INF 4130 Exercise set 3, 12th Sept 2013 w/solutions

Exercise 1

1.1 Solve exercise 20.19 (B&P); the solution to (a) was shown in the lecture, but solve (b).

a) See the lecture slides.

b) The two nested FOR-loops give a running time of $O(n*m)$.

1.2 Run the algorithm (on paper) with two similar words, e.g., "algori" og "logari", and with two identical words.

```
          l o g a r i
        0 1 2 3 4 5 6
      --------------
    0 | 0 1 2 3 4 5 6 <- Initialization
  a 1 | 1 1 2 3 3 4 5
  l 2 | 2 1 2 3 4 4 5
  g 3 | 3 2 2 2 3 4 5
  o 4 | 4 3 2 3 3 4 5
  r 5 | 5 4 3 3 4 3 4
  i 6 | 6 5 4 4 4 4 3
      --------------
        ^
        Initialization
```

```
          l i k e
        0 1 2 3 4
      ----------
    0 | 0 1 2 3 4
  l 1 | 1 0 1 2 3
  i 2 | 2 1 0 1 2
  k 3 | 3 2 1 0 1
  e 4 | 4 3 2 1 0
      ----------
```

1.3 It might be difficult to convince oneself that the arguments used at slide 6 of those from the lecture at September 3 are good enough. That is: How can we be sure that we get the best edit distance between P[1..i] and T[1..j] by looking at the three cases discussed at that slide?

So the exercises is to look at this once more, and to work out more detailed argument showing that this is correct.

When proving that the algorithm is correct, we shall use an example where T = ”logaru” and P = ”algor”, and we assume that we shall transform T into P with as few operations as possible. With the algorithm used in 1.2 we get the following table:

```
                                J=6
          T =    l o g a r u
              0 1 2 3 4 5 6
   D         --------------
          0 | 0 1 2 3 4 5 6
P =   a   1 | 1 1 2 3 3 4 5
      l   2 | 2 1 2 3 4 4 5
      g   3 | 3 2 2 2 3 4 5
      o   4 | 4 3 2 3 3 4 5
i=5   r   5 | 5 4 3 3 4 3 4
```

Our task is to explain why this algorithm is correct. This is obviously the case if can prove that if D[i,j-1], D[i-1,j] and D[i-1,j-1] are the (minimum) edit distance for the corresponding prefix strings of T and P, we can compute the correct value of D[i,j] by taking the minimum of these three values, and add one (where we assume that T[j] is different from P[i]),

We will use the computation of D[5,6] as an example. So, generally, we want to transform “logaru” to “algor” with as few operations as possible. We start by picking such a shortest sequence of operations leading from “logaru” to “algor”, and we then sort these operations so that they are made from left to right in T. For our example, these steps could be:

We start with T = “logaru”

1. Insert a at the start of T, getting “alogaru”
   (Do nothing with the l, still having “alogaru”)
2. Remove the o, getting “algaru”
   (Do nothing with the g, still having “algaru”)
3. Change the a to o, getting “algoru”
   (Do nothing with the r, still having “algoru”)
4. Remove the u, getting “algor”, which is P!

As we assumed that this is a shortest transformation, the edit distance between alogaru and algor is obviously d = 4 (as we, encouragingly, also got in the table above). We now look at the last step made, which was to remove u. We then claim that the edit-distance before the last step (that is, between the strings T = “logaru” and “algoru”) must have been d -1 = 3, as the ED increases by one with each operation we perform.

As we assume that the last character of T and of P are not equal, the last step must have been one of the three legal ones, and must involve either the last charcter of T (delete it), the last character of P (insert it at the end of T), or both (change the last character of T to the last character of P).

Thus, if D[i-1,j], D[i,j-1], and D[i-1,j-1] are the correct optimal values for the corresponding cases, one of these must be d-1, and which it is depends on what the last operation was. Also, none of the other two can be smaller than d-1, for then a transformation between T = ”logaru”

and P = ”algor” with a smaller edit distance than d could be found, and this is contrary to the assumption.

Thus, if D[i-1,j], D[i,j-1], and D[i-1,j-1] , are the correct ED between the corresponding strings, then the ED between the strings corresponding D[i,j] is the smallest of these pluss one. Thus we have shown that the basic step of the algorithm is correct, and with the obviously correct initialization and by doing the operation in a bottom up fashion, we have shown that the algorithm will work correct.

1.4 Show how to implement the algorithm using only one column (or row) plus a few additional variables.

We want to calculte the values in the table is it is described above, we assume it has dimensions $D[0{:}m,0{:}n]$. We index it with $D[i,j]$, and want the value of $D[m,n]$.

We calculate row by row from the top down, in our algorithm we now use an array DR[0:*n*] that we initialize with 0, 1, 2, ..., *n.* During execution this array will contain values from row *i* in $DR[0{:}j]$, and values from row *i*-1 in $DR[j{+}1{:}n]$
We also need two new variables, "newDij" and "prevoius". (One, as the exercise suggested, is not enough.) One such row plus the two extra variables are marked with bold and italics in the answer for 1.2 above. The program looks like this:

```
for j = 0 to n do { DR[j] = j }  // Initializing DR (row zero)
previous = 0                             // In general: the value of D[i-1, j-1]
for i = 1 to m do {
   DR[0] = i                             // Initialization of column zero
   for j = 1 to m do {
      if P[i] == T[j] then newDij = previous
      else newDij = min(DR[j], previous, DR[j-1])
      previous = DR[j]
      DR[i] = newDij
   }
}
```

Exercise 2

Look into memoization – using a table as in standard dynamic programming, but with an algorithm following the recursive formula top-down. The trick is now that each recursive call first looks in the table, checking if the answer to the current sub-problem is already calculated and can be returned, or if it needs to be calculated, stored, and returned.

a) Write such an algorithm for exercise 20.19.

We initialize the array in $D[0{:}m,0{:}n]$ just like in 20.19 (in zeroth row and zeroth column), and initialize the rest of the array to -1 to indicate that no value is calculated for this sub-problem (0 is a possible calculated value).

```
function EdDist(i,j): int {              // Called from outside with (m,n)
   if D[i,j] >= 0 then return D[i,j]
   else {
```

```
        if P[i] == T[j] then D[i,j] = EdDist[i-1,j-1]
        else D[i,j] = min(EdDist[i-1,j], EdDist[i-1,j-1], EdDist[i,j-1]) + 1
        return D[i,j]
    }
}
```

Note that the recursion always stops because of the initialization.

Exercise 3

Solve exercise 20.20 (B&P) Hint: try some clever initialisations. Run the algorithm (on paper) on a few examples, for instance, P="hvis" og T="haiehus" with K=2, and "lag" and "varelager" with K=1.

(As far as I (SK) can see now, the solution to this exercise that has been used for a number of years (with initiazation of the zeroth row to zero) , is not correct. It does not seem possible to trace down the the correct parts of T from the numbers in the resulting table. The only satesfying solution I have found is the following, but it takes |P| times more time than the earlier proposal. Can somebody find a better solution?)

We simply look at all parts of T that can possibly be a match. We start by comparing the |P|+K first letters of T with P, and do it as in Exercise 1 above. We then take away the first character of T, and repeat the process until there are less than |P| - K letters left in T. The reason that we look at parts of T of length |P|+ K is that this is the longest parts that can possibly contain a match og edit-distance at most K. We can stop when less than |P| - K charcter are left in T, as no shorter interval of T can have ED smaller than K to P. To avoid getting the same match twice, we, in each step, only register matches that starts at the beginning of the current interval. If we look at the second example from the exercise ("lag" and "varelager" with K=1), we get the following comparisons:

Compare “lag” with “vare”: No match with ED = 1 or 0
Compare “lag” with “arel”: No match with ED = 1 or 0
Compare “lag” with “rela”: No match with ED = 1 or 0
Compare “lag” with “elag”: One match with ED = 1 (“elag”)
Compare “lag” with “lage”: One match with ED = 0 (“lag”) and two with ED = 1 (“la” and “lage”)
Compare “lag” with “ager”: One match with ED = 1 (“ag”)
Compare “lag” with “ger”: No match with ED = 1 or 0
Compare “lag” with “er”: No match with ED = 1 or 0 Last one, as |P| - K = 2.

We show two of the steps below

```
        e l a g
      0 1 2 3 4
    ----------
  0 | 0 1 2 3 4
l 1 | 1 1 1 2 3
a 2 | 2 2 2 1 2
g 3 | 3 3 3 2 1    Can  be traced back to “elag”
    ----------

        l a g e
      0 1 2 3 4
    ----------
  0 | 0 1 2 3 4
l 1 | 1 0 1 2 3
a 2 | 2 1 0 1 2
g 3 | 3 2 1 0 1    Can be traced back to “la”, “lag”, and “lage”
    ----------
```

We leave the first example in the exercise (P="hvis" og T="haiehus" with K=2) to the audience.

Exercise 4 (if there is time)

We are going to solve the Money-Changing Problem using dynamic programming: The problem is to return *K* pence (cents/øre, or whatever) using as few coins as possible. Let $\{v_1, v_2, \ldots, v_n\}$, be the coin denominations, for instance, $v_1 = 25$, $v_2 = 10$, $v_3 = 5$, $v_4 = 1$. We assume the denominations are integer, and that $v_{i-1} < v_i$. We further assume $v_n = 1$, so that a solution always exists. We do, of course, always have enough coins.

For the currency described above, a greedy solution will work. (And it is an interesting exercise to prove just that.) BUT if we remove the 5 pence coin, we end up in trouble: A greedy algorithm returning change for 30 pence, will return one 25 pence coin, and five pennies, six coins in total, twice as many as the optimal three 10 pence coins. Dynamic programming finds optimal solutions for all possible currencies, including those where greedy algorithms fail.

Let $C[j]$ be the number of coins in a solution where we return change for *j* pence. If a solution uses a coin of denomination v_i, we have $C[j] = C[j - v_i] + 1$.

a) Give a recursive formula for the value of an optimal solution.

Usually one can just take the largest coin available, but with strange currencies we have to check every possible coin (among the *n* possible). The formula is therefore:

$$C[j] = \begin{cases} 0 & \text{if } j = 0 \\ \min\limits_{1 \le i < n} C[j - v_i] + 1 & \text{if } j \ge 1 \end{cases}$$

b) Write (pseudo) code for an algorithm based on the formula from a).

```
Change(K, V, n)            // K is the amount, V[i:n] the coin denominations
{
   C[0] = 0
   for j = 1 to K do {
      C[j] = MAXINT        // infinity
      for i = 1 to n do {
         if j >= V[i] and C[j - V[i]] + 1 < C[j] then C[j] = C[j - V[i]] + 1
         // j >= V[i] avoids indices below zero (index out of bounds)
      }
   }
   return C[K]
}
```

Note that we only find the number of coins here… If we want to know which coins, we use another table CV and set CV [*j*] = V [*i*] when we update C[*j*] .

c) Run the algorithm with $v_1 = 30$, $v_2 = 24$, $v_3 = 12$, $v_4 = 6$, $v_5 = 3$, $v_6 = 1$, og $K = 48$. (English coins, not too long ago!) [The greedy solution is 30+12+6, by the way.]

We show how the table evolves during the first few steps of the algorithm. The array is one-dimensional; each line below represents an iteration of the algorithm.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	…
0	1																		
0	1	2																	
0	1	2	1																
0	1	2	1	2															
0	1	2	1	2	3														
0	1	2	1	2	3	1													

We find the answer in cell 48. We can use two 24's (twoshilling or florin). We get one from C[24], and another one in C[48].

d) What is the running time of your algorithm?

The running time is, as usual, dominated by the FOR-loops. (In this case there is a bit more FOR-loop than table, as we check all possible coins for every amount.) The running time is $O(K*n)$, or more exactly $\Theta(K*n)$. However, the n here is very small, there are rarely more than 4-5 coins in any currency. For a fixed currency this n will be a constant and disappear in the O-notation, but in the general case, where the coins are part of our input, it needs to be taken into account.