Solution 7.1  Placing Wind Turbines.

a) Such an example can already be constructed for \( n = 3 \) wind turbines. For example, we define the possible positions to be \( d_1 = 0 \), \( d_2 = 1 \), \( d_3 = 2 \), and the minimum distance to be \( D = 2 \). Furthermore, we set the energy production values to be \( e_1 = 2 \), \( e_2 = 3 \) and \( e_3 = 2 \). The greedy strategy described above places a wind turbine on position 2. Since the distance to the positions 1 and 3 is less than \( D \), no more wind turbines can be placed and the energy gain is exactly 3 units. However, it would be optimal to place the wind turbines on the positions 1 and 3 with an energy gain of \( 2 + 2 = 4 \) units.

b) Definition of the DP table: We use a one-dimensional (!) table \( T \) with \( n \) entries. The entry \( T[k] \) describes the maximum producible energy when the wind turbines can be only placed on the positions 1, ..., \( k \).

Computation of an entry: If \( d_k < D \) for a position \( k \in \{1, ..., n\} \), then the distance between the \( k \)-th position and the first possible position is smaller than the allowed minimum distance, and we can select at most one of the first \( k \) positions. In this case, the maximum energy is produced when we choose the position from \( \{1, ..., k\} \) that yields maximum energy. We set

\[
T[1] := e_1 \quad \text{and} \quad T[k] := \max\{T[k-1], e_k\} \quad \text{for every} \quad k > 1 \quad \text{with} \quad d_k < D, \quad (1)
\]

and with this definition, we have \( T[k] = \max\{e_1, ..., e_k\} \) for each \( k \) where \( d_k < D \).

In the case \( d_k \geq D \), we have two possibilities:

- A wind turbine is placed on position \( k \). If we set

\[
v_k := \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\}, \quad (2)
\]

then \( v_k \) is the index of the position on the left of \( k \) which is closest to \( k \), and that has a distance of at least \( D \). Thus, on every position \( k' \) with \( v_k < k' < k \), no wind turbines can be placed because the minimum distance to \( k \) is not respected.

The maximum producible energy for the positions 1, ..., \( k \) now is the sum of \( e_k \) (energy yield of the position \( k \)) and the maximum producible energy if only the positions from \( \{1, ..., v_k\} \) are used.

- No wind turbine is placed at the position \( k \). In this case, the maximum producible energy using the positions 1, ..., \( k \) is the same as the maximum producible energy using the positions 1, ..., \( k - 1 \).

Thus, for every \( k \) with \( d_k \geq D \) we set

\[
v_k := \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\}, \quad (3)
\]

\[
T[k] := \max\{T[k-1], e_k + T[v_k]\}. \quad (4)
\]
**Calculation order:** Since a table entry depends only on the entries with smaller index, the entries in $T[k]$ can be calculated gradually for ascending $k$. Similarly, the values for $v_k$ can be calculated gradually for ascending $k$.

**Extracting the solution:** At the end, the solution is stored in the entry $T[n]$.

c) A careful implementation of the above process requires only time $\Theta(n)$ if $v_k$ is not calculated naively in each step. It is clear that $v_k$ needs to be calculated only for $k$ with $d_k \geq D$. For the first such $k$, $v_k$ can be calculated as in the above formula. For all other $k$ we know that $v_{k-1}$ has already been calculated, and we have

$$v_k = \max\{i \in \{1, \ldots, k-1\} : d_i \leq d_k - D\}$$
$$= \max\{i \in \{v_{k-1}, \ldots, k-1\} : d_i \leq d_k - D\}. \quad (5)$$

Therefore, to calculate $v_k$ from $v_{k-1}$, we can initially set $v_k = v_{k-1}$ and then repeatedly increase this index by 1 until $d_{v_k+1} > d_k - D$. In this manner, we can calculate all $v_k$ in time $\Theta(n)$. Since the running time to compute a single item $T[k]$ is constant, the total running time is $\Theta(n)$.

d) We first fill in the whole table and store all the values of $v_k$. The optimum positioning of wind turbines can now be achieved by tracing the values in the table as follows. Initially, we set $k = n$. Then we repeat the following steps as long as $k \geq 1$.

- If $d_k \geq D$ and $e_k + T[v_k] \geq T[k-1]$, then output $k$ and set $k := v_k$.
- If $d_k \geq D$ and $e_k + T[v_k] < T[k-1]$, then more energy is produced if the position $k$ is not used for a wind turbine. In this case, do not output anything and set $k := k - 1$.
- If $d_k < D$, then only one wind turbine can be placed on the positions $1, \ldots, k$. In this case, output the position $i \in \{1, \ldots, k\}$ having maximum energy gain $e_i$, and terminate the algorithm.

The running time of this process again is $\Theta(n)$.

**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 $m \times n$. 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 left 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 $A[x][y]$. We 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. We can then retrace it to reconstruct the solution.

Overall, we fill $m \cdot n$ 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 needs $O(nm)$ 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.

The process can be seen as a mix 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.