $y[1..j]$.

To transform $x[1..i]$ to $y[1..j]$, we consider each of the five operations Copy, Replace, Delete, Insert, and Twiddle as a possible last operation, and compute the cost conditioned on that. Let $c'[i,j,t]$ denote this conditional cost, conditioned on the last operation being t. Let $c[i,j]$ be the unconditional cost. We have:

- If Copy is the last operation, then $c'[i,j,\text{Copy}] = c[i-1,j-1] + \text{cost}[\text{Copy}]$. Only consider this if $x_i = y_j$; otherwise, $c'[i,j,\text{Copy}] = +\infty$.
- If Replace is the last operation, then $c'[i,j,\text{Replace}] = c[i-1,j-1] + \text{cost}[\text{Replace}]$.
- If Delete is the last operation, then $c'[i,j,\text{Delete}] = c[i-1,j] + \text{cost}[\text{Delete}]$.
- If Insert is the last operation, then $c'[i,j,\text{Insert}] = c[i,j-1] + \text{cost}[\text{Insert}]$.
- If Twiddle is the last operation, then $c'[i,j,\text{Twiddle}] = c[i-2,j-2] + \text{cost}[\text{Twiddle}]$. Only consider this if $x_{i-2} = y_{j-1}$ and $x_{i-1} = y_{j-2}$; otherwise, $c'[i,j,\text{Twiddle}] = +\infty$.

Finally, let $c[i,j] = \min_t c'[i,j,t]$.

We can store along with $c[i,j]$ one of the values of t (*i.e.*, the choice of final operation) that leads to the minimum cost.

Note that, to compute $c[i,j]$, we only require $c[k,\ell]$ for $k \le i, \ell \le j$, and either $k < i$ or $\ell < j$. It follows that we can fill in the table one diagonal at time, in order of increasing $i + j$.

By induction and the definition of the operations, it follows that $c[i,j]$ properly computes a minimal cost, as desired. Once the table is filled in, we can consider $\min(c[m,n], \min_{i<m} c[i,n] + \text{cost}[\text{Kill}])$; that is the minimal cost. We can print out a sequence of transformations by tracing backwards over the stored choices (including the choices involving the final Kill).

The time for this algorithm is $\Theta(mn)$, since there are mn cells and each cell takes constant time to fill in, and the final minimization over kills is quick. The space cost of this algorithm is also $\Theta(mn)$.

[Note: the instructions say "analyze the ... requirements," but it does *not* say "find the most efficient algorithm." Any polynomial time algorithm is ok here. There are other possible space/time tradeoffs, depending on whether we want just the cost or also want to print the sequence, and how fast we want to print the sequence.]

Part b.

Use the following costs for edit distance:

Operation	Cost
Copy	-1
Replace	$+1$
Delete	$+2$
Insert	$+2$
Twiddle	$+\infty$
Kill	$+\infty$

We will show that, given two sequences, an optimal alignment (*i.e.*, an alignment with maximal score), corresponds to a sequence of edits with minimal cost. To do this, we need to show that the set of optimal alignments corresponds to the set of optimal edit sequences.

Given any alignment, we read off an edit sequence from left to right. Corresponding symbols gives a Copy, differing symbols gives a Replace, space-nonspace gives an Insert and nonspace-space gives a Delete. The alignment score equals the negative of the edit cost, position by position. The converse is similar.

It follows that the maximum alignment score is the maximum of the negative of the edit cost; *i.e.*, the negative of the minimum edit cost, which is the negative of the edit distance.

[Note: My hint incorrectly said that the alignment score should be the edit distance, whereas these should be negatives of each other.]

3 Problem 16.3-8

The task is as follows:

```
We are trying to compress uniformly random files of exactly n
characters, where n is known and there are 256 = 2^8 different
characters.  We will compress the file x by the bit string f(x).  We
require that, if x and y are different, then f(x) and f(y) are
different.  It is not necessary that the range of f be prefix.

Show that, for any such f,  E[|f(x)|] > 8n - 2.
```

The idea is that any such encoding f is a map into the complete binary tree. In an optimal encoding, the tree fills up layer by layer, starting with the root; that is, if $|f(x)| > c$ for some x and some c and there is some target string t of length c that is unused, it would be at least as good to make $f(x) = t$ instead. By hypothesis, there are 2^{8n} nodes used in the tree, which means all nodes at depth less than $8n$ and one node at depth $8n$. The nodes have equal frequency, so the expectation is

$$2^{-8n}\left(\sum_{0\le j<8n} j2^j + 8n\right),$$

where j is the depth in the tree (the length $|f(x)|$ of the codeword $f(x)$), 2^{-8n} is the uniform probability of selecting a particular x, and 2^j is the number of x with $|f(x)| = j$. The extra $8n$ corresponds to the lonely codeword at depth $8n$.

It's convenient to think about the compression savings compared with $8n$. The one lonely x gets no savings. For each of 2^{8n-1} of the x's (*i.e.*, half of the x's) the savings is 1 bit. For 1/4 of the x's, the savings is two bits, etc. Thus the total savings is $\sum_{1\le k\le 8n} k2^{-k}$. The infinite sum $\sum_{0\le k} k2^{-k}$ is 2, so the partial sum is less than 2, as desired.

To see the infinite sum, let $S = \sum_{0\le k} k2^{-k} = \sum_{1\le k} k2^{-k}$. Then $2S = \sum_{1\le k} k2^{1-k} = \sum_{0\le k}(k+1)2^{-k}$, so that $S = 2S - S = \sum_{0\le k} 2^{-k}$. Repeating this argument, the sum of this geometric series is 2.