# Question 1

1. **Least Total Cost Recurrence**

The goal here is to write down a recurrence for , where represents the least total cost of going from the source square to square in the grid. Since this problem possesses optimal substructure, then this means that an optimal solution to a problem contains within it optimal solutions to subproblems. In this case, for the least total cost to get from the source square to any square , there exists within the grid the least total costs to get from the source square to any other square not , so long as it is reachable from the source square and is not an illegal move. This implies that the least total costs are dependent on other least total costs, a property it possesses via optimal substructure.

For any square , let represent the square immediately before , where its least total cost is dependent on it. Then, let represent the cost to get from square to square as stored in the cost function . Thus, the following recurrence can be written for as follows:

For the third case, this recurrence says that: for any square that does not fall under the first two cases, take the minimum between either what is already there as the least total cost for square , or take the least total cost for square plus the cost to get from square to square . For part (c) of this question, this conditional logic is coded in the Python program as follows:

This says that if the stored least total cost in square is greater than the stored least total cost in square plus the cost to get from to , then update the least total cost of square . This logic is needed because for a square , assuming it is not on the edge of the grid, there are at most ways for its least total cost to be updated: , , , or no update. This recurrence stores the least of these, assuming validity, as and performs the stated comparison.

1. **Least Cost Path Recurrence**

The goal here is to write down a recurrence for , where represents the least cost path to get from the source square to some square . Similar to what was mentioned in part (a), the optimal or least cost path can be “lopped off” or “eaten into” from the right side of the problem, i.e., starting from the destination and going backwards towards the source. First, going in the forward direction of the algorithm, the final least total cost will be obtained once it processes all the set of moves in the cost function . The total cost in the set of destination squares with the lowest value is the last value in the optimal path, since the algorithm seeks the least cost path.

From here, the recurrence can be realized by working backwards, meaning from an optimal destination square , determine all the immediately previous squares that it could have come from. If at the left edge of the grid , it would have either come from or . Similarly, if at the right edge of the grid , it would have either come from or . Lastly, if anywhere in between (not including source squares), it would have come from either , , or .

The algorithm determines which one it comes from by choosing the preceding square with the least total cost from the source up to that point, hence it is dependent on the function from part (a). However, for the algorithm this is already known, as the program in part (c) tracks not only the optimal squares to take, but also the parent of those optimal squares, or in other words, where they came from when building the dynamic programming array. This strategy can be extended for every square in the optimal path, in a backwards manner, until the source square is reached. With that being said, the following recurrence can be written for as follows:

In words, for some square , the algorithm chooses the least cost square from one of , , and , and appends that to its least cost path. This decision is based on each of their costs from part (a). If represents the least cost immediately previous square, then this is taken because is the least total cost.

# Question 2

1. **Optimal Substructure**
2. **Feasibility**
3. **Fewest Stops Recurrence**

* The notation means you must leave by a certain time to ensure that you reach by the end of the day.
* Other wording: Assume that we always start to drive at a particular time of day, say 7 AM, and always end driving at a particular time, say 7 PM. So, when we say, “within a day,” we mean if we leave by 7 AM, then we arrive by 7 PM.
* What means is that if we leave at 7 AM, then we’re guaranteed to reach no later than 7 PM.

1. **Pseudocode**

* Space efficiency characterizes the space, in addition to that needed to encode the input, the algorithm consumes/requires.
* In other words, ignore the space needed to encode the input, but count every other space that is allocated. And yes, you would include the space needed to encode the output.

# Question 3

* To be clear, we do not need resources such as ways to measure time and distance.

1. **Lower and Upper Bounds on Alignment Length**
2. **Worst-Case Number of Possible Alignments**

* May end up with something more than . The problem asks only that you establish as a lower bound, as that is what means.