Problem 1: LRU Cache

The LRU cache consists of 2 parts, a dictionary and a linked list. The linked list tracks the head and tail of the linked list and each node in the list points to the previous and next nodes. This linked list is used to track the usage of each node where the least used node will be nearest to the head. The dictionary takes the key and uses it to find the corresponding node in the linked list. Dictionary references are O(1) in time complexity. The combination of these 2 data structures allows us access the nodes and their relative positions in O(1) time complexity.

When the set function is called, it adds a new node to the tail of the linked list and includes the reference in the dictionary. If there is a duplicate key and node, the set function will remove it before adding the new entry. The set function also removes the oldest node and the corresponding dictionary entry if the cache limit is reached.

The get function is similar to the set function. The key is used to retrieve the node and then the node is moved to the back of the linked list as it is most recently accessed.

Overall, all operations are O(1) in time complexity. Space complexity of the set function is O(M) where M is the capacity defined for the size of the cache. Space complexity for the get function is O(1) as it uses the predefined memory.

Problem 2: File Recursion

A recursive method is used to iterate through the file paths and directories. A preorder DFS is used as there was no requirement to the order in which the files and paths must be visited. Additionally, a preorder DFS is a simpler implementation.

The program runs through all the files and directories once and checks if the files have the indicated extension, hence the time complexity is O(n) where n is the number of files and paths. Space complexity is dependent on the folder path where the greater number sub folders within folders, the more space will be consumed by the program to run this recursive function, therefore it can be estimated as O(m) where m is the number of subfolders.

Problem 3: Huffman Coding

To create the Huffman encoding, the data is first sorted into a dictionary where the key is the character, and the value is the frequency of occurrence of that character. A dictionary was used here we can access the frequency value and change it without having to search through the other items, such as with a list. The dictionary is then converted and sorted into an ordered list to be used to build the Huffman tree. The program goes through each of the characters in “data” only once, hence the time complexity here is O(n) where n is the number of characters in “data”.

The encoding function then builds the tree and traverses the tree to create a new dictionary with the character and their corresponding encoded string. This dictionary is used to convert the characters in “data” to the encoded string.

In the decoding function, the 0 and 1 in the encoded string is used to traverse the tree. The characters are found when the traversal reaches a leaf node. Across both encoding and decoding functions, the data items are only traversed once each item in various parts. As we are mainly concerned with the order of time complexity, the time complexity will only be O(n).

The space complexity is O(m) where m is the number of characters in data. This is because throughout the program, we are just storing the characters of data in different forms and with information about the characters.

Problem 4: Active Directory

This problem is similar the problem 2, so similarly, a recursive function/approach in the form of a preorder DFS is used. In this case, however, the advantage with using a preorder DFS is that we can return when the user is found, and we do not need to complete the search of all sub-groups.

The time complexity at worst case is when the user is in the very last subgroup, where the time complexity is O(N) for N number of subgroups. Space complexity is also O(M) where M represents the number of sub folders as we go through each sub-group to find the user.

Problem 5: Blockchain

The implementation of a blockchain is a linked list of nodes where each block is contained within a single node. This is a singly linked list where each node is only linked to the previous node, as what is required with the blockchain. The linked list tracks the tail as well, so we can append new block nodes without traversing the entire linked list.

The time complexity is O(1) since the function is essentially taking the inputs and transforming it into a node to be added to the chain. The space complexity is also O(1) because it takes the same amount of space each time a new block is created.

Problem 6: Union and Intersection

In this problem, the provided lists are converted into sets. Sets are useful here as they have inbuilt union and intersection methods. Also sets cannot have duplicate values. In the program, the linked lists are converted into sets and they the union and intersections of the lists are returned.

Time complexity is O(N+M) where the N and M are the sizes of the linked lists, which we traverse through once each time to convert them into sets. Likewise, space complexity is also O(N+M) since depending on the size, the sets that they are converted to would be of similar size.