Problem 1: Square Root of an Integer

To find the square root of a target number without using any inbuilt functions, I used a binary search of numbers between 0 and the target number, and checked if the square of the searched number is equal to the original number. In the case where the square root of a number is not an integer, the algorithm would look for the floor value by returning the searched number that is the highest integer that is lower than the square root of the target number.

A binary search is executed in O(nlogn) time which fits the question's requirements.

Problem 2 Search in a Rotated Sorted Array

In this problem, my approach is to first find the point at which the array is sorted using a binary search. The array is split at this point and then another binary search is used on the subarray to find the input number.

The algorithm uses 2 binary searches. The first binary search has time complexity O(nlogn) and the second binary search has the same time complexity in the worst case, where it needs to search through the entire array. Hence, the total time complexity will be O(2nlogn), which is estimated as O(nlogn).

Problem 3 Rearrange Array Digits

To create the largest sum of 2 numbers from an array of elements, I first sort the array of elements and then I append the digits together. This would put the digits into the numbers in descending order and would ensure larger digits are placed in positions with greater significance. The sum of these 2 numbers would hence yield the largest sum.

To guarantee that the answer has time complexity of O(nlogn), I implemented a merge sort that has time complexity O(nlogn) in the best case, average case and worst case scenarios. The assignment of the digits to their numbers goes through the array once and is hence O(n) time complexity, which is less significant than merge sort, hence overall time complexity remains at O(nlogn).

Problem 4 Dutch National Flag Problem

This solution was referenced to the course materials. The solution uses 2 pointers at the start and at the end of the list to track the next positions that a 0 or a 2 should be. We then iterate through the array using the front index and if we find a 0 or a 2, we put them in their next positions given by the pointers. 0 will line up the start of the list whereas 2 will line up the end. The remaining 1s will automatically be moved in between. We will traverse the array only 1 time.

The worst case is where there are few or there are no 2s and we must traverse the entire array. Hence, time complexity is O(n).

Problem 5 Autocomplete with Tries

The Trie contains Trienodes that are linked together to store words. Each Trienode contains a dictionary that uses a character as the key and returns the next Trienode if it exists. The Trienode also stores a Boolean value to tell if a word is formed upon reaching the node. A recursive function is also implemented within the Trienode to allow for all the suffixes thereafter that Trienode to be retrieved.

In the Trie, the insert function iterates through each character of the word once and are hence O(n) in time complexity. The find function is also O(n*m) in time complexity where m represents the number of keys that are stored at each layer. The suffixes function of the Trienode is a recursive function and it depends on the number of word suffixes that are linked to it. Therefore, the time complexity of suffixes function is O(m) is the number of words with suffixes linked to that Trienode.

Problem 6 Unsorted Integer Array

2 variables are used to track the min and max values. These numbers are updated while traversing the list of ints. The traversal only needs to happen once to get the min and max values, hence, the time complexity of this solution is O(n). There is no need to sort the list when using this method.

Problem 7 Request Routing in a Web Server with a Trie

A RouteTrie is implemented in a similar way to problem 5. Instead of characters, the RouteTrieNode takes a string as a key and returns the respective node. The Router then wraps the RouteTrie. When the Router is initiated, it creates 2 handlers for the main RouteTrie and for another RouteTrie handler to handler when a path does not exist. The splitpath function is a helper function that decomposes a path into a list of strings that represent that path. This method is O(n) in time complexity with respect to the length of the path.

Similarly, the add_handler function is time complexity O(n) based on the size of the path. For the lookup function, the time complexity is O(n*m) where m is the number of possible paths (represented by the keys of the node’s dictionary) at each node.