Answers to weekly exercises - 12/9

Reidar André Brenna
reidarab
reidarab@ifi.uio.no

16. september 2013

1

1.1.a

See the lecture notes.

1.1.b

The double(nested) for loops give us a running time of O(n*m).

1.2

Hint when solving this by hand note that you have two options: if P[i] = T[j] then you fill the square"with the value from one diagonaly up to the left. if P[i] != T[j] then you fill the square with 1 + the lowest value of these squares: the one directly to the left, the one directly above and the one diagonaly up to the left.

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

```
        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
```

Note The answear is located in the bottom right square.

1.3

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:

```
          l o g a r u
        0 1 2 3 4 5 6
       --------------
   0 | 0 1 2 3 4 5 6
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
```

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

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.) The program looks like this:

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

2

2.a

We initialize the array in D[0:m,0:n] just like we did 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).

```
/** Finds the value in the table from the given indexes.
 * Called from outside with (m,n).
 *
 * @param i the i index (m)
 * @param j the j index (n)
public int EdDist(i,j) {

//checks if the value have been previously calculated.
if (D[i,j] >= 0) {
    return D[i,j];
    } else {
        //check if P[i] and P[j] are equal
        if (P[i] == T[j]) {
            //sets the value in the table to the one diagonaly up to the right.
            D[i,j] = EdDist[i-1,j-1];
        } else {
            //sets the value in the table to the minimum of the left, the one
            //above and the one up to the right pluss one.
            D[i,j] = min(EdDist[i-1,j], EdDist[i-1,j-1], EdDist[i,j-1]) + 1;
            return D[i,j];
        }
    }
}
```

Note The recursion always stops because of the initialization.

3

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 and 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’ll 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”
```

4

4.a

In general one can just take the largest coin possible, but with strange currencies we have to check every possible coin of the n available to us.

4.b

```
/** Finds the number of coins.
 * If we want to find which coins, we would need another table,
 * cv for instance, and set cv[j] = v[i] when updating c[j].
 *
 * @param K the amount.
 * @param V the coin denominations available (V[i:n]).
 * @param n the amount of different coins.
 */
public int Change(K, V, n) {
    C[0] = 0;
    for (j = 1; j < K; j++) {
        C[j] = MAXINT;              // infinity
        for (i = 1; i <= n; i++) {
            // j >= V[i] avoids indices below zero (index out of bounds)
            if ((j >= V[i]) && ((C[j - V[i]] + 1) < C[j]) {
                C[j] = C[j - V[i]] + 1;
            }
        }
    }
    return C[K]
}
```

c

4.d

Like before the double foor loop is the dominating source of run time in this algorithm, so in general we would end up with a run time of O(k*n). One could argue that the run time would be closer to $\Theta(k * n)$. In any given example run of the code n would be a (rather smal) constant and therfore be insignificant to the run time, providing a run time of $\Theta(k)$, but we need to include the n for general purposes.