

Longest Increasing Subsequence

Input: Array $A[1 \dots n]$ of integers.

Goal: Find the length of the longest increasing subsequence. Specifically, find the length k of the longest string of indices $1 \leq i_1 < \dots < i_k \leq n$ such that for all $1 \leq j < k$, $A[i_j] < A[i_{j+1}]$

As this is a subsequence problem, consider the first element of A . The longest increasing subsequence of A is either,

- the longest increasing subsequence of $A[2 \dots n]$ or
- $A[1]$ followed by the longest increasing subsequence of $A[2 \dots n]$ such that all elements are $> A[1]$.

To make this fully recursive we augment A s.t. $A[0] = -\infty$. Thus every subproblem can be described as being required to be larger than some previous element (even the initial problem where we have the trivially satisfied $> -\infty$ requirement)

Globally define $A[1 \dots n]$, and augment it s.t. $A[0] = -\infty$. Assume $0 \leq prev < start$.

```
1: procedure LIS(prev, start)
2:   if start > n then
3:     return 0
4:   ignore = LIS(prev, start + 1)
5:   best = ignore
6:   if  $A[start] > A[prev]$  then
7:     include = 1 + LIS(start, start + 1)
8:     if include > ignore then
9:       best = include
10:  return best
```

Claim: $LIS(prev, start)$, for $prev < start$, returns the longest increasing subsequence in $A[start \dots n]$ s.t. all elements are greater than $A[prev]$.

Proof: If $start > n$, there are no elements left in the remaining part of A , and so the algorithm correctly returns 0. Otherwise $LIS(prev, start)$ either includes $A[start]$ or not.

- if not, then $LIS(prev, start) = LIS(prev, start + 1)$.
- if so, then it must be that $A[start] > A[prev]$, and all remaining element of the LIS that come after $A[start]$ must be great than $A[start]$. Therefore, $LIS(prev, start) = LIS(start, start + 1) + 1$, where the +1 counts $A[start]$.

If $A[start] \leq A[prev]$ the solution must be $LIS(prev, start + 1)$, which is what the algorithm returns, i.e. in this case the if statement is not executed. If $A[start] > A[prev]$ the solution is either $LIS(prev, start + 1)$ or $LIS(start, start + 1)$, whichever is bigger. Since we

don't know which is bigger, our algorithm tries both and takes the max. Note in both case the problem is reduced to a subproblem on a strictly smaller array (i.e. $A[start + 1 \dots n]$), and so can be assumed to be correctly handled by induction (where the base case is handled by the initial $start > n$ conditional). \square

To compute the LIS of $A[1 \dots n]$ we call $LIS(0, 1)$ as this is the LIS in $A[1 \dots n]$ s.t. all elements $> -\infty$, which is the same as the LIS of $A[1 \dots n]$.

Applying DP: $LIS(prev, start)$ depends on two parameters, each ranging over $O(n)$ values, as they are both indices into $A[0 \dots n]$. Hence the above recursive algorithm can be turned into a DP algorithm using a 2D array, of total size $O(n^2)$. Note that $LIS(prev, start)$ only depends on $LIS(prev, start + 1)$ and $LIS(start, start + 1)$, both of which have a strictly large value of the second parameter. Therefore this table can be filled in any order such that all $LIS(\cdot, start + 1)$ values are computed before any $LIS(\cdot, start)$ value. Namely with a decreasing for loop for the second parameter, and a second inner loop going over all values of the first parameter (in any order). Ignoring the time for computing recursive calls, the above algorithm runs in $O(1)$ time. Therefore, if processed in the right order, each table entry takes $O(1)$ time to compute and so the total running time is $O(n^2)$.

```

1: procedure LISDP( $A[1 \dots n]$ )
2:    $A[0] = -\infty$ 
3:   Define  $B[0 \dots n][1 \dots n + 1]$ 
4:   for  $i = 0$  to  $n$  do
5:      $B[i][n + 1] = 0$ 
6:   for  $start = n$  to  $1$  do
7:     for  $prev = start - 1$  to  $0$  do
8:        $ignore = B[prev][start + 1]$ 
9:        $best = ignore$ 
10:      if  $A[start] > A[prev]$  then
11:         $include = 1 + B[start][start + 1]$ 
12:        if  $include > ignore$  then
13:           $best = include$ 
14:         $B[prev][start] = best$ 
15:   return  $B[0][1]$ 

```

Longest Common Subsequence

Input: Character arrays $A[1 \dots n]$ and $B[1 \dots m]$.

Goal: Find the length of the longest common subsequence. Specifically, find the length k of the longest strings of indices $1 \leq i_1 < \dots < i_k \leq n$ and $1 \leq j_1 < \dots < j_k \leq m$ such that for all $1 \leq l \leq k$, $A[i_l] = B[j_l]$

As this is a subsequence problem, similar to LIS, the focus is to figure out how to handle the very first element of A and B. We have the following.

- If A or B is empty, return 0.
- If $A[1] \neq B[1]$ then $A[1]$ and $B[1]$ cannot both be used, so it should be the best of either throwing out $A[1]$ or $B[1]$, i.e.

$$LCS(A[1 \dots n], B[1 \dots m]) = \max\{LCS(A[2 \dots n], B[1 \dots m]), LCS(A[1 \dots n], B[2 \dots m])\}.$$
- If $A[1] = B[1]$ then we can either match or throw out so $LCS(A[1 \dots n], B[1 \dots m]) = \max\{1 + LCS(A[2 \dots n], B[2 \dots m]), LCS(A[2 \dots n], B[1 \dots m]), LCS(A[1 \dots n], B[2 \dots m])\}.$

Note if $A[1] = B[1]$ then one can prove $LCS(A[1 \dots n], B[1 \dots m]) = 1 + LCS(A[2 \dots n], B[2 \dots m])$. While this may seem obvious, unless it is proven you cannot assume it, so we won't.

Now to turn this into a recursive algorithm, define $LCS(curA, curB)$ to be the longest common subsequence of $A[curA \dots n]$ and $B[curB \dots m]$ (i.e. $LCS(A[curA \dots n], B[curB \dots m])$). Based on the above observations we have the following.

Again assume $A[1 \dots n]$ and $B[1 \dots m]$ are defined globally, and $curA, curB > 0$.

```

1: procedure LCS( $curA, curB$ )
2:   if  $curA > n$  or  $curB > m$  then
3:     return 0
4:    $ignore = \max\{LCS(curA + 1, curB), LCS(curA, curB + 1)\}$ 
5:    $best = ignore$ 
6:   if  $A[curA] = B[curB]$  then
7:      $include = 1 + LCS(curA + 1, curB + 1)$ 
8:     if  $include > ignore$  then
9:        $best = include$ 
10:  return  $best$ 

```

To find the longest common subsequence of $A[1 \dots n]$ and $B[1 \dots m]$ we then call $LCS(1, 1)$.

The correctness follows immediately from the above (arguing the same way as for LIS).

Applying DP. $LCS(curA, curB)$ depends on two parameters, the first ranging over $O(n)$ values and the second over $O(m)$ values, since they are indices into $A[1 \dots n]$ and $B[1 \dots m]$, respectively. Hence the above recursive algorithm can be turned into a DP algorithm using a 2D array, of total size $O(mn)$. $LCS(curA, curB)$ makes at most three recursive call to $LCS(curA + 1, curB)$, $LCS(curA, curB + 1)$, and $LCS(curA + 1, curB + 1)$. In each case

at least one of the two parameters increases and the other does not decrease. Therefore, the 2D array can be filled in using a pair of nested for loops, the outer one ranging over the first parameter and starting at n and going down to 1, and the inner one ranging over the second parameter and starting at m and going down to 1. Ignoring the time for computing recursive calls, the above algorithm runs in $O(1)$ time. Therefore, if processed in the right order, each table entry takes $O(1)$ time to compute and so the total running time is $O(mn)$.

```

1: procedure LCSDP( $A[1 \dots n], B[1 \dots m]$ )
2:   Define  $C[1 \dots n + 1][1 \dots m + 1]$ 
3:   for  $i = 1$  to  $n + 1$  do
4:      $C[i][m + 1] = 0$ 
5:   for  $i = 1$  to  $m + 1$  do
6:      $C[n + 1][i] = 0$ 
7:   for  $curA = n$  to 1 do
8:     for  $curB = m$  to 1 do
9:        $ignore = \max\{C[curA + 1][curB], C[curA][curB + 1]\}$ 
10:       $best = ignore$ 
11:      if  $A[curA] = B[curB]$  then
12:         $include = 1 + C[curA + 1][curB + 1]$ 
13:        if  $include > ignore$  then
14:           $best = include$ 
15:         $C[curA][curB] = best$ 
16:   return  $C[1][1]$ 

```
