Class 11: Dynamic Programming

Definitions

Dynamic programming is similar to the divide-and-conquer method in that it solves problems by combining solutions of subproblems. ("Programming" refers to a tabular method, not writing computer code.) Unlike the divide-and-conquer approach, dynamic programming is applicable when the subproblems are not independent. A dynamic-programming algorithm solves each problem just once and saves its answer in a table to be used by other subproblems.

It is typically applied to combinatorial optimization problems. In such problems there can be many solutions, from which one is an optimal one that results in the best value. The development of a dynamic-programming algorithm can be broken into the following steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Longest common subsequence

In biological applications, the DNAs of two or more organisms are often compared. A strand of DNA consists of a string of molecules called *bases*: adenine(A), guanine(G), cytosine(C), and thymine(T). A strand of DNA can then be represented as a string over the finite set {A,G,C,T}. For example: ACCGTTTTAAA. To measure the similarity of two strands S1 and S2 we find a third strand S3 in which the bases in S3 appear in each of S1 and S2; these bases must appear in the same order, but not necessarily consecutively. For example:

```
S1: AGCTAGCT
S2: TCGAGATC
S3: TAGT
```

We formalize this notion by formulating it as the longest-common-subsequence (LCS) problem. A subsequence of a given sequence is the given sequence with 0 or more elements left out. Formally given a sequence X=<x1, ... , x_m>, another sequence Z=<z1, ... , z_k> is subsequence of X if there exists a strictly increasing sequence <i1, ... , i_k> of indices of X such that for all j=1,2,...,k, we have $x_{i\_j} = z\_j$.

Given two sequences X and Y, we say that a sequence Z is a *common subsequence* of X and Y if Z is a subsequence of both X and Y.

In the longest-common-subsequence problem, we are given two sequences X and Y and wish to find a maximum-length common subsequence. We show below that this problem can be solved efficiently using dynamic programming.

Step 1: Characterizing a longest common subsequence

A brute-force approach:

- Find every subsequence of X starting with the longest possible (size of Y)
- Check if it is also subsequence of Y and return if it is.

In the worst case the above algorithm will check every subsequence of X, which corresponds to a subset of indices {1,...,m} that leads to 2^m subsequences of X. This makes the running time exponential and the problem impractical.

However, the LCS problem has an optimal-substructure property. For a sequence X = <x1, x2, ... , x_m>, define the ith ***prefix*** of X as X_i = <x1, x2, ... , x_i>.

Theorem 15.1 (Optimal substructure of LCS)

Let X = <x1, ... , x_m> and Y = <y1, ... , y_n> be sequences, and let Z = <z1, ... , z_k> be any LCS of X and Y.

1. If x_m = y_n, then z_k = x_m = y_n, and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If x_m ≠ y_n, then z_k ≠ x_m implies that Z is an LCS of X_{m-1} and Y.
3. If x_m ≠ y_n, then z_k ≠ y_n implies that Z is an LCS of X and Y_{n-1}.

Proof. (1) If z_k ≠ x_m, then we can append x_m to Z and obtain a (k+1)-length common subsequence, which is a contradiction that Z is LCS. This also means that Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is also LCS. To prove by contradiction, assume there is subsequence W of X_{m-1} and Y_{n-1} with length > (k-1). But then, appending z_k to it would produce a subsequence of length > k, which is contradiction.

(2) If z_k ≠ x_m then Z is common subsequence of X_{m-1} and Y. If there were common subsequence W greater than Z, it would also be a common subsequence of X and Y, which is a contradiction.

(3) The proof is symmetric to (2). ÿ

The characterization of this theorem shows that an LCS of two sequences contains within it an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal substructure property.

Step 2: A recursive solution

Theorem 15.1 implies that a solution to the LCS problem for X = <x1,...,x_m> and Y=<y1,...,y_m> depends on one or two subproblems:

- If x_m = y_n, we must find LCS on X_{m-1} and Y_{n-1};
- If x_m <> y_n we must choose the greater LCS of the two subroplems:
  - X_{m-1} and Y
  - X and Y_{n-1}

This shows the overlapping-subproblem property in the LCS problem: both (X_{m-1}, Y} and (X and Y_{n-1}) subproblems depend on a solution to (X_{m-1},Y_{n-1}) subproblem.

Define c[i,j] to be the length of an LCS of the sequences X_i and Y_j. The optimal substructure of the

LCS problem gives the following recursive formula:

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x\_i = y\_j \\ \max(c[i-1,j],c[i,j-1]) & \text{if } x\_i \neq y\_j \end{cases}$$

Based on this equation we can easily construct a recursive algorithm to compute the length of LCS. However, the worst-case running time of such algorithm would be exponential. Assume N=m+n and for simplicity (without the loss of generality): n=m. The worst case occurs when x_i <> y_j for all i,j. In this case the running time can be calculated as follows:

$T(N) = T(N-1) + T(N-1) + c$

When building a recursion tree for the above equation we observe that it yields to c2^N cost at the last level, which shows that the algorithm runs in exponential time.

Step 3: Computing the length of LCS

The procedure below implements a bottom-up approach to computing the length of LCS. It stores the c[i,j] values in a table c[0..m,0..n] whose entries are computed row by row. It also maintains the table b[1..m,1..n], in which each entry corresponds to the optimal subproblem solution chosen when computing c[i,j].

```
LCS-Length(X,Y)
1     m = length[X]
2     n = length[Y]
3     for i = 1 to m
4           c[i,0] = 0
5     for i = 0 to n
6           c[0,i] = 0
7     for i = 1 to m
8           for j = 1 to n
9                 if x_i = y_i
10                      c[i,j] = c[i-1,j-1] + 1
11                      b[i,j] = UPLEFT
12                else
13                      if c[i-1,j] >= c[i,j-1]
14                            c[i,j] = c[i-1,j]
15                            b[i,j] = UP
16                      else
17                            c[i,j] = c[i,j-1]
18                            b[i,j] = LEFT
19    return b
```

The running time of this procedure is O(mn) as steps 9-18 are repeated mn times and each time they take a constant time to compute.

Step 4: Constructing LCS

The b table returned by the algorithm above can be used to quickly construct an LCS for X and Y. We simply begin at b[m,n] and trace through the table following arrows. The following procedures print an

LCS on X and Y in the proper, forward order.

```
Print-LCS(b,X,i,j)
1     if i=0 or j=0
2           return
3     if b[i,j] = UPLEFT
4           Print-LCS(b,X,i-1,j-1)
5           print x[i]
6     else
7           if b[i,j] = UP
8                 Print-LCS(b,X,i-1,j)
9           else
10                Print-LCS(b,X,i,j-1)

LCS(X,Y)
1     b = LCS-Length(X,Y)
2     Print-LCS(b, X, length[X], length[Y])
```

The running time of Print-LCS algorithm is O(m+n) since at each step of recursion i or j are decremented by at least 1. Hence the running time of LCS algorithm is O(m+n) + O(mn) ~ O(mn)