1 Written Problems

Problem 10.1

Binary Search Tree Review

In a binary search tree, \( T \), in-order traversal will visit the nodes in increasing-key order, thus providing a linear-time mechanism for extracting all keys in order.

Suppose that you have some range of keys, say \( k_1 \leq k \leq k_2 \), and you’d like to extract all keys in \( T \) that lie within this range, and you’d like to get the results, as before, in increasing-key order. (Note that neither \( k_1 \) nor \( k_2 \) needs to actually be a key in the BST, and that the returned list of keys might well be empty!) Once again, inorder traversal with a test for in-range-ness will solve the problem in worst case \( O(n) \) time, where \( n \) is the number of keys in the tree.

You are to describe how to solve the problem in time \( O(h + s) \), where \( h \) is the height of the tree \( T \), and \( s \) is the number of keys returned.

- Give a brief description of your approach to this problem.
- Write pseudocode for your algorithm.
- Explain briefly why your algorithm is \( O(h + s) \).

Solution:

Using what we know about the order of binary search trees we can avoid searching sections of the tree that aren’t within the \( k_1, k_2 \) range. If a node is less than our \( k_1 \) key then we know all nodes to the left of this node will also be smaller and therefore don’t need to be visited. Also, if a node value is higher than \( k_2 \) we know we don’t have to visit any children to the right of that node. Using this we can traverse the tree with a modified in-order traversal in which we only visit the left child if the current node is greater than \( k_1 \) and only visit the right child if the current node is less than \( k_2 \).
function `traverseRange(tree, k1, k2)`
   """ Transforms: Tree, int, int -> array of TreeNodes
   Purpose: calls the `traversalHelper` to find and
   add nodes with keys in a given range (k1<k2) to
   the 'results' array
   """
   result = []
   if root is not null:
       traversalHelper(tree.root(), k1, k2, result)
   return result

function `traversalHelper(current, k1, k2, result)`
   """ Transforms: TreeNode, int, int, array -> nothing
   Purpose: checks left and right children for range,
   traversing the graph in modified in-order.
   'result' may be modified
   """
   leftChild = current.leftChild()
   rightChild = current.rightChild()
   if leftChild is not null and current.key > k1:
       traversalHelper(leftChild, k1, k2, result)
   if current.key >= k1 and current.key <= k2:
       result.add(current.key)
   if rightChild is not null and current.key < k2:
       traversalHelper(rightChild, k1, k2, result)

This algorithm runs in $O(h + s)$ because you may need to travel down the
entire height (h) of the tree, and you must visit every node (s) between k1 and
k2 exactly once.

Problem 10.2

Dictionary Segmentation

Given an input string and a dictionary of words, develop an efficient algorithm
to determine if the input string can be segmented into a space-separated se-
quence of dictionary words. There is no limit to the number of times you may
use each word in the dictionary.

For example, suppose we have the following dictionary of known words:

```
dictionary = {"i", "like", "ice", "cream", "icecream", "popsicle"}
```
Consider the following input/output pairs.

Input: "ilike"
Output: True
We can segment the string into "i like" using words from the dictionary.

Input: "ilikeicecream"
Output: True
We can represent the string as either "i like icecream" or "i like ice cream."

Input: "ieatpopsicles"
Output: False
We can’t segment the string into words contained in our dictionary.

Note that a greedy approach will not always guarantee the correct solution! For example, consider the following test case:

```
dictionary = {'a', 'an', 'ill', 'pill', 'pillow', 'case'}
Input: "pillowcase"
Output: True
We can segment the string into "pillow case."
```

In this case, if we tried to find the first matching prefix, we would pick “pill.” However, that makes the rest of the string ("owcase") unsegmentable given our dictionary.

Write pseudocode for your algorithm.

**Solution:**

The key to solving this problem efficiently is determining what our subproblems are. The question of whether a given string can be segmented into dictionary-recognized words can be split into the problem of whether the string has a combination of *substrings* that can be segmented into dictionary-recognized words.

Our first instinct may be to approach this problem recursively. The idea is to consider each possible prefix and search for it in the dictionary. If the prefix is present in the dictionary, we recur for the rest of the string (the suffix). If the recursive call for the suffix returns true, we return true. Otherwise we try the next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.

The pseudocode for the recursive solution is as follows:

```
Note that in our pseudocode, we are using python-like indexing for slicing strings: string[0:i] indicates the substring of string up to and including the i – 1th character.
```
# Input: string to be segmented, dict (hashset) of recognized words
# Output: boolean
# Purpose: returns whether or not a string can be segmented
# into a space-separated sequence of dictionary words

function word_break(string, dict):
    size = len(string)
    if size is 0:
        return true

    # For each possible prefix
    for i = 1 to i <= size:
        if dict.contains(string[0:i]) and word_break(string[i:size], dict):
            return true

    return false

While the recursive implementation is straightforward, it is not the most efficient solution. The issue is that we are computing the results of several subproblems more than once. Ideally, we would save ourselves from doing repeated work by storing the results of these subproblems.

**Dynamic Programming Solution**

# Input: string to be segmented, dict (hashset) of recognized words
# Output: boolean
# Purpose: returns whether or not a string can be segmented
# into a space-separated sequence of dictionary words

function word_break(string, dict):
    size = len(string)
    if size is 0:
        return true

    # Make array to store results of subproblems: wordbreak[i] stores
    # if the substring string[i:size] can be segmented or not
    wordbreak = array of length size+1
    initialize all values of wordbreak as false
    set wordbreak[size] = true

    for i = size-1 to i >= 0:
        for j = i to j < size:
            if dict.contains(string[i:j+1]) and wordbreak[j+1] = true:
                wordbreak[i] = true
                break
return wordbreak[0]