Dynamic Programming

Lecture 13
Friday, March 6, 2020
Part I

Dynamic programming
Dynamic Programming

Dynamic Programming is **smart recursion plus memoization**
Dynamic Programming

Dynamic Programming is **smart recursion plus memoization**

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

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

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

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

---

Miller, Hassanieh (UIUC)  
CS374  
Spring 2020
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.

**Q:** 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.

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

Miller, Hassanieh (UIUC)  CS374 3 / 32
Part II

Checking if a 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
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.
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
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.
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 $L$
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
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 $\text{English}$ and we have a procedure to check whether a string/word is in the $\text{English}$ dictionary.

- Is the string “isthisanenglishsentence” in $\text{English}^*$?
- Is “stampstamp” in $\text{English}^*$?
- Is “zibzzzad” in $\text{English}^*$?
Recursive Solution

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

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

\[ w \in L^* \text{ if } w = \epsilon \text{ or } w \in L \text{ or if } w = uv \text{ where } u \in L \text{ and } v \in L^*, |u| \geq 1 \]
Recursive Solution

When is $w \in L^*$?

$w \in L^*$ if $w = \epsilon$ or $w \in L$ 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 ($n = 0$) Output YES
    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
```

Miller, Hassanieh (UIUC)
Recursive Solution

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

\[
\text{IsStringinLstar}(A[1..n]) : \\
\quad \text{If } (n = 0) \text{ Output YES} \\
\quad \text{If } (\text{IsStrInL}(A[1..n])) \\
\quad \quad \text{Output YES} \\
\quad \text{Else} \\
\quad \quad \text{For } (i = 1 \text{ to } n - 1) \text{ do} \\
\quad \quad \quad \text{If } (\text{IsStrInL}(A[1..i]) \text{ and } \text{IsStrInLstar}(A[i + 1..n])) \\
\quad \quad \quad \quad \text{Output YES} \\
\quad \quad \text{End For} \\
\quad \text{Output NO} \\
\]

Question:

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

\( O(n) \)
Recursive Solution

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

```plaintext
IsStringinLstar($A[1..n]$):
  If ($n = 0$) Output YES
  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?
Recursive Solution

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

\[
\text{IsStringinLstar}(A[1..n]): \\
\quad \text{If } (n = 0) \text{ Output YES} \\
\quad \text{If } (\text{IsStrInL}(A[1..n])) \\
\quad \quad \text{Output YES} \\
\quad \text{Else} \\
\quad \quad \text{For } (i = 1 \text{ to } n - 1) \text{ do} \\
\quad \quad \quad \text{If } (\text{IsStrInL}(A[1..i]) \text{ and } \text{IsStrInLstar}(A[i + 1..n])) \\
\quad \quad \quad \quad \text{Output YES} \\
\quad \quad \text{Output NO}
\]

Question: How many distinct sub-problems does \( \text{IsStrInLstar}(A[1..n]) \) generate? \( O(n) \)
Consider string *samiam*
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:** $\text{ISL}(n + 1) = 1$ 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:** $\text{ISL}(n + 1) = 1$ interpreting $A[n + 1..n]$ as $\epsilon$

**Recursive relation:**

- $\text{ISL}(i) = 1$ if
  \[ \exists i < j \leq n + 1 \text{ s.t } \text{ISL}(j) \text{ and IsStrInL}(A[i..(j - 1)]) \]
- $\text{ISL}(i) = 0$ otherwise
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:** $\text{ISL}(n + 1) = 1$ interpreting $A[n + 1..n]$ as $\epsilon$

**Recursive relation:**

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

**Output:** $\text{ISL}(1)$
Removing recursion to obtain iterative algorithm

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

Why?
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.
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

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

```java
boolean ISL[1..(n + 1)]

ISL[n + 1] = TRUE

for (i = n down to 1)
    ISL[i] = FALSE

for (j = i + 1 to n + 1)
    If (ISL[j] and IsStrInL(A[i..j - 1]))
        ISL[i] = TRUE
        Break

If (ISL[1] = 1) Output YES
Else Output NO
```

Running time: \(O(n^2)\) (assuming call to \(\text{IsStrInL}\) is \(O(1)\) time)

Space: \(O(n)\)
Iterative Algorithm

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

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

- **Running time:**

\[O(n^2)\text{ (assuming call to IsStrInL is } O(1) \text{ time)}\]

\[O(n)\text{ space)\]
Iterative Algorithm

Iterative Algorithm:

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

\[
\text{boolean ISL}[1..(n + 1)] \\
\text{ISL}[n + 1] = \text{TRUE} \\
\text{for (} i = n \text{ down to } 1 \) \\
\hspace{1cm} \text{ISL}[i] = \text{FALSE} \\
\text{for (} j = i + 1 \text{ to } n + 1 \) \\
\hspace{1cm} \text{If (ISL}[j] \text{ and \ IsStrInL}(A[i..j - 1])) \\
\hspace{1.5cm} \text{ISL}[i] = \text{TRUE} \\
\hspace{1.5cm} \text{Break} \\
\]

If (ISL[1] = 1) Output YES
Else Output NO

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

\textbf{IsStringinLstar-Iterative}(A[1..n]):
\begin{itemize}
    \item boolean \textbf{ISL}[1..(n + 1)]
    \item \textbf{ISL}[n + 1] = \textbf{TRUE}
    \item for (\textbf{i} = n down to 1)
        \begin{itemize}
            \item \textbf{ISL}[i] = FALSE
            \item for (\textbf{j} = i + 1 to n + 1)
                \begin{itemize}
                    \item If (\textbf{ISL}[j] and \textbf{IsStrInL}(A[i..j - 1]))
                        \begin{itemize}
                            \item \textbf{ISL}[i] = \textbf{TRUE}
                            \item Break
                        \end{itemize}
                \end{itemize}
        \end{itemize}
\end{itemize}
\begin{itemize}
    \item If (\textbf{ISL}[1] = 1) Output YES
    \item Else Output NO
\end{itemize}

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

Iterative Algorithm

**Iterative Algorithm**

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

- boolean ISL[1..(n + 1)]
- ISL[n + 1] = TRUE

for \( i = n \) down to 1

- ISL[i] = FALSE

for \( j = i + 1 \) to \( n + 1 \)

- If (ISL[j] and IsStrInL(A[i..j − 1]))
  - ISL[i] = TRUE
  - Break

If (ISL[1] = 1) Output YES
Else Output NO

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

Consider string \textit{samiam}
Part III

Longest Increasing Subsequence
Sequences

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...

Example

Sequence: 6, 3, 5, 2, 7, 8, 1, 9
Subsequence of above sequence: 5, 2, 1
Increasing sequence: 3, 5, 9, 17, 54
Decreasing sequence: 34, 21, 7, 5, 1
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**\((A[1..n])\):
Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

\[ \text{LIS}(A[1..n]): \]
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 \( \text{LIS\_smaller}(A[1..n], x) \) 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 $A$

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

\begin{verbatim}
LIS_{\text{smaller}}(A[1..n], x):
    if (n = 0) then return 0
    m = LIS_{\text{smaller}}(A[1..(n − 1)], x)
    if (A[n] < x) then
        m = max(m, 1 + LIS_{\text{smaller}}(A[1..(n − 1)], A[n]))
    Output m
\end{verbatim}

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

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

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

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

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

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

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

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

```python
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
```

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

- How many distinct sub-problems will \( \text{LIS}\_\text{smaller}(A[1..n], ∞) \) 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.
Recursive Approach

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

\[
\text{LIS}(A[1..n]) :
\begin{align*}
&\text{return } \text{LIS\_smaller}(A[1..n], \infty)
\end{align*}
\]

- How many distinct sub-problems will \text{LIS\_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.
- How much space for memoization?
Recursive Approach

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

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

- How many distinct sub-problems will \( \text{LIS\_smaller}(A[1..n], \infty) \) 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.
- 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 $i < j$)
Naming subproblems and recursive equation

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: $LIS(0, j) = 0$ for $1 \leq j \leq n + 1$
Naming subproblems and recursive equation

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: $LIS(0, j) = 0$ for $1 \leq j \leq n + 1$

Recursive relation:

- $LIS(i, j) = LIS(i - 1, j)$ if $A[i] > A[j]$
- $LIS(i, j) = \max\{LIS(i - 1, j), 1 + LIS(i - 1, i)\}$ if $A[i] \leq A[j]$
Naming subproblems and recursive equation

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: $LIS(0, j) = 0$ for $1 \leq j \leq n + 1$

Recursive relation:

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

Output: $LIS(n, n + 1)$
Iterative algorithm

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

- $A[n + 1] = \infty$
- int $LIS[0..n, 1..n + 1]$
- for ($j = 1$ to $n + 1$) do
  - $LIS[0,j] = 0$

- for ($i = 1$ to $n$) do
  - for ($j = i + 1$ to $n$)
    - Else $LIS[i,j] = \max\{LIS[i − 1,j], 1 + LIS[i − 1,i]\}$

Return $LIS[n, n + 1]$

**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:

- \( \text{LIS}(i, j) = \text{LIS}(i - 1, j) \) if \( A[i] > A[j] \)
- \( \text{LIS}(i, j) = \max\{\text{LIS}(i - 1, j), 1 + \text{LIS}(i - 1, i)\} \) if \( A[i] \leq A[j] \)
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?

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.
Definition

\[ \text{LISEnding}(A[1..n]): \text{ length of longest increasing sub-sequence that ends in } A[n]. \]

Question: can we obtain a recursive expression?
Recursive Algorithm: Take 2

**Definition**

\( \text{LISEnding}(A[1..n]) \): length of longest increasing sub-sequence that ends in \( A[n] \).

**Question:** can we obtain a recursive expression?

\[
\text{LISEnding}(A[1..n]) = \max_{i : A[i] < A[n]} \left( 1 + \text{LISEnding}(A[1..i]) \right)
\]
Example

Sequence:  $A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9$
Recursive Algorithm: Take 2

\[
\text{LIS\_ending\_alg}(A[1..n]):
\]
\[
\begin{align*}
&\text{if } (n = 0) \text{ return } 0 \\
&m = 1 \\
&\text{for } i = 1 \text{ to } n - 1 \text{ do} \\
&\quad \text{if } (A[i] < A[n]) \text{ then} \\
&\quad \quad m = \max\left( m, 1 + \text{LIS\_ending\_alg}(A[1..i]) \right) \\
&\text{return } m
\end{align*}
\]

\[
\text{LIS}(A[1..n]):
\]
\[
\begin{align*}
&\text{return } \max_{i=1}^{n} \text{LIS\_ending\_alg}(A[1..i])
\end{align*}
\]

How many distinct sub-problems will \(\text{LIS\_ending\_alg}(A[1..n])\) generate?

\(O(n)\)

What is the running time if we memoize recursion?

\(O(n^2)\) since each call takes \(O(n)\) time

How much space for memoization?

\(O(n)\)

Miller, Hassanieh (UIUC)  
CS374  28  
Spring 2020  28 / 32
Recursive Algorithm: Take 2

\[
\text{LIS}_\text{ending}_\text{alg}(A[1..n]) : \\
\quad \text{if } (n = 0) \text{ return } 0 \\
\quad m = 1 \\
\quad \text{for } i = 1 \text{ to } n - 1 \text{ do} \\
\quad \quad \text{if } (A[i] < A[n]) \text{ then} \\
\quad \quad \quad m = \max(m, 1 + \text{LIS}_\text{ending}_\text{alg}(A[1..i])) \\
\quad \text{return } m
\]

\[
\text{LIS}(A[1..n]) : \\
\quad \text{return } \max_{i=1}^{n} \text{LIS}_\text{ending}_\text{alg}(A[1 \ldots i])
\]

- How many distinct sub-problems will \text{LIS}_\text{ending}_\text{alg}(A[1..n]) generate?
Recursive Algorithm: Take 2

\[
\text{LIS\_ending\_alg}(A[1..n]):
\]
\[
\begin{align*}
& \text{if } (n = 0) \text{ return } 0 \\
& m = 1 \\
& \text{for } i = 1 \text{ to } n - 1 \text{ do} \\
& \quad \text{if } (A[i] < A[n]) \text{ then} \\
& \quad \quad m = \max \left( m, 1 + \text{LIS\_ending\_alg}(A[1..i]) \right) \\
& \text{return } m
\end{align*}
\]

\[
\text{LIS}(A[1..n]):
\]
\[
\begin{align*}
& \text{return } \max_{i=1}^{n} \text{LIS\_ending\_alg}(A[1 \ldots i])
\end{align*}
\]

- How many distinct sub-problems will \text{LIS\_ending\_alg}(A[1..n]) generate? \(O(n)\)
Recursive Algorithm: Take 2

\[
\text{LIS\_ending\_alg}(A[1..n]) : \\
\quad \text{if } (n = 0) \text{ return } 0 \\
\quad m = 1 \\
\quad \text{for } i = 1 \text{ to } n - 1 \text{ do} \\
\quad \quad \text{if } (A[i] < A[n]) \text{ then} \\
\quad \quad \quad m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i])) \\
\quad \text{return } m \\
\]

\[
\text{LIS}(A[1..n]) : \\
\quad \text{return } \max_{i=1}^{n} \text{LIS\_ending\_alg}(A[1 \ldots i])
\]

- How many distinct sub-problems will \text{LIS\_ending\_alg}(A[1..n]) generate? \(O(n)\)
- What is the running time if we memoize recursion?
Recursive Algorithm: Take 2

```python
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n - 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```python
LIS(A[1..n]):
    return max^n_{i=1} LIS_ending_alg(A[1..i])
```

- How many distinct sub-problems will `LIS_ending_alg(A[1..n])` generate? $O(n)$
- What is the running time if we memoize recursion? $O(n^2)$ since each call takes $O(n)$ time
Recursive Algorithm: Take 2

```
LIS_ending_alg(A[1..n]):
    if (n = 0) return 0
    m = 1
    for i = 1 to n − 1 do
        if (A[i] < A[n]) then
            m = max(m, 1 + LIS_ending_alg(A[1..i]))
    return m
```

```
LIS(A[1..n]):
    return max_{i=1}^{n} LIS_ending_alg(A[1..i])
```

- How many distinct sub-problems will \( \text{LIS}_\text{ending}_\text{alg}(A[1..n]) \) generate? \( O(n) \)
- What is the running time if we memoize recursion? \( O(n^2) \) since each call takes \( O(n) \) time
- How much space for memoization?
Recursive Algorithm: Take 2

\[
LIS_{\text{ending alg}}(A[1..n]):
\]
\[
\begin{align*}
& \text{if } (n = 0) \text{ return } 0 \\
& m = 1 \\
& \text{for } i = 1 \text{ to } n - 1 \text{ do} \\
& \quad \text{if } (A[i] < A[n]) \text{ then} \\
& \quad \quad m = \max (m, 1 + LIS_{\text{ending alg}}(A[1..i])) \\
& \text{return } m
\end{align*}
\]

\[
LIS(A[1..n]):
\]
\[
\text{return } \max_{i=1}^{n} LIS_{\text{ending alg}}(A[1 \ldots i])
\]

- How many distinct sub-problems will \( LIS_{\text{ending alg}}(A[1..n]) \) generate? \( O(n) \)
- What is the running time if we memoize recursion? \( O(n^2) \) since each call takes \( O(n) \) time
- How much space for memoization? \( O(n) \)
Iterative Algorithm via Memoization

Compute the values \( \text{LIS\textunderscore ending\textunderscore alg}(A[1..i]) \) iteratively in a bottom up fashion.

\[
\text{LIS\textunderscore ending\textunderscore alg}(A[1..n]):
\]

1. **Array** \( L[1..n] \) (* \( L[i] = \text{value of } \text{LIS\textunderscore ending\textunderscore alg}(A[1..i]) \) *)
2. **for** \( i = 1 \) to \( n \) **do**
   - \( L[i] = 1 \)
   - **for** \( j = 1 \) to \( i - 1 \) **do**
       - \( L[i] = \max(L[i], 1 + L[j]) \)
3. **return** \( L \)

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

\[
L = \text{LIS\textunderscore ending\textunderscore alg}(A[1..n])
\]

**return** the maximum value in \( L \)
Iterative Algorithm via Memoization

Simplifying:

\[
\text{LIS}(A[1..n]): \quad \text{Array } L[1..n] \quad (* \text{ } L[i] \text{ } \text{stores the value LisEnding}(A[1..i]) \text{ } *) \\
\text{m} = 0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad L[i] = 1 \\
\text{for } j = 1 \text{ to } i - 1 \text{ do} \\
\quad \text{if } (A[j] < A[i]) \text{ do} \\
\quad \quad L[i] = \max(L[i], 1 + L[j]) \\
\quad m = \max(m, L[i]) \\
\text{return } m
\]
Iterative Algorithm via Memoization

Simplifying:

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

\[
\text{Array } L[1..n] \quad (* L[i] \text{ stores the value } \text{LISEnding}(A[1..i]) \quad *)
\]

\[
m = 0
\]

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

\[
L[i] = 1
\]

\[
\text{for } \ j = 1 \ \text{to } \ i - 1 \ \text{do}
\]

\[
\text{if } (A[j] < A[i]) \ \text{do}
\]

\[
L[i] = \max(L[i], 1 + L[j])
\]

\[
m = \max(m, L[i])
\]

\[
\text{return } m
\]

Correctness: Via induction following the recursion

Running time:
Iterative Algorithm via Memoization

Simplifying:

\[\text{LIS}(A[1..n]) : \]
Array \( L[1..n] \) (* \( L[i] \) stores the value \( \text{LISEnding}(A[1..i]) \) *)

\( m = 0 \)

for \( i = 1 \) to \( n \) do

\( L[i] = 1 \)

for \( j = 1 \) to \( i - 1 \) do

if \((A[j] < A[i])\) do

\( L[i] = \max(L[i], 1 + L[j]) \)

\( m = \max(m, L[i]) \)

return \( m \)

Correctness: Via induction following the recursion

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

Space:
Iterative Algorithm via Memoization

Simplifying:

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

Array \( L[1..n] \) (\(* L[i] \) stores the value LISEnding(A[1..i]) \*)

\( m = 0 \)

for \( i = 1 \) to \( n \) do

\( L[i] = 1 \)

for \( j = 1 \) to \( i - 1 \) do


\( L[i] = \max(L[i], 1 + L[j]) \)

\( m = \max(m, L[i]) \)

return \( m \)

Correctness: Via induction following the recursion

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

Space: \( \Theta(n) \)
Iterative Algorithm via Memoization

Simplifying:

\[
\text{LIS}(A[1..n]):
\]
\[
\text{Array } L[1..n] \text{ (* } L[i] \text{ stores the value } \text{LISEnding}(A[1..i]) \text{ *)}
\]
\[
m = 0
\]
\[
\text{for } i = 1 \text{ to } n \text{ do}
\]
\[
L[i] = 1
\]
\[
\text{for } j = 1 \text{ to } i - 1 \text{ do}
\]
\[
\text{if } A[j] < A[i] \text{ do}
\]
\[
L[i] = \max(L[i], 1 + L[j])
\]
\[
m = \max(m, L[i])
\]
\[
\text{return } m
\]

Correctness: Via induction following the recursion
Running time: \(O(n^2)\)
Space: \(\Theta(n)\)

\(O(n \log n)\) run-time achievable via better data structures.
Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Longest increasing subsequence: 3, 5, 7, 8
Example

Sequence: 6, 3, 5, 2, 7, 8, 1

Longest increasing subsequence: 3, 5, 7, 8

1. \( L[i] \) is value of longest increasing subsequence ending in \( A[i] \)
2. Recursive algorithm computes \( L[i] \) from \( L[1] \) to \( L[i − 1] \)
3. Iterative algorithm builds up the values from \( L[1] \) to \( L[n] \)
Find a “smart” recursion for the problem in which the number of distinct subproblems is small; polynomial in the original problem size.

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.

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.

Optimize the resulting algorithm further.