20.2 The Longest Upsequence

Consider a sequence of values $(v_0, \cdots, v_{n-1})$. If one deletes i (not necessarily adjacent) values from the list, one has a subsequence of length $n-i$. This subsequence is called an *upseqence* if its values are in non-decreasing order. For example, the list (1,3,4,6,2,4) has a subsequence (1,3,2), which is not an upsequence, and another subsequence (1,3,6), which is an upsequence.

We want to write a program that, given a sequence in $b[0:n-1]$, where $n>0$, calculates the length of the longest upsequence of $b[0:n-1]$. As an abbreviation, use the notation $lup(s)$ to mean:

$$lup(s) = \text{the length of the longest upsequence of sequence } s$$

Thus, using a variable k to contain the answer, the program has the pre- and postconditions:

$$Q: n>0$$
$$R: k = lup(b[0:n-1])$$

Note that a change in any one value of a sequence could change its longest upsequence, and this means that possibly every value of a sequence s must be interrogated to determine $lup(s)$. This suggests a loop. Begin by writing a possible invariant and an outline of the loop.

The loop will interrogate the values of $b[0:n-1]$ in some order. Since $lup(b[0:0])$ is 1, a possible invariant can be derived by replacing the constant n of R by a variable:

$$P: 1 \leqslant i \leqslant n \wedge k = lup(b[0:i-1])$$

The loop itself will have the form

```
i, k:= 1, 1;
do i ≠ n → increase i, maintaining P od
```

Increasing i extends the sequence $b[0:i-1]$ for which k is the length of a longest upsequence, and hence may call for an increase in k. Whether k is to be increased depends on whether $b[i]$ is at least as large as a value that ends a longest upsequence of $b[0:i-1]$ (there may be more than one longest upsequence). It makes sense to maintain information in other variables so that such a test can be efficiently made. What is the minimum information needed to ascertain whether k should be increased?

The *smallest* value m (say) that ends an upsequence of length k of $b[0:i-1]$ must be known, for then $b[0:i]$ has an upsequence of length $k+1$ *iff* $b[i] \geqslant m$. Therefore, we revise invariant P to include m:

$P: 1 \leqslant i \leqslant n \wedge k = lup(b[0:i-1]) \wedge$
m is the smallest value in $b[0:i-1]$ that ends an upsequence of length k

In the case $b[i] \geqslant m$, k can be increased and m set to $b[i]$, so that the program thus far looks like

```
i, k, m:= 1, 1, b[0]; {P}
do i ≠ n → if b[i]≥m → k, m:= k+1, b[i]
            [] b[i]<m → ?
            fi;
            i:= i+1
od
```

The question now becomes what to do if $b[i]<m$. Variable k should not be changed, but what about m? Under what condition must m be changed?

If $b[0:i-1]$ contains an upsequence of length $k-1$ that ends in a value $\leqslant b[i]$, then $b[i]$ ends an upsequence of length k of $b[0:i]$. If, in addition, $b[i]<m$, then m must be changed. In order to check this condition, consider maintaining the minimum value $m1$ that ends an upsequence of length $k-1$ of $b[0:i-1]$.

This means that two values are needed: the minimum value m that ends an upsequence of length k and the minimum value $m1$ that ends an upsequence of length $k-1$. Judging by the development thus far, can you generalize this?

Maintaining m caused us to introduce $m1$; maintaining $m1$ will cause us to introduce $m2$ to contain the minimum value that ends an upsequence of length $k-2$. And so on. Therefore, an array of values is needed. We modify the invariant once more:

(20.2.1) $P: 1 \leqslant i \leqslant n \wedge k = lup(b[0:i-1]) \wedge$
$(A j: 1 \leqslant j \leqslant k: m[j]$ is the smallest value that ends an upsequence of length j of $b[0:i-1])$

And the program is changed to

```
i, k, m[1]:= 1, 1, b[0]; {P}
do i≠n → if b[i]≥m[k] → k:= k+1; m[k]:= b[i]
         ▯ b[i]<m[k] → ?
         fi;
         i:= i+1
od
```

Before proceeding further, it makes sense to investigate array m; does it have any properties that might be useful?

Array m is ordered, because the minimum value that ends an upsequence of length j (say) must be at most the minimum value that ends an upsequence of length $j+1$.

We are now faced with determining which values of $m[1:k]$ must be changed in case $b[i]<m[k]$. Solve this problem.

The case $b[i]<m[1]$ is the easiest to handle. Since $m[1]$ is the smallest value that ends an upsequence of length 1 of $b[0:i-1]$, if $b[i]<m[1]$, then $b[i]$ is the smallest value in $b[0:i]$ and it should become the new $m[1]$. No other value of m need be changed, since all upsequences of $b[0:i-1]$ end in a value larger than $b[i]$.

Finally, consider the case $m[1]\leqslant b[i]<m[k]$. Which values of m should be changed? Clearly, only those greater than $b[i]$ can be changed, since they represent minimum values. So suppose we find the j satisfying

$$m[j-1]\leqslant b[i]<m[j]$$

Then $m[1:j-1]$ should not be changed. Next, since $m[j-1]$ ends an upsequence of length $j-1$ of $b[0:i-1]$, $b[i]$ ends an upsequence of length j of $b[0:i]$. Hence, $m[j]$ should be changed to $b[i]$. Finally, $m[j+1:k]$ should not be changed (why?).

Binary search (exercise 4 of section 16.3) can be used to locate j. The final program is given in (20.2.2).

The execution time of program (20.2.2) is proportional to $(n \log n)$ in the worst case and to n in the best. It requires space proportional to n in the worst case, for array m. It uses a technique called "dynamic programming", although it was developed without conscious knowledge of that technique.


```
(20.2.2) i, k, m[1]:= 1, 1, b[0]; {P}
         {inv: (20.2.1); bound: n-i}
         do i≠n → if b[i] ≥m[k]          → k:= k+1; m[k]:= b[i]
                  ▯ b[i] <m[1]           → m[1]:= b[i]
                  ▯ m[1]≤b[i]<m[k] →
                    Establish m[j-1]≤b[i]<m[j]:
                      h, j:= 1, k;
                      {inv: 1≤h<j≤k ∧ m[h]≤b[i]<m[j]}
                      {bound: j-h-1}
                      do h≠j-1 → e:= (h+j)÷2;
                                 if m[e]≤b[i] → h:= e
                                 ▯ m[e]>b[i] → j:= e
                                 fi
                      od;
                    m[j]:= b[i]
                  fi;
                  i:= i+1
         od
```

Exercises for Chapter 20

1. (Unique 5-bit Sequences). Consider sequences of 36 bits. Each such sequence has 32 5-bit sequences consisting of *adjacent* bits. For example, the sequence 1101011... contains the 5-bit sequences 11010, 10101, 01011, ... Write a program that prints all 36-bit sequences with the two properties

(1) The first 5 bits of the sequence are 00000.
(2) No two 5-bit subsequences are the same.

2. (The Next Higher Permutation). Suppose array $b[0:n-1]$ contains a sequence of (not necessarily different) digits, e.g. $n=6$ and $b[0:5]=(2,4,3,6,2,1)$. Consider this sequence as the integer 243621. For any such sequence (except for the one whose digits are in decreasing order) there exists a permutation of the digits that yields the next higher integer (using the same digits). For the example, it is $(2,4,6,1,2,3)$, which represents the integer 246123.

Write a program that, given an array $b[0:n-1]$ that has a next higher permutation, changes b into that next higher permutation.

3. (Different Adjacent Subsequences). Consider sequences of 1's, 2's and 3's. Call a sequence *good* if no two adjacent non-empty subsequences of it are the same. For example, the following sequences are *good*:

2
32
32123
1232123