Question 1: Explanation

In Question1, the function takes two strings (s and t) and finds out whether one string (t) is anagram of the other (s). Here, the the string t and words in string s convert to dictionaries. The dictionaries contain the characters of the string as keys and their count as values. Then the dictionaries of t and words of s compare each other. If the two dictionaries are matching each other, the t is an anagram of s.

The efficiency of the program depends on how long the given strings are (O(len(s)). Since, each dictionary can have maximum of 26 letters the space efficiency is still O(1).

Question 2: Explanation

In this function, the longest palindromic substring is found in a given string. To achieve this, a substring is divided in to two from the center and check whether one part is matching to reverse of the other.

The overall efficiency of this function also depends on the length of the string (O(len(s)). Minimum number of variables used to increase the space efficacy. The space efficiency is O(1). Using isOdd() function, code repetition is minimized. Also, to minimize the unnecessary looping, following string reverse method is used.

second\_part = substring[center:][::-1]

These measures helped to improve time efficiency.

Question 3: Explanation

In this question, the minimum spanning path for a given graph is determined. First the dictionary form of the graph is converted to Graph object. The Graph object contains the Node and Edge objects. Several potential minimum spanning paths are determined in the graph. Among them the minimum path is extracted using the totals of the edge values (lowest values) from each possible path that contain all the nodes.

The time taken to run the program is depend on how many nodes and edges are in the graphs. The high numbers of nodes and edges can be increase the computational time exponentially (typical for graphs).

To increase the time efficacy, some iterations are minimized by using inbuilt methods like in following code line. Depth first search algorithm used to find paths.

path[::-1] ## To reverse a list

Several functions are used to minimize the code repetition.

Following data structures used to define Graph object.

class GNode(object):

def \_\_init\_\_(self, value):

self.value = value

self.edges = []

self.visited = False

class Edge(object):

def \_\_init\_\_(self, value, node\_from, node\_to):

self.value = value

self.node\_from = node\_from

self.node\_to = node\_to

class Graph(object):

def \_\_init\_\_(self, nodes=None, edges=None):

self.nodes = nodes or []

self.edges = edges or []

self.node\_names = []

self.\_node\_map = {}

Question 4: Explanation

This function finds the least common ancestor for given two nodes. the least common ancestor is the farthest node from the root that is an ancestor of both nodes. Th the beginning of the function, the binary search tree (BST) in a matrix form is converted in to BST object. The following codes shows the definition of BST object.

class BNode(object):

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

class BST(object):

def \_\_init\_\_(self, root):

self.root = BNode(root)

Inherently, BST data structures are easy to navigate. Based on the BST rules, the best case happens if the following conditions are met.

if (n1 < r and n2 > r) or (n2 < r and n1 > r)

In this case, there is no need to search through the BST, because the least common ancestor for the given numbers is the root. Hence, to increase the efficiency of the program, above conditions is checked before implementing other tasks. If the conditions are true function returns the root as least common ancestor and complete the run. In this question also, several local functions and an in-built function (find\_path\_to\_helper) to BST class are used to minimize the code repetition. In the worst-case scenario, the time efficiency depend on number of nodes (O(len(T)). The space efficiency stays almost constant for any given situation (O(1)).

Question 5: Explanation

The function question5() is written to find an element in a singly linked list that's m elements from the end.

Following data objects were designed to store data.

class LLNode(object):

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class LinkedList(object):

def \_\_init\_\_(self, head=None):

self.head = head

The function get\_length() is designed to get the length of a linked list before start to find the m th element from the end. Getting the length of the linked list first is making easier and efficient for the latter implementation such as navigation to the specific value of the linked list. Specially in this particular case, navigate to a specific number from the end of the linked list can be efficiently done when the length of the list is known. In the worst case scenario the efficiency can be calculated as O(len(linked list). Since, navigation happens to each noted, the space efficiency is O(1).