Part C:

Parameters of considerations:

d: Total number of characters in the dictionary (dictionary size)

l_m: The maximum length of the longest word in the dictionary

k: The size of the alphabet

n: The dimension of the puzzle board

Note that: $k <= n^2$, since if $k > n^2$, we can simply restrict our alphabet to all the unique characters on the board, and filtering out irrelevant words containing any other characters while reading in the dictionary, since it won't appear on the board anyway.

Worst case assumption:

For worst case analysis, we will assume that the dictionary is designed to maximize the size of the prefix-tree (which might be different in each case) and the characters used inside the grid is set to against any potential advantages.

FOR ORIGINAL PROBLEM:

Algorithm used:

The algorithm used to solve this problem uses recursion/ backtracking. The details are included in the code implementation for part A, but rough steps are:

1. Start from a cell[i][j] on the board
2. If already reached the leaves of the prefix tree, then mark the found words (by mark that leave node on the prefix-tree for later retrieval. Details below) and return
3. If the cells are already marked as visited or the location is off the board, return early
4. If not, then check if the character on the board[i][j] matches the character on the edge of any child of the current prefix tree node
5. If not, return early
6. If yes, then it is a (potentially) valid path, thus:
    a. Mark current cell as visited
    b. Move down the prefix tree following the edge with the matched character
    c. Recursively calls step 2 to 5 on the 8 cells nearby from the current cell

7. After finish step 6, mark the current cell as unvisited while backtracking, to allow different path to visit cell[i][j] if haven't visited it before in such path

Essentially, the algorithm tests all possible path of length l_m starting from each cell[i][j] in the board.

Worst case implication:

Since there is no limit/ constraints on the type of input, and there are k possible characters at each position in the string, and the maximum string length is l_m, at worst, the number of all possible combinations are:

$k^{lm} + k^{lm-1} + k^{lm-2} + \cdots + k = \frac{(k^{lm+1} - k)}{k-1} \approx k^{lm}$ (for large value of k -> infinity).

For part A, the worst input would be when all the paths on the board are valid, causing returning early impossible. A simple case is when the entire grid is filled with 16 character "A", and the dictionary contains a word that is "aaa … a" with 16 "a", making all possible paths valid.

Space complexity: $O(n^2 + k*k^{lm})$ for the table to store the puzzle table (and a table with the same size to check if a cell is visited or not) and the prefix tree.

For the prefix tree, since there are k^{lm} different string combinations, and each of them will have an extra k-sized array storing their single terminal null value, the total size of the tree is therefore $k*k^{lm}$. For the intermediate character, the overlapping between the string ensures that the prefix tree contains only k^{lm} nodes for all the string combinations.

Time complexity:

The algorithm involves 2 steps: building the prefix-tree, retrieving found words and searching for matched words, which are best analyzed separately, since they serve different purposes and have different basic operations, so attempting to fuse them into 1 complexity would yield either an incomplete or meaningless result.

Basic operation: Allocate memory for the prefix-tree or the scanning of the dictionary

For prefix-tree building:

$O(\max(k * k^{lm}, d))$, as explained above, this is the maximum size of the prefix-tree in the worst possible tree.

Note: If the dictionary has an infinite duplicate of the same string, then the scanning operations can become dominant, yielding an O(d) worst case, with d being the total number of characters in the dictionary. Memory allocation is sometimes higher, since for 1 character, it might need k memory allocations for the k-sized pointers, if the character leads to a brand-new node.

For the final output retrieval:

Basic operations: Check if the node is marked during the search

The tree is traversed and get the output words in lexicographic order by checking the leaf nodes of which word is marked as found and used the dictionary index of the word (eg: which word is that in the input array of words) stored in that leaf to copy the correct string to the solution list.

Only the paths toward the leaf containing found words are marked, so we can simply ignore the irrelevant path that isn't marked. This optimization leads to time complexity of O(n) with n being the total characters in the found words on the grid. This is the same for part D.

The marking is done for a node, if it's the end of a searched word on the grid or it's the parent of a marked node during the search recursion. This is O(1), so no complexity added.

For matched word searching:

Basic operation: Character comparison (between the cell's and the prefix-tree's character)

Let $r = \min(n^2, l_m)$

For **each** cell[i][j], the upper bound for the number of paths is 7^r as there are at most 7 viable directions at any position in the path (ignoring 1 direction coming back to the previous cell, which will be terminated immediately), and we need to pick these directions r times for the path with max length r (equivalent to the min(max height of the prefix-tree, all grid's cells)). At each character, we process O(1) character comparison, so $O(r * 7^r)$ in total.

There are n * n cells on the board, so the total time complexity for searching the matched words is: **$O(n^2 * r * 7^r)$ (with $r = \min(n^2, l_m)$)**

FOR INCORPORATING THE FACT THAT THE LETTER CAN ONLY APPEAR ONCE:

Note:

For convenience, each cell[i][j] on the board will be represented by a node (labelled uniquely by (dimension*i + j)), connecting initially to its surrounding cells/ nodes. The edge represents that we can move between the 2 connected node.

Children of cell[i][j] refers to the nodes connected to cell[i][j], signify that these nodes can be treated as reachable from cell[i][j]. Initially, 1 cell/node has at max 8 children.

Algorithm used:

The fact that each word can only contains each letter once can be incorporated by:

1. On the tree: Filter the dictionary and only use the words with unique letters for building the prefix tree. This will be grouped inside the prefix-tree building complexity.
2. On the board traversal: From each cell, before traversing to the children, compress all the child nodes with the same characters. Details included below.

For filtering, the algorithm is:

1. Use a k-sized array letterCount as a dictionary
2. Scan through the letter of each word
3. If letterCount[letter] = 0, mark letterCount[letter] = 1 (saw first time)
4. If letterCount[letter] = 1, then the letter is repeated, so terminate the algorithm and return false
5. If reach the end of the string without termination, return true

Since the algorithm needs O(k) to allocate the temp array, and O(d) to scan through all the characters, the total complexity of the checking is lower than the required scanning and initialize the single root for the prefix-tree, so doesn't affect the total complexity of tree building.

For **compression step** referred above, before traversing down the children, 4 main steps can be incorporated:

1. Split the children into groups sharing the same characters. By allocating an array of size k, with each position storing a linked list to store the node with same characters, then iterate through the children, putting them into the correct position (similar to counting sort), this takes $O(k + n^2)$, since there are max n^2 children (explained below).

2. Pick the first node from the group as a primaryNode (referred as primaryCell in my code), and compress these nodes by:
    a. Combine all the children of all nodes in the group into the children of the primaryNode:
        i. This involves a temp array of size (n * n), using as a hash set to check in O(1) if the children is already added to the primaryNode, to removing duplicates.
        ii. Then, scan through the children of the entire group, and put them into the primaryNode list of children (if not already in primaryNode).

            Since there are max n^2 node on the grid, there are max $8 * n^2$ children (for this algorithm, we never compress a group more than once, so all the nodes in the group still have their original children list, with at max 8 nodes). Thus, this takes $O(n^2)$.
    b. Remove all the edges/ links between nodes with the same group, since each word can only have non-repeated characters, so these links are unnecessary.
3. From the parent node's viewpoint, only process this single primaryNode instead of the entire group, since they are equivalent. The process, in this case, involves recursively traversing down the primaryNode with the new list of combined children.
4. After finished processing, "decompress" the primaryNode (by reverting its children list back to its original state, taking O(1) to re-construct the children list of the node, which have at max 8 children).

    This decompression is crucial for correctness. From the parent's view, since it can reach all children with the same path, the children with same characters only differ in their children, which is encompassed by the primaryNode. However, from other nodes (eg: the "brother" node), this might not be true, since for those nodes, the path leading from them to those grouped children might not be the same.

Initially, create a null node connecting to all nodes (cells) on the board, to ensure that even the choice to start the path is also compressed and grouped based on the characters by step 2, 3, 4. Fully optimized, all 4 steps take overall $O(k+ n^2) = O(n^2)$ (as $k <= n^2$, stated at the start), with the dominant operations being the grouping of characters.

The compression step is feasible, since the valid path/ word doesn't contain repeated characters, allowing the parent node to ignore all the edges/ potential path between the nodes with same characters and treat this group of nodes as a single node.

More importantly, with such compression, the overlapping children and grandchildren are fused into one. This will speed up the process since the surrounding children might share the exact same cells as their grandchildren, great-grandchildren and so on, causing the same cells to be processed repeatedly, growing exponentially as we go down the searching path. The grouping of the children ensures those overlapped cells are detected and treated as a single cell for each step down the path, thus ensuring each step only processing 1 cells at most 3 times.

The 2 sequences listed below of operations on the same cells at each step down the search path in case of worst overlapping (the extreme board given in post #606 on Ed) can help to show my point:

Original: 3 (step 1, 1 cells stored as 3) -> 9 (step 2, since each cell is overlapped 3 times by their parents, and their parents are overcounted 3x in previous step) -> 27 (step 3) -> ...

With grouping: 3 (step 1); group to 1 cell -> 3 (step 2); group to 1 cell -> 3 (step 3); group to 1 cell -> ...

This type of grouping is only feasible for path with unique characters, since we can safely ignore the links between children nodes with the same characters and the possibility that 1 path might need to go through multiple of them, thus treating them as a single node.

Note that, as stated above, the advantages arising from the reduction in overlapped operations on the exact same cells. If there are repeated values, but they are far away enough that their paths do not overlap at some point, then the algorithm is only as efficient as part A (also confirmed this in post #606 with Grady).

Besides the introduction of the 4 steps above, the same backtracking algorithm with cross-comparison with the prefix-tree similar to part A is used.

Worst case implications:

As explained above, unlike without constraints, the algorithm utilizing the constraints benefits from the repeated characters on the grid (which cut down the number of unique paths on the grid). Thus, the worst input for this is when the grid is filled with unique characters. As $k <= n^2$ as explained at the start, this basically means pushing $k = n^2$.

After the filtering, the remaining words will have max length $l_m <= k$, as it can only contain up to all letters in the k alphabet. This will also limit the possible letters arrangement, in the worst case, to all the permutation of k letters of length from 1 to k. Total unique string, thus, is:

$kPk + kP(k-1) + \cdots + k = k! + \frac{k!}{1!} + \frac{k!}{2!} + \cdots + \frac{k!}{(k-1)!} = k!\,(\sum_{i=0}^{k-1} \frac{1^i}{i!}) \approx k!\,e^1$, as k -> infinity

Space complexity: For the total unique string, note that all of the string with length < k are simply the prefix of the kPk set of strings. However, due to the current implementation suggested in the assignment, we still need to store an additional k-sized array for the single null pointer for each of them. Thus, the tree size ends up as O(k * k!).

During implementation, the board needs to be constructed as a graph. If implemented as an adjacency list, then it takes 8 * n^2, which is O(n^2). During compression, the total number of links change at most by a factor of 2 (since we only copies the links into the primary node once during compression, and since there is no repeated character in a path, no node contains more than one compression). Other tables (eg: visited, found, ...) might also be used to store flags/ info, but none of them has size larger than the grid or the prefix-tree. The total complexity is therefore **O(n^2 + k * k!)**

Time complexity:

For building prefix-tree:

The additional filtering, as explained above, contributes nothing to the overall complexity.

Basic operation: Allocate memory for the prefix-tree

For prefix-tree building: O(k * k!), as explained, this is the maximum size of the prefix-tree in the worst possible tree. The same caution apply as for original, since O(d) might become dominant if the dictionary contains multiple duplicated entries.

For matched word searching:

Basic operation: Character comparison (between the cell's and the prefix-tree's character, as well as during the grouping of nodes with the same character).

As shown above, there are O(n^2) basic operations required for the compression/ decompression and comparison between the current cell character and the prefix-tree. Thus, it takes O(n^2) for each step down the paths in the prefix-tree and grid (whichever is shorter).

O(n^2) is a loose bound. For example, with the assumption of unique characters, there are no possible compression, thus, except the initial null node, the maximum number of children node is 8, making the additional computation only O(1) per character. Still, since

this operation is input-sensitive, I'd take the cautious, absolute worst performance as the bounds.

For each cell[i][j], the upper bound for the number of paths is 7^k as there are at most 7 viable directions at any position in the path (ignoring 1 direction coming back to the previous cell, which will be terminated immediately), and we need to pick these directions k times for the path with max length k (equivalent to the min(max height of the prefix-tree, all cells in the grid), assuming $k = n^2$).

Use all the unique paths on the grid, that is also in the dictionary to build a prefix tree.

Let n_{unique} = number of characters in such prefix tree

$n_{unique} \leq number\ of\ unique\ characters\ in\ the\ dictionary\ prefix - tree \leq k * k!$

$n_{unique} \leq upper\ bound\ of\ number\ character\ in\ all\ paths\ in\ the\ grid \leq k * 7^k$

Thus, with the constraint of the algorithm, and the compression given, the worst time complexity is below, since we process $O(n^2)$ operations for each character in such tree:

$O(n^2 * n_{unique}) <= O(n^2 * k * \min(k!, 7^k)) = O(n^2 * k * 7^k)$ as k -> infinity (since $k! > 7^k$)

Notice that the bound is essentially similar to part A. This is because in the absolute worst input with a board filled with unique characters, there are no repeated characters to take advantage of, and the 2 algorithms yield similar results. However, with repeated characters (that also "clusters" relatively, so that path can overlap), as shown above, the 2nd algorithm reduces repeated, unnecessary operations significantly. Details of comparison are given in the following section.

OVERALL COMPARISON:

Note: The advantage is compared between the upper bound big O, which is technically not a good metrics, since having a tighter upper bound doesn't necessarily mean the algorithm is asymptotically better. Still, I tried to make the bound the tightest (still not θ because the algorithm is input-sensitive, so quite hard to find θ), thus having a tighter upper bound likely indicate a better algorithm.

The 2nd version discussed above is clearly input-sensitive, since we are utilizing repeated characters on the grid to speed up the process. Obviously, if the grid only has unique characters, all possible paths are unique and the 2nd version still traverses all the paths available, making it equally or less efficient than the 1st version, due to the overhead

incurred for scanning through the children to check if able to compress. Still, this is easily overcome by simply running an $O(k + n^2) = O(n^2)$ through the grid first to check if there are repeated values (using an k-sized array to count). If yes, then use the 2nd version, otherwise use the first version. The additional check contributes nothing compared to the massive complexity during the actual search. Such hybrid approach would ensure the 2nd version performs equally or more efficiently than the 1st version.

For the cases when the 2nd version is significantly better than the 1st version due to it's being more efficient and also because the input is more constrained:

1. When k << max length of the string in the dictionary: This will be handled by the filtering initially, capping the length to only k and only path with unique characters, which restricting available paths to traverse.
2. When the number of unique path in the grid or dictionary < 7^k: Since the edited version traverses the number of unique path in the grid or dictionary, this can happen with limited unique string/ path in the dictionary itself.
3. Most importantly, when there are repeated characters (clustering relatively together), which optimizes on the exponential growth of repeated operations on the exact same cells compared to 1st version, ax explained above.

Due to it being input-sensitive, the best method would be to practically compare the average runtime between the 2. For theoretical analysis, the 2nd version is at least as efficient as the 1st version, with the potential advantages in cases as listed above.