Question 1:

Assumptions:

- An anagram is valid in this case if the word can be found the English dictionary. The dictionary corpus used is from WordNet. Names of people for example, are not valid anagrams.

- Any input with special characters or numbers is not a valid anagram or string.

- Multi-word/sentence anagrams are not allowed. Only single word inputs are allowed.

Data Structures:

- The Python Counter subclass from the collections module is used. Counter is basically comprised of a Python dictionary data structure of a set of unordered key-value pairs of data.

- This allows us to store characters as keys and the number of times they appear in a string as values.

- The property of dictionaries allows us to compare them by taking the intersection of the two, giving us the positive minimums. Then, this can be compared with the substring anagram for validity that characters in the anagram is indeed found in the main string. For example:

Main string = {a: 1, b: 2, c: 1}

Anagram = {a: 1, b: 1}

Output of intersection = {a: 1, b: 1}

In this case, Anagram does equal to Intersection, thus, this is a valid anagram substring.

- Regular Python dictionaries can also be used in place of Counter, but Counter needs less lines of code since it has a built-in iterator counter.

- The Counter objects are also assigned its own variable to reduce compute resource wastage by recalculating the Counter. For example:

c = Counter(s)

d = Counter(t)

c & d == d

Uses less resources than:

Counter(s) & Counter(t) == Counter(t)

As Counter(t) needed 2 calculations.

Efficiency:

- The code requires an iterative count of all keys in the dictionary, thus, the code iterates over each dictionary once.

- Worse case, the code must iterate over both dictionaries for long strings once, where both strings are of equal length, and the runtime is hence O(2n)

- However, for an approximate case, this can be written as O(n), or linear time.

- Space efficiency is also linear, O(n), as the required variables to be stored are the two Counter objects.

Question 2:

Assumptions:

- String can take in any alphanumeric, special characters and whitespaces, it will find for a valid palindrome regardless.

- Palindromes are not checked for the English dictionary validation. Thus, any combination of characters which structurally form a palindrome is valid. For example, "aaabbaaa" is a valid palindrome.

- Punctuation such as (,.!?:-) and whitespaces are removed while processing the string, but special characters are maintained. This is because in some cases these are ignored.

Data Structure:

- Two lists: "a" - To iterate each character over the string, "r" - To store any found palindromes.

- Recursion for palindrome checker over window spans for a character in string, and iteratively over each character in string.

- Why recursion: To allow for cleaner code readability.

- A depth-first search is performed at each character in string "a".

- To reduce iteration time, first and last index of the string is ignored, as we focus on finding a palindrome starting with either 2 (for even palindromes) or 3 (for odd palindromes) characters long. For example: In the string "reviver", we want to start our search windows from "rev", "evi", "viv", "ive" and "ver". This has 5 iterations over the string, whereas if we were to include the first and last index, we would have 2 more iterations which would yield no valid palindromes.

Efficiency:

- The algorithm finds palindromes iteratively over string "a", then for each character in "a", the algorithm finds the valid palindromes recursively, increasing the window spread of adjacent characters to the left and right.

- This needs to run twice, once to find odd palindromes and twice to find even palindromes. Thus, the runtime is O(n^2), where we need to iterate over string "a" twice.

- Space efficiency: Stores list of palindromes, and stores recursion of window spans in temporary memory. This expands as the string "a" is longer. Thus, it is O(n).

Question 3:

Assumptions:

- We consider only a single graph as the input for every run. There are no multiple graphs for the input.

- All edges must have a weight value.

- Vertices must be unique strings.

Data Structure:

- A priority queue is implemented with a Python dictionary. The priority queue is used to find the minimum weight of an edge in all neighboring vertices.

- Another dictionary is used to grow the MST as the algorithm runs iteratively.

- Dictionaries are used as it is easy to retrieve the key or value pair, compared to using a list of tuples.

- Prim's algorithm is implemented, as we assume to only have one graph for the input, where as other algorithms namely Kruskal's are suitable for multiple tree graphs in a forest.

Efficiency:

- The algorithm iterates over the priority queue when it is not empty. It has to iterate over the queue to find the vertex with the minimum priority and also to iterate over each neighboring vertex to check the edge weights. Thus, this algorithm requires a linear time with increasing number of vertices (V) and edges (E)and in log time for increasing length of the priority queue. O((V + E)log V) = O(E log V)

- Space efficiency: The growing MST and priority queue are stored in dictionaries and will increase linearly with more complex graphs, thus, the space requirement is O(n)

Question 4:

Assumptions:

- Both nodes are in the tree

- The tree itself adheres to all BST properties

Data Structure:

- A Python dictionary is created to map the relationship of a node to its parent node in a tree. By doing this, we can retrieve the parent of a node easily for traversal down the tree.

- The main logic checks if the nodes n1 and n2 are larger than or smaller than the root node. By adhering to Binary Search Tree properties, all child nodes to the right of the parent node has a larger value than the parent, while all child nodes to the left have smaller values than the parent.

- When traversing down the tree, the algorithm checks for the appropriate tree branch to search deeper based on this BST property.

Efficiency:

- The node-parent map needs only to be iterated over once, when building the path from the matrix. Then, when traversing down the tree, it is done only once as the target nodes (n1, n2) are checked while traversing until a LCA is found. If the target nodes are on one half of the tree, then the algorithm will only search for the correct half. One level down the node, the algorithm will split its search into half again. Thus, the time complexity is maintained at log time O(log n).

- Space increases with increasing tree depth to store the key-value pairs of node-parent relationships, but no additional copies of the dictionary is required when traversing down the tree, thus, the space complexity can be approximated to constant time O(1).

Question 5:

Assumptions:

- Input is a valid singly linked list. The function does not check for doubly linked lists.

- Linked list elements can be of any data type (integer, character, string).

Data Structure:

- A linked list is used and any operations are based on it.

- To get the m'th node from the end of the list, the length of the list is first determined. Then, the position is calculated with (length - m). Finally, the list is traversed through forward until the calculated position is found. That will be the m'th element from the end of the list.

- The reason why this method is implemented is because singly linked lists do not have any references backwards; only the next node in the list is known, not the previous node. Thus, the total length of the list helps to find the forward-traversing position.

Efficiency:

- The function iterates through the linked list once to get the length, and another time to find the m'th node.

- This is technically O(2n), but it can be simplified to O(n). A linear increase in the linked list length will increase the search time linearly.

- As for space complexity, we are storing a few items: the linked list, a counter variable for length of list, and a counter for the m'th node search function. This is linear O(n) as well.