1. See course web page for code.

2. Consider the strings \( v_1 \ldots v_m \) and \( w_1 \ldots w_n \). The three cases we considered originally are given below.

\[
\begin{array}{ccc}
\ldots v_m & \ldots - & \ldots v_m \\
\ldots - & \ldots w_n & \ldots w_n
\end{array}
\]

The additional condition given here would result in certain alignments getting double-counted. Precisely, the alignments from the first group (out of the three listed above) ending with \( \ldots - v_m \ldots - \ldots w_n \ldots w_n \) get double-counted as part of the second group, as each of the sequences counted above will be identical to the corresponding sequence which ends with \( \ldots v_m - \ldots - \ldots w_n \ldots w_n \). Hence, \( g(m-1, n-1) \) alignments get double-counted. Adjusting for this double-counting gives the recursion

\[
g(m, n) = g(m-1, n-1) + g(m, n-1) + g(m-1, n) - g(m-1, n-1) = g(m-1, n) + g(m, n-1)
\]

\[
g(m, n) = \binom{m+n}{n}
\]

is the general solution of the above recursion (Pascal’s formula).

Under the new set up, we ignore permutations of a series of indels. Another way to count the number of alignments is to identify pairs \( v_i w_j \) which are aligned (as matches or mismatches). In any alignment, there will be \( k \) such pairs with \( 0 \leq k \leq \min\{m, n\} \). For any given \( k \), there are \( \binom{m}{k} \) ways to choose the \( v_i \)'s and \( \binom{n}{k} \) ways to choose the \( w_j \)'s. Hence, the total number of alignments is given by

\[
\sum_{k \geq 0} \binom{m}{k} \binom{n}{k} = \binom{m+n}{n}.
\]

3. Note that a score of +1 is added for each match. Thus, \( s(v, w) \) is exactly the maximum number of matching \( v_i \)'s and \( w_j \)'s. Hence \( n - s(v, w) \) represents the number of \( v_i \)'s which do not have a matched \( w_j \) in the best alignment. Similarly, there are \( m - s(v, w) \) number of \( w_j \)'s which are not matched with a \( v_i \). To transform \( v \) to \( w \), we need to account for these unmatched \( v_i \)'s and \( w_j \)'s using indels, each of which contributes a +1 to \( d(v, w) \). Hence, it must be that

\[
d(v, w) = m + n - 2s(v, w).
\]

4. Since we want to match a suffix \( v_i \ldots v_n \) of \( v \) with a prefix of \( w \), the optimal path could start from any one of the \( v_i \)'s. Hence we initialize \( s_{i,0} = 0 \) for all \( i = 1, \ldots, n \) (note that \( s_{0,0} = 0 \) in all cases). This step is equivalent to adding arcs from \((0,0)\) to \((i,0)\) each with weight zero. Similarly, since we are matching a suffix of \( v \) with \( w_1 \ldots w_j \), the optimal path could end on any of the points in the last row of the grid (corresponding to \( v_n \)). Hence, to find the optimal overlap alignment, we need to look for the point with maximum score along the last row and trace back till it hits the first column (corresponding to \( w_1 \)). Equivalently, we could add edges from \((n,i)\) to \((n,m)\) each with weight zero (for \( i = 0, \ldots, m-1 \)). \( s(m, n) \) will be the score of the optimal overlap alignment in this case.

The general recursion remains the same as that for the global alignment problem. Hence the algorithm also runs in \( O(mn) \) time (just as the global alignment algorithm).