‘INF 4130 Exercises, Sept. 18 and 20, 2018 w/solutions

Exercise 1

1.1 Run the edit distance 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
     ----------
```

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

Answer:

We want to calculte the values in the table is it as 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" og "prevoius". The program will look 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
   }
}
```

1.3 Solve the problem given in the last sentence of section 20.5 on page 645. That is: In the slides we originally wanted to find an algorithm for searching through a string T, and look for substrings S = T[p], T[p+1], … , T[q] of T similar to a given string P. We can assume that we want to find the first substring of T whose edit distance to P is less than or equal to a given K (or report that no such substring occurs).

Answer:
The trick is to initialize row zero (along the direction of T) with only zeroes. This has the effect that we allow a new substring S of T with small enough ED to P to start anywhere in T (but see below). We look at the following example:

```
T = a b a e g b c d b a c d g a . . .
P = a b c d
K = 1
       a  b  a  e  g  b  c  d  b  a  c  d  g  a  .  .  .
    ----------------------------------------------------
    0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  .  .  .
                     |              \
a   1  0  1  0  1  1  1  1  1  1  0  1  1  1  0  .  .  .
                        \              |
b   2  1  0  1  1  2  1  2  2  1  1  1  2  2  1  .  .  .
                           \              \
c   3  2  1  1  2  2  2  1  1  2  2  1  2  3  2  .  .  .
                              \              \
d   4  3  2  2  2  3  3  2  1  2  3  2  1  2  3  .  .  .
```

We can here observe that we get 1 (≤K) two times in the last row, and for these we can find the corresponding subsequence S of T by going backwards from each of the 1-values in the last row, as shown in the picture (and as we did for the simple edit distance case). Thus we see that these

are *S = gbcd* and *S = acd* respectively, and we can also see what the correct edit operation is (even if this requires a little thinking!).

One might protest to the above argument for initializing the top row with only zeroes (which was: "we then allow a new substring of T to start anywhere in T") by saying that we might then get false small values in the bottom row, as the top row along the found substring is only zeroes, instead of 0, 1, 2, … as we usually have when computing the edit distance. However, there can be no such influence as the backwards path we found from the lower row describes the influence we have used, and this path do not reach the top row until the start of S.

When executing this algorithm it is natural to fill column after column (starting each time with a zero at the top), and when we get K or less in the last row entry and we only want the first occurrence in T, we can stop and find the corresponding substring S of T.

We can obviously also do this with only one array of the same lengh as P plus a few variables, as in Exercise 1.3 above. If we then want all occurences of legal S-strings in T, we could then, during the search, simply remember at what indices in T we get edit distance ≤ K, and then afterwards go back to these places in T and find the corresponding substrings S.

Example, time usage: How much time would this algorithm use to search through our entire genome (about $3*10^9$ letters), for a string that is e.g. 100 = 10^2 letters long. Then we would have to compute the recurrence formula $3*10^{11}$ times. Assuming a machine (with caching etc.) using an average of 10 ns to fetch data from the store, we may assume 100 ns = 10^{-7} seconds for each computation of the recurrence formula. Thus, a full search would take $3*10^{11} * 10^{-7} = 3*10^4$ seconds, which is about eight hours. Thus, this is doable, but the biologists usually also need some extra "weight values" in the recurrence formula, and they usually want to search for longer strings than 100 letters (often more than 1000 letters). Thus, doing it straight-forward as above usually takes too much time. On can to same extent optimize the above algorithm, but for many real cases one still has to introduce special tricks to speed up the process, which usually also has the bad effect that the search becomes approximate.

For more information, see e.g. https://en.wikipedia.org/wiki/Human_genome. We will also later have a guest lecture by Torbjørn Rognes from the Bio-Informatics group about the algorithms they are using.

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 into the table, and checks if the answer to the current sub-problem is already calculated. If it is, this value is used, otherwise we have to do recursive calls to solve the necessary smaller problems.

Write such an algorithm for finding the edit distance between two strings P and T.

The array *D*[0:*m*,0:*n*] is just like in 20.19, initialize it the same way, 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 (repetition and extension of a problem from the lecture)

We assume that we have an integer array "C[0:m-1, 0:n-1]" filled with positive intergers. We want to find the cheapest path from C[0,0] to C[m-1, n-1], where the cost of a path is the sum of the integers in the array-entries it passes through. A path can only pass from one entry C[i,j] to the one to the right of it (C[i, j+1]), to the one below it (C[i+1, j]) or to the one diagonally down to the right of it (C[i+1,j+1]), and it has to stay within the borders of the array.

3.1 Describe what sort of table you want to use for solving this problem with dynamic programming and write down the recurrence formula you want to use. How will you initialize the table?

Answer 3.1

One can use an integer array T of the same form and size as the array C, and let T[i, j] be the length of the shortest path from T[0, 0] to T[i, j].

The recurrence relation will be: T[i, j] = min (T[i-1, j] T[i, j-1], T[i-1, j-1]) + C[i, j]

One can initialize the topmost row and the leftmost column. For each of these entries these is only one possible path from C[0, 0]. This can be done as follows:

```
T[0, 0] = C[0, 0];
for (i = 1, i < m, i++) do { T[i, 0] = T[i-1, 0] +C[i, 0]; }
for (j = 1, j < n, j++) do { T[0, j] = T[0, j-1] +C[0, j]; }
```

3.2 Sketch a program for solving the problem

Answer 3.2

```
<The initialization as above>;
for (i = 1, i < m, i++) do {
  for (j = 1, j < n, j++) do {
    T[i, j] = min (T[i-1, j] T[i, j-1], T[i-1, j-1]) + C[i, j];
  }
}
answer = T[m-1, n-1];
```

3.3 This problem can be seen as a shortest path problem with (m x n) nodes, and is thereby solvable with Dijkstra's algorithm. Compare the speed of the two methods, and try to find the source of the difference.

Answer 3.3

The underlying problem can be seen as a shortest path problem in a graph with (n x m) nodes where all the tree edges leading out of each node (with some exceptions) have the same cost as the cost of that entry.

Dijkstra's algorithm will for D nodes and E edges in general take $O(V^2)$ or O(E + V logV), depending on the implementation. With Dynamic Programming the execution time will be O(n x m), which is O(V), which is smaller than each of those above, even if we know that E ≈ 3 x V. The important property of the current graph is that it has no circuits, which simplifies the problem.

3.4 Assume we want to solve the above problem with top-down recursive memorization. In what cases can we avoid computing all the entries in our table?

Answer 3.4

At the outset we will always have to compute all entries of T. We will never know where a small C-value will occur (e.g. in the lower left or upper right corner), and this may change the whole picture. However, as only positive integers are allowed in D, we may in some cases know that we do not need to compute certain areas of T. We leave the details of this to the reader.

Exercise 4

Make sure that everybody understand why the Fibonacci algorithm we looked in the lecture (see slides) is *not* a polynomial time algorithm, but that it uses exponential time in the size of the input (the number of digits in n). Compare with the algorithms for addition and multiplication of two integers, which indeed are polynomial time algorithms in the size (number of digits) of the problem.