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 the characters must be in a coherent sequence.

Construct an efficient algorithm that given two strings $a_1a_2\ldots a_m$ and $b_1b_2\ldots b_n$ calculates and returns the length of the longest common substring. The algorithm is based on dynamic programming and runs in time $O(nm)$.

Solution to Longest Common Substring
Let $M[i,j]$ be the number of letters to the left of (and including) a_i complying with the same number of letters to the left (and including) b_j. The length of the longest common string is the largest number in the matrix M.

M can be defined recursively as follows:

$$M[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ M[i-1,j-1]+1 & \text{if } a_i = b_j, \\ 0 & \text{otherwise.} \end{cases}$$

The following algorithm computes M and returns the largest number in M.

```
max ← 0
for j ← 0  to n
    M[0, j] ← 0
for i ← 1  to m
    M[i, 0] ← 0
    for j ← 1  to n
        if a_i = b_j then
            M[i, j] ← M[i − 1, j − 1] + 1
            if M[i, j] > max then max ← M[i, j]
        else M[i, j] ← 0
return max
```

The running time is dominated by the nested *for* loops and is therefore $\Theta(nm)$. □

Sequences You are given two sequences of positive integers $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, 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 $c_1, c_2, \ldots, c_n$ 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 that it also computes the sequence of the solution, when $c_i = a_i$ or $c_i = b_i$ for $1 \leq i \leq n$.

Solution to Sequences
We create a boolean $n \times B$-matrix M that is initially filled with zeros. A one in $M[k,s]$ means that there is a choice of $c_1, \ldots, c_k$ such that $\sum_{i=1}^{k} c_i = s$.

The recursive equation for $M[k, s]$ becomes:

$$M[k,s] = \begin{cases} 1 & \text{if } k = 1 \text{ and } (s = a_1 \text{ or } s = b_1), \\ 1 & \text{if } k > 1 \text{ and } (M[k-1, s-a_k] = 1 \text{ or } M[k-1, s-b_k] = 1), \\ 0 & \text{otherwise.} \end{cases}$$

The calculation begins with $M[1, a_1]$ and $M[1, b_1]$ set to 1. Then we put ones in $M[2, a_1 + a_2]$, $M[2, a_1 + b_2]$, $M[2, b_1 + a_2]$ and $M[2, b_1 + b_2]$ in the second row of the array, then we put it ones in the third row and so on. If it ends up a one in $M[n, B]$, the answer to the problem yes.

The algorithm may look like this in C:

```
int ExistsC(int n, int a[], int b[], int B)
{ char M[n + 1][B + 1]; /* Dynamc arrays are an extension of gcc */
  int i, j;
  for (i = 1; i <= n; i++)
    for (j = 1; j <= B; j++) M[i][j] = 0;
  M[1][a[1]] = M[1][b[1]] = 1;
  for (i = 2; i <= n; i++)
    for (j = 1; j <= B; j++) {
      if (j - a[i] > 0 && M[i - 1][j - a[i]]) M[i][j] = 1;
      if (j - b[i] > 0 && M[i - 1][j - b[i]]) M[i][j] = 1;
    }
  return M[n][B];
}
```

The nested *for* loop runs $(n-1)B$ times and each time it performs more than four comparisons and two assignments, ie a constant number. The whole algorithm is therefore in $O(n^2) + O((n-1)B) = O(n^2 + nB) \subseteq O(n^4)$. By going from the end (from the n-th row and up) and looking at what positions there are ones you can compute the solution. The ones in $M[n, B]$ must be there because of a one in either $M[n-1, B-a_n]$ or $M[n-1, B-b_n]$. Remember which of these it is. If there is a one in two positions, just choose one of them (as only one solution is requested). Continue in this way from the selected position until a row has been reached. Implementation in C is as follows.

```
void WriteC(int n, int a[], int b[], int B)
{ int c[n + 1], pos;

  /* Procedure ExistsC goes here*/

  pos = B;
  for (i = n; i > 1; i++) {
    if (pos - a[i] > 0 && M[i - 1][pos - a[i]]) c[i] = a[i];
    else c[i] = b[i];
    pos -= c[i];
  }
  c[1] = pos;
  printf("The answer is %d", c[1]);
  for (i = 2; i <= n; i++) printf(" + %d", c[i]);
}
```

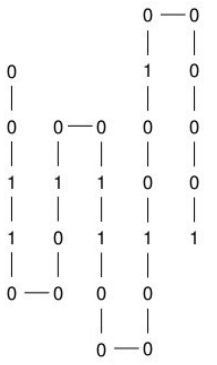
□

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.

Solution to Protein folding

Let $q_{a,b}$ be the maximum value of the objective function obtained for the folding of part $p[a..n]$

of the protein, where the first trait of the folding are the endpoints a and b. We can express $q_{a,b}$ recursively as follows :

$$q_{a,b} = \max_{b+1<c\leq n}(\texttt{profit}(a,b,c) + q_{b+1,c}).$$

The base case is $q_{a,n} = 0$ if $1 \leq a < n$. Then, we answer by computing $\max_{1<b\leq n} q_{1,b}$.
Now it remains to calculate according to these formulas in the right order:

```
for a←1 to n-1 do q[a,n]←0;
for b←n-1 downto 2 do
for a←1 to b-1 do
t←-1;
for c←b+2 to n do
v←profit(a,b,c)+q[b+1,c];
if v>t then t←v;
q[a,b]←t;
max←0;
for b←2 to n do
if q[1,b]>max then max←q[1,b];
return max;
```

Since we mostly have three nested `for`-loops and a call to `profit` takes time $O(n)$, the time complexity is obviously $O(n^4)$. □

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*.

Solution to Analyzer for context-free grammars
We use dynamic programming to solve this problem. Here we will determine the order and the substring rules to apply.

The input is a set of rules R and a vector `w[1..n]` which is indexed from 1 to n. Let us build a matrix `M[1..n,1..n]`, where the elements of `M[i,j]` indicate the set of non-final symbols from which one can derive the substring `w[i..j]` by means of a chain of rewriting rules. `M[i,j]` is recursively defined as:

$$\texttt{M}[i,j] = \begin{cases} \{X : (X \to \texttt{w}[i]) \in R\} & \text{if } i = j \\ \{X : (X \to AB) \in R \wedge \exists k : A \in \texttt{M}[i,k-1] \wedge B \in \texttt{M}[k,j]\} & \text{if } i < j \end{cases}$$

Since each position in the array is a set of non-final symbols, we must choose an appropriate data structure for this. Let us represent a variety of non-final symbols as a bit vector indexed by non-final symbols. A 1 means that the symbol is in the set and a 0 means that it is not in the set. Example: If `M[i,j][`B`]`$= 1$, the non-final symbol B is in the set `M[i,j]`, namely, there is a chain of rewrite rules from B to the substring `w[i..j]`.

The algorithm computes the matrix `M[i,j]` and returns true if the string belongs to the language generated by the grammar:

```
for i←1 to n do
M[i,i]←0; /* all bits are reset */
for each rule X →w[i] do
M[i,i][X]←1;
for len←2 to n do
for i←1 to n-len+1 do
j←i+len-1;
M[i,j]←0;
for k←i+1 to j do
for each rule X → AB do
if M[i,k-1][A]=1 and M[k,j][B]=1 then
M[i,j][X]←1;
return M[1,n][S]= 1;
```

Time complexity: $O(n^3 m)$. Memory: $O(n^2 m)$ (because m is an upper limit on the number of non-final symbols). □