

Pre-lecture brain teaser

Write a (very simple) recursive algorithm that calculates the Fibonacci n^{th} number.

$$F_n = F_{n-1} + F_{n-2} \text{ where } F_0 = 0, F_1 = 1$$

ECE-374-B: Lecture 12 - Dynamic Programming I

Instructor: Nickvash Kani

October 09, 2025

University of Illinois Urbana-Champaign

Pre-lecture brain teaser

Write a (very simple) recursive algorithm that calculates the Fibonacci n^{th} number.

$$F_n = F_{n-1} + F_{n-2} \text{ where } F_0 = 0, F_1 = 1$$

Recursion and Memoization

Fibonacci Numbers

Fibonacci numbers defined by recurrence:

$$F(n) = F(n - 1) + F(n - 2) \text{ and } F(0) = 0, F(1) = 1.$$

These numbers have many interesting properties. A journal The Fibonacci Quarterly¹!

Fibonacci Numbers

Fibonacci numbers defined by recurrence:

$$F(n) = F(n - 1) + F(n - 2) \text{ and } F(0) = 0, F(1) = 1.$$

These numbers have many interesting properties. A journal The Fibonacci Quarterly¹!

- Binet's formula: $F(n) = \frac{\varphi^n - (1 - \varphi)^n}{\sqrt{5}} \approx \frac{1.618^n - (-0.618)^n}{\sqrt{5}} \approx \frac{1.618^n}{\sqrt{5}}$
 φ is the golden ratio $(1 + \sqrt{5})/2 \simeq 1.618$.
- $\lim_{n \rightarrow \infty} F(n + 1)/F(n) = \varphi$

Recursive Algorithm for Fibonacci Numbers

Question: Given n , compute $F(n)$.

```
Fib( $n$ ):  
    if ( $n = 0$ )  
        return 0  
    else if ( $n = 1$ )  
        return 1  
    else  
        return Fib( $n - 1$ ) + Fib( $n - 2$ )
```


Recursive Algorithm for Fibonacci Numbers

Question: Given n , compute $F(n)$.

```
Fib( $n$ ):  
    if ( $n = 0$ )  
        return 0  
    else if ( $n = 1$ )  
        return 1  
    else  
        return Fib( $n - 1$ ) + Fib( $n - 2$ )
```

Running time? Let $T(n)$ be the number of additions in Fib(n).

Recursive Algorithm for Fibonacci Numbers

Question: Given n , compute $F(n)$.

```
Fib( $n$ ):  
    if ( $n = 0$ )  
        return 0  
    else if ( $n = 1$ )  
        return 1  
    else  
        return Fib( $n - 1$ ) + Fib( $n - 2$ )
```

Running time? Let $T(n)$ be the number of additions in $\text{Fib}(n)$.

$$T(n) = T(n - 1) + T(n - 2) + 1 \text{ and } T(0) = T(1) = 0$$

Recursive Algorithm for Fibonacci Numbers

Question: Given n , compute $F(n)$.

```
Fib( $n$ ):  
    if ( $n = 0$ )  
        return 0  
    else if ( $n = 1$ )  
        return 1  
    else  
        return Fib( $n - 1$ ) + Fib( $n - 2$ )
```

Running time? Let $T(n)$ be the number of additions in $\text{Fib}(n)$.

$$T(n) = T(n - 1) + T(n - 2) + 1 \text{ and } T(0) = T(1) = 0$$

Roughly same as $F(n)$: $T(n) = \Theta(\varphi^n)$.

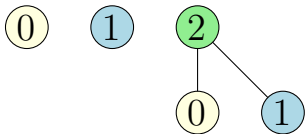
The number of additions is exponential in n . Can we do better?

Recursion tree for the Recursive Fibonacci

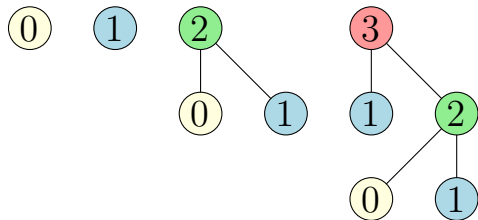
0

1

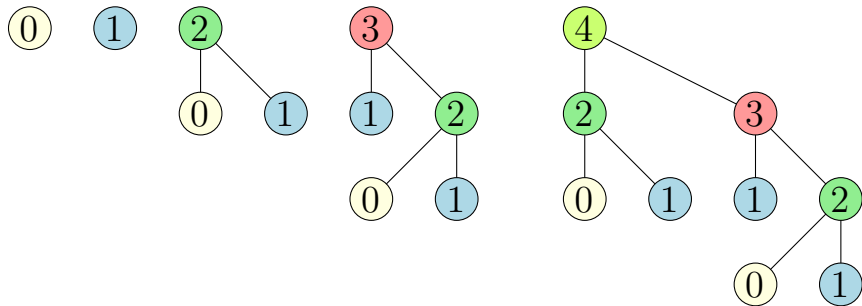
Recursion tree for the Recursive Fibonacci



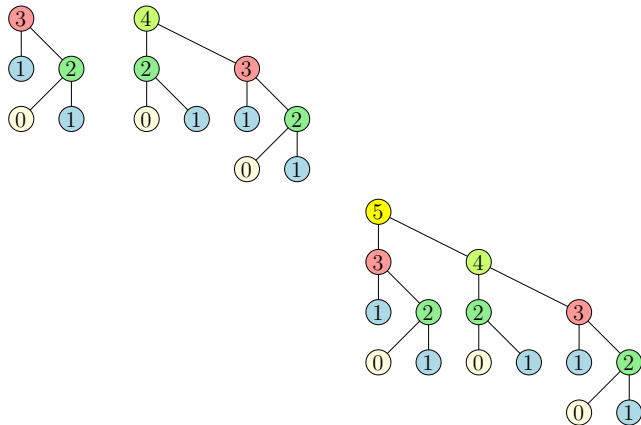
Recursion tree for the Recursive Fibonacci



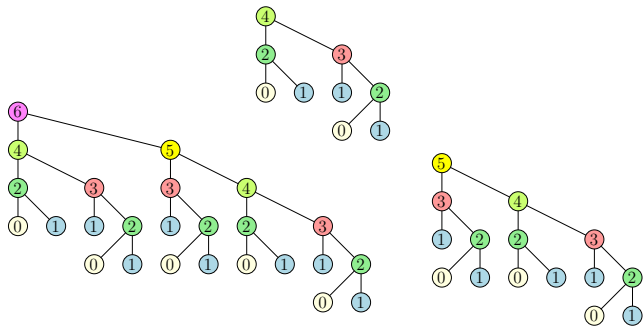
Recursion tree for the Recursive Fibonacci



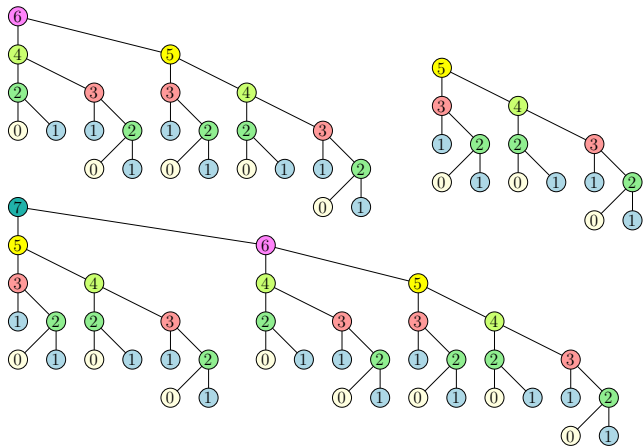
Recursion tree for the Recursive Fibonacci



Recursion tree for the Recursive Fibonacci



Recursion tree for the Recursive Fibonacci



An iterative algorithm for Fibonacci numbers

```
FibIter( $n$ ):  
    if ( $n = 0$ ) then  
        return 0  
    if ( $n = 1$ ) then  
        return 1  
     $F[0] = 0$   
     $F[1] = 1$   
    for  $i = 2$  to  $n$  do  
         $F[i] = F[i - 1] + F[i - 2]$   
    return  $F[n]$ 
```

An iterative algorithm for Fibonacci numbers

```
FibIter( $n$ ):  
    if ( $n = 0$ ) then  
        return 0  
    if ( $n = 1$ ) then  
        return 1  
     $F[0] = 0$   
     $F[1] = 1$   
    for  $i = 2$  to  $n$  do  
         $F[i] = F[i - 1] + F[i - 2]$   
    return  $F[n]$ 
```

What is the running time of the algorithm?

An iterative algorithm for Fibonacci numbers

```
FibIter( $n$ ):  
    if ( $n = 0$ ) then  
        return 0  
    if ( $n = 1$ ) then  
        return 1  
     $F[0] = 0$   
     $F[1] = 1$   
    for  $i = 2$  to  $n$  do  
         $F[i] = F[i - 1] + F[i - 2]$   
    return  $F[n]$ 
```

What is the running time of the algorithm? $O(n)$ additions.

What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value.

What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. **Memoization**.

What is the difference?

- Recursive algorithm is computing the same numbers again and again.
- Iterative algorithm is storing computed values and building bottom up the final value. **Memoization**.

Dynamic Programming: Finding a recursion that can be effectively/efficiently memorized.

Leads to polynomial time algorithm if number of sub-problems is polynomial in input size.

Automatic/implicit memoization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

Automatic Memorization

Can we convert recursive algorithm into an efficient algorithm without explicitly doing an iterative algorithm?

```
Fib( $n$ ):  
    if ( $n = 0$ )  
        return 0  
    if ( $n = 1$ )  
        return 1  
    if (Fib( $n$ ) was previously computed)  
        return stored value of Fib( $n$ )  
    else  
        return Fib( $n - 1$ ) + Fib( $n - 2$ )
```

How do we keep track of previously computed values?

Two methods: explicitly and implicitly (via data structure)

Automatic implicit memoization

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  $val$ 
```

Use hash-table or a map to remember which values were already computed.

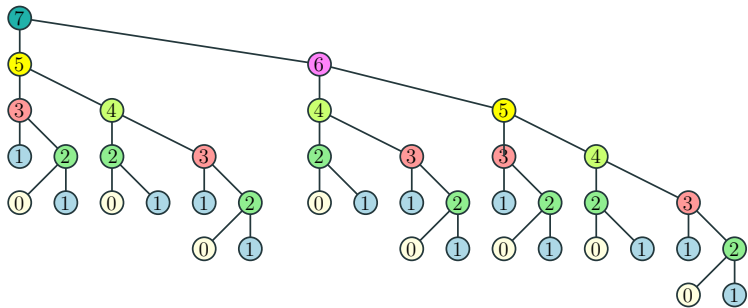
Explicit memoization (not automatic)

- Initialize table/array M of size n : $M[i] = -1$ for $i = 0, \dots, n$.
- Resulting code:

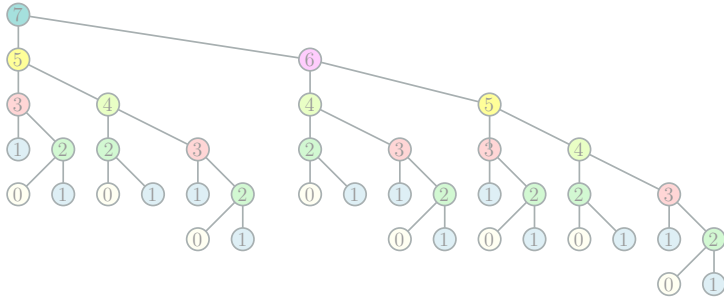
```
Fib( $n$ ):  
    if ( $n = 0$ )  
        return 0  
    if ( $n = 1$ )  
        return 1  
    if ( $M[n] \neq -1$ ) //  $M[n]$ : stored value of Fib( $n$ )  
        return  $M[n]$   
     $M[n] \leftarrow$  Fib( $n - 1$ ) + Fib( $n - 2$ )  
    return  $M[n]$ 
```

- Need to know upfront the number of sub-problems to allocate memory.

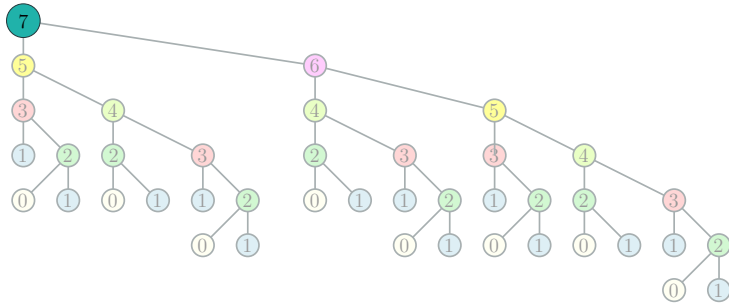
Recursion tree for the memoized Fib...



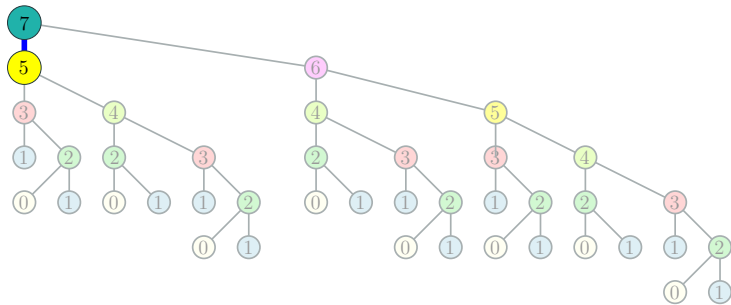
Recursion tree for the memoized Fib...



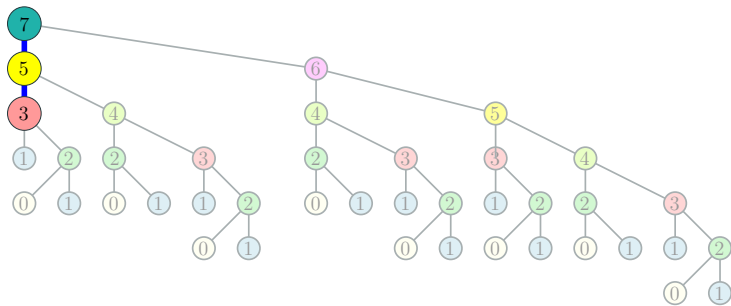
Recursion tree for the memoized Fib...

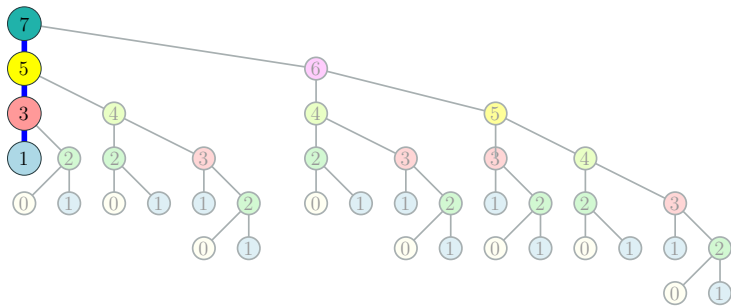


Recursion tree for the memoized Fib...

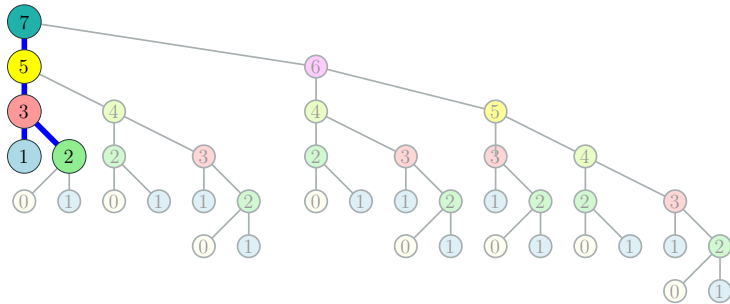


Recursion tree for the memoized Fib...

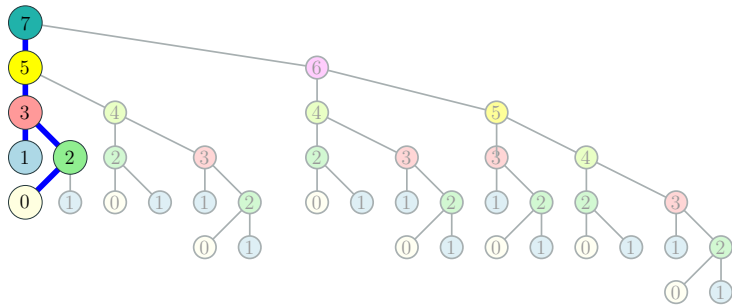




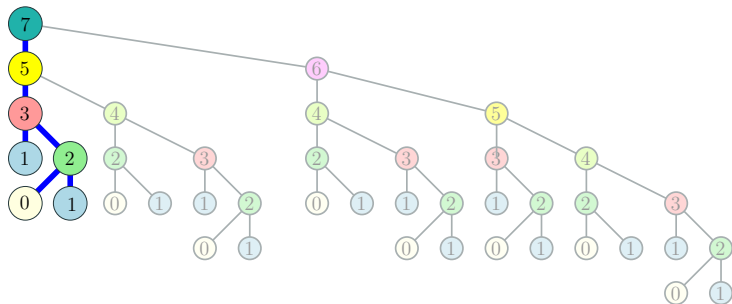
Recursion tree for the memoized Fib...



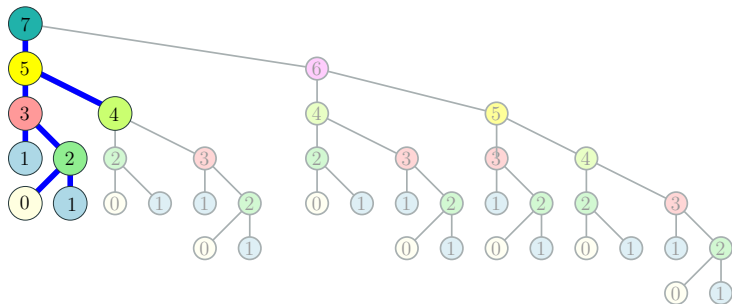
Recursion tree for the memoized Fib...



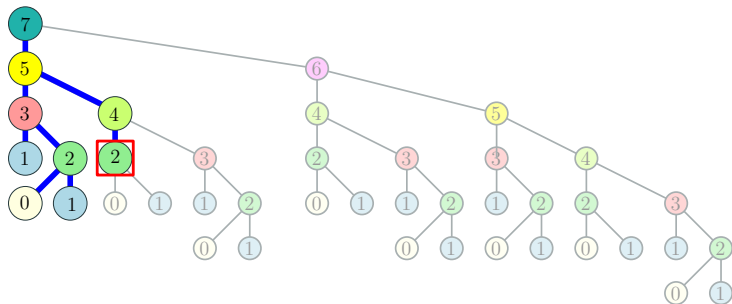
Recursion tree for the memoized Fib...



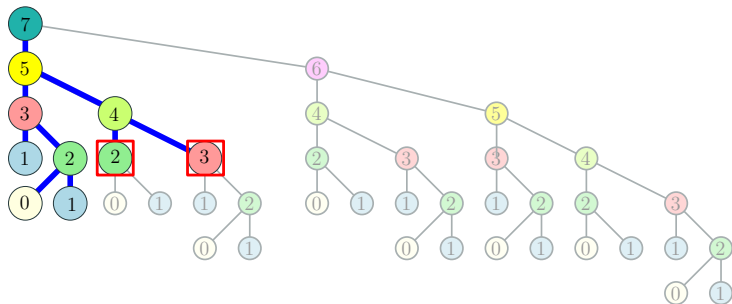
Recursion tree for the memoized Fib...



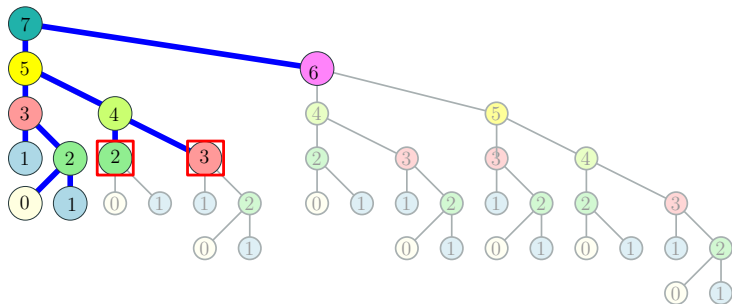
Recursion tree for the memoized Fib...



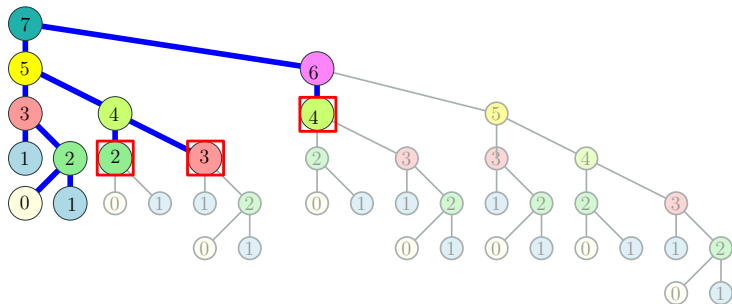
Recursion tree for the memoized Fib...



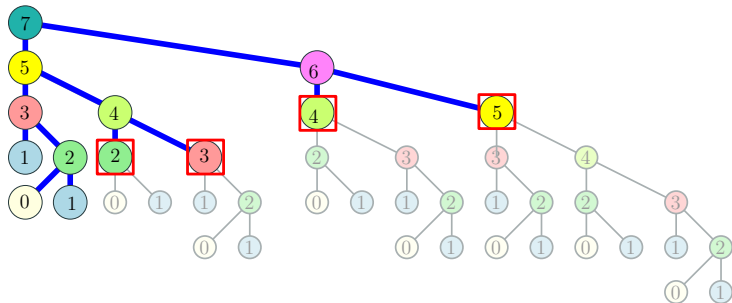
Recursion tree for the memoized Fib...



Recursion tree for the memoized Fib...



Recursion tree for the memoized Fib...



Automatic (Implicit) Memorization

- Recursive version:

```
f(x1, x2, ..., xd):  
    CODE
```

- Recursive version with memoization:

```
g(x1, x2, ..., xd):  
    if f already computed for (x1, x2, ..., xd) then  
        return value already computed  
    NEW_CODE
```

- NEW_CODE:
 - Replaces any “**return** α ” with
 - Remember “ $f(x_1, \dots, x_d) = \alpha$ ”; **return** α .

Explicit vs Implicit Memoization

- Explicit memoization (on the way to iterative algorithm) preferred:
 - analyze problem ahead of time
 - Allows for efficient memory allocation and access.
- Implicit (automatic) memoization:
 - problem structure or algorithm is not well understood.
 - Need to pay overhead of data-structure.
 - Functional languages (e.g., LISP) automatically do memoization, usually via hashing based dictionaries.

Explicit/implicit memoization for Fibonacci

```
Init:  $M[i] = -1, i = 0, \dots, n.$ 
```

```
Fib( $k$ ):  
    if ( $k = 0$ )  
        return 0  
    if ( $k = 1$ )  
        return 1  
    if ( $M[k] \neq -1$ )  
        return  $M[k]$   
     $M[k] \leftarrow \text{Fib}(k-1) + \text{Fib}(k-2)$   
    return  $M[k]$ 
```

Explicit memoization

```
Init: Init dictionary  $D$ 
```

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

Implicit memoization

Dynamic programming

Removing the recursion by filling the table in the right order

```
Fib(n):  
    if (n = 0)  
        return 0  
    if (n = 1)  
        return 1  
    if (M[n] ≠ -1)  
        return M[n]  
    M[n] ← Fib(n - 1) + Fib(n - 2)  
    return M[n]
```

```
FibIter(n):  
    if (n = 0) then  
        return 0  
    if (n = 1) then  
        return 1  
    F[0] = 0  
    F[1] = 1  
    for i = 2 to n do  
        F[i] = F[i - 1] + F[i - 2]  
    return F[n]
```

Dynamic programming: Saving space!

Saving space. Do we need an array of n numbers? Not really.

```
FibIter( $n$ ):  
    if ( $n = 0$ ) then  
        return 0  
    if ( $n = 1$ ) then  
        return 1  
     $F[0] = 0$   
     $F[1] = 1$   
    for  $i = 2$  to  $n$  do  
         $F[i] = F[i - 1] + F[i - 2]$   
    return  $F[n]$ 
```

```
FibIter( $n$ ):  
    if ( $n = 0$ ) then  
        return 0  
    if ( $n = 1$ ) then  
        return 1  
     $prev2 = 0$   
     $prev1 = 1$   
    for  $i = 2$  to  $n$  do  
         $temp = prev1 + prev2$   
         $prev2 = prev1$   
         $prev1 = temp$   
  
    return  $prev1$ 
```

Dynamic programming – quick review

Dynamic Programming is smart recursion

Dynamic programming – quick review

Dynamic Programming is smart recursion

+ explicit memorization

Dynamic programming – quick review

Dynamic Programming is smart recursion

- + explicit memorization

- + filling the table in right order

- + removing recursion.

Analyzing memorized recursive function

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.

Analyzing memorized recursive function

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 memorize the recursion.

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

Analyzing memorized recursive function

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 memorize 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 memorized version of $foo(x)$ if $|x| = n$?

Analyzing memorized recursive function

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 memorize 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 memorized version of $foo(x)$ if $|x| = n$? $O(A(n)B(n))$.

Fibonacci numbers are big –
corrected running time analysis

Back to Fibonacci Numbers

T Is the iterative algorithm a polynomial time algorithm? Does it take $O(n)$ time?

- input is n and hence input size is $\Theta(\log n)$
- output is $F(n)$ and output size is $\Theta(n)$. Why?
- Hence output size is exponential in input size so no polynomial time algorithm possible!
- Running time of iterative algorithm: $\Theta(n)$ additions but number sizes are $O(n)$ bits long! Hence total time is $O(n^2)$, in fact $\Theta(n^2)$. Why?

Longest Increasing Sub-sequence Revisited

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 sub-sequence 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

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

Longest Increasing Subsequence Problem

Input A sequence of numbers a_0, a_1, \dots, a_{n-1}

Goal Find an increasing subsequence $a_{i_0}, a_{i_1}, \dots, a_{i_k}$ of maximum length

Longest Increasing Subsequence Problem

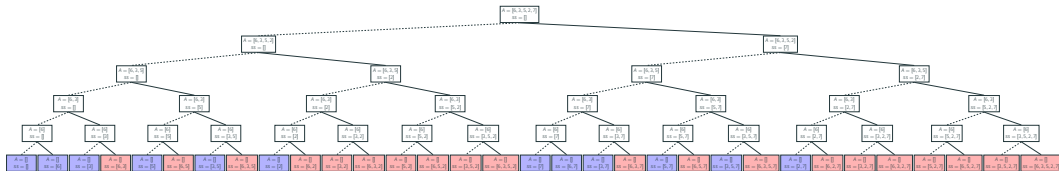
Input A sequence of numbers a_0, a_1, \dots, a_{n-1}

Goal Find an increasing subsequence $a_{i_0}, a_{i_1}, \dots, a_{i_k}$ of maximum length

Example

- Sequence: 6, 3, 5, 2, 7, 8, 1
- Increasing subsequences: 6, 7, 8 and 3, 5, 7, 8 and 2, 7 etc
- Longest increasing subsequence: 3, 5, 7, 8

Naive Recursion Enumeration - State Tree



- This is just for [6,3,5,2,7]! (Tikz won't print larger trees)
- How many leafs are there for the full [6,3,5,2,7, 8, 1] sequence
- What is the running time?

Naive Recursion Enumeration - Code

Assume a_1, a_2, \dots, a_n is contained in an array A

```
algLISNaive( $A[1..n]$ ):  
     $max = 0$   
    for each subsequence  $B$  of  $A$  do  
        if  $B$  is increasing and  $|B| > max$  then  
             $max = |B|$   
  
    Output  $max$ 
```

Running time: $O(n2^n)$.

2^n subsequences of a sequence of length n and $O(n)$ time to check if a given sequence is increasing.

Backtracking Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[0..n - 1]$):

Backtracking Approach: LIS: Longest increasing subsequence

Can we find a recursive algorithm for LIS?

LIS($A[0..n - 1]$):

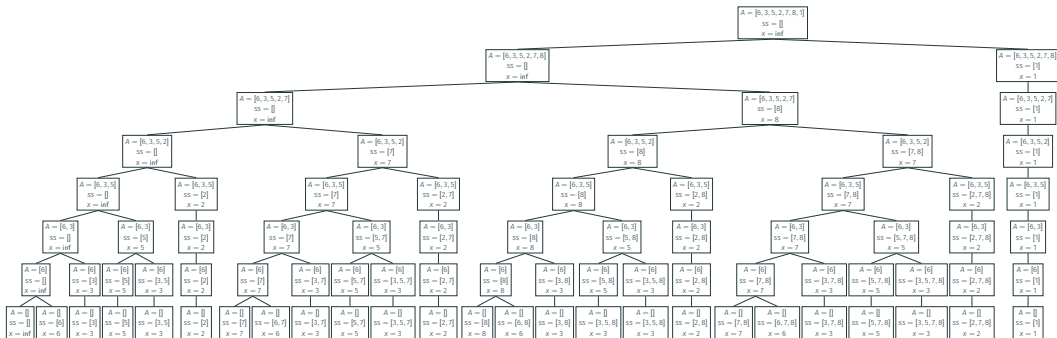
- **Case 1:** Does not contain $A[n - 1]$ in which case $\text{LIS}(A[0..n - 1]) = \text{LIS}(A[0..(n - 1)])$
- **Case 2:** contains $A[n - 1]$ in which case $\text{LIS}(A[0..n - 1])$ is not so clear.

Observation

For second case we want to find a subsequence in $A[1..(n - 2)]$ that is restricted to numbers less than $A[n - 1]$. This suggests that a more general problem is $\text{LIS_smaller}(A[0..n - 1], x)$ which gives the longest increasing subsequence in A where each number in the sequence is less than x .

Example

Sequence: $A[0..6] = 6, 3, 5, 2, 7, 8, 1$



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

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

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

Recursive Approach

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

Recursive Approach

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

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

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

Recursive Approach

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

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

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

Naming sub-problems and recursive equation

After seeing that number of sub-problems 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 sub-problems and recursive equation

After seeing that number of sub-problems 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: $LIS(n, n + 1)$.

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter
		1	2	3	4	5	6	7	8	j
[]	0									
[6]	1									
[6,3]	2									
[6,3,5]	3									
[6,3,5,2]	4									
[6,3,5,2,7]	5									
[6,3,5,2,7,8]	6									
[6,3,5,2,7,8,1]	7									
Represents sub-array i										

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter
		1	2	3	4	5	6	7	8	j
[]	0	0	0	0	0	0	0	0	0	
[6]	1									
[6,3]	2									
[6,3,5]	3									
[6,3,5,2]	4									
[6,3,5,2,7]	5									
[6,3,5,2,7,8]	6									
[6,3,5,2,7,8,1]	7									
Represents sub-array i										

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter j
		1	2	3	4	5	6	7	8	
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2									
[6,3,5]	3									
[6,3,5,2]	4									
[6,3,5,2,7]	5									
[6,3,5,2,7,8]	6									
[6,3,5,2,7,8,1]	7									

Represents sub-array i

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter j
		1	2	3	4	5	6	7	8	
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3									
[6,3,5,2]	4									
[6,3,5,2,7]	5									
[6,3,5,2,7,8]	6									
[6,3,5,2,7,8,1]	7									

Represents sub-array i

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter j
		1	2	3	4	5	6	7	8	
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3				0	2	2	0	2	
[6,3,5,2]	4									
[6,3,5,2,7]	5									
[6,3,5,2,7,8]	6									
[6,3,5,2,7,8,1]	7									

Represents sub-array i

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter j
		1	2	3	4	5	6	7	8	
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3				0	2	2	0	2	
[6,3,5,2]	4					2	2	0	2	
[6,3,5,2,7]	5									
[6,3,5,2,7,8]	6									
[6,3,5,2,7,8,1]	7									

Represents sub-array i

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter
		1	2	3	4	5	6	7	8	j
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3				0	2	2	0	2	
[6,3,5,2]	4					2	2	0	2	
[6,3,5,2,7]	5						3	0	3	
[6,3,5,2,7,8]	6									
[6,3,5,2,7,8,1]	7									

Represents sub-array i

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter
		1	2	3	4	5	6	7	8	j
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3				0	2	2	0	2	
[6,3,5,2]	4					2	2	0	2	
[6,3,5,2,7]	5						3	0	3	
[6,3,5,2,7,8]	6							0	4	
[6,3,5,2,7,8,1]	7									

Represents sub-array i

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

How to order bottom up computation?

		A[1] = 6 1	A[2] = 3 2	A[3]=5 3	A[4]=2 4	A[5]=7 5	A[6]=8 6	A[7]=1 7	inf 8	Represents limiter j
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3				0	2	2	0	2	
[6,3,5,2]	4					2	2	0	2	
[6,3,5,2,7]	5						3	0	3	
[6,3,5,2,7,8]	6							0	4	
[6,3,5,2,7,8,1]	7								4	
Represents sub-array i										

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

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

Iterative algorithm

The dynamic program for longest increasing subsequence

```
LIS-Iterative( $A[1..n]$ ):  
   $A[n + 1] = \infty$   
  int  $LIS[0..n - 1, 0..n]$   
  for  $j = 0 \dots n$  if  $A[i] \leq A[j]$  then  $LIS[0][j] = 1$   
  
  for  $i = 1 \dots n - 1$  do  
    for  $j = i \dots n - 1$  do  
      if ( $A[i] \geq A[j]$ )  
         $LIS[i, j] = LIS[i - 1, j]$   
      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: $O(n^2)$

Iterative algorithm

The dynamic program for longest increasing subsequence

```
LIS-Iterative( $A[1..n]$ ):  
   $A[n + 1] = \infty$   
  int  $LIS[0..n - 1, 0..n]$   
  for  $j = 0 \dots n$  if  $A[i] \leq A[j]$  then  $LIS[0][j] = 1$   
  
  for  $i = 1 \dots n - 1$  do  
    for  $j = i \dots n - 1$  do  
      if ( $A[i] \geq A[j]$ )  
         $LIS[i, j] = LIS[i - 1, j]$   
      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: $O(n^2)$ Can be done in linear space. How?

Finding the sub-sequence

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter j
		1	2	3	4	5	6	7	8	
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3				0	2	2	0	2	
[6,3,5,2]	4					2	2	0	2	
[6,3,5,2,7]	5						3	0	3	
[6,3,5,2,7,8]	6							0	4	
[6,3,5,2,7,8,1]	7								4	

Represents sub-array i

Sequence: A[1...7]

= [6, 3, 5, 2, 7, 8, 1]

We know the LIS length
(4) but how do we find
the LIS itself?

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

Finding the sub-sequence

		A[1] = 6	A[2] = 3	A[3]=5	A[4]=2	A[5]=7	A[6]=8	A[7]=1	inf	Represents limiter
		1	2	3	4	5	6	7	8	j
[]	0	0	0	0	0	0	0	0	0	
[6]	1		0	0	0	1	1	0	1	
[6,3]	2			1	0	1	1	0	1	
[6,3,5]	3				0	2	2	0	2	
[6,3,5,2]	4					2	2	0	2	
[6,3,5,2,7]	5						3	0	3	
[6,3,5,2,7,8]	6							0	4	
[6,3,5,2,7,8,1]	7								4	

Represents sub-array i

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

We know the LIS length
 (4) but how do we find
 the LIS itself?

$$LIS(i, j) = \begin{cases} 0 & i = 0 \\ LIS(i-1, j) & A[i] \geq A[j] \\ \max \begin{cases} LIS(i-1, j) \\ 1 + LIS(i-1, i) \end{cases} & A[i] < A[j] \end{cases}$$

Two comments

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

Yes!

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.

How to come up with dynamic programming algorithm: summary

Dynamic Programming

- Find a “smart” recursion for the problem in which the number of distinct sub-problems is small; polynomial in the original problem size.
- Estimate the number of sub-problems, the time to evaluate each sub-problem and the space needed to store the value.
- Come up with an explicit memorization algorithm for the problem.
- ...need to find the right way or order the sub-problems evaluation. This leads to an a dynamic programming algorithm.
- Profit!

