Dynamic Programming

Lecture 14
March 10, 2015
Part I

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**.
Example

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

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

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

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

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

**Example**

1. Sequence: $6, 3, 5, 2, 7, 8, 1$
2. Increasing subsequences: $6, 7, 8$ and $3, 5, 7, 8$ and $2, 7$ etc
3. Longest increasing subsequence: $3, 5, 7, 8$
Naïve Enumeration

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

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

Output \( \text{max} \)

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

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 \( A \)

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

Running time:
Naïve Enumeration

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

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

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.
Recursive Approach: Take 1

**LIS**: Longest increasing subsequence

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

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

---

Chandra & Lenny (UIUC)  
CS374  
Spring 2015
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
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
   \( \text{LIS}(A[1..n]) = \text{LIS}(A[1..(n-1)]) \)

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.
Example

Sequence: 6, 3, 5, 2, 7, 8, 1
Recursive Approach: Take 1

\textbf{algLIS}(A[1..n]):
\begin{enumerate}
\item if \((n = 0)\) then return 0
\item \(m = \text{algLIS}(A[1..(n - 1)])\)
\item \(B\) is subsequence of \(A[1..(n - 1)]\) with only elements less than \(A[n]\)
\item (* let \(h\) be size of \(B\), \(h \leq n - 1\) *)
\item \(m = \max(m, 1 + \text{algLIS}(B[1..h]))\)
\end{enumerate}
Output \(m\)
Recursive Approach: Take 1

\textbf{algLIS}(A[1..n]):
\begin{enumerate}
\item if (n = 0) then return 0
\item m = \text{algLIS}(A[1..(n - 1)])
\item B is subsequence of A[1..(n - 1)] with only elements less than A[n]
\item (* let h be size of B, h \leq n - 1 *)
\item m = \max(m, 1 + \text{algLIS}(B[1..h]))
\end{enumerate}
Output m

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

\texttt{algLIS}(A[1..n]):
\begin{itemize}
    \item if \((n = 0)\) then return 0
    \item \(m = \text{algLIS}(A[1..(n - 1)])\)
    \item \(B\) is subsequence of \(A[1..(n - 1)]\) with only elements less than \(A[n]\)
      (* let \(h\) be size of \(B\), \(h \leq n - 1\) *)
    \item \(m = \max(m, 1 + \text{algLIS}(B[1..h]))\)
\end{itemize}
Output \(m\)

Recursion for running time: \(T(n) \leq 2T(n - 1) + O(n)\).
Easy to see that \(T(n)\) is \(O(n^2^n)\).
How many different recursive calls does \( \text{algLIS}_1(A[1..n]) \) really make?

\[
\text{algLIS}(A[1..n]): \\
\text{if } (n = 0) \text{ then return } 0 \\
m = \text{algLIS}(A[1..(n-1)]) \\
B \text{ is subsequence of } A[1..(n-1)] \text{ with} \\
\text{only elements less than } A[n] \\
(* \text{let } h \text{ be size of } B, h \leq n-1 *) \\
m = \max(m, 1 + \text{algLIS}(B[1..h])) \\
\text{Output } m
\]

(A) \( \Theta(n^2) \)
(B) \( \Theta(2^n) \)
(C) \( \Theta(n2^n) \)
(D) \( \Theta(2^{n^2}) \)
(E) \( \Theta(n^n) \)
Recursion 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**

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: Take 2

**LIS**\_**smaller**(A[1..n], x) : length of longest increasing subsequence in A[1..n] with all numbers in subsequence less than x

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

**LIS**(A[1..n]) :
```
return LIS_smallerr(A[1..n], ∞)
```
**Recursive Approach: Take 2**

**LIS_smaller(A[1..n], x)**: length of longest increasing subsequence in A[1..n] with all numbers in subsequence less than x

```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
```

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

```
return LIS_smaller(A[1..n], \infty)
```

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

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

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

**LIS**(A[1..n]):
```
    return LIS\_smaller(A[1..n], \infty)
```

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

**Question**: Is there any advantage?
Example

Sequence: 6, 3, 5, 2, 7, 8, 1
Recursive Algorithm: Take 2

Observation

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

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

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

The number of different subproblems generated by 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 LIS_smaller(A[1..n], x)?
Observation

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

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

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

\[
\text{For } 0 \leq i < n \text{ LIS}\_\text{smaller}(A[1..i], y) \text{ where } y \text{ is either } x \text{ or one of } A[i + 1], \ldots, A[n].
\]
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]) : length of longest increasing sub-sequence that ends in $A[n]$.

Question: can we obtain a recursive expression?
Recursive Algorithm: Take 3

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

$L[7] = 1$
$L[6] = 1 + \max \left\{ L[5], L[4], L[3] \right\}$
$L[5] = 1 + \max \left\{ \right\}$
$L[4] = 1 + \max \left\{ \right\}$
Recursive Algorithm: Take 3

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 \ldots i])
Recursive Algorithm: Take 3

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

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

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

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

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

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

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

\[
\text{LIS\_ending\_alg}(A[1..n]):
\begin{array}{l}
\text{Array } L[1..n] \quad (* \text{L[i]} = \text{value of } \text{LIS\_ending\_alg}(A[1..i]) *) \\
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad L[i] = 1 \\
\quad \text{for } j = 1 \text{ to } i - 1 \text{ do} \\
\qquad \text{if } (A[j] < A[i]) \text{ do} \\
\qquad\qquad \text{L[i] = max}(L[i], 1 + L[j]) \\
\end{array}
\]

\text{return } L

\[
\text{LIS}(A[1..n]):
\begin{array}{l}
\quad L = \text{LIS\_ending\_alg}(A[1..n]) \\
\quad \text{return the maximum value in } L
\end{array}
\]
Iterative Algorithm via Memoization

Simplifying:

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

Simplifying:

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
    - 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
Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

Running time:
Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

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

Simplifying:

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

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

\[
m = 0
\]

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

\[
L[i] = 1
\]

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

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

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

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

\[
\text{return} \ m
\]

Correctness: Via induction following the recursion

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

Simplifying:

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

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

\( m = 0 \)

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

\( L[i] = 1 \)

\( \text{for} \ j = 1 \ \text{to} \ i - 1 \ \text{do} \)

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

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

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

\( \text{return} \ m \)

Correctness: Via induction following the recursion

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

Simplifying:

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

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

\[
m = 0
\]

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

\[
L[i] = 1
\]

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

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

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

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

return \( m \)

Correctness: Via induction following the recursion

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

\( O(n \log n) \) 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

\[
\begin{align*}
L[1] &= 1 \quad \text{(6)} \\
L[2] &= 1 + \max \{L[0] \} = 1 (3) \\
L[3] &= 1 + \max \{L[2] \} = 2 \quad \text{(3.5)} \\
L[4] &= 1 + \max \{3 \} = 1 \\
L[7] &= 1 + \{ \} = 2
\end{align*}
\]
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 \( L[n] \)
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

Two methods

1. **Explicit**: For each subproblem find an optimum solution for that subproblem while computing the optimum value for that subproblem. Typically slow but automatic.

2. **Implicit**: For each subproblem keep track of sufficient information (decision) on how optimum solution for subproblem was computed. Reconstruct optimum solution later via stored information. Typically much more efficient but requires more thought.
**LIS**($A[1..n]$):

Array $L[1..n]$ (* $L[i]$ stores the value **LIS**Ending($A[1..i]$) *)
Array $S[1..n]$ (* $S[i]$ stores the sequence achieving $L[i]$ *)

$m = 0$
$h = 0$

for $i = 1$ to $n$ do

$L[i] = 1$
$S[i] = [i]$

for $j = 1$ to $i - 1$ do


$L[i] = 1 + L[j]$
$S[i] = concat(S[j], [i])$

if ($m < L[i]$) $m = L[i]$, $h = i$

return $m$, $S[h]$
Computing Solution: Explicit method for LIS

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

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

Array **S[1..n]** (* S[i] stores the sequence achieving L[i] *)

m = 0
h = 0

for i = 1 to n do

    L[i] = 1
    S[i] = [i]

    for j = 1 to i - 1 do

        if (A[j] < A[i]) and (L[i] < 1 + L[j]) do

            L[i] = 1 + L[j]
            S[i] = concat(S[j], [i])

        if (m < L[i]) m = L[i], h = i

return m, S[h]

Running time: \(O(n^3)\) Space: \(O(n^2)\). Extra time/space to store, copy
Computing Solution: Implicit method for LIS

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

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

Array \( D[1..n] \)  (* \( D[i] \) stores how \( L[i] \) was computed * )

\( m = 0 \)

\( h = 0 \)

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

\( L[i] = 1 \)

\( D[i] = i \)

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

if \( (A[j] < A[i]) \) and \( (L[i] < 1 + L[j]) \) do

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

\( D[i] = j \)

if \( (m < L[i]) \) \( m = L[i], \ h = i \)

\( m = L[h] \) is optimum value
Computing Solution: Implicit method for LIS

LIS(A[1..n]):

Array L[1..n] (* L[i] stores the value LISEnding(A[1..i]) *)
Array D[1..n] (* D[i] stores how L[i] was computed *)

m = 0
h = 0

for i = 1 to n do

    L[i] = 1
    D[i] = i

    for j = 1 to i - 1 do

        if (A[j] < A[i]) and (L[i] < 1 + L[j]) do
            L[i] = 1 + L[j]
            D[i] = j

    if (m < L[i]) m = L[i], h = i

m = L[h] is optimum value

Question: Can we obtain solution from stored D values and h?
**LIS**(*A[1..n]*):

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

Array **D[1..n]** (* D[i] stores how L[i] was computed *)

**m** = 0, **h** = 0

**for** i = 1 **to** n **do**

- **L[i] = 1**
- **D[i] = 0**

**for** j = 1 **to** i − 1 **do**

  - if (A[j] < A[i]) and (L[i] < 1 + L[j]) do
    - **L[i] = 1 + L[j]**, **D[i] = j**
  - if (m < L[i]) **m = L[i]**, **h = i**

**S = empty sequence**

**while** (h > 0) **do**

  - add **L[h]** to front of **S**
  - **h = D[h]**

Output optimum value **m**, and an optimum subsequence **S**
Computing Solution: Implicit method for LIS

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

Array **L**[1..n] (* L[i] stores the value **LISEnding**(A[1..i]) *)
Array **D**[1..n] (* D[i] stores how L[i] was computed *)

\[ m = 0, \ h = 0 \]

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

\[ L[i] = 1 \]
\[ D[i] = 0 \]

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

if \( (A[j] < A[i]) \) and \( (L[i] < 1 + L[j]) \) do

\[ L[i] = 1 + L[j], \ D[i] = j \]

if \( (m < L[i]) \) \( m = L[i], \ h = i \)

\( S = \) empty sequence

while \( (h > 0) \) do

add \( L[h] \) to front of \( S \)

\[ h = D[h] \]

Output optimum value \( m \), and an optimum subsequence \( S \)

Running time: \( O(n^2) \) Space: \( O(n) \).
Dynamic Programming

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

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

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

4. Optimize the resulting algorithm further.
Part II

Weighted Interval Scheduling
**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!

![Diagram of intervals and weights](image)
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!
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).

---

[Diagram showing jobs on a timeline, with one job highlighted in red.]

---
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.

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.

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

Can use structure of original problem for efficient algorithm?
There is a reduction from Weighted Interval Scheduling to Independent Set.

Can use structure of original problem for efficient algorithm?

Independent Set in general is NP-Complete.
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**

\[
\begin{align*}
1 &: v_1 = 2, \quad p(1) = 0 \\
2 &: v_2 = 4, \quad p(2) = 0 \\
3 &: v_3 = 4, \quad p(3) = 1 \\
4 &: v_4 = 7, \quad p(4) = 0 \\
5 &: v_5 = 2, \quad p(5) = 3 \\
6 &: v_6 = 1, \quad p(6) = 3
\end{align*}
\]
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.
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.

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

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

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

Time Analysis

Running time is $T(n) = T(p(n)) + T(n - 1) + O(1)$ which is ...
The solution to the following recurrence is?

\[ T(n) = T(n - 2) + T(n - 17) + 65 \]

(A) \( 2^{\Theta(n)} \).

(B) \( \Theta(n) \).

(C) 65.

(D) \( \Theta(F_n) \), where \( F_n \) is the \( n \)th Fibonacci number.

(E) \( \Theta(0) \).
Running time on this instance is

\[ T(n) = T(n - 1) + T(n - 2) + O(1) \]
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.
Memo(r)ization

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

```plaintext
schdlMem(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(vj) + schdlMem(p(j)), schdlMem(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[]. 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...)

Recursive Solution with Memoization

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

Time Analysis

- Each invocation, \( O(1) \) time plus: either return a computed value, or generate 2 recursive calls and fill one \( M[\cdot] \)
Recursive Solution with Memoization

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

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
Recursive Solution with Memoization

```plaintext
schdlMem(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) + schdlMem(p(j)), schdlMem(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 $M[]$ is filled; at the end all entries of $M[]$ are filled.
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) + schdIMem(p(j)), \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...)

Fact

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

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

$p(5) = 2, p(4) = 1, p(3) = 1, p(2) = 0, p(1) = 0$
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

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

Na"ively 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?

\[
\begin{align*}
M[0] &= 0 \\
S[0] &= \text{empty schedule} \\
\text{for } i &= 1 \text{ to } n \text{ do} \\
& \quad M[i] = \max (w(v_i) + M[p(i)], M[i - 1]) \\
& \quad \text{if } w(v_i) + M[p(i)] < M[i - 1] \text{ then} \\
& \quad \quad S[i] = S[i - 1] \\
& \quad \text{else} \\
& \quad \quad S[i] = S[p(i)] \cup \{i\}
\end{align*}
\]
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.
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

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

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

Total running time is \(O(n^2)\)
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 \( O(n) \).
Computing Implicit Solutions

Observation

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

```plaintext
findSolution(j)
    if (j = 0) then return empty schedule
    if ($v_j + M[p(j)] > M[j - 1]$) then
        return findSolution(p(j)) ∪ {j}
    else
        return 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 $M[i]$?
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

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

\[S = S \cup \{i\}\]

\[i = p(i)\]

else

\[i = i - 1\]

return \(S\)
Running time with memoization?

If we memoize the following function, what would be the running time of the resulting function, if we call $\text{Confused}(n, n)$?

$$\text{Confused}(x, y)$$
if $x > y$ or $x < 0$ then if $x = 0$ then return $2y$
$\alpha = \text{Confused}(x - 1, y)$, $\beta = \text{Confused}(x - 1, y - 1)$,
$\gamma = \text{Confused}(x - 1, y - 1)$, $\delta = \text{Confused}(x - 1, y - 17)$,
$\mu = \text{Confused}(x - 32, y - 17)$,
return $1 + \max(\alpha, \beta, \gamma, \delta, \mu)$

(A) $\Theta(n)$
(B) $\Theta(n^2)$
(C) $\Theta(n^3)$
(D) $\Theta(n^4)$
(E) $\Theta(n^5)$