Making a Difference

The UNIX utility `diff` compares two files, and outputs a list of changes necessary for transforming the first into the second. For this question we will look at a possible way of implementing a difference program; one that works with files containing one word per line.

There are three types of transformation: appending, deleting and changing. An output of the form “n1 a n2,n3” means we append lines n2 to n3 of the second file, after line n1 of the first file. The output “n1,n2 c n3,n4” means change lines n1 to n2 of the first file to lines n3 to n4 of the second. Finally, “n1,n2 d n3” tells us to delete lines n1 to n2 in the first file. Each of these instructions are followed by the lines that are affected in each file; a ‘<’ indicating a line from the first file, and a ‘>’ indicating a line from the second.

For example, if we compare the following two files:

```
she
sells
sea
shells
on
the
sea
shore

the
sea
shells
are
sea
shore
```

A possible response would be:

```
0a1,3
> the
> sea
> shells
2a6,6
> are
5,8c9,10
< on
< the
< sea
< shore
---
> I’m
> sure
```

**Question 1.1**

Calculate valid difference outputs for the following pairs of files (if written with one word on each line):

(a) it was the best of times it was the worst of times
   it was the age of wisdom it was the age of foolishness

(b) permuting this sentence may render it meaningless
    may it sentence permuting render this meaningless
Question 1.2
There may be many ways of transforming one file into another. The method we choose might depend on several criterion; for example we may wish to minimize the number of lines that are transformed, or try to keep large blocks of lines intact. Suppose we are transforming an $n$ line file into a (different) $m$ line one. What is the minimum number of transformations we can use? What is the maximum number?

Question 1.3
Suppose we have run our program and calculated how to transform file1 into file2. How can we use this information to generate a transformation from file2 to file1, and hence why does the deletion transformation have the format “n1,n2 d n3”?

2. Longest common subsequence

A subsequence of a sequence $X$ is just $X$ with some of the elements, possibly none, removed. Suppose we have two sequences $X$ and $Y$: $Z$ is a common subsequence of $X$ and $Y$ if it is a subsequence of both of them. If there does not exist a common subsequence with more elements in it, we say $Z$ is a longest common subsequence (LCS). [Note that there may be more than one] Given $X$ and $Y$ we may be interested in determining a longest common subsequence.

Question 2.1
(b) Give a pair a sequences which have more than one LCS, along with example LCSs.

Question 2.2
A naive algorithm for calculating an LCS would be to calculate all the subsequences of $X$, and check to see which are also subsequences of $Y$. Why would this impractical for all but the smallest sequences?

We can recursively calculate an LCS of $X$ and $Y$, by considering the LCSs of pairs of prefixes of $X$ and $Y$. Let $X_n$ be the sequence $<x_1, x_2, \ldots , x_n>$ and $Y_m$ the sequence $<y_1, y_2, \ldots , y_m>$. Furthermore let $Z_k = <z_1, \ldots , z_k>$ be any LCS of $X_n$ and $Y_m$.

- If $x_n = y_m$, then $z_k = x_n = y_m$ and $Z_{k-1}$ is an LCS of $X_{n-1}$ and $Y_{m-1}$.
- If $x_n \neq y_m$, then if $z_k \neq x_n$, $Z_k$ must be an LCS of $X_{n-1}$ and $Y_m$.
- If $x_n \neq y_m$, then if $z_k \neq y_m$, $Z_k$ must be an LCS of $X_n$ and $Y_{m-1}$.

Question 2.3
If $x_n \neq y_m$, how does the length of the LCS of $X_n$ and $Y_m$ relate to the LCS of $X_{n-1}$ and $Y_m$ and the LCS of $X_n$ and $Y_{m-1}$?

Question 2.4
Suppose the $X$ and $Y$ are stored in the arrays $x$ and $y$. Using the above relationships write a recursive procedure $\text{LCS\_Length}$ which is called with the parameters $n$ and $m$, and returns the length of the LCS of $X_n$ and $Y_m$. [Indicate whether your arrays begin at 0 or 1.]

The recursive procedure needs to call itself since we base our solution of the problem, on the results of smaller problems. The flaw with this approach is that $\text{LCS\_Length}$ may get called many times for the same values of $n$ and $m$. For example
LCS\(_{\text{Length}(5,5)}\) may need to look at \(\text{LCS\_Length}(4,5)\) and \(\text{LCS\_Length}(5,4)\), both of which may need to look at \(\text{LCS\_Length}(4,4)\).

One way to prevent repeated calculation of the same problem would be to store the results of the sub-problems we have solved. When we are asked to solve a problem for the second time, all we need to do is recall this stored value. The dynamic programming approach is to calculate the sub-problems in a different order, so that whenever we need the result of another problem we have already solved it.

The outline of a dynamic programming procedure, might look as follows:

\[
\text{for } i := 0 \text{ to } n \text{ do} \\
\quad \text{for } j := 0 \text{ to } m \text{ do} \\
\quad \quad \text{lcsl}[i,j] := \text{"length of LCS of } X_i \text{ and } Y_j\" \\
\]

This procedure builds up a table lcsl, where lcsl\([i,j]\) contains the length of the LCS of \(X_i\) and \(X_j\). Observe that by the time we calculate lcsl\([i,j]\) all the smaller sub-problems have already been solved.

**Question 2.5**

Suppose we have calculated the table lcsl, and hence know the length of the LCS. How can we use this information to find an actual LCS?

3. **Creating a difference output**

We will now return to the problem of finding a transformation between two files. If we treat the individual words in the two files as the elements of our sequences, we can use our LCS algorithm to find sequences of words in common between the two files. The output from our algorithm might look something like

\[
(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)
\]

where the \(i_n^{th}\) word in the first file is the same as the \(j_n^{th}\) word in the second, and both \(i_1, \ldots, i_k\) and \(j_1, \ldots, j_k\) are strictly increasing sequences. For example, if we look back at original example, the output would be

\[
(1, 4), (2, 5), (3, 7), (4, 8)
\]

Of course, we are really interested in the parts of the files that are not common. ie.

- words 1 to \(i_1 - 1\) in the first file, compared with 1 to \(j_1 - 1\) in the second.
- words \(i_1 + 1\) to \(i_2 - 1\) in the first file, compared with \(j_1 + 1\) to \(j_2 - 1\) in the second.
  
  : 

- word \(i_k + 1\) to the end of the first file, compared with \(j_k + 1\) to the end of the second file.

[NB “words \(x\) to \(y\)” where \(x > y\), signifies an empty sequence.]

**Question 3**

Suppose you have (from above) that “words \(n1\) to \(n2\) in the first file have nothing in common with words \(n3\) to \(n4\) in the second file.” When, and how, does this convert to

- (a) an ‘append’ transformation?
- (b) a ‘delete’ transformation?
- (c) a ‘change’ transformation?