

# CSC 373 H 1 Y — Summer 2006

University of Toronto — St. George Campus

## Lecture Summary for Week 4

This summary is not a replacement for the lecture. If you miss a class, please arrange with a friend to take note for you.

### The Longest Common Subsequence Problem cont'd

Computing the length  $L_{i,j}$  of  $S_{i,j}$ :

$$L_{0,j} = L_{i,0} = 0 \quad \text{for } 0 \leq i \leq n, 0 \leq j \leq m \quad (1)$$

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & \text{if } X[i] = Y[j] \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{otherwise} \end{cases} \quad (2)$$

#### Program to compute $L$ :

1.  $L$ :  $n \times m$  array
2. For  $j = 1$  to  $m$  do  $L[0, j] \leftarrow 0$
3. For  $i = 1$  to  $n$  do  $L[i, 0] \leftarrow 0$
4. For  $i = 1$  to  $n$  do
5.     For  $j = 1$  to  $m$  do
6.         If  $X[i] = Y[j]$  do  $L[i, j] \leftarrow L[i - 1, j - 1] + 1$
7.         Else  $L[i, j] \leftarrow \max$  of  $\{L[i - 1, j], L[i, j - 1]\}$
8.         End If
9.     End For
10. End For

#### Compute a longest common subsequence using $L$ :

The idea is to examine how  $L[i, j]$  are computed in order to decide whether to include  $X[i]$  in the common sequence.

Print-LCS( $X, Y, L, n, m$ ):

1.  $\ell \leftarrow L[n, m]$    % length of a longest common subsequence of  $X$  and  $Y$ .
2.  $S$ : array of length  $\ell$    % the output common subsequence
3.  $i \leftarrow n, j \leftarrow m$
4. While  $\ell > 0$  do
5.     If  $X[i] = Y[j]$  then
6.          $S[\ell] \leftarrow X[i]$
7.          $i \leftarrow i - 1, j \leftarrow j - 1, \ell \leftarrow \ell - 1$
8.     Elseif  $L[i, j] = L[i - 1, j]$  then
9.          $i \leftarrow i - 1$
10.    Else  $j \leftarrow j - 1$
11.    End If
12. End While

### 13. Output $S$ .

## Dynamic Programming Technique

In a recursive algorithm where the solutions to the subproblems are used repeatedly, we might want to store them in order to avoid repeated computation. In fact, the problems are solved in the bottom-up fashion: solutions to the subproblems are used in computing the solutions to bigger problems.

We may have to make a detour to another related but simpler problem (e.g. computing the length of a longest common subsequence, instead of computing a longest such common subsequence). Here are four steps in designing an algorithm using the dynamic programming technique:

1. Define an array for solving the (related) problem. (For example, the array  $L$  in the LCS problem.)
2. Give the initialization and recursive formulas for computing the elements of the array. (For example, the initial values in (1) and recursive formula in (2).)
3. Give a program for computing the array.
4. Give a program for computing the solution to our problem using the array.

## Interval Scheduling with Profits

This is a generalization of the Interval Scheduling problem that we consider in week 1. Here each interval (or request)  $i$  is associated with a profit (or weight)  $w(i)$ , and we want a compatible schedule with maximum total profit. (We obtain the Interval Scheduling Problem by assigning the profits  $w(i) = 1$  for all  $i$ .)

### Designing a Recursive Algorithm

Consider an optimal solution  $OPT$ . (Write  $w(OPT)$  for the total profit of the intervals in  $OPT$ .) For each interval  $i$ , there are two cases: either  $i \in OPT$  or  $i \notin OPT$ .

- $i \in OPT$  then  $w(OPT) = w(i) + w(OPT')$
- $i \notin OPT$  then  $w(OPT) = w(OPT'')$

(So  $w(OPT) = \max\{w(i) + w(OPT'), w(OPT'')\}$ ) where

- $OPT'$ : optimal solution for the subproblem where  $i$  and all intervals overlapping with  $i$  are removed.
- $OPT''$ : optimal solution for the subproblem without  $i$ .

An issue is to compute the intervals for the first subproblem efficiently. One way is to order the intervals in increasing order of finishing time:

$$f(1) \leq f(2) \leq \dots \leq f(n) \tag{3}$$

For each  $i$ , let  $p[i]$  be the last index  $j < i$  such that interval  $j$  does not overlap with  $i$ .

Now to solve the problem with intervals  $\{1, \dots, i\}$  as described above, the first subproblem is defined by the intervals  $\{1, \dots, p[i]\}$ , and the second subproblem is given by the intervals  $\{1, \dots, i-1\}$ .

**Computing  $p[i]$ :** Here is a simple way, which runs in time  $\mathcal{O}(n^2)$ . The linear search in the While loop below can be replaced by a binary search, and the running time will be  $\mathcal{O}(n \log(n))$ .

1.  $p[1] \leftarrow 0$
2. For  $i = 2$  to  $n$  do
3.      $j \leftarrow i - 1$
4.     While  $f(j) > s(i)$  do    $j \leftarrow j - 1$    End While
5.      $p[i] \leftarrow j$
6. End For

For the given order of the intervals as in (3), here are the steps of designing an algorithm for solving the Interval Scheduling with Profits problem.

1. Let  $M[i]$  be to maximum total profit of a schedule for intervals  $\{1, \dots, i\}$ .
2. Initial values and recurrence:

$$M[0] = 0, \quad M[i] = \max\{M[i - 1], w(i) + M[p[i]]\} \quad \text{for } 1 \leq i \leq n$$

3. Program:

1.  $M[0] \leftarrow 0$
2. For  $i = 1$  to  $n$  do
3.      $M[i] \leftarrow \max\{M[i - 1], w(i) + M[p[i]]\}$
4. End For

4. Computing an optimal solution using  $M$ :

1.  $S \leftarrow \emptyset$    % solution
2.  $i \leftarrow n$
3. While  $i \leq 1$  do
4.     If  $M[i] = M[i - 1]$  then  $i \leftarrow i - 1$
5.     Else
6.          $S \leftarrow S \cup \{i\}$
7.          $i \leftarrow p[i]$
8.     End If
9. End While
10. Return  $S$ .

### Running time:

- Sorting the intervals:  $\mathcal{O}(n \log(n))$
- Computing the array  $p$ :  $\mathcal{O}(n \log(n))$ .
- Computing  $M$ :  $\mathcal{O}(n)$ .
- Constructing an optimal solution:  $\mathcal{O}(n)$ .
- Total:  $\mathcal{O}(n \log(n))$ .