ECM5605(5075) S'20 Algorithms

Lecture 04

Dynamic Programming (1): *Longest Common Sequence*

Review Problems

- What is Order Statistics?
  - How to design an efficient Algorithm?
  - Worst-case complexity? Average-case complexity?
  - How to analyze the average-case of RAND-SELECT?
  - What's the key idea to guarantee O(n)? Hint: BFPRT-Select
  - Why groups of 5? Can we use 3?
- Back to Quick Sort
  - How to prove the average-case complexity?
  - Can guarantee O($n$lg$n$) if using BFPRT-Select?

From Divide-and-Conquer …

- Essence: solve problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms
  - Partition a problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
  - What’s the potential problem??

    ⇨ *Inefficient to solve repeated subproblems*
- Therefore, if many independent subproblems are overlapping

  ⇨ only need to each subproblem just once.

  ⇨ **Dynamic programming**

Birth of Dynamic Programming

- Richard Bellman, 1952
  - multistage stochastic decision processes
- "Dynamic Programming" so named because
  - Originally associated with movements (trading) between time and space, thus "dynamic"
  - "programming" refers to the process of formulating the constraints of a problem ⇨ by analogy to "linear programming" and other forms of optimization
- Can be also used in deterministic problems ⇨ (**recursive**) **optimization** problems

Dynamic Programming (DP)

- Typically apply to ***optimization*** problems:
  - Find ***a (not "the")*** solution with the optimal (maximum / minimum) value
- 4 steps in developing a DP algorithm

1. ***Characterize*** the structure of an optimal solution.

2. ***Recursively define*** the ***formula*** of an optimal solution.

3. ***Compute*** the value of an optimal solution either ***bottom-up*** in a table (or top-down with caching)

4. ***Construct*** an ***optimal solution*** from computed information (this step can be omitted if only the value is required)

When to Use DP

- Characteristics of DP problems:
  1. problem can be divided into ***stages (subproblems)***
  2. each stage has one/more ***states (substructures)***
  3. you make a ***decision*** at each stage
  4. the decision you make affects the state for the next stage
  5. there is a ***recursive relationship*** between the value of the decision at the stage and the previously ***found optima*** (***principle of optimality***)
- **Hopeless configurations:** for an n-element set
  - # of permutations: $n!$
  - # of subsets: 2^n

Longest Common Subsequence

- Problem Formulation:

  Given 2 sequences, $\boldsymbol{X} = \langle x_1,\ldots,x_m \rangle$ and $\boldsymbol{Y} = \langle y_1,\ldots,y_n \rangle$, find a common subsequence whose length is maximum.

- Example: Subsequence need not be consecutive, but must be in order.

***X*:**	springtime	ncaa tournament	basketball
***Y*:**	printing	north carolina	krzyzewski
***LCS*:**	printi	ncarna	ke

Naïve Algorithm and Analysis

- Brute-force LCS algorithm:

  For every subsequence of *X*, check whether it's a subsequence of *Y*
- Analysis:
  - Each subsequence takes O(*n*) time to check: scan *Y* for first letter, for second, and so on.
  - 2^m subsequences of *X* to check.
  - Worst-case runtime: O($n2^m$) exponential time
- Read code for the naïve version and watch the number of comparisons

Towards a Better Algorithm

- Simplification:
  1. Look at the ***length*** of a longest-common subsequence
     - Denote the length for *a* sequence of ***S*** by |***S***|
  2. Extend the algorithm to find the LCS itself

- Strategy: consider ***prefixes*** of ***X*** and ***Y***
  - Define $c[i,j] = |LCS(X[1..i], Y[1..j])|$
  - Then $c[m,n] = |LCS(X,Y)|$

Step 1: Optimal Substructure of LCS

Theorem

Let $Z = \langle z_1, . . . , z_k\rangle$ be any LCS of X and Y .

1 If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k\text{-}1}$ is an LCS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$.

2 If $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of $X_{m\text{-}1}$ and Y .

3 or $z_k \neq y_n$ and Z is an LCS of X and $Y_{n\text{-}1}$.

Proof: (case 1: if $x_m = y_n$)

Any sequence ***Z'*** that does not end in $x_m = y_n$ can be made longer by adding $x_m = y_n$ to the end. Therefore,

1) longest common subsequence (LCS) Z must end in $x_m = y_n$.
2) $Z_{k\text{-}1}$ is a common subsequence of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, and
3) there is no *longer* CS of $X_{m\text{-}1}$ and $Y_{n\text{-}1}$, or $Z_{k\text{-}1}$ would not be an LCS.

Step 1: Optimal Substructure of LCS (cont'd)

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ 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 $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y.

3 or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}.

Proof: (case 2: $x_m \neq y_n$, and $z_k \neq x_m$)

Since Z does not end in x_m,

1) Z is a common subsequence of X_{m-1} and Y, and

2) there is no ***longer*** CS of X_{m-1} and Y, or Z would not be an LCS

Step 2: Recursive Formula for Solution

Theorem

Let $Z = \langle z_1, \ldots, z_k \rangle$ 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 $x_m \neq y_n$, then either $z_k \neq x_m$ and Z is an LCS of X_{m-1} and Y

3 or $z_k \neq y_n$ and Z is an LCS of X and Y_{n-1}

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

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**

LCS Example (I)

i	j	0	1	2	3	4	5
		Y_j	**B**	**D**	**C**	**A**	**B**
0	X_i						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; $m = |X| = 4$;

Y = BDCAB; $n = |Y| = 5$;

⇨Allocate 5x6 Matrix ***C***

LCS Example (1)

	j	0	1	2	3	4	5
i		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

ABCB
BDCAB

LCS Example (2)

		j	0	1	2	3	4	5
i			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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (3)

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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (4)

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (5)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (6)

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	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (7)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (8)

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

if ($x_i == y_j$) $c[i,j] = c[i-1,j-1] + 1$

else $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (10)

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

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

ABCB
BDCAB

LCS Example (11)

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

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

ABCB
BDCAB

LCS Example (12)

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	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 $c[i,j] = \max(c[i-1,j], c[i,j-1])$

ABCB
BDCAB

LCS Example (13)

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

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

ABCB
BDCAB

LCS Example (14)

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

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

ABCB
BDCAB

LCS Example (15)

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**	**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**

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

LCS Algorithm Runtime

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the runtime to compute ***C***?

$$\mathbf{O}(m \times n)$$

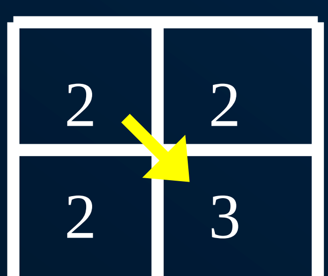
since each c[i,j] is calculated in constant time, and there are $m \times n$ elements in the array

Step 4: Construct a LCS

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

Each *c[i,j]* depends on either *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:



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

Step 4: Construct a LCS (cont'd)

- Remember that

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

- 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

Example of Constructing a LCS

	j	0	1	2	3	4	5
i		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**	**1**	**2**	**2**	**3**

Example of Constructing a LCS (cont'd)

	j	0	1	2	3	4	5
i		yj	**B**	**D**	**C**	**A**	**B**
0	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**

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

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

(this string turned out to be a palindrome)

Analysis DP on LCS Problem

	A	B	C	B	D	A	B
B	0	1	1	1	1	1	1
D	0	1	1	1	2	2	2
C	0	1	2	2	2	2	2
A	1	1	2	2	2	3	3
B	1	2	2	3	3	3	4
A	1	2	2	3	3	4	4

LCS_1 = BCBA

LCS_2 = BDAB

- Space complexity: $O(m \times n)$
- Time complexity:
  - Computing the table: time = $O(m \times n)$
  - Constructing an LCS: time = $O(m+n)$

Optimizing Space in LCS

- Recap memory usage

	j	0	1	2	3	4	5
i		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**	**1**	**2**	**2**	**3**

  – Do we really need a $m \times n$ table to store the LCS values? Any improvement?

  – Implement your space-optimized version

Summary

- What are common/different points between Divide-and-Conquer and DP?
- 2 Hallmarks of dynamic programming:
  - ***Optimal substructure***: an optimal solution to a problem (instance) contains optimal solutions to subproblems
  - ***Overlapping subproblems***: a recursive solution contains a “small” number of distinct subproblems repeated many times
- What is longest common sequence problem?
  - How to use DP to solve this problem
  - Space complexity? Time complexity??

Ex: Longest Increasing Subsequence

- The **Longest Increasing Subsequence (LIS)** problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.
- For example, length of LIS for {10, 22, 9, 33, 21, 50, 41, 60, 80 } is 6
  - LIS = {10, 22, 33, 50, 60, 80}

DP for LIS

- Let arr be the input array and $L(i)$ be the length of the LIS ending at index i
  - ⇨ $arr[i]$ is the last element of the LIS
- Then, $L(i)$ can be recursively written as:

$$L(i) = \begin{cases} 1 + \max\big(L(j)\big), \text{if } 0 < j < i \text{ and } arr[j] < arr[i] \\ 1, \text{ if no such } j \text{ exists} \end{cases}$$

- Complexity is ???
- Refer to LIS_v1.cpp and develop your DP solution in C/C++