Dynamic Programming

Lecture 13
Recursion types

1. **Divide and Conquer**: Problem reduced to multiple independent sub-problems.
   
   Examples: Merge sort, quick sort, multiplication, median selection.
   
   Each sub-problem is a fraction smaller.

2. **Backtracking**: A sequence of decision problems. Recursion tries all possibilities at each step.
   
   Each subproblem is only a constant smaller, e.g. from $n$ to $n - 1$. 
Recursion types

1. **Divide and Conquer**: Problem reduced to multiple independent sub-problems.
   Examples: Merge sort, quick sort, multiplication, median selection.
   Each sub-problem is a fraction smaller.

2. **Backtracking**: A sequence of decision problems. Recursion tries all possibilities at each step.
   Each subproblem is only a constant smaller, e.g. from $n$ to $n - 1$.

3. **Dynamic Programming**: Smart recursion with memoization
Part I

Fibonacci Numbers
Fibonacci Numbers

Fibonacci numbers defined by recurrence:

\[ F(n) = F(n - 1) + F(n - 2) \]  and  \[ F(0) = 0, F(1) = 1. \]

These numbers have many interesting and amazing properties. A journal *The Fibonacci Quarterly*!

1. \[ F(n) = (\phi^n - (1 - \phi)^n) / \sqrt{5} \] where \( \phi \) is the golden ratio \( (1 + \sqrt{5}) / 2 \approx 1.618. \)

2. \[ \lim_{n \to \infty} F(n + 1) / F(n) = \phi \]
Recursive Algorithm for Fibonacci Numbers

Question: Given \( n \), compute \( F(n) \).

\[
\text{Fib}(n):
\begin{align*}
\text{if } (n = 0) & \quad \text{return } 0 \\
\text{else if } (n = 1) & \quad \text{return } 1 \\
\text{else} & \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]
Question: Given $n$, compute $F(n)$.

$$\text{Fib}(n):$$
- if ($n = 0$) return 0
- else if ($n = 1$) return 1
- else return $\text{Fib}(n - 1) + \text{Fib}(n - 2)$

Running time? Let $T(n)$ be the number of additions in Fib(n).
Question: Given $n$, compute $F(n)$.

$$
\text{Fib}(n):
\begin{align*}
\text{if } (n = 0) & \quad \text{return 0} \\
\text{else if } (n = 1) & \quad \text{return 1} \\
\text{else} & \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
$$

Running time? Let $T(n)$ be the number of additions in Fib(n).

$$
T(n) = T(n - 1) + T(n - 2) + 1 \text{ and } T(0) = T(1) = 0
$$
Recursive Algorithm for Fibonacci Numbers

Question: Given \( n \), compute \( F(n) \).

\[
\text{Fib}(n) :
\begin{align*}
\text{if } (n = 0) & \quad \text{return } 0 \\
\text{else if } (n = 1) & \quad \text{return } 1 \\
\text{else} & \quad \text{return } \text{Fib}(n - 1) + \text{Fib}(n - 2)
\end{align*}
\]

Running time? Let \( T(n) \) be the number of additions in \( \text{Fib}(n) \).

\[
T(n) = T(n - 1) + T(n - 2) + 1 \quad \text{and} \quad T(0) = T(1) = 0
\]

Roughly same as \( F(n) \)

\[
T(n) = \Theta(\phi^n)
\]

The number of additions is exponential in \( n \). Can we do better?
Recursion Tree
Memoization

- The recursive algorithm is slow because it computes the same Fibonacci numbers over and over.
Memoization

- The recursive algorithm is slow because it computes the same Fibonacci numbers over and over.

Memoization

1. Write down the results of recursive calls and look them up later
Memoization

- The recursive algorithm is slow because it computes the same Fibonacci numbers over and over.

1. Write down the results of recursive calls and look them up later
2. An array $F(n)$, where $F(i)$ stores the result of $\text{Fib}(i)$
The recursive algorithm is slow because it computes the same Fibonacci numbers over and over.

Memoization

1. Write down the results of recursive calls and look them up later
2. An array $F(n)$, where $F(i)$ stores the result of $Fib(i)$
3. Evaluation order: From bottom up, $i = 2$ then $i = 3$ and so on
An iterative algorithm for Fibonacci numbers

```plaintext
FibIter(n):
  if (n = 0) then
    return 0
  if (n = 1) then
    return 1
  F[0] = 0
  F[1] = 1
  for i = 2 to n do
    F[i] = F[i - 1] + F[i - 2]
  return F[n]
```

What is the running time of the algorithm?

$O(n)$ additions.
An iterative algorithm for Fibonacci numbers

\[
\text{FibIter}(n): \\
\begin{align*}
\text{if } (n = 0) \text{ then} & \quad \text{return } 0 \\
\text{if } (n = 1) \text{ then} & \quad \text{return } 1 \\
F[0] & = 0 \\
F[1] & = 1 \\
\text{for } i = 2 \text{ to } n \text{ do} & \quad F[i] = F[i - 1] + F[i - 2] \\
\text{return } F[n]
\end{align*}
\]

What is the running time of the algorithm?
An iterative algorithm for Fibonacci numbers

\texttt{FibIter}(n):
\begin{algorithmic}
\Function{FibIter}{n}
\If{\(n = 0\)}
    \State \Return 0
\EndIf
\If{\(n = 1\)}
    \State \Return 1
\EndIf
\State \(F[0] = 0\)
\State \(F[1] = 1\)
\For{\(i = 2\) to \(n\)}
    \State \(F[i] = F[i - 1] + F[i - 2]\)
\EndFor
\State \Return \(F[n]\)
\EndFunction
\end{algorithmic}

What is the running time of the algorithm? \(O(n)\) additions.
DP prunes recursion tree
What is the difference?

**Dynamic Programming:**
Finding a recursion that can be *effectively/efficiently* memoized.

Leads to polynomial time algorithm if number of distinct sub-problems is polynomial in input size.
Do we need an array of $n$ numbers? Not really.

```python
FibIter(n):
    if (n == 0) then
        return 0
    if (n == 1) then
        return 1
    prev2 = 0
    prev1 = 1
    for i = 2 to n do
        temp = prev1 + prev2
        prev2 = prev1
        prev1 = temp
    return prev1
```
Dynamic Programming

**Dynamic Programming**: Smart recursion with memoization
**Dynamic Programming**: Smart recursion with memoization

- Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
Dynamic Programming: Smart recursion with memoization

- Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
- Use **memoization** to avoid recomputation of common solutions, hence optimizing running time and space.
Dynamic Programming

**Dynamic Programming**: Smart recursion with memoization

- Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct recursion.
- Use **memoization** to avoid recomputation of common solutions, hence optimizing running time and space.
  - 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.
- Often an *iterative* algorithm with *bottom up* computation.
Part II

Text Segmentation
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}*? 
- Is “stampstamp” in \textit{English}*? 
- Is “zibzzzad” in \textit{English}*?
Text Segmentation

**Backtracking**

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Let the recursion fairy make all remaining decisions
Text Segmentation

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Let the recursion fairy make all remaining decisions

Only the suffix matters.

HEARTHANDSATURNSPIN
Text Segmentation

Backtracking

- Changes the problem into a sequence of decision problems
- Each tries all possibilities for the current decision
- Let the recursion fairy make all remaining decisions

Only the suffix matters.

Base case

- zero-length string
Recursive Solution

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

```python
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 `IsStringInLstar(A[1..n])` generate?

$O(n)$
Recursive Solution

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

\[
\text{IsStringInLstar}(A[1..n]):
\]
\[
\begin{align*}
\text{If } (n = 0) & \text{ Output YES} \\
\text{If } (\text{IsStrInL}(A[1..n])) & \text{ Output YES} \\
\text{Else} & \\
\text{For } (i = 1 \text{ to } n - 1) & \text{ do} \\
\text{If } (\text{IsStrInL}(A[1..i]) \text{ and IsStrInLstar}(A[i + 1..n])) & \text{ Output YES} \\
\text{Output NO}
\end{align*}
\]

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

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

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

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

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 $\epsilon$
Evaluate subproblems

Recursive relation:

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

Alternatively: \( ISL(i) = \max_{i < j \leq n+1} ISL(j)IsStrInL(A[i..(j - 1)]) \)
Evaluate subproblems

Recursive relation:

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

Alternatively: $\text{ISL}(i) = \max_{i < j \leq n+1} \text{ISL}(j)\text{IsStrInL}(A[i..(j - 1)])$

Output: $\text{ISL}(1)$
Iterative Algorithm

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

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

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

1. boolean ISL[1..(n + 1)]
2. \textbf{ISL}[n + 1] = TRUE
3. for (i = n down to 1)
   1. \textbf{ISL}[i] = FALSE
   2. for (j = i + 1 to n + 1)
      1. If (ISL[j] and IsStrInL(A[i..j − 1]))
      2. ISL[i] = TRUE
      3. Break

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

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

**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:**
Iterative Algorithm

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

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

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

How to design DP algorithms

1. Find a “smart” recursion (The hard part)
   1. Formulate the sub-problem
   2. so that the number of distinct subproblems is small; polynomial in the original problem size.

Memoization

1. Identify distinct subproblems
2. Choose a memoization data structure
3. Identify dependencies and find a good evaluation order
4. An iterative algorithm replacing recursive calls with array lookups
How to design DP algorithms

1. Find a “smart” recursion (The hard part)
   1. Formulate the sub-problem
   2. so that the number of distinct subproblems is small; polynomial in the original problem size.

2. Memoization
   1. Identify distinct subproblems
   2. Choose a memoization data structure
   3. Identify dependencies and find a good evaluation order
   4. An iterative algorithm replacing recursive calls with array lookups
Part III

Longest Increasing Subsequence
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
Can we find a recursive algorithm for \textbf{LIS}?

\textbf{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.
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 \( \text{LIS}_{\text{smaller}}(A[1..n], x) \) which gives the longest increasing subsequence in \( A \) where each number in the sequence is less than \( x \).
**Recursive Approach**

$LIS(A[1..n])$: 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], ∞)
```
Recursive Approach

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

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

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

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

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

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

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

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

- 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? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from two recursive calls and no other computation.
```plaintext
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 two recursive calls and no other computation.
- How much space for memoization?
Recursive Approach

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

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

- How many distinct sub-problems will \( \text{LIS\_smaller}(A[1..n], \infty) \) generate? \( O(n^2) \)
- What is the running time if we memoize recursion? \( O(n^2) \) since each call takes \( O(1) \) time to assemble the answers from two 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 $i < j$)
How to order bottom up computation?

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

- **Recursive relation:**
  \[
  \text{LIS}(i, j) = \begin{cases} \\
  \text{LIS}(i-1, j) & \text{if } A[i] > A[j] \\
  \max\{ \text{LIS}(i-1, j), 1 + \text{LIS}(i-1, i) \} & \text{if } A[i] \leq A[j]
  \end{cases}
  \]
How to order bottom up computation?

Base case: \( \text{LIS}(0, j) = 0 \) for \( 1 \leq j \leq n + 1 \)
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: $A[1..7] = 6, 3, 5, 2, 7, 8, 1$
Iterative algorithm

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

1. \( A[n + 1] = \infty \)
2. int \( LIS[0..n, 1..n + 1] \)
3. for (\( j = 1 \) to \( n + 1 \)) do
   \( LIS[0,j] = 0 \)
4. for (\( i = 1 \) to \( n \)) do
   for (\( j = i + 1 \) to \( n \))
     Else \( LIS[i,j] = \max\{LIS[i - 1,j], 1 + LIS[i - 1,i]\} \)
5. Return \( LIS[n,n+1] \)

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

**Space:** \( O(n^2) \)