# Question1

To determine whether some anagram of string t is a substring of string s, I search s to find all characters in t. For simplicity, I covert both strings to lists. Since s is not order the runtime for each search is O(n) where n is the length of the string s and in total the runtime is O(nm) where m is the length of the string t. when I find a character in s, save its location to check if string t is a substring of s. The run time for this step is O(m) and space efficiency is O(m).

# Question2

To find a palindromic substring, we check the left and right character of a character. If they are the same, there is a palindromic substring and we need to check the next left and right character to increase the length of the palindromic substring if possible. The runtime in the best case is O(n) where n is the length of the string. In the worst case where the length of the palindromic identical to the length of the string the runtime is O(n2). The code saves all palindromic substrings and returns the length of the longest palindromic substring. Therefore, the space efficiency is O(m) where m is the number of the substrings.

# Question3

First, built an edge list, containing two nodes and the weight. The list is sorted to find the edges with largest weight. Then, the edge with largest weight is removed from the graph. This step is continue as long as the graph is connected. Due to complexity of the code, it is not easy to calculate the runtime accurately. The function which check the connection of the graph is the most expensive part of the code. If a graph has n nodes the recursive function must run at least (n-1) times to check the connections. If the graph has m edges (m>n) we need to remove m-n+1 edges. Since each time we remove an edge we have to check the connection, we have to run the recursive function (n-1)(m-n+1). Using recursive function is also not efficient with space since each time we call the function we allocate new space. My algorithm is not a good solution for large graph with many edges. An easy way to improve the performance is to start to build a new graph by connecting the nodes with edges with small weights.

# Question4

I assumed each node has maximum two children at left and right hand side. The left child has a value smaller than its parent while the right child has a value greater than its parent. Since the tree is in matrix form, the elements in the lower triangle represent left children and the elements in the upper triangle characterize the right children. To find the children for a given node with value I, we need to stand on the element at row I and column I (diagonal of the matrix). Searching elements in the row I with columns smaller than I leads to find the left child. The minimum runtime is O(0) where I=0 (first row), and the maximum runtime is O(n) where I= number of nodes (last row). In average the runtime for finding left child is O(n/2) where n is the number of nodes. The same scenario applies for the right child, therefore the average runtime for finding children is O(n). To find the distance or path between root and a node, I start from root and find the left or right child, depends on the value of the node and root. Then I consider the child as a new root or parent, and I jump to the corresponding row. I repeat searching until I find a given node. The runtime depends on the height of the tree O(h) and since we have to calculate the distance of two nodes from root the total runtime is O(2h).

# Question5

First, I calculate the length of the link list which has the runtime O(n) where n is the length of the link list. Then, I subtract m from the length of the link list to find the location of element. Finally, I find the element based on its location. The total runtime is O(2n) and space is O(1).