BBM402-Lecture 4: Dynamic Programming:
Longest Increasing Subsequence, String splitting

Lecturer: Lale Özkahya

Resources for the presentation:
https://courses.engr.illinois.edu/cs374/fa2016/lectures.html
https://courses.engr.illinois.edu/cs473/fa2016/lectures.html
https://courses.engr.illinois.edu/cs374/fa2015/lectures.html
Fibonacci

• Fibonacci Numbers (circa 13th century)

<table>
<thead>
<tr>
<th>( F_n = )</th>
<th>0 if ( n = 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 if ( n = 1 )</td>
</tr>
<tr>
<td></td>
<td>( F_{n-1} + F_{n-2} ) o/w</td>
</tr>
</tbody>
</table>

Given \( n \), how long does it take to compute \( F_n \)?
Fibonacci

• Translates line by line to code:

```python
RecFibo(n):
    if (n < 2)
        return n
    else
        return RecFibo(n - 1) + RecFibo(n - 2)
```

We will move from mathematical function format to recursive program a lot!
Fibonacci

- Translates line by line to code:

```python
RecFIBO(n):
    if (n < 2)
        return n
    else
        return RecFIBO(n - 1) + RecFIBO(n - 2)
```

Running time? (backtracking recurrence)

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

\[
= \Theta(F_n) = \Theta(1.618^n) = \Theta(((\sqrt{5}+1)/2)^n)
\]
Leaves are always 0 or 1.
How many 1’s? How many 0s?
There are $F_n$ 1s and $F_{n-1}$ 0s
$F_{n+1}$ leaves total!
Running time via Rec Tree

How many intermediate nodes does a full binary tree with \( m \) leaves have?

\[
F_n = F_{n-1} + F_{n-2} + F_{n-3} + F_{n-4}
\]

\[
\begin{align*}
F_0 &= 0 \\
F_1 &= 1 \\
F_2 &= 1 \\
F_3 &= 0 \\
F_4 &= 1 \\
\end{align*}
\]
Running time via Rec Tree

\[ 2F_{n+1} - 1 \text{ nodes (additions)} \]
Running time via Rec Tree

F_5

F_4

F_3

F_2

F_1

F_0
Running time via Rec Tree

Keep an array to remember the previous values!
Running time via Rec Tree

```
F[5]
F[4]
F[3]
F[3]
F[2]
F[2]
F[1]
F[2]
F[1]
F[1]
F[0]
F[1]
F[0]
F[0]
F[0]
F[0]
F[0]
...
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Running time via Rec Tree

look up array for $F_2$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Running time via Rec Tree

look up array for $F_3$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Memoization = when I look at the table to see the values I computed before
Given any recursive backtracking algorithm, you can add memoization and will save time, provided the subproblems repeat.
How many times did I have to call the recursive function? 
exponential!

How many different values did I have to compute? 
O(n)!

Memoization decreases running time : performs only O(n) 
additions, exponential improvement
Memoized algorithm fills in the table from left to right. Why not just do that?
Memoized algorithm fills in the table from left to right. Why not just do that?

We get an iterative algorithm

\[
\text{IterFibo}(n):
\]

\[
\begin{align*}
F[0] & \leftarrow 0 \\
F[1] & \leftarrow 1 \\
\text{for } i & \leftarrow 2 \text{ to } n \\
F[i] & \leftarrow F[i-1] + F[i-2] \\
\text{return } F[n]
\end{align*}
\]
• Clear that the number of additions it does it $O(n)$.
• In practice this is faster than memoized algo, cause we don’t use stack/ look up the table etc.
• Structure mirrors the recurrence
• Only subtle thing is that we want to fill in the array in increasing order.
• This is Dynamic Programming Algorithm!
• Dynamic Programming = pretend to do Memoization but do it on purpose

• Memoization: accidentally use something efficient
• Backwards induction = Dynamic Programming

`IterFIBO(n):
    F[0] ← 0
    F[1] ← 1
    for i ← 2 to n
        F[i] ← F[i - 1] + F[i - 2]
    return F[n]`
Dynamic Programming

- Dynamic programming is about smart recursion.
- Not about filling out tables!
- How do I solve the problem, how do I not repeat work, then how to fill up my data structure.
Dynamic Programming

• How can I speed up my algorithm?

```
ITERFIBO(n):
    F[0] ← 0
    F[1] ← 1
    for i ← 2 to n
        F[i] ← F[i − 1] + F[i − 2]
    return F[n]
```

• I only need to keep my last two elements of the array.
• Even more efficient algorithm
Dynamic Programming

• How can I speed up my algorithm?

ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
    next ← curr + prev
    prev ← curr
    curr ← next
  return curr

• I only need to keep my last two elements of the array.
• Even more efficient algorithm
• Where is the recursion?
Dynamic Programming

- I only need to keep my last two elements of the array.
- Even more efficient algorithm
- Where is the recursion?
- Saves space, sometimes important

IterFIBO2(n):

pre 1
curr 0
for i ← 1 to n
    next ← curr + prev
    prev ← curr
    curr ← next
return curr

- How can I speed up my algorithm?
Dynamic Programming

• How can I speed up my algorithm?

```plaintext
ITERFIBO2(n):
prev ← 1
curr ← 0
for i ← 1 to n
   next ← curr + prev
   prev ← curr
   curr ← next
return curr
```

• Is this the fastest Algorithm for Fibonacci?
Dynamic Programming

- How can I speed up my algorithm?

```
IterFIBO2(n):
    prev ← 1
    curr ← 0
    for i ← 1 to n
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

This matrix vector multiplication does exactly the same thing as one iteration of the loop!

\[
\begin{bmatrix}
0 & 1 \\
1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
\end{bmatrix}
= 
\begin{bmatrix}
y \\
x + y \\
\end{bmatrix}
\]

What to do to compute the nth Fibonacci number?
Dynamic Programming

• How can I speed up my algorithm?

Compute the \( n \)th power of the matrix.

\[
\begin{bmatrix}
0 & 1 \\
1 & 1 \\
\end{bmatrix}^n \begin{bmatrix}
1 \\
0 \\
\end{bmatrix} = \begin{bmatrix}
F_{n-1} \\
F_n \\
\end{bmatrix} \quad \begin{bmatrix}
0 & 1 \\
1 & 1 \\
\end{bmatrix} \begin{bmatrix}
x \\
y \\
\end{bmatrix} = \begin{bmatrix}
y \\
x + y \\
\end{bmatrix}
\]

Compute the \( n \)th power of the matrix.

• With repeated squaring, \( O(\log n) \) multiplications
• Compute \( F_n \) in \( O(\log n) \) arithmetic operations
• Double exponential speedup!
Dynamic Programming

- How can I speed up my algorithm?

\[
\begin{bmatrix}
0 & 1 \\
1 & 1
\end{bmatrix}^n \begin{bmatrix}
1 \\
0
\end{bmatrix} = \begin{bmatrix}
F_{n-1} \\
F_n
\end{bmatrix} \quad \begin{bmatrix}
0 & 1 \\
1 & 1
\end{bmatrix} \begin{bmatrix}
x \\
y
\end{bmatrix} = \begin{bmatrix}
y \\
x + y
\end{bmatrix}
\]

Compute the \( n \)th power of the matrix.

- But how many bits is the \( n \)th Fibonacci number?
- \( O(n)! \)
- Can’t perform arbitrary precision arithmetic in constant time
Longest Increasing Subsequence (LIS)

- 31 4 1 5 9 2 6 5 3 8 2 7 9 4 6 1 0 4 8
Longest Increasing Subsequence (LIS)

- 3 1 4 1 5 9 2 6 5 3 8 2 7 9 4 6 1 0 4 8

- LIS(A[1…n],p) = length of LIS of A[1…n] where everything is bigger than p
Longest Increasing Subsequence (LIS)

- $\pi = 3.1415926538279461048$ 

- $\text{LIS}(A[1\ldots n], p) =$
  
  \[ \begin{cases} 
  0 & \text{if } n = 0 \\
  \text{LIS}(A[2\ldots n], p) & \text{if } A[1] \leq p \\
  \max \{ \text{LIS}(A[2\ldots n], p) \, 1+\text{LIS}(A[2\ldots n], A[1]) \} 
  \end{cases} \]
Longest Increasing Subsequence (LIS)

\[
\text{LIS}(A[1\ldots n], p) = \begin{cases} 
0 & \text{if } n = 0 \\
\text{LIS}(A[2\ldots n], p) & \text{if } A[1] \leq p \\
\max \{ \text{LIS}(A[2\ldots n], p), 1 + \text{LIS}(A[2\ldots n], A[1]) \} & \text{otherwise}
\end{cases}
\]

• The argument \( p \) is always either \(-\infty\) or an element of the array \( A \)
• Add \( A[0] = -\infty \)
• We can identify any recursive subproblem with two array indices.
• \( \text{LIS}(i, j) = \text{length or LIS of } A[j\ldots n] \text{ with all elements larger than } A[i] \)
Longest Increasing Subsequence (LIS)

For $i<j$

$\text{LIS}(i, j) = \begin{cases} 
0 & \text{if } j > n \\
\text{LIS}(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{\text{LIS}(i, j + 1), 1 + \text{LIS}(j, j + 1)\} & \text{otherwise}
\end{cases}$

- $\text{LIS}(i,j) = \text{length or LIS of } A[j...n] \text{ with all elements larger than } A[i]$
- We want to compute $\text{LIS}(0,1)$
- Memoize? what data structure to use?
- Two dimensional Array $\text{LIS}[0...n, 1...n+1]$
For $i < j$

$LIS(i, j) = \begin{cases} 
0 & \text{if } j > n \\
LIS(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{LIS(i, j + 1), 1 + LIS(j, j + 1)\} & \text{otherwise}
\end{cases}$
For $i < j$

$$LIS(i, j) = \begin{cases} 
0 & \text{if } j > n \\
LIS(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{LIS(i, j + 1), 1 + LIS(j, j + 1)\} & \text{otherwise}
\end{cases}$$

Figure out an order to fill out the table that works!
For $i < j$

\[
LIS(i, j) = \begin{cases} 
  0 & \text{if } j > n \\
  LIS(i, j + 1) & \text{if } A[i] \geq A[j] \\
  \max\{LIS(i, j + 1), 1 + LIS(j, j + 1)\} & \text{otherwise}
\end{cases}
\]
For $i < j$

$$LIS(i, j) = \begin{cases} 
0 & \text{if } j > n \\
LIS(i, j+1) & \text{if } A[i] \geq A[j] \\
\max\{LIS(i, j+1), 1 + LIS(j, j+1)\} & \text{otherwise}
\end{cases}$$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th></th>
<th>j</th>
<th>n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Purple squares must be filled before pink
For $i < j$

\[
LIS(i, j) = \begin{cases} 
0 & \text{if } j > n \\
LIS(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{LIS(i, j + 1), 1 + LIS(j, j + 1)\} & \text{otherwise}
\end{cases}
\]
**Longest Increasing Subsequence (LIS)**

\[
\text{LIS}(A[1..n]):
A[0] \leftarrow -\infty \\
\text{for } i \leftarrow 0 \text{ to } n \\
\quad \text{LIS}[i, n+1] \leftarrow 0 \\
\text{for } j \leftarrow n \text{ downto } 1 \\
\quad \text{for } i \leftarrow 0 \text{ to } j-1 \\
\quad \quad \text{if } A[i] \geq A[j] \\
\quad \quad \quad \text{LIS}[i, j] \leftarrow \text{LIS}[i, j+1] \\
\quad \quad \quad \text{else} \\
\quad \quad \quad \text{LIS}[i, j] \leftarrow \max\{\text{LIS}[i, j+1], 1 + \text{LIS}[j, j+1]\}
\]

doesn’t matter what order I fill the columns in
Longest Increasing Subsequence (LIS)

- Running time?
- \(O(n^2)\)
- Two nested for loops
- How many values are there in the recurrence?

```latex
LIS(A[1..n]):
A[0] \leftarrow -\infty
\langle Add a sentinel \rangle
for i \leftarrow 0 \text{ to } n
\langle Base cases \rangle
\quad LIS[i, n+1] \leftarrow 0
\quad for j \leftarrow n \text{ downto } 1
\quad \quad for i \leftarrow 0 \text{ to } j-1
\quad \quad \quad if A[i] \geq A[j]
\quad \quad \quad \quad LIS[i, j] \leftarrow LIS[i, j+1]
\quad \quad \quad else
\quad \quad \quad \quad LIS[i, j] \leftarrow \max\{LIS[i, j+1], 1 + LIS[j, j+1]\}
return LIS[0, 1]
```
Longest Increasing Subsequence (LIS)

For $i < j$

$$\text{LIS}(i, j) = \begin{cases} 
0 & \text{if } j > n \\
\text{LIS}(i, j + 1) & \text{if } A[i] \geq A[j] \\
\max\{\text{LIS}(i, j + 1), 1 + \text{LIS}(j, j + 1)\} & \text{otherwise}
\end{cases}$$

• As general rule of thumb:
• # variables on the left = space $O(n^2)$ array for $i, j$ taking $n$ values each
• # variables on the right = time $O(n^2)$
Dynamic Programming
General Recipe for DP

- **Step 1**: Find Backtracking Recursive algorithm (e.g. for LIS we leveraged the recursive def. Either empty or there is something that comes first) (6 pts)

- **Step 2**: Identify the subproblems (e.g. indices i,j for LIS), need english description

- **Step 3**: Analyze time and space

- **Step 4**: Choose a memoization data structure (e.g. two dim array)

- **Step 5**: Find evaluation order (draw picture!!!)
Dynamic Programming

General Recipe for DP

• **Step 3**: Analyze time and space

• **Step 6**: write iterative pseudocode
Dynamic Programming is \textit{smart recursion plus memoization}.
Dynamic Programming

Dynamic Programming is smart recursion plus memoization

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

- On input of size $n$ the number of *distinct* sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time *not counting* the time for its recursive calls.

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

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

Chandra & Manoj (UIUC) CS374 Fall 2015
Dynamic Programming is smart recursion plus memoization

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

- On input of size $n$ the number of distinct sub-problems that $\text{foo}(x)$ generates is at most $A(n)$
- $\text{foo}(x)$ spends at most $B(n)$ time *not counting* the time for its recursive calls.

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

Chandra & Manoj (UIUC)
CS374
Fall 2015
2 / 32
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
Can we find a recursive algorithm for $\text{LIS}$?

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

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 $LIS_{\text{smaller}}(A[1..n], x)$ which gives the longest increasing subsequence in $A$ where each number in the sequence is less than $x$. 

Chandra & Manoj (UIUC) CS374 Fall 2015 7
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**

*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

LIS(A[1..n]): the length of longest increasing subsequence in A

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

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

if (n = 0) then return 0

m = LIS\_smaller(A[1..(n − 1)], x)

if (A[n] < x) then

    m = max(m, 1 + LIS\_smaller(A[1..(n − 1)], A[n]))

Output m

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

return LIS\_smaller(A[1..n], \infty)
Sequence: $A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9$
Recursive Approach

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

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

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

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

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

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

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

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

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

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

Output \( m \)

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

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

- What is the running time if we memoize recursion? \( O(n^2) \) since each call takes \( O(1) \) time to assemble the answers from to recursive calls and no other computation.
Recursive Approach

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

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

- How many distinct sub-problems will \( \text{LIS\_smaller}(A[1..n], \infty) \) generate? \( O(n^2) \)
- What is the running time if we memoize recursion? \( O(n^2) \) since each call takes \( O(1) \) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization?
Recursive Approach

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

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

- How many distinct sub-problems will \text{LIS\_smaller}(A[1..n], \infty) generate? \(O(n^2)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(1)\) time to assemble the answers from recursive calls and no other computation.
- How much space for memoization? \(O(n^2)\)
Recursive Algorithm: Take 2

**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?

\[ \text{LISE}(A[1..8]) = 4 \quad (3,5,7,8,9) \]
\[ \text{LISE}(A[1..7]) = 1 \quad (1) \]
\[ (A[1..6]) = 3 \quad (3,5,7,8) \]
Definition

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

Question: can we obtain a recursive expression?

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

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

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

\[
\text{LIS}(A[1..n]): \\
\quad \text{return } \max_{i=1}^{n} \text{LIS}_\text{ending\_alg}(A[1 \ldots i])
\]
Recursive Algorithm: Take 2

LIS\_ending\_alg(A[1..n]):
\hspace{1em} if (n = 0) return 0
\hspace{1em} m = 1
\hspace{1em} for i = 1 to n − 1 do
\hspace{2em} if (A[i] < A[n]) then
\hspace{3em} \hspace{1em} m = \max(m, 1 + LIS\_ending\_alg(A[1..i]))
\hspace{1em} return m

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

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

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

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

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

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

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

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

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

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

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

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

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

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

LIS\_ending\_alg(A[1..n]):

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

LIS(A[1..n]):

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

- How many distinct sub-problems will LIS\_ending\_alg(A[1..n]) generate? \(O(n)\)
- What is the running time if we memoize recursion? \(O(n^2)\) since each call takes \(O(n)\) time
- How much space for memoization? \(O(n)\)
Removing recursion to obtain iterative algorithm

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

Why?
Removing recursion to obtain iterative algorithm

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

Why? Mainly for further optimization of running time and space.
Removing recursion to obtain iterative algorithm

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

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.
Removing recursion to obtain iterative algorithm

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

Why? Mainly for further optimization of running time and space.

How?

- First, allocate a data structure (usually an array or a multi-dimensional array that can hold values for each of the subproblems)
- Figure out a way to order the computation of the sub-problems starting from the base case.

Caveat: Dynamic programming is not about filling tables. It is about finding a smart recursion. First, find the correct
Iterative Algorithm via Memoization

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

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

\text{return} \( L \)

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

Simplifying:

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

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

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

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

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.

Chandra & Manoj (UIUC)
CS374 16 Fall 2015 16 / 32
Iterative Algorithm via Memoization

Simplifying:

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

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

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

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

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

return \(m\)

Correctness: Via induction following the recursion

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

Space:
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 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: \( \Theta(n) \)

\( O(n \log n) \) 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

\[ L[i] = \text{Lengthening} (A[1..i]) \]

\[ L[1] = 1 \]
\[ L[2] = \max (L[1], 1 + 0) = 1 \]
\[ L[3] = \max (1, 1 + L[2]) = 1 \]
\[ L[4] = \max (1, 1 + 0) \]
\[ L[5] = \]
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 sub-sequence?

Two methods

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

**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.
Computing Solution: Explicit method for LIS

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

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

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

\( m = 0 \)

\( h = 0 \)

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

\( L[i] = 1 \)

\( S[i] = [i] \)

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

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

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

\( S[i] = \text{concat}(S[j], [i]) \)

\textbf{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

Chandra & Manoj (UIUC) CS374 19 Fall 2015
**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] *)

\[
\begin{align*}
m &= 0 \\
h &= 0 \\
\text{for } i &= 1 \text{ to } n \text{ do} \\
\quad L[i] &= 1 \\
\quad S[i] &= [i] \\
\quad \text{for } j &= 1 \text{ to } i - 1 \text{ do} \\
\quad \quad \text{if } (A[j] < A[i]) \text{ and } (L[i] < 1 + L[j]) \text{ do} \\
\quad \quad \quad L[i] &= 1 + L[j] \\
\quad \quad \quad S[i] &= \text{concat}(S[j], [i]) \\
\quad \quad \end{align*}
\]

\[
\text{if } (m < L[i]) \quad m = L[i], \quad h = i
\]

\[
\text{return } m, \quad S[h]
\]

**Running time:** \(O(n^3)\)  **Space:** \(O(n^2)\). Extra time/space to store, copy.
**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


$L[i] = 1 + L[j]$

$D[i] = j$

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

$m = L[h]$ is optimum value
**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 *)

\[ m = 0 \]

\[ h = 0 \]

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

\[ L[i] = 1 \]

\[ D[i] = i \]

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

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

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

\[ D[i] = j \]

\textbf{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 \( h \)?
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).

Chandra & Manoj (UIUC)
Computing Solution: Implicit method for LIS

**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

\(L[i] = 1\)

\(D[i] = 0\)

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

\(S = \text{empty sequence}\)

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

Checking if string in $L^*$
Problem

Input A string $w \in \Sigma^*$ and access to a language $L \subseteq \Sigma^*$ via function $\text{IsStringinL}(\text{string } x)$ that decides whether $x$ is in $L$.

Goal Decide if $w \in L^*$ using $\text{IsStringinL}(\text{string } x)$ as a black box sub-routine.

Example

Suppose $L$ is English and we have a procedure to check whether a string/word is in the English dictionary.

- Is the string “isthisanenglishsentence” in English*?
- Is “stampstamp” in English*?
- Is “zibzzzad” in English*?
Recursive Solution

When is $w \in L^*$?

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

IsStringinLstar($A[1..n]$):
If (IsStringinL($A[1..n]$)) Output YES
Else
For ($i = 1$ to $n - 1$) do
If (IsStringinL($A[1..i]$) and IsStringinLstar($A[i + 1..n]$)) Output YES
Output NO
Recursive Solution

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

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

When is $w \in L^*$?

$w \in L^*$ if $w \in L$ or if $w = uv$ where $u \in L$ and $v \in L^*$

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

```plaintext
IsStringinLstar(A[1..n]):
    If (IsStringinL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n - 1) do
            If (IsStringinL(A[1..i]) and IsStringinLstar(A[i + 1..n]))
                Output YES
        Output NO
```
Recursive Solution

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

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

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

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

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

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

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

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

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

- How many distinct sub-problems does \( \text{IsStringinLstar}(A[1..n]) \) generate? \( O(n) \)
- What is running time of memoized version of \( \text{IsStringinLstar}(A[1..n]) \)?
Recursive Solution

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

\[
\text{IsStringinLstar}(A[1..n]):
\]
\[
\text{If (IsStringinL}(A[1..n]))
\]
\[
\text{Output YES}
\]
\[
\text{Else}
\]
\[
\text{For (i = 1 to n − 1) do}
\]
\[
\text{If (IsStringinL}(A[1..i]) \text{ and IsStringinLstar}(A[i + 1..n]))
\]
\[
\text{Output YES}
\]
\[
\text{Output NO}
\]

- How many distinct sub-problems does \( \text{IsStringinLstar}(A[1..n]) \) generate? \( O(n) \)
- What is running time of memoized version of \( \text{IsStringinLstar}(A[1..n]) \)? \( O(n^2) \)
Assume \( w \) is stored in array \( A[1..n] \)

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

\[
\text{If } (\text{IsStringinL}(A[1..n])) \n\]

\[
\text{Output YES} \n\]

\[
\text{Else} \n\]

\[
\text{For } (i = 1 \text{ to } n - 1) \text{ do} \n\]

\[
\text{If } (\text{IsStringinL}(A[1..i]) \text{ and } \text{IsStringinLstar}(A[i + 1..n])) \n\]

\[
\text{Output YES} \n\]

\[
\text{Output NO} \n\]

- How many distinct sub-problems does \( \text{IsStringinLstar}(A[1..n]) \) generate? \( O(n) \)
- What is running time of memoized version of \( \text{IsStringinLstar}(A[1..n]) \)? \( O(n^2) \)
- What is space requirement of memoized version of \( \text{IsStringinLstar}(A[1..n]) \)?
Recursive Solution

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

IsStringinLstar($A[1..n]$):
  If (IsStringinL($A[1..n]$))
    Output YES
  Else
    For ($i = 1$ to $n - 1$) do
      If (IsStringinL($A[1..i]$) and IsStringinLstar($A[i+1..n]$))
        Output YES
    Output NO

- How many distinct sub-problems does IsStringinLstar($A[1..n]$) generate? $O(n)$
- What is running time of memoized version of IsStringinLstar($A[1..n]$)? $O(n^2)$
- What is space requirement of memoized version of IsStringinLstar($A[1..n]$)? $O(n)$
A variation

**Input** A string \( w \in \Sigma^* \) and access to a language \( L \subseteq \Sigma^* \) via function \( \text{IsStringinL}(\text{string } x) \) that decides whether \( x \) is in \( L \), and non-negative integer \( k \)

**Goal** Decide if \( w \in L^k \) using \( \text{IsStringinL}(\text{string } x) \) as a black box sub-routine

**Example**

Suppose \( L \) is **English** and we have a procedure to check whether a string/word is in the **English** dictionary.

- Is the string “isthisanenglishsentence” in **English**\(^5\)?
- Is the string “isthisanenglishsentence” in **English**\(^4\)?
- Is “asinineat” in **English**\(^2\)?
- Is “asinineat” in **English**\(^4\)?
- Is “zibzzzzad” in **English**\(^1\)?
Recursive Solution

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

- \( k = 0 \): \( w \in L^k \) iff \( w = \epsilon \)
- \( k = 1 \): \( w \in L^k \) iff \( w \in L \)
- \( k > 1 \): \( w \in L^k \) if \( w = uv \) with \( u \in L \) and \( v \in L^{k-1} \)
When is \( w \in L^k \)?

- \( k = 0 \): \( w \in L^k \) iff \( w = \epsilon \)
- \( k = 1 \): \( w \in L^k \) iff \( w \in L \)
- \( k > 1 \): \( w \in L^k \) if \( w = uv \) with \( u \in L \) and \( v \in L^{k-1} \)
Recursive Solution

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

\( k = 0: \ w \in L^k \iff w = \epsilon \)

\( k = 1: \ w \in L^k \iff w \in L \)

\( k > 1: \ w \in L^k \) if \( w = uv \) with \( u \in L \) and \( v \in L^{k-1} \)

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

\[
\text{IsStringinLk}(A[1..n], k):
\]

- If (\( k = 0 \))
  - If (\( n = 0 \)) Output YES
  - Else Output NO

- If (\( k = 1 \))
  - Output \( \text{IsStringinL}(A[1..n]) \)

- Else
  - For (\( i = 1 \) to \( n - 1 \)) do
    - If (\( \text{IsStringinL}(A[1..i]) \) and \( \text{IsStringinLk}(A[i + 1..n], k - 1) \))
      - Output YES

- Output NO
Analysis

\textbf{IsStringinLk}(A[1..n], k):
\begin{itemize}
  \item If \((k = 0)\)
    \begin{itemize}
      \item If \((n = 0)\) Output YES
      \item Else Output NO
    \end{itemize}
  \item If \((k = 1)\)
    \begin{itemize}
      \item Output \textbf{IsStringinL}(A[1..n])
    \end{itemize}
  \item Else
    \begin{itemize}
      \item For \((i = 1 \text{ to } n - 1)\) do
        \begin{itemize}
          \item If \((\text{IsStringinL}(A[1..i]) \text{ and IsStringinLk}(A[i + 1..n], k - 1))\)
            \begin{itemize}
              \item Output YES
            \end{itemize}
        \end{itemize}
    \end{itemize}
\end{itemize}

Output NO

\textbullet How many distinct sub-problems are generated by \textbf{IsStringinLk}(A[1..n], k)?
IsStringinLk(A[1..n], k):
  If (k = 0)
    If (n = 0) Output YES
    Else Output NO
  If (k = 1)
    Output IsStringinL(A[1..n])
  Else
    For (i = 1 to n − 1) do
      If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
        Output YES
    Output NO

How many distinct sub-problems are generated by IsStringinLk(A[1..n], k)? O(nk)
**Analysis**

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

- How many distinct sub-problems are generated by \( \text{IsStringinLk}(A[1..n], k) \)? \( O(nk) \)
- How much space?
Analysis

IsStringinLk(A[1..n], k):
If (k = 0)
  If (n = 0) Output YES
  Else Output NO
If (k = 1)
  Output IsStringinL(A[1..n])
Else
  For (i = 1 to n - 1) do
    If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k - 1))
      Output YES
  Output NO

- How many distinct sub-problems are generated by IsStringinLk(A[1..n], k)? O(nk)
- How much space? O(nk) pause
- Running time?

Chandra & Manoj (UIUC) CS374 29 Fall 2015 29 / 32
IsStringinLk(A[1..n], k):
If (k = 0)
  If (n = 0) Output YES
  Else Output NO
If (k = 1)
  Output IsStringinL(A[1..n])
Else
  For (i = 1 to n − 1) do
    If (IsStringinL(A[1..i]) and IsStringinLk(A[i + 1..n], k − 1))
      Output YES
  Output NO

How many distinct sub-problems are generated by
IsStringinLk(A[1..n], k)? O(nk)
How much space? O(nk) pause
Running time? O(n^2k)
Question: What if we want to check if $w \in L^i$ for some $0 \leq i \leq k$? That is, is $w \in \bigcup_{i=0}^{k} L^i$?
Exercise

Definition

A string is a palindrome if $w = w^R$.
Examples: I, RACECAR, MALAYALAM, DOOFFOOD

Problem: Given a string $w$ find the longest subsequence of $w$ that is a palindrome.

Example: MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM has MHYMRORMYHM as a palindromic subsequence.
Exercise

Definition
A string is a palindrome if $w = w^R$.
Examples: I, RACECAR, MALAYALAM, DOOFFOOD

Problem: Given a string $w$ find the longest subsequence of $w$ that is a palindrome.

Example
MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM has MHYMRORMYHM as a palindromic subsequence
Assume \( w \) is stored in an array \( A[1..n] \)

\( \text{LPS}(A[1..n]): \) length of longest palindromic subsequence of \( A \).

Recursive expression/code?
**Definition**

Edit distance between two words $X$ and $Y$ is the number of letter insertions, letter deletions and letter substitutions required to obtain $Y$ from $X$.

**Example**

The edit distance between FOOD and MONEY is at most 4:

$\text{FOOD} \rightarrow \text{MOOD} \rightarrow \text{MONOD} \rightarrow \text{MONED} \rightarrow \text{MONEY}$
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

FOOD
MONEY

Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

\[
\begin{array}{cccc}
F & O & O & D \\
M & O & N & E \\
E & Y
\end{array}
\]

Formally, an alignment is a set \( M \) of pairs \((i, j)\) such that each index appears at most once, and there is no “crossing”: \( i < i' \) and \( i \) is matched to \( j \) implies \( i' \) is matched to \( j' > j \). In the above example, this is \( M = \{(1, 1), (2, 2), (3, 3), (4, 5)\} \).
Alignment

Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

\[
\begin{array}{cccccc}
F & O & O & D \\
M & O & N & E & Y \\
\end{array}
\]

Formally, an alignment is a set \( M \) of pairs \((i, j)\) such that each index appears at most once, and there is no “crossing”: \( i < i' \) and \( i \) is matched to \( j \) implies \( i' \) is matched to \( j' > j \). In the above example, this is \( M = \{(1, 1), (2, 2), (3, 3), (4, 5)\} \). Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.
Edit Distance Problem

Problem

Given two words, find the edit distance between them, i.e., an alignment of smallest cost.
Edit Distance

Basic observation

Let $X = \alpha x$ and $Y = \beta y$

$\alpha, \beta$: strings. $x$ and $y$ single characters.

Possible alignments between $X$ and $Y$

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta$</td>
<td>$y$</td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta y$</td>
<td></td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th>$\alpha x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\beta$</td>
</tr>
<tr>
<td>$y$</td>
</tr>
</tbody>
</table>

Observation

Prefixes must have optimal alignment!
## Edit Distance

### Basic observation

Let $X = \alpha x$ and $Y = \beta y$

$\alpha, \beta$: strings. $x$ and $y$ single characters.

Possible alignments between $X$ and $Y$

<table>
<thead>
<tr>
<th></th>
<th>$X$</th>
<th>$Y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha$</td>
<td>$x$</td>
<td>$eta$</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\alpha$</td>
<td>$x$</td>
<td>$eta y$</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\alpha x$</td>
<td>$y$</td>
<td></td>
</tr>
</tbody>
</table>

### Observation

*Prefixes must have optimal alignment!*

$$EDIST(X, Y) = \min \begin{cases} 
EDIST(\alpha, \beta) + [x = y] \\
1 + EDIST(\alpha, Y) \\
1 + EDIST(X, \beta)
\end{cases}$$
Recursive Algorithm

Assume $X$ is stored in array $A[1..m]$ and $Y$ is stored in $B[1..n]$

\[
EDIST(A[1..m], B[1..n])
\]

If ($m = 0$) return $n$
If ($n = 0$) return $m$
$m_1 = 1 + EDIST(A[1..(m − 1)], B[1..n])$
$m_2 = 1 + EDIST(A[1..m], B[1..(n − 1)])$
If ($A[m] = B[n]$) then
\[
m_3 = EDIST(A[1..(m − 1)], B[1..(n − 1)])
\]
Else
\[
m_3 = 1 + EDIST(A[1..(m − 1)], B[1..(n − 1)])
\]
return $\min(m_1, m_2, m_3)$
DEED and DREAD
Subproblems and Recurrence

Each subproblem corresponds to a prefix of $X$ and a prefix of $Y$

Optimal Costs

Let $\text{Opt}(i, j)$ be optimal cost of aligning $x_1 \cdots x_i$ and $y_1 \cdots y_j$. Then

$$\text{Opt}(i, j) = \min \begin{cases} [x_i = y_j] + \text{Opt}(i - 1, j - 1), \\ 1 + \text{Opt}(i - 1, j), \\ 1 + \text{Opt}(i, j - 1) \end{cases}$$

Base Cases: $\text{Opt}(i, 0) = i$ and $\text{Opt}(0, j) = j$
Memoizing the Recursive Algorithm

int M[0..m][0..n]
Initialize all entries of M[i][j] to \( \infty \)
return \( EDIST(A[1..m], B[1..n]) \)

\[ EDIST(A[1..m], B[1..n]) \]
If \( (M[i][j] < \infty) \) return \( M[i][j] \)  (* return stored value *)
If \( (m = 0) \)
    \( M[i][j] = n \)
ElseIf \( (n = 0) \)
    \( M[i][j] = m \)
Else
    \( m_1 = 1 + EDIST(A[1..(m - 1)], B[1..n]) \)
    \( m_2 = 1 + EDIST(A[1..m], B[1..(n - 1)]) \)
If \( (A[m] = B[n]) \) \( m_3 = EDIST(A[1..(m - 1)], B[1..(n - 1)]) \)
Else \( m_3 = 1 + EDIST(A[1..(m - 1)], B[1..(n - 1)]) \)
\( M[i][j] = \min(m_1, m_2, m_3) \)
return \( M[i][j] \)
Removing Recursion to obtain Iterative Algorithm

\begin{align*}
EDIST(A[1..m], B[1..n])
\int M[0..m][0..n]
\text{for } i = 1 \text{ to } m \text{ do } M[i, 0] = i
\text{for } j = 1 \text{ to } n \text{ do } M[0, j] = j
\text{for } i = 1 \text{ to } m \text{ do }
\text{for } j = 1 \text{ to } n \text{ do }
\begin{cases}
[x_i = y_j] + M[i - 1][j - 1], \\
1 + M[i - 1][j], \\
1 + M[i][j - 1]
\end{cases}
M[i][j] = \min
\end{align*}
Removing Recursion to obtain Iterative Algorithm

\[ EDIST(A[1..m], B[1..n]) \]
\[ \text{int } M[0..m][0..n] \]
\[ \text{for } i = 1 \text{ to } m \text{ do } M[i, 0] = i \]
\[ \text{for } j = 1 \text{ to } n \text{ do } M[0, j] = j \]
\[ \text{for } i = 1 \text{ to } m \text{ do } \]
\[ \text{for } j = 1 \text{ to } n \text{ do } \]
\[ M[i][j] = \min \begin{cases} [x_i = y_j] + M[i - 1][j - 1], \\ 1 + M[i - 1][j], \\ 1 + M[i][j - 1] \end{cases} \]

Analysis

Running time is \( O(mn) \).

Chandra & Ruta (UIUC) CS473 26 Fall 2016
Removing Recursion to obtain Iterative Algorithm

EDIST(A[1..m], B[1..n])

int M[0..m][0..n]

for i = 1 to m do $M[i, 0] = i$

for j = 1 to n do $M[0, j] = j$

for i = 1 to m do
  for j = 1 to n do
    $M[i][j] = \min\left\{ [x_i = y_j] + M[i - 1][j - 1], 1 + M[i - 1][j], 1 + M[i][j - 1] \right\}$

Analysis

1. Running time is $O(mn)$.
2. Space used is $O(mn)$. 

Chandra & Ruta (UIUC) CS473 Fall 2016
Figure: Iterative algorithm in previous slide computes values in row order.
The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

**Question:** Can we find a specific alignment which achieves the minimum?
Finding an Optimum Solution

The DP algorithm finds the minimum edit distance in $O(nm)$ space and time.

**Question:** Can we find a specific alignment which achieves the minimum?

**Exercise:** Show that one can find an optimum solution after computing the optimum value. Key idea is to store back pointers when computing $\text{Opt}(i,j)$ to know how we calculated it. See notes for more details.
Longest Palindromic Subsequence

Definition

A sequence is a *palindrome* if the sequence is equal to its reverse. Examples: m,a,l,a,y,a,l,a,m and 1,10,10,1 and a.
Longest Palindromic Subsequence

Definition

A sequence is a palindrome if the sequence is equal to its reverse.
Examples: \(m, a, l, a, y, a, l, a, m\) and \(1, 10, 10, 1\) and \(a\).

Problem: Given a sequence \(a_0, a_1, \ldots, a_n\) find the longest palindromic sub-sequence.

Examples:
- \(1, 10, 11\)
- \(a, c, c, r, a\)
Dynamic Programming Template

1. Come up with a recursive algorithm to solve problem
2. Understand the structure/number of the subproblems generated by recursion
3. Memoize the recursion
   - set up compact notation for subproblems
   - set up a data structure for storing subproblems
4. Iterative algorithm
   - Understand dependency graph on subproblems
   - Pick an evaluation order (any topological sort of the dependency dag)
5. Analyze time and space
6. Optimize