Dynamic Programming

Madhavan Mukund

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

Programming and Data Structures with Python

Lecture 22, 09 Nov 2023

Memoizing recursive implementations

```
def fib(n):
  if n in fibtable.keys():
    return(fibtable[n])
  if n <= 1:
    value = n
  else:
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

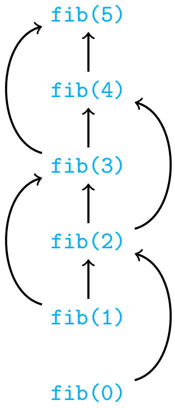
In general

```
def f(x,y,z):
  if (x,y,z) in ftable.keys():
    return(ftable[(x,y,z)])
  recursively compute value
    from subproblems
  ftable[(x,y,z)] = value
  return(value)
```

Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic

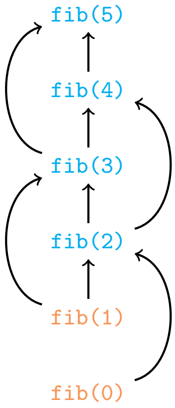
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies

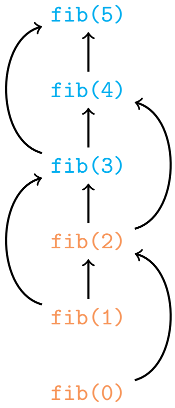
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

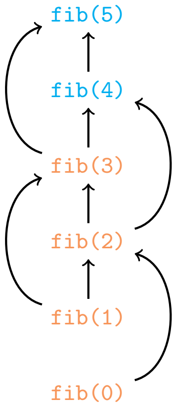
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

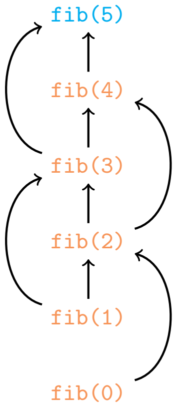
Evaluating `fib(5)`



Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

Evaluating `fib(5)`

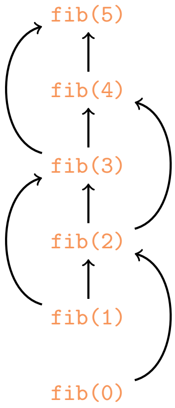


Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available

n	0	1	2	3	4	5
fib(n)	0	1	1	2	3	5

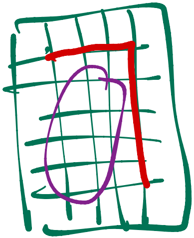
Evaluating `fib(5)`

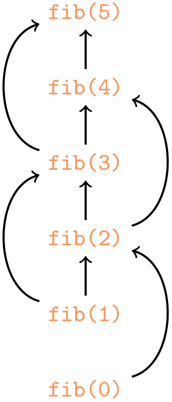


Dynamic programming

- Anticipate the structure of subproblems
  - Derive from inductive definition
  - Dependencies are acyclic
- Solve subproblems in appropriate order
  - Start with base cases — no dependencies
  - Evaluate a value after all its dependencies are available
  - Fill table iteratively
  - Never need to make a recursive call

Evaluating `fib(5)`





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

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
- Formally
  - $u = a_0 a_1 \ldots a_{m-1}$
  - $v = b_0 b_1 \ldots b_{n-1}$

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" — "ec", "re", length 2
- Formally
  - $u = a_0 a_1 \ldots a_{m-1}$
  - $v = b_0 b_1 \ldots b_{n-1}$
  - Common subword of length k — for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

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" — "ec", "re", length 2
- Formally
  - $u = a_0 a_1 \ldots a_{m-1}$
  - $v = b_0 b_1 \ldots b_{n-1}$
  - Common subword of length k — for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
  - Find the largest such k — length of the longest common subword

abdbab
acdcac

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, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$

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, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- 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

Brute force

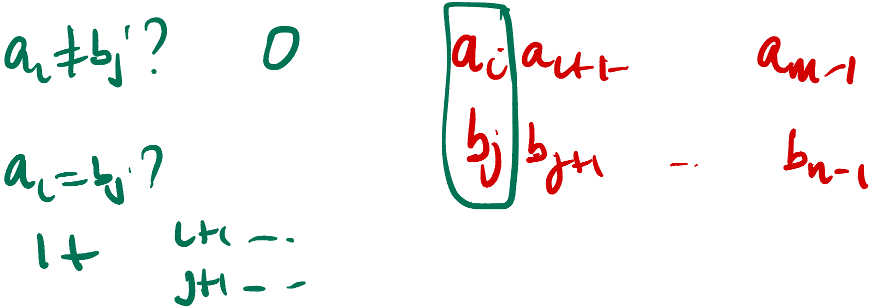
- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Find the largest k such that for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- Try every pair of starting positions i in u, j in v
  - Match $(a_i, b_j), (a_{i+1}, b_{j+1}), \ldots$ as far as possible
  - Keep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

$m = n$

$O(n^3)$

Inductive structure

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



Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Find the largest k such that for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i,j)$ — length of longest common subword in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
  - If $a_i \neq b_j$, $LCW(i,j) = 0$
  - If $a_i = b_j$, $LCW(i,j) = 1 + LCW(i+1, j+1)$

$LCW(0,0) = 0$

2
bisect
trisect
3

$LCW(2,3) = 4$

$\Rightarrow LCW(1,2) = 1+4 = 5$

Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Find the largest k such that for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i,j)$ — length of longest common subword in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
  - 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: $LCW(m,n) = 0$

Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Find the largest k such that for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i,j)$ — length of longest common subword in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
  - 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

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Find the largest k such that for some positions i and j, $a_i a_{i+1} a_{i+k-1} = b_j b_{j+1} b_{j+k-1}$
- $LCW(i,j)$ — length of longest common subword in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
  - 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 $0 \leq j \leq n$

Equivalently formulate for

$LCW(i,j)$ $a_0 \ldots a_i$
$b_0 \ldots b_j$

Subproblem dependency

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

≤ covers the base cases LCW(m, j)
LCW(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

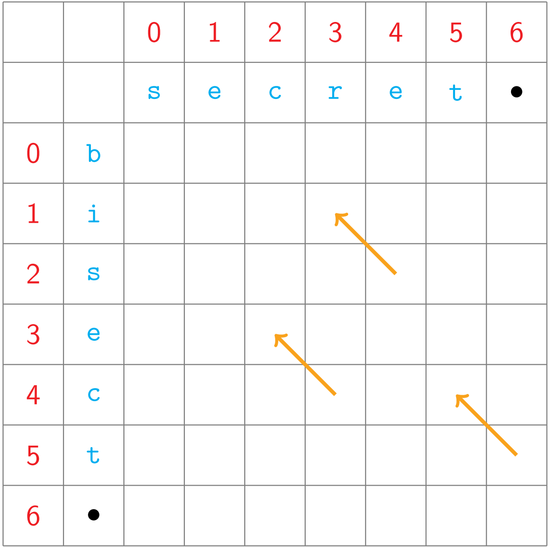
	$i \rightarrow$	0	1	2	3	4	5	6
$j \downarrow$		s	e	c	r	e	t	•
0	b							
1	i							
2	s							
3	e							
4	c							
5	t							
6	•							

LCW(2,3)

1+?

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



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**

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					1 +	**0**	**0**
4	c					0	**0**	**0**
5	t					0	**1**	**0**
6	•					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**
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**

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		1+	**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	1+	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

		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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcw[i,j] = 1 + lcw[i+1,j+1]
      else:
        lcw[i,j] = 0
      if lcw[i,j] > maxlcw:
        maxlcw = lcw[i,j]

  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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcw[i,j] = 1 + lcw[i+1,j+1]
      else:
        lcw[i,j] = 0
      if lcw[i,j] > maxlcw:
        maxlcw = lcw[i,j]

  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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcw[i,j] = 1 + lcw[i+1,j+1]
      else:
        lcw[i,j] = 0
      if lcw[i,j] > maxlcw:
        maxlcw = lcw[i,j]

  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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcw[i,j] = 1 + lcw[i+1,j+1]
      else:
        lcw[i,j] = 0
      if lcw[i,j] > maxlcw:
        maxlcw = lcw[i,j]

  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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcw[i,j] = 1 + lcw[i+1,j+1]
      else:
        lcw[i,j] = 0
      if lcw[i,j] > maxlcw:
        maxlcw = lcw[i,j]

  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", "trisect" — "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", "trisect" — "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", "trisect" — "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

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$

Inductive structure

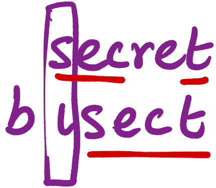
- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$

Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- If $a_i = b_j$, $LCS(i, j) = 1 + LCS(i+1, j+1)$
  - Can assume (a_i, b_j) is part of LCS

Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- $LCS(i, j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- 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



Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- 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?

Inductive structure

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- 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

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
  - 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 $0 \leq j \leq n$

$a_i \neq b_j$

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

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 $LCS(i+1,j+1)$, $LCS(i,j+1)$, $LCS(i+1,j)$,

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

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

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

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

Subproblem dependency

- Subproblems are $LCS(i,j)$, for $0 \le i \le m$, $0 \le j \le n$
- 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	4	3	2	2	2	1	0
1	i	4	3	2	2	2	1	0
2	s	4	3	2	2	2	1	0
3	e	3	3	2	2	2	1	0
4	c	2	2	2	1	1	1	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	2	2	1	0
1	i	4	3	2	2	2	1	0
2	s	4	3	2	2	2	1	0
3	e	3	3	2	2	2	1	0
4	c	2	2	2	1	1	1	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
- 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	2	2	1	0
1	i	4	3	2	2	2	1	0
2	s	4	3	2	2	2	1	0
3	e	3	3	2	2	2	1	0
4	c	2	2	2	1	1	1	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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcs[i,j] = 1 + lcs[i+1,j+1]
      else:
        lcs[i,j] = max(lcs[i+1,j],
                       lcs[i,j+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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcs[i,j] = 1 + lcs[i+1,j+1]
      else:
        lcs[i,j] = max(lcs[i+1,j],
                       lcs[i,j+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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcs[i,j] = 1 + lcs[i+1,j+1]
      else:
        lcs[i,j] = max(lcs[i+1,j],
                       lcs[i,j+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 j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        lcs[i,j] = 1 + lcs[i+1,j+1]
      else:
        lcs[i,j] = max(lcs[i+1,j],
                       lcs[i,j+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

Document similarity

- “The students were able to appreciate the concept optimal substructure property and its use in designing algorithms”
- “The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms”

Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"
- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another

Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"
- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another

- "The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructures ~~property~~ cand ~~it~~bse used in designing algorithms"
- insert, ~~delete~~, substitute

Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"
- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another

- "The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructures ~~property~~ cand itbse used in designing algorithms"
- insert, ~~delete~~, substitute

Edit distance

Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"
- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another

- "The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructures ~~property~~ cand itbse used in designing algorithms"
- insert, ~~delete~~, substitute

Edit distance

- Minimum number of edit operations needed

Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"
- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another

- "The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructures ~~property~~ cand itbse used in designing algorithms"
- insert, ~~delete~~, substitute

Edit distance

- Minimum number of edit operations needed
- In our example, 24 characters inserted, 18 ~~deleted~~, 2 substituted

Document similarity

- "The students were able to appreciate the concept optimal substructure property and its use in designing algorithms"
- "The lecture taught the students to appreciate how the concept of optimal substructures can be used in designing algorithms"
- Edit operations to transform documents
  - Insert a character
  - Delete a character
  - Substitute one character by another

- "The lecture taught the students ~~were able~~ to appreciate how the concept of optimal substructures ~~property~~ cand itbse used in designing algorithms"
- insert, ~~delete~~, substitute

Edit distance

- Minimum number of edit operations needed
- In our example, 24 characters inserted, 18 ~~deleted~~, 2 substituted
- Edit distance is at most 44

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance and LCS

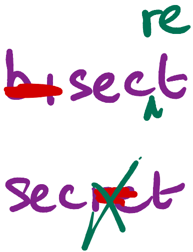
- Longest common subsequence of u, v

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v
  - Minimum number of deletes needed to make them equal



Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v
  - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v
  - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
  - `bisect`, `secret` — LCS is `sect`

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v
  - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
  - `bisect`, `secret` — LCS is `sect`
  - Delete `b`, `i` in `bisect` and `r`, `e` in `secret`

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v
  - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
  - `bisect`, `secret` — LCS is `sect`
  - Delete `b`, `i` in `bisect` and `r`, `e` in `secret`
  - Delete `b`, `i` and then insert `r`, `e` in `bisect`

Edit distance

- Minimum number of editing operations needed to transform one document to the other
  - Insert a character
  - Delete a character
  - Substitute one character by another
- Also called Levenshtein distance
  - Vladimir Levenshtein, 1965
- Applications
  - Suggestions for spelling correction
  - Genetic similarity of species

Edit distance and LCS

- Longest common subsequence of u, v
  - Minimum number of deletes needed to make them equal
- Deleting a letter from u is equivalent to inserting it in v
  - `bisect`, `secret` — LCS is `sect`
  - Delete `b`, `i` in `bisect` and `r`, `e` in `secret`
  - Delete `b`, `i` and then insert `r`, `e` in `bisect`
- From LCS, we can compute edit distance without substitution

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$

- Recall LCS

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$

- Recall LCS
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- If $a_i = b_j$,
  $LCS(i,j) = 1 + LCS(i+1, j+1)$
- If $a_i \neq b_j$,
  $LCS(i,j) = \max[\ LCS(i, j+1),$
  $LCS(i+1, j)\ ]$

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$

- Recall LCS
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- If $a_i = b_j$, $LCS(i,j) = 1 + LCS(i{+}1, j{+}1)$
- If $a_i \neq b_j$, $LCS(i,j) = \max[\ LCS(i, j{+}1),\ LCS(i{+}1, j)\ ]$

- Edit distance — aim is to transform u to v

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$

- Recall LCS
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- If $a_i = b_j$,
  $LCS(i,j) = 1 + LCS(i{+}1, j{+}1)$
- If $a_i \neq b_j$,
  $LCS(i,j) = \max[\ LCS(i, j{+}1),$
  $LCS(i{+}1, j)\ ]$

- Edit distance — aim is to transform u to v
- If $a_i = b_j$, nothing to be done

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

- Recall LCS
- $LCS(i,j)$ — 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 $a_i = b_j$,
  $LCS(i,j) = 1 + LCS(i{+}1, j{+}1)$
- If $a_i \neq b_j$,
  $LCS(i,j) = \max[\ LCS(i, j{+}1),\ LCS(i{+}1, j)\ ]$

- Edit distance — aim is to transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$

- Recall LCS
- $LCS(i,j)$ — 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 $a_i = b_j$, $LCS(i,j) = 1 + LCS(i+1, j+1)$
- If $a_i \neq b_j$, $LCS(i,j) = \max[\ LCS(i,j+1),\ LCS(i+1,j)\ ]$

- Edit distance — aim is to transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$

- Recall LCS
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- If $a_i = b_j$,
  $LCS(i,j) = 1 + LCS(i{+}1, j{+}1)$
- If $a_i \neq b_j$,
  $LCS(i,j) = \max[\ LCS(i, j{+}1),\ LCS(i{+}1, j)\ ]$

- Edit distance — aim is to transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$

- Recall LCS
- $LCS(i,j)$ — length of longest common subsequence in $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$
- If $a_i = b_j$,
  $LCS(i,j) = 1 + LCS(i{+}1, j{+}1)$
- If $a_i \neq b_j$,
  $LCS(i,j) = \max[\ LCS(i, j{+}1),$
  $LCS(i{+}1, j)\ ]$

- Edit distance — aim is to transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i

Delete b_j

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i
- $ED(i,j)$ — edit distance for $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i
- $ED(i,j)$ — edit distance for $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$

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

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i
- $ED(i,j)$ — edit distance for $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$

- If $a_i = b_j$,
  $ED(i,j) = ED(i{+}1, j{+}1)$
- If $a_i \neq b_j$,
  $$ED(i,j) = 1 + \min[\ ED(i{+}1, j{+}1),\ ED(i{+}1, j),\ ED(i, j{+}1)\ ]$$

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i
- $ED(i,j)$ — edit distance for $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$

- If $a_i = b_j$,
  $ED(i,j) = ED(i{+}1, j{+}1)$
- If $a_i \neq b_j$,
  $ED(i,j) = 1 + \min[\ ED(i{+}1, j{+}1),$ subst
  $ED(i{+}1, j),$ delete a
  $ED(i, j{+}1)\ ]$ deleted bj
- Base cases
  - $ED(m,n) = 0$

Inductive structure for edit distance

- $u = a_0 a_1 \ldots a_{m-1}$
- $v = b_0 b_1 \ldots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i
- $ED(i,j)$ — edit distance for $a_i a_{i+1} \ldots a_{m-1}$, $b_j b_{j+1} \ldots b_{n-1}$

- If $a_i = b_j$,
  $ED(i,j) = ED(i{+}1, j{+}1)$
- If $a_i \neq b_j$,
  $ED(i,j) = 1 + \min[\ ED(i{+}1,j{+}1),\ ED(i{+}1,j),\ ED(i,j{+}1)\ ]$
- Base cases
  - $ED(m,n) = 0$
  - $ED(i,n) = m - i$ for all $0 \leq i \leq m$
    Delete $a_i a_{i+1} \ldots a_{m-1}$ from u

Inductive structure for edit distance

- $u = a_0 a_1 \dots a_{m-1}$
- $v = b_0 b_1 \dots b_{n-1}$
- Edit distance — transform u to v
- If $a_i = b_j$, nothing to be done
- If $a_i \neq b_j$, best among
  - Substitute a_i by b_j
  - Delete a_i
  - Insert b_j before a_i
- $ED(i,j)$ — edit distance for $a_i a_{i+1} \dots a_{m-1}$, $b_j b_{j+1} \dots b_{n-1}$

- If $a_i = b_j$,
  $ED(i,j) = ED(i+1,j+1)$
- If $a_i \neq b_j$,
  $$ED(i,j) = 1 + \min[\ ED(i+1,j+1),\ ED(i+1,j),\ ED(i,j+1)\ ]$$
- Base cases
  - $ED(m,n) = 0$
  - $ED(i,n) = m - i$ for all $0 \leq i \leq m$
    Delete $a_i a_{i+1} \dots a_{m-1}$ from u
  - $ED(m,j) = n - j$ for all $0 \leq j \leq n$
    Insert $b_j b_{j+1} \dots b_{n-1}$ into u

Subproblem dependency

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

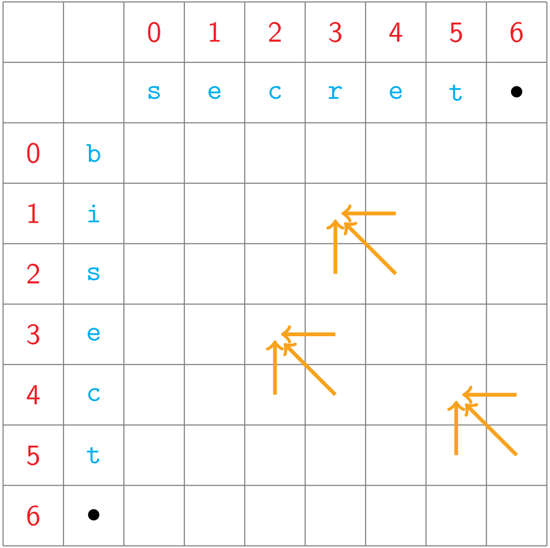
Subproblem dependency

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



Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i{+}1, j{+}1)$, $ED(i, j{+}1)$, $ED(i{+}1, j)$
- No dependency for $ED(m,n)$ — 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							**6**
1	i							**5**
2	s							**4**
3	e							**3**
4	c							**2**
5	t							**1**
6	•							**0**

Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i{+}1,j{+}1)$, $ED(i,j{+}1)$, $ED(i{+}1,j)$
- No dependency for $ED(m,n)$ — 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						**5**	**6**
1	i						**4**	**5**
2	s						**3**	**4**
3	e						**2**	**3**
4	c						**1**	**2**
5	t						**0**	**1**
6	•						**1**	**0**

Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i{+}1,j{+}1)$, $ED(i,j{+}1)$, $ED(i{+}1,j)$
- No dependency for $ED(m,n)$ — 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					4	5	6
1	i					3	4	5
2	s					2	3	4
3	e					1	2	3
4	c					1	1	2
5	t					1	0	1
6	•					2	1	0

Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i{+}1,j{+}1)$, $ED(i,j{+}1)$, $ED(i{+}1,j)$
- No dependency for $ED(m,n)$ — 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				4	4	5	6
1	i				3	3	4	5
2	s				2	2	3	4
3	e				2	1	2	3
4	c				2	1	1	2
5	t				2	1	0	1
6	•				3	2	1	0

Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i{+}1,j{+}1)$, $ED(i,j{+}1)$, $ED(i{+}1,j)$
- No dependency for $ED(m,n)$ — 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			**4**	**4**	**4**	**5**	**6**
1	i			**3**	**3**	**3**	**4**	**5**
2	s			**3**	**2**	**2**	**3**	**4**
3	e			**3**	**2**	**1**	**2**	**3**
4	c			**2**	**2**	**1**	**1**	**2**
5	t			**3**	**2**	**1**	**0**	**1**
6	•			**4**	**3**	**2**	**1**	**0**

Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i{+}1,j{+}1)$, $ED(i,j{+}1)$, $ED(i{+}1,j)$
- No dependency for $ED(m,n)$ — 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		4	4	4	4	5	6
1	i		4	3	3	3	4	5
2	s		3	3	2	2	3	4
3	e		2	3	2	1	2	3
4	c		3	2	2	1	1	2
5	t		4	3	2	1	0	1
6	•		5	4	3	2	1	0

Subproblem dependency

- Subproblems are $ED(i,j)$, for $0 \leq i \leq m$, $0 \leq j \leq n$
- Table of $(m+1) \cdot (n+1)$ values
- Like LCS, $ED(i,j)$ depends on $ED(i{+}1,j{+}1)$, $ED(i,j{+}1)$, $ED(i{+}1,j)$
- No dependency for $ED(m,n)$ — 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	4	4	4	4	4	5	6
1	i	3	4	3	3	3	4	5
2	s	2	3	3	2	2	3	4
3	e	3	2	3	2	1	2	3
4	c	4	3	2	2	1	1	2
5	t	5	4	3	2	1	0	1
6	•	6	5	4	3	2	1	0

Subproblem dependency

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

Reading off the solution

- Transform `bisect` to `secret`

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

Subproblem dependency

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

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b`

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

Subproblem dependency

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

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b` , Delete `i`

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

Subproblem dependency

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

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b` , Delete `i` , Insert `r`

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

Subproblem dependency

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

Reading off the solution

- Transform `bisect` to `secret`
- Delete `b` , Delete `i` , Insert `r` , Insert `e`

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

Implementation

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

  for i in range(m-1,-1,-1):
    ed[i,n] = m-i
  for j in range(n-1,-1,-1):
    ed[m,j] = n-j

  for j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        ed[i,j] = ed[i+1,j+1]
      else:
        ed[i,j] = 1 + min(ed[i+1,j+1],
                          ed[i,j+1],
                          ed[i+1,j])
  return(ed[0,0])
```

Implementation

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

  for i in range(m-1,-1,-1):
    ed[i,n] = m-i
  for j in range(n-1,-1,-1):
    ed[m,j] = n-j

  for j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        ed[i,j] = ed[i+1,j+1]
      else:
        ed[i,j] = 1 + min(ed[i+1,j+1],
                          ed[i,j+1],
                          ed[i+1,j])
  return(ed[0,0])
```

Complexity

Implementation

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

  for i in range(m-1,-1,-1):
    ed[i,n] = m-i
  for j in range(n-1,-1,-1):
    ed[m,j] = n-j

  for j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        ed[i,j] = ed[i+1,j+1]
      else:
        ed[i,j] = 1 + min(ed[i+1,j+1],
                          ed[i,j+1],
                          ed[i+1,j])
  return(ed[0,0])
```

Complexity

- Again $O(mn)$, using dynamic programming or memoization

Implementation

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

  for i in range(m-1,-1,-1):
    ed[i,n] = m-i
  for j in range(n-1,-1,-1):
    ed[m,j] = n-j

  for j in range(n-1,-1,-1):
    for i in range(m-1,-1,-1):
      if u[i] == v[j]:
        ed[i,j] = ed[i+1,j+1]
      else:
        ed[i,j] = 1 + min(ed[i+1,j+1],
                          ed[i,j+1],
                          ed[i+1,j])
  return(ed[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