COMP2123 - Assignment 3

UID: 500557727

Problem 1.

a) My friend's algorithm is unfortunately wrong as it doesn't efficiently assign heroes to dragons. This algorithm will fail by assigning heroes with high determinations to dragons whose assigned heroes' sums of p is almost d. For example, if H in non-increasing order was {3, 2, 1, 1}, and if D in non-increasing order was {4, 3}, this algorithm would return false since it assigns the heroes with p-values of 3 and 2 to the dragon of d-value 4, making it impossible to persuade the dragon with d-value 3. However, this should return true since we can simply assign the hero with p-value 3 to the dragon with d-value 3, and the rest to the dragon with d-value 4 (and other configurations are also possible).

b) HeapAlgorithm will indeed always return the correct answer as it maximises the utility of each hero's p-value at each iteration. By adding the difference between the dragon's d-value and the hero's p-value to the dragon heap and only comparing the maximums of each heap at once, the algorithm ensures that each hero's persuasion is used to diminish as much of the dragons' total determination as possible. I can prove this by induction.

In the base case, where |H| = |D| = 1, the maximum of the hero heap will be the only element in set H (p), and the maximum of the dragon heap will be the only element in set D (d). Where A = d – p, A is inserted into the dragon heap and if A >= 1, then the single dragon still has some determination and the algorithm returns false as per line 8. Otherwise, the hero prevails and the algorithm returns true.

As the sizes of H and D grow, we can assume that the algorithm has computed the correct answer for some k, k <= |H| - 1 of the heroes. For the k + 1th hero, we know that their determination is the highest determination left of the hero heap. Once we calculate A = d – p and that is added to the dragon heap, we can continue iterating if there are no more heroes to assign, or check if the maximum determination left in the dragon heap is greater than 0. Either way, we've assigned the k+1th hero to a dragon/dragon remainder that has the closest relative d-value to its p-value and make sure that each hero uses its persuasion level as much as possible, making for an efficient allocation and correct results.

Problem 2.

a) For this problem, I will implement a self-balancing AVL tree to maintain a tree height of logn. This will take O(n) space and since each operation will traverse the height of the tree, they will each take O(n) time. Compared to the slides' AVL trees, I will tweak the insert and remove operations to save the height of each node's left children 'left_height' and the height of its right_children 'right_height' to calculate its imbalance. I will also save the size of the subtree at their nodes to calculate the smallest missing in range.

Each node of the tree will have its card number as its key, and its right_height, left_height, imbalance, and subtree size 'size' as instance variables. As we insert and remove cards, we will update and check the imbalance and sizes of the parent nodes so that we can maintain the AVL property (that the heights of any two children of a specific node can only differ by at most 1) and implement trinode restructuring if the imbalance at a parent node is more than 1.

Then, to implement the 'smallestMissingInRange' operation, I will conduct a binary search from the root, checking the size of the subtree at both children to see which side of the node does not have the correct number of cards between k1 and the node's value for the left branch, and between k2 and the node's value for the right children. If a branch has less than the required size, then it's missing some cards in that range and we recurse into it.

- Insert(i): Since we can assume that the card will not have a duplicate in our set, we can perform a binary tree search by recursively checking node values for where to place the node as shown in the slides.

Next, to re-establish the AVL property, we should update the imbalances of the parents of node i tree and check for which ancestor of i now has children whose heights differ by more than 1. This will be achieved by recursing to the parents of node i and adding 1 to either its left_height or right_height depending on whether its child was its left child or right_child. Then I would reassign its imbalance to the absolute value of the difference between the new left_height and right_height.

After I update the imbalance, I perform trinode restructuring if it has an imbalance of greater than 1 to maintain the height of the tree as logn and hence maintain O(logn) time for all operations.

Next, I need to update the size of the subtree at each ancestor of i. This is simpler than recalculating imbalance: I simply need to recurse through the ancestors of i and add 1 to the size value of each ancestor.

- Delete(i): similar to insert, we should perform a binary tree search and when we find i, we should replace it with an empty node as in the slides. Then we check for an

ancestor of i that may now not have the AVL property by recalculating imbalance as described above, only rather than adding 1 to right_child or left_child if the child is the right or left child of its parent, we remove 1.

Similarly, we also need to update the size variable by removing one from each of the node's ancestors. Then we perform a trinode restructuring on i to re-establish the AVL property.

- smallestMissingInRange(k1, k2): I've designed this operation to use the size variable at each node to see which child we have to recurse into to find the lowest missing card value in the range. If the left child's size tells us that it doesn't have every single card until the node's value, then we don't have all the cards in that range and that branch is where the missing card value is. We do the opposite for the right children and that tells us that the missing card is in the right branch. We recurse over all the right and left children until we get to a node where the node is in the range k1 – 1, k2 and we know from the size of the tree that all the previous cards have been found, then we've found the right node and we return that node + 1.

We would've found the smallest card right before the smallest missing in range if we will find all the cards from k1 up to this node lower down in the tree. We can determine this by checking the size of the subtree and if v.left_child size – k = 0. If so, we can return that node's value + 1, i.e. the lowest card that's missing. If we've reached a node with no children and we haven't been able to find a node where the rest of the tree's size tells us that we've found cards from k1 to v.value, then the tree has all the cards in the given range and we return null.

```
def smallestMissingInRange(k1, k2):
    missing_card ← missingHelper(T.root, k1, k2)
    return missing_card

def missingHelper(v, k1, k2):
    #Boolean here checks that there are missing values in the subtree at v and that we're within the range
    if v.left_child exists and not v.left_child.size = v.value – 1 and v.value > k1 then
        missingHelper(v.left_child, k1, k2)

    if v.value in range k1 -1, k2 and v.left_child size – k = 0 then
        return v.value + 1
```

#Boolean here checks that there are missing values in the subtree at v and that we're within the range

if v.right_child exists and **not** v.right_child.size = v.value – 1 and v.value < k2 **then**

missingHelper(v.right_child, k1, k2)

b) For insert and remove, these must find the position of the element by recursing throughout the tree and performing trinode restructuring on i if its imbalance is greater than 1 (as proven in the lectures). They must also update the imbalance and size of the node's ancestors. To prove that these operations correctly update the imbalance and size variables when nodes are inserted and removed, I will use induction since they are similar.

To update imbalance, my base case will be when a single node is added/removed from the right child of the root. Originally the root will have right_height = left_height = imbalance = 0 and size = 1. My imbalance operations will find that the height of the left subtree is -1, and the size of the right subtree is 1 since it will add 1 to right_height since the new node is the right child of the root. So imbalance = abs(-1 -- 1) = 0. Similarly, size will add 1 to the size of the ancestor and the root will now have subtree size 2. Remove will also work similarly but it would subtract 1 where add would increase 1.

As more nodes are added and removed and n = k, we can assume that the imbalances and sizes will be calculated correctly. As n = k + 1 or n = k -1 (for remove and add), the update_imbalance and update_size methods will be called on each ancestor of i so right_height or left_height will be updated and size will be decremented or incremented. The key to these operations functioning is in the observation that a node insertion or removal only affects the node's ancestors, which these operations update.

My smallestMissingInRange operation makes visitations by checking the size of the subtree at each node. I will prove that this method finds the smallest missing card by contradiction. As we visit each node, we choose to either traverse to its right or left child depending on whether that child's subtree size indicates that it includes all the cards before it. We define a range k1-1 and k2-1 since we return k + 1 if we decide it's the smallest missing in range. We see if we've checked that all the lower cards in the range are there. If we are not at this node when these conditions are met, then that means that there are cards missing in the range lower than this card, meaning that an ancestor actually does have all previous cards below it, causing a contradiction. Hence, smallestMissingInRange finds the smallest missing card.

c) In terms of space, I only save n nodes where each node is a card from the n cards of our collection. Each node saves its value, left_height, right_height, imbalance, and size, so each node takes O(4) space so the n nodes take O(n) space.

– Insert(i): This method first recurses through the height of the tree to find this value's position by comparing against a node at that height. Since this is an AVL tree, the height is logn and inserting i takes O(logn) time as shown in the slides.

The next section is a recursion which updates the imbalance of each ancestor of i. This performs two O(1) checks and assignments: first check if the node was the right or left child of its ancestor, adding 1 to its left and right child depending. Second, it recalculates the imbalance and reassigns it to the node in O(1) time, amounting to O(1) time logn times for each ancestor of i, equating to O(logn).

Similarly, the size updating recurses once for each ancestor of i, recursing logn times. In each recursion, we make no checks and just increment and reassign 'size' to the node in O(1) time, all equating to O(logn) time.

Lastly, we perform trinode restructuring as we update imbalances which takes O(logn) time as shown in the slides.

Added up, O(logn) + O(logn) +, O(logn) + O(logn) = O(logn).

- Remove(i): this operation has the same time analysis as insert, since its only difference is that its reassignments differ by making the node null and restructuring the node depending on what case it is. These restructurings still take O(logn) time. Also, we simply decrement for the imbalance and size functions, amounting to O(logn) time.

- smallestMissingInRange(k1, k2): at each recursion, a node can go through a maximum of 3 O(1) Booleans (three if statements) and 1 O(1) recursion or assignment. These are to check if we should recurse to the left child, if we should return the node's value + 1, or to recurse to the right child. Since recursion occurs by moving down one level, it makes logn traversals for the height of the tree as each traversal is O(1) time, adding up to O(logn) time.

Problem 3.

a) We could calculate this in O(n^3) time if we use three for loops nested in each other, with the aim of calculating the sum of each combination of 2 coins from Y and F, and then comparing each sum against each element in V. First, we initialise a Boolean 'can_buy' as false which we return at the end. Our first for loop would iterate through all the coins 'my_coin' in Y, and then we'd iterate through all 'friend_coin' in set F. We would calculate a variable 'sum' which would be the addition of my_coin and friend_coin. We would then use another nested for loop to compare 'sum' to all the prices 'snack_cost' in V. If sum and snack_cost are equal, then we make can_buy true.

b) A simple O(n^2) solution would be to map each sum of coins in Y and F in a hashtable 'h_sums' of size n^3 (since that's more than the number of combinations of the coins there are: n^2). We then iterate over V and get() all the prices in V from h_sums and if that price exists, then one of our coins' sums is exactly equal to the price and we can buy our snack and we return that price. Otherwise, when we've finished iterating through V and we haven't had a collision, then we return null since we can't eat.

Our hash function is also relevant to this implementation. We can simply use h(k) = k mod n^3 where k is the sum of a coin in Y and a coin in F. We detect collisions by applying open addressing using linear probing. If we encounter collisions as we insert the n^2 combinations of our coins, then we can put() them in the hashtable by moving forward and putting it in the earliest available space as in the slides. For the comparison between the prices in V and the sums in the hashtable, I detect collisions by performing get() and if it does not return null, then my friend and I can buy from the vending machine.

c) I will prove that this functions as intended by contraposition: i.e. I need to prove that if there is no collision in h_sums when we try to insert the prices in V, then none of the prices and sums in Y and F are equal. This is evident from the hash function: if a price 'p' from V is inserted into h_sums, then if it is equal to a sum of a coin in Y and a coin in F, then it will be mapped to its same position in h_sums, thus causing a collision. This is because of the hash function … Otherwise, each p has been mapped to a unique position in h_sums and there have been no collisions, meaning no equal insertions.

d) First, we compute the sums the coins in Y and F using two nested for loops, each iteration performing one O(1) time sum of the coins, one O(1) time hashing of that sum, and one O(1) time insertion into the hashtable (expected). Therefore, each loop is in O(3) = O(1) time and the creation of the hashtable takes O(n^2) time since each loop runs n times for the number of elements in each array.

Next, we iterate over the prices in V n times, each time hashing that price in O(1) time, and then attempting to get() that value from h_sums in O(1) (expected) time. We perform an O(1) Boolean check if the value is not null. If that is true, then we return the result of the

get() operation in O(1) time. Otherwise, we continue checking each value in V and return null after the loop since we haven't yet found a match (O(1)). Thus each loop equates to O(3) = O(1). This equates to n iterations and 1 possible return statement, meaning checking the prices in V takes O(n) + O(1) = O(n) time.

In total, the algorithm is the addition of the above two parts: O(n^2) + O(n) = O(n^2) time.

e) Since I'm using open addressing, the load factor would always be n^2/n^3 = 1/n (n/N) as the n^2 sums of the coins in Y and F would always each find their own position in the hashtable. However, although each value is distinct, we could still put several values with the same key and we could have to traverse n elements to find a suitable spot, hence the worst case time of each put() operation is still O(n). So, when we add the coins in Y and F together and then insert them into the hashtable, that would be n^2 traversals in O(n^3) time.

When we try to compare each price in V against the elements in h_sums, as argued above, our get() operations can also each take O(n) time as we could have to traverse our entire hashtable to find an empty spot. So each loop for each of the n elements in V can take O(n) time, amounting to O(n^2) time.

Together, these add up to O(n^3) + O(n^2) = O(n^3) time.