PART B:

Describe DTW:

Dynamic Time Wrapping (DTW) is the algorithm to align 2 sequences A and B, by stretching/ contracting sequence A along the time axis in order to minimize the total sum of the absolute difference (or distance) between matched points (points occurring at the same time). The details of stretching and contracting are covered in the answer to the next question below.

Explain how DTW differs from simple sequence matching:

By “simple” sequence matching, I assume the question is referring to directly match sequence A against sequence B without any transformation. Obviously, this will result in poor alignment if the speed varies between the 2 sequences A and B.

DTW handles this misalignment by utilizing stretching and contracting on sequence A.

For stretching sequence A, it is equivalent to changing a data point in A into multiple duplicates in the sequence across multiple timesteps, and compare the same data point against multiple data points in B.

For contracting sequence A, it is equivalent to stretching sequence B: by comparing different data points in sequence A against the same point in sequence B (thus synonymous to replicating the data points in sequence B into multiple timesteps).

More importantly, the stretching and contracting is not done linearly (eg: stretch the sequence A 2x across all time steps) but is applied at varying rate at different points to minimize the difference between signals, thus allowing DTW to handle non-linear variations in speed and timing.

With the stretching and contracting as described above, the relative order is maintained, which fits with the usual requirements for time series matching, so another plus for DTW.

Detail the initial setup of the cost matrix:

As shown in the diagram, given the row index (vertical axis) representing the sequence A and the column index (horizontal axis) representing the sequence B.

Define segment(A, start, end) as the segment/ sequence from the start-th data points to the end-th data points of sequence A, and similar for B. Define the 0-th data points of both sequence A and B to be 0 (for convenience purposes, since setting the matrix padded with fake 0-th data points helps us to avoid the need for checking out of bounds during implementation).

Meaning of each cell: Each cell (i, j) shows the total minimum cost to align segment(A, 0, i) and segment(B, 0, j) (with stretching and contracting if needed).

Initial values are set as:

1. At [0, 0], the matrix is set to 0, as the distance between 0-th data point between A and B (as defined above) is 0 – 0 = 0.
2. For all other cells, labelled it as infinity, since these values are unknown at the start before traversing through both sequences. Any impossible values (such as -1) can be used to signify that these cells' values are undefined. As we're computing the minimum cost in each cell, choosing infinity (an unrealistically large number in practice) is convenient, as:
    a. When an unknown value is used in computation, the infinity value will guarantee that when we apply the Min function to select the minimum cost, any calculation with undefined/ infinity involved will always be ignored, giving preference to results computed from known values.

Describe the process of calculating the optimal path through the cost matrix:

The criteria for a valid optimal path are:

1. Cost definition: The cost/ distance between 2 data points are the absolute difference between their values.
2. Only stretching (or insertion in the pseudocode) or contracting (or deletion in the pseudocode) are permitted, which implies, in the cost matrix, a value of a cell (i, j) can only be:
    a. Total cost at (i, j) = cost at (i , j) + total min cost at (i-1, j)
    (Stretching: Keep matching the i-th point in A with the j-th point in B (which was used to match with i-1-th point in A), equivalent to duplicate/ insert the same j-th point in B)
    b. Total cost at (i, j) = cost at (i , j) + total min cost at (i, j-1)
    (Contracting: Keep matching the j-th point in B with the i-th point in A (which was used to match with j-1-th point in B), equivalent to contracting the j-th point of B to the same timestep as its j-1-th point. I dislike the term deletion used in the pseudocode, as we still takes into account of the cost of the j-1-th

point in B and it’s more of overlapping the different points onto the same timestep rather than deleting point j-1-th)

   c. Total cost at (i, j) = cost at (i , j) + total min cost at (i-1, j-1)
   (Matching: Finish matching the i-1-th point of A and j-1-th point of B, so move on to start the next pair)

3. The path needs to have minimum total distance cost, so we simply takes the minimum of the 3 possible ways of matching described below, resulting in the formula listed in the pseudocode: total min cost at (i, j) = cost at (i, j) + min(total min cost (i-1,j), total min cost (i, j-1), total min cost (i-1, j-1)))

Part C:

Refer to the last part in part B for the explanation of the recurrence relation used.

Dynamic programming (DP) is used to optimize the recursion shown above with the cost matrix. The recurrence facilities DP as it:

1. Shows how to compute the cell (i, j) using sub-problems from (i, j-1), (i-1, j), (i-1, j-1).
2. Indicates the large overlapping of sub-problems (eg: both cell (i, j), (i-1, j) and (i, j-1) needs the value of cell (i-1, j-1), thus can be optimized by storing the value of computation once).

Since the total minimum cost at point (i, j) only requires the total minimum cost at point (i-1, j), (i, j-1), (i-1, j-1), by iterating down and to the right, we can ensure that all the required information to compute the minimum cost at point (i, j) were computed before reaching the point (i, j) in a bottom-up manners.

DP speeds up the process massively by storing the value of total minimum cost in the cost matrix, avoiding the needs for re-computation of the total minimum cost for several repeated states.

Part E:

Computational complexity: Discuss the computational complexity of the algorithm implemented in Part D (i.e., DTW with boundary constraints). is it different from the algorithm in Part A (i.e., standard DTW)? State and explain the recurrence relation.

For space complexity: Both algorithms (with or without constraints) use the same amount of space: O(m * n) for the m * n table.

For time complexity:

With n, m being the length of sequence A and B respectively.

Let w be the window size, and w <= m (since it's useless setting window size = grid's width).

Basic operations: Cost calculation for each cell

Original time complexity: θ(n * m) (shown through the 2 nested loop) in general case

DTW with constraints:

Define function l(lower, upper, value) to limit a value to be between lower and upper:
$l(lower, upper, value) = \min(\max(value, lower), upper)$

The 2 bounds for the column range in row i is l(1, m, i – w) and l(1, m, i + w).
If (i – w) is outside of [1, m], then no cost calculation is performed, and the program should break out of the loop, thus 0 cost:

Let $F(i) = \begin{cases} l(1, m, i + w) - l(1, m, i - w) + 1 \; if \; (i - w) \leq m \\ 0, \; otherwise \end{cases}$

$\theta(w, m, n)$= T(w, m, n) = $\sum_{i=0}^{n} F(i)$

In the general case, there are no possible ways to simplify the expression for θ, since the best case is 1 (when w = 0, m = 1) and the tightest worst case varies, depending on which constraints are used for n, m and w (since it might affect the chopped part outside the board of the window, which can reduce total operations), but bounded by (2w + 1) * n.

For example, for n > w and m >= n + w:

(2w + 1) * n – (w + (w -1) + … + 1) (Subtraction due to the missing of the left part at the top)

= (2w + 1) * n – (w + 1) * w/2, which varies depending on whether n is significant larger than w, or just slightly larger (n = w + 1).

Still, looking at the expression, an upper bound (although might not be the tightest) is clearly O((2w + 1) * n), since:

$$\mathrm{l}(1, \mathrm{m}, \mathrm{i} + \mathrm{w}) - \mathrm{l}(1, \mathrm{m}, \mathrm{i} - \mathrm{w}) + 1 \leq (i + w) - (i - w) + 1 = 2w + 1$$

A simpler (yet not exact, due to edge cases for rectangle grid) bounds are:

Best case: $\theta(1)$

Worst case: O((2w + 1) * n)

Compared to the original DTW, the time complexity formula, as shown, is not the same. Key insights can be drawn from the simplified bounds are:

1. if w << m (eg: w is a fixed constant), then the constrained version will have superior performance, since O((2w + 1) * n) < θ(n * m)
2. if w = k * m + c (where k and c are real-valued constants and w <= m), then the time performance are comparable, since for each row, at least w items are scanned.
3. Since w <= m, then there is no case when the time complexity of the constraint version is worse than the unconstrained.

Recurrence relation:

The same recurrence relation is used from part B (with dtwMatrix[i][j] storing the minimum total cost to match the i-th point in A with the j-th point in B from the pseudocode):

cost <- abs(sequenceA[i-1] - sequenceB[j-1])

dtwMatrix[i][j] <- cost + min(dtwMatrix[i-1][j], dtwMatrix[i][j-1], dtwMatrix[i-1][j-1])

However, for each row i-th, to respect the constraints:

1. Check if i – w > m. If it is, the entire window is outside of the grid, so stop the iteration.
2. Limit j to be between (max(0, i - w), min(m, i + w))

Although we should only consider the cells from previous row that is within the specified window (eg: dtwMatrix[i-1][j] might not be in the window), but no additional check is required, because if the cell's not in the window region, then its value is still set at INFINITY, thus guaranteed to be ignored if there is valid cell in the min function during cumulative cost calculation.

Thus, we can apply the recursion from part B as is, with only the extra check for the bounds.

Part G:

Computational complexity: Discuss the computational complexity of the algorithm implemented in Part F (i.e., DTW with total path length constraint). Is it different from the algorithm in Part A (i.e., standard DTW)? Explain the recurrence relation.

Is it different from the algorithm in Part A:

Yes, the question required an additional constraint on path length. Details explained below.

Recurrence relation and algorithm explanations:

The recurrence will be shown first in a 3D matrix representation for simplicity, and further shows how it can be optimized to only 2D matrix during implementation.

Given a 3D table dtw, the cell dtw[i][j][k] represents the minimum cumulative cost to match the i-th point of A to the j-th point of B, at the path length of k.

The modified recurrence from the pseudocode:

cost <- abs(sequenceA[i-1] - sequenceB[j-1])

dtwMatrix[i][j][k] <- cost + min(dtwMatrix[i-1][j][k-1], dtwMatrix[i][j-1][k-1], dtwMatrix[i-1][j-1][k-1])

The same logic applies as explained in part B/C, with just a slight change to ensure that we only consider cumulative cost from the previous path length of k-1, since we only want to compute the minimum cost for the node at end of path length k, which is calculated based only on the nodes at end of path length k-1 with the exact same recurrence as part B.

Few notes for implementation:

1. Initialize a board filled with inifinity at 0 steps, then start iterating from path length 1 to avoid index issue
2. INFINITY + positive number = INFINITY theoretically. Practically might want to capped it as some large value to avoid overflowing.

Notice that for each path length k, we only need to refer to path length (k – 1) cost table, thus there is no need to store more than 2 n * m tables at any given step with path length k.

To avoid having to repeatedly either copying values or malloc-ing new table for each step, I used 2 tables for my implementation: 1 cumCost table storing the cumulative cost at path length k, and 1 pathLength table storing the max path length at each cell.

The same recurrence relation stated above is used, but instead of using a third dimension, my algorithm, for the path length k, for each cell [i][j] in the cumCost table:

1. For the minimum cost of cells at path length k-1 used in the calculation:
   Check the pathLength table to see if the cell's maximum path length is k-1.
   If yes, uses the value in the recurrence relation. If not, then skip it.

2. If the new cost is less than INFINITY (possible path), then update the cost in cumCost[i][j] and update the pathlength[i][j] to be k.

3. Since we're mutating the same table, before updating the cell[i][j], we need to update all the cells that depends on the old value of cell[i][j] first, which includes: cells at [i+1][j], [i][j+1], [i+1][j+1] (derived from the recursion). This can be guaranteed by iterating from the bottom right cell, going left and up, instead of right and down in part A.

The correctness is equivalent, but the algorithm is much easier to implement, and saving space (and other copying/ malloc-ing operations).

Space complexity: O(m * n), for the 2 m * n tables storing the cumulative cost and path length state.

Basic operations: Cost calculation for each cell

Time complexity:

With m and n being the length of sequence A and B respectively; k is the maximum path length.

$\theta(m * n * k)$ as shown through the 3 nested loop in the implementations.