Explanations

Question 1

Given two strings s and t, determine whether some anagram of t is a substring of s. For example: if s = "udacity" and t = "ad", then the function returns True. Your function definition should look like: question1(s, t) and return a boolean True or False.

Solution:

Count frequency of characters in t. For each substring in s with same length as t, check if the frequency of characters equals frequency of characters of t.

Consider m the length of and n the length of t. To get the char frequency in t, we need to iterate once through all chars, O(n).

For each substring in s which has same length as t, we also need to iterate once through all chars to count char frequency, and another time to compare the char frequency of the substring with char frequency of t.

The number of substrings is m-n-1 - so the runtime is O((m-n)*2n), simplified to O((m-n)*n)

Space requirements is O(2n), or O(n), since we only need to store the char frequency of t and the current substring of s which is being compared to.

The solution using char frequency is more effective compared to the original solution, using permutations to get all possible anagrams of t, since it is executed in linear time. Running time for the test samples using permutations equals 0.0361, while the second approach using char frequency runs the same tests in 0.0002, two order of magnitudes lower.

Question 2

Given a string a, find the longest palindromic substring contained in a. Your function definition should look like question2(a), and return a string.

Solution:

Get all substrings in a, in order, ordered from shortest to longest. Loop through each one; if the substring is a palindrome, set it as the longest palindrome.

To get all substrings, we have to loop through the collection once for every element in the collection (collection here is the string which we will traverse to find the substrings). But there is catch. As we move down the string, we don't need to loop through the entire collection, only through the elements that are in fornt of it. That is O((n^2)/2), or O(n^2).

There is also a sort step, that takes linear time O(n log(n)). So total time would be O(n^2 + nlog(n)). Space requirement is the requirement to store the substrings, which is also O((n^2)/2), or O(n^2), linear.

Question 3

Given an undirected graph G, find the minimum spanning tree within G. A minimum spanning tree connects all vertices in a graph with the smallest possible total weight of edges. Your function should take in and return an adjacency list structured like this:

{'A': [('B', 2)], 'B': [('A', 2), ('C', 5)], 'C': [('B', 5)]} Vertices are represented as unique strings. The function definition should be question3(G)

Solution:

This solution is similar to a travel salesman problem, finding the shortest distance between two nodes. There are several optmized solutions for this problem, but the most simple one is to calculate the possible paths that connects all vertices and evaluating which one is shorter. A more optimized solution would be to use dynamic programming.

The brute force solution is a graph traversal of all possible paths, which is implemented below.

To get all possible paths, we need to traverse the entire graph starting from every node. The time then is O((n+v)^2), where v is the number of vertices in the graph and n the number of nodes.

There is also a sort involved in the paths found. Let m be the number of paths found, the time complexity to traverse is O(mlog(m)). If we approximate to state there is one path for each node, then total time is O((n+v)^2 + nlog(n)).

Space requirement is to store each path. The path is a combination of nodes and vertexs, and assuming there is one path for each node, space complexity is O((n+v)*n)

Question 4

Find the least common ancestor between two nodes on a binary search tree. The least common ancestor is the farthest node from the root that is an ancestor of both nodes. For example, the root is a common ancestor of all nodes on the tree, but if both nodes are descendents of the root's left child, then that left child might be the lowest common ancestor. You can assume that both nodes are in the tree, and the tree itself adheres to all BST properties. The function definition should look like question4(T, r, n1, n2), where T is the tree represented as a matrix, where the index of the list is equal to the integer stored in that node and a 1 represents a child node, r is a non-negative integer representing the root, and n1 and n2 are non-negative integers representing the two nodes in no particular order. For example, one test case might be

question4([[0, 1, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 1], [0, 0, 0, 0, 0]], 3, 1, 4) and the answer would be 3.

Solution:

Reviewed Solution: Traverse the tree from the root. If one of the nodes matches root, return root as least common ancestor.

If the left node (input node with the smallest value) is less than root, and right node (input node with highest value) is greater than root, than return root as least common ancestor.

If both left and right node are greather than root, set root to the right-most node descendant of root, and iterate. If both left and right node are smaller than root, set root to the left-most node descendant of root, and iterate.

This solution is based on two premises:

- It is a balanced tree
- Both n1 and n2 are in the tree

It takes O(h) to traverse the tree once, being h the height of the tree, which equals n/2 for a binary tree in worst case scenario. At each step we also need to linearly traverse the row to find the next node, which is also length n, which is O(n) in worst case scenario. So run time is O(n*h).

Space requirement is constant O(1), as there is no need to keep track of the paths visited.

Question 5

Find the element in a singly linked list that's m elements from the end. For example, if a linked list has 5 elements, the 3rd element from the end is the 3rd element. The function definition should look like question5(ll, m), where ll is the first node of a linked list and m is the "mth number from the end". You should copy/paste the Node class below to use as a representation of a node in the linked list. Return the value of the node at that position.

Solution:

A straighforward solution is to get the length of the linked list, and substract m from the length to the get element position in the list. After getting the position traverse the linked list again (length-m) times to get to the element.

To get the length of the linked list, we need to traverse it entirely. So time complexity is O(n). We then need to traverse it again, to find the element. Worst case scenario is also O(n). So complexity is O(n^2).

Space complexity is constant time, O(1), since we only need to store the position and the value of the node.