Dynamic Programming

Lecture 13

Thursday, February 28, 2019

LaTeXed: February 28, 2019 16:32

Part I

Dynamic programming

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question: Suppose we have a recursive program $foo(x)$ that takes an input x.

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

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question: Suppose we have a recursive program $foo(x)$ that takes an input x.

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

Suppose we*memoize* the recursion.

Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question: Suppose we have a recursive program $foo(x)$ that takes an input x.

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

Suppose we*memoize* the recursion.
Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.
Q: What is an upper bound on the running time of *memoized* version of $foo(x)$ if $|x| = n$?

Dynamic Programming

Dynamic Programming is smart recursion plus memoization

Question: Suppose we have a recursive program $foo(x)$ that takes an input x.

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

Suppose we*memoize* the recursion.
Assumption: Storing and retrieving solutions to pre-computed problems takes $O(1)$ time.
Q: What is an upper bound on the running time of *memoized* version of $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

Part II

Checking if a string is in L^*

Problem

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

Goal Decide if $w \in L^*$ using **IsStrInL**$(string\ \ x)$ as a black box sub-routine

Problem

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

Goal Decide if $w \in L^*$ using **IsStrInL**(*string* x) as a black box sub-routine

Problem

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

$$*$$

Goal Decide if $w \in L$ using **IsStrInL**(*string* x) as a black box sub-routine

Problem

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

$*$

Goal Decide if $w \in L$ using **IsStrInL**(*string* x) as a black box sub-routine

Problem

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

Problem

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

Goal Decide if $w \in L^*$ using **IsStrInL**$(string\ \ x)$ as a black box sub-routine

Problem

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

Goal Decide if $w \in L^*$ using **IsStrInL**$(string\ \ x)$ 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 $w \in L^*$?

Recursive Solution

When is $w \in L^*$?

$w \in L^*$ if $w = \epsilon$ or $w \in L$ or if $w = uv$ where $u \in L$ and $v \in L^*$, $|u| \geq 1$

Recursive Solution

When is $w \in L^*$?

$w \in L^*$ if $w = \epsilon$ or $w \in L$ or if $w = uv$ where $u \in L$ and $v \in L^*$, $|u| \geq 1$

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

```
IsStringinLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n - 1) do
            If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                Output YES

    Output NO
```

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

```
IsStringinLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
         Output YES
    Else
         For (i = 1 to n − 1) do
              If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                   Output YES

    Output NO
```

times call IsStrInL

$$T(n) = \sum_{i=1}^{n-1} T(n-i) + n$$

$$= O(2^n)$$

Recursive Solution

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

```
IsStringinLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n − 1) do
            If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                Output YES

    Output NO
```

Question: How many distinct sub-problems does IsStrInLstar($A[1..n]$) generate?

Recursive Solution

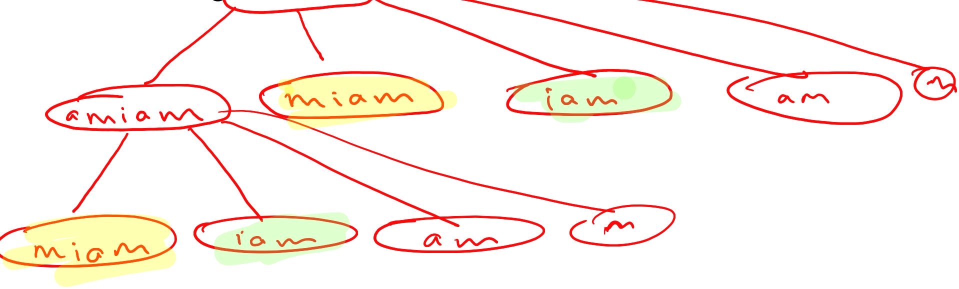
Assume w is stored in array $A[1..n]$

```
IsStringinLstar(A[1..n]):
    If (n = 0) Output YES
    If (IsStrInL(A[1..n]))
        Output YES
    Else
        For (i = 1 to n − 1) do
            If (IsStrInL(A[1..i]) and IsStrInLstar(A[i + 1..n]))
                Output YES

    Output NO
```

Question: How many distinct sub-problems does IsStrInLstar($A[1..n]$) generate? $O(n)$

Example

Consider string ***samiam***



Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

ISL(i): a boolean which is 1 if $A[i..n]$ is in L^*, 0 otherwise

Base case: **ISL**$(n+1) = 1$ interpreting $A[n+1..n]$ as ϵ

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

ISL(i): a boolean which is 1 if $A[i..n]$ is in L^*, 0 otherwise

Base case: $\text{ISL}(n+1) = 1$ interpreting $A[n+1..n]$ as ϵ

Recursive relation:

$A[i...n] \in L^*$ suffix prefix

$A[j...n] \in L^*$ $A[i...(j-1)] \in L$

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

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n)$ we name them to help us understand the structure better.

$\mathrm{ISL}(i)$: a boolean which is 1 if $A[i..n]$ is in L^*, 0 otherwise

Base case: $\mathrm{ISL}(n+1) = 1$ interpreting $A[n+1..n]$ as ϵ
Recursive relation:

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

Output: $\mathrm{ISL}(1)$

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 *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.

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 recursion.

Iterative Algorithm

```
IsStringinLstar-Iterative(A[1..n]):
    boolean ISL[1..(n + 1)]
    ISL[n + 1] = TRUE
    for (i = n down to 1)
        ISL[i] = FALSE
        for (j = i + 1 to n + 1)                n  n
                If (ISL[j] and IsStrInL(A[i..j − 1]))
                    ISL[i] = TRUE
                    Break

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

Iterative Algorithm

```
IsStringinLstar-Iterative(A[1..n]):
    boolean ISL[1..(n+1)]
    ISL[n+1] = TRUE
    for (i = n down to 1)
        ISL[i] = FALSE
        for (j = i+1 to n+1)
                If (ISL[j] and IsStrInL(A[i..j-1]))
                    ISL[i] = TRUE
                    Break

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

- **Running time:**

Iterative Algorithm

```
IsStringinLstar-Iterative(A[1..n]):
    boolean ISL[1..(n + 1)]
    ISL[n + 1] = TRUE
    for (i = n down to 1)
        ISL[i] = FALSE
        for (j = i + 1 to n + 1)
            If (ISL[j] and IsStrInL(A[i..j − 1]))
                ISL[i] = TRUE
                Break

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

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

Iterative Algorithm

```
IsStringinLstar-Iterative(A[1..n]):
    boolean ISL[1..(n+1)]
    ISL[n+1] = TRUE
    for (i = n down to 1)
        ISL[i] = FALSE
        for (j = i+1 to n+1)
              If (ISL[j] and IsStrInL(A[i..j-1]))
                 ISL[i] = TRUE
                 Break

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

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

Iterative Algorithm

```
IsStringinLstar-Iterative(A[1..n]):
    boolean ISL[1..(n + 1)]
    ISL[n + 1] = TRUE
    for (i = n down to 1)
        ISL[i] = FALSE
        for (j = i + 1 to n + 1)
                If (ISL[j] and IsStrInL(A[i..j − 1]))
                    ISL[i] = TRUE
                    Break

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

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

Example

1 2 3 4 5 6

Consider string ***samiam***

ISL[7] = True

ISL[6] = True ∧ IsStrInL("m") = False

ISL[5] = ISL[6] ∧ IsStrInL("a") = Fals
OR ISL[7] ∧ IsStrInL("am") = True
= Tru

I

Part III

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...

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

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

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

Recursive Approach: Take 1

LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

$\mathrm{LIS}(\boldsymbol{A}[1..\boldsymbol{n}])$:

1. Case 1: Does not contain $\boldsymbol{A}[\boldsymbol{n}]$ in which case $\mathrm{LIS}(\boldsymbol{A}[1..\boldsymbol{n}]) = \mathrm{LIS}(\boldsymbol{A}[1..(\boldsymbol{n}-1)])$
2. Case 2: contains $\boldsymbol{A}[\boldsymbol{n}]$ in which case $\mathrm{LIS}(\boldsymbol{A}[1..\boldsymbol{n}])$ is not so clear.

Observation

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

Recursive Approach

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

$\textbf{LIS\_smaller}(A[1..n], x)$: 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
```

don't include A[n]

include A[n]

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

Example

Sequence: $A[1..7] = 6, 3, 5, 2, 7, 8, 1$

Recursive Approach

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

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

- How many distinct sub-problems will LIS_smaller($A[1..n], \infty$) generate?

Recursive Approach

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

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

- How many distinct sub-problems will **LIS_smaller**$(A[1..n], \infty)$ generate? $O(n^2)$

Recursive Approach

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

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

- How many distinct sub-problems will **LIS_smaller**$(A[1..n], \infty)$ generate? $O(n^2)$
- What is the running time if we memoize recursion?

Recursive Approach

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

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

- How many distinct sub-problems will **LIS_smaller**$(A[1..n], \infty)$ 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

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

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

- How many distinct sub-problems will **LIS_smaller**$(A[1..n], \infty)$ 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.
- How much space for memoization?

Recursive Approach

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

```
LIS(A[1..n]):
    return LIS_smaller(A[1..n], ∞)
```

- How many distinct sub-problems will **LIS_smaller**$(A[1..n], \infty)$ 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.
- How much space for memoization? $O(n^2)$

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add ∞ at end of array (in position $n+1$)   A[n+1] = ∞

$\mathrm{LIS}(i,j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add ∞ at end of array (in position $n+1$)

$\mathrm{LIS}(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

Base case: $\mathrm{LIS}(0, j) = 0$ for $1 \leq j \leq n+1$

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add ∞ at end of array (in position $n + 1$)

$\mathrm{LIS}(i, j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

Base case: $\mathrm{LIS}(0, j) = 0$ for $1 \leq j \leq n + 1$
Recursive relation:

- $\mathrm{LIS}(i, j) = \mathrm{LIS}(i - 1, j)$ if $A[i] > A[j]$
- $\mathrm{LIS}(i, j) = \max\{LIS(i - 1, j), 1 + LIS(i - 1, i)\}$ if $A[i] \leq A[j]$

A[i] is not   A[i] is in LIS

Naming subproblems and recursive equation

After seeing that number of subproblems is $O(n^2)$ we name them to help us understand the structure better. For notational ease we add ∞ at end of array (in position $n+1$)

$\mathrm{LIS}(i,j)$: length of longest increasing sequence in $A[1..i]$ among numbers less than $A[j]$ (defined only for $i < j$)

Base case: $\mathrm{LIS}(0,j) = 0$ for $1 \le j \le n+1$
Recursive relation:

- $\mathrm{LIS}(i,j) = \mathrm{LIS}(i-1,j)$ if $A[i] > A[j]$
- $\mathrm{LIS}(i,j) = \max\{LIS(i-1,j), 1 + LIS(i-1,i)\}$ if $A[i] \le A[j]$

Output: $\mathrm{LIS}(n, n+1)$

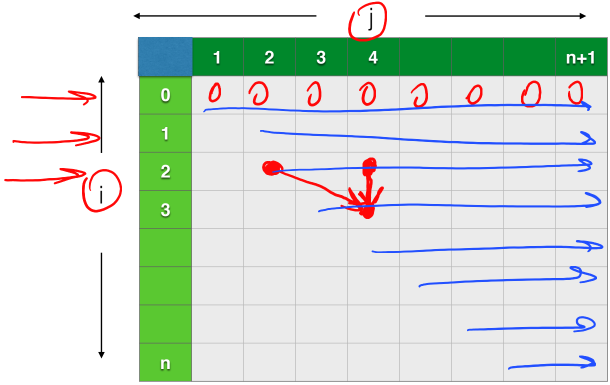
Iterative algorithm

```
LIS-Iterative(A[1..n]):
    A[n + 1] = ∞
    int LIS[0..n, 1..n + 1]
    for (j = 1 to n + 1) do
        LIS[0, j] = 0

    for (i = 1 to n) do
        for (j = i + 1 to n)
            If (A[i] > A[j]) LIS[i, j] = LIS[i − 1, j]
            Else LIS[i, j] = max{LIS[i − 1, j], 1 + LIS[i − 1, i]}

    Return LIS[n, n + 1]
```

Running time: $O(n^2)$

Space: $O(n^2)$

How to order bottom up computation?



Base case: $\mathrm{LIS}(0, j) = 0$ for $1 \leq j \leq n + 1$

Recursive relation:

- $\mathrm{LIS}(i, j) = \mathrm{LIS}(i - 1, j)$ if $A[i] > A[j]$
- $\mathrm{LIS}(i, j) = \max\{LIS(i - 1, j), 1 + LIS(i - 1, i)\}$ if $A[i] \leq A[j]$

How to order bottom up computation?

Sequence: $A[1..7] = 6, 3, 5, 2, 7, 8, 1, \infty$

j (columns) / i (rows)

	1	2	3	4				n+1
0	0	0	0	0	0	0	0	0
1	1	0	0	0	1	-	-	-
2		1						
3								
n								

Two comments

Question: Can we compute an optimum solution and not just its value?

Two comments

Question: Can we compute an optimum solution and not just its value?
Yes! See notes.

Question: Is there a faster algorithm for LIS?

Two comments

Question: Can we compute an optimum solution and not just its value?
Yes! See notes.

Question: Is there a faster algorithm for LIS? Yes! Using a different recursion and optimizing one can obtain an $O(n \log n)$ time and $O(n)$ space algorithm. $O(n \log n)$ time is not obvious. Depends on improving time by using data structures on top of dynamic programming.

Recursive Algorithm: Take 2

Definition

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

Question: can we obtain a recursive expression?

Recursive Algorithm: Take 2

Definition

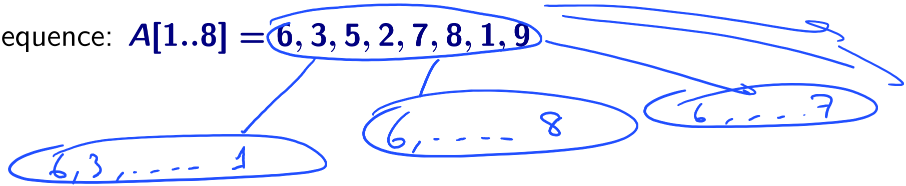
LISEnding($A[1..n]$): length of longest increasing sub-sequence that *ends* in $A[n]$.

Question: can we obtain a recursive expression?

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

Example

Sequence: $A[1..8] = 6, 3, 5, 2, 7, 8, 1, 9$



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])
```

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 **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 LIS_ending_alg($A[1..n]$) generate? $O(n)$

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 **LIS_ending_alg**$(A[1..n])$ generate? $O(n)$
- What is the running time if we memoize recursion?

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 **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

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 **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]):
    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 **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? $O(n)$

Iterative Algorithm via Memoization

Compute the values **LIS_ending_alg($A[1..i]$)** iteratively in a bottom up fashion.

```
LIS_ending_alg(A[1..n]):
    Array L[1..n]  (* L[i] = value of LIS_ending_alg(A[1..i]) *)
    for i = 1 to n do
        L[i] = 1
        for j = 1 to i − 1 do
            if (A[j] < A[i]) do
                L[i] = max(L[i], 1 + L[j])
    return L
```

```
LIS(A[1..n]):
        L = LIS_ending_alg(A[1..n])
        return the maximum value in L
```

Iterative Algorithm via Memoization

Simplifying:

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

Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

Running time:

Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

Running time: $O(n^2)$

Space:

Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

Running time: $O(n^2)$

Space: $\Theta(n)$

Iterative Algorithm via Memoization

Simplifying:

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

Correctness: Via induction following the recursion

Running time: $O(n^2)$

Space: $\Theta(n)$

$O(n \log n)$ run-time achievable via better data structures.

Example

Example

1. Sequence: 6, 3, 5, 2, 7, 8, 1
2. Longest increasing subsequence: 3, 5, 7, 8

Example

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 $L[n]$

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