Better and Better

November 8, 2017

1. Consider this sequence of integers:

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

   (a) Write an increasing subsequence of length 4:

   (b) What is the length of the *longest* increasing subsequence?

2. Let's consider an incremental addition, somewhere in the middle of the algorithm:

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

   Can we add the 6?

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

   Can we add the 6?

   Should we add the 6?

3. Let's put all of those observations together into a recursive function that returns the length of the longest increasing subsequence in A, whose values are all greater than prev:

```
LISone( prev , A[1 .. n]):

    if (len(A) == 0):

        return ____

    else:

        if A[1] <= ________:

            return LISone(________, A[____ .. n])

        else:

            opt1 = LISone(________, A[____ .. n])

            opt2 = LISone(________, A[____ .. n]) + ____

            return ________ { opt1, opt2 }
```

4. Assume that the original array, `A`, is global to the algorithm. Adapt the pseudocode above into a function called `LIStwo` that changes the parameterization into a pair of indices, based on the following observations:

   (a) parameter `prev` is a value in the array, so you might as well just store its index, ___.

   (b) in any recursive execution of the function parameter `A[1 .. n]` is really just "the rest of the list", so let's parameterize it by a location in the global list, ___.

   (c) to give the algorithm a place to start, let the global `A = [____] + A`.

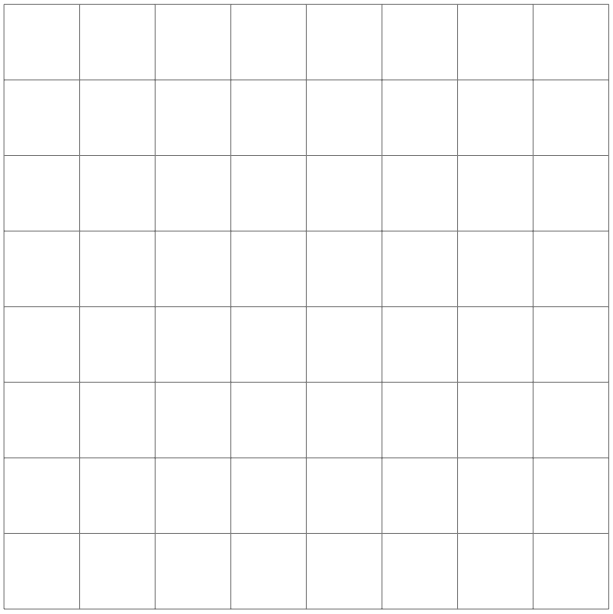
5. How should `LIStwo` be called, in order to solve the problem?

6. Does this problem lend itself to a memoized solution? Why? And what kind of structure would you use to keep track of previous computation?

7. Assume we initialize a 2d table `T[0..n, 1..n+1]` so that its values are `None` to indicate that no computation has been done. Adapt the algorithm above into `LISthree` to make use of this table.

```
LIStwo(i,j):

    if ( j > n):

        return 0

    else:

        if A[j] <= A[i]:

            return LIStwo( i , j+1 )

        else:

            opt1 = LIStwo( i , j+1 )

            opt2 = LIStwo( j , j+1 ) + 1

            return max { opt1, opt2 }
```

8. It's hard to tell exactly what order the code above fills in the data, and in fact, the order depends on the input values! In this part of the problem we design a principled approach to filling in the table, with an aim toward making the whole process iterative.



9. Based on the discussion above, fill in the pseudocode below for an iterative version of the solution:

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

    A[0] = ______

    for i = ____ to ____:

        T[ i, n+1 ] = 0

    for j = _________________:

        for i = _________________:

            if (A[ j ] <= A[ i ]):

                T[ i, j ] = T[ i, j+1 ]

            else:

                T[ i, j ] = max{ T[ i, j+1 ], T[ j, j+1 ]}

    return ____________
```

10. Analyze the final algorithm with respect to running time and space.

11. Can you solve this problem using less memory? If so, how?