Problem Set 3

Problem 3-1. [0 points] **Instructions**

All parts are due on March 11, 2020 at 10PM. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct. Solutions should be submitted on Gradescope, and any code should be submitted for automated checking on `https://alg.mit.edu`.

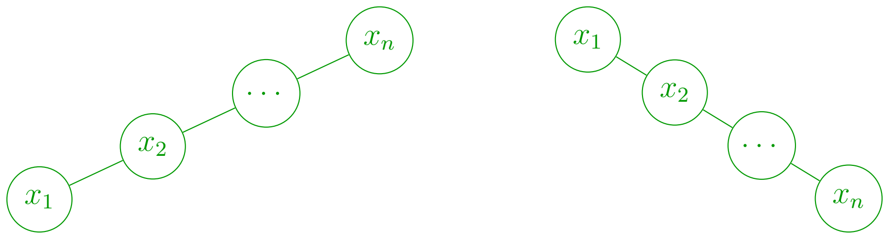
Please abide by the collaboration policy as stated in the course information handout. Note that the first two problems are *Individual* problems and collaboration on these problems is not allowed.

Problem 3-2. [15 points] **Individual: Binary Search Trees**

(1) [4 points] Starting with an empty tree, and n elements $x_1 < x_2 < \cdots < x_n$, describe an insertion order that would result in a BST of height $n-1$. What is the time complexity of performing all n inserts, following the INSERT method described in class?

Solution:

Either insert all the keys in decreasing order (obtaining the picture on the left), or in increasing order (obtaining the picture on the right). The time complexity of all n inserts is $\sum_{i=0}^{n-1} i = O(n^2)$. This is because the i-th item is inserted at depth $i-1$.

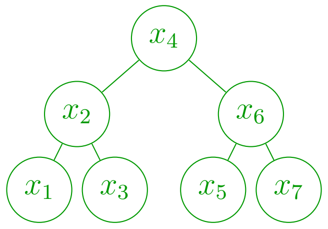


Rubric:

- 2 points for a correct insertion process
- 2 points for a correct time complexity calculation

(2) [3 points] Starting with an empty tree, and 7 elements $x_1 < x_2 < \cdots < x_7$, describe an insertion order that would result in a BST of height 2 (remember that a complete tree of height 2 has 3 levels and 7 nodes). Draw the final tree obtained (you can upload a picture of a drawing).

Solution: We should insert first the middle element x_4. Then we could insert x_2 and x_6 in any order and, finally, x_1, x_3, x_5 and x_7 in any order. The tree could look as follows.



Rubric:

- 2 points for a correct order
- 1 points for a correct tree

(3) [3 points] Given 15 distinct elements $x_1 < x_2 < \cdots < x_{15}$, how many binary search trees of height 3 can be made? Explain how you got your answer.

Solution: Only one binary search tree can be made, as there is only one binary search tree of height 4 (the full tree) that can accommodate all 15 elements.

Rubric:

- 1 points for a correct solution
- 2 point for correct reasoning

(4) [5 points] Given 7 distinct elements $x_1 < x_2 < \cdots < x_7$, how many different insertion orders can produce a binary tree of height 2? Explain your reasoning.

Solution: There are 80 possible orderings. x_4 must come first. The next element must be either x_2 or x_6. WLOG, say it is x_2. Now, the children of x_2, which are x_1 and x_3, either both come before x_6 (4 options since then there are two ways to order x_1 and x_3, and two ways to order $x5$ and x_7), both come after x_6 (24 options since there are $4!$ ways of ordering x_1, x_3, x_5, x_7), or one comes before and the other comes after (12 options since there are two choices for which leaf comes before x_6, and then 3! = 6 ways to order the remaining leaves). This means there are 40 possible orderings if x_2 comes after x_4. If x_6 comes after, this case is symmetric and we get another 40 orderings for a total of 80 possible insertion orders.

Alternatively, think of the five slots after x_4, x_2 (again, wlog. x_2 comes right after x_4). We need to choose 2 of these for x_1 and x_3, and there are $\binom{5}{2}$ ways of doing so. Then there are 2 ways of ordering x_1 and x_3 and two ways of ordering the remaining 3, as x_6 should precede the other two. We get $10 \times 2 \times 2 = 40$ as before.

Rubric:

- 2 points for a correct solution
- 3 points for correct reasoning

Problem 3-3. [25 points] **Individual: Mirrored BST**

The Evil Queen is a big fan of her magic mirror. Such a fan, in fact, that she turned your favorite binary search tree into a *mirrored binary search tree*. In a mirrored binary search tree, keys in a node's left subtree are greater than the key stored at the node, and keys in a node's right subtree are less than the key at the node. Although the structures are essentially equivalent, you would like to set things back to normal by reversing her evil plot.

Given a mirrored binary search tree T, describe an $O(n)$ time algorithm that rearranges the nodes of T such that T becomes a regular binary search tree. You can assume that the input to the function (call it UNMIRROR) is the root node of the tree and that, for a tree node x, you have access to its children via $x.left$ and $x.right$ (these fields will store the value `None` if the child does not exist).

Please describe your algorithm, and prove its correctness and time complexity.

Solution:

```
procedure UNMIRROR($x$)
    if $x$ == None then
        return
    else
        UNMIRROR($x.left$)
        UNMIRROR($x.right$)
        SWAP($x.left$, $x.right$)
        return
```

Call UNMIRROR($T.root$). This call takes $O(n)$ time. To see why this is true, observe that each node in the tree is the argument to the UNMIRROR function exactly once. Furthermore, the number of calls to UNMIRROR where $x ==$ `None` is $O(n)$ since each node can have at most two children pointers that point to `None`. Without counting the recursive calls, there is $O(1)$ work done inside UNMIRROR to swap the left and right subtrees. Therefore, the total time is $O(n)$.

The correctness can be proved inductively. The base case is a subtree with one node, and UNMIRROR will not affect it (since in this case a normal BST and a mirrored BST are the same). Now assume that calls to UNMIRROR on subtrees with fewer than n nodes correctly unmirror the subtrees. For a tree with n nodes, both the left and right subtrees of the root can be unmirrored correctly by the inductive hypothesis. Since we know everything in the left subtree is greater than the key at the root, and everything in the right subtree is less than the key at the root (by definition of mirrored BSTs), if we make the left subtree the new right subtree and vice versa we will end up back in regular BST order. Thus, the tree is unmirrored correctly.

Rubric:

- 4 points for correct base case in the algorithm.
- 8 points for correct inductive step in the algorithm.
- 5 points for running time justification.

- 8 points for correctness proof.

Problem 3-4. [30 points] **Pizza Problems**

Maria and her friends have started the Mushroom Pizza Parlor, but are running into a variety of sorting problems. For each of the following scenarios, determine which one of the following sorting algorithms { insertion sort, merge sort, counting sort, tuple sort, or radix sort } would be the most efficient for sorting the given n items, and state whether the asymptotic complexity would be $\Theta(n)$, $\Theta(n \log n)$, or $\Theta(n^2)$. Briefly justify your answers.

(1) [6 points] Maria receives an order for n 14-inch diameter pizzas (of many different types) with a special request for the pizzas to be delivered sorted in order from least to most pieces of pepperoni. Maria always uses pepperoni pieces of exactly 1-inch in diameter and she always arranges them so that they do not overlap.

Solution: Since the area of one pizza is $\pi(14/2)^2$ and each pepperoni slice covers an area of $\pi(1/2)^2$, we can upper-bound the maximum number of pepperoni slices per pizza by $\frac{\pi(14/2)^2}{\pi(1/2)^2} = 49 \times 4 = 196$.

Because the sorting key comes from a constant-size set, we use **counting sort** which runs in $\Theta(n)$ time.

Rubric:

- 2 point for identifying counting (or radix) sort
- 1 point for $\Theta(n)$ (or $O(n)$) complexity
- 3 points for justification
- 2 point (total) for correct $\Theta(n \log n)$ algorithm

(2) [6 points] Maria's pal Pete likes to test new shipments of peppers. Given n boxes of peppers, he likes to keep the 90% most spicy boxes (and compost the rest). Pete compares the spiciness of peppers two boxes at a time by doing a blind taste test.

Solution: We must use a comparison based sorting algorithm here, e.g. **merge sort** in $\Theta(n \log n)$ time. Once sorted, it is easy to find the top 90% spicy boxes.

Note that there are comparison based algorithms that find the 90% precentile element in a list in $O(n)$ time. You will learn about these in 6.046.

Rubric:

- 2 point for identifying merge sort (or another efficient comparison sorting algorithm or selection algorithm)
- 1 point for $\Theta(n \log n)$ (or $O(n \log n)$) complexity
- 3 points for justification
- 2 point (total) for correct $\Theta(n^2)$ algorithm

(3) [6 points] Maria's cousin Waria handles the online curbside pick-up orders. Each batch is a list of n requests (p_i, t_i) for pizza type p_i to be picked-up at a unique time t_i (a date and time of day, down to the nearest minute). Each type of pizza p takes a known duration $dur(p)$ to cook, ranging from 15 minutes to 1 hour (rounded to the nearest minute). Initially, the batch of requests is ordered by pick-up time. Help Waria rearrange the requests so that they are sorted by the time the chefs should start cooking the pizzas to meet the requested pick-up times.

Solution: Since there is at most one request per minute, each request can only be out-of-order with respect to $\Theta(1)$ nearby requests in the list. Thus we can use **insertion sort** with keys $t_i - dur(p_i)$, which will only need to do $\Theta(n)$ work.

Rubric:

- 2 point for identifying insertion sort
- 1 point for $\Theta(n)$ (or $O(n)$) complexity
- 3 points for justification
- 2 point (total) for a correct $\Theta(n \log n)$ algorithm

(4) [6 points] Maria's sister Loise wants to create a daily high-score leaderboard for the arcade machine in the parlor. Each of the n people who play the game achieves an integer score with $\lceil \log_\pi n \rceil$ decimal digits.

Solution: The range of scores is $\Theta(10^{\log_\pi n}) = \Theta(n^{\log_\pi 10})$, which is polynomial, so we can use **radix sort** in $\Theta(n)$ time.

Rubric:

- 1 point for identifying radix sort
- 1 point for $\Theta(n)$ (or $O(n)$) complexity
- 4 points for justification
- 1 point (total) for correct $\Theta(n \log n)$ algorithm

(5) [6 points] Last fall, Maria's roommate Roshi planted a row of n saplings in the parlor's flower bed that have sprouted flowers of different hues on the rainbow color spectrum, and now he wants to rearrange the flowers into rainbow order. Unfortunately, Roshi only has one pot to hold a dug-up plant (the rest must stay planted in the ground), and he can only visually compare the hues of two flowers in front of him if one is next to the other in the ground or in his pot.

Solution: With only one pot, we need an in-place sorting algorithm, so Roshi should use **insertion sort** which will only do adjacent comparisons and swaps and take $\Theta(n^2)$ time.

Rubric:

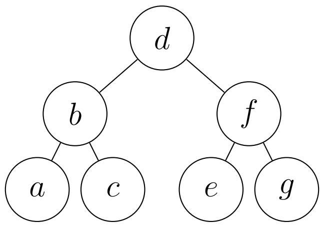
- 2 point for identifying (in-place) insertion sort (or sorting virtually and then arranging correctly)
- 1 point for $\Theta(n^2)$ complexity
- 3 points for justification
- 1 points (total) for otherwise correct algorithm that uses too many pots

Problem 3-5. [30 points] **Constructing Tree Traversals**

Unlike the data structures we've seen previously in this class (Direct Access Arrays, Linked Lists) which have only one way to traverse the elements, Binary Trees have multiple. In this problem you will be introduced to three different ways to traverse a Binary Tree; the `InOrder`, `PreOrder` and `PostOrder` traversals. These three traversals are all recursive traversals, calling themselves on the smaller subtrees of the original tree, and are defined for a tree `T` as follows:

```
InOrder(T) = InOrder(T.left), T.val, InOrder(T.right)
PreOrder(T) = T.val, PreOrder(T.left), PreOrder(T.right)
PostOrder(T) = PostOrder(T.left), PostOrder(T.right), T.val
```

For example, for the following tree T:



The `InOrder` traversal would be `[a, b, c, d, e, f, g]`, the `PostOrder` traversal would be `[a, c, b, e, g, f, d]`, and the `PreOrder` traversal would be `[d, b, a, c, f, e, g]`.

For this problem, assume that all node values are distinct.

(1) [2 points] A tree T has `InOrder(T) = [h, i, j, k, l, m]` and `PostOrder(T) = [h, k, m, l, j, i]`. What is `PreOrder(T)`?

Solution: `PreOrder(T) = [i, h, j, l, k, m]`

Rubric:

- 2 points for the correct preorder traversal

(2) [13 points] Assuming unique node values, given `InOrder` and `PostOrder` traversals, there is a unique tree T that gives raise to those traversals. Therefore, the `InOrder` and `PostOrder` traversals, as a pair, determine the `PreOrder` traversal! Give a worst-case $\Theta(n^2)$ algorithm to determine the `PreOrder` traversal given the `InOrder` and `PostOrder` traversals of a tree T. Note that the correctness of the algorithm *proves* that, indeed, the `PreOrder` sequence is determined by the `InOrder` and `PostOrder` sequences. Remember, to show a runtime of $\Theta(n^2)$, you need to analyze the complexity for all inputs for the $O(n^2)$ side, and exhibit a bad input for the $\Omega(n^2)$ side.

Solution: We know that the root is always the last item in the `PostOrder` traversal. Next, we know that the `InOrder` traversal has the structure `Left_SubTree,` `Root, Right_SubTree`, so we can search the `InOrder` for the root. We know that the left subtree contains all nodes to the left of the root in the `InOrder` traversal and the right subtree contains all nodes to its right. sImilarly, all left nodes appear before any of the right nodes in the `PostPorder`. So, once we find the index of the root in the `InOrder` traversal, we can also split at this index in the `PostOrder` to have that split into left and right subtrees.

Once we have the traversals split into the two subtrees, we recurse and run the same procedure again, first with the left subtree, and then with the right subtree.

We then form the `PreOrder` traversal by placing first the root, then the `PreOrder` of the left subtree and finally the `PreOrder` of the right subtree.

Correctness: We are correctly obtaining the root node in each recursive call as the root node will always be the last item in the `PostOrder` traversal. We are also splitting at the correct node to determine left and right subtrees since each node value is unique.

Runtime: The worst case runtime is $O(n^2)$ since each node will be the root exactly once, and each time a node is the root we have to search for it in the `InOrder`, which takes at most $O(n)$ time (the length of the list of nodes we have to search through). We are able to show this bound is tight by considering the case where the `InOrder` and `PostOrder` are the same, making a tree that is a chain of left children. This exhibits worst case behavior since each time we have to search for the root we must iterate through all of the remaining nodes in the traversal, giving a runtime of $\sum_{i=0}^{n-1}(n-i) = \Theta(n^2)$.

Rubric:

- 7 points for the correct algorithm
- 3 points for arguing correctness
- 3 points for correct analysis of runtime

Partial credit can be given.

(3) [5 points] We now want to speed up the runtime of our `PreOrder` traversal computation. One piece of your algorithm above should be pretty slow as is, but could be greatly spread up with the help of hashing. Give an *average* O(n) time algorithm to compute the `PreOrder` traversal given the `InOrder` and `PostOrder` traversals. Don't forget to analyze the runtime of your algorithm.

Solution: At the start of the algorithm, create a dictionary that maps values in the `InOrder` traversal to the index that the value is at. This takes $O(n)$ time, and then lets us get expected $O(1)$ time to search for the index of any node. We then run the same algorithm as above, replacing the slow $O(n)$ search for root in `InOrder` with a quick lookup in our dictionary, taking our runtime down to average $O(n)$ time.

One note, when doing recursive calls, we need to keep track of start and end indices for the two traversals rather than splitting the traversal lists. This is both to save time in not having to create new lists, and also so that the indices dictionary will still map to the correct locations.

Rubric:

- 3 points for correctly using hashing to search for the root in the inorder traversal
- 2 points for correctly arguing the average $O(n)$ runtime.

(4) [10 points] Implement your `PreOrder` traversal construction from part 3. For the purpose of this exercise, you can assume that the cost of concatenating lists in Python is O(1). Download a template and submit your code at `https://alg.mit.edu/spring21/PS3`.

Please submit a screenshot of your report that includes your `alg.mit.edu` student ID (top right corner) and the percentage of tests passed to Gradescope.

Solution:

```python

import sys
sys.setrecursionlimit(10000)
def construct_preorder_traversal(inorder, postorder):
    """
    Args:
        inorder (list): the inorder traversal of the tree
        postorder (list): the postorder traversal of the tree

    Output:
        list: the preorder traversal of the tree.
    """
    # Runs in expected O(n) time assuming fast list concat (linked lists),
    # O(nlogn) with pythons list implementation.

    inorder_indices = {}
    for i, value in enumerate(inorder):
        inorder_indices[value] = i

    return preorder_recursive(inorder, postorder, 0, len(inorder) - 1,
                              0, len(postorder) - 1, inorder_indices)


def preorder_recursive(inorder, postorder, inorder_start, inorder_end,
                       postorder_start, postorder_end, inorder_indices):
    if inorder_start > inorder_end:
      return []
    if inorder_start == inorder_end:
        return [inorder[inorder_start]]

    root = postorder[postorder_end]
    root_index = inorder_indices[root]

    left_subtree = preorder_recursive(inorder, postorder, inorder_start,
                                      root_index - 1, postorder_start,
                                      root_index - 1 - inorder_start + postorder_start,
                                      inorder_indices)

    right_subtree = preorder_recursive(inorder, postorder, root_index + 1, inorder_end,
                                       root_index - inorder_start + postorder_start,
                                       postorder_end - 1, inorder_indices)

    return  [root] + left_subtree + right_subtree
```

Rubric:

- This part is automatically graded at `alg.mit.edu`.