Solution 7.1  Mars mission.

Definition of the DP table: We use an $m \times n$-table $T$, and $T[i, j]$ contains the maximum achievable value of the rock samples on a south-east path from $(1, 1)$ to $(i, j)$.

Computation of an entry: For $(i, j)$ with $1 < i, j \leq n$ we observe the following: if a path ends at the position $(x, y)$, then we got there either from above or from the left. Thus, the maximum value we can achieve at that position is exactly

$$T[i, j] = A[i, j] + \max\{T[i - 1, j], T[i, j - 1]\},$$

since the value $A[i, j]$ of the rock sample at the position $(i, j)$ must be added to the maximum value of the samples collected on the way to this position. For the cases at the top and at the left we define:

- $T[1, 1] = A[1, 1] = 0$, because $(1, 1)$ is the starting position,
- $T[i, 1] = A[i, 1] + T[i - 1, 1]$ for $i > 1$, because the position $(i, 1)$ can only be reached from above from the position $(i - 1, 1)$,
- $T[1, j] = A[1, j] + T[1, j - 1]$ for $j > 1$, because the position $(1, j)$ can only be reached from the left from the position $(1, j - 1)$.

Calculation order: The entry $T[i, j]$ depends only on entries for smaller values of $i$ and $j$. Therefore, the entries can be calculated, for example, increasingly in the value of $i = 1, \ldots, m$ and for the same $i$, increasingly in the value of $j = 1, \ldots, n$.

Extracting the solution: In the end, the maximum achievable value of the rock samples is stored in the entry $T[m, n]$.

Reconstruction of the path: We start in the entry $T[m, n]$ and output $(m, n)$. Next, we check if $T[m, n] = A[m, n] + T[m - 1, n]$. If this is the case, then we came from above, i.e., from the entry $(m - 1, n)$, and we continue there. Otherwise, $T[m, n] = A[m, n] + T[m, n - 1]$, and we came from the left, so we proceed with the entry $(m, n - 1)$. The reconstruction continues until the starting position $(1, 1)$ is reached. In the end, we have determined the positions of the path in reverse order.

Running time: The table has size $m \cdot n$ and each entry can be computed in time $O(1)$. Thus, the computation of the maximum value can be done in time $O(mn)$.

The reconstructed path has length $m + n - 1$. Since for each position we can decide in time $O(1)$ whether we came from the top or from the left, the whole path can be determined in time $O(m + n)$. Together with the time needed to fill the table, we get an overall running time of $O(mn)$. 

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 $O(nm)$ steps. The running time is thus dominated by the sorting and is $O(nm \log(nm))$.

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 $O(nm)$ steps. The total running time is therefore also $O(nm)$, which is linear in the input size.