NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 8, Lecture 3

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Longest common subword

* Given two strings, find the (length of the) longest common subword
  * “secret”, “secretary” — “secret”, length 6
  * “bisect”, “trisect” — “isect”, length 5
  * “bisect”, “secret” — “sec”, length 3
  * “director”, “secretary” —“ec”, “re”, length 2

More formally ...

- Two strings $u = a_0a_1...a_{m-1}$, $v = b_0b_1...b_{n-1}$
- If $a_ia_{i+1}...a_{i+k-1} = b_jb_{j+1}...b_{j+k-1}$ for some i and j, u and v have a common subword of length k
- Aim: Find the length of the longest common subword of u and v

Brute force

- $u = a_0a_1 \ldots a_{m-1}$ and $v = b_0b_1 \ldots b_{n-1}$
- Try every pair of starting positions i in u, j in v
  - Match (a_i, b_i), (a_{i+1}, b_{i+1}),… as far as possible
  - Keep track of the length of the longest match
- Assuming $m > n$, this is $O(mn^2)$
  - mn pairs of positions
  - From each starting point, scan can be $O(n)$

Inductive structure

- $a_i a_{i+1} \ldots a_{i+k-1} = b_j b_{j+1} \ldots b_{j+k-1}$ is a common subword of length k at (i,j) iff
  - $a_i = b_j$ and
  - $a_{i+1} \ldots a_{i+k-1} = b_{j+1} \ldots b_{j+k-1}$ is a common subword of length $k-1$ at $(i+1,j+1)$
- LCW(i,j): length of the longest common subword starting at a_i and b_j
  - If $a_i \neq b_j$, LCW(i,j) is 0, otherwise 1+LCW(i+1,j+1)
  - Boundary condition: when we have reached the end of one of the words

Inductive structure

- Consider positions 0 to m in u, 0 to n in v
  - m, n means we have reached the end of the word
- $LCW(m,j) = 0$ for all j
- $LCW(i,n) = 0$ for all i
- $LCW(i,j) = 0$, if $a_i \neq b_j$,

  $1 + LCW(i+1,j+1)$, if $a_i = b_j$

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							0
1	i							0
2	s							0
3	e							0
4	c							0
5	t							0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					0	0	0
1	i					0	0	0
2	s					0	0	0
3	e					1	0	0
4	c					0	0	0
5	t					0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				0	0	0	0
1	i				0	0	0	0
2	s				0	0	0	0
3	e				0	1	0	0
4	c				0	0	0	0
5	t				0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b			0	0	0	0	0
1	i			0	0	0	0	0
2	s			0	0	0	0	0
3	e			0	0	1	0	0
4	c			1	0	0	0	0
5	t			0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b		0	0	0	0	0	0
1	i		0	0	0	0	0	0
2	s		0	0	0	0	0	0
3	e		2	0	0	1	0	0
4	c		0	1	0	0	0	0
5	t		0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCW(i,j) depends on LCW(i+1,j+1)
- Last row and column have no dependencies
- Start at bottom right corner and fill by row or by column

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Reading off the solution

- Find (i,j) with largest entry
  - LCW(2,0) = 3
- Read off the actual subword diagonally

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Reading off the solution

- Find (i,j) with largest entry
  - LCW(2,0) = 3
- Read off the actual subword diagonally

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

LCW(u,v), DP

```
def LCW(u,v): # u[0..m-1], v[0..n-1]
  for r in range(len(u)+1):
    LCW[r][len(v)+1] = 0 # r for row
  for c in range(len(v)+1):
    LCW[len(u)+1][c] = 0 # c for col
  maxLCW = 0
  for c in range(len(v)+1,-1,-1):
    for r in range(len(u)+1,-1,-1):
      if u[r] == v[c]:
        LCW[r][c] = 1 + LCW[r+1][c+1]
      else:
        LCW[r][c] = 0
      if LCW[r][c] > maxLCW:
        maxLCW = LCW[r][c]
  return(maxLCW)
```

Complexity

- Recall that the brute force approach was $O(mn^2)$
- The inductive solution is $O(mn)$ if we use dynamic programming (or memoization)
  - Need to fill an $O(mn)$ size table
  - Each table entry takes constant time to compute

Longest common subsequence

- Subsequence: can drop some letters in between
- Given two strings, find the (length of the) longest common subsequence
  - “secret”, “secretary” — “secret”, length 6
  - “bisect”, “trisect” — “isect”, length 5
  - “bisect”, “secret” — “sect”, length 4
  - “director”, “secretary” —“ectr”, “retr”, length 4

LCS

- LCS is longest path we can find between non-zero LCW entries, moving right and down

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	0	0	0	0	0	0	0
1	i	0	0	0	0	0	0	0
2	s	3	0	0	0	0	0	0
3	e	0	2	0	0	1	0	0
4	c	0	0	1	0	0	0	0
5	t	0	0	0	0	0	1	0
6	•	0	0	0	0	0	0	0

Applications

- Analyzing genes
  - DNA is a long string over A,T,G,C
  - Two species are closer if their DNA has longer common subsequence
- UNIX diff command
  - Compares text files
  - Find longest matching subsequence of lines

Inductive structure

u	a_0	a_1	a_2	…	…	a_{m-1}
v	b_0	b_1	b_2	…	b_{n-1}	

- If $a_0 = b_0$,

  $LCS(a_0 \ldots a_{m-1}, b_0 \ldots b_{n-1}) = 1 + LCS(a_1 a_2 \ldots a_{m-1}, b_1 b_2 \ldots b_{n-1})$

  - Can force (a_0, b_0) to be part of LCS
- If not, a_0 and b_0 cannot both be part of LCS
  - Not sure which one to drop
  - Solve both subproblems $LCS(a_1 a_2 \ldots a_{m-1}, b_0 b_1 \ldots b_{n-1})$ and
    $LCS(a_0 a_1 \ldots a_{m-1}, b_1 b_2 \ldots b_{n-1})$ and take the maximum

Inductive structure

u	a_i	a_{i+1}	a_{i+2}	…	…	a_{m-1}
v	b_j	b_{j+1}	b_{j+2}	…	b_{n-1}	

- LCS(i,j) stands for LCS($a_ia_{i+1}\ldots a_{m-1}$, $b_jb_{j+1}\ldots b_{n-1}$)
- If $a_i = b_j$, LCS(i,j) = 1 + LCS(i+1,j+1)
- If $a_i \neq b_j$, LCS(i,j) = max(LCS(i+1,j), LCS(i,j+1))
- As with LCW, extend positions to m, n
  - LCS(m,j) = 0 for all j
  - LCS(i,n) = 0 for all i

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b							0
1	i							0
2	s							0
3	e							0
4	c							0
5	t							0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b						0	0
1	i						0	0
2	s						0	0
3	e						0	0
4	c						0	0
5	t						1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b					1	0	0
1	i					1	0	0
2	s					1	0	0
3	e					1	0	0
4	c					1	0	0
5	t					1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b				1	1	0	0
1	i				1	1	0	0
2	s				1	1	0	0
3	e				1	1	0	0
4	c				1	1	0	0
5	t				1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b			2	1	1	0	0
1	i			2	1	1	0	0
2	s			2	1	1	0	0
3	e			2	1	1	0	0
4	c			2	1	1	0	0
5	t			1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b		3	2	1	1	0	0
1	i		3	2	1	1	0	0
2	s		3	2	1	1	0	0
3	e		3	2	1	1	0	0
4	c		2	2	1	1	0	0
5	t		1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Subproblem dependency

- LCS(i,j) depends on LCS(i+1,j+1) as well as LCS(i+1,j) and LCS(i,j+1)
- Dependencies for LCS(m,n) are known
- Start at LCS(m,n) and fill by row, column or diagonal

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

Recovering the sequence

- Trace back the path by which each entry was filled
- Each diagonal step is an element of the LCS
    - “sect”

		0	1	2	3	4	5	6
		s	e	c	r	e	t	•
0	b	4	3	2	1	1	0	0
1	i	4	3	2	1	1	0	0
2	s	4	3	2	1	1	0	0
3	e	3	3	2	1	1	0	0
4	c	2	2	2	1	1	0	0
5	t	1	1	1	1	1	1	0
6	•	0	0	0	0	0	0	0

LCS(u,v), DP

```
def LCS(u,v): # u[0..m-1], v[0..n-1]
  for r in range(len(u)+1):
    LCS[r][len(v)+1] = 0 # r for row
  for c in range(len(v)+1):
    LCS[len(u)+1][c] = 0 # c for col
  for c in range(len(v),-1,-1):
    for r in range(len(u),-1,-1):
      if (u[r] == v[c])
        LCS[r][c] = 1 + LCS[r+1][c+1]
      else
        LCS[r][c] = max(LCS[r+1][c],
                        LCS[r][c+1])

  return(LCS[0][0])
```

Complexity

- Again O(mn) using dynamic programming (or memoization)
  - Need to fill an O(mn) size table
  - Each table entry takes constant time to compute