Longest increasing subsequence with dynamic programming (Brown CS 200, Spring 2022, Milda Zizyte)

Wednesday, April 6, 2022 1:29 PM

Another example of a problem that can be solved with Dynamic Programming is **longest increasing subsequence**: find the length of the longest increasing subsequence of a list
Approach: write down examples of what the answers would look like when we start with each element of the list. Since the longest increasing sequence will have to start with one of those elements, the maximum such length will be our answer.

	A	B	C	D	E	F	G
1	3	7	4	2	5	lis starting w/ 3:	3 the max
2							
3		7	4	2	5	lis starting w/ 7:	1
4							
5			4	2	5	lis starting w/ 4:	2
6							
7				2	5	lis starting w/ 2:	2
8							
9					5	lis starting w/ 5:	1
10							
11							

Note: why is 1 the answer for the LIS starting w/ 7? Because the sub-problem we are solving is to find the subsequence that starts with the element in question. This way of framing the problem will let us use the answers for the smaller sub-problems to answer the bigger sub-problems more efficiently!

How do we populate the table? One way to think about this is recursively: if we are considering the length of the longest subsequence starting with the element at index *start_index*, and we consider every *possible_next_index* after start index, the answer for the longest increasing sequence starting with the element at *start_index* will be (in pseudocode)

```
l_i_s(start_index, input_lst):
  if start_index == len(input_lst) - 1:
     return 1
  else:
     return 1 + max(l_i_s(possible_next_index, input_lst))
                # for all possible_next_index > start_index with
                # input_lst[possible_next_index] > start_index
```

So, the computation for index 3 of the list would look like:
`l_i_s(3, input_lst) = 1 + max(l_i_s(4, input_lst))` *(because 4 > 3 and input_lst[4] > input_lst[3] (5 > 2)*

`1` *(because len(input_lst) is 4)*

=2

The computation for index 1 of the list would look like:
`l_i_s(1, input_lst) = 1 + max(nothing)` *(because there are no elements larger than input_lst[1] (7) in the rest of the list)* =1

The computation for index 0 of the list would look like:
`l_i_s(0, input_lst) = 1 + max(l_i_s(1, input_lst), l_i_s(2, input_lst), l_i_s(4, input_lst))` =3

`1 + max(nothing)` =1

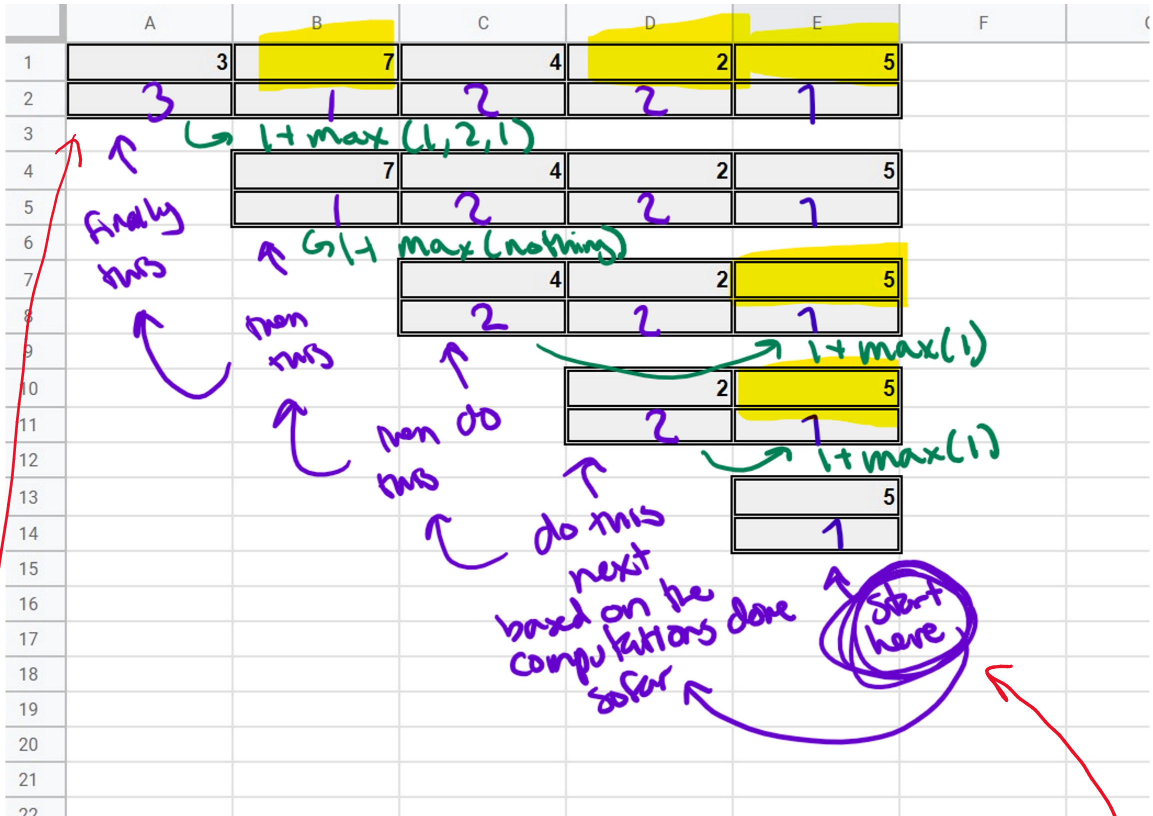
`1 + max(l_i_s(4, input_lst)` =2

`1`

redundant computations

`1`

We notice that that last computation has some redundancy when implemented recursively. Instead, we recognize that, if we store the result of these computations, from the largest starting index (smallest sub-list) to the smallest, then we could re-use them to do the computations for the smaller starting indices:



How do we translate this to python code? First, we set up the loop that allows us to fill in the computation table (longest_seq_so_far) back-to-front:

we are filling in this table

```
def l_i_s(lst : "list[int]") -> int:
    """Produces the length of the LIS in lst"""
    longest_seq_so_far = [0] * len(lst)
    for start_index in range(len(lst) - 1, -1, -1):
        …
        longest_seq_so_far[start_index] = …

    return max(longest_seq_so_far)
```

this loop accomplishes this order of computation

Then, we fill in the inner computation:

```
longest_seq_so_far = [0] * len(lst)
for start_index in range(len(lst) - 1, -1, -1):
    len_of_max_seq_from_start_index = 1
    for possible_next_index in range(start_index + 1, len(lst)):
        if lst[possible_next_index] > lst[start_index]:
            if longest_seq_so_far[possible_next_index] + 1 > len_of_max_seq_from_start_index:
                len_of_max_seq_from_start_index = longest_seq_so_far[possible_next_index] + 1
    longest_seq_so_far[start_index] = len_of_max_seq_from_start_index

return max(longest_seq_so_far)
```

this corresponds to the highlighted cells above

this keeps the max updated

We need the inner-most if statement to compute the max, by going through each possible_next_index and examining whether it will help us find the maximum possible subsequence starting with start_index. We could have equivalently written the code using list comprehensions:

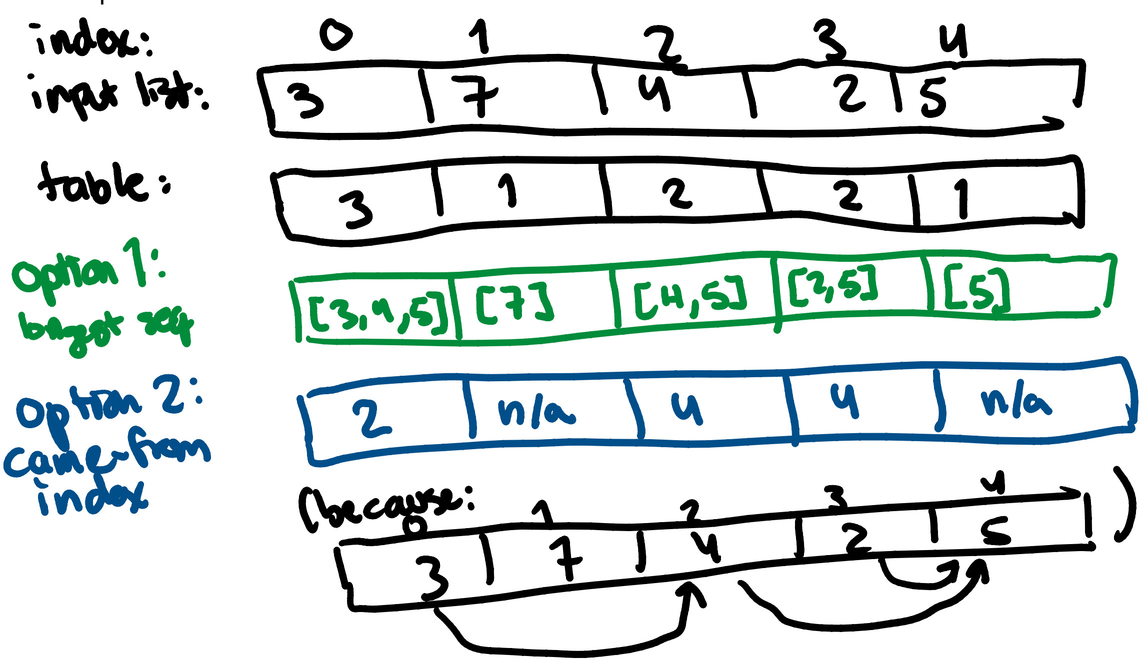
```
longest_seq_so_far = [0] * len(lst)
for start_index in range(len(lst) - 1, -1, -1):
    possible_next_indices = [next_index for next_index in range(start_index + 1, len(lst)) \
                             if lst[next_index] > lst[start_index]]
    len_of_next_index_sequences = [longest_seq_so_far[next_index] for next_index in possible_next_indices]
    longest_seq_so_far[start_index] = 1 + max(len_of_next_index_sequences, default = 0)

return max(longest_seq_so_far)
```

this corresponds to the highlighted cells

this gets the computed table vals at those indices

Instead of returning the length of the longest sequence, how would we return the sequence itself? There are two possible approaches we might take: keeping track of the sequence itself, *or* keeping track of the indices that got us to the sequence.



How would we insert this into our existing code?

```
longest_seq_so_far = [0] * len(lst)

for start_index in range(len(lst) - 1, -1, -1):
    len_of_max_seq_from_start_index = 1
    for possible_next_index in range(start_index + 1, len(lst)):
        if lst[possible_next_index] > lst[start_index]:
            if longest_seq_so_far[possible_next_index] + 1 > len_of_max_seq_from_start_index:
                len_of_max_seq_from_start_index = longest_seq_so_far[possible_next_index] + 1
    longest_seq_so_far[start_index] = len_of_max_seq_from_start_index

return max(longest_seq_so_far)
```

initialize array here (e.g. came_from = [-1] * len(lst))

update new value here: (e.g. came_from[start_index] = possible_next_index)

for came_from, loop back through, just like we did for graphs:
compute argmax (index that got us the max)
loop through came_from until we get to -1,
adding on the value at the list at each index