Dynamic Programming

Lecture 8
September 23, 2014
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**.
Sequences

Example...

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

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

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

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 \( A \)

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

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]):
    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.
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**?

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

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**?

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

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 **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 **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

\textbf{algLIS}(A[1..n]):
\begin{itemize}
  \item \textbf{if} (n = 0) \textbf{then} return 0
  \item m = \textbf{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 + \textbf{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 \( O(n2^n) \).
Recursive Approach: Take 1

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

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 \( O(n2^n) \).
How many different recursive calls does \textit{algLIS}_1(A[1..n]) really make?

\begin{verbatim}
\textbf{algLIS}(A[1..n]):
    if (n = 0) then return 0
    m = \textbf{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 \leq n-1 *)
    m = max(m, 1 + \textbf{algLIS}(B[1..h]))

Output m
\end{verbatim}

(A) \( \Theta(n^2) \)
(B) \( \Theta(2^n) \)
(C) \( \Theta(n2^n) \)
(D) \( \Theta(2^{n^2}) \)
(E) \( \Theta(n^n) \)
Recursive Approach: Take 2

LIS(A[1..n]):

1. Case 1: Does not contain A[n] in which case
   LIS(A[1..n]) = LIS(A[1..(n − 1)])

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

Observation

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

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

$LIS\_smaller(A[1..n], x)$: 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?
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], ∞)
```

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

- 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 \( \text{LIS\_smaller}(A[1..n], x) \) is \( O(n^2) \).

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

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

- 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)?

- 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.
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 \( \text{LIS}_{\text{smaller}}(A[1..n], x) \)?

1. For \( 0 \leq i < n \) \( \text{LIS}_{\text{smaller}}(A[1..i], y) \) 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)?

- 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.
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?

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

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

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

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

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

```plaintext
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
```

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

**Question:**

How many distinct subproblems generated by \( LIS_ending_alg(A[1..n]) \)? \( n \).
Recursive Algorithm: Take 3

\textbf{LIS\textunderscore ending\textunderscore alg}(A[1..n]) :
  \textbf{if} (n = 0) \textbf{return} 0
  \textit{m} = 1
  \textbf{for} i = 1 \textbf{to} n - 1 \textbf{do}
  \hspace{1em}\textbf{if} (A[i] < A[n]) \textbf{then}
    \hspace{2em} m = \max(m, 1 + \text{LIS\textunderscore ending\textunderscore alg}(A[1..i]))
  \textbf{return} m

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

\textbf{Question:}
How many distinct subproblems generated by \textbf{LIS\textunderscore ending\textunderscore alg}(A[1..n])? n.
Iterative Algorithm via Memoization

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

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

\[
\text{Array } L[1..n] \quad (* \text{ } L[i] = \text{ value of } \text{LIS}_{\text{ending alg}}(A[1..i]) * )
\]

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

\[
L[i] = 1
\]

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

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

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

\[
\text{return } L
\]

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

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

\[
\text{return the maximum value in } L
\]
Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

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

Simplifying:

Let LIS(A[1..n]) be the length of the longest increasing subsequence of A. We define an array L[1..n] where L[i] stores the value LISEnding(A[1..i]).

1. Initialize m = 0
2. For i = 1 to n:
   a. Set L[i] = 1
   b. For j = 1 to i - 1:
      i. If A[j] < A[i], then L[i] = max(L[i], 1 + L[j])
      ii. m = max(m, L[i])
3. Return m

Correctness: Via induction following the recursion

Running time: O(n^2), Space: Θ(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


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

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

return \( m \)

Correctness: Via induction following the recursion

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

Simplifying:

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

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

Simplifying:

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

Correctness: Via induction following the recursion

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

Simplifying:

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

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 \texttt{LIS\_smaller}

\texttt{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*)
  \begin{itemize}
    \item (* $L[i, j]$ for $j \geq i$ stores the value \texttt{LIS\_smaller}(A[1..i], A[j])*)
  \end{itemize}
  \item for $j = 1$ to $n + 1$ do
    \begin{itemize}
      \item $L[0, j] = 0$
    \end{itemize}
  \item for $i = 1$ to $n + 1$ do
    \begin{itemize}
      \item for $j = i$ to $n + 1$ do
        \begin{itemize}
          \item $L[i, j] = L[i - 1, j]$
          \item if ($A[i] < A[j]$) then
            \begin{itemize}
              \item $L[i, j] = \max(L[i, j], 1 + L[i - 1, i])$
            \end{itemize}
        \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

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

\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 $L[i, j] = L[i - 1, j]$
        \STATE $L[i, j] = \max(L[i, j], 1 + L[i - 1, i])$
      \ENDIF
    \ENDFOR
  \ENDFOR
  \STATE \text{return } L[n, (n + 1)]
\end{algorithmic}

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

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

\texttt{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*),
    \begin{itemize}
    \item (* $L[i, j]$ for $j \geq i$ stores the value $\texttt{LIS\_smaller}(A[1..i], A[j])$ *)
    \end{itemize}
\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 $L[i, j] = \max(L[i, j], 1 + L[i - 1, i])$
            \EndIf
        \EndFor
    \EndFor
    \Return $L[n, (n + 1)]$
\end{algorithmic}

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

\texttt{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*} *) \\
&(* \text{ } L[i, j] \text{ for } j \geq i \text{ stores the value } \text{LIS\_smaller}(A[1..i], A[j]) \text{ } *) \\
\text{for } j = 1 \text{ to } n + 1 \text{ do} \\
&L[0, j] = 0 \\
\text{for } i = 1 \text{ to } n + 1 \text{ do} \\
&\quad \text{for } j = i \text{ to } n + 1 \text{ do} \\
&\quad \quad L[i, j] = L[i - 1, j] \\
&\quad \quad \text{if } (A[i] < A[j]) \text{ then} \\
&\quad \quad \quad L[i, j] = \max(L[i, j], 1 + L[i - 1, i]) \\
\text{return } L[n, (n + 1)]
\end{align*}

Correctness: Via induction following the recursion (take 2)
Running time: \textbf{O}(n^2), \text{ 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*)
\quad (* $L[i, j]$ for $j \geq i$ stores the value LIS\_smaller($A[1..i], A[j]$) *)
\end{itemize}

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

\textbf{return} $L[n, (n + 1)]$
\end{verbatim}

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

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

\textbf{Correctness:} Via induction following the recursion (take 2)
\textbf{Running time:} \(O(n^2)\), \textbf{Space:} \(\Theta(n^2)\)
Longest increasing subsequence
Another way to get quadratic time algorithm

1. $G = (\{s, 1, \ldots, n\}, \emptyset)$: directed graph.
   \begin{align*}
   &\forall i, j: \text{If } i < j \text{ and } A[i] < A[j] \text{ then} \\
   &\quad \text{add the edge } i \rightarrow j \text{ to } G. \\
   &\forall i: \text{Add } s \rightarrow i. \\
   
   \end{align*}

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 $O(|V(G)| + |E(G)|) = O(n^2)$.

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

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).
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).
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. Largest weight/profit first
2. Largest weight to length ratio first
3. Shortest length first
4. ... 

None of the above strategies lead to an optimum solution.

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

1. Largest weight/profit first
2. Largest weight to length ratio first
3. Shortest length first
4. ... 

None of the above strategies lead to an optimum solution.

Moral: Greedy strategies often don’t work!
There is a polynomial time reduction from **Weighted Interval Scheduling** to **Independent Set**. Assume **Independent Set** cannot be solved in polynomial time.

It follows that **Weighted Interval Scheduling** cannot be solved in polynomial time. This statement is

(A) True

(B) False.

(C) IDK.
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

\[
\begin{align*}
1 & \quad v_1 = 2 \\
2 & \quad v_2 = 4 \\
3 & \quad v_3 = 4 \\
4 & \quad v_4 = 7 \\
5 & \quad v_5 = 2 \\
6 & \quad v_6 = 1 \\
\end{align*}
\]

\[
\begin{align*}
p(1) &= 0 \\
p(2) &= 0 \\
p(3) &= 1 \\
p(4) &= 0 \\
p(5) &= 3 \\
p(6) &= 3 \\
\end{align*}
\]
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 $O$

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.
A Recursive Algorithm

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

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

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.

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

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) \( \Theta(0) \).
Bad Example

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.
**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)
    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)), 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...)

Recursive Solution with Memoization

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

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...)
Recursive Solution with Memoization

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

Time Analysis

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

Alexandra (UIUC)  
CS473  
Fall 2014  
33 / 45
Recursive Solution with Memoization

```plaintext
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(vj) + schdIMem(p(j)), 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 $M[·]$
- 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

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

Time Analysis

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

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

\(M\): 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.
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.
$p(5) = 2, \ p(4) = 1, \ p(3) = 1, \ p(2) = 0, \ p(1) = 0$
Computing Solutions + First Attempt

1. Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

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

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

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

4. 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?

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

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 \( O(n) \).
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

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

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 \( O(n) \).
Memoization + Recursion/Iteration allows one to compute the optimal value. What about the actual schedule?

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

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

2. Naïvely updating \( S[] \) takes \( O(n) \) time
3. Total running time is \( O(n^2) \)
4. 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?

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

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

```python
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 $\text{findSolution}$ runs in $O(n)$ time.
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.
Computing Implicit Solutions

\[ M[0] = 0 \]

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

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

else

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

\[ S = \emptyset, \ i = n \]

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 \textbf{Confused}(n, n)?

\begin{verbatim}
Confused(x, y)
    if x > y or x < 0 then if x = 0 then return 2y
    α = Confused(x - 1, y), β = Confused(x - 1, y - 1),
    γ = Confused(x - 1, y - 1), δ = Confused(x - 1, y - 17),
    μ = Confused(x - 32, y - 17),
    return 1 + max(α, β, γ, δ, μ)
\end{verbatim}

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