Algorithms and Complexity. Exercise session 3+4

Dynamic Programming

Longest Common Substring The string `ALGORITHM` and the string `PLÅGORIS` have the common substring `GORI`. The *longest common substring* of these strings has thus length 4. In a substring characters must be in a coherent sequence.

Construct an efficient algorithm that given two strings $a_1 a_2 \dots a_m$ and $b_1 b_2 \dots b_n$ calculates and returns the length of the longest common substring. The algorithm is based on dynamic programming and runs in time $O(nm)$.

Sequences You are given two sequences of positive integers $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_n$, where all numbers are less than n^2, and a positive integer B, such that $B \leq n^3$. The problem is to determine if there is a sequence $c_1, c_2, \dots, c_n$ such that $\sum_{i=1}^{n} c_i = B$ and $c_i = a_i$ or $c_i = b_i$ for $1 \leq i \leq n$.

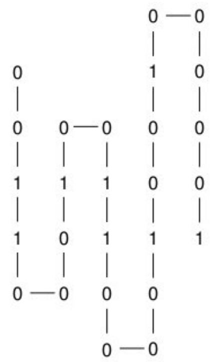
Describe and analyze an algorithm that solves this problem by using dynamic programming. Moreover, describe how to extend the algorithm so it also talks about how the solution looks like, when $c_i = a_i$ or $c_i = b_i$ for $1 \leq i \leq n$.

Protein Folding A protein is a long chain of amino acids. The protein chain is not straight but it is folded in an intricate way that minimizes the potential energy. It would nice to be able to figure out how a protein will fold itself. In this exercise we will therefore consider a simple model of protein folding in which amino acids are either *hydrophobic* or *hydrophilic*. Hydrophobic amino acids tend to clump together.

For simplicity, we can see the protein as a binary string in which ones correspond to hydrophobic amino acids and zeros hydrophilic amino acids. The string (protein) should then be folded into a two-dimensional square lattice. The goal is to make the hydrophobic amino acids to stick together, that is to get as many ones as possible to be close to each other. So we have an optimization problem where the objective function is the number of pairs of ones that are next to each other in the grid (vertically or horizontally) without being next to each other in the string.

You will design an algorithm using dynamic programming to construct an optimal *accordion fold* of a given protein string of length n. An *accordion fold* is a fold where the first string is a stretch straight down, then goes straight up, then goes straight down, and so on. In such a fold, it can be observed that the vertical pairs of adjacent ones will always result in the string, so it's only horizontal couple of ones that contribute to the objective function.

The following figure shows the string 00110001001100001001000001 of accordion weights in such a way that the objective function becomes 4.



Definition of the problem PROTEIN ACCORDION FOLD:

INPUT A binary string of n characters.

PROBLEM: Find the accordion fold of input string that provides the greatest value to the objective function, ie the largest number of pairs of ones located next to each other, but not close to each other in the string.

Construct and analyze the time complexity of an algorithm that solves protein accordion folding problem with dynamic programming.

Use the following algorithm which calculates the number of pairs of ones in a row (ie between two lines) lying next to each other (but not to each other in the string). Suppose that the protein is stored in an array `p[1..n]`. The parameters `a` and `b` indicate the index in the array for the first trait endpoints. The parameter `c` indicates the index for the second trait endpoint. Look at the figure below on the right.

```
profit(a,b,c) =
shortest←min(b-a,c-(b+1));
s←0;
for i←1 to shortest do
if p[b-i]=1 and p[b+1+i]=1 then
s←s+1;
return s;
```

Note: Protein folding is an important algorithmic problem studied in bioinformatics. Similar problems are studied in the *Algoritmisk bioinformatik* course.

Analyzer for context-free grammars A context-free grammar is usually used to describe the syntax of particular programming language. A context-free grammar in *Chomsky Normal Form* is described by

- a variety of final symbols T (usually written in small letters),
- a variety of non-final symbol N (usually written in capital letters),
- the start symbol S (a non-final symbol in N),
- a number of rewrite rules that are either in the form $A \to BC$ or $A \to a$, for $A, B, C \in N$ and $a \in T$.

If $A \in N$ we define $\mathcal{L}(A)$ as

$$\mathcal{L}(A) = \{bc : b \in \mathcal{L}(B) \text{ and } c \in \mathcal{L}(C) \text{ where} A \to BC\} \cup \{a : A \to a\}.$$

The language generated by the grammar is now defined as $\mathcal{L}(S)$, ie. by all the strings of the final symbols that can be formed by a rewriting chain starting with the start symbol S.

Example: Consider the grammar $T = \{a, b\}$, $N = \{S, A, B, R\}$, start symbol S and rules $S \to AR$, $S \to AB$, $A \to a$, $B \to b$, $R \to SB$. We can see that the string $aabb$ belongs to the language generated by the grammar using the following chain of rewritings:

$$S \to AR \to aR \to aSB \to aSb \to aABb \to aaBb \to aabb.$$

In fact, one can show that the language generated by the grammar is all strings consisting of k symbols a followed by k symbols b, where k is a positive integer.

Your task is to *design* and *analyze* an efficient algorithm that determines if a string belongs to the language generated by a grammar. The input is thus a context-free grammar in Chomsky Normal Form, and a string of final symbols. The output is true or false depending on whether the string could be generated by the grammar or not. Calculate the time complexity of your algorithm in terms of number of rules m of the grammar and the length n of the string.

You can read more on grammars in the course *Artificiella språk och syntaxanalys*.
