CS 6363.003.21S Lecture 8—February 16, 2021

Main topics are #dynamic_programming with #example/edit_distance and #example/longest_increasing_subsequence.

Edit Distance

- We're going to continue with dynamic programming today by looking at a couple problems on sequences and strings.
- The *edit distance* between two strings is the minimum number of character insertions, deletions, and substitutions required to transform one string into the other.
- For example, the edit distance between FOOD and MONEY is at most four:

$$\texttt{\underline{F}OOD} \longrightarrow \texttt{MO\underline{O}D} \longrightarrow \texttt{MON}_{\wedge}\texttt{D} \longrightarrow \texttt{MONE\underline{D}} \longrightarrow \texttt{MONEY}$$

- This distance function was independently proposed by Vladimir Levenshtein (working in coding theory), Taras Vintsyuk (working on speech recognition), and Stanislaw Ulam (working on biological sequences), so it's often referred to as Levenshtein or Ulam distance (but never Vintsyuk distance for some reason).
- The way I defined the problem lets us do edits in any order, but adding some organization will help us design a dynamic programming algorithm.
- One way to add some organization is to ask what happens to each character of each string.
- So let's visualize the editing process by aligning the two strings on top of one another to represents what happens to a character of the first string or where the character of the second string comes from.

```
F O O   D
M O N E Y
```

- There's a gap in the first word for every insertion and a gap in the second word for every deletion. If a column contains two *different* characters then it represents a substitution.
- Another example, ALGORITHM vs ALTRUISTIC. The edit distance is at most 6.

```
A L G O R   I   T H M
A L   T R U I S T I C
```

- But how do we compute the edit distance exactly? Let's design a dynamic programming algorithm that, given two strings A[1 .. m] and B[1 .. n], returns the edit distance between A and B.

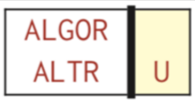
Recursive Structure

- The first step in any dynamic programming algorithm is developing a recursive algorithm or recurrence.
- This alignment representation suggests a recursive structure.

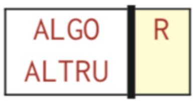
- **If we remove the last column from the optimal alignment structure, then the remaining columns must represent the shortest edit sequence for the remaining prefixes.**
- In other words, let's figure out what one operation should happen to the last character of one or both strings. And then recursively do the optimal operations to deal with the earlier characters.
- In order to turn this into a proper recursive strategy, we need to figure out what precisely the input to the recursive subproblems should be.
- Because we're always working with prefixes of both strings, we only need to pass in the index of the last characters.
- Let Edit(i, j) denote the edit distance between A[1 .. i] and B[1 .. j]. We ultimately need to compute Edit(m, n).

Recurrence

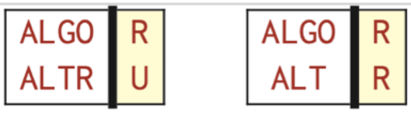
- When i and j are both positive, there are three possibilities for the last column.
    - Insertion: The last entry of the top row is empty. Here, the edit distance is equal to 1 + Edit(i, j - 1). The 1 is the "cost" of doing one insertion and the recursive call is us having to handle all i characters of A[1 .. i] still but only the first j - 1 characters of B[1 .. j].



    - Deletion: The last entry of the bottom row is empty. Here, the edit distance is equal to 1 + Edit(i - 1, j). Here, we have all of B[1 .. j] left, but we already took care of the last character of A[1 .. i].



    - Substitution(?): Both rows have characters in the last column. If the characters are different, the edit distance is 1 + Edit(i - 1, j - 1). Otherwise, the "substitution" is free so the edit distance is Edit(i - 1, j - 1).



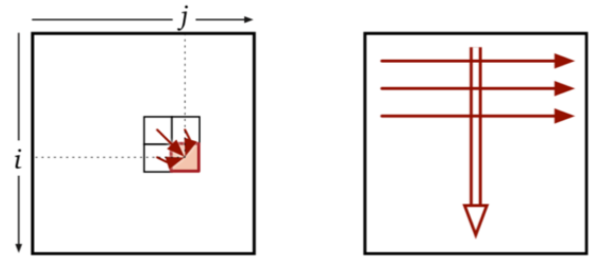
- This case analysis doesn't work when one of i or j is equal to 0, but we can handle the boundary cases easily.
    - Converting an empty string into a string of length j requires j insertions, so Edit(0, j) = j.
    - Converting a string of length i into the empty string requires i deletions, so Edit(i, 0) = i.
- And either of those rules imply the edit distance of two empty strings Edit(0, 0) = 0. Checks out.

- Given a proposition P, let [P] = 1 if P is true and 0 otherwise.
- We either follow one of those base cases, or pick the cheapest of the three options. Therefore, the Edit function follows this recurrence:

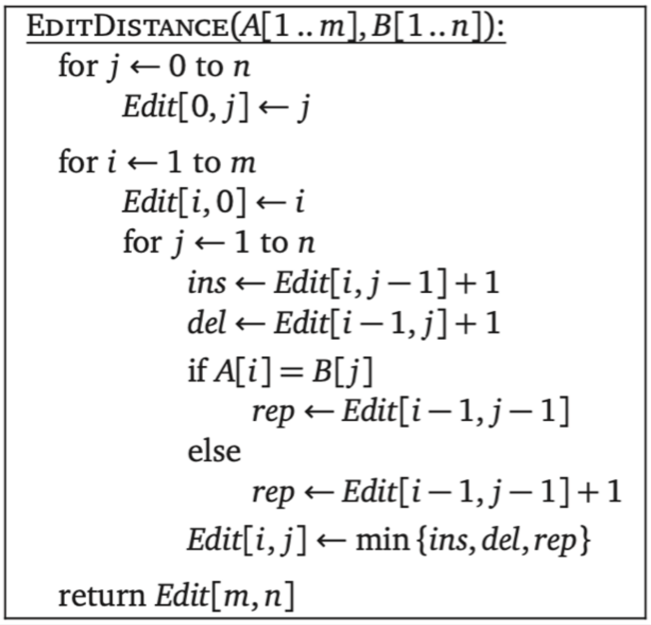
$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

Dynamic Programming

- And now we can find our efficient algorithm using the mechanical memoization process.
  - Subproblems: Each recursive subproblem takes a pair of indices 0 ≤ i ≤ m and 0 ≤ j ≤ n.
  - Memoization: So we can memoize all possible values of Edit(i, j) in a two-dimensional array Edit[0 .. m, 0 .. n].
  - Dependencies: Each entry Edit[i, j] depends only on its three neighboring entires Edit[i, j - 1], Edit[i - 1, j], and Edit[i - 1, j - 1]. Here's a picture of what it looks like.

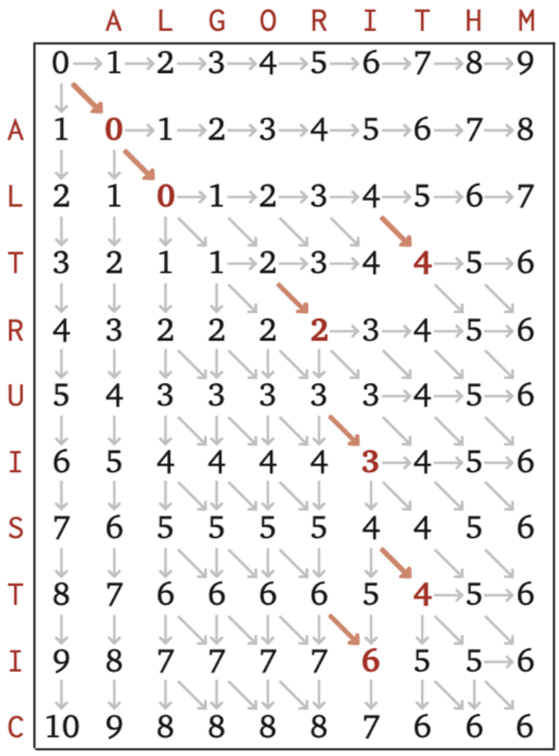


  - Evaluation order: We only need things from earlier in the same row or from the previous row, so let's fill the array in row-major order: row by row from top down, each row from left to right.
  - Space and time: At this point, we can already figure out the running time without writing any code! The memoization structure uses **O(mn) space**. We can compute each entry Edit(i, j) in O(1) time once we know its predecessors, so computing all the entries will take **O(mn) time**.
- And finally, here's the algorithm!



```
EDITDISTANCE($A[1..m], B[1..n]$):
    for $j \leftarrow 0$ to $n$
        $Edit[0, j] \leftarrow j$
    for $i \leftarrow 1$ to $m$
        $Edit[i, 0] \leftarrow i$
        for $j \leftarrow 1$ to $n$
            $ins \leftarrow Edit[i, j-1] + 1$
            $del \leftarrow Edit[i-1, j] + 1$
            if $A[i] = B[j]$
                $rep \leftarrow Edit[i-1, j-1]$
            else
                $rep \leftarrow Edit[i-1, j-1] + 1$
            $Edit[i, j] \leftarrow \min\{ins, del, rep\}$
    return $Edit[m, n]$
```

- We started by finding the recursive structure and eventually got to a relatively simple iterative process.
- A brief historical note: Most people attribute this algorithm to Wagner and Fischer who described it in 1974. But it was found by several others at the same time or even a bit earlier.
- Now, similar to how we have recursion trees to explain what's going on with our recursive algorithms, we can also look at how the array is filled. I want to emphasize again, though, that **designing the recursive algorithm was the key step**. We're only looking at the array to see what happens in hindsight.



		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

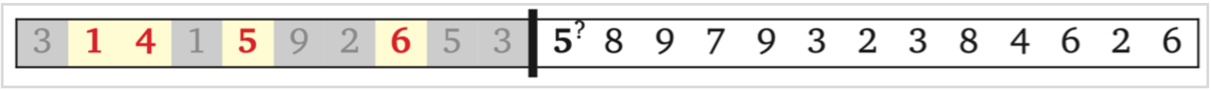
- Suppose we're transforming ALTRUISTIC into ALGORITHM.
- A number at position (i, j) is the value of Edit(i, j). The arrows indicate which predecessors

could have defined each entry. A horizontal arrow means we did an insertion. A vertical arrow represents deletion, and a diagonal arrow represents substitution. Bold arrows are the “free" substitutions that don't increase the cost.

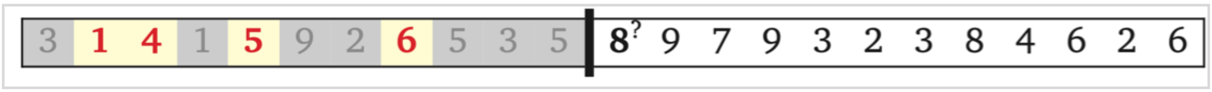
- The algorithm we wrote only computes the total edit distance, not the optimal sequence of operations. However, any path of arrows from (0, 0) to (m, n) represents an optimal sequence of operations.
- If we want to recover an optimal sequence, we can figure out which arrows lead into any entry (i, j) in O(1) time by checking its three dependences and see which one gives us the correct cost. In O(m + n) additional time over what it takes to compute the edit distance, we can reconstruct an optimal sequence by tracing *backwards* from (m, n).
- This is one example of a trend. If you want to find an optimal solution for some problem using dynamic programming, its easiest to first write an algorithm to find the *value* of the solution and then make any changes necessary so the algorithm obtains the solution using the values.

Longest Increasing Subsequence

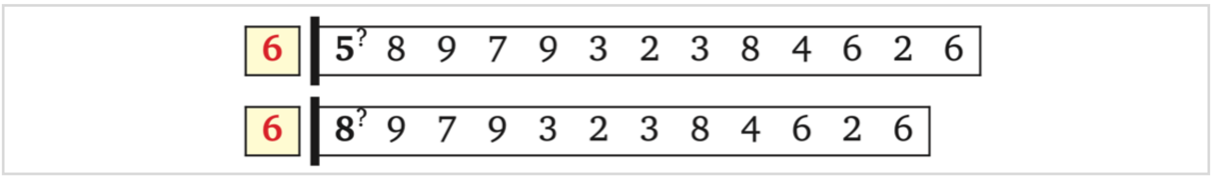
- And time permitting, here’s one more example.
- Given a sequence S, we can define a *subsequence* as another sequence obtained from S by deleting zero or more elements. The subsequence elements remain in the same order but aren't necessarily contiguous.
- In contrast, a *substring*'s elements are contiguous in the original sequence. So MASHER and LAUGHTER are subsequences of MANSLAUGHTER, but only LAUGHTER is a substring.
- Now, suppose we're given a sequence of *integers* as an array A[1 .. n]. The *longest increasing subsequence* of A is the longest subsequence where subsequent elements only increase in value. It can be described as a sequence of indices 1 ≤ i_1 < i_2 < ... < i_ell ≤ n such that A[i_k] < A[i_{k+1}] for all k. Suppose we want to find the length of the longest increasing subsequence.
- So we should start by designing a backtracking approach. Consider each element of the sequence in order. We need to decide for each index j from 1 to n whether or not to include A[j] in the subsequence we output. Here's what the world looks like partway through this process.



- So we’ve decided to include 1, 4, 5, and 6. We still need to process everything to the right of the black bar. And right at this moment, we need to decide whether or not to include the 5. So should we?
- Yeah, we can't based on our previous decisions, so we move on



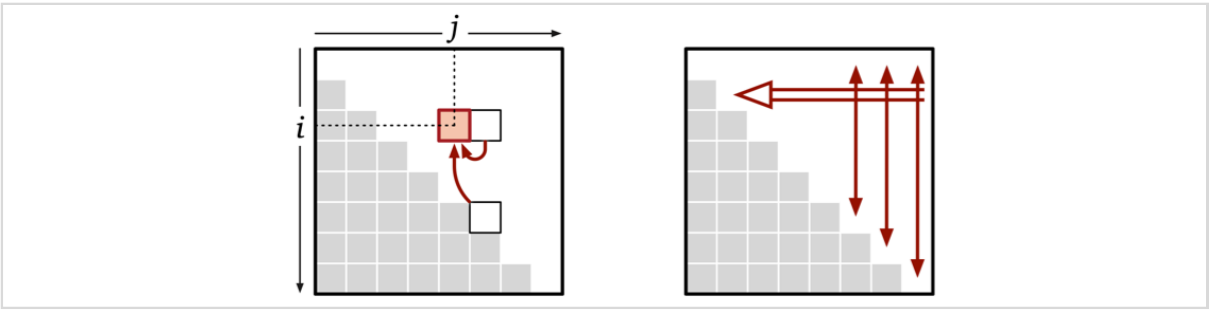
- Now, we *can* include the 8, but we don't know if we should. We'll have to do two recursive calls. One where we include the 8 and one where we don't. We'll take the one that returns the longer increasing subsequence of what's remaining.
- Part of designing a successful backtracking algorithm is telling the Recursion Fairy exactly as much information as necessary to be consistent with earlier decisions. What do we need to remember about our past decisions here for the recursive call to be consistent?
- Well, if we assume the previous decisions describe an increasing subsequence, all we need to remember is the last (and biggest) term.
- All the Recursion Fairy needs to see is the following:



- At this point, we could write out a recursive algorithm where we pass in the value of the previously selected element. But for the sake of dynamic programming, it would be a lot easier if we could restrict how many different numbers can be passed to our recursive call.
- So we'll pass in the *index* of the previous element. And since we're always working with suffixes, we'll also pass the index of the suffix's first term.
- Let LISbigger(i, j) denote the length of the longest increasing subsequence of A[j .. n] in which every element is larger than A[i]. If we pretend A[0] = -∞, then our algorithm needs to compute LISbigger(0, 1).
- Based on the earlier discussion, we can get the following recursion definition for LISbigger(i, j). Again, an empty string only has empty subsequences (which are trivially increasing).

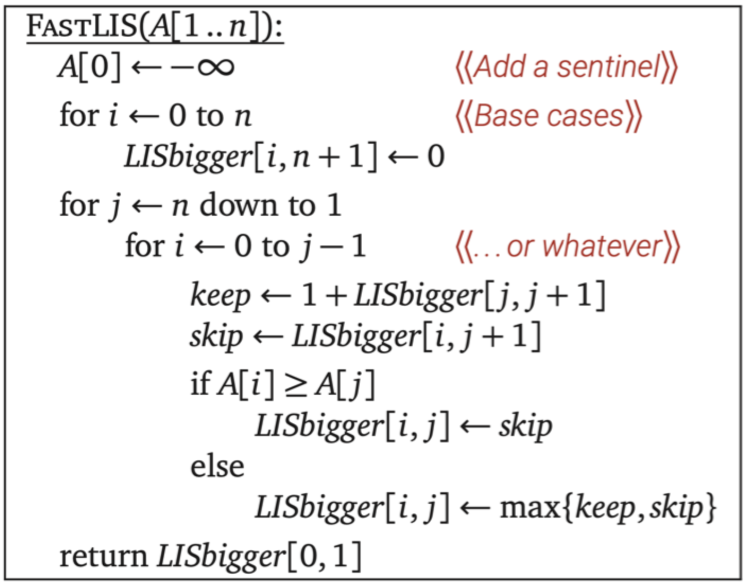
$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\left\{\begin{array}{c} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{array}\right\} & \text{otherwise} \end{cases}$$

- And now we're ready to speed things up by using memoization.
  - Subproblems: Each recursive subproblem takes a pair of indices 0 ≤ i ≤ n and 1 ≤ j ≤ n.
  - Memoization: So we can memoize all possible values of LISbigger(i, j) in a two-dimensional array LISbigger[0 .. n, 1 .. n].
  - Dependencies: Each entry LISbigger[i, j] depends on at most two other entires LISbigger[i, j + 1] and LISbigger[j, j + 1]. Here's a picture of what it looks like.



- Evaluation order: We only need things from a later column, so let’s fill the array in column by column from right to left. In this case, the order in which we fill a column’s entries doesn't matter, so we can do it in either direction.
- Space and time: The memoization structure uses **O(n^2) space**. We can compute each entry LISbigger(i, j) in O(1) time once we know its dependencies, so the algorithm will run in **O(n^2) time**.

- And here's the algorithm!

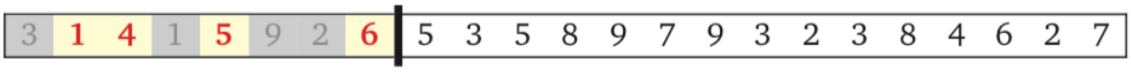


```
FastLIS($A[1..n]$):
    $A[0] \leftarrow -\infty$                          ⟨⟨Add a sentinel⟩⟩
    for $i \leftarrow 0$ to $n$                        ⟨⟨Base cases⟩⟩
        $LISbigger[i, n+1] \leftarrow 0$
    for $j \leftarrow n$ down to 1
        for $i \leftarrow 0$ to $j-1$                  ⟨⟨…or whatever⟩⟩
            $keep \leftarrow 1 + LISbigger[j, j+1]$
            $skip \leftarrow LISbigger[i, j+1]$
            if $A[i] \geq A[j]$
                $LISbigger[i, j] \leftarrow skip$
            else
                $LISbigger[i, j] \leftarrow \max\{keep, skip\}$
    return $LISbigger[0, 1]$
```

- For this problem, we could reduce the space to O(n) by only remembering the previous column. Edit distance is similar. Our goal should be to get a working (recursive) algorithm first, though, then polynomial time and space, and then worry about extra details like reducing the space further.

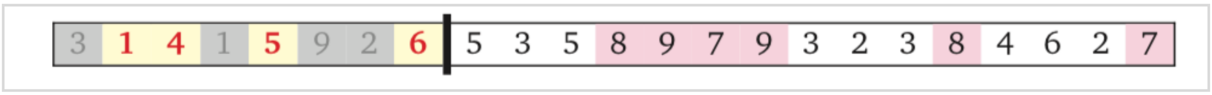
Longest Increasing Subsequence 2

- But that's not the only way to solve the problem. Suppose instead of deciding for each element in the input sequence, we instead just directly asked, what is the next element in the output sequence?
- So here, we've just decided to include the 6.

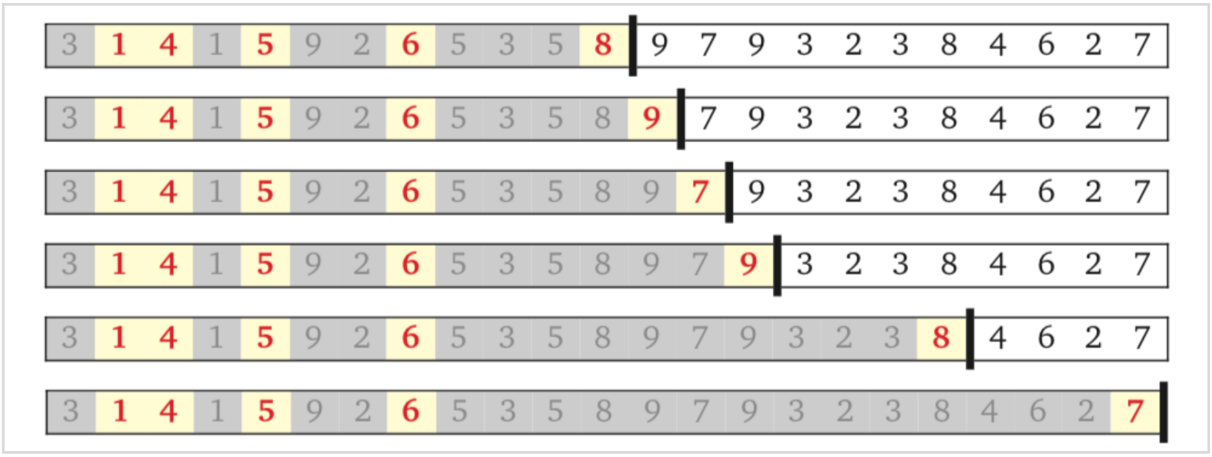


3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	2	3	8	4	6	2	7

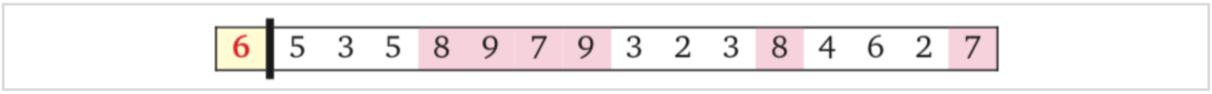
- The next number in the output must be bigger than 6.



- And so we'll try all the possibilities by finding longest increasing subsequences that start with things larger than the 6.



- Since we only need to remember that last number we included before deciding on the next, we can ignore everything before it.



- So we're sending this 6, and then *everything* after it to the Recursion Fairy, asking for a longest increasing subsequence that begins with that 6.
- Let LISfirst(i) denote the length of the longest increasing subsequence of A[i .. n] that begins with A[i].

$$LISfirst(i) = 1 + \max\left\{LISfirst(j) \;\middle|\; j > i \text{ and } A[j] > A[i]\right\}$$

- If we pretend the max over nothing is 0, then the base case LISfirst(n) = 1 without explicitly saying so.
- We can pretend A[0] = -∞ like before, but we don't actually want to include it as part of our subsequence, so we want to return LISfirst(0) - 1.
- And now to memoize.
  - Subproblems: Each recursive subproblem takes one index 0 ≤ i ≤ n.
  - Memoization: So we can memoize all possible values of LISfirst(i) in an array LISfirst[0 .. n].
  - Dependencies: Each entry LISfirst[i] depends on entires LISfirst[j] with j > i.
  - Evaluation order: So we fill the array in decreasing index order.
  - Space and time: The memoization structure uses **O(n) space**. We can compute each entry LISfirst[i] in O(n) time once we know its dependencies, so the algorithm will run in **O(n^2) time**.

```
FASTLIS2($A[1\,..\,n]$):
  $A[0] = -\infty$                      ⟨⟨Add a sentinel⟩⟩
  for $i \leftarrow n$ downto 0
      $LISfirst[i] \leftarrow 1$
      for $j \leftarrow i+1$ to $n$          ⟨⟨...or whatever⟩⟩
          if $A[j] > A[i]$ and $1 + LISfirst[j] > LISfirst[i]$
              $LISfirst[i] \leftarrow 1 + LISfirst[j]$
  return $LISfirst[0] - 1$             ⟨⟨Don't count the sentinel⟩⟩
```