Dynamic Programming

Lecture 13
March 2, 2017
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question:
Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$.

$\text{foo}(x)$ spends at most $B(n)$ time not counting the time for its recursive calls.

Suppose we memoize the recursion.

Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

Question: What is an upper bound on the running time of the memoized version of $\text{foo}(x)$ if $|x| = n$?

$O(A(n)B(n))$. 
Dynamic Programming is smart recursion plus memoization

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time *not counting* the time for its recursive calls.

Suppose we memoize the recursion.
Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

**Question:** What is an upper bound on the running time of memoized version of $\text{foo}(x)$ if $|x| = n$?

$O(A(n)B(n))$. 
Dynamic Programming

Dynamic Programming is **smart recursion** plus memoization

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time *not counting* the time for its recursive calls.

Suppose we *memoize* the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

**Question:** Suppose we have a recursive program $\text{foo}(x)$ that takes an input $x$.

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time not counting the time for its recursive calls.

Suppose we *memoize* the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

**Question:** What is an upper bound on the running time of *memoized* version of $\text{foo}(x)$ if $|x| = n$?
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

**Question:** Suppose we have a recursive program \( \text{foo}(x) \) that takes an input \( x \).

- On input of size \( n \) the number of distinct sub-problems that \( \text{foo}(x) \) generates is at most \( A(n) \).
- \( \text{foo}(x) \) spends at most \( B(n) \) time *not counting* the time for its recursive calls.

Suppose we memoize the recursion.

**Assumption:** Storing and retrieving solutions to pre-computed problems takes \( O(1) \) time.

**Question:** What is an upper bound on the running time of memoized version of \( \text{foo}(x) \) if \(|x| = n\)? \( O(A(n)B(n)) \).
Part I

Checking if string is in $L^*$
Problem

**Input** A string \( w \in \Sigma^* \) and access to a language \( L \subseteq \Sigma^* \) via function \( \text{IsStrInL}(\text{string } x) \) that decides whether \( x \) is in \( L \)

**Goal** Decide if \( w \in L^* \) using \( \text{IsStrInL}(\text{string } x) \) as a black box sub-routine

**Example**

Suppose \( L \) is \textit{English} and we have a procedure to check whether a string/word is in the \textit{English} dictionary.

- Is the string “isthisinanenglishsentence” in \textit{English} star? 
- Is “stampstamp” in \textit{English} star? 
- Is “zibzzzzad” in \textit{English} star?
Recursive Solution

When is $w \in L^*$?
Recursive Solution

When is $w \in L^*$?

A $w \in L^*$ if $w \in L$ or if $w = uv$ where $u \in L$ and $v \in L^*$, $|u| \geq 1$.
Recursive Solution

When is \( w \in L^* \)?

A \( w \in L^* \) if \( w \in L \) or if \( w = uv \) where \( u \in L \) and \( v \in L^* \), \( |u| \geq 1 \)

Assume \( w \) is stored in array \( A[1..n] \)

```plaintext
IsStringinLstar(A[1..n]):

If (IsStrInL(A[1..n]))
    Output YES
Else
    For (i = 1 to n − 1) do
        If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
            Output YES
    Output NO
```

Chandra Chekuri (UIUC)  CS374  Spring 2017
Recursive Solution

Assume $w$ is stored in array $A[1..n]

```plaintext
IsStringinLstar(A[1..n]):
  If (IsStrInL(A[1..n]))
    Output YES
  Else
    For ($i = 1$ to $n - 1$) do
      If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
        Output YES
    Output NO
```

Question: How many distinct sub-problems does $\text{IsStrInLstar}(A[1..n])$ generate?

$O(n)$
Recursive Solution

Assume $w$ is stored in array $A[1..n]$

\begin{align*}
\text{IsStringinLstar}(A[1..n]) : \\
\text{If (IsStrInL}(A[1..n])) \\
\text{Output YES} \\
\text{Else} \\
\text{For (} & i = 1 \text{ to } n - 1 \text{ do} \\
\text{If (IsStrInL}(A[1..i]) \text{ and IsStrInLstar}(A[i + 1..n])) \\
\text{Output YES} \\
\text{Output NO}
\end{align*}

**Question:** How many distinct sub-problems does \(\text{IsStrInLstar}(A[1..n])\) generate?
Recursive Solution

Assume \( w \) is stored in array \( A[1..n] \)

\[
\text{IsStringinLstar}(A[1..n]) :
\]
\[
\text{If (IsStrInL}(A[1..n]))
\]
\[
\text{Output YES}
\]
\[
\text{Else}
\]
\[
\text{For (} i = 1 \text{ to } n - 1 \text{) do}
\]
\[
\text{If (IsStrInL}(A[1..i]) \text{ and IsStrInLstar}(A[i + 1..n]))
\]
\[
\text{Output YES}
\]
\[
\text{Output NO}
\]

**Question:** How many distinct sub-problems does \( \text{IsStrInLstar}(A[1..n]) \) generate? \( O(n) \)
Example

Consider string *samiam*
After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

**ISL($i$):** a boolean which is 1 if $A[i..n]$ is in $L^*$, 0 otherwise

**Base case:** $ISL(n + 1) = 1$ interpreting $A[n + 1..n]$ as $\epsilon$
Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

**ISL(i):** a boolean which is 1 if $A[i..n]$ is in $L^*$, 0 otherwise

**Base case:** $ISL(n + 1) = 1$ interpreting $A[n + 1..n]$ as $\epsilon$

**Recursive relation:**

- $ISL(i) = 1$ if $\exists i < j \leq n + 1$ s.t $ISL(j)$ and $IsStrInL(A[i..(j - 1)])$
- $ISL(i) = 0$ otherwise
Naming subproblems and recursive equation

After seeing that number of subproblems is \( O(n) \) we name them to help us understand the structure better.

**ISL\((i)\):** a boolean which is 1 if \( A[i..n] \) is in \( L^* \), 0 otherwise

**Base case:** \( ISL(n + 1) = 1 \) interpreting \( A[n + 1..n] \) as \( \epsilon \)

**Recursive relation:**

- \( ISL(i) = 1 \) if \( \exists i < j \leq n + 1 \) s.t \( ISL(j) \) and \( IsStrInL(A[i..(j - 1)]) \)
- \( ISL(i) = 0 \) otherwise

**Output:** \( ISL(1) \)
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why?
Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an *iterative* algorithm via *explicit memoization* and *bottom up* computation.

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.
Removing recursion to obtain iterative algorithm

Typically, after finding a dynamic programming recursion, we often convert the recursive algorithm into an iterative algorithm via explicit memoization and bottom up computation.

Why? Mainly for further optimization of running time and space.

How?
- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.

Caveat: Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
Iterative Algorithm

**Iterative Algorithm**

\[
\text{IsStringInLstar-Iterative}(A[1..n]) : \\
\text{boolean } \text{ISL}[1..(n + 1)] \\
\text{ISL}[n + 1] = \text{TRUE} \\
\text{for } (i = n \text{ down to } 1) \\
\phantom{\text{ISL}[i]} = \text{FALSE} \\
\text{for } (j = i + 1 \text{ to } n + 1) \\
\phantom{\text{ISL}[i]} \text{If (ISL}[j] \text{ and } \text{IsStrInL}(A[i..j])) \\
\phantom{\text{ISL}[i]} = \text{TRUE} \\
\text{If (ISL}[1] = 1) \text{ Output YES} \\
\text{Else Output NO}
\]
Iterative Algorithm

\textbf{IsStringinLstar-Iterative}(A[1..n]):

\begin{itemize}
  \item boolean ISL[1..(n + 1)]
  \item ISL[n + 1] = \text{TRUE}
  \item for (i = n down to 1)
    \begin{itemize}
      \item ISL[i] = \text{FALSE}
      \item for (j = i + 1 to n + 1)
        \begin{itemize}
          \item If (ISL[j] and IsStrInL(A[i..j]))
            \begin{itemize}
              \item ISL[i] = \text{TRUE}
            \end{itemize}
        \end{itemize}
    \end{itemize}
  \end{itemize}

  \begin{itemize}
    \item If (ISL[1] = 1) Output YES
    \item Else Output NO
  \end{itemize}
\end{itemize}

\textbf{Running time:}

$O(n^2)$ (assuming call to IsStrInL is $O(1)$ time)
Iterative Algorithm

\begin{align*}
\text{IsStringInLstar-Iterative}(A[1..n]) : \\
&\text{boolean } ISL[1..(n + 1)] \\
&ISL[n + 1] = TRUE \\
&\text{for } (i = n \text{ down to } 1) \\
&\quad ISL[i] = FALSE \\
&\quad \text{for } (j = i + 1 \text{ to } n + 1) \\
&\quad \quad \text{If } (ISL[j] \text{ and } \text{IsStrInL}(A[i..j])) \\
&\quad \quad \quad ISL[i] = TRUE \\
\end{align*}

\text{If } (ISL[1] = 1) \text{ Output YES} \\
\text{Else Output NO}

- **Running time:** \( O(n^2) \) (assuming call to IsStrInL is \( O(1) \) time)
Iterative Algorithm

\textbf{IsStringinLstar-Iterative}(A[1..n]):

\begin{itemize}
  \item boolean \text{ISL}[1..(n + 1)]
  \item \text{ISL}[n + 1] = \text{TRUE}
  \item for (i = n down to 1)
    \begin{itemize}
      \item \text{ISL}[i] = \text{FALSE}
      \item for (j = i + 1 to n + 1)
        \begin{itemize}
          \item If (\text{ISL}[j] \text{ and } \text{IsStrInL}(A[i..j]))
            \begin{itemize}
              \item \text{ISL}[i] = \text{TRUE}
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

If (\text{ISL}[1] = 1) Output YES
Else Output NO

- \textbf{Running time: } $O(n^2)$ (assuming call to \text{IsStrInL} is $O(1)$ time)
- \textbf{Space: }
Iterative Algorithm

\[ \text{IsStringInLstar-Iterative}(A[1..n]): } \]

\[
\begin{align*}
\text{boolean } & \text{ ISL}[1..(n + 1)] \\
\text{ISL}[n + 1] & = \text{ TRUE} \\
\text{for } (i = n \text{ down to 1}) \\
\text{ISL}[i] & = \text{ FALSE} \\
\text{for } (j = i + 1 \text{ to } n + 1) \\
\text{If } (\text{ISL}[j] \text{ and IsStrInL}(A[i..j])) \\
\text{ISL}[i] & = \text{ TRUE} \\
\text{If } (\text{ISL}[1] = 1) \text{ Output YES} \\
\text{Else Output NO}
\end{align*}
\]

- **Running time:** \( O(n^2) \) (assuming call to \( \text{IsStrInL} \) is \( O(1) \) time)
- **Space:** \( O(n) \)
Consider string *samiam*
Part II

Longest Increasing Subsequence
**Definition**

**Sequence**: an ordered list \(a_1, a_2, \ldots, a_n\). **Length** of a sequence is number of elements in the list.

**Definition**

\(a_{i_1}, \ldots, a_{i_k}\) is a **subsequence** of \(a_1, \ldots, a_n\) if 
\[1 \leq i_1 < i_2 < \ldots < i_k \leq n.\]

**Definition**

A sequence is **increasing** if \(a_1 < a_2 < \ldots < a_n\). It is **non-decreasing** if \(a_1 \leq a_2 \leq \ldots \leq a_n\). Similarly **decreasing** and **non-increasing**.
Sequences

Example...

1. Sequence: 6, 3, 5, 2, 7, 8, 1, 9
2. Subsequence of above sequence: 5, 2, 1
3. Increasing sequence: 3, 5, 9, 17, 54
4. Decreasing sequence: 34, 21, 7, 5, 1
5. Increasing subsequence of the first sequence: 2, 7, 9.
Longest Increasing Subsequence Problem

Input  A sequence of numbers $a_1, a_2, \ldots, a_n$

Goal  Find an increasing subsequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length
Longest Increasing Subsequence Problem

**Input**  A sequence of numbers $a_1, a_2, \ldots, a_n$

**Goal**  Find an *increasing subsequence* $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ of maximum length

**Example**

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
3. Longest increasing subsequence: 3, 5, 7, 8
Recursive Approach: Take 1

**LIS: Longest increasing subsequence**

Can we find a recursive algorithm for **LIS**?

**LIS**($A[1..n]$):

1. Case 1: Does not contain $A[n]$ in which case $\text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)])$

2. Case 2: contains $A[n]$ in which case $\text{LIS}(A[1..n])$ is not so clear.

Observation

For second case we want to find a subsequence in $A[1..(n-1)]$ that is restricted to numbers less than $A[n]$. This suggests that a more general problem is $\text{LIS}_{\text{smaller}}(A[1..n], x)$ which gives the longest increasing subsequence in $A$ where each number in the sequence is less than $x$. 
Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

$LIS(A[1..n])$:

1. **Case 1**: Does not contain $A[n]$ in which case
   \[
   LIS(A[1..n]) = LIS(A[1..(n-1)])
   \]

2. **Case 2**: contains $A[n]$ in which case $LIS(A[1..n])$ is not so clear.

**Observation**

For second case we want to find a subsequence in $A[1..(n-1)]$ that is restricted to numbers less than $A[n]$. This suggests that a more general problem is $LIS_{smaller}(A[1..n], x)$ which gives the longest increasing subsequence in $A$ where each number in the sequence is less than $x$. 
Recursive Approach

$LIS(A[1..n])$: the length of longest increasing subsequence in $A$

$LIS_{smaller}(A[1..n], x)$: length of longest increasing subsequence in $A[1..n]$ with all numbers in subsequence less than $x$

```
LIS_{smaller}(A[1..n], x):
    if $n = 0$ then return 0
    $m = LIS_{smaller}(A[1..(n-1)], x)$
    if $A[n] < x$ then
        $m = max(m, 1 + LIS_{smaller}(A[1..(n-1)], A[n]))$
    Output $m$
```

$LIS(A[1..n])$: return $LIS_{smaller}(A[1..n], \infty)$
Example

Sequence: $A[1..7] = 6, 3, 5, 2, 7, 8, 1$
Recursive Approach

\[
\text{LIS\_smaller}(A[1..n], x): \\
\quad \text{if } (n = 0) \text{ then return } 0 \\
\quad m = \text{LIS\_smaller}(A[1..(n - 1)], x) \\
\quad \text{if } (A[n] < x) \text{ then} \\
\quad \quad m = \max(m, 1 + \text{LIS\_smaller}(A[1..(n - 1)], A[n])) \\
\quad \text{Output } m
\]

\[
\text{LIS}(A[1..n]): \\
\quad \text{return } \text{LIS\_smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate?
Recursive Approach

\[
\text{LIS} \_\text{smaller}(A[1..n], x) :
\]
\[
\quad \text{if } (n = 0) \text{ then return } 0
\]
\[
\quad m = \text{LIS} \_\text{smaller}(A[1..(n - 1)], x)
\]
\[
\quad \text{if } (A[n] < x) \text{ then}
\]
\[
\quad \quad m = \max(m, 1 + \text{LIS} \_\text{smaller}(A[1..(n - 1)], A[n]))
\]
\[
\quad \text{Output } m
\]

\[
\text{LIS}(A[1..n]) :
\]
\[
\quad \text{return } \text{LIS} \_\text{smaller}(A[1..n], \infty)
\]

- How many distinct sub-problems will \text{LIS} \_\text{smaller}(A[1..n], \infty) generate? \(O(n^2)\)
Recursive Approach

$LIS_{smaller}(A[1..n], x)$:

if \( n = 0 \) then return 0

\[ m = LIS_{smaller}(A[1..(n - 1)], x) \]

if \( A[n] < x \) then

\[ m = \max(m, 1 + LIS_{smaller}(A[1..(n - 1)], A[n])) \]

Output \( m \)

$LIS(A[1..n])$:

return $LIS_{smaller}(A[1..n], \infty)$

- How many distinct sub-problems will $LIS_{smaller}(A[1..n], \infty)$ generate? $O(n^2)$
- What is the running time if we memoize recursion?
Recursive Approach

\[ \text{LIS}\_\text{smaller}(A[1..n], x) : \]
\[
\begin{align*}
\text{if } (n = 0) \text{ then return } 0 \\
 m &= \text{LIS}\_\text{smaller}(A[1..(n-1)], x) \\
\text{if } (A[n] < x) \text{ then} \\
 m &= \max(m, 1 + \text{LIS}\_\text{smaller}(A[1..(n-1)], A[n])) \\
\text{Output } m
\end{align*}
\]

\[ \text{LIS}(A[1..n]) : \\
\text{return } \text{LIS}\_\text{smaller}(A[1..n], \infty) \]

- How many distinct sub-problems will \text{LIS}\_\text{smaller}(A[1..n], \infty) generate? \( O(n^2) \)
- What is the running time if we memoize recursion? \( O(n^2) \) since each call takes \( O(1) \) time to assemble the answers from recursive calls and no other computation.
Recursive Approach

LIS_smaller(A[1..n], x):
  if (n = 0) then return 0
  m = LIS_smaller(A[1..(n − 1)], x)
  if (A[n] < x) then
    m = max(m, 1 + LIS_smaller(A[1..(n − 1)], A[n]))
  Output m

LIS(A[1..n]):
  return LIS_smaller(A[1..n], ∞)

How many distinct sub-problems will LIS_smaller(A[1..n], ∞) generate? $O(n^2)$

What is the running time if we memoize recursion? $O(n^2)$ since each call takes $O(1)$ time to assemble the answers from to recursive calls and no other computation.

How much space for memoization?
Recursive Approach

\[ \text{LIS}\_\text{smaller}(A[1..n], x) : \]
- if \((n = 0)\) then return 0
- \(m = \text{LIS}\_\text{smaller}(A[1..(n - 1)], x)\)
- if \((A[n] < x)\) then
  - \(m = \max(m, 1 + \text{LIS}\_\text{smaller}(A[1..(n - 1)], A[n]))\)
- Output \(m\)

\[ \text{LIS}(A[1..n]) : \]
- return \(\text{LIS}\_\text{smaller}(A[1..n], \infty)\)

- How many distinct sub-problems will \(\text{LIS}\_\text{smaller}(A[1..n], \infty)\) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from to recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add $\infty$ at end of array (in position $n + 1$)

$LIS(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)
After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add $\infty$ at end of array (in position $n + 1$)

**LIS**$(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

**Base case:** $\text{LIS}(0, j) = 0$ for $1 \leq j \leq n + 1$

**Recursive relation:**
- $\text{LIS}(i, j) = \text{LIS}(i - 1, j)$ if $A[i] > A[j]$
- $\text{LIS}(i, j) = \max\{\text{LIS}(i - 1, j), 1 + \text{LIS}(i - 1, i)\}$ if $A[i] \leq A[j]$

**Output:** $\text{LIS}(n, n + 1)$
Iterative algorithm

\[
\text{LIS-Iterative}(A[1..n]):
\]

\[
A[n + 1] = \infty
\]

\[
\text{int } \text{LIS}[0..n, 1..n + 1]
\]

\[
\text{for (} j = 1 \text{ to } n + 1 \text{) do}
\]

\[
\text{LIS}[0, j] = 0
\]

\[
\text{for (} i = 1 \text{ to } n \text{) do}
\]

\[
\text{for (} j = i + 1 \text{ to } n \text{)}
\]

\[
\text{If (} A[i] > A[j] \text{) } \text{LIS}[i, j] = \text{LIS}[i - 1, j]
\]

\[
\text{Else } \text{LIS}[i, j] = \max\{\text{LIS}[i - 1, j], 1 + \text{LIS}[i - 1, i]\}
\]

\[
\text{Return } \text{LIS}[n, n + 1]
\]

Running time: \(O(n^2)\)

Space: \(O(n^2)\)
**How to order bottom up computation?**

**Base case:** \( \text{LIS}(0, j) = 0 \) for \( 1 \leq j \leq n + 1 \)

**Recursive relation:**

1. \( \text{LIS}(i, j) = \text{LIS}(i - 1, j) \) if \( A[i] > A[j] \)
2. \( \text{LIS}(i, j) = \max\{\text{LIS}(i - 1, j), 1 + \text{LIS}(i - 1, i)\} \) if \( A[i] \leq A[j] \)
How to order bottom up computation?

Sequence: \( A[1..7] = 6, 3, 5, 2, 7, 8, 1 \)
Two comments

**Question:** Can we compute an optimum solution and not just its value?
Two comments

**Question:** Can we compute an optimum solution and not just its value?
Yes! See notes.

**Question:** Is there a faster algorithm for LIS?

Yes! Using a different recursion and optimizing one can obtain an \( O(n \log n) \) time and \( O(n) \) space algorithm. \( O(n \log n) \) time is not obvious. Depends on improving time by using data structures on top of dynamic programming.
Two comments

**Question:** Can we compute an optimum solution and not just its value?
Yes! See notes.

**Question:** Is there a faster algorithm for LIS? Yes! Using a different recursion and optimizing one can obtain an $O(n \log n)$ time and $O(n)$ space algorithm. $O(n \log n)$ time is not obvious. Depends on improving time by using data structures on top of dynamic programming.
Dynamic Programming

1. Find a “smart” recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.

2. Estimate the number of subproblems, the time to evaluate each subproblem and the space needed to store the value. This gives an upper bound on the total running time if we use automatic memoization.

3. Eliminate recursion and find an iterative algorithm to compute the problems bottom up by storing the intermediate values in an appropriate data structure; need to find the right way or order the subproblem evaluation. This leads to an explicit algorithm.

4. Optimize the resulting algorithm further