Dynamic Programming – a Quick Review

Kun-Mao Chao[1,2]

[1]Department of Computer Science and Information Engineering
[2]Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan 106
Email: kmchao@csie.ntu.edu.tw

October 5, 2005

1 Introduction

Dynamic programming is a class of solution methods for solving sequential decision problems with a compositional cost structure. It is one of the major paradigms of algorithm design in computer science. The name was given in 1957 by Richard Bellman. The word "programming" both here and in linear programming refers to a tabular method that makes a series of choices, not to writing computer code. The word "dynamic" in this context conveys the idea that choices may depend on the current state, rather than being decided ahead of time.

Typically, dynamic programming is applied to optimization problems. In such problems, there exist many possible solutions. Each solution has a value, and we wish to find a solution with the optimum value. There are two ingredients for an optimization problem to be suitable for a dynamic-programming approach. One is that it satisfies the principle of optimality, *i.e.*, each substructure is optimal. The other is that it has overlapping subproblems, otherwise a divide-and-conquer approach is the choice.

The development of a dynamic-programming algorithm has three basic components: the recurrence relation (for defining the value of an optimal solution), the tabular computation (for computing the value of an optimal

solution), and the traceback (for delivering an optimal solution). Here we introduce these basic ideas by developing dynamic-programming solutions for problems from different application areas.

2 Elementary Dynamic-Programming Algorithms

The Fibonacci numbers example is used to demonstrate how a tabular computation can avoid recomputation. The longest increasing subsequence problem and the longest common subsequence problem are all very classical and instructive for introducing dynamic-programming approaches to solving sequence-related problems [1, 2].

2.1 Fibonacci numbers

The Fibonacci numbers were first created by Leonardo Fibonacci in 1202. It is a simple series, but its applications are nearly everywhere. It has fascinated mathematicians for over 800 years. The *Fibonacci numbers* are defined by the following recurrence:

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2. \end{cases}$$

Given a positive integer n, how would you compute F_n? You might say that it can be easily solved by a straightforward recursive method based on the recurrence. That's right. But is it efficient? Take the computation of F_{10} for example. By definition, F_{10} is derived by adding up F_9 and F_8. What about the values of F_9 and F_8? Again, F_9 is derived by adding up F_8 and F_7; F_8 is derived by adding up F_7 and F_6. Working towards this direction, we'll finally reach the values of F_1 and F_0, i.e., the end of the recursive calls. By adding them up backwards, we have the value of F_{10}. It can be shown that the number of recursive calls we have to make for computing F_n is exponential in n.

Those who are ignorant of history are doomed to repeat it. A major drawback of this recursive approach is to solve many of the subproblems repeatedly. A tabular method solves every subproblem just once and then

Table 1: A tabular method can avoid recomputation.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered. Table 1 illustrates that F_n can be computed in $O(n)$ steps by a tabular computation. It should be noted that F_n can be computed in just $O(\log n)$ steps by applying matrix computation [1].

2.2 Longest increasing subsequence

Given a sequence of real numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, the longest increasing subsequence problem is to find an increasing subsequence in A whose length is maximum. Without loss of generality, we assume these numbers are distinct. Formally, given a sequence of distinct real numbers $A = \langle a_1, a_2, \ldots, a_n \rangle$, another sequence $B = \langle b_1, b_2, \ldots, b_k \rangle$ is a subsequence of A if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of A such that for all $j = 1, 2, \ldots, k$, we have $a_{i_j} = b_j$. We say the subsequence B is increasing if $b_1 < b_2 < \ldots < b_k$. The longest increasing subsequence problem is to find a maximum-length increasing subsequence of A.

For example, suppose $A = \langle 9, 2, 5, 3, 7, 11, 8, 10, 13, 6 \rangle$, both $\langle 2, 3, 7 \rangle$ and $\langle 5, 7, 10, 13 \rangle$ are increasing subsequences of A, while $\langle 9, 7, 11 \rangle$ and $\langle 3, 5, 11, 13 \rangle$ are not.

Let $L(i)$ be the length of a longest increasing subsequence ending at position i. They can be computed by the following recurrence:

$$L(i) = \begin{cases} 1 + \max_{j=0,\ldots,i-1}\{L(j) \mid a_i < a_j\} & \text{if } i > 0, \\ 0 & \text{if } i = 0. \end{cases}$$

Here we assume a_0 is a dummy element and smaller than any element in A, and $L(0)$ is equal to 0. By tabular computation, each $L(i)$ can be computed in $O(i)$ steps. Therefore, they require in total $\sum_{i=1}^{n} O(i) = O(n^2)$ steps. For each position i, we record the best previous element for current element to concatenate with. By tracing back from the element with the

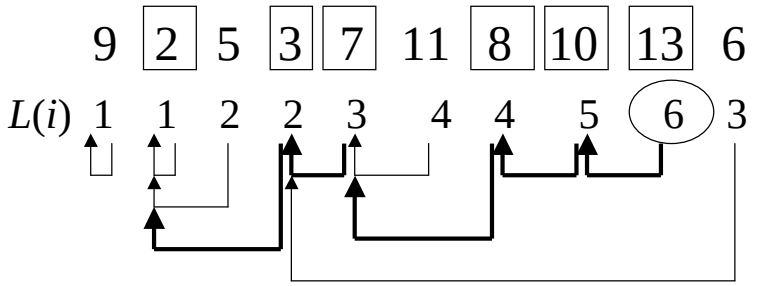

Figure 1: $A = \langle 9, 2, 5, 3, 7, 11, 8, 10, 13, 6\rangle$. The reported longest increasing subsequence is $\langle 2, 3, 7, 8, 10, 13\rangle$. Notice $\langle 2, 5, 7, 8, 10, 13\rangle$ is an alternative longest increasing subsequence.

largest L value, we derive a longest increasing subsequence. Figure 1 gives an example.

In the following, we briefly describe a more efficient dynamic-programming algorithm for delivering a longest increasing subsequence [2]. Let $BestEnd(k)$ denote the smallest ending element of all possible increasing subsequences of length k ending before current position i. How do we update $BestEnd(k)$ when we consider a_i? Notice first the elements in $BestEnd$ are in increasing order. In fact, a_i will affect only one entry in $BestEnd$. If a_i is larger than all the elements in $BestEnd$, then we can concatenate a_i to the longest increasing subsequence computed so far. That is, one more entry is added to the end of $BestEnd$. A backtracking pointer is recorded by pointing to the previous last element of $BestEnd$. Otherwise, let $BestEnd(k')$ be the smallest element that is larger than a_i. We replace $BestEnd(k')$ by a_i because now we have a smaller ending element of an increasing subsequence of length k'. Since $BestEnd$ is a sorted array, the above process can be done by a binary search. For each position i, it takes $O(\log i)$ time to determine the appropriate entry to be updated by a_i. Therefore, in total we have an $O(n \log n)$-time algorithm for the longest increasing subsequence problem. Figure 2 illustrates the idea.

2.3 Longest common subsequence

A subsequence of a sequence S is obtained by deleting zero or more symbols from S. For example, these are all subsequences of the sequence *president*: *pred, sdn, predent.*

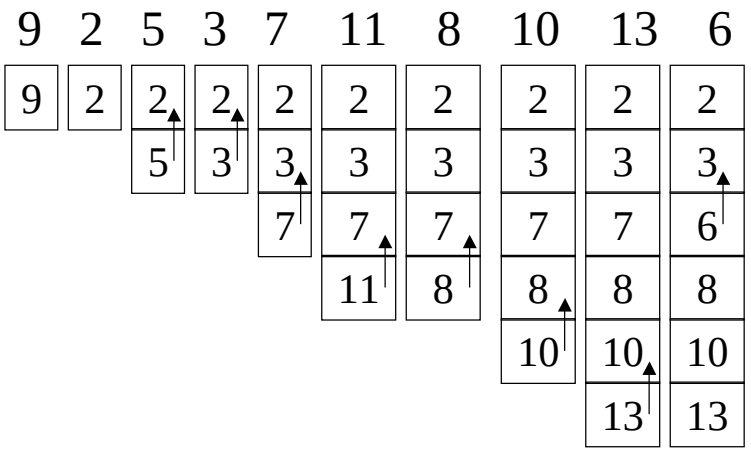

Figure 2: $A = \langle 9, 2, 5, 3, 7, 11, 8, 10, 13, 6\rangle$. Notice that each a_i modifies only one entry in *BestEnd*. We also record a pointer to its previous entry for backtracking. It should be noted, however, that the final *BestEnd* ($\langle 2, 3, 6, 8, 10, 13\rangle$) is not a longest increasing subsequence.

Given two sequences, the longest common subsequence problem is to find a subsequence that is common to both sequences and its length is maximized. For example, given two sequences *president* and *providence*, the subsequence *prd* is a common subsequence of them, while the subsequence *prv* is not. We are interested in finding a maximum-length common subsequence between two sequences.

We are given two sequences $A = \langle a_1, a_2, \ldots, a_m\rangle$, and $B = \langle b_1, b_2, \ldots, b_n\rangle$. Let $len(i, j)$ denote the length of the longest common sequence between $\langle a_1, a_2, \ldots, a_i\rangle$, and $\langle b_1, b_2, \ldots, b_j\rangle$. They can be computed by the following recurrence:

$$len(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ len(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } a_i = b_j, \\ \max\{len(i, j-1), len(i-1, j)\} & \text{otherwise.} \end{cases}$$

In other words, if any sequence of the two is an empty sequence, the length of the longest common subsequence is of course zero. If a_i matches with b_j, the longest common subsequence between $\langle a_1, a_2, \ldots, a_i\rangle$, and $\langle b_1, b_2, \ldots, b_j\rangle$ is the longest common subsequence of $\langle a_1, a_2, \ldots, a_{i-1}\rangle$, and $\langle b_1, b_2, \ldots, b_{j-1}\rangle$ followed by a_i. Therefore, in this case $len(i, j) = len(i-1, j-1) + 1$. Otherwise, a_i doesn't match with b_j. The length of the longest common subsequence is thus the larger of $len(i, j-1)$ and $len(i-1, j)$.

Figure 3 gives the pseudo-code for computing $len(i, j)$. The array $prev(i, j)$ is used to record the backtracking information. The total running time is $O(mn)$.

```
procedure LCS-Length(A, B)
1.  for i ← 0 to m do  len(i,0) = 0
2.  for j ← 1 to n do  len(0,j) = 0
3.  for i ← 1 to m do
4.      for j ← 1 to n do
5.          if a_i = b_j then [ len(i, j) = len(i−1, j−1) + 1
                              [ prev(i, j) =" ↖ "
6.          else if len(i−1, j) ≥ len(i, j−1)
7.              then [ len(i, j) = len(i−1, j)
                     [ prev(i, j) =" ↑ "
8.              else [ len(i, j) = len(i, j−1)
                     [ prev(i, j) =" ← "
9.  return len and prev
```

Figure 3: Computing $len(i, j)$

Figure 4 illustrates the tabular computation. The length of the longest common subsequence of the sequences *president* and *providence* is 6.

i		j	0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0			0	0	0	0	0	0	0	0	0	0	0
1	p		0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2	r		0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3	e		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	s		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5	i		0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6	d		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7	e		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8	n		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9	t		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Figure 4: The tabular computation.

Figure 5 lists the pseudo-code for delivering the longest common subsequence. We backtrack recursively according the direction of the arrow. Only when a diagonal arrow is encountered, we append the current matched letter to the end. It takes $O(m + n)$ time to do the backtracking.

Figure 6 illustrates the backtracking process. Recall that we will output a matched letter when a diagonal arrow is reached.

```
procedure Output-LCS(A, prev, i, j)
1  if i = 0 or j = 0 then return
2  if prev(i, j)="↖" then [ Output − LCS(A, prev, i − 1, j − 1)
                          [ print a_i
3  else if prev(i, j)="↑" then Output-LCS(A, prev, i-1, j)
4  else Output-LCS(A, prev, i, j-1)
```

Figure 5: Traceback.

i		j	0	1 p	2 r	3 o	4 v	5 i	6 d	7 e	8 n	9 c	10 e
0			0	0	0	0	0	0	0	0	0	0	0
1	p		0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2	r		0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3	e		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4	s		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5	i		0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6	d		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7	e		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8	n		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9	t		0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Figure 6: The longest common subsequence is *priden*. The shaded area is the trace of backtracking.

Acknowledgements

Kun-Mao Chao was supported in part by NSC grants 94-2213-E-002-018 and 94-2213-E-002-091 from the National Science Council, Taiwan.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein *Introduction to Algorithms*, the MIT Press, Cambridge, Massachusetts, 1999.

[2] U. Manber *Introduction to Algorithms*, Addison Wesley, 1989.