Exercise 7.1  Numerical Puzzle.

a) Definition of the DP Table: we define a table $S$ of size $(n+1) \times (\sigma + 1)$. The entry $S[n'][\sigma']$ is 'true' if if the sum $\sigma'$ can be obtained using the first $n'$ digits, otherwise is 'false'.

Computation of an Entry: We set $S[0][0] \leftarrow \text{'true'},$ and for all $0 < \sigma' \leq \sigma$ we set $S[0][\sigma'] \leftarrow \text{'false'}$. To compute the entry $S[n'][\sigma']$, we iterate over all $i = 1, 2, \ldots, n'$ and calculate the number $z$ composed of the last $i$ digits. If $z > \sigma'$ we stop, otherwise we check the entry $S[n' - i][\sigma' - z]$. This entry indicates whether the number $\sigma' - z$ can be formed using the first $n' - i$ digits. If so, we can form $\sigma'$ by adding the last $i$ digits. We then set $S[n'][\sigma'] \leftarrow \text{'true'}$ and stop the iteration. Otherwise, we continue with the next choice for $i$. If there are no choices left for $i$, we set $S[n'][\sigma'] \leftarrow \text{'false'},$ since it is not possible to form the sum.

Calculation order: Every entry $S[n'][\sigma']$ depends only on entries for smaller values of $n'$ and $\sigma'$. We can therefore sort the entries, for example, by increasing values of $n'$, and for the same $n'$ by increasing values of $\sigma'$.

Extracting the solution: The solution is simply given by the entry $S[n][\sigma]$.

Code sample:

```cpp
// returns whether the digits can be decomposed into numbers summing up to sigma
bool admitsSum(const vector<int>& digits, int sigma)
{
    int n = (int)digits.size();

    // create 2D array of size n*sigma filled with 'false' everywhere except S[0][0]
    vector<vector<bool>> S(n+1, vector<bool>(sigma+1, false));
    S[0][0] = true;

    for(int n_ = 1; n_ <= n; ++n_)
        for(int sigma_ = 1; sigma_ <= sigma; ++sigma_)
        {
            int z = 0, power = 1;

            for(int i = 1; i <= n_; ++i)
            {
                z += power * digits[n_ - i];
                power *= 10;

                if(z > sigma_)
                    break;

                if(S[n_ - i][sigma_ - z])
                {
                    S[n_][sigma_] = true;
                    break;
                }
            }
        }

    return S[n][sigma];
}
```
b) The number of entries is $O(n\sigma)$. For every entry $S[n'][\sigma']$ we iterate over the values $i = 1, 2, \ldots, n'$. We also need to compute the number formed by the last $i$ digits. We can derive this number in constant time from the number that was formed using the last $i - 1$ digits (or infer it directly if $i = 1$). Alternatively, we can calculate these values in advance for every possible combination of $n'$ and $i$.

We have an overall running time of $O(n^2\sigma)$. This means that the running time depends on $\sigma$ and is not polynomial, but only pseudo-polynomial. The solution is polynomial if it is known in advance that $\sigma$ is bounded (that is, $\sigma = O(n^c)$ for a constant $c$).

c) We proceed exactly as in the calculation of the entries. Instead of simply looking whether an entry is 'true', we recursively determine all combinations that yield the corresponding number and append the number formed by the last $i$ digits to each of them. Also, we do not stop once we encounter a 'true', but we try all combinations.

Code sample:

```cpp
// returns a vector of possible decompositions of the digits to sum up to sigma
vector<vector<int>> enumResults(const vector<vector<bool>>& S,
                                 const vector<int>& digits,
                                 int n, int sigma)
{
    vector<vector<int>> ret;
    if(n == 0 && sigma == 0)
        ret.push_back(vector<int>(0));
    int z = 0, power = 1;
    for(int i = 1; i <= n; ++i)
    {
        z += power*digits[n - i];
        power *= 10;
        if(z > sigma)
            break;
        if(S[n - i][sigma - z])
        {
            vector<vector<int>> prefixes = enumResults(S, digits, n - i, sigma - z);
            for(int j = 0; j < (int)prefixes.size(); ++j)
            {
                prefixes[j].push_back(z);
                ret.push_back(prefixes[j]);
            }
        }
    }
    return ret;
}
```
Exercise 7.2 Ascending Sequences (from an exam in February 2010).

a) For this part, we consider each row and each column separately. For each row/column, we determine the longest increasing (and contiguous) sub-sequence in linear time in both directions. To do this, we iterate over each row/column once, and remember the longest ascending suffix (as in programming exercise 2, where we maintain the maximum suffix sum). The overall running time is $O(n \cdot 2m + m \cdot 2n) = O(nm)$, since there are $n$ rows and $m$ columns, and we go through each one linearly twice (i.e., in $m$ resp. $n$ steps).

b) 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 4 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 $O(nm \log(nm))$.

**Code sample:**

```cpp
// for sorting
struct Field {
    int x, y, value;
    Field(int x_, int y_, int value_) : x(x_), y(y_), value(value_) {}
    bool operator<(const Field& other) const
      { return value < other.value; }
};

// entry of the DP table
struct Entry {
    int preX, preY, seqLength;
    Entry(int preX = -1, int preY = -1, int seqLength = 0)
      : preX(preX), preY(preY), seqLength(seqLength) {}
};

// for iterating elegantly over neighbors
const int DX[4] = {1, -1, 0, 0}, DY[4] = {0, 0, 1, -1};

vector<int> ascendingSequence1(const vector<vector<int>>& A, int n, int m) {
  // sort fields
  vector<Field> fields;
  for(int x = 0; x < n; ++x)
    for(int y = 0; y < m; ++y)
      fields.push_back(Field(x, y, A[x][y]));
  // sort fields
  sort(fields.begin(), fields.end());
  
  // entry of the DP table
  Entry T[n][m];
  
  // for computing the DP table
  for(int y = 0; y < m; ++y)
    for(int x = 0; x < n; ++x)
      if(A[x][y] < T[x-1][y].value)
        T[x][y] = T[x-1][y];
      else if(A[x][y] < T[x][y-1].value)
        T[x][y] = T[x][y-1];
      else if(A[x][y] < T[x+1][y].value)
        T[x][y] = T[x+1][y];
      else if(A[x][y] < T[x][y+1].value)
        T[x][y] = T[x][y+1];
      else
        T[x][y] = Entry(x, y, T[x][y-1].value + 1);
  
  // extracting solution
  int longest = 0, x = 0, y = 0;
  for(int y = 0; y < m; ++y)
    for(int x = 0; x < n; ++x)
      if(T[x][y].seqLength > longest)
        longest = T[x][y].seqLength, x = T[x][y].preX, y = T[x][y].preY;
  
  // reconstructing solution
  vector<int> sequence;
  while(T[x][y].seqLength > 0)
    sequence.push_back(A[x][y]),
    x = T[x][y].preX, y = T[x][y].preY;
  return sequence;
}
```
for (int y = 0; y < m; ++y)
    fields.push_back(Field(x, y, A[x][y]));
sort(fields.begin(), fields.end());

    // fill table
vector< vector<Entry> > T(n, vector<Entry>(m));
for (int i = 0; i < (int)fields.size(); ++i)
{
    int x = fields[i].x, y = fields[i].y;
    T[x][y].seqLength = 1;
    for (int j = 0; j < 4; ++j)
    {
        int neighborX = x + DX[j], neighborY = y + DY[j];
        if (neighborX < n && neighborX >= 0 && neighborY < m && neighborY >= 0)
            // seqLength == 0 for unvisited entries!
            if (T[neighborX][neighborY].seqLength + 1 > T[x][y].seqLength)
            {
                T[x][y] = Entry(neighborX, neighborY, T[neighborX][neighborY].seqLength + 1);
            }
    }
}
return findBestSequence(A, n, m, T);

vector<int> findBestSequence(const vector< vector<int> >& A, int n, int m
                           const vector< vector<Entry> >& T)
{
    // find best result
    int bestX = 0, bestY = 0, bestLen = 0;
    for (int x = 0; x < n; ++x)
        for (int y = 0; y < m; ++y)
            if (T[x][y].seqLength > bestLen)
            {
                bestLen = T[x][y].seqLength;
                bestX = x;
                bestY = y;
            }

    // extract result
    vector<int> ret(bestLen);
    int i = bestLen - 1;
    while (bestX != -1)
    {
        ret[i] = A[bestX][bestY];
        int tmp = T[bestX][bestY].preX;
        bestY = T[bestX][bestY].preY;
        bestX = tmp;
        --i;
    }
    return ret;
}

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.

**Code sample:**

```cpp
vector<int> ascendingSequence2(const vector<vector<int>>& A, int n, int m)
{
    // fill table
    vector<vector<Entry>> T(n, vector<Entry>(m));
    for (int x = 0; x < n; ++x)
        for (int y = 0; y < m; ++y)
            fillEntry(A, n, m, T, x, y);
    return findBestSequence(A, n, m, T);
}

void fillEntry(const vector<vector<int>>& A, int n, int m,
               vector<vector<Entry>>& T, int x, int y)
{
    // entry has already been filled?
    if (T[x][y].seqLength != 0)
        return;
    T[x][y].seqLength = 1;
    for (int j = 0; j < 4; ++j)
    {
        int neighborX = x + DX[j], neighborY = y + DY[j];
        if (neighborX < n && neighborX >= 0 && neighborY < m && neighborY >= 0)
            if (T[neighborX][neighborY] < A[x][y])
                fillEntry(A, n, m, T, neighborX, neighborY);
            if (T[neighborX][neighborY].seqLength + 1 > T[x][y].seqLength)
                T[x][y] = Entry(neighborX, neighborY, T[neighborX][neighborY].seqLength + 1);
    }
}
```