

Dynamic Programming

Lecture 13

March 6, 2025

Dynamic Programming

Dynamic Programming is **smart recursion** plus **memoization**

```
foo(x):  
  do stuff  
  x = foo(x1)  
  do stuff  
  y = foo(x2)  
  Output blah
```

Recursive program $foo(x)$. On input x of size n

- number of *distinct* sub-problems $foo(x)$ generates is $\leq A(n)$
- $foo(x)$ spends $\leq B(n)$ time *not counting* recursive calls.

Dynamic Programming

Dynamic Programming is **smart recursion** plus **memoization**

```
foo(x):  
  do stuff  
  x = foo(x1)  
  do stuff  
  y = foo(x2)  
  Output blah
```

Recursive program $foo(x)$. On input x of size n

- number of *distinct* sub-problems $foo(x)$ generates is $\leq A(n)$
- $foo(x)$ spends $\leq B(n)$ time *not counting* recursive calls.

Suppose we *memoize* the recursion.

Question: What is an upper bound on the running time of *memoized* version of $foo(x)$ if $|x| = n$?

Dynamic Programming

Recursive program $foo(x)$. On input x of size n

- number of *distinct* sub-problems $foo(x)$ generates is $\leq A(n)$
- $foo(x)$ spends $\leq B(n)$ time *not counting* recursive calls.

Suppose we *memoize* the recursion.

Assumption: Storing & retrieving solutions to pre-computed problems takes $O(1)$ time. Recursion tree evaluated in preorder/DFS fashion

Question: What is an upper bound on the running time of *memoized* version of $foo(x)$ if $|x| = n$?

Dynamic Programming

Recursive program $foo(x)$. On input x of size n

- number of *distinct* sub-problems $foo(x)$ generates is $\leq A(n)$
- $foo(x)$ spends $\leq B(n)$ time *not counting* recursive calls.

Suppose we *memoize* the recursion.

Assumption: Storing & retrieving solutions to pre-computed problems takes $O(1)$ time. Recursion tree evaluated in preorder/DFS fashion

Question: What is an upper bound on the running time of *memoized* version of $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

Dynamic Programming

Recursive program $foo(x)$. On input x of size n

- number of *distinct* sub-problems $foo(x)$ generates is $\leq A(n)$
- $foo(x)$ spends $\leq B(n)$ time *not counting* recursive calls.

Suppose we *memoize* the recursion.

Assumption: Storing & retrieving solutions to pre-computed problems takes $O(1)$ time. Recursion tree evaluated in preorder/DFS fashion

Question: What is an upper bound on the running time of *memoized* version of $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

Without assumption one has to account for the time to store and retrieve from the data structure. It depends very much on the problem. This is one reason why we want to memoize explicitly.

Example: Fibonacci recurrence

Initialize a (dynamic) dictionary data structure D to empty

```
Fib( $n$ ):  
  if ( $n = 0$ )  
    return 0  
  if ( $n = 1$ )  
    return 1  
  if ( $n$  is already in  $D$ )  
    return value stored with  $n$  in  $D$   
     $val \leftarrow \mathbf{Fib}(n - 1) + \mathbf{Fib}(n - 2)$   
  Store ( $n, val$ ) in  $D$   
  return  $val$ 
```

$A(n) = ?$ and $B(n) = ?$

Part I

Checking if string is in Kleene star of a language

Problem

Input A string $w \in \Sigma^*$, and 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 $w \in L^*$? $w \in L^*$ iff

- $w = \varepsilon$ or
- $w \in L$ or
- $w = uv$ where $u \in L$ and $v \in L^*$ and $|u| \geq 1$

Recursive Solution

When is $w \in L^*$? $w \in L^*$ iff

- $w = \varepsilon$ or
- $w \in L$ or
- $w = uv$ where $u \in L$ and $v \in L^*$ and $|u| \geq 1$

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
```

Recursive Solution

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

```
IsStringInLstar( $A[1..n]$ ):
```

```
  If ( $n = 0$ ) Output YES
```

```
  If ( $\text{IsStrInL}(A[1..n])$ )
```

```
    Output YES
```

```
  Else
```

```
    For ( $i = 1$  to  $n - 1$ ) do
```

```
      If ( $\text{IsStrInL}(A[1..i])$  and  $\text{IsStrInLstar}(A[i + 1..n])$ )
```

```
        Output YES
```

```
  Output NO
```

Recursive Solution

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

```
IsStringInLstar( $A[1..n]$ ):  
  If ( $n = 0$ ) Output YES  
  If ( $\text{IsStrInL}(A[1..n])$ )  
    Output YES  
  Else  
    For ( $i = 1$  to  $n - 1$ ) do  
      If ( $\text{IsStrInL}(A[1..i])$  and  $\text{IsStrInLstar}(A[i + 1..n])$ )  
        Output YES  
  
  Output NO
```

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

Recursive Solution

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

```
IsStringInLstar( $A[1..n]$ ):
  If ( $n = 0$ ) Output YES
  If ( $\text{IsStrInL}(A[1..n])$ )
    Output YES
  Else
    For ( $i = 1$  to  $n - 1$ ) do
      If ( $\text{IsStrInL}(A[1..i])$  and  $\text{IsStrInLstar}(A[i + 1..n])$ )
        Output YES

  Output NO
```

Question: How many *distinct* sub-problems does $\text{IsStrInLstar}(A[1..n])$ generate? $O(n)$. Why?

Recursive Solution

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

```
IsStringInLstar( $A[1..n]$ ):  
  If ( $n = 0$ ) Output YES  
  If ( $\text{IsStrInL}(A[1..n])$ )  
    Output YES  
  Else  
    For ( $i = 1$  to  $n - 1$ ) do  
      If ( $\text{IsStrInL}(A[1..i])$  and  $\text{IsStrInLstar}(A[i + 1..n])$ )  
        Output YES  
  
  Output NO
```

Question: How many *distinct* sub-problems does

$\text{IsStrInLstar}(A[1..n])$ generate? $O(n)$. Why?

Each sub-problem corresponds to a *suffix* of the input string w

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.

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

Base case: **IsStrInLstar($n + 1$) = 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.

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

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

Recursive relation:

- **IsStrInLstar(i) = 1** if $\exists i < j \leq n + 1$ such that
(**IsStrInLstar(j) = 1** and **IsStrInL($A[i..(j - 1)]$) = 1**)
- **IsStrInLstar(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.

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

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

Recursive relation:

- **IsStrInLstar(i) = 1** if $\exists i < j \leq n + 1$ such that
(**IsStrInLstar(j) = 1** and **IsStrInL($A[i..(j - 1)]$) = 1**)
- **IsStrInLstar(i) = 0** otherwise

Output: **IsStrInLstar(1)**

Naming subproblems and recursive equation

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

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

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

Recursive relation:

- **IsStrInLstar(i) = 1** if $\exists i < j \leq n + 1$ such that
(**IsStrInLstar(j) = 1** and **IsStrInL($A[i..(j - 1)]$) = 1**)
- **IsStrInLstar(i) = 0** otherwise

Output: **IsStrInLstar(1)**

Naming the problems allows us to easily see how many sub-problems are there. Typically we use indices while original recursive algorithm may have more complex structures (such as arrays, trees, etc)

Removing recursion: 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: 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: 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: 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 IsStrInLstar[ $1..(n + 1)$ ]  
  IsStrInLstar[ $n + 1$ ] = TRUE  
  for ( $i = n$  down to 1)  
    IsStrInLstar[ $i$ ] = FALSE  
    for ( $j = i + 1$  to  $n + 1$ )  
      If (IsStrInLstar[ $j$ ] and IsStrInL( $A[i..j - 1]$ ))  
        IsStrInLstar[ $i$ ] = TRUE  
        Break  
  
  If (IsStrInLstar[1] = 1) Output YES  
  Else Output NO
```

Iterative Algorithm

```
IsStringInLstar-Iterative( $A[1..n]$ ):  
  boolean IsStrInLstar[ $1..(n + 1)$ ]  
  IsStrInLstar[ $n + 1$ ] = TRUE  
  for ( $i = n$  down to 1)  
    IsStrInLstar[ $i$ ] = FALSE  
    for ( $j = i + 1$  to  $n + 1$ )  
      If (IsStrInLstar[ $j$ ] and IsStrInL( $A[i..j - 1]$ ))  
        IsStrInLstar[ $i$ ] = TRUE  
        Break  
  
  If (IsStrInLstar[1] = 1) Output YES  
  Else Output NO
```

- Running time:

Iterative Algorithm

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

  If (IsStrInLstar[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 IsStrInLstar[ $1..(n + 1)$ ]  
  IsStrInLstar[ $n + 1$ ] = TRUE  
  for ( $i = n$  down to 1)  
    IsStrInLstar[ $i$ ] = FALSE  
    for ( $j = i + 1$  to  $n + 1$ )  
      If (IsStrInLstar[ $j$ ] and IsStrInL( $A[i..j - 1]$ ))  
        IsStrInLstar[ $i$ ] = TRUE  
        Break  
  
  If (IsStrInLstar[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 IsStrInLstar[1..( $n + 1$ )]
  IsStrInLstar[ $n + 1$ ] = TRUE
  for ( $i = n$  down to 1)
    IsStrInLstar[ $i$ ] = FALSE
    for ( $j = i + 1$  to  $n + 1$ )
      If (IsStrInLstar[ $j$ ] and IsStrInL( $A[i..j - 1]$ ))
        IsStrInLstar[ $i$ ] = TRUE
        Break

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

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

Example

Consider string *samiam*

1	2	3	4	5	6	7

Part II

Longest Increasing Subsequence

Sequences

Definition

Sequence: an ordered list a_1, a_2, \dots, a_n . **Length** of a sequence is number of elements in the list.

Definition

a_{i_1}, \dots, a_{i_k} is a **subsequence** of a_1, \dots, a_n if
 $1 \leq i_1 < i_2 < \dots < i_k \leq n$.

Definition

A sequence is **increasing** if $a_1 < a_2 < \dots < a_n$. It is **non-decreasing** if $a_1 \leq a_2 \leq \dots \leq a_n$. 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, \dots, a_n

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

Longest Increasing Subsequence Problem

Input A sequence of numbers a_1, a_2, \dots, a_n

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

Example

- 1 Sequence: 6, 3, 5, 2, 7, 8, 4
- 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?

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  $LIS\_smaller(A[1..n], \infty)$ 
```

Example

Sequence: $A[1..7] = 6, 3, 5, 2, 7, 8, 4$

Recursive Approach

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

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

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

Recursive Approach

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

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

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

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

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

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

Recursive Approach

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

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

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

```
LIS( $A[1..n]$ ):  
  return 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? $O(n^2)$

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 $n + 1$)

$LIS(i, j)$: 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 $n + 1$)

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

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

Recursive relation:

- $LIS(i, j) = LIS(i - 1, j)$ if $A[i] \geq A[j]$
- $LIS(i, j) = \max\{LIS(i - 1, j), 1 + LIS(i - 1, i)\}$ if $A[i] < A[j]$

Output: $LIS(n, n + 1)$

Iterative algorithm

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

$A[n + 1] = \infty$

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 ($A[i] > A[j]$) $LIS[i, j] = LIS[i - 1, j]$

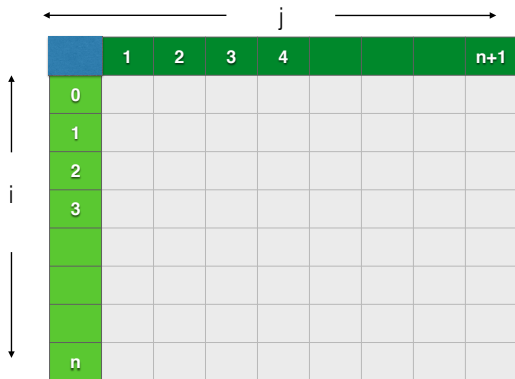
Else $LIS[i, j] = \max\{LIS[i - 1, j], 1 + LIS[i - 1, i]\}$

Return $LIS[n, n + 1]$

Running time: $O(n^2)$

Space: $O(n^2)$

How to order bottom up computation?



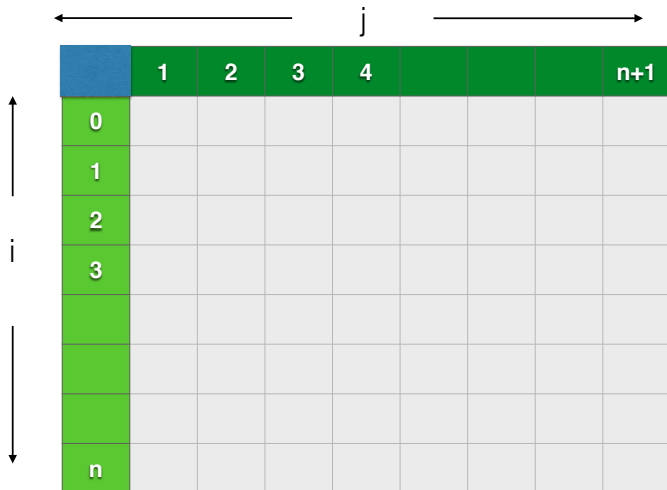
Base case: $LIS(0, j) = 0$ for $1 \leq j \leq n + 1$

Recursive relation:

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

How to order bottom up computation?

Sequence: $A[1..7] = 6, 3, 5, 2, 7, 8, 4$



Two comments

Question: compute an optimum solution in addition to value?

Two comments

Question: compute an optimum solution in addition to value?

Yes! See notes.

Question: Is there a faster algorithm for LIS?

Two comments

Question: compute an optimum solution in addition to 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?

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

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  
       $m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i]))$   
  return  $m$ 
```

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

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  
       $m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i]))$   
  return  $m$ 
```

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

- How many distinct sub-problems will $\text{LIS_ending_alg}(A[1..n])$ 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  
       $m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i]))$   
  return  $m$ 
```

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

- How many distinct sub-problems will $\text{LIS_ending_alg}(A[1..n])$ 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  
       $m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i]))$   
  return  $m$ 
```

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

- 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?

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  
       $m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i]))$   
  return  $m$ 
```

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

- 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

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  
       $m = \max(m, 1 + \text{LIS\_ending\_alg}(A[1..i]))$   
  return  $m$ 
```

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

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

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

- 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? $O(n)$

Iterative Algorithm via Memoization

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

```
LIS_ending_alg( $A[1..n]$ ):  
  Array  $L[1..n]$  (*  $L[i]$  = value of  $\text{LIS\_ending\_alg}(A[1..i])$  *)  
  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( $A[1..n]$ ):  
   $L = \text{LIS\_ending\_alg}(A[1..n])$   
  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, 4
- 2 Longest increasing subsequence: 3, 5, 7, 8

Example

Example

- 1 Sequence: 6, 3, 5, 2, 7, 8, 4
- 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