Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Department of Computer Science
Markus Püschel
Peter Widmayer
Thomas Tschager
Tobias Pröger

10th November 2016

# Data Structures & Algorithm — Solutions to Sheet 7 — AS 16

**Lösung 7.1** *Enumerating Palindromes.*

a) *Definition of the DP table:* We use an $n \times n$ table $T$ with entries that are 0 or 1. For $1 \leq i \leq j \leq n$ let $T[i,j] = 1$ if and only if $\langle A[i], \ldots, A[j] \rangle$ is a palindrome.

*Computation of an entry:* We distinguish three cases.

1. *Case:* $1 \leq i = j \leq n$. $A[i]$ is a palindrome of length 1, thus we set $T[i,i] = 1$ for every $i$, $1 \leq i \leq n$.
2. *Case:* $1 \leq i \leq n$, $j = i+1 \leq n$. Then we consider palindromes of length 2, and we set $T[i,i+1] = 1$ if and only if $A[i] = A[i+1]$.
3. *Case:* $1 \leq i \leq n$, $i+1 < j \leq n$. Let $\langle A[i], \ldots, A[j] \rangle$ be the considered sequence. By definition it is a palindrome iff $A[i] = A[j]$ and additionally, $\langle A[i+1], \ldots, A[j-1] \rangle$ is a palindrome. Thus we set $T[i,j] = 1$ if and only if $A[i] = A[j]$ and $T[i+1, j-1] = 1$.

*Calculation order:* We compute the entries $T[i,j]$ in increasing difference of $j - i$, i.e. we start with $T[i,i]$ for $1 \leq i \leq n$. After that we continue to compute $T[i,i+1]$ for $1 \leq i \leq n-1$, after that $T[i,i+2]$ for $1 \leq i \leq n-2$, and so on. Finally we compute $T[1,n]$.

*Extracting the solution:* We iterate over all entries $T[i,j]$, $1 \leq i \leq j \leq n$, and output $(i,j)$ if $T[i,j] = 1$.

*Running time:* The table has $n^2$ entries. Each of these entries can be computed in time $\mathcal{O}(1)$. To extract the solution every entry of the table is considered again, and this also only costs $\mathcal{O}(1)$ per entry. Thus the overall running time is $\mathcal{O}(n^2)$.

b) We consider the entries of the table in reverse calculation order, i.e., we start with $T[1,n]$. After that we consider $T[1,n-1]$ and $T[2,n]$, and so on. As soon as we find some entry $T[i,j]$ having value 1, the procedure terminates and returns $\langle A[i], \ldots, A[j] \rangle$.

As before, the consideration of each entry only takes time $\mathcal{O}(1)$. Since there are $n^2$ entries and the palindrome has at most length $n$, the overall running time still is $\mathcal{O}(n^2)$.

**Solution 7.2** *Ascending Sequences.*

There are two possible solutions. We start with the more direct one and adapt it later:

**Solution 1:**

*Definition of the DP table:* We define a table $T$ of size $n \times m$. The entry $T[x][y]$ contains the length of the longest ascending sequence $S_{x,y}$ that ends in $A[x][y]$. Also, $T[x][y]$ contains the coordinates of the predecessor of $(x, y)$ in $S_{x,y}$ if it exists.

*Computation of an entry:* The sequence $S_{x,y}$ (and thus the entry at position $T[x][y]$) can be calculated from the sequences $S_{x-1,y}, S_{x+1,y}, S_{x,y-1}, S_{x,y+1}$, as far as these exist. To do this, we take the longest sequence belonging to a neighbor with smaller value than $A[x][y]$ and simply append $(x, y)$ to this sequence.

*Calculation order:* For each entry, we only need to know the entries for smaller values in the array. We can thus calculate the entries in ascending order according to their value in the array.

*Extracting the solution:* To find the solution, we have to look at all entries and locate the longest sequence. From there, we can reconstruct the solution by following the corresponding predecessors (stored in $T[x][y]$).

*Running time:* In overall, we fill $n \cdot m$ entries, and for each we have to consider four neighbors. However, we first need to sort the elements in ascending order. To find the solution, we must once again look at each entry and then reconstruct the sequence – both need $\mathcal{O}(nm)$ steps. The running time is thus dominated by the sorting and is $\mathcal{O}(nm \log(nm))$.

**Solution 2:** We use the above dynamic program, but we modify the calculation order so that we can avoid the sorting.

*Calculation order:* Instead of sorting the values, we go through the array in any order. If we come across an entry that was already calculated (how this can happen will become clear), we skip it. Otherwise we need the entries corresponding to smaller neighbors. If these are already known, we are lucky and we can fill our entry as before. Otherwise, we recursively determine the entries of the neighbors first. In this way, we start a sort of depth-first search, filling the deepest entries in the search first.

*Running time:* The process can be seen as a mixture of dynamic programming and memoization. It is important that we do not have to sort. To be efficient, we need to be sure that we do not visit entries too often. For each entry we start a depth-first search once. This means that each entry can be visited at most 4 times during a depth-first search, since we start exactly one depth-first search at each neighbor. Overall, the repeated depth-first search requires $\mathcal{O}(nm)$ steps. The total running time is therefore also $\mathcal{O}(nm)$, which is linear in the input size.