**1. Search strategy used, implementation details, data structures, time/space complexity.**

We implemented two search strategies during our experimentations, Breadth-First Search (BFS) and A* search. We ended up using the BFS strategy.

First, we came up with an uninformed strategy, the BFS. We came up with this idea as all edge costs are equal if we treat each action (moving adjacent and jump sequence) as the cost of 1. Given the circumstances, BFS should yield a lowest-cost path.

To implement this, we hash the current board to mark visited grids and initiated a FIFO queue to maintain the current position of the red frog, current board status and all valid moves from this node using helper functions. Note that we also adopt depth-first search (DFS) to store all possible moves for one position. Then, we loop through each move in all the possible moves. If the new board has not occurred before after the move, we add it to our queue and keep searching using the same method. We keep looping through the queue and apply the possible moves until the first time we have reached the bottom row. If a solution is found, we return the moves, and none otherwise.

The time and space complexities are both $O(b^d)$, as followed from the standard BFS.

Here, the branching factor is the number of actions for each state. The adjacent action for moving is 5. The jump sequence accounts for the rest of the possible actions. For our single frog problem, this is fairly small too. The number of frogs influences the possible jump sequence, and possible sequences are fairly limited since the direction and blue frogs' position are fixed (our red frog can't go back and have no need to jump over the same frog). Thus, even though finding jump-sequence used a DFS, the performance is negligible. However, it is still required to check for 5 directions no matter a jump is valid. Thus, the branching factor is >=10(and should be very close to 10 on average case). In worse case, depth $d = 8 * 8 = 64$ as the length of the board is 8. This is the total number of reachable states. However, 64 is a very high estimate because on average case the path are likely to be around 8 , which the frog keep going downwards, instead of visit every node. It is even less if jump sequence exists.

Then, we experimented with the A* search. We defined an appropriate Manhattan Distance function and a heap as priority queue, so that we are always exploring the paths with the lowest f-score (total cost). We use the queue to store the current f-score, g-score, frog position, board status and valid moves. Then we loop through the heap, get all possible moves using a DFS helper function and apply the moves to find the path with the lowest f-score. We terminate the loop until we reach the bottom row.

The space and time complexities are both $O(b^d)$, and the priority queue is of $O(logN)$. However, in practice, A* search usually performs better than BFS due to the heuristic function and priority queue, such that it will try to search for the lowest cost path every move and thus exploring much fewer nodes.

We ended up adopting the BFS approach, as it always gives us a correct and optimal solution. Although A* search is more efficient, it was hard to define a heuristic function that is admissible for jump-over sequences while being close to the actual cost before even seeing the board. This makes A* search suboptimal in edge cases. When we try to define a heuristic function that is underestimated the cost, the performance get worse very quickly and have no advantage over BFS, while not promise 100% being optimal

**2. If you did not use a heuristic based approach, justify this choice.**

We ended up using BFS which is not a heuristic based approach, as we encounter difficulties defining a heuristic function that also considers jump-over sequences.
In our A* search attempt, we tried with a modified Manhattan distance as our heuristic function. We first compute the vertical distance from the current frog's position to the bottom row. Then, we measure the current horizontal distance by calculating how far the frog is from the two middle columns (column 4 and 5). Finally, we take the minimum of these two values as our heuristic.
However, A* search tends to overestimate in cases where there is a long jump-over sequence. Consider cases where the red frog starts on positions $(r, c)$ where $r < (BOARD\_N)/2\,(4), c < 3\ or\ c > 5$, but can jump consecutively over blue frogs that end up being at least 2 distances vertically downward. The whole action counts as 1 cost, but the heuristic could easily overestimate the jump sequence at a much greater cost. For A* to be optimal, the heuristic must be admissible but still being as close as possible to the actual cost. Therefore, we decided to go with BFS instead.

**3. Imagine that all six Red frogs had to be moved. Discuss how this impacts the nature of the search problem, and how solution would need to be modified in order to accommodate it.**

For a native suboptimal solution, we can just do BFS for each frog once and move them to goal state one by one. The issue is instead of a single-agent problem, having multiple frogs assuming no grow function just yet, will require a more sophisticated searching algorithm. There will be scenarios where orders of frogs is important, the red frogs can block each other when some frog move before frog that is supposed to move first, block its possible path to bottom row. Moreover, given lily-pad disappear after frog move, a frog moving can result in another frog isolated in corner without adjacent lily-pad and stuck, even when stuck frog can jump over the first frog. In one word, a solution might exist but not to be found by our current algorithm, because we didn't handle the order of movement from different frogs. However, this approach have relatively small impact on performance, since it just run BFS six times.
If we aim for an optimal solution, we can track state of all frogs and their possible moves at the same time, this will be very expensive in time/space complexity and branching factor and depth will all multiply by 6 in worst case, given we have $O(b^d)$ for both. An exponential relation will both base and power multiplied will be very impractical in large boards and many frogs. On the bright side, it fixed many issues we listed above.
This modification will significantly increase the complexity of the search problem. The state complexity for a single frog is $O((BOARD\_N)^2)$, but it is $O(6(BOARD\_N)^2)$ for 6 frogs to track their positions and moves to modify the board and for other frogs' jump sequences.
Additionally, in response to the new rule, we also need to change the goal state check to check if all 6 frogs are at the bottom row. We might also change the hash function for the board and the DFS valid-move storing helper function to accommodate the frogs. For our search logic, we now need to handle jump-overs between the red frogs instead of just blue frogs and track their progress simultaneously.

**Reference:**
LLM has been used to improve readability and formatting. we used some helps from llm to

generate a* search algorithm to play around however a* is not part of the submission formalLY