# CS314 - Dynamic Programming: LIS

Note Title

9/11/2013

## Announcements

- HW0 is graded
- HW1 due Friday
- BBQ is today at 4

# What is dynamic programming?

designate "as you go"

↳ filling in an array, instead of "proper" recursion

Example (from reading): Fibonacci numbers

Computing $F(n) = F(n-1) + F(n-2)$

exponential : $O(2^n)$
versus
linear : $O(n)$

Key tool: memoization

# Longest Increasing Subsequence

Input: an array $A[1..n]$

Output: indices $i_1, \dots, i_k$ (with $k$ as large as possible) s.t. $A[i_j] < A[i_{j+1}]$ for all $j$.

Ex: A: $\boxed{5, 2, 8, 6, 3, 6, 9, 7}$

3, 6, 9 indices: 5, 6, 7

2, 3, 6, 9

# Step 1: A recursive formulation

A subsequence is either

- empty
- an integer followed by a sequence

Here, given A[1..n]:

- A[1] followed by subseq. of A[2..n]
- subseq of A[2..n]

Modify:

Longest increasing subsequence:

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

- size 0 or 1 — trivial - return 0
- $LIS(A[2..n])$ ← length

  or
- $LIS(A[2..n]$ with all $> A[1])$ ← length $+1$

# Pseudo code :

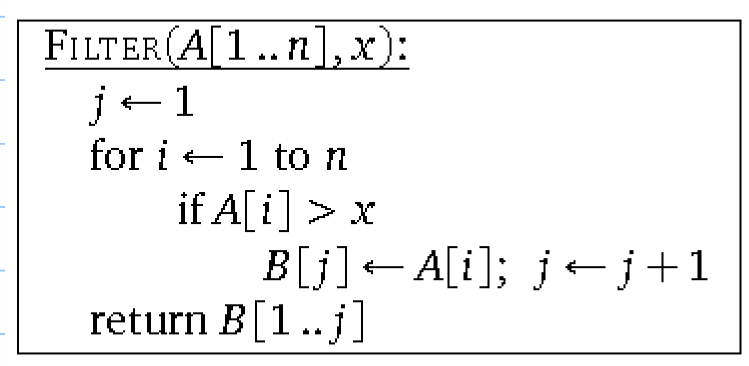
```
LIS($A[1..n]$):
    return LISBigger($-\infty, A[1..n]$)
```

```
LISBigger($prev, A[1..n]$):
    if $n = 0$
        return 0
    else
        $max \leftarrow$ LISBigger($prev, A[2..n]$)
        if $A[1] > prev$
            $L \leftarrow 1 +$ LISBigger($A[1], A[2..n]$)
            if $L > max$
                $max \leftarrow L$
        return $max$
```

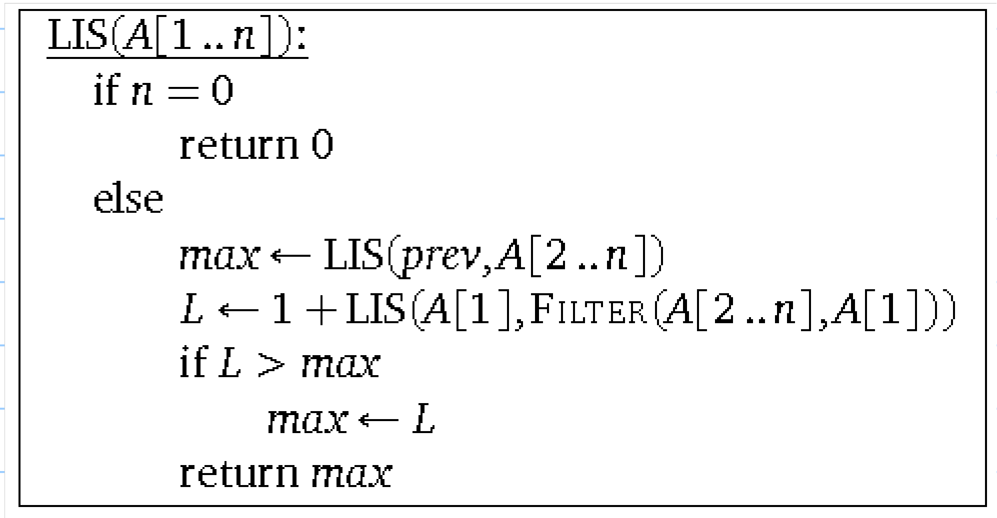
skip A[1]

$prev, \underbrace{\qquad}_{A[1]}$

# Alternative:



```
FILTER($A[1..n], x$):
    $j \leftarrow 1$
    for $i \leftarrow 1$ to $n$
        if $A[i] > x$
            $B[j] \leftarrow A[i]$;  $j \leftarrow j + 1$
    return $B[1..j]$
```



```
LIS($A[1..n]$):
    if $n = 0$
        return 0
    else
        $max \leftarrow$ LIS($prev, A[2..n]$)
        $L \leftarrow 1 +$ LIS($A[1]$, FILTER($A[2..n], A[1]$))
        if $L > max$
            $max \leftarrow L$
        return $max$
```

## Runtime:

$$L(n) = 2L(n-1) + O(1)$$

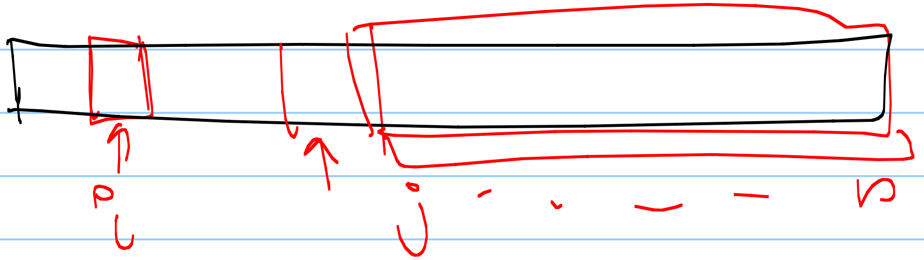
$$\Rightarrow \quad L(n) = \underline{O(2^n)}$$

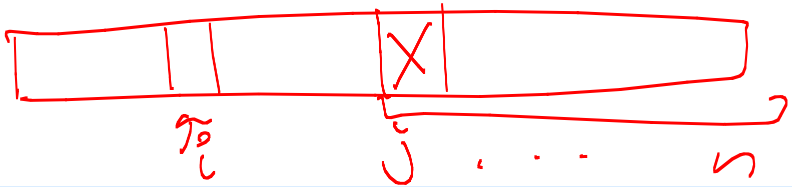
BAD

## Speeding up: Memoization!

**Note:**
- Input to LISBIGGER is always either $-\infty$ or an element of A.
- Other input is a suffix of A.

Turn this into a recursive formulation!



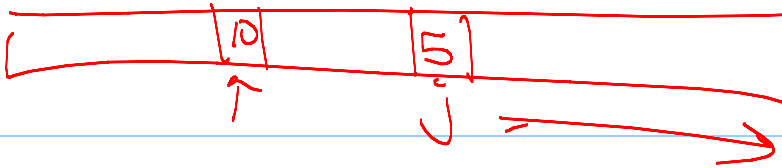


So: Add sentinal $A[0] = -\infty$.

Let $LIS(i,j)$ = longest increasing subsequence of $A[j..n]$ with all elements $> A[i]$.

What are our cases?

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



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

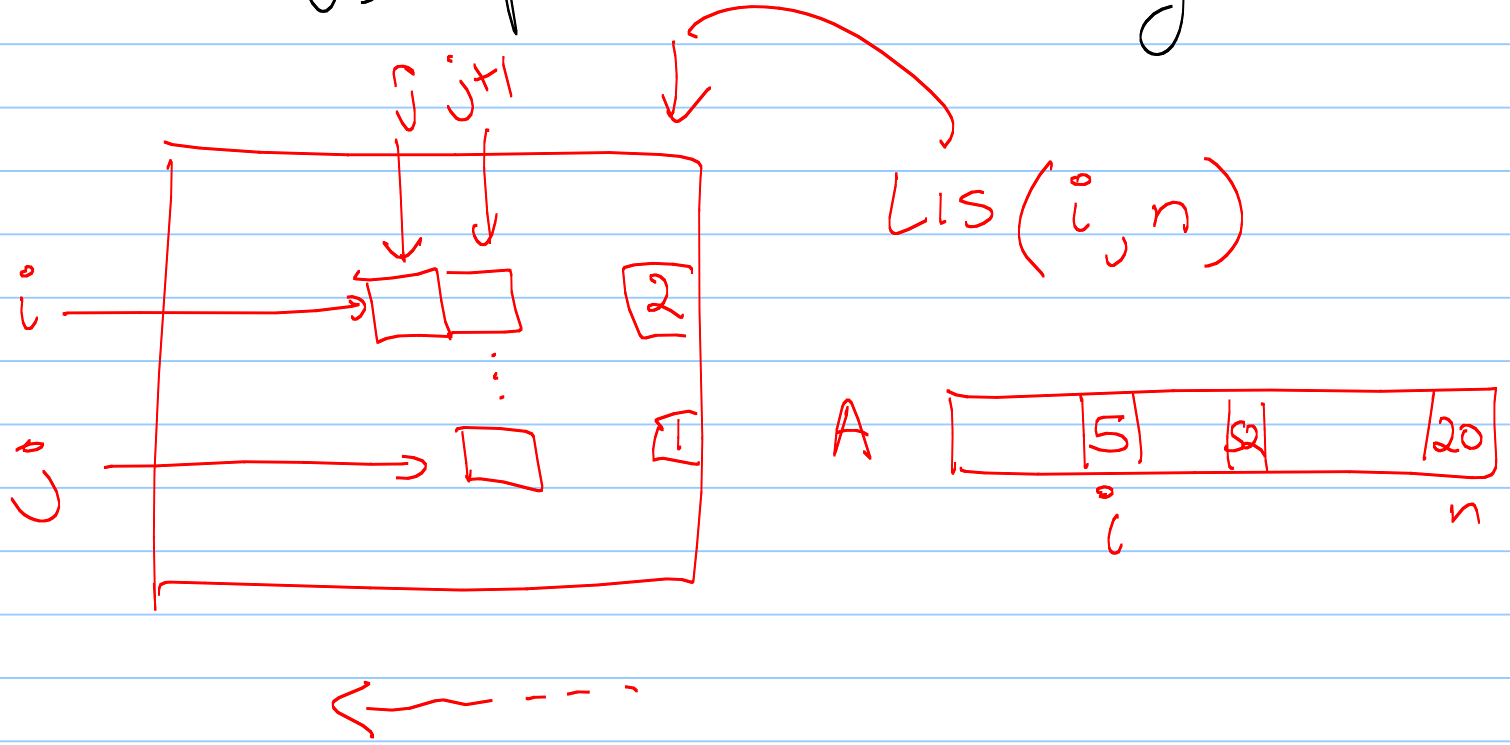
Cases again:

→ A[j] is smaller than A[i], so can't take it

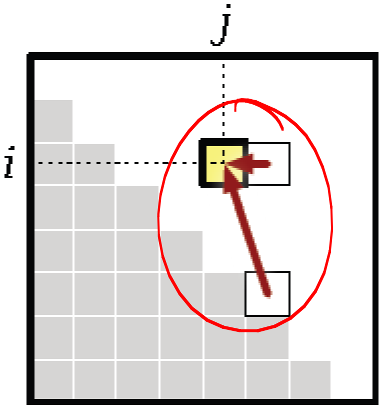
→ try with & without A[j]

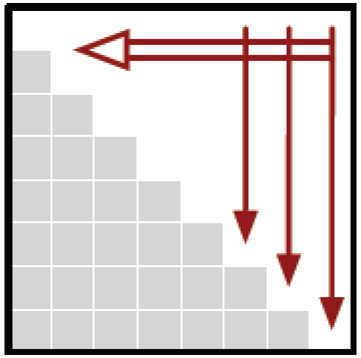
So how to memoize?

LIS(i,j) depends on knowing 2 values:



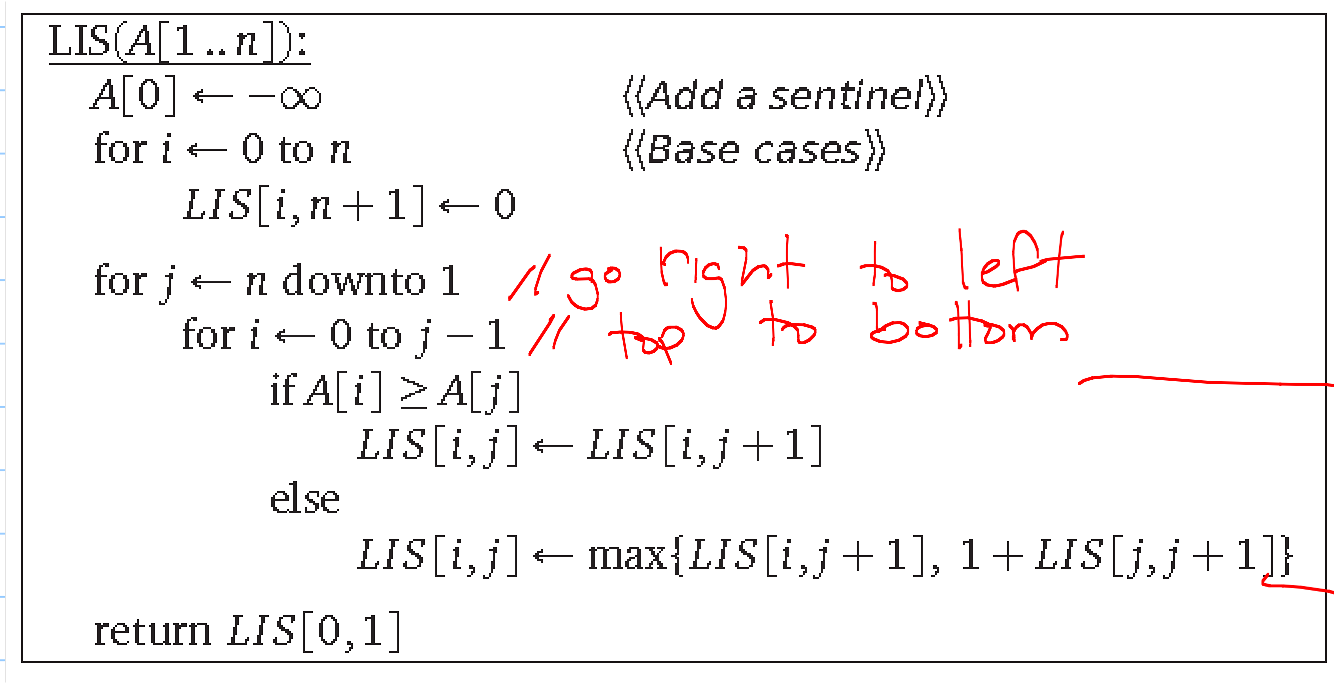
Picture: dependencies are limited.





For each entry in table, do:
- if (or 2)
- ≤2 table lookups
- add 1
- take max

} $O(1)$



```
LIS(A[1..n]):
  $A[0] \leftarrow -\infty$                    ⟨⟨Add a sentinel⟩⟩
  for $i \leftarrow 0$ to $n$                  ⟨⟨Base cases⟩⟩
      $LIS[i, n+1] \leftarrow 0$

  for $j \leftarrow n$ downto 1        // go right to left
      for $i \leftarrow 0$ to $j - 1$  // top to bottom
          if $A[i] \geq A[j]$
              $LIS[i, j] \leftarrow LIS[i, j+1]$
          else
              $LIS[i, j] \leftarrow \max\{LIS[i, j+1],\ 1 + LIS[j, j+1]\}$

  return $LIS[0, 1]$
```

$O(1)$

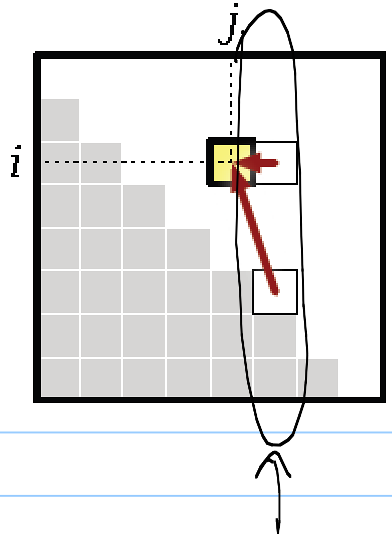
## Time and space:

$O(n^2)$ time – $O(1)$ per spot in table

$O(n^2)$ space

# Better space: Do we need entire table?

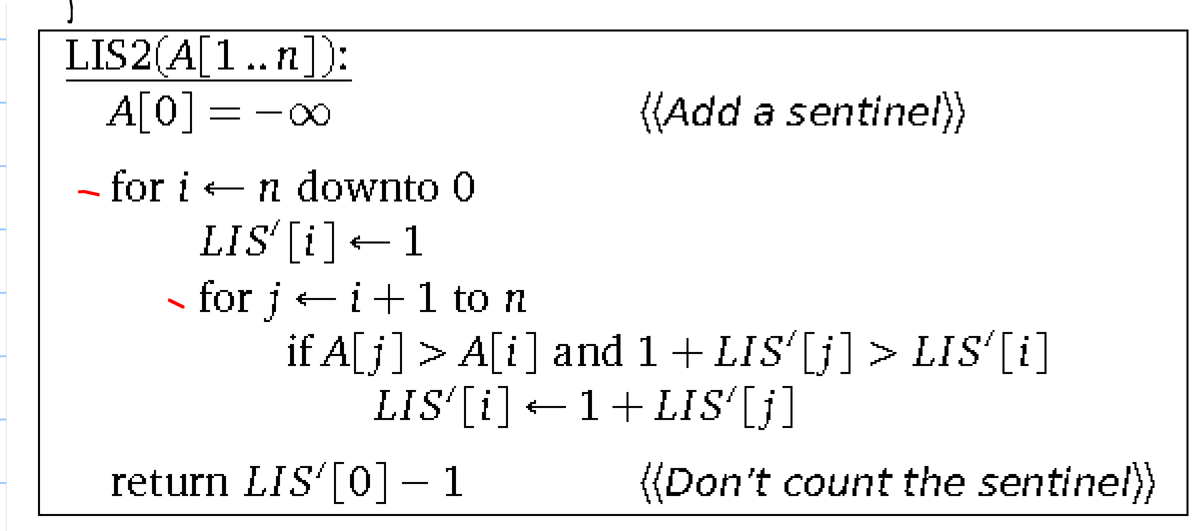
LIS(i, j) depends only on LIS(i, j+1) and LIS(j, j+1)



Conclusion:

store last row only

Better pseudocode:



```
LIS2(A[1 .. n]):
  $A[0] = -\infty$                                  ⟨⟨Add a sentinel⟩⟩
  for $i \leftarrow n$ downto 0
      $LIS'[i] \leftarrow 1$
      for $j \leftarrow i+1$ to $n$
          if $A[j] > A[i]$ and $1 + LIS'[j] > LIS'[i]$
              $LIS'[i] \leftarrow 1 + LIS'[j]$
  return $LIS'[0] - 1$                              ⟨⟨Don't count the sentinel⟩⟩
```

Time & space: $O(n^2)$ time

$O(n)$ space

## Recap:

Dynamic programming is smart recursion:

- recurse, but don't repeat!

Often will store previous values in some kind of "table", + then "recursely look-up" in table in some sensible order.

## Steps:

① Formulate recursion.

② Build solution from base case up.

- identify subproblems
- identify dependencies
  ie $F(n)$ depends on $F(n-1)$ + $F(n-2)$
- choose data structure to store results
- choose evaluation order
- write pseudocode, analyze time + space