Dynamic Programming II

CS/MCS 401: Computer Algorithms I
Lecture 19
Professor: Jonathan Liu

Icebreaker: What do you talk about most?

Announcements

Both HW4 and Project Phase 1 are due tomorrow! I want to release the inputs as soon as possible so you have as much time to work on them, so please make sure to submit Phase 1 by tomorrow.

One-on-One Office Hours are now bookable. See Piazza post for more details.

Today's Overview

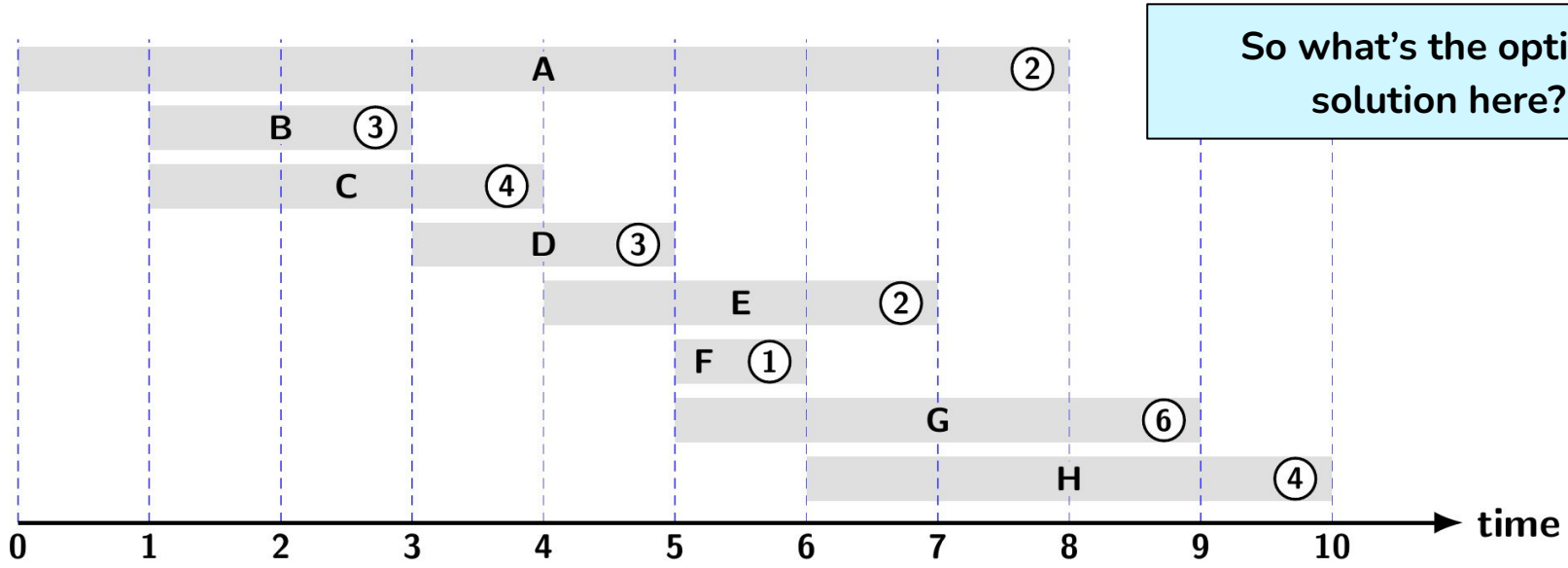
Today we're going to take a more careful look at the second dynamic programming problem, longest increasing subsequence, and see what the two algorithms tell us about Dynamic Programming in general.

Then, we'll see a new classic Dynamic Programming problem, with many similarities to the first two but some key differences as well.

Recap: Weighted Interval Scheduling

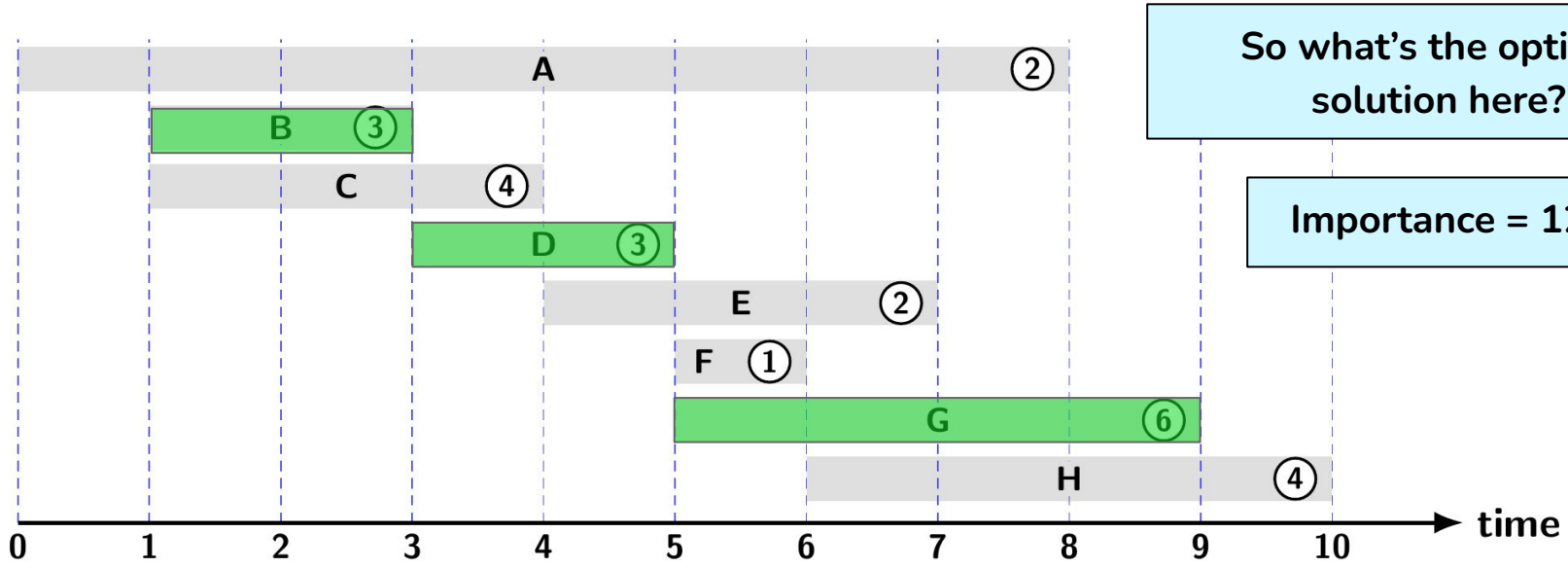
Weighted Course Scheduling

You're only willing to come onto campus one day per week. Each class has an importance level displayed. Find the feasible schedule with maximum importance.



Weighted Course Scheduling

You're only willing to come onto campus one day per week. Each class has an importance level displayed. Find the feasible schedule with maximum importance.



Weighted Course Scheduling: Putting it Together

Approach Idea

Define OPT(t) to be value of the best schedule ending at time t.

We want to find OPT(10).

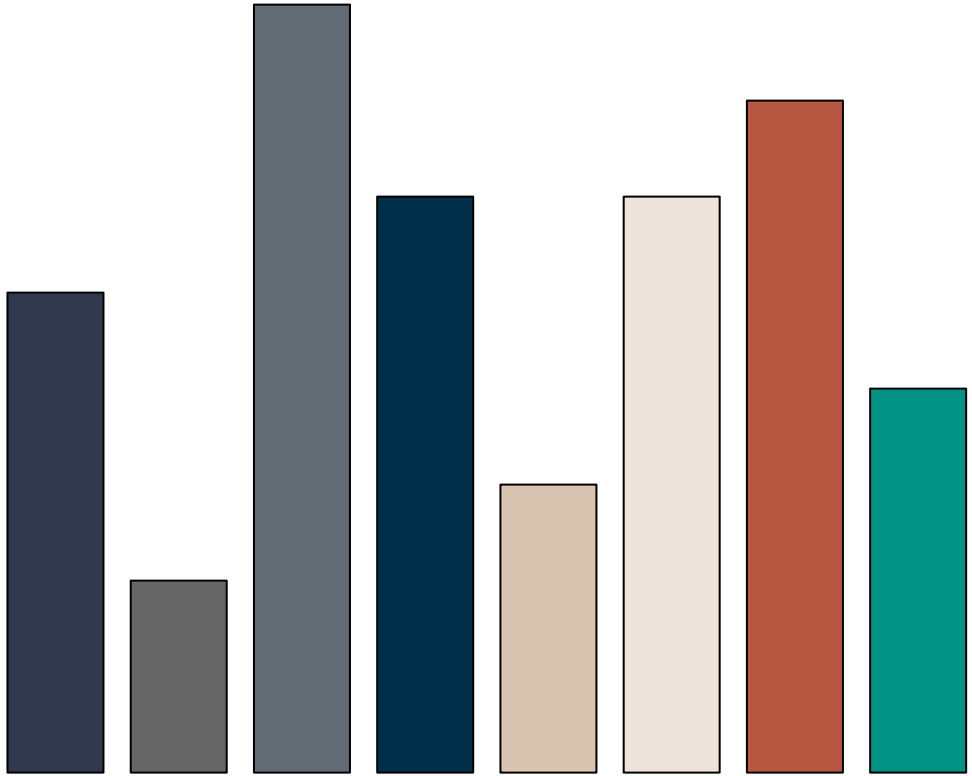
We can start with OPT(0).

If that class starts at s, ends at t, and has value v, then **OPT(t) = max(OPT(t-1), OPT(s) + v)**.

```
def WCS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 0
    for class in classes, sorted by end time:
        opt[t] = max(opt[t-1], opt[s] + v)
    return opt[n]
```

Another Dynamic Programming Algorithm

Reshelving Books

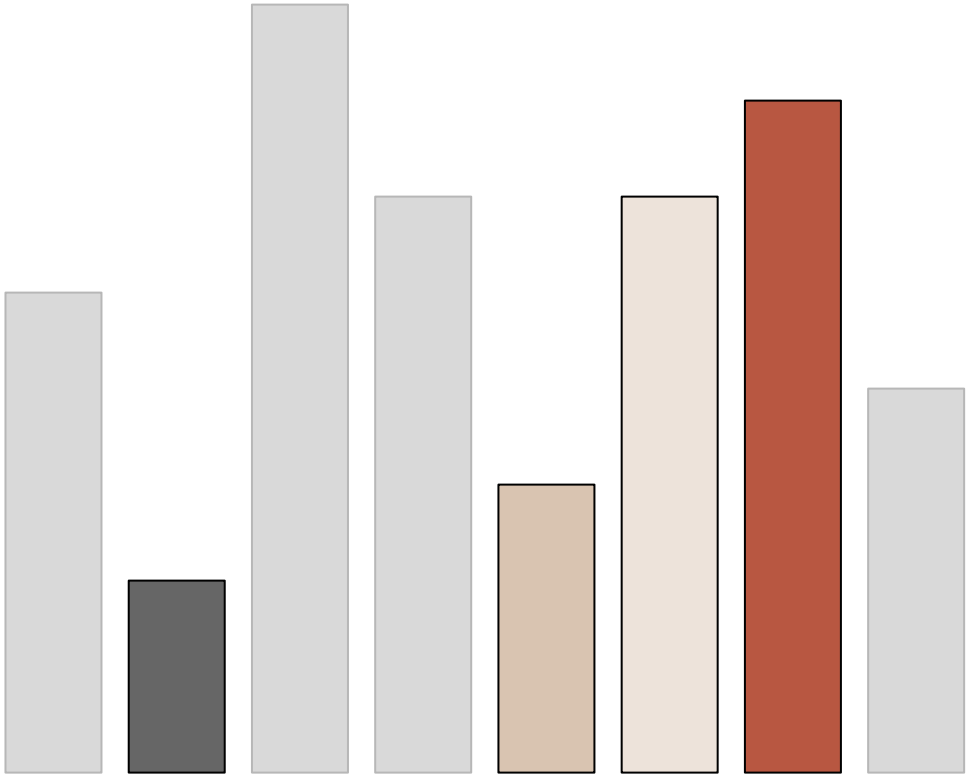


You have a lot of books, and you'd like to rearrange them so that they're in order by height.

The books are heavy, so you'd like to take as few books off the shelf as you can.

What's the largest number of books you can keep on the shelf?

Reshelving Books



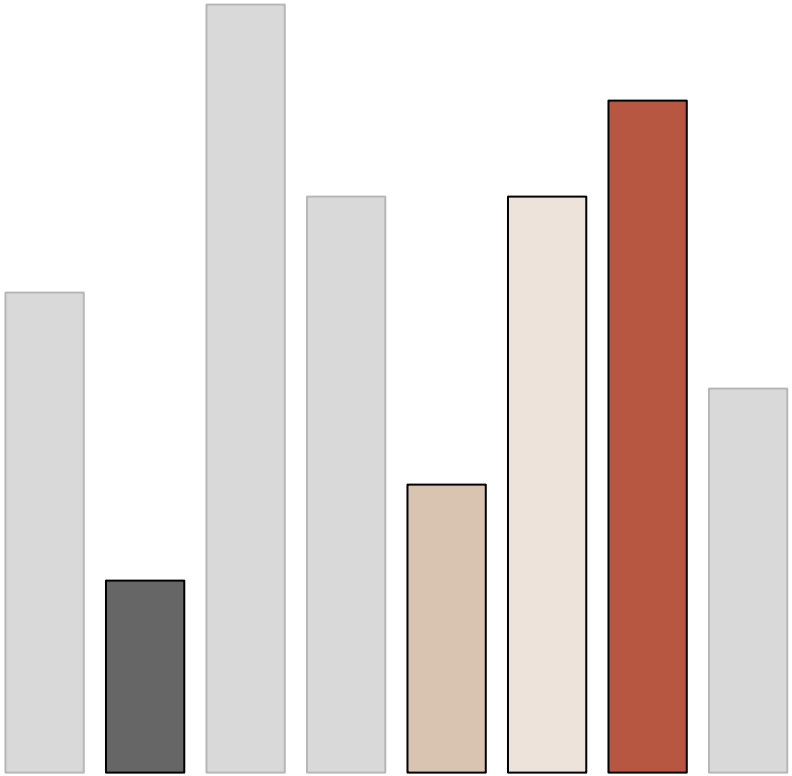
You have a lot of books, and you'd like to rearrange them so that they're in order by height.

The books are heavy, so you'd like to take as few books off the shelf as you can.

What's the largest number of books you can keep on the shelf?

It's the **longest increasing subsequence.**

Reshelving Books



```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

Take the max opt[i] value among all i where i < j and A[i] < A[j].

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]											

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]											

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1										

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1										

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1									

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2								

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2							

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2						

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3					

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4				

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4			

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3		

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3	3	

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

This is the max opt[i] value among all i where i < j and A[i] < A[j]. If there are no valid i values, this is 0.

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3	3	2

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

So what does OPT(j) mean?

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3	3	2

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

Heights (A)

5, 2, 8, 6, 3, 6, 9, 7, 5, 4, 3

OPT(j): Length of longest increasing subsequence that ends at index j.

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3	3	2

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

OPT(j): Length of longest increasing subsequence that ends at index j.

What decision are we making?

We're looking for the longest increasing subsequence we can add term j to.

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3	3	2

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

OPT(j): Length of longest increasing subsequence that ends at index j.

Why is opt[0] = 1?

The term at index 0 is, by itself, a subsequence of length 1.

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3	3	2

Understanding the Pseudocode

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

OPT(j): Length of longest increasing subsequence that ends at index j.

Why don't we return opt[n]?

We want the best subsequence, not one that ends with the last term.

j	0	1	2	3	4	5	6	7	8	9	10
A[j]	5	2	8	6	3	6	9	7	5	4	3
OPT[j]	1	1	2	2	2	3	4	4	3	3	2

Dynamic Programming Algorithm Design Process

Dynamic Programming Algorithm Structure

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

```
def WCS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 0
    for class in classes, sorted by end time:
        opt[t] = max(opt[t-1], opt[s] + v)
    return opt[n]
```

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

```
def WCS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 0
    for class in classes, sorted by end time:
        opt[t] = max(opt[t-1], opt[s] + v)
    return opt[n]
```

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

```
def WCS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 0
    for class in classes, sorted by end time:
        opt[t] = max(opt[t-1], opt[s] + v)
    return opt[n]
```

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

```
def WCS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 0
    for class in classes, sorted by end time:
        opt[t] = max(opt[t-1], opt[s] + v)
    return opt[n]
```

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

```
def WCS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 0
    for class in classes, sorted by end time:
        opt[t] = max(opt[t-1], opt[s] + v)
    return opt[n]
```

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

```
def LIS(A):
    n = len(A)
    opt = array of length n
    opt[0] = 1
    for j in range(1, n):
        opt[j] = 1 + max(opt[i]: i < j, A[i] < A[j])
    return max(opt)
```

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

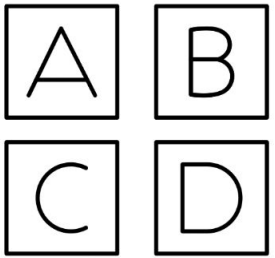
Design Order ≠ Implementation Order!

Edit Distance

Problem 5: Sign Editing

You need to update your block-letter storefront sign with your new promotion. Every letter added or removed takes some time. Find the minimum number of additions and removals that need to be made to update the sign.





Problem Modeling

What is the input data?
The current message and the desired new message.

What is the output data?
The minimum number of additions and removals needed to update the sign.

Edit Distance Problem

When you make a typo, spellcheck tries to figure out the “closest” words to what you typed. What is the correct measure of “closeness”?

There are three edits you can take: add a letter, remove a letter, or replace a letter. Note that letters can be slid around for free. Find the minimum number of edits you need to make to convert one word into the other.

P	L	A	N	E

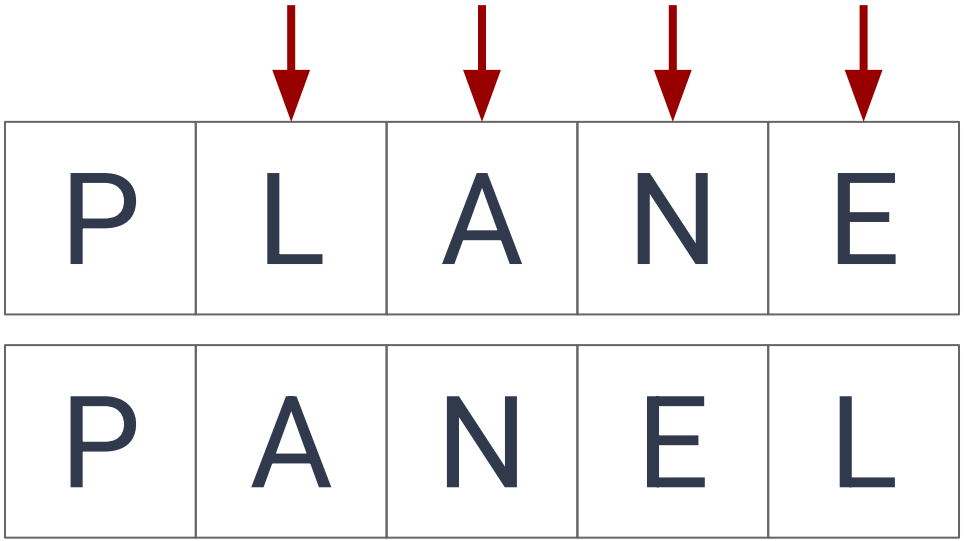
P	A	N	E	L

<u>Other applications</u>

- DNA/Protein analysis
- Speech Recognition
- Translation

Edit Distance Problem

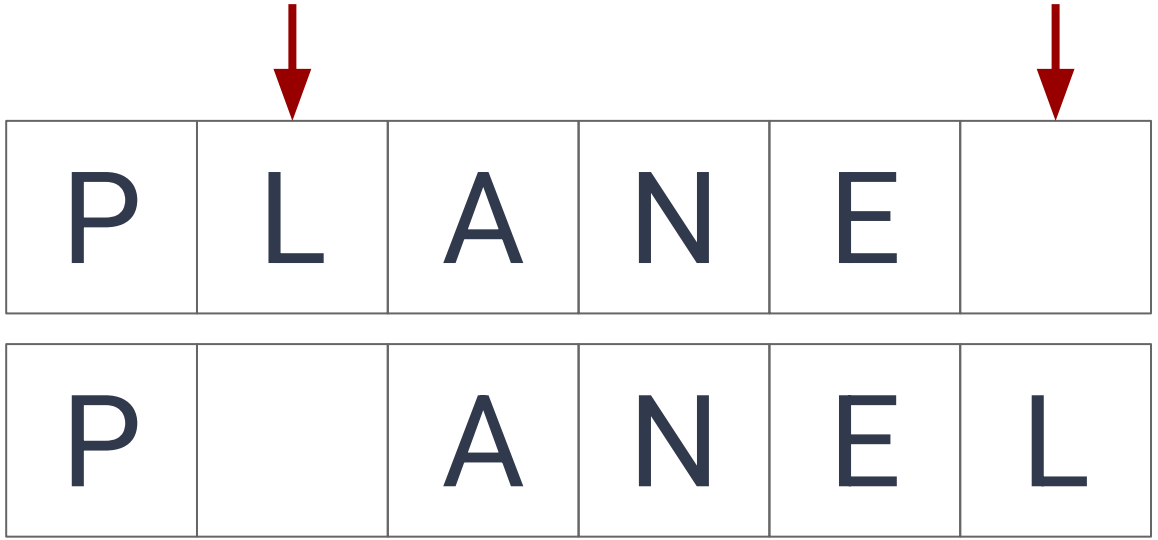
There are three edits you can take: add a letter, remove a letter, or replace a letter. Note that letters can be slid around for free. Find the minimum number of edits you need to make to convert one word into the other.



Distance = 4: Change four letters.

Edit Distance Problem

There are three edits you can take: add a letter, remove a letter, or replace a letter. Note that letters can be slid around for free. Find the minimum number of edits you need to make to convert one word into the other.



Distance = 2: Remove one letter, add one letter.

Edit Distance Example

There are three edits you can take: add a letter, remove a letter, or replace a letter. Note that letters can be slid around for free. Find the minimum number of edits you need to make to convert one word into the other.

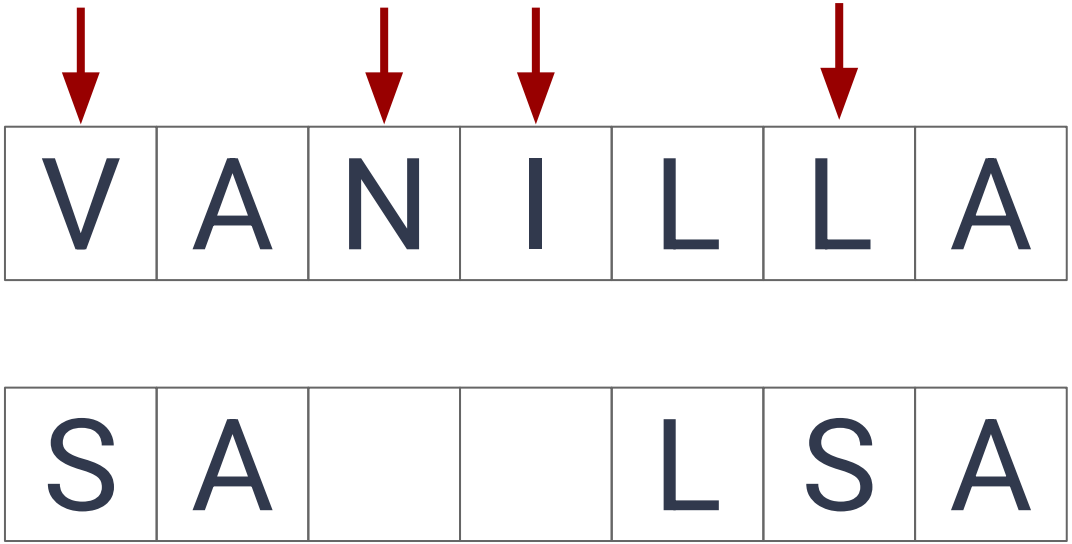
V	A	N	I	L	L	A

S	A	L	S	A

Distance = ?

Edit Distance Example

There are three edits you can take: add a letter, remove a letter, or replace a letter. Note that letters can be slid around for free. Find the minimum number of edits you need to make to convert one word into the other.



Distance = 4

Constructing an Algorithm

Dynamic Programming Algorithm Structure

Longest Increasing Subsequence
OPT(j): Length of longest increasing subsequence that ends at index j.

Weighted Interval Scheduling
OPT(t): Weight of best schedule that ends by time t

What is similar between these two?
Smaller versions of the original problem!

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

Longest Increasing Subsequence
OPT(j): Length of longest increasing subsequence that ends at index j.

Weighted Interval Scheduling
OPT(t): Weight of best schedule that ends by time t

First Attempt at a Subproblem:
OPT(i) = edit distance between first i letters of A and first i letters of B.

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

Longest Increasing Subsequence
OPT(j): Length of longest increasing subsequence that ends at index j.

Weighted Interval Scheduling
OPT(t): Weight of best schedule that ends by time t

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

OPT[4][3] = Edit Distance of:

V	A	N	I

S	A	L

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

Longest Increasing Subsequence
OPT(j) = best increasing subsequence that includes entry j

Weighted Interval Scheduling
OPT(t) = decide whether or not to take the class that ends at time t

What is similar between these two?
Decides between all possibilities for the last entry.

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

What does a recurrence do?
Decides between all possibilities for the last entry.

What are the possibilities for A[i] and B[j]?

V	A	N	I
S	A	L	

Design Steps for Dynamic Programming

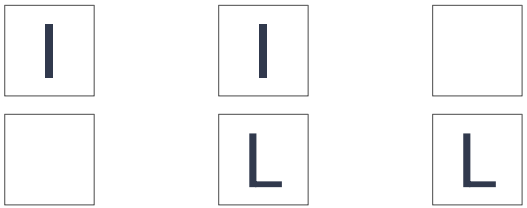
1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

What does a recurrence do?
Decides between all possibilities for the last entry.

What are the possibilities for A[i] and B[j]?



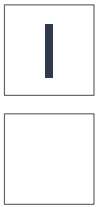
Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

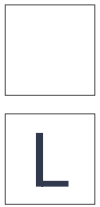
Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

For each of these, what is the edit distance?



1 + OPT[i-1][j]



1 + OPT[i][j-1]



1 + OPT[i-1][j-1] if A[i]≠B[j]
0 + OPT[i-1][j-1] if A[i]=B[j]

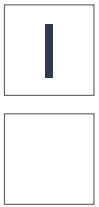
Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

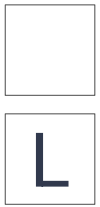
Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

For each of these, what is the edit distance?



1 + OPT[i-1][j]



1 + OPT[i][j-1]



1 + OPT[i-1][j-1] if A[i]≠B[j]
0 + OPT[i-1][j-1] if A[i]=B[j]

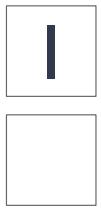
Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

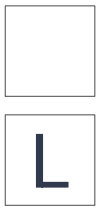
Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

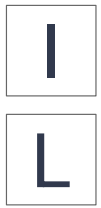
For each of these, what is the edit distance?



1 + OPT[i-1][j]



1 + OPT[i][j-1]



1 + OPT[i-1][j-1] if A[i]≠B[j]
0 + OPT[i-1][j-1] if A[i]=B[j]

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(

1 + OPT[i-1][j], // Remove last letter of A

1 + OPT[i][j-1], // Insert last letter of B

1 + OPT[i-1][j-1] if A[i]≠B[j], // replace

0 + OPT[i-1][j-1] if A[i]=B[j] // match

)

Base Case: What is the first value in opt?

		S	A	L	S	A
V						
A						
N						
I						
L						
L						
A						

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Base Case: What is the first value in opt?

		S	A	L	S	A
	0					
V						
A						
N						
I						
L						
L						
A						

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Base Case: What is the first value in opt?

		S	A	L	S	A
	0	1				
V						
A						
N						
I						
L						
L						
A						

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(

1 + OPT[i-1][j], // Remove last letter of A

1 + OPT[i][j-1], // Insert last letter of B

1 + OPT[i-1][j-1] if A[i]≠B[j], // replace

0 + OPT[i-1][j-1] if A[i]=B[j] // match

)

Base Case: What is the first value in opt?

		S	A	L	S	A
	0	1	2	3	4	5
V						
A						
N						
I						
L						
L						
A						

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Base Case: What is the first value in opt?
All of the first row and first column.

		S	A	L	S	A
	0	1	2	3	4	5
V	1					
A	2					
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt?

		S	A	L	S	A
	0	1	2	3	4	5
V	1					
A	2					
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt?

		S	A	L	S	A
	0	1	2	3	4	5
V	1					
A	2					
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1				
A	2					
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1				
A	2					
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1	2	3	4	5
A	2	2				
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1	2	3	4	5
A	2	2	1			
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1	2	3	4	5
A	2	2	1	2	3	
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1	2	3	4	5
A	2	2	1	2	3	4
N	3					
I	4					
L	5					
L	6					
A	7					

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1	2	3	4	5
A	2	2	1	2	3	4
N	3	3	2	2	3	4
I	4	4	3	3	3	4
L	5	5	4	3	4	4
L	6	6	5	4	4	5
A	7	7	6	5	5	4

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Return Value: What do you return after the array has been filled in?

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1	2	3	4	5
A	2	2	1	2	3	4
N	3	3	2	2	3	4
I	4	4	3	3	3	4
L	5	5	4	3	4	4
L	6	6	5	4	4	5
A	7	7	6	5	5	4

Dynamic Programming Algorithm Structure

2D Subproblem:

OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

OPT[i][j] = min(
1 + OPT[i-1][j], // Remove last letter of A
1 + OPT[i][j-1], // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j], // replace
0 + OPT[i-1][j-1] if A[i]=B[j] // match
)

Return Value: What do you return after the array has been filled in? *Bottom right*.

		S	A	L	S	A
	0	1	2	3	4	5
V	1	1	2	3	4	5
A	2	2	1	2	3	4
N	3	3	2	2	3	4
I	4	4	3	3	3	4
L	5	5	4	3	4	4
L	6	6	5	4	4	5
A	7	7	6	5	5	4

Dynamic Programming Algorithm Structure

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence Relationship

```
OPT[i][j] = min(
1 + OPT[i-1][j],          // Remove last letter of A
1 + OPT[i][j-1],          // Insert last letter of B
1 + OPT[i-1][j-1] if A[i]≠B[j],     // replace
0 + OPT[i-1][j-1] if A[i]=B[j]      // match
)
```

Base Case: What is the first value in opt? *All of the first row and first column.*

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

Return Value: What do you return after the array has been filled in? *Bottom right.*

Edit Distance Pseudocode

```
def EditDistance(A, B):
    m, n = lengths of A and B
    Initialize A[][] to be a 2D m x n array
    A[0][0] = 0.
    for all i: A[i][0] = i-1.
    for all j: A[0][j] = j-1.
    for i in range(m):
        for j in range(n):
            OPT[i][j] = [the recurrence]
    return OPT[m][n]
```

2D Subproblem:
OPT[i][j] = edit distance between first i letters of A and first j letters of B.

Recurrence: See prior slide.

Base Case: What is the first value in opt? *All of the first row and first column.*

Iteration Order: In what order do you fill in opt? *One row (or column) at a time.*

Return Value: What do you return after the array has been filled in? *Bottom right.*

A note about designing these algorithms

One of the most difficult parts of designing dynamic programming algorithms is that there is not a good way to check whether any of these are right until the whole algorithm is designed.

You can't write a proof until you've designed the whole algorithm, which means you can't check your work until then.

Your first guess for any of these steps will be wrong sometimes, and that's normal!

Design Steps for Dynamic Programming

1. **Define Subproblems:** Pick what opt[i] represents.
2. **Create a Recurrence:** How do you use prior values of opt to compute a new value of opt?
3. **Iteration Order:** In what order do you fill in opt?
4. **Base Case:** What is the first value in opt?
5. **Return Value:** What do you return after the array has been filled in?

Lecture Recap

Today we saw two new examples of dynamic programming algorithms. We identified some key similarities between different algorithms, but also saw some really stark differences.

Using the similarities, we introduced a design process for designing these algorithms. Although it'll typically follow these five steps, that doesn't make it easy. After all, there isn't a great way to check whether you've made the right decisions for each of the steps until the end of the process.

It's important to remember that this is natural and that resilience is important here!