

Dynamic Programming 3

Chapter 7, Algorithm Design

Hantao Zhang
<http://www.cs.uiowa.edu/~hzhang/c31>

Properties of a typical problem that can be solved with dynamic programming

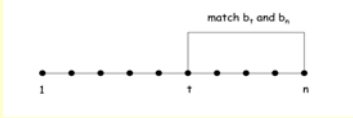
- ◆ **Simple Subproblems**
 - » We should be able to break the original problem to smaller subproblems that have the same structure
- ◆ **Optimal Substructure of the problems**
 - » The solution to the problem must be a composition of subproblem solutions
- ◆ **Subproblem Overlap**
 - » Optimal subproblems to unrelated problems can contain subproblems in common

RNA Secondary Structure

- ◆ Given a chain of basic units of DNA, { A, C, G, T }, how to fold the chain so that we have a maximal number of pairs (A – T) and (C – G)? The folded structure is called the RNA secondary structure.

Subproblems: First Try

- ◆ Let $OPT(i)$ be the maximal base pairs of the secondary structure from the substring $b_1b_2\dots b_i$
- ◆ $OPT(n)$ is the solution we wanted.
- ◆ If b_t matches b_n , we have two substrings:

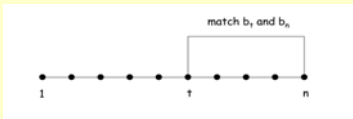


- ◆ $b_1b_2\dots b_{t-1}$ and $b_{t+1}\dots b_{n-1}$.
- ◆ The solution of the first substring is $OPT(t-1)$.
- ◆ How to get the solution of the second string?

7

Subproblems: Second Try

- ◆ Let $OPT(i, j)$ be the maximal base pairs of the secondary structure from the substring $b_ib_{i+1}\dots b_j$
- ◆ $OPT(1, n)$ is the solution we wanted.
- ◆ If b_t matches b_n , we have two substrings:



- ◆ $b_1b_2\dots b_{t-1}$ and $b_{t+1}\dots b_{n-1}$.
- ◆ The solution of the first substring is $OPT(1, t-1)$.
- ◆ The solution of the second substring is $OPT(t+1, n-1)$.

8

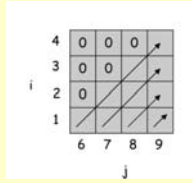
Optimal Structure of Subproblems

- ◆ Let $OPT(i, j)$ be the maximal base pairs of the secondary structure from the substring $b_ib_{i+1}\dots b_j$
- ◆ $OPT(1, n)$ is the solution we wanted.
- ◆ If b_j matches b_t ,
 - » $OPT(i, j) = 1 + OPT(i, t-1) + OPT(t+1, j-1)$
- ◆ If b_j matches no bases
 - » $OPT(i, j) = OPT(i, j-1)$
- ◆ We choose the larger one for $OPT(i, j)$.
- ◆ b_j matches b_t means $t+4 < j$, and (b_j, b_t) is a base pair.

9

Algorithm from Recursive Relation

- Let $OPT(i, j)$ be the maximal base pairs of the secondary structure from the substring $b_i b_{i+1} \dots b_j$
- Max-Base-Pair(n)**
 array $M[1..n, 1..n]$
 For any $i \leq j \leq i+4$, $M[i, j] := 0$,
 For $k = 5, 6, \dots, n-1$
 For $i = 1, 2, \dots, n-k$
 $j := i+k$; $x := 0$
 For $t = i, i+1, \dots, j-5$
 If (b_j, b_t) is a base pair
 If $(i > t - 6)$ $x := \max(x, 1 + M[t+1, j-1])$
 Else $x := \max(x, 1 + M[i, t-1] + M[t+1, j-1])$
 $M[i, j] := \max(x, M[i, j-1])$
 Return $M[1, n]$



10

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either **top-down** with caching or **bottom-up** in a table.
4. Construct an optimal solution from computed values.

11

Longest Common Subsequence (LCS)

- ♦ **Problem:** Given 2 sequences, $X = \langle x_1, \dots, x_m \rangle$ and $Y = \langle y_1, \dots, y_n \rangle$, find a common subsequence whose length is maximum.



Subsequence **need not be consecutive**, but **must be in order**.

12

Longest Common Subsequence (LCS)

- ◆ Given two sequences $x[1..m]$ and $y[1..n]$, find the longest subsequence which occurs in both
- ◆ Ex: $x = \text{“A B C B D A B”}$, $y = \text{“B D C A B A”}$
- ◆ “B C” and “A A” are both subsequences of both
What is the LCS?
- ◆ Brute-force algorithm: For every subsequence of x , check if it's a subsequence of y .
 - » *How many subsequences of x are there?*
 - » *What will be the running time of the brute-force algorithm?*

13

LCS Algorithm

- ◆ Brute-force algorithm: 2^m subsequences of x to check against n elements of y : $O(n 2^m)$
- ◆ We can do better: For now, let's only worry about the problem of finding the *length* of LCS
 - » When finished we will see how to backtrack from this solution back to the actual LCS
- ◆ Using “Divide One Less”

14

Finding LCS Length

- ◆ Define $c[i,j]$ to be the length of the LCS of $X_i = x[1..i]$ and $Y_j = y[1..j]$
 - » *What is the length of LCS of x and y ?*

- ◆ Theorem:

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- ◆ *What is this really saying?*

15

LCS Length Algorithm 1

LCS-Length(m, n)

1. If (m = 0 or n = 0) **return** 0
2. If (A[m] = A[n])
 return LCS-Length(m-1, n-1)+1
3. a := LCS-Length(m, n-1);
4. b := LCS-Length(m-1, n);
5. **return** max(a, b);

What's the complexity?

16

LCS Length Algorithm 1

LCS-Length(m, n)

For i := 0 to m For j := 0 to n do C[i,j] := -1
return LCS-Length2(m, n)

LCS-Length2(i, j)

1. If (i = 0 or j = 0) **return** 0
2. If C[i,j] != -1 **return** C[i, j];
3. If (A[i] = A[j]) C[i,j] := LCS-Length(i-1, j-1)+1
4. else a := LCS-Length2(i, j-1);
5. b := LCS-Length2(i-1, j);
6. C[i,j] := max(a, b);
7. **return** C[i,j]

What's the complexity?

17

Properties of a typical problem that can be solved with dynamic programming

- ◆ **Simple Subproblems**
 - » We should be able to break the original problem to smaller subproblems that have the same structure
- ◆ **Optimal Substructure of the problems**
 - » The solution to the problem must be a composition of subproblem solutions
- ◆ **Subproblem Overlap**
 - » Optimal subproblems to unrelated problems can contain subproblems in common

18

LCS Length Algorithm

LCS-Length(X, Y)

1. $m := \text{length}(X)$ // get the # of symbols in X
2. $n := \text{length}(Y)$ // get the # of symbols in Y
3. for $i := 1$ to m $c[i,0] := 0$ // special case: Y_0
4. for $j := 1$ to n $c[0,j] := 0$ // special case: X_0
5. for $i := 1$ to m // for all X_i
6. for $j := 1$ to n // for all Y_j
7. if ($X_i = Y_j$)
8. $c[i,j] := c[i-1,j-1] + 1$
9. else $c[i,j] := \max(c[i-1,j], c[i,j-1])$
10. return c

19

LCS Example

We'll see how LCS algorithm works on the following example:

- ◆ X = ABCB
- ◆ Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB
X = A **B** C B
Y = B D C A **B**

20

LCS Example (0)

		ABCB BDCAB						
		j	0	1	2	3	4	5
		Y _j	B	D	C	A	B	
i	X _i							
0								
1	A							
2	B							
3	C							
4	B							

X = ABCB; $m = |X| = 4$
Y = BDCAB; $n = |Y| = 5$
Allocate array c[5,4]

21

LCS Example (1) ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0					
2	B	0					
3	C	0					
4	B	0					

for $i = 1$ to m $c[i,0] = 0$
for $j = 1$ to n $c[0,j] = 0$

22

LCS Example (2) ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

23

LCS Example (3) ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0		
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

24

LCS Example (4) ABCB
BDCAB

	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
i	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

25

LCS Example (5) ABCB
BDCAB

	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
i	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0					
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

26

LCS Example (6) ABCB
BDCAB

	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
i	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1				
3	C	0					
4	B	0					

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

27

LCS Example (7)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Y _j	B	D	C	A	B	
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

if (X_i == Y_j)
 c[i,j] = c[i-1,j-1] + 1
 else c[i,j] = max(c[i-1,j], c[i,j-1])

28

LCS Example (8)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Y _j	B	D	C	A	B	
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

if (X_i == Y_j)
 c[i,j] = c[i-1,j-1] + 1
 else c[i,j] = max(c[i-1,j], c[i,j-1])

29

LCS Example (10)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Y _j	B	D	C	A	B	
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1			
4	B	0					

if (X_i == Y_j)
 c[i,j] = c[i-1,j-1] + 1
 else c[i,j] = max(c[i-1,j], c[i,j-1])

30

LCS Example (11)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2		
4	B	0					

if (X_i == Y_j)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

31

LCS Example (12)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

if (X_i == Y_j)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

32

LCS Example (13)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Y _j		B	D	C	A	B
0	X _i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

if (X_i == Y_j)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

33

LCS Example (14)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	1	1
2	B	0	1	1	1	2
3	C	0	1	1	2	2
4	B	0	1	1	2	2

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

34

LCS Example (15)

ABCB
BDCAB

j	0	1	2	3	4	5
i	Yj	B	D	C	A	B
0	Xi	0	0	0	0	0
1	A	0	0	0	1	1
2	B	0	1	1	1	2
3	C	0	1	1	2	2
4	B	0	1	1	2	3

if ($X_i == Y_j$)
 $c[i,j] = c[i-1,j-1] + 1$
 else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

35

LCS Algorithm Running Time

- ♦ LCS algorithm calculates the values of each entry of the array $c[m,n]$
- ♦ So what is the running time?

$O(m*n)$

since each $c[i,j]$ is calculated in constant time, and there are $m*n$ elements in the array

36

How to Find Actual LCS

- ◆ So far, we have just found the *length* of LCS, but not LCS itself.
- ◆ We want to modify this algorithm to make it output LCS of X and Y

Each $c[i,j]$ depends on $c[i-1,j]$ and $c[i,j-1]$
or $c[i-1, j-1]$

For each $c[i,j]$ we can say how it was acquired:

2	2
2	3

For example, here
 $c[i,j] = c[i-1,j-1] + 1 = 2+1=3$

37

How to find actual LCS

- ◆ Remember that

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$
- ◆ So we can start from $c[m,n]$ and go backwards
- ◆ Whenever $c[i,j] = c[i-1,j-1] + 1$, remember $x[i]$ (because $x[i]$ is a part of LCS)
- ◆ When $i=0$ or $j=0$ (i.e. we reached the beginning), output remembered letters in reverse order

38

Finding LCS

		j					
		0	1	2	3	4	5
		Yj	B	D	C	A	B
i	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

39

Finding LCS (2)

i \ j	0	1	2	3	4	5	
0	Xi	Yj	B	D	C	A	B
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): **B C B**

LCS (straight order): **B C B**
 (this string turned out to be a palindrome)

40

Summary: Dynamic programming

- ◆ DP is a method for solving certain kind of problems
- ◆ DP can be applied when the solution of a problem includes solutions to subproblems
- ◆ We need to find a recursive formula for the solution
- ◆ We can recursively solve subproblems, starting from the trivial case, and save their solutions in memory
- ◆ In the end we'll get the solution of the whole problem

41
