Input File: –
Output File: –
Source File: trip.pas/.c/.cpp

100 Points
Time limit: 1 s
Memory limit: 16 MB

Solution

The problem asks for printing all different longest common subsequences of two given strings. The first idea might be to use the standard dynamic programming algorithm for determining the length of the longest common subsequences and then construct each longest common subsequence one by one. Let us define $A = (a[1], a[2], \ldots, a[n])$ to be the first string, $B = (b[1], b[2], \ldots, b[m])$ to be the second string, and $c[i, j]$ be the length of the longest common subsequence of $(a[1], \ldots, a[i])$ and $(b[1], \ldots, b[j])$.

Then

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } a[i] = b[j] \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } a[i] \neq b[j] \end{cases}$$

The length of a longest common subsequence of A and B is then $c[n, m]$.

With this recurrence equation of $c[i, j]$ it is easy to come up with the following dynamic programming algorithm:

```
for i:=1 to n do
    c[i,0] := 0
for j:=0 to m do
    c[0,j] := 0
for i:=1 to n do
    for j:=1 to m do
        if a[i] = b[j] then
            c[i,j] := c[i-1,j-1]+1
        else
            c[i,j] := max(c[i-1,j],c[i,j-1])
```

To construct a longest common subsequence, we have to trace back in the table c where the maximum value has come from. This can be done recursively like this:

```
traceback(i,j,sequence)
    if i = 0 or j = 0
        add sequence to the set of longest common subsequences
        return
    if a[i] = b[j] then
        prepend a[i] to sequence
        traceback(i-1,j-1,sequence)
        return
    if c[i-1,j] = c[i,j] then
        traceback(i-1,j,sequence)
    if c[i,j-1] = c[i,j] then
```

```
    traceback(i,j-1,sequence)
```

If we look closer to this code, we will notice that in the case that $c[i-1,j] = c[i,j-1]$ there are two calls to traceback. Even if it is said there are at most 1000 longest common subsequences, this doesn't tell about the number of possibilities to construct them. This function will try all ways to construct all possible longest common subsequences. For the two strings "aaaaaaabcccccccd" and "abbbbbbbcddddddd" there are already 1778966 possibilities to construct the only longest common subsequence of these strings, "abcd".

Who can one avoid to construct the same sequence several time? It can avoid this by constructing the sequences simultaneously with the length. We need a table of sets that contain all longest common subsequences of $(a[1], \ldots, a[i])$ and $([b1], \ldots, [bj])$.

The new dynamic programming algorithm now looks like this:

```
for i:=1 to n do
    c[i,0] := 0
    set[i,0] := empty
for j:=0 to m do
    c[0,j] := 0
    set[0,j] := empty
for i:=1 to n do
    for j:=1 to m do
        if a[i] = b[j] then
            c[i,j] := c[i-1,j-1]+1
            set[i,j] := set[i-1,j-1]
            append a[i] to the end of all sequences in set[i,j]
        else
            c[i,j] := max(c[i-1,j],c[i,j-1])
            if c[i-1,j]>c[i,j-1] then
                set[i,j] := set[i-1,j]
            else if c[i,j-1]>c[i-1,j] then
                set[i,j] := set[i,j-1]
            else
                set[i,j] := union(set[i-1,j],set[i,j-1])
```

Still, there remains a problem: How much memory does the table of sets take? There are at most 1000 longest common subsequences with a maximum length of 80, and the dimension of the table is 80×80. This would be 512 MB if a static array is used, or a bit less if instead a table of linked lists is used. However, with the full table it is only possible to solve nine of the ten test cases within the memory constraint of 16 MB.

Which lines are actually needed during the computation? You can see that inside the for i:=1 to n loop only the actual row and the previous row of the table is needed. So it is possible to use a table with dimensions 2×80 and index into the table with i modulo 2.

So the final algorithm will be:

```
for i:=1 to n do
  c[i,0] := 0
for j:=0 to m do
```

```
   c[0,j] := 0
   set[0,j] := empty
   set[1,j] := empty
for i:=1 to n do
   for j:=1 to m do
      if a[i] = b[j] then
         c[i,j] := c[i-1,j-1]+1
         set[i mod 2,j] := set[1-(i mod 2),j-1]
         append a[i] to the end of all sequences in set[i mod 2,j]
      else
         c[i,j] := max(c[i-1,j],c[i,j-1])
         if c[i-1,j]>c[i,j-1] then
            set[i mod 2,j] := set[1-(i mod 2),j]
         else if c[i,j-1]>c[i-1,j] then
            set[i mod 2,j] := set[i mod 2,j-1]
         else
            set[i mod 2,j] := union(set[1 - (i mod 2),j],set[i mod
               2,j-1])
print all sequences in set[n mod 2, m]
```

There is also a much faster solution. The idea is to use a modified version of the traceback function described above. We try to evaluate only constructions of longest common subsequences x where letter $x[i]$ is the first occurence after letter $x[i-1]$ in both strings. All other constructions wouldn't lead to other longest common subsequences. To achieve this requirement breadth first search can be used.

This may look like this:

```
traceback2(i,j,sequence)
   if i = 0 or j = 0
      add sequence to the set of longest common subsequences
      return
   insert (i,j) in queue
   while queue is not empty
      (k,l) := first element of queue
      if a[k] = b[l] then
         if traceback2 was not called with letter a[k] prepended to
            sequence
            prepend a[k] to sequence
            traceback2(i-1,j-1,sequence)
      else
         if c[k-1,l] = c[k,l] and (k-1,l) is not already in the
           queue then
            append (k-1,l) to the queue
         if c[k,l-1] = c[k,l] and (k,l-1) is not already in the
           queue then
            append (k,l-1) to the queue
```