|  |
| --- |
|  |
|  | 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. |
|  |  |
|  | First, we design a function to calculate the unique hash for each string. We keep a sliding window of length t and use the function to calculate the hash of both s and t. If they are the same we have found an anagram. If not we move the sliding window forward one character at a time. This new substring adds one new character and loses one character compared to the previous substring. We update the hash map based on the current substring inside the window and compare it to the hash map of t. |
|  |  |
|  | We have to iterate through the string at least once. So the time complexity of this algorithm is O(len(s)). The space complexity of this algorithm is constant and hence O(1). |
|  |  |
|  | 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. |
|  |  |
|  | For each character in the string the program checks for the character before and after that character (in the case of odd palindrome). If they don't match it stops looking. However, if it does match it continues looking to the left and the right. In the worst case, if the string is a single repeated character this process might be repeated for all the characters since each substring is a palindrome. So, the time complexity of this algorithm is O((len(a))^2). This program does not require any extra space, hence the space complexity of this algorithm is O(1). |
|  |  |
|  | 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) |
|  |  |
|  | I used Kruskal's algorithm to solve this problem. |
|  | 1. Sort all the edges in increasing order of their weight. |
|  | 2. Iterate through each edge starting from the one with the smallest weight (Greedy Algorithm). |
|  | 3. If it does not form a cycle include it, else discard it. |
|  | To detect cycles, I used a disjoint-set data structure (Union&Find) to keep track of which vertices are in which subsets. |
|  |  |
|  | The default sorting of edges in Python takes O(ElogE). We iterate through each edge once O(E) and apply find-union algorithm which can take at most O(logV). Hence the time complexity of this algorithm is O(ElogE) + O(ElogV). The space complexity of this algorithm is O(E + V). |
|  |  |
|  | E is the number of edges in the graph and V is the number of vertices. |
|  |  |
|  | 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. |
|  |  |
|  | I have used the properties of Binary Search Tree (BST) to recursively traverse the tree. If the root is greater than both the nodes, then we look to the left of the root for LCA (Least Common Ancestor). If the root is smaller than both the nodes look to the right of the root for LCA. If the root is in the middle of two nodes, then the root is the LCA. |
|  |  |
|  | The time complexity of above solution is O(h) where h is the height of the tree. Also, the above function makes recursive calls hence requires O(h) extra space. |
|  |  |
|  | 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. |
|  |  |
|  | class Node(object): |
|  | def \_\_init\_\_(self, data): |
|  | self.data = data |
|  | self.next = None |
|  |  |
|  | The above solution implements a technique called linked list traversal in which we systematically visit each node. We start a pointer at the first node in the list and as we visit each node, we move the pointer to the next node. The worst case time complexity of the above solution is O(len(ll)). In the worst case, we have to traverse all the way to the end of the singly linked list and hence the space complexity is O(len(ll)). |
|  |  |