DYNAMIC PROGRAMMING

David Kauchak
CS 140 – Spring 2023

1

Admin

Back to normal schedule

Assignment 7 out tomorrow

2

Knapsack problems: Greedy or not?

0-1 Knapsack – A thief robbing a store finds n items worth v_1, v_2, .., v_n dollars and weight w_1, w_2, …, w_n pounds, where v_i and w_i are integers. The thief can carry at most W pounds in the knapsack. Which items should the thief take if they want to maximize value.

Fractional knapsack problem – Same as above, but the thief happens to be at the bulk section of the store and can carry fractional portions of the items. For example, the thief could take 20% of item i for a weight of $0.2w_i$ and a value of $0.2v_i$.

3

Algorithmic "techniques"

Iterative/incremental: solve problem of size n by first solving problem of size n-1.

Divide-and-conquer: divide problem into independent subproblems. Solve each subproblem independently. Combine solutions to subproblem to create solution to the original problem.

Greedy: make locally optimal choice and repeat on remaining subproblem.

4

Dynamic programming

Method for solving problems where optimal solutions can be defined in terms of optimal solutions to subproblems

AND

the subproblems are overlapping

5

Fibonacci: a first attempt

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

6

Running time

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Each call creates two recursive calls

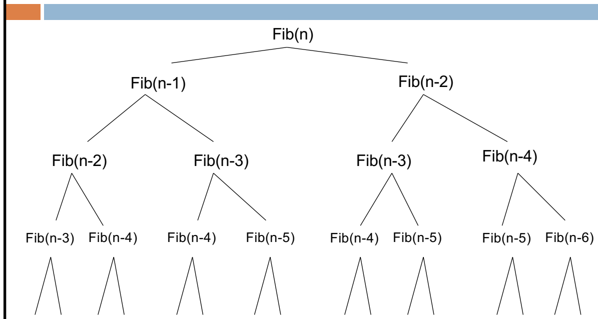
Each call reduces the size of the problem by 1 or 2

Creates a full binary of depth n

$O(2^n)$

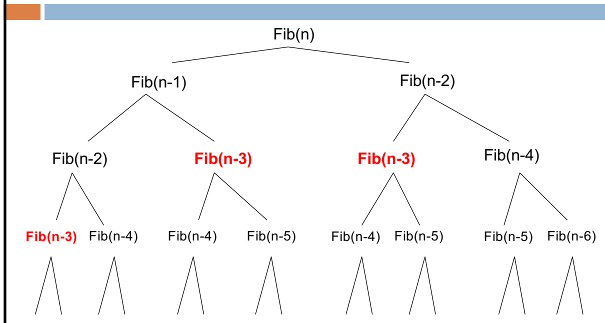
7

Can we do better?



8

A lot of repeated work!



9

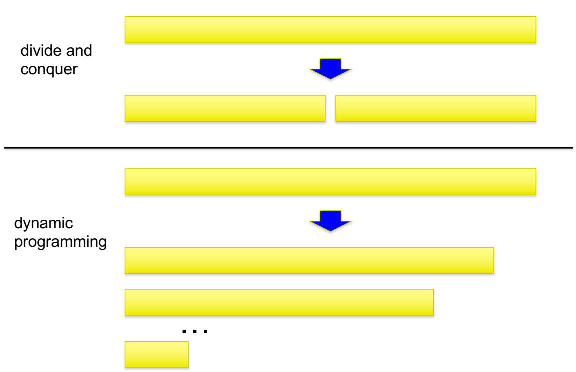
Identifying a dynamic programming problem

The solution can be defined with respect to solutions to subproblems

The subproblems created are *overlapping*, that is **we see the same subproblems repeated**

10

Overlapping sub-problems



11

Dynamic programming: steps

1a) optimal substructure: optimal solutions to the problem incorporate optimal solutions to related subproblems

- convince yourself that there is optimal substructure

1b) recursive definition: use this to recursively define the value of an optimal solution

2) DP solution: describe the dynamic programming table:

- size, initial values, order in which it's filled in, location of solution

3) Analysis: analyze space requirements, running time

12

1a: optimal substructure

optimal solutions to a problem incorporate optimal solutions to subproblems

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

?

13

1a: optimal substructure

optimal solutions to a problem incorporate optimal solutions to subproblems

```
FIBONACCI(n)
if n = 1 or n = 2
        return 1
else
        return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Sometimes the problem setup/structure meets the optimal substructure criteria by definition

14

1b: recursive definition

Define a function and *clearly* define the inputs to the function

The function definition should be *recursive* with respect to multiple subproblems

- pretend like you have a working function, but it only works on smaller problems

Key: subproblems will be *overlapping*, i.e., inputs to subproblems will not be disjoint

15

1b: recursive definition

Fibonacci:

F(n) = ?

16

1b: recursive definition

Fibonacci:

F(n) = F(n-1) + F(n-2)

17

2: DP solution

The recursive solution will generally be top-down, i.e., working from larger problems to smaller

DP solution:

- work bottom-up, from the smallest versions of the problem to the largest
- store the answers to subproblems in a table (often an array or matrix)
- to build bigger problems, lookup solutions in the table to subproblems

18

2: DP solution

F(n) = F(n-1) + F(n-2)

What are the smallest possible values (subproblems)?

To calculate F(n), what are all the subproblems we need to calculate? This is the "table".

How should we fill in the table?

19

2: DP solution

F(n) = F(n-1) + F(n-2)

What are the smallest possible values (subproblems)? F(1) = 1, F(2) = 1

To calculate F(n), what are all the subproblems we need to calculate? This is the "table". F(1) … F(n-1)

How should we fill in the table? F(1) → F(n)

20

2: DP solution

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

Store the intermediary values in an array (*fib*)

21

3: Analysis

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

Space requirements?

Running time?

22

3: Analysis

```
FIBONACCI-DP(n)
fib[1] ← 1
fib[2] ← 1
for i ← 3 to n
        fib[i] ← fib[i − 1] + fib[i − 2]
return fib[n]
```

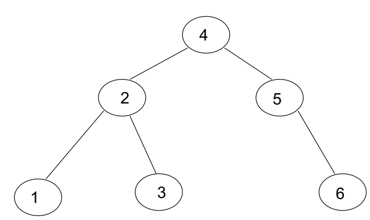
Space requirements: $\Theta(n)$

Running time: $\Theta(n)$

23

Counting binary search trees

How many unique binary search trees can be created using the numbers 1 through n?



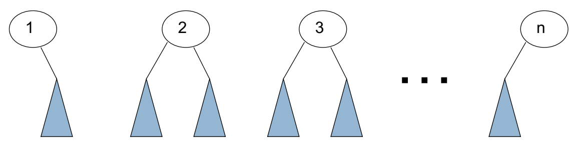
24

Step 1: What is the subproblem?

Assume we have a working function (call it T) that can give us the answer to smaller subproblems

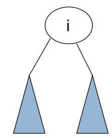
How can we use the answer from this to answer our question?

How many options for the root are there?



25

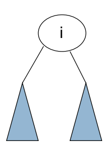
Subproblems



How many trees have i as the root?

26

Subproblems

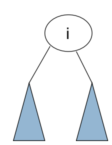


1, 2, …, i-1   i+1, i+2, …, n

?

27

Subproblems



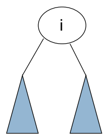
1, 2, …, i-1   i+1, i+2, …, n

T(i-1)   ?

subproblem of size i-1

28

Subproblems



1, 2, …, i-1   i+1, i+2, …, n

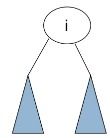
T(i-1)

Number of trees for i+1, i+2, …, i+n is the same as the number of trees from 1, 2, …, n-i

subproblem of size i-1

29

Subproblems



1, 2, …, i-1   i+1, i+2, …, n

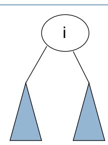
T(i-1)   T(n-i)

subproblem of size i-1   subproblem of size n-i

30

1a: optimal substructure

optimal solutions to a problem incorporate optimal solutions to related subproblems



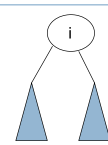
T(i-1) T(n-i)

?

31

1a: optimal substructure

optimal solutions to a problem incorporate optimal solutions to related subproblems

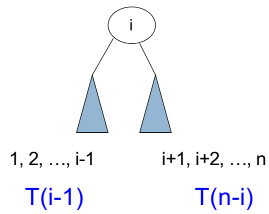


T(i-1) T(n-i)

By definition of binary trees: binary trees are recursive structures

32

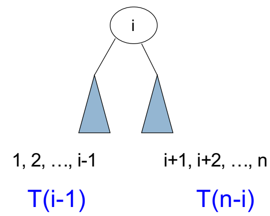
1b: recursive definition



Given solutions for T(i-1) and T(n-i) how many trees are there with i as the root?

33

1b: recursive definition



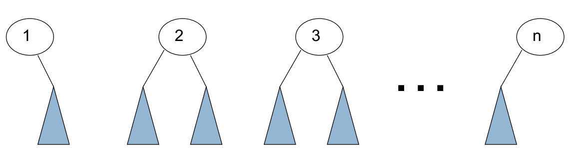
Trees with i as root = T(i-1) * T(n-i)

34

1b: recursive definition

Trees with i as root = T(i-1) * T(n-i)

How many trees total?

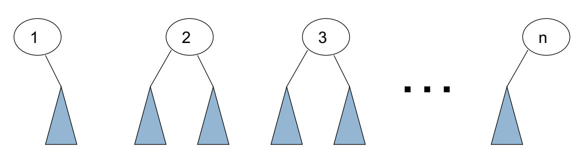


35

1b: recursive definition

Trees with i as root = T(i-1) * T(n-i)

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$



36

A recursive implementation

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

```
BST-COUNT(n)
  if n = 0
        return 1
  else
        sum = 0
        for i ← 1 to n
              sum ← sum + BST-COUNT(i − 1) * BST-COUNT(n − i)
  return sum
```

Like with Fibonacci, we're repeating a lot of work

37

2: DP solution (from the bottom-up)

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

What are the smallest possible subproblems?

To calculate T(n), what are all the subproblems we need to calculate? This is the "table".

How should we fill in the table?

38

2: DP solution (from the bottom-up)

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

What are the smallest possible subproblems?
T(0)=1, T(1) = 1

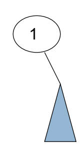
Why do we need T(0) and why is it 1?

39

2: DP solution (from the bottom-up)

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

What are the smallest possible subproblems?
T(0)=1, T(1) = 1



Need to think carefully about base cases/edge cases

40

2: DP solution (from the bottom-up)

$$T(n) = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

What are the smallest possible subproblems?
T(0)=1, T(1) = 1

To calculate T(n), what are all the subproblems we need to calculate? This is the "table". T(0) … T(n-1)

How should we fill in the table? T(0) → T(n)

41

2: DP solution (from the bottom-up)

```
BST-COUNT(n)
if n = 0
        return 1
else
        sum = 0
        for i ← 1 to n
                sum ← sum + BST-COUNT(i − 1) * BST-COUNT(n − i)
return sum
```

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

42

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

Fill in the first 4 values

0 1 2 3 4 5 … n

43

```
BST-COUNT-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1 1
0 1 2 3 4 5 … n

44

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

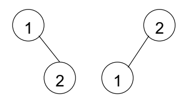
c[0]*c[1] + c[1]*c[0]

1 1

0 1 2 3 4 5 … n

45

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```



c[0]*c[1] + c[1]*c[0]

1 1

0 1 2 3 4 5 … n

46

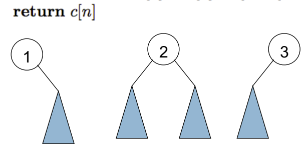
```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```

1 1 2

0 1 2 3 4 5 … n

47

```
BST-Count-DP(n)
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] ∗ c[k − i]
return c[n]
```



c[0]*c[2] + c[1]*c[1] + c[2]*c[0]

1 1 2

0 1 2 3 4 5 … n

48

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1	1	2	5			
0	1	2	3	4	5	… n

49

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

1	1	2	5	…		
0	1	2	3	4	5	… n

50

3: Analysis

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

Space requirements?

Running time?

51

3: Analysis

BST-COUNT-DP(n)

```
c[0] = 1
c[1] = 1
for k ← 2 to n
        c[k] ← 0
        for i ← 1 to k
                c[k] ← c[k] + c[i − 1] * c[k − i]
return c[n]
```

Space requirements: $\Theta(n)$

Running time: $\Theta(n^2)$

52

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

ABA?

53

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

ABA

54

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

ACA?

55

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \leq i_1 < i_2 < \ldots < i_k \leq n$

X = A B A C D A B A B

ACA

56

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

DCA?

57

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

~~DCA~~

58

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

AADAA?

59

Subsequences

For a sequence $X = x_1, x_2, \ldots, x_n$, a subsequence is a subset of the sequence defined by a set of increasing indices $(i_1, i_2, \ldots, i_k)$ where
$1 \le i_1 < i_2 < \ldots < i_k \le n$

X = A B A C D A B A B

AADAA

60

Longest common subsequence (LCS)

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_n$

What is the longest common subsequence?

61

LCS problem

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_n$

What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

62

LCS problem

Given two sequences X and Y, a **common subsequence** is a subsequence that occurs in both X and Y

Given two sequences $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_n$

What is the longest common subsequence?

X = A B C B D A B

Y = B D C A B A

63

1a: optimal substructure

optimal solutions to a problem incorporate optimal solutions to subproblems

Often a proof by contradiction:

- Show: optimal solutions incorporate optimal solutions to subproblems
- Assume the optimal solution does not contain optimal solutions to subproblems
- Show this leads to a contradiction (often that we could create a better solution using the solution to the subproblem)

64

1a: optimal substructure

Prove: optimal solutions to the problem incorporate optimal solutions to related subproblems

Proof by contradiction:

Assume: $s_1, s_2, \ldots, s_m$ is the LCS(X,Y), but $s_2, \ldots, s_m$ is not the optimal solution to
LCS(substring_after(s_1, X), substring_after(s_1, Y)).

If that were the case, then we could make a longer subsequence by:
s_1 LCS(substring_after(s_1, X), substring_after(s_1, Y))

contradiction

65

1b: recursive solution

X = A B C B D A B

Y = B D C A B A

Assume you have a solver for smaller problems

66

1b: recursive solution

X = A B C B D A ?

Y = B D C A B ?

Is the last character part of the LCS?

67

1b: recursive solution

X = A B C B D A ?

Y = B D C A B ?

Two cases: either the characters are the same or they're different

68

1b: recursive solution

X = $\boxed{\text{A B C B D A}}$ A

LCS

Y = $\boxed{\text{B D C A B}}$ A

The characters are part of the LCS

What is the recursive relationship?

If they're the same

$$LCS(X,Y) = LCS(X_{1...n-1}, Y_{1...m-1}) + x_n$$

69

1b: recursive solution

X = $\boxed{\text{A B C B D A}}$ B

LCS

Y = $\boxed{\text{B D C A B A}}$

If they're different

$$LCS(X,Y) = LCS(X_{1...n-1}, Y)$$

70

1b: recursive solution

X = $\boxed{\text{A B C B D A B}}$

LCS

Y = $\boxed{\text{B D C A B}}$ A

If they're different

$$LCS(X,Y) = LCS(X, Y_{1...m-1})$$

71

1b: recursive solution

X = $\boxed{\text{A B C B D A}}$ B

Y = $\boxed{\text{B D C A B A}}$

?

X = $\boxed{\text{A B C B D A B}}$

Y = $\boxed{\text{B D C A B}}$ A

If they're different

72

1b: recursive solution

X = A B C B D A B

Y = B D C A B A

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1...n-1}, Y_{1...m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1...n-1}, Y), LCS(X, Y_{1...m-1})) & \text{otherwise} \end{cases}$$

(for now, let's just worry about counting the length of the LCS)

73

2: DP solution

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1...n-1}, Y_{1...m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1...n-1}, Y), LCS(X, Y_{1...m-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_{1...j}, Y_{1...k})$

two different indices

74

2: DP solution

$$LCS(X,Y) = \begin{cases} 1 + LCS(X_{1...n-1}, Y_{1...m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1...n-1}, Y), LCS(X, Y_{1...m-1})) & \text{otherwise} \end{cases}$$

What types of subproblem solutions do we need to store?

$LCS(X_{1...j}, Y_{1...k})$

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

75

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i							
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							

For Fibonacci and tree counting, we had to initialize some entries in the array. Any here?

76

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \textit{if } x_i=y_j\\ \max(LCS[i-1,j],LCS[i,j-1]) & \text{otherwise}\end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Need to initialize values within 1 smaller in either dimension.

77

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \textit{if } x_i=y_j\\ \max(LCS[i-1,j],LCS[i,j-1]) & \text{otherwise}\end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

How should we fill in the table?

78

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \textit{if } x_i=y_j\\ \max(LCS[i-1,j],LCS[i,j-1]) & \text{otherwise}\end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0			?			
4	B	0						
5	D	0						
6	A	0						
7	B	0						

To fill in an entry, we may need to look:

- up one
- left one
- diagonal up and left

Just need to make sure these exist

79

$$LCS[i,j]=\begin{cases}1+LCS[i-1,j-1] & \textit{if } x_i=y_j\\ \boxed{\max(LCS[i-1,j],LCS[i,j-1]) \quad \text{otherwise}}\end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	?					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, B)

80

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0					
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

81

$$LCS[i,j] = \begin{cases} \boxed{1 + LCS[i-1, j-1] \quad \text{if } x_i = y_j} \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	?		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

82

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1		
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

LCS(A, BDCA)

83

$$LCS[i,j] = \begin{cases} \boxed{1 + LCS[i-1, j-1] \quad \text{if } x_i = y_j} \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	?	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

84

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

85

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Where's the final answer?

86

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Space requirements?

Running time?

87

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Space requirements: $\Theta(nm)$

Running time: $\Theta(nm)$

88

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i−1, j−1]
                elseif c[i−1, j] > c[i, j−1]
                        c[i,j] ← c[i−1, j]
                else
                        c[i,j] ← c[i, j−1]
return c[m,n]
```

89

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i−1, j−1]
                elseif c[i−1, j] > c[i, j−1]
                        c[i,j] ← c[i−1, j]
                else
                        c[i,j] ← c[i, j−1]
return c[m,n]
```

Base case initialization

90

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i−1, j−1]
                elseif c[i−1, j] > c[i, j−1]
                        c[i,j] ← c[i−1, j]
                else
                        c[i,j] ← c[i, j−1]
return c[m,n]
```

Fill in the matrix

91

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i−1, j−1]
                elseif c[i−1, j] > c[i, j−1]
                        c[i,j] ← c[i−1, j]
                else
                        c[i,j] ← c[i, j−1]
return c[m,n]
```

92

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i−1, j−1]
                elseif c[i−1, j] > c[i, j−1]
                        c[i,j] ← c[i−1, j]
                else
                        c[i,j] ← c[i, j−1]
return c[m,n]
```

93

The algorithm

```
LCS-LENGTH(X, Y)
m ← length[X]
n ← length[Y]
c[0,0] ← 0
for i ← 1 to m
        c[i,0] ← 0
for j ← 1 to n
        c[0,j] ← 0
for i ← 1 to m
        for j ← 1 to n
                if x_i = y_i
                        c[i,j] ← 1 + c[i−1, j−1]
                elseif c[i−1, j] > c[i, j−1]
                        c[i,j] ← c[i−1, j]
                else
                        c[i,j] ← c[i, j−1]
return c[m,n]
```

94

Keeping track of the solution

Our LCS algorithm only calculated the length of the LCS between X and Y

What if we wanted to know the actual sequence?

95

$$LCS[i,j] = \begin{cases} \boxed{1 + LCS[i-1, j-1] \quad \text{if } x_i = y_j} \\ \max(LCS[i-1, j], LCS[i, j-1] & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	?	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

96

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1,j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCAB)

97

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1,j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	?
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCABA)

98

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1,j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0						
6	A	0						
7	B	0						

LCS(ABCB, BDCABA)

99

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1,j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1,j], LCS[i,j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

How do we generate the solution from this?

100

$$LCS[i, j] = \begin{cases} 1 + LCS[i-1, j-1] & \text{if } x_i = y_j \\ \max(LCS[i-1, j], LCS[i, j-1]) & \text{otherwise} \end{cases}$$

i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

We can follow the arrows to generate the solution

BCBA

101