Common Mistake: Confusing indices with miles. i is an index, but m_i and k are in miles. Using (i - k) doesn’t really make sense.

Common Mistake: Base case L[0] should be 1 (True). If you set it to 0 (False), the whole array becomes 0 (False).

Common Mistake: Making the max function overly complicated
Keep in mind: L[i-1, j-i] ≤ L[i-1, j] ≤ L[i-1, i-1] + 1
L[i-1, j-i] ≤ L[i, j-1] ≤ L[i-1, i-1] + 1

Common Mistake: Always setting L[i, j] = 0 or 1 for case when (x_i ≠ y_j) will cause problems later on.

Consider the example:

<table>
<thead>
<tr>
<th>X = A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y = A</td>
<td>B</td>
<td>G</td>
<td>F</td>
<td>E</td>
</tr>
</tbody>
</table>

What is L[5, 5]?

1 2 3 4 5
DYNAMIC PROGRAMMING TIPS

STEP 0) Make sure you understand the question
Nothing worse than finding the right answer to the wrong problem
Subset ≠ Subsequence ≠ Substring

STEP 1) Figure out what the subproblems are
HINT: For many problems, the solution is just the final instance of the subproblem
If a 1D solution is possible, it is usually simpler, but sometimes 2D is needed
If you draw out a table or array, each cell represents one subproblem

STEP 2) Once you have identified the subproblems, determine the base case(s)
Which instances of the subproblem are given/trivial before you even start?
If you have drawn out a table/array, you can fill in the base case immediately
- Label the columns and rows so you don’t make mistakes filling it in

STEP 3) Define your recurrence
How can you find the value of one subproblem based on previous subproblems?
How will the cells in your array/table depends on the other cells?
HINT: If unsure, start with the subproblems that depend directly on the basecase
HINT: Pay attention to ‘cases’

STEP 4) Turn your recurrence into an algorithm
Be sure to set your basecase and instantiate the array/table if necessary
Pay attention to your indices
HINT: For many problems you just need a nested loop to apply your recurrence

General Tips
You should know how to reduce problems to instances of well-known problems (i.e. Knapsack)

For simple problems, you may be able to just construct a DAG and find the longest / shortest path.