Dynamic Programming

Lecture 09
February 17, 2015
Part I

Longest Increasing Subsequence
9.1: 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

**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

**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.
Sequences

Example...

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: $2, 7, 9$. 
Sequences

Example...

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.
Sequences

Example...

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.
Sequences

Example...

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**.

Sariel (UIUC)  OLD CS473  5  Spring 2015  5 / 40
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
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
Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array $A$

```
algLISNaive(A[1..n]):
    max = 0
    for each subsequence $B$ of $A$ do
        if $B$ is increasing and $|B| > max$ then
            max = $|B|$
    Output max
```

Running time: $O(n2^n)$.

$2^n$ subsequences of a sequence of length $n$ and $O(n)$ time to check if a given sequence is increasing.
Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array $A$

```python
algLISNaive(A[1..n]):
    \text{max} = 0
    \text{for each subsequence } B \text{ of } A \text{ do}
    \quad \text{if } B \text{ is increasing and } |B| > \text{max} \text{ then}
    \quad \quad \text{max} = |B|

Output \text{max}
```

Running time: $O(n2^n)$.

$2^n$ subsequences of a sequence of length $n$ and $O(n)$ time to check if a given sequence is increasing.
Naïve Enumeration

Assume $a_1, a_2, \ldots, a_n$ is contained in an array $A$

\begin{verbatim}
algLISNaive(A[1..n]):
    max = 0
    for each subsequence $B$ of $A$ do
        if $B$ is increasing and $|B| > max$ then
            max = $|B|$
    Output max
\end{verbatim}

Running time: $O(n2^n)$.

$2^n$ subsequences of a sequence of length $n$ and $O(n)$ time to check if a given sequence is increasing.
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**

*If* $A[n]$ *is in the longest increasing subsequence then all the elements before it must be smaller.*
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

if A[n] is in the longest increasing subsequence then all the elements before it must be smaller.
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**

if \(A[n]\) is in the longest increasing subsequence then all the elements before it must be smaller.
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**

_if $A[n]$ is in the longest increasing subsequence then all the elements before it must be smaller._
Recursive Approach: Take 1

**algLIS**(*A*[1..*n*]):

1. if (*n* = 0) then return 0
2. *m* = **algLIS**(*A*[1..(*n* − 1)])
3. *B* is subsequence of *A*[1..(*n* − 1)] with only elements less than *A*[*n*]
   (*let *h* be size of *B*, *h* ≤ *n* − 1 *)
4. *m* = max(*m*, 1 + **algLIS**(B[1..*h*]))

Output *m*

Recursion for running time: \( T(n) \leq 2T(n − 1) + O(n) \).

Easy to see that \( T(n) \) is \( O(n2^n) \).
Recursive Approach: Take 1

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

\[
\text{if } (n = 0) \text{ then return 0}
\]

\[
m = \text{algLIS}(A[1..(n - 1)])
\]

\[B \text{ is subsequence of } A[1..(n - 1)] \text{ with only elements less than } A[n]\]

\[(* \text{ let } h \text{ be size of } B, \ h \leq n - 1 \ *)\]

\[
m = \max(m, 1 + \text{algLIS}(B[1..h]))
\]

Output \( m \)

Recursion for running time: \( T(n) \leq 2T(n - 1) + O(n) \).

Easy to see that \( T(n) \) is \( O(n2^n) \).
Recursive Approach: Take 1

algLIS(A[1..n]):
    if (n = 0) then return 0
    m = algLIS(A[1..(n – 1)])
    B is subsequence of A[1..(n – 1)] with
    only elements less than A[n]
    (* let h be size of B, h ≤ n − 1 *)
    m = max(m, 1 + algLIS(B[1..h]))
    Output m

Recursion for running time: \( T(n) \leq 2T(n − 1) + O(n) \).
Easy to see that \( T(n) \) is \( O(n2^n) \).
Recursive Approach: Take 2

\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.

**Observation**

1. **Case 2:** find a subsequence in \( A[1..(n-1)] \) that is restricted to numbers less than \( A[n] \).

2. **Generalization** \( \text{LIS\_smaller}(A[1..n], x) \): longest increasing subsequence in \( A \), all numbers in sequence is \( \leq x \).
Recursive Approach: Take 2

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

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**

1. **Case 2:** find a subsequence in \( A[1..(n-1)] \) that is restricted to numbers less than \( A[n] \).

2. **Generalization** \( \text{LIS\_smaller}(A[1..n], x) \): longest increasing subsequence in \( A \), all numbers in sequence is \( \leq x \).
Recursive Approach: Take 2

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**

1. **Case 2**: find a subsequence in $A[1..(n-1)]$ that is restricted to numbers less than $A[n]$.

2. **Generalization** \[ \text{LIS}\_\text{smaller}(A[1..n], x) \]: longest increasing subsequence in $A$, all numbers in sequence is \( \leq x \).
Recursive Approach: Take 2

\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.

\textbf{Observation}

1. Case 2: find a subsequence in \(A[1..(n - 1)]\) that is restricted to numbers less than \(A[n]\).

2. Generalization \(\text{LIS\_smaller}(A[1..n], x)\): longest increasing subsequence in \(A\), all numbers in sequence is \(\leq x\).
Recursive Approach: Take 2

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

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**

1. **Case 2:** find a subsequence in \( A[1..(n-1)] \) that is restricted to numbers less than \( A[n] \).

2. **Generalization** \( \text{LIS}_{\text{smaller}}(A[1..n], x) \): longest increasing subsequence in \( A \), all numbers in sequence is \( \leq x \).
Recursive Approach: Take 2

\textbf{LIS\_smaller}(A[1..n], x): length of longest increasing subsequence in \(A[1..n]\) with all numbers in subsequence less than \(x\)

<table>
<thead>
<tr>
<th>LIS_smaller(A[1..n], x):</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{if} ((n = 0)) \textbf{then} return 0</td>
</tr>
<tr>
<td>(m = \text{LIS_smaller}(A[1..(n - 1)], x))</td>
</tr>
<tr>
<td>\textbf{if} ((A[n] &lt; x)) \textbf{then}</td>
</tr>
<tr>
<td>(m = \text{max}(m, 1 + \text{LIS_smaller}(A[1..(n - 1)], A[n])))</td>
</tr>
<tr>
<td>Output (m)</td>
</tr>
</tbody>
</table>

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

Recursion for running time: \(T(n) \leq 2T(n - 1) + O(1)\).

\textbf{Question:} Is there any advantage?
Recursive Approach: Take 2

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

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

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

Recursion for running time: \[ T(n) \leq 2T(n - 1) + O(1). \]

Question: Is there any advantage?
Recursive Approach: Take 2

\textbf{LIS\_smaller}(A[1..n], x) : length of longest increasing subsequence in A[1..n] with all numbers in subsequence less than x

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

Output \(m\)

\textbf{LIS}(A[1..n]):
\begin{itemize}
  \item return \(\text{LIS\_smaller}(A[1..n], \infty)\)
\end{itemize}

Recursion for running time: \(T(n) \leq 2T(n-1) + O(1)\).

Question: Is there any advantage?
Observation

The number of different subproblems generated by \( \text{LIS\_smaller}(A[1..n], x) \) is \( O(n^2) \).

Memoization the recursive algorithm leads to an \( O(n^2) \) running time!

Question: What are the recursive subproblem generated by \( \text{LIS\_smaller}(A[1..n], x) \)?

1. For \( 0 \leq i < n \) \( \text{LIS\_smaller}(A[1..i], y) \) where \( y \) is either \( x \) or one of \( A[i + 1], \ldots, A[n] \).

Observation

Previous recursion also generates only \( O(n^2) \) subproblems. Slightly harder to see.
Observation

The number of different subproblems generated by \texttt{LIS\_smaller}(\texttt{A[1..n]}, \texttt{x}) is \(O(n^2)\).

Memoization the recursive algorithm leads to an \(O(n^2)\) running time!

Question: What are the recursive subproblem generated by \texttt{LIS\_smaller}(\texttt{A[1..n]}, \texttt{x})?

1. For \(0 \leq i < n\) \texttt{LIS\_smaller}(\texttt{A[1..i]}, \texttt{y}) where \texttt{y} is either \texttt{x} or one of \texttt{A[i + 1]}, \ldots, \texttt{A[n]}.

Observation

Previous recursion also generates only \(O(n^2)\) subproblems. Slightly harder to see.
Observation

The number of different subproblems generated by \texttt{LIS\_smaller}(A[1..n], x) is \(O(n^2)\).

Memoization the recursive algorithm leads to an \(O(n^2)\) running time!

Question: What are the recursive subproblem generated by \texttt{LIS\_smaller}(A[1..n], x)?

1. For \(0 \leq i < n\) \texttt{LIS\_smaller}(A[1..i], y) where \(y\) is either \(x\) or one of \(A[i+1], \ldots, A[n]\).

Observation

Previous recursion also generates only \(O(n^2)\) subproblems. Slightly harder to see.
**Recursive Algorithm: Take 2**

**Observation**

The number of different subproblems generated by \( \text{LIS\_smaller}(A[1..n], x) \) is \( O(n^2) \).

Memoization the recursive algorithm leads to an \( O(n^2) \) running time!

**Question:** What are the recursive subproblem generated by \( \text{LIS\_smaller}(A[1..n], x) \)?

1. For \( 0 \leq i < n \) \( \text{LIS\_smaller}(A[1..i], y) \) where \( y \) is either \( x \) or one of \( A[i + 1], \ldots, A[n] \).

**Observation**

Previous recursion also generates only \( O(n^2) \) subproblems. Slightly harder to see.
Observation

The number of different subproblems generated by \texttt{LIS\_smaller}(A[1..n], x) is $O(n^2)$.

Memoization the recursive algorithm leads to an $O(n^2)$ running time!

**Question:** What are the recursive subproblem generated by \texttt{LIS\_smaller}(A[1..n], x)?

1. For $0 \leq i < n$ \texttt{LIS\_smaller}(A[1..i], y) where y is either x or one of A[i + 1], \ldots, A[n].

Observation

Previous recursion also generates only $O(n^2)$ subproblems. Slightly harder to see.
Recursive Algorithm: Take 3

**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?

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

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

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

\[
\text{LISEnding}(A[1..n]) = \max_{i: A[i] < A[n]} ([])1 + \text{LISEnding}(A[1..i])
\]
Recursive Algorithm: Take 3

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

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

**Question:**
How many distinct subproblems generated by \( \text{LIS\_ending\_alg}(A[1..n]) \)? \( n \).
Recursive Algorithm: Take 3

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

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

**Question:**

How many distinct subproblems generated by \( \text{LIS\_ending\_alg}(A[1..n]) \)? \( n \).
Recursive Algorithm: Take 3

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

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

Question:

How many distinct subproblems generated by \(\text{LIS\_ending\_alg}(A[1..n])\)? \(n\).
Iterative Algorithm via Memoization

Compute the values \texttt{LIS\_ending\_alg}(A[1..i]) iteratively in a bottom up fashion.

\begin{algorithm}
\textbf{LIS\_ending\_alg}(A[1..n]):
\begin{algorithmic}
\State Array \texttt{L}[1..n] (* \texttt{L}[i] = value of \texttt{LIS\_ending\_alg}(A[1..i]) *)
\For {\texttt{i} = 1 to \texttt{n}}
\State \texttt{L}[i] = 1
\For {\texttt{j} = 1 to \texttt{i} - 1}
\If {\texttt{A}[j] < \texttt{A}[i]}
\State \texttt{L}[i] = max(\texttt{L}[i], 1 + \texttt{L}[j])
\EndIf
\EndFor
\EndFor
\Return \texttt{L}
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\textbf{LIS}(A[1..n]):
\begin{algorithmic}
\State \texttt{L} = \texttt{LIS\_ending\_alg}(A[1..n])
\Return the maximum value in \texttt{L}
\end{algorithmic}
\end{algorithm}
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:

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

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

\[
m = 0
\]

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

\[
L[i] = 1
\]

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

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

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

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

\[
\text{return } m
\]

Correctness: Via induction following the recursion

Running time: \( O(n^2) \), Space: \( \Theta(n) \)
Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion
Running time: \(O(n^2)\), Space: \(\Theta(n)\)
Iterative Algorithm via Memoization

Simplifying:

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

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

Simplifying:

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

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

return \( m \)

Correctness: Via induction following the recursion
Running time: \( O(n^2) \), Space: \( \Theta(n) \)
Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

Running time: \(O(n^2)\), Space: \(\Theta(n)\)
Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. 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 $L[n]$
Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. 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 $L[n]$
Memoizing LIS

\[ \text{LIS}(A[1..n]) : \]
\[ A[n+1] = \infty \text{ (* add a sentinel at the end *)} \]
Array \( L[(n+1),(n+1)] \) (* two-dimensional array*)
\[ (* L[i,j] \text{ for } j \geq i \text{ stores the value LIS}_\text{smaller}(A[1..i],A[j]) \) \]
\[ \text{for } j = 1 \text{ to } n+1 \text{ do} \]
\[ L[0,j] = 0 \]
\[ \text{for } i = 1 \text{ to } n+1 \text{ do} \]
\[ \text{for } j = i \text{ to } n+1 \text{ do} \]
\[ L[i,j] = L[i-1,j] \]
\[ \text{if } (A[i] < A[j]) \text{ then} \]
\[ L[i,j] = \max(L[i,j], 1 + L[i-1,i]) \]
\[ \text{return } L[n,(n+1)] \]

Correctness: Via induction following the recursion (take 2)
Running time: \( O(n^2) \), Space: \( \Theta(n^2) \)
Memoizing \texttt{LIS\_smaller}

\begin{verbatim}
\textbf{LIS}(A[1..n]):
\begin{enumerate}
\item \(A[n + 1] = \infty\) (\* add a sentinel at the end \*)
\item Array \(L[(n + 1), (n + 1)]\) (* two-dimensional array*)
\begin{enumerate}
\item (* \(L[i, j]\) for \(j \geq i\) stores the value \texttt{LIS\_smaller}(A[1..i], A[j]) \*)
\item for \(j = 1\) to \(n + 1\) do
\begin{enumerate}
\item \(L[0, j] = 0\)
\end{enumerate}
\item for \(i = 1\) to \(n + 1\) do
\begin{enumerate}
\item for \(j = i\) to \(n + 1\) do
\begin{enumerate}
\item \(L[i, j] = L[i - 1, j]\)
\item if \((A[i] < A[j])\) then
\begin{enumerate}
\item \(L[i, j] = \max(L[i, j], 1 + L[i - 1, i])\)
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{enumerate}
\end{verbatim}

\textbf{Correctness:} Via induction following the recursion (take 2)
\textbf{Running time:} \(O(n^2)\), \textbf{Space:} \(\Theta(n^2)\)
Memoizing \texttt{LIS}\_smaller

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

\begin{align*}
A[n + 1] &= \infty \quad \text{(* add a sentinel at the end *)} \\
\text{Array } L[(n + 1), (n + 1)] \quad \text{(* two-dimensional array*)} \\
\quad \text{(* } L[i, j] \text{ for } j \geq i \text{ stores the value } \text{LIS}\_\text{smaller}(A[1..i], A[j]) \text{ *)}
\end{align*}

\begin{algorithm}
\begin{algorithmic}
\For{$j = 1$ to $n + 1$}
\State $L[0, j] = 0$
\EndFor
\For{$i = 1$ to $n + 1$}
\For{$j = i$ to $n + 1$}
\State $L[i, j] = L[i - 1, j]$
\If{$A[i] < A[j]$}
\State $L[i, j] = \max(L[i, j], 1 + L[i - 1, i])$
\EndIf
\EndFor
\EndFor
\Return $L[n, (n + 1)]$
\end{algorithmic}
\end{algorithm}

\textbf{Correctness:} Via induction following the recursion (take 2)

\textbf{Running time:} $O(n^2)$, \textbf{Space:} $\Theta(n^2)$
Memoizing $\text{LIS}_{\text{smaller}}$

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

$A[n + 1] = \infty$ (* add a sentinel at the end *)

Array $L[(n + 1), (n + 1)]$ (* two-dimensional array*)

(* $L[i, j]$ for $j \geq i$ stores the value $\text{LIS}_{\text{smaller}}(A[1..i], A[j])$ *)

for $j = 1$ to $n + 1$ do
  $L[0, j] = 0$
for $i = 1$ to $n + 1$ do
  for $j = i$ to $n + 1$ do
    $L[i, j] = L[i - 1, j]$
    if ($A[i] < A[j]$) then
      $L[i, j] = \max(L[i, j], 1 + L[i - 1, i])$

return $L[n, (n + 1)]$

Correctness: Via induction following the recursion (take 2)

Running time: $O(n^2)$, Space: $\Theta(n^2)$
Memoizing LIS\_smaller

\textbf{LIS}(A[1..n]):
\begin{itemize}
  \item \(A[n + 1] = \infty\) (* add a sentinel at the end *)
  \item Array \(L[(n + 1), (n + 1)]\) (* two-dimensional array*)
    (* \(L[i, j]\) for \(j \geq i\) stores the value \(\text{LIS\_smaller}(A[1..i], A[j])\)*)
\end{itemize}

\begin{algorithmic}
  \For {\(j = 1\) to \(n + 1\)}
    \State \(L[0, j] = 0\)
  \EndFor

  \For {\(i = 1\) to \(n + 1\)}
    \For {\(j = i\) to \(n + 1\)}
      \State \(L[i, j] = L[i - 1, j]\)
      \If {\(A[i] < A[j]\)}
        \State \(L[i, j] = \text{max}(L[i, j], 1 + L[i - 1, i])\)
      \EndIf
    \EndFor
  \EndFor

  \Return \(L[n, (n + 1)]\)
\end{algorithmic}

\textbf{Correctness:} Via induction following the recursion (take 2)
\textbf{Running time:} \(O(n^2)\), Space: \(\Theta(n^2)\)
Memoizing **LIS\_smaller**

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

\[A[n + 1] = \infty\] (*add a sentinel at the end*)

Array \(L[(n + 1), (n + 1)]\) (*two-dimensional array*)

(*\(L[i, j]\) for \(j \geq i\) stores the value **LIS\_smaller**(\(A[1..i], A[j]\))*)

\[
\text{for } j = 1 \text{ to } n + 1 \text{ do } \\
L[0, j] = 0
\]

\[
\text{for } i = 1 \text{ to } n + 1 \text{ do } \\
\text{for } j = i \text{ to } n + 1 \text{ do } \\
L[i, j] = L[i - 1, j]
\]

\[
\text{if } (A[i] < A[j]) \text{ then } \\
L[i, j] = \max(L[i, j], 1 + L[i - 1, i])
\]

\[
\text{return } L[n, (n + 1)]
\]

**Correctness:** Via induction following the recursion (take 2)

**Running time:** \(O(n^2)\), **Space:** \(\Theta(n^2)\)
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: **6, 3, 5, 2, 7, 8, 1, 9**.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence
Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: $6, 3, 5, 2, 7, 8, 1, 9$. 
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Input sequence: \(6, 3, 5, 2, 7, 8, 1, 9\).
Longest increasing subsequence
Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.

\[ 6 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 7 \rightarrow 8 \rightarrow 1 \rightarrow 9 \]
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence
Another way to get quadratic time algorithm

Input sequence: **6, 3, 5, 2, 7, 8, 1, 9.**
Longest increasing subsequence
Another way to get quadratic time algorithm

Input sequence: $6, 3, 5, 2, 7, 8, 1, 9$. 
Longest increasing subsequence

Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.
Longest increasing subsequence
Another way to get quadratic time algorithm

Input sequence: 6, 3, 5, 2, 7, 8, 1, 9.

Longest increasing subsequence: 3, 5, 7, 8, 9.
Longest increasing subsequence
Another way to get quadratic time algorithm

1. $G = (\{s, 1, \ldots, n\}, \{\})$: directed graph.
   1. $\forall i, j$: If $i < j$ and $A[i] < A[j]$ then add the edge $i \rightarrow j$ to $G$.
   2. $\forall i$: Add $s \rightarrow i$.

2. The graph $G$ is a DAG. LIS corresponds to longest path in $G$ starting at $s$.

3. We know how to compute this in $O(|V(G)| + |E(G)|) = O(n^2)$.

4. Comment: One can compute LIS in $O(n \log n)$ time with a bit more work.
Longest increasing subsequence
Another way to get quadratic time algorithm

1. \( G = (\{s, 1, \ldots, n\}, \emptyset) \): directed graph.

   \( \forall i, j: \) If \( i < j \) and \( A[i] < A[j] \) then add the edge \( i \to j \) to \( G \).

2. \( \forall i: \) Add \( s \to i \).

3. The graph \( G \) is a \textbf{DAG}. \textbf{LIS} corresponds to longest path in \( G \) starting at \( s \).

4. We know how to compute this in \( O(|V(G)| + |E(G)|) = O(n^2) \).

Comment: One can compute \textbf{LIS} in \( O(n \log n) \) time with a bit more work.
Longest increasing subsequence

Another way to get quadratic time algorithm

1. \( G = (\{s, 1, \ldots, n\}, \{\}) \): directed graph.
   
   - \( \forall i, j: \) If \( i < j \) and \( A[i] < A[j] \) then add the edge \( i \rightarrow j \) to \( G \).
   
   - \( \forall i: \) Add \( s \rightarrow i \).

2. The graph \( G \) is a DAG. LIS corresponds to longest path in \( G \) starting at \( s \).

3. We know how to compute this in \( O(|V(G)| + |E(G)|) = O(n^2) \).

4. Comment: One can compute LIS in \( O(n \log n) \) time with a bit more work.
Longest increasing subsequence

Another way to get quadratic time algorithm

1. $G = (\{s, 1, \ldots, n\}, \{\})$: directed graph.
   - $\forall i, j$: If $i < j$ and $A[i] < A[j]$ then add the edge $i \to j$ to $G$.
   - $\forall i$: Add $s \to i$.

2. The graph $G$ is a **DAG**. **LIS** corresponds to longest path in $G$ starting at $s$.

3. We know how to compute this in $O(|V(G)| + |E(G)|) = O(n^2)$.

4. Comment: One can compute **LIS** in $O(n \log n)$ time with a bit more work.
Longest increasing subsequence
Another way to get quadratic time algorithm

1. $G = (\{s, 1, \ldots, n\}, \{\})$: directed graph.
   - $\forall i, j$: If $i < j$ and $A[i] < A[j]$ then add the edge $i \rightarrow j$ to $G$.
   - $\forall i$: Add $s \rightarrow i$.

2. The graph $G$ is a DAG. LIS corresponds to longest path in $G$ starting at $s$.

3. We know how to compute this in $O(|V(G)| + |E(G)|) = O(n^2)$.

4. Comment: One can compute LIS in $O(n \log n)$ time with a bit more work.
Longest increasing subsequence
Another way to get quadratic time algorithm

1. \( G = (\{s, 1, \ldots, n\}, \{\}) \): directed graph.
   - \( \forall i, j: \) If \( i < j \) and \( A[i] < A[j] \) then add the edge \( i \rightarrow j \) to \( G \).
   - \( \forall i: \) Add \( s \rightarrow i \).

2. The graph \( G \) is a DAG. LIS corresponds to longest path in \( G \) starting at \( s \).

3. We know how to compute this in \( O(|V(G)| + |E(G)|) = O(n^2) \).

4. Comment: One can compute LIS in \( O(n \log n) \) time with a bit more work.
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.
Part II

Weighted Interval Scheduling
9.2: Weighted Interval Scheduling
9.2.1: The Problem
Weighted Interval Scheduling

**Input** A set of jobs with start times, finish times and *weights* (or profits).

**Goal** Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

```plaintext
2 1 2 3
1 4 10
10 1 1
1 10
```

2 1 2 3
1 4 10
10 1 1
1 10
**Input** A set of jobs with start times, finish times and *weights* (or profits).

**Goal** Schedule jobs so that total weight of jobs is maximized.

1. Two jobs with overlapping intervals cannot both be scheduled!

---

![Diagram of weighted interval scheduling](image)
9.2.2: Greedy Solution
Interval Scheduling

Greedy Solution

Input  A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.

Goal  Schedule as many jobs as possible.

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).
Interval Scheduling

Greedy Solution

Input  A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.

Goal  Schedule as many jobs as possible.

   Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

---

\[
\text{Input} \quad A \text{ set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.}
\]

\[
\text{Goal} \quad \text{Schedule as many jobs as possible.}
\]

1. Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).

---

\[
\text{Sariel (UIUC)}
\]

\[
\text{OLD CS473} \quad 27 \quad \text{Spring 2015} \quad 27 / 40
\]
Input  A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight \(1\).

Goal  Schedule as many jobs as possible.

Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).
Interval Scheduling

Greedy Solution

**Input**
A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.

**Goal**
Schedule as many jobs as possible.

Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).
Interval Scheduling

Greedy Solution

Input  A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.

Goal  Schedule as many jobs as possible.

Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).
Interval Scheduling

Greedy Solution

Input  A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.

Goal  Schedule as many jobs as possible.

Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).
Interval Scheduling

Greedy Solution

**Input**  A set of jobs with start and finish times to be scheduled on a resource; special case where all jobs have weight 1.

**Goal**  Schedule as many jobs as possible.

Greedy strategy of considering jobs according to finish times produces optimal schedule (to be seen later).
Greedy Strategies

1. Earliest finish time first
2. Largest weight/profit first
3. Largest weight to length ratio first
4. Shortest length first
5. ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don’t work!
Greedy Strategies

1. Earliest finish time first
2. Largest weight/profit first
3. Largest weight to length ratio first
4. Shortest length first
5. ...

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don’t work!
Reduction to...

Max Weight Independent Set Problem

1. Given weighted interval scheduling instance $I$ create an instance of max weight independent set on a graph $G(I)$ as follows.
   - 1. For each interval $i$ create a vertex $v_i$ with weight $w_i$.
   - 2. Add an edge between $v_i$ and $v_j$ if $i$ and $j$ overlap.

2. **Claim:** max weight independent set in $G(I)$ has weight equal to max weight set of intervals in $I$ that do not overlap.
Reduction to...
Max Weight Independent Set Problem

1. Given weighted interval scheduling instance $I$, create an instance of max weight independent set on a graph $G(I)$ as follows.
   1. For each interval $i$ create a vertex $v_i$ with weight $w_i$.
   2. Add an edge between $v_i$ and $v_j$ if $i$ and $j$ overlap.

2. Claim: max weight independent set in $G(I)$ has weight equal to max weight set of intervals in $I$ that do not overlap.
Reduction to...

Max Weight Independent Set Problem

1. Given weighted interval scheduling instance $I$ create an instance of max weight independent set on a graph $G(I)$ as follows.
   1. For each interval $i$ create a vertex $v_i$ with weight $w_i$.
   2. Add an edge between $v_i$ and $v_j$ if $i$ and $j$ overlap.

2. Claim: max weight independent set in $G(I)$ has weight equal to max weight set of intervals in $I$ that do not overlap.
**Reduction to...**

**Max Weight Independent Set Problem**

1. Given weighted interval scheduling instance $I$ create an instance of max weight independent set on a graph $G(I)$ as follows.
   1. For each interval $i$ create a vertex $v_i$ with weight $w_i$.
   2. Add an edge between $v_i$ and $v_j$ if $i$ and $j$ overlap.

2. Claim: max weight independent set in $G(I)$ has weight equal to max weight set of intervals in $I$ that do not overlap.
Given weighted interval scheduling instance $I$ create an instance of max weight independent set on a graph $G(I)$ as follows.

1. For each interval $i$ create a vertex $v_i$ with weight $w_i$.
2. Add an edge between $v_i$ and $v_j$ if $i$ and $j$ overlap.

Claim: max weight independent set in $G(I)$ has weight equal to max weight set of intervals in $I$ that do not overlap.
Reduction to...
Max Weight Independent Set Problem

1. There is a reduction from **Weighted Interval Scheduling** to **Independent Set**.

2. Can use structure of original problem for efficient algorithm?

3. **Independent Set** in general is **NP-Complete**.
Reduction to...
Max Weight Independent Set Problem

1. There is a reduction from \textit{Weighted Interval Scheduling} to \textit{Independent Set}.
2. Can use structure of original problem for efficient algorithm?
3. \textit{Independent Set} in general is \textit{NP-Complete}.
9.2.3: Recursive Solution
Conventions

Definition

1. Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$

2. Define $p(j)$ to be the largest $i$ (less than $j$) such that job $i$ and job $j$ are not in conflict

Example
Conventions

Definition

1. Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$

2. Define $p(j)$ to be the largest $i$ (less than $j$) such that job $i$ and job $j$ are not in conflict

Example
Conventions

Definition

1. Let the requests be sorted according to finish time, i.e., $i < j$ implies $f_i \leq f_j$.

2. Define $p(j)$ to be the largest $i$ (less than $j$) such that job $i$ and job $j$ are not in conflict.

Example

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_2$</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_3$</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_4$</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$v_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

$p(1) = 0$, $p(2) = 0$, $p(3) = 1$, $p(4) = 0$, $p(5) = 3$, $p(6) = 3$
Towards a Recursive Solution

Observation

Consider an optimal schedule $O$

Case $n \in O$ : None of the jobs between $n$ and $p(n)$ can be scheduled. Moreover $O$ must contain an optimal schedule for the first $p(n)$ jobs.

Case $n \notin O$ : $O$ is an optimal schedule for the first $n - 1$ jobs.
Towards a Recursive Solution

Observation

Consider an optimal schedule $\mathcal{O}$

Case $n \in \mathcal{O}$ : None of the jobs between $n$ and $p(n)$ can be scheduled. Moreover $\mathcal{O}$ must contain an optimal schedule for the first $p(n)$ jobs.

Case $n \notin \mathcal{O}$ : $\mathcal{O}$ is an optimal schedule for the first $n - 1$ jobs.
A Recursive Algorithm

Let $O_i$ be value of an optimal schedule for the first $i$ jobs.

\[
\text{Schedule}(n): \\
\begin{align*}
\text{if } n &= 0 \text{ then return } 0 \\
\text{if } n &= 1 \text{ then return } w(v_1) \\
O_{p(n)} &\leftarrow \text{Schedule}(p(n)) \\
O_{n-1} &\leftarrow \text{Schedule}(n-1) \\
\text{if } (O_{p(n)} + w(v_n) < O_{n-1}) \text{ then } & \\
O_n &= O_{n-1} \\
\text{else } & \\
O_n &= O_{p(n)} + w(v_n) \\
\text{return } O_n
\end{align*}
\]

Time Analysis

Running time is $T(n) = T(p(n)) + T(n-1) + O(1)$ which is ...
A Recursive Algorithm

Let $O_i$ be value of an optimal schedule for the first $i$ jobs.

\[
\text{Schedule}(n): \\
\quad \text{if } n = 0 \text{ then return } 0 \\
\quad \text{if } n = 1 \text{ then return } w(v_1) \\
\quad O_{p(n)} \leftarrow \text{Schedule}(p(n)) \\
\quad O_{n-1} \leftarrow \text{Schedule}(n-1) \\
\quad \text{if } (O_{p(n)} + w(v_n) < O_{n-1}) \text{ then} \\
\quad \quad O_n = O_{n-1} \\
\quad \text{else} \\
\quad \quad O_n = O_{p(n)} + w(v_n) \\
\quad \text{return } O_n
\]

Time Analysis

Running time is $T(n) = T(p(n)) + T(n - 1) + O(1)$ which is ...
Bad Example

Figure: Bad instance for recursive algorithm

Running time on this instance is

\[ T(n) = T(n - 1) + T(n - 2) + O(1) = \Theta(\phi^n) \]

where \( \phi \approx 1.618 \) is the golden ratio.
Running time on this instance is

\[ T(n) = T(n - 1) + T(n - 2) + O(1) = \Theta(\phi^n) \]

where \( \phi \approx 1.618 \) is the golden ratio.
Analysis of the Problem

Figure: Label of node indicates size of sub-problem. Tree of sub-problems grows very quickly.
9.2.4: Dynamic Programming
Observation

1. Number of different sub-problems in recursive algorithm is \( O(n) \); they are \( O_1, O_2, \ldots, O_{n-1} \)

2. Exponential time is due to recomputation of solutions to sub-problems

Solution

Store optimal solution to different sub-problems, and perform recursive call only if not already computed.
Recursive Solution with Memoization

\[
schdIMem(j) = 
\begin{cases} 
    0 & \text{if } j = 0 \\
    M[j] & \text{if } M[j] \text{ is defined} \\
    \max \left( w(v_j) + schdIMem(p(j)), schdIMem(j - 1) \right) & \text{if } M[j] \text{ is not defined}
\end{cases}
\]

Time Analysis

- Each invocation, \(O(1)\) time plus: either return a computed value, or generate 2 recursive calls and fill one \(M[\cdot]\).
- Initially no entry of \(M[]\) is filled, at the end all entries of \(M[]\) are filled.
- So total time is \(O(n)\) (Assuming input is presorted...)

Sariel (UIUC) OLD CS473 39 Spring 2015 39 / 40
Recursive Solution with Memoization

\[ \text{schdlMem}(j) \]

\[
\begin{align*}
\text{if } j = 0 & \text{ then return 0} \\
\text{if } M[j] \text{ is defined then } (* \text{ sub-problem already solved } *) & \text{ return } M[j] \\
\text{if } M[j] \text{ is not defined then} & \\
M[j] &= \max\left( w(v_j) + \text{schdlMem}(p(j)), \text{schdlMem}(j - 1) \right) \\
\text{return } M[j]
\end{align*}
\]

Time Analysis

- Each invocation, \( O(1) \) time plus: either return a computed value, or generate 2 recursive calls and fill one \( M[\cdot] \)
- Initially no entry of \( M[\cdot] \) is filled; at the end all entries of \( M[\cdot] \) are filled
- So total time is \( O(n) \) (Assuming input is presorted...)

Sariel (UIUC)
OLD CS473
Spring 2015
Recursive Solution with Memoization

\textbf{schdlMem}(j)

\begin{enumerate}
  \item if \( j = 0 \) then return 0
  \item if \( M[j] \) is defined then (* sub-problem already solved *)
    \begin{enumerate}
      \item return \( M[j] \)
    \end{enumerate}
  \item if \( M[j] \) is not defined then
    \begin{align*}
      M[j] &= \max \left( w(v_j) + \text{schdlMem}(p(j)), \text{schdlMem}(j - 1) \right) \\
      \text{return } M[j]
    \end{align*}
\end{enumerate}

\textbf{Time Analysis}

\begin{itemize}
  \item Each invocation, \( O(1) \) time plus: either return a computed value, or generate 2 recursive calls and fill one \( M[\cdot] \)
  \item Initially no entry of \( M[] \) is filled; at the end all entries of \( M[] \) are filled
  \item So total time is \( O(n) \) (Assuming input is presorted...)
\end{itemize}
Recursive Solution with Memoization

\[
schdIMem(j) =
\begin{cases}
\text{if } j = 0 \text{ then return 0} \\
\text{if } M[j] \text{ is defined then } (*) \text{ sub-problem already solved * } \text{ return } M[j] \\
\text{if } M[j] \text{ is not defined then } \text{return } M[j] = \max \left( w(v_j) + schdIMem(p(j)), \text{ schdIMem}(j - 1) \right)
\end{cases}
\]

Time Analysis

- Each invocation, \( O(1) \) time plus: either return a computed value, or generate 2 recursive calls and fill one \( M[\cdot] \)
- Initially no entry of \( M[\cdot] \) is filled; at the end all entries of \( M[\cdot] \) are filled
- So total time is \( O(n) \) (Assuming input is presorted...)
Recursive Solution with Memoization

```
schdIMem(j)
  if j = 0 then return 0
  if M[j] is defined then (* sub-problem already solved *)
    return M[j]
  if M[j] is not defined then
    M[j] = max \( w(v_j) + \text{schdIMem}(p(j)), \quad \text{schdIMem}(j - 1) \)
  return M[j]
```

Time Analysis

- Each invocation, \( O(1) \) time plus: either return a computed value, or generate 2 recursive calls and fill one \( M[\cdot] \)
- Initially no entry of \( M[\cdot] \) is filled; at the end all entries of \( M[\cdot] \) are filled
- So total time is \( O(n) \) (Assuming input is presorted...)
Automatic Memoization

Fact

Many functional languages (like LISP) automatically do memoization for recursive function calls!
Iterative Solution

\[
\begin{align*}
M[0] &= 0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
M[i] &= \max(w(v_i) + M[p(i)], M[i - 1])
\end{align*}
\]

\textbf{M: table of subproblems}

1. Implicitly dynamic programming fills the values of \( M \).
2. Recursion determines order in which table is filled up.
3. Think of decomposing problem first (recursion) and then worry about setting up table — this comes naturally from recursion.
Back to Weighted Interval Scheduling

Iterative Solution

\[
M[0] = 0 \\
\text{for } i = 1 \text{ to } n \text{ do} \\
M[i] = \max \left( w(v_i) + M[p(i)], M[i - 1] \right)
\]

\(M\): table of subproblems

1. Implicitly dynamic programming fills the values of \(M\).
2. Recursion determines order in which table is filled up.
3. Think of decomposing problem first (recursion) and then worry about setting up table — this comes naturally from recursion.
Example

\[ p(5) = 2, p(4) = 1, p(3) = 1, p(2) = 0, p(1) = 0 \]
9.2.5: Computing Solutions
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[
\begin{align*}
M[0] &= 0 \\
S[0] &= \text{empty schedule} \\
\text{for } i = 1 \text{ to } n \text{ do} \\
M[i] &= \max \left( w(v_i) + M[p(i)], \ M[i - 1] \right) \\
\text{if } w(v_i) + M[p(i)] < M[i - 1] \text{ then} \\
S[i] &= S[i - 1] \\
\text{else} \\
S[i] &= S[p(i)] \cup \{i\}
\end{align*}
\]

1. Naïvely updating \( S[] \) takes \( O(n) \) time
2. Total running time is \( O(n^2) \)
3. Using pointers and linked lists running time can be improved to \( O(n) \).
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[
M[0] = 0 \\
S[0] \text{ is empty schedule} \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad M[i] = \max \left( w(v_i) + M[p(i)], \ M[i - 1] \right) \\
\quad \text{if } w(v_i) + M[p(i)] < M[i - 1] \text{ then} \\
\quad \quad S[i] = S[i - 1] \\
\quad \text{else} \\
\quad \quad S[i] = S[p(i)] \cup \{i\}
\]

1. Naïvely updating \( S[] \) takes \( O(n) \) time
2. Total running time is \( O(n^2) \)
3. Using pointers and linked lists running time can be improved to \( O(n) \).
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[
\begin{align*}
M[0] &= 0 \\
S[0] & \text{is empty schedule} \\
\text{for } i = 1 \text{ to } n \text{ do} \\
M[i] &= \max\left( w(v_i) + M[p(i)], \ M[i - 1] \right) \\
\text{if } w(v_i) + M[p(i)] &< M[i - 1] \text{ then} \\
S[i] &= S[i - 1] \\
\text{else} \\
S[i] &= S[p(i)] \cup \{i\}
\end{align*}
\]

Naïvely updating \( S[] \) takes \( O(n) \) time

Total running time is \( O(n^2) \)

Using pointers and linked lists running time can be improved to \( O(n) \).
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[
\begin{align*}
M[0] &= 0 \\
S[0] &\text{ is empty schedule} \\
\text{for } i = 1 \text{ to } n \text{ do} \\
&\quad M[i] = \max \left( w(v_i) + M[p(i)], \ M[i - 1] \right) \\
&\quad \text{if } w(v_i) + M[p(i)] < M[i - 1] \text{ then} \\
&\qquad S[i] = S[i - 1] \\
&\quad \text{else} \\
&\qquad S[i] = S[p(i)] \cup \{i\}
\end{align*}
\]

Naïvely updating \( S[] \) takes \( O(n) \) time

Total running time is \( O(n^2) \)

Using pointers and linked lists running time can be improved to \( O(n) \).
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[ M[0] = 0 \]
\[ S[0] \text{ is empty schedule} \]
\[ \text{for } i = 1 \text{ to } n \text{ do} \]
\[ M[i] = \max(w(v_i) + M[p(i)], M[i-1]) \]
\[ \text{if } w(v_i) + M[p(i)] < M[i-1] \text{ then} \]
\[ S[i] = S[i-1] \]
\[ \text{else} \]
\[ S[i] = S[p(i)] \cup \{i\} \]

Naïvely updating \( S[] \) takes \( O(n) \) time

Total running time is \( O(n^2) \)

Using pointers and linked lists running time can be improved to \( O(n) \).
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

\[
\begin{align*}
M[0] &= 0 \\
S[0] &\text{ is empty schedule} \\
\text{for } i &= 1 \text{ to } n \text{ do} \\
\quad M[i] &= \max \left( w(v_i) + M[p(i)], \ M[i - 1] \right) \\
\quad \text{if } w(v_i) + M[p(i)] &< M[i - 1] \text{ then} \\
\quad \quad S[i] &= S[i - 1] \\
\quad \text{else} \\
\quad \quad S[i] &= S[p(i)] \cup \{i\}
\end{align*}
\]

Naïvely updating \(S[]\) takes \(O(n)\) time

Total running time is \(O(n^2)\)

Using pointers and linked lists running time can be improved to \(O(n)\).
Observation

*Solution* can be obtained from $M[]$ in $O(n)$ time, without any additional information

\[
\text{findSolution}(j) \\
\text{if } (j = 0) \text{ then return empty schedule} \\
\text{if } (v_j + M[p(j)] > M[j - 1]) \text{ then} \\
\text{return } \text{findSolution}(p(j)) \cup \{j\} \\
\text{else} \\
\text{return } \text{findSolution}(j - 1)
\]

Makes $O(n)$ recursive calls, so *findSolution* runs in $O(n)$ time.
Computing Implicit Solutions

A generic strategy for computing solutions in dynamic programming:

1. Keep track of the *decision* in computing the optimum value of a sub-problem. Decision space depends on recursion.

2. Once the optimum values are computed, go back and use the decision values to compute an optimum solution.

**Question:** What is the decision in computing $M[i]$?

**A:** Whether to include $i$ or not.
Computing Implicit Solutions

A generic strategy for computing solutions in dynamic programming:

1. Keep track of the decision in computing the optimum value of a sub-problem. Decision space depends on recursion.

2. Once the optimum values are computed, go back and use the decision values to compute an optimum solution.

Question: What is the decision in computing $M[i]$?
A: Whether to include $i$ or not.
\( M[0] = 0 \)

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

\( M[i] = \max(v_i + M[p(i)], M[i - 1]) \)

if \( (v_i + M[p(i)] > M[i - 1]) \) then

\( \text{Decision}[i] = 1 \) (* 1: \( i \) included in solution \( M[i] \) *)

else

\( \text{Decision}[i] = 0 \) (* 0: \( i \) not included in solution \( M[i] \) *)

\( S = \emptyset, \ i = n \)

while \( (i > 0) \) do

\( \text{if} \ (\text{Decision}[i] = 1) \) then

\( S = S \cup \{i\} \)

\( i = p(i) \)

else

\( i = i - 1 \)

return \( S \)