[9 MARKS]

This question is about running time. In parts (b) and (c), you are asked to explain your answer. We are looking for 2 to 4 clear and concise sentences that make it clear to us that you understand how to reason about the runtime of algorithms.

Type your answers into the file we have provided called **Q5_answers.txt**. Use PyCharm to open that file and add your answers to it. Hand in this file on MarkUs.

Part (a) [3 MARKS]

Suppose we have binary search tree methods to accomplish the tasks below, and that each uses a sensible algorithm.

What is the worst-case big-oh time complexity of each of these methods, assuming it is called on a completely balanced tree containing n nodes? (In a completely balanced tree, every internal node has two subtrees, and all leaves are on the same level at the bottom.)

1. Find the range of values in the tree (the difference between the largest and smallest value)
2. Set the value in every node of the tree to 0
3. Find the sum of all values at depth ≤ 3

Part (b) [2 MARKS]

Suppose we have a linked list class that has two instance attributes: a reference to the first node, and an integer storing the length of the list. Among its methods are `__getitem__`, `__len__`, and a method to insert a new item at a given index. Assume each method uses a sensible algorithm.

Suppose we are writing a client function to go outside the class, that will take a linked list and return a copy of it. It will have a loop that iterates over the indices from `len(lst) - 1` down to 0. For each index, it will use the `__getitem__` method to get the item at that index from the original linked list and the `insert` method to insert that item into the new list at index 0.

What is the worst-case big-oh time complexity of this implementation of the copy function for a linked list of n items? Explain your answer.

Part (c) [4 MARKS]

Consider these two implementations of a function that is equivalent to the `WordLadderPuzzle.extensions` method you were asked to implement in A2 this term.

```
with open("words") as f:
    WORD_SET = set([w.strip() for w in f.readlines()])  # A set of valid English words.

CHARS = 'abcdefghijklmnopqrstuvwxyz'  # All of the lowercase letters.

# helper used to determine if two strings are the same length and differ by a single letter
def is_one_away(s1: str, s2: str) -> bool:
    if len(s1) != len(s2):
        return False
    num_diff = 0
    for i in range(len(s1)):
        if s1[i] != s2[i]:
            num_diff += 1
    return num_diff == 1

def extensions_v1(cur_word: str) -> List[str]:
    exts = []
    for word in WORD_SET:
        if is_one_away(cur_word, word):
            exts.append(word)
    return exts

def extensions_v2(cur_word: str) -> List[str]:
    exts = []
    for c in CHARS:
        for i in range(len(cur_word)):
            if c != cur_word[i]:
                word_to_try = cur_word[:i] + c + cur_word[i + 1:]
                if word_to_try in WORD_SET:
                    exts.append(word_to_try)
    return exts
```

The running times of these functions may depend on some or all of these values:

- s: the length of set `WORD_SET`
- w: the length of word `cur_word`
- c: the length of `CHARS`

Note: Both checking if an object is in a `set` and determining the length of a string are constant time operations. Remember that slicing a string always makes a new string.

1. Give a big-oh expression for the worst case running time of `extensions_v1`. Explain your answer.
2. Give a big-oh expression for the worst case running time of `extensions_v2`. Explain your answer.