Better and Better

December 6, 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:

   SOLUTION: An example is 1, 5, 6, 8. The point of this exercise was simply to prompt you to ask whether or not a subsequence had to consist of adjacent elements. The answer is no!

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

   JUST KIDDING!!! The point of today's exercise is to help you develop an algorithm to solve this very problem.

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?

   SOLUTION: In this part of the exercise, we assume that the algorithm has generated 3, 4, 9, as the incremental solution from the shaded region of the table, and we're now asking if we can add the 6 at this point in the algorithm. The answer is no, because 6 is less than the previous max value, 9. If our algorithm iterates from front to rear, building an optimal solution, then we have one of the criteria by which we can assess whether or not a new value should be added. In order to check, we need to keep track of that previous largest value.

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

   Can we add the 6?

   SOLUTION: This time, 6 is a feasible choice at this stage in the algorithm. The previous max value selected was 5, so we are not violating monotonicity if we select 6.

   Should we add the 6?

   SOLUTION: That is a question for the optimization portion of the algorithm to decide!!

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 _0_

    else:

        if A[1] <= _prev_:

            return LISone(_prev_, A[_2_ .. n])

        else:

            opt1 = LISone(_prev_, A[_2_ .. n])

            opt2 = LISone(_A[1]_, A[_2_ .. n]) + _1_

            return _max_ { 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, i.

   (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, j.

   (c) to give the algorithm a place to start, let the global `A = [ 0 ] + A`.

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

   SOLUTION: `LIStwo( 0, 1)`

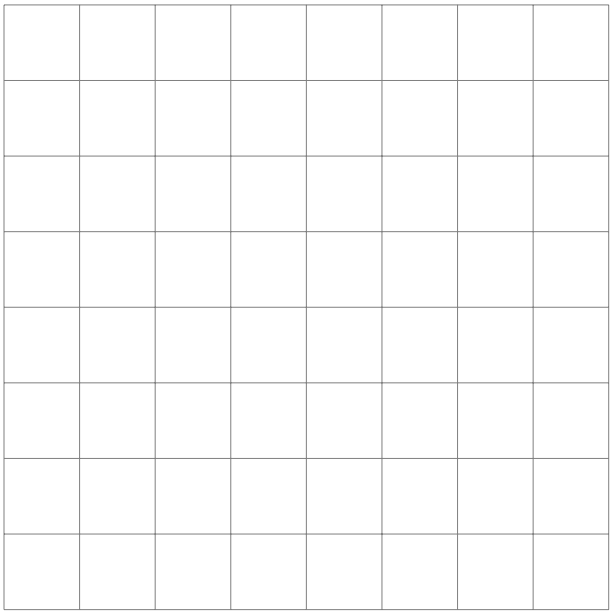
6. Does this problem lend itself to a memoized solution? Why? And what kind of structure would you use to keep track of previous computation?

   SOLUTION: Yes, we can implement a memoized solution by considering the parameters i, and j to be indices into a 2D array whose entries are the solutions to the optimal subproblems.

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 }

LISthree(i,j):

if T[i,j] == None:

    if ( j > n):

        T[i,j] = 0

    else:

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

            T[i,j] = LISthree( i , j+1 )

        else:

            opt1 = LISthree( i , j+1 )

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

            T[i,j] = max { opt1, opt2 }
return T[i,j]
```

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.



Suppose that the rows of the table represent the values `0 .. n` taken by parameter i, and that the columns represent the values `1 .. n+1` taken by parameter j. Next, consider entry i, j: it will depend upon entries $i, j+1$ or $j, j+1$, where we know that $i < j$. based on this observation, we first set the $n+1$st column to 0 (or $-\infty$ if we're allowing negative numbers). Then we iterate by columns from right to left, top to bottom.

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

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

    A[0] = _0_

    for i = _0 to n_:

        T[ i, n+1 ] = 0

    for j = _n down to 0_:

        for i = _0 to n _:

            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 ] + 1}

    return _T[0, 1]_
```

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

    SOLUTION: The algorithm requires $\Theta(n^2)$ space and $\Theta(n^2)$ time, since all operations inside the nested `for` loops are constant time.

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

    SOLUTION: Notice that each column is only dependent on the column before it. The problem can be solved by essentially maintaining column buffer consisting of the immediately previous column. When a column is filled, it is moved to the buffer and used to fill the next column.