```
#
# Longest Increasing Subsequence (LIS)
#
# Given a sequence A of numbers, return the maximal length of any
# subsequence of A consisting of strictly increasing numbers.
#
# We develop a dynamic-programming solution: we model the problem as a
# choice between a series of sub-problem defined as follows: let
# DP(A,i) denote the maximal length of any sub-sequence ending at
# position i.  Then, DP(A,i) = min { DP(A,j) + 1 } over all positions
# j < i such that A[j] < A[i] (since A[i] *extends* the sequence
# ending at position j).  With DP(A,i), we can compute the maximal
# overall: LIS(A) = min { DP(A,i) } for all i = 0,...,n-1
#
def LIS (A):
    # return the length of the longest increasing subsequence in A
    l = 1
    for i in range(len(A)):
        l = max(l, DP(A,i))
    return l

def DP (A,i):
    # return the length of the longest increasing subsequence in A
    # that ends at position i
    l = 1
    for j in range(i):
        if A[j] < A[i]:
            l = max(l, DP(A,j) + 1)
    return l

#
# The above solution is a straightforward, recursive implementation of
# the dynamic-programming solution.  However, this solution is
# inefficient.  One way to make it efficient is to develop it in a
# simple sequence.  Below is the same dynamic-programming solution
# written as an iterative algorithm.
#
def LIS_itr (A):
    n = len(A)
    assert n > 0
    DP = [1]*n
    best = 1                    # best length seen so far
    for i in range(1,n):
        for j in range(i):
            if A[j] < A[i]:
                DP[i] = max(DP[i], DP[j] + 1)
        best = max(best, DP[i])
    return best

#
# The solutions seen so far compute the maximal *length* but do not
# output a maximal subsequence.  We can easily modify the iterative
# solution above to do just that.
#
def LIS_itr_seq (A):
    n = len(A)
    assert n > 0
    DP = [1]*n
    P = [None]*n                # "previous" elements in the maximal
                                # sequence ending at position i
    best_i = 0                  #
    for i in range(1,n):
        for j in range(i):
            if A[j] < A[i] and DP[j] + 1 > DP[i]:
                DP[i] = DP[j] + 1
                P[i] = j
```

```
    if DP[i] > DP[best_i]:
        best_i = i
S = []                      # S will contain the maximal sequence
i = best_i
while i != None:            # We build the maximal sequence
    S.append(A[i])          # backwards, following the "previous"
    i = P[i]                # chain, starting from best_i
i = 0
j = len(S) - 1
while i < j:                # Then we simply reverse S
    S[i], S[j] = S[j], S[i]
    i = i + 1
    j = j - 1
return S
```