Problem 1:

I chose to use an Ordered Dictionary to solve this problem. At first, I implemented everything except keeping track of the least-recently used item using a regular dictionary. I looked for a way to order a dictionary and came across this data structure, which is a built-in implementation of a Doubly-Linked List.

get() – time complexity O(1), space complexity O(1)

set() - time complexity O(1), space complexity O(1)

Overall worst time & space complexity: **O(1)**

Problem 2:

The time complexity of the find_files method is O(n), where n is the total number of files and folders under a directory, assuming that file names and suffix name lengths are all bounded by a constant. Otherwise, it is O(n*m) where m is the limit on the file name length. The space complexity is O(n).

Problem 3:

find_freq() – time complexity O(n), space complexity O(k), where k is the number of distinct characters
queue_setup() – time complexity O(n), space complexity O(k)
encode() – time complexity O(n), space complexity O(k)
huffman_encoding_helper() – time complexity O(n), space complexity O(k)
huffman_decoding() – time complexity O(length of encoded data), or O(nlogk), where n is the length of the string of characters, and k is the number of distinct characters. The space complexity is O(n).

The overall worst time complexity is **O(nlogk)**, and the worst space complexity is **O(n)**.

Problem 4:

The time complexity of is_user_in_group() is O(the total number of users in the group & all of its subgroups). The space complexity is O(1).

Problem 5:

append() – time complexity O(1), space complexity O(1)
get_timestamp() – time complexity O(1), space complexity O(1)
get_chain() – time complexity O(n), space complexity O(n)

Overall worst time & space complexity: **O(n)**

Problem 6:

union() – time complexity O(n), space complexity O(n)
intersection() – time complexity O(n), space complexity O(n)

Overall worst time & space complexity: **O(n)**