

Session 4

Corbet Elkins

October 23, 2026

Purdue University

Questions over lecture content?

See title

Thoughts on DP

Recall from last week I said “DP is recursion with caching”

Thoughts on DP

Recall from last week I said “DP is recursion with caching”

However, in this class we don’t implement DP as recursion, but rather with a “bottom-up” approach, i.e. filling a table

Nonetheless, I still encourage you to think about DP as recursion with caching, once you have the recursive algorithm, it is straight forward to implement caching

Corbet's step-by-step guide for solving DP

1. Try to think about what information from a smaller subproblem would be helpful for solving a larger problem.
2. Using the information you want, implement a recursive algorithm with a *recursive spec* for information you want.
3. Actually write the code that implements the recursive spec.
4. Figure out how to implement caching, most of the time this will be some table with the various function inputs. Also figure out how to fill the table.
5. Runtime will almost always be $\# \text{subproblems} \cdot \text{runtime for each subproblem}$
6. Your recursive spec should be your inductive hypothesis for the proof, so make sure it is strong enough.

Example Problem

Problem

A sequence $a_1, \dots, a_k$ is a *palindrome* if the reversed sequence is the same; i.e. $(a_1, \dots, a_k) = (a_k, \dots, a_1)$. Compute the length of the longest palindrome subsequence of A .

1. What information from a smaller subproblem would be helpful for solving this?

Example Problem

Problem

A sequence $a_1, \dots, a_k$ is a *palindrome* if the reversed sequence is the same; i.e. $(a_1, \dots, a_k) = (a_k, \dots, a_1)$. Compute the length of the longest palindrome subsequence of A .

1. What information from a smaller subproblem would be helpful for solving this?

Presumably we want some information about the longest palindrome subsequence for some subarray of A .

Example Problem: Recursive Spec

2. What should our recursive spec be, remember this just specifies the inputs and what we want our function to output.

Example Problem: Recursive Spec

2. What should our recursive spec be, remember this just specifies the inputs and what we want our function to output.

Recursive Spec for LPS: Given a subarray $A[i..j]$, return the length of the longest palindrome subsequence of $A[i..j]$.

Example Problem: Recursive Spec

2. What should our recursive spec be, remember this just specifies the inputs and what we want our function to output.

Recursive Spec for LPS: Given a subarray $A[i..j]$, return the length of the longest palindrome subsequence of $A[i..j]$.

3. How do we implement this recursive spec? how can we reduce this problem to subproblems?

Example Problem: Recursive Spec

2. What should our recursive spec be, remember this just specifies the inputs and what we want our function to output.

Recursive Spec for LPS: Given a subarray $A[i..j]$, return the length of the longest palindrome subsequence of $A[i..j]$.

3. How do we implement this recursive spec? how can we reduce this problem to subproblems?

Hint: consider the cases when $A[i]$ is and is not part of the longest palindrome subsequence of $A[i..j]$.

Example Problem: Function Implementation

We split this into 2 cases based on $A[i]$:

Example Problem: Function Implementation

We split this into 2 cases based on $A[i]$: If $A[i]$ is not part of the longest palindrome subsequence of $A[i..j]$, then we return $LPS(A[i..j])$.

Example Problem: Function Implementation

We split this into 2 cases based on $A[i]$: If $A[i]$ is not part of the longest palindrome subsequence of $A[i..j]$, then we return $LPS(A[i..j])$.

If $A[i]$ is part of the longest palindrome subsequence, then we have two cases, if $A[j] = A[i]$, then we return $LPS(A[i + 1..j - 1]) + 2$, otherwise $A[j]$ is not in the LPS of $A[i..j]$ and we return $LPS(A[i..j - 1])$.

Example Problem: Function Implementation

We split this into 2 cases based on $A[i]$: If $A[i]$ is not part of the longest palindrome subsequence of $A[i..j]$, then we return $LPS(A[i..j])$.

If $A[i]$ is part of the longest palindrome subsequence, then we have two cases, if $A[j] = A[i]$, then we return $LPS(A[i + 1..j - 1]) + 2$, otherwise $A[j]$ is not in the LPS of $A[i..j]$ and we return $LPS(A[i..j - 1])$.

$$LPS(A[i..j]) = \begin{cases} \max\{LPS(A[i + 1..j]), LPS(A[i..j - 1])\} & A[i] \neq A[j] \\ 2 + LPS(A[i + 1..j - 1]) & A[i] = A[j] \end{cases}$$

Example Problem: Function Implementation

We split this into 2 cases based on $A[i]$: If $A[i]$ is not part of the longest palindrome subsequence of $A[i..j]$, then we return $LPS(A[i..j])$.

If $A[i]$ is part of the longest palindrome subsequence, then we have two cases, if $A[j] = A[i]$, then we return $LPS(A[i + 1..j - 1]) + 2$, otherwise $A[j]$ is not in the LPS of $A[i..j]$ and we return $LPS(A[i..j - 1])$.

$$LPS(A[i..j]) = \begin{cases} \max\{LPS(A[i + 1..j]), LPS(A[i..j - 1])\} & A[i] \neq A[j] \\ 2 + LPS(A[i + 1..j - 1]) & A[i] = A[j] \end{cases}$$

What's the base case? What do we return here?

Example Problem: Function Implementation

We split this into 2 cases based on $A[i]$: If $A[i]$ is not part of the longest palindrome subsequence of $A[i..j]$, then we return $LPS(A[i..j])$.

If $A[i]$ is part of the longest palindrome subsequence, then we have two cases, if $A[j] = A[i]$, then we return $LPS(A[i + 1..j - 1]) + 2$, otherwise $A[j]$ is not in the LPS of $A[i..j]$ and we return $LPS(A[i..j - 1])$.

$$LPS(A[i..j]) = \begin{cases} \max\{LPS(A[i + 1..j]), LPS(A[i..j - 1])\} & A[i] \neq A[j] \\ 2 + LPS(A[i + 1..j - 1]) & A[i] = A[j] \end{cases}$$

What's the base case? What do we return here? $i = j$, we return 1.

Example Problem: Caching and Runtime

4. The only information we really need to pass in for the function is i, j , thus we can store the result of LPS in a table, T , where $LPS(A[i..j]) = T[i, j]$.

We can fill the table by $|i - j|$, that is we fill the diagonal (base case) and then one off the diagonal, and so forth.

5. How many problem instances are there?

Example Problem: Caching and Runtime

4. The only information we really need to pass in for the function is i, j , thus we can store the result of LPS in a table, T , where $LPS(A[i..j]) = T[i, j]$.

We can fill the table by $|i - j|$, that is we fill the diagonal (base case) and then one off the diagonal, and so forth.

5. How many problem instances are there? $O(n^2)$

Example Problem: Caching and Runtime

4. The only information we really need to pass in for the function is i, j , thus we can store the result of LPS in a table, T , where $LPS(A[i..j]) = T[i, j]$.

We can fill the table by $|i - j|$, that is we fill the diagonal (base case) and then one off the diagonal, and so forth.

5. How many problem instances are there? $O(n^2)$

How long does it take to solve each subproblem?

Example Problem: Caching and Runtime

4. The only information we really need to pass in for the function is i, j , thus we can store the result of LPS in a table, T , where $LPS(A[i..j]) = T[i, j]$.

We can fill the table by $|i - j|$, that is we fill the diagonal (base case) and then one off the diagonal, and so forth.

5. How many problem instances are there? $O(n^2)$

How long does it take to solve each subproblem? $O(1)$

Example Problem: Proof of Correctness

Almost all DP problems we will prove by induction, with the number we are inducting on is how we fill the table.

Base Case:

Example Problem: Proof of Correctness

Almost all DP problems we will prove by induction, with the number we are inducting on is how we fill the table.

Base Case:

$i = j$, then $LPS(A[i..i]) = 1$, this is obvious

Example Problem: Proof of Correctness

Almost all DP problems we will prove by induction, with the number we are inducting on is how we fill the table.

Base Case:

$i = j$, then $LPS(A[i..i]) = 1$, this is obvious

Inductive Hypothesis: Suppose $LPS(A[i'..j'])$ returns the LPS for $A[i'..j']$ for $|j' - i'| < |j - i|$

Example Problem: Proof of Correctness

Almost all DP problems we will prove by induction, with the number we are inducting on is how we fill the table.

Base Case:

$i = j$, then $LPS(A[i..i]) = 1$, this is obvious

Inductive Hypothesis: Suppose $LPS(A[i'..j'])$ returns the LPS for $A[i'..j']$ for $|j' - i'| < |j - i|$

Inductive step:

We already did this when we reasoned through the possible cases on if $A[i]$ is or is not in the LPS of $A[i..j]$.

Note on LIS

In lecture Mike talks about doing some sweep operation, to increase LIS runtime, I find this to be a bit confusing.

Our DP relation is

$$LIS(A[i..n]) = \max(\{LIS(A[j..n]) + 1 : A[j] > A[i]\} \cup \{1\})$$

Note on LIS

In lecture Mike talks about doing some sweep operation, to increase LIS runtime, I find this to be a bit confusing.

Our DP relation is

$$LIS(A[i..n]) = \max(\{LIS(A[j..n]) + 1 : A[j] > A[i]\} \cup \{1\})$$

We can brute force this calculation which would take $O(n)$ time, giving us an $O(n^2)$ runtime.

Note on LIS

In lecture Mike talks about doing some sweep operation, to increase LIS runtime, I find this to be a bit confusing.

Our DP relation is

$$LIS(A[i..n]) = \max(\{LIS(A[j..n]) + 1 : A[j] > A[i]\} \cup \{1\})$$

We can brute force this calculation which would take $O(n)$ time, giving us an $O(n^2)$ runtime.

The idea to speed this up to binary search on “good” indices so we can determine the optimal in $O(\log n)$ instead.

Speeding up LIS with Binary Search

Definition: We say an index i *dominates* j if $x_j < x_i$ and $LIS(A[j..n]) \leq LIS(A[i..n])$.

We edit our LIS recursive spec to the following: $LIS(A[i..n])$ return the LIS of $A[i..n]$ which begins with $A[i]$ and a sorted list of $(x_j, LIS(A[j..n]))$ such that no index dominates another.

Speeding up LIS with Binary Search

Definition: We say an index i *dominates* j if $x_j < x_i$ and $LIS(A[j..n]) \leq LIS(A[i..n])$.

We edit our LIS recursive spec to the following: $LIS(A[i..n])$ return the LIS of $A[i..n]$ which begins with $A[i]$ and a sorted list of $(x_j, LIS(A[j..n]))$ such that no index dominates another.

Then to find the $\max\{LIS(A[j..n]) + 1 : A[j] > A[i]\}$, we simply perform binary search on the shortest list. Call this sorted list ℓ , note that $\ell[k][1] > \ell[k+1][1]$. Then we insert $(x_i, LIS(A[i..n]))$ into the list and delete elements such that no element dominates i or vice versa. If we use a BST for this it is $O(\log(n))$ amortized time.