

Common subwords and subsequences

Madhavan Mukund

<https://www.cmi.ac.in/~madhavan>

Programming, Data Structures and Algorithms using Python

Week 9

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" — "ee", "re", length 2

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" — "ee", "re", length 2
- Formally
 - $u = a_0a_1 \dots a_{m-1}$
 - $v = b_0b_1 \dots b_{n-1}$

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" — "ee", "re", length 2
- Formally
 - $u = a_0a_1 \dots a_{m-1}$
 - $v = b_0b_1 \dots b_{n-1}$
 - Common subword of length k — for some positions i and j ,
 $a_ia_{i+1}a_{i+k-1} = b_jb_{j+1}b_{j+k-1}$

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" — "ee", "re", length 2

- Formally

- $u = a_0 a_1 \dots a_{m-1}$

 1st character of u

- $v = b_0 b_1 \dots b_{n-1}$

- Common subword of length k — for some positions i and j ,
 $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$

- Find the largest such k — length of the longest common subword

Brute force

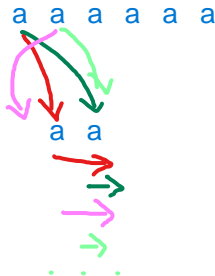
- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_ia_{i+1}a_{i+k-1} = b_jb_{j+1}b_{j+k-1}$

Brute force

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$
- Try every pair of starting positions i in u , j in v
 - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$ as far as possible
 - Keep track of longest match

Brute force

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- Try every pair of starting positions i in u , j in v
 - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \dots$ as far as possible
 - Keep track of longest match
- Assuming $m > n$, this is $O(mn^2)$
 - mn pairs of starting positions
 - From each starting position, scan could be $O(n)$



Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_i a_{i+1} \dots a_{i+k-1} = b_j b_{j+1} \dots b_{j+k-1}$

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_ia_{i+1}a_{i+k-1} = b_jb_{j+1}b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_ia_{i+1} \dots a_{m-1}$, $b_jb_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$

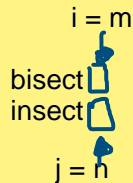
– if character at index i and j of u and v respectively are the same then increase the Longest Common Subword count by 1.

Note $LCW(i+1, j+1)$ simply means the longest common subword count with starting positions $i+1$ and $j+1$

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_ia_{i+1}a_{i+k-1} = b_jb_{j+1}b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_ia_{i+1} \dots a_{m-1}$, $b_jb_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$

Both `m` and `n` are out of range of v and u , thus it acts as a base. As both `m` and `n` are out of range there exist no common subwords with starting position `m` and `n`



Inductive structure

- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- Find the largest k such that for some positions i and j ,
 $a_ia_{i+1}a_{i+k-1} = b_jb_{j+1}b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_ia_{i+1} \dots a_{m-1}$, $b_jb_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$
 - In general, $LCW(i, n) = 0$ for all $0 \leq i \leq m$

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$ $\text{len}(u) = m$
- $v = b_0b_1 \dots b_{n-1}$ $\text{len}(v) = n$
- Find the largest k such that for some positions i and j ,
 $a_ia_{i+1}a_{i+k-1} = b_jb_{j+1}b_{j+k-1}$
- $LCW(i, j)$ — length of longest common subword in $a_ia_{i+1} \dots a_{m-1}$, $b_jb_{j+1} \dots b_{n-1}$
 - If $a_i \neq b_j$, $LCW(i, j) = 0$
 - If $a_i = b_j$, $LCW(i, j) = 1 + LCW(i+1, j+1)$
 - Base case: $LCW(m, n) = 0$
 - In general, $LCW(i, n) = 0$ for all $0 \leq i \leq m$
 - In general, $LCW(m, j) = 0$ for all $0 \leq j \leq n$

If starting position of LCW of either words is at the `m` or `n` respectively then this means there exist no common subwords

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values

		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

DYNAMIC PROGRAMMING

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$

Solving the problem using DYNAMIC PROGRAMMING will require to start from bottom up.

NOTE: In $LCW(i, j)$ and $LCW(i+1, j+1)$ the arguments are starting positions of u and v

Example:

$u = \text{"bisect"}$
 $v = \text{"secret"}$
For getting longest common subword from here you need to know longest common subword starting from here

		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

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

As we are solving using dynamic programming we follow bottom up approach

(m, j)

		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

(i, n)

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column 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

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

$LCW(i+1, j+1) = 0$

+

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column 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

if $a_i \neq b_j$
then $LCW = 0$

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column 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

Subproblem dependency

- Subproblems are $LCW(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCW(i,j)$ depends on $LCW(i+1,j+1)$
- Start at bottom right and fill row by row or column 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

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column 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

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column 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

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value

		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

Subproblem dependency

- Subproblems are $LCW(i,j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- $LCW(i,j)$ depends on $LCW(i+1,j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i,j) with largest LCW value
- 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

Subproblem dependency

- Subproblems are $LCW(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCW(i, j)$ depends on $LCW(i+1, j+1)$
- Start at bottom right and fill row by row or column by column

Reading off the solution

- Find entry (i, j) with largest LCW value
- Read off the actual subword diagonally on projecting this diagonal on either side you will get LCW

		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

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-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)
```

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-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

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-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 brute force was $O(mn^2)$

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-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 brute force was $O(mn^2)$
- Inductive solution is $O(mn)$, using dynamic programming or memoization

Implementation

```
def LCW(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcw = np.zeros((m+1,n+1))

    maxlcw = 0

    for c in range(n-1,-1,-1):
        for r in range(m-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 brute force was $O(mn^2)$
- Inductive solution is $O(mn)$, using dynamic programming or memoization
 - Fill a table of size $O(mn)$
 - 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", "trisection" — "isect", length 5
 - "bisect", "secret" — "sect", length 4
 - "director", "secretary" — "ectr", "retr", length 4

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", "trisection" — "isect", length 5
 - "bisect", "secret" — "sect", length 4
 - "director", "secretary" — "ectr", "retr", length 4
- LCS is the longest path connecting non-zero LCW entries, moving right/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

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", "trisection" — "isect", length 5
 - "bisect", "secret" — "sect", length 4
 - "director", "secretary" — "ectr", "retr", length 4
- LCS is the longest path connecting non-zero LCW entries, moving right/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 similar if their DNA has long common subsequences

		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 similar if their DNA has long common subsequences

- `diff` command in Unix/Linux

- Compares text files
- Find the longest matching subsequence of lines
- Each line of text is a “character”

		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

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$

- $v = b_0 b_1 \dots b_{n-1}$

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_ia_{i+1} \dots a_{m-1}, b_jb_{j+1} \dots b_{n-1}$

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_ia_{i+1} \dots a_{m-1}, b_jb_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS

secret

bisect

Inductive structure

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_i a_{i+1} \dots a_{m-1}, b_j b_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS

secret

bisect

here $i = 2$ and $j = 2$

for $i = 3$ and $j = 3$ we already have computed LCS
so $LCS(3, 3) = 2$ ("et")

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$

secret

- $v = b_0b_1 \dots b_{n-1}$

bisect

- $LCS(i, j)$ — length of longest common subsequence in

$a_ia_{i+1} \dots a_{m-1}, b_jb_{j+1} \dots b_{n-1}$

if we drop "s" we get two possible LCS "et" or "ct".

- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$

If we drop "c" we get LCS "et"

- Can assume (a_i, b_j) is part of LCS

In both cases LCS length is 2 so we can drop either "c" or "s"

- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS

- Which one should we drop?

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_ia_{i+1} \dots a_{m-1}, b_jb_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$

secret

bisect

- Can assume (a_i, b_j) is part of LCS

- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS

implicitly dropping "c" in other words
considering this part of "secret"

- Which one should we drop?

- Solve $LCS(i, j+1)$ and $LCS(i+1, j)$ and take the maximum

So, $LCS(i, j) = \max(LCS(i, j+1), LCS(i+1, j))$

implicitly dropping "s" in other words
considering this part of "bisect"

Inductive structure

- $u = a_0a_1 \dots a_{m-1}$
- $v = b_0b_1 \dots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_ia_{i+1} \dots a_{m-1}, b_jb_{j+1} \dots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
 - Can assume (a_i, b_j) is part of LCS
- If $a_i \neq b_j$, a_i and b_j cannot both be part of the LCS
 - Which one should we drop?
 - Solve $LCS(i, j+1)$ and $LCS(i+1, j)$ and take the maximum
- Base cases as with LCW
 - $LCS(i, n) = 0$ for all $0 \leq i \leq m$
 - $LCS(m, j) = 0$ for all $0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values

		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

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1), LCS(i+1, j)$,

When finding LCS of secret and bisect
 $LCS(i+1, j+1)$ is nothing but LCS of
 ecret and isect

		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

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right 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



Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right 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

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right 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

take maximum

diagonal + 1
as its a match

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right and fill by row, column or diagonal

reminder: we are computing from bottom to top

		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

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right and fill by row, column or diagonal

THINK: If bisect and secret has LCS of 2 then of course bisect and secret has minimum LCS of 2

		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

drop s => "ect" and "cret"
drop e => "ret" and "sect"

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right 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

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right and fill by row, column or diagonal

Your answer LCS length is always on the index (0, 0)

That is because $LCS(0, 0)$ means LCS of bisect and secret. And $LCS(1, 1)$ means LCS of "ecret" and "isect"

		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

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right and fill by row, column or diagonal

Reading off the solution

- Trace back the path by which each entry was filled

		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

Subproblem dependency

- Subproblems are $LCS(i, j)$, for $0 \leq i \leq m, 0 \leq j \leq n$
- Table of $(m + 1) \cdot (n + 1)$ values
- $LCS(i, j)$ depends on $LCS(i+1, j+1)$, $LCS(i, j+1)$, $LCS(i+1, j)$,
- No dependency for $LCS(m, n)$ — start at bottom right and fill by row, column or diagonal

on every diagonal a new element is added to the LCS

Reading off the solution

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

		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

Implementation

```
def LCS(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcs = np.zeros((m+1,n+1))

    for c in range(n-1,-1,-1):
        for r in range(m-1,-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])
```


Implementation

```
def LCS(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcs = np.zeros((m+1,n+1))

    for c in range(n-1,-1,-1):
        for r in range(m-1,-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

Implementation

```
def LCS(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcs = np.zeros((m+1,n+1))

    for c in range(n-1,-1,-1):
        for r in range(m-1,-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

Implementation

```
def LCS(u,v):
    import numpy as np
    (m,n) = (len(u),len(v))
    lcs = np.zeros((m+1,n+1))

    for c in range(n-1,-1,-1):
        for r in range(m-1,-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
 - Fill a table of size $O(mn)$
 - Each table entry takes constant time to compute