Universitetet i Oslo
Institutt for Informatikk
I. Yu, D. Karabeg

INF2220: algorithms and data structures

Series 11

Topic Dynamic programming (Exercises with hints for solution)

Issued: 2. 11. 2016

Classroom

Exercise 1 (Fibonacci) Design an algorithm to compute Fibonacci numbers in *linear* time.

Solution:

```
public static int fibonacci(int n)
{
  if(n <= 1)
    return 1;

  int last = 1;
  int nextToLast = 1;
  int answer = 1;
  for(int i = 2; i <= n; i++)
  {
    answer = last + nextToLast;
    nextToLast = last;
    last = answer;
  }
  return answer;
}
```

Exercise 2 (Matrix) Let A be an N-by-N matrix of zeros and ones. A submatrix S of A is any group of contiguous entries that forms a square. Design an algorithm that determines the number of elements of the largest submatrix of "ones" in A. For instance, in the matrix that follows, the largest submatrix is a 4-by-4 square.

```
10111000
00010100
00111000
```

```
00111010
00111111
01011110
01011110
00011110
```

Solution: [of Exercise 2] *Hint*: A simple dynamic programming algorithm, in which we scan in row order suffices. Each square computes the maximum sized square for which it could be in the lower right corner. That information is easily obtained from the square that is above it and the square that is to its left.

Exercise 3 Product-sum (from 2014 exam)

Given a list of n integers, $v_1, \ldots, v_n$, the *product-sum* of the list is the *largest* sum that can be formed by multiplying adjacent elements in the list. Each element can be matched with at most one of its neighbors. For example, the product-sum of the list $1, 2, 3, 1$ is 8 $(= 1 + (2 \times 3) + 1)$ and the produckt-sum of the list $2, 2, 1, 3, 2, 1, 2, 2, 1, 2$ is 19 $(= (2 \times 2) + 1 + (3 \times 2) + 1 + (2 \times 2) + 1 + 2)$.

1. Optimization

   Given a list of n integers for $n \geq 2$. What is the optimal product-sum $OPT(j)$ for the first j elements in the list? Meaning, you should find out what $OPT(j)$ is for $j \in \{0, 1, \ldots, n\}$.

2. Dynamic programming

   Give a dynamic programming solution for this problem. Implement the method `prodSum` which returns the product-sum of the first `j` elements of a list `v`:

```
int prodSum(int [] v, j) {

.....     // your code

}
```

   The method only needs to return the product-sum.

Solution:

1. $OPT(0) = 0$
   $OPT(1) = v_1$
   $OPT(j) = max(OPT(j-1) + v_j, OPT(j-2) + (v_j \times v_{j-1}))$ if $j \geq 2$

2.
```
int prodSum (int [] v, j){
  if(j==0) return 0;

  int [] OPT = new int [j+1];
  OPT[0] = 0;
  OPT[1] = v[1];
  if (j >= 2) {
     for int k = 2 to j
        OPT[k] = max ( OPT[k-1] + v[k],  OPT[k-2] + v[k] * v[k-1]);
  }
  return OPT[j]

}
```

Exercise 4 Assume that you have a table with size $n \times m$, with positive values in each cell. Further, assume that you start in the left upper corner, and that you are allowed to go only to the right or downwards. For each cell you enter, you add the cell's value to your counter. Use dynamic programming to find the maximum value you can get by taking any route from your position to the lower right corner.

Solution: Start at the upper left corner. For each cell, going from left to right rowwise, assign a value which is the maxmum of the sum of the cell's value and the value of the cell to the left (if it exists) and the cell's value and the value of the cell above (if it exists).

Exercise 5 (Skip this if you not are familiar with matrix multiplication)

Assume that you are going to multiply n matrices in a given sequence. This can be done in many ways; e.g. as (AB)C or A(BC). The sequence you use will determine the number of multiplications and additions (floating point operations) you need to perform the multiplication. Use dynamic programming to find the most optimal sequence. (You can assume that all multiplications are well definied, i.e. that the dimensions matches. The number of multiplications needed to multiply two matrices of size $(l \times m), (m \times n)$ is of order $O(lmn)$.)

Solution: First two cases are trivial; let $M_1 \in \mathbb{R}^{m_1 \times m_2}$, $M_2 \in \mathbb{R}^{m_2 \times m_3}$ and assign $d_1 = m_1 m_2 m_3$. For the case $n = 3$, assume that $M_3 \in \mathbb{R}^{m_3 \times m_4}$. We can do chain multiplication by $(M_1 M_2)M_3$, which gives

$$O(m_1 m_2 m_3) * O(m_1 m_3 m_4) = O(m_1^2 m_2 m_3^2 m_4) \tag{1}$$

operations, or by $M_1(M_2 M_3)$, which gives

$$O(m_2 m_3 m_4) * O(m_1 m_2 m_4) = O(m_1 m_2^2 m_3 m_4^2) \tag{2}$$

operations. The minimum of these is determined by the minimum of $\{m_1 m_3, m_2 m_4\}$; take

$$d_2 = min\{m_1 m_3, m_2 m_4\} * (m_1 m_2 m_3 m_4) \tag{3}$$

Dynamic programming step: You know the number which minimizes the last two cases. When introducing a new matrix $M_4 \in \mathbb{R}^{m_4 \times m_5}$, take the minimum of the operations given by $(M_1 M_2)(M_3 M_4)$ and $(M_1 M_2 M_3)M_4$, i.e. save the value

$$min\{d_1(m_3 m_4 m_5)(m_1 m_3 m_5), d_2(m_1 m_4 m_5)\} \tag{4}$$

(possibly use a temporary variable tmp, assign $d_1 = d_2; d_2 = tmp$, which makes it easy to generalize). In general, for the $k'th$ multiplication: Compute all values up to the k'th iteration, and then take

$$min\{d_1 m_1 m_{k-1}^2 m_k m_{k+1}^2, d_2 m_1 m_k m_{k+1}\} \tag{5}$$

Lab

Exercise 6 Create a simplified implementation of a *Huffman* compression where you:

1. Reads the input file and create the frequency table for each letter.
2. Use the frequency table plus a binary heap to create Huffman tree.

3. Saves the bit representation of characters as String (ex: "`001`").

4. Print {`letter, frequency, binary representation`} and about how much space would be saved by compressing the file with your Huffman tree.

Exercise 7 The *longest common subsequence* problem is as follows: Given two sequences $A = a_1, a_2, ...a_m$, and $B = b_1, b_2, ...b_n$, find the length, k , of the longest sequence $C = c_1, c_2, ...c_k$ such that C is a subsequence (not necessarily contiguous) of both A and B. As an example, if

$$A = \text{d,y,n,a,m,i,c}$$
$$B = \text{p,r,o,g,r,a,m,m,i,n,g,}$$

then the *longest common subsequence* is `a,m,i` and has a length 3. Give an algorithm to solve the *longest common subsequence* problem. Your algorith should run in $O(MN)$ time.