IN3130 Exercise set 4

Dynamic Programming Problems

Exercise 1

Look at the problem of finding the "best path" (lowest weight) from the upper left to the lower right corner. (First example from the lecture.) Make sure everybody understands what is going on, by discussing the following points:

a) Indicate what different orders the matrix P can be filled out, if we want to have the necessary results when we need them.
b) How can we find the shortest path itself, not only its weight?
c) What is the (time) complexity of this algorithm?

This is only to make sure that everybody has understood the simple example, and shouldn't take too much time. Look at the slides from the lecture. The complexity of the algorithm is O(n*m).

Exercise 2

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

Answer:

		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	< Initialization
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	

^
Initialization

b) Show how to implement the Edit Distance algorithm using only one row or one column plus a few extra variables.

Answer:
We want to calculate 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" and "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
   }
}
```

c) On the lecture slides (the example on slide 12) we looked at the output from an algorithm for searching through a string T, looking for substrings S = T[p], T[p+1], …, T[q] of T that are similar to a string P. The example says nothing about how similar is similar enough. It was just used as an intro to the Edit Distance concept. Describe an algorithm that finds the first substring of T whose Edit Distance to P is less than or equal to a given K (or report that no such string 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 BioInformatics group about the algorithms they are using.

Exercise 3

Look into Memoization – using an algorithm following a recursive formula top-down, while storing solutions to sub-problems in a table like in standard Dynamic Programming. The memorization trick is that each recursive call first checks the table, to see if the solution to a sub-problem has been calculated already. If it is, that value is used. Otherwise, we have to do recursive calls to solve the necessary (smaller) subproblems.

a) Write a memorized algorithm for finding the edit distance between sting P and T.

ANSWER:

The array *D*[0:*m*,0:*n*] is just like in B&P 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 4

The Fibonacci numbers F(n) are defined by the formulas:

F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2), for $n \geq 2$

One can compute F(n) for a given n by building up the sequence F(0), F(1), F(2), …, F(n) like this:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

a) This computation can be seen as Dynamic Programming computation where we have a one-dimensional table holding all the values we have already computed, and use some of them to compute the next value. Of which order is this computation when expressed in O-notation of the value n?

ANSWER: $O(n)$

b) In what sense is it ***not*** reasonable to say that this is a polynomial algorithm? Compare with the speed of adding by hand two numbers n and m. What if that computation used time e.g. $O(m + n)$?

COMMENT: This sort of "polynomial time algorithm" is often said to run in "quasi-polynomial time".

Answer:

The point here is that the speed of an algorithm is usually expressed as a function of the length of the input, and in this case the length is the number of digits in n, which is about $\log_{10}(n)$. From

this viewpoint the time used by the algorithm is time $O(10^L)$, where L is the length of the instance, and it is therefore exponential! We should observe that the procedure to e.g. multiply two numbers p and q by hand has order O($\log_{10}(p)$ *$\log_{10}(q)$), which really *is* polynomial in the number of digits in p and q. One can try to imagine how hopeless things would be if this procedure took "quasi-polynomial" time, using time of order O(p+q).

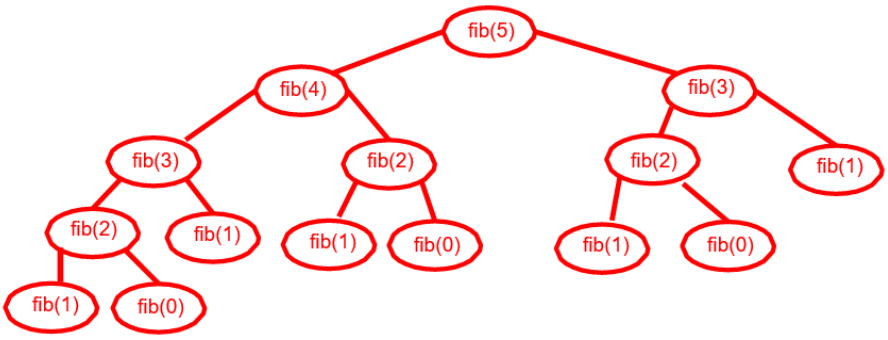
We generally say that an algorithm runs in "quasi-polynomial time" when it becomes polynomial- time if we say that the length of the instance should be computed by letting each *number* in the instance contribute with their *value* (and *not their number of digits*). This corresponds to representing each number in the instance in "unary coding", as defined in the answer to problem 1 in the exercises for the first week of this course. A number of Dynamic Programming algorithms run in quasi-polynomial time, e.g. the one occurring in mandatory exercise 1.

c) Assume we used the formula F(n) = F(n-1) + F(n-2) to write a simple recursive program for F(n), without trying to remember any previously computed values. What would be the execution time for such a program?

Answer: The recursive procedure could e.g. be:

```
Procedure fib(interger n )
{
   if (n<0) {error … ;}
   else {
   if (n=0) {return 0;}
   else {
   if (n=1) {return 1;}
    else {
       return fib(n-1) + fib(n-2);
   }}}
}
```

The calls of an execution of this program, would execute as slightly skew binary tree, e.g. like this:



Even if this tree is somewhat skewed, its number of nodes will be $O(2^n)$. Thus, expressed in the *length* of n (its numbers of digits!) this algorithm will be *doubly* exponential! This is because we do a number of computations again and again.

[end]