**Problem 1**:

(1) Argue that your recursive relation is correct.

In the first question, there are two given arrays and , which consist of elements indicating points such as in the coordinate, and the length of a given band , which is a threshold to allow frogs to traverse. We wrote a code to return true or false when the points that are the elements of two arrays respectively can be traversed by frogs if and only if their distance is less than or equal to the length of the band.

To solve this problem, we use recursive ways to find appropriate answers that fulfill the requirements of it. A pseudo code is below to explain the flow of the algorithm.

def Cross ():

**# Base case**  
if the end of : return True

**# Condition that frogs can traverse**  
if distance b/w is further than : return False

**# To set result false for checking true**  
result=False

**# Recursion**  
if can move forward together: result = result or Cross ()  
if can move forward: result = result or Cross ()  
if can move forward: result = result or Cross ()

return result

< Algorithm explanation >

The code of the algorithm is aimed at determining whether frogs represented by arrays and can traverse a certain distance, , which is the length of the band, while moving along their respective paths. In the base case, if both and have reached their ends, the program returns True, indicating that the frogs have successfully traversed the paths.

Then, it checks the distance between the point in and the point in whether the specified distance is exceeded the length of the band . If the distance is further than the length of the band, the program will return False. The variable ‘result’ is initialized to False, which will be used to track whether the frogs can traverse the paths.

Lastly, there are three paths to be used in recursive ways. The first recursion checks if the frogs can move forward together, which means both indices increased by 1. The second recursion checks if only can move forward, which indicates index incremented by 1 while index remains the same. The last recursion checks if only can move forward, which stands for index added by 1 while index remains the same. If these are possible, they update the ‘result’ variable accordingly, and thus the program will return the result.

The total time complexity is which can be the worst case because whenever the points move there are three branches to run by recursions. It means that the length of arrays such as and is a determinant affecting a total time complexity.

< Correctness >

To prove the correctness of the code, we need to examine whether the points are a base case or not. If the last indices in and in are less than or equal to 0, it means frogs cannot traverse to next point, and when the value of the distance between the points is greater than the length of the band, it can be a negative value, which means also frogs cannot move forward.

On the other hand, if and is large enough, the program would successfully run recursions until reaching the base case above. All recursions generate three branches, and these three branches should return true or false without any problems. Unless the pointers move out of the range of list, they always traverse the right direction, thus the true or false values will return.

**Problem 2**:

(2) Explain why the algorithm of part (1) is slow, and how you can speed it up(substantially) using Memoization

< The reason why it is slow >

The main reason why the algorithm of part(1) has a slow processing speed to run the program is that it potentially re-computes the results for the same subproblems multiple times due to its recursive nature. For instance, and are given arrays and the length of the band respectively. During the recursion, the program starts from to traverse next points individually. In the case of the recursion when moving the point forward to reach , the program processes a result from the case, which has the same result from when the point moves to . As we can guess, the program would be able to produce a huge number of same results that make the processing speed slow if the size of two arrays are large enough.

< how you can speed it up(substantially) using Memoization >

To avoid the situation above and speed up the algorithm substantially, memoization technique can be a solution, which is one of useful tools to reduce the number of calculations the program has already done. In fact, memoization is used for storing the results of expensive function calls and returning the cached result when the same inputs occur again, which means we can store the results of previously computed subproblems to avoid redundant computations. A pseudo code is below.

def Cross ():

**# Hash for memoization**  
memo = { }

def Cross\_memo(:

**# Base case**  
if the end of : return True

**# Check the points in memorization hash**  
if in memo: return

**# Condition that frogs can traverse**  
if distance b/w is further than : return False

**# Recursion**  
result = Cross\_memo()

or Cross\_memo()

or Cross\_memo()

= result

return result

return Cross\_memo(

we need to define a hash table for memoization, which is a dictionary to store the results of subproblems. Before making a recursive call, the program checks if the result for the current subproblem exists in the memorization table. If it does, the program return the cached result instead of recomputing it. After computing the result for a subproblem, the result is stored in the memorization table for future calculations. The number of computations would be substantially reduced more than previous code without memorization. As a result, by using memorization, the algorithm avoids redundant computations and significantly speeds up the process of determining whether the frogs can traverse the paths within the specified distance .