Algorithms and Analysis - Assessment 2

Simon Karumbi
s3455453

Problem 1

a.
This algorithm computes the number of leaves in a binary tree. It achieves this recursively, visiting each node of the tree, starting at the root and traversing to the left and right branches of the tree . If the node that is currently being visited is null, it exits the algorithm adding 0 to the running total. If the node being visited is a valid, non-null node, 1 is added to the running total and the next two children are visited.

b.
This algorithm belongs to the brute force paradigm, as it is a depth first search approach. We are searching the leftmost leaves of the binary trees until we reach an empty node, and then repeat the process on the last successful node visited, on its right node instead of left.

c.

$$C(h) = \begin{cases} n = 0, & \text{-1} \\ n = 1, & 0 \\ n > 1, & 2\mathrm{C(n/2)+1} \end{cases}$$

d.
$C(h) = 2C(\frac{n}{2}) + 1$
$C(h) = 2[2C(\frac{n}{4}) + \frac{n}{2}] + 1$
$C(h) = 4C\frac{n}{4} + 1 + 1 \quad = 4C(\frac{n}{4}) + 2$
$C(h) = 4[2C(\frac{n}{8}) + \frac{n}{4}] + 2$
$C(h) = 8C(\frac{n}{8}) + 3$
$\text{therefore} \quad C(h) = 2^i(\frac{n}{2^i}) + i, \quad n = 2^i, \quad i = \log_2 n$
$\text{hence} \quad t(h) = C_{op}(\log n)$

e.
$C(h) = \mathcal{O}(\log n)$

Problem 2

a.

1. Create an empty array to store the ID's of people who will receive a reminder message
2. Select the first person that appears in the the `list_double` list
3. With that person's ID selected, compare it to each person's ID in the `list_triple` list
4. If you find the same ID, you can move on to the next person in the `list_double` list without performing any action

5. If you reach the end of `list_triple` without matching the ID's, add that person's ID to the array created in the first step
6. Continue this process with the next people in the `list_double` list until there are no more people left

As we are checking every single person in `list_double` and comparing them with almost every single person in `list_triple`, the time complexity is $\mathcal{O}(mn)$.

More accurately, in the worst case scenario every single person may have had their third dose, in which case the complexity is $\mathcal{O}(n^2)$

b.
For this equation, an assumption on the ID's is being made whereby the ID number being used is unique and comparable across the two lists of people. If this was not the case, for example, passport numbers and social security numbers are used and they are not comparable as passport numbers occasionally have letters are well as numbers, then a hashing algorithm could be utilised to the same effect.

1. Using the first input of the function, `list_double`, create a binary tree using the comparable ID value, inserting values starting from the root node.
2. For each new item in `list_double`, if the node is empty add the value, if there is a value and it is larger move to the left child node, and if there is a value and it is smaller move the right child node
3. Then, once the binary tree is completed proceed to compare values from `list_triple`, removing values from the binary tree if they are equal
4. The assumption here is that $m \epsilon n$, `m` is a subset of `n` and every value will be found within the binary search tree
5. Once you have finished going through `list_triple` you can return the resulting binary tree of people who are yet to get their third dose

Note: In order to further maximise the average time complexity for this algorithm, while creating the binary tree, you can 'heapify' it by creating a self balanced binary tree, whereby if the difference between the height of leaf nodes cannot be larger than one and the tree reshuffles if this is the case. This ensures that when searching for `list_triple` values, the time complexity for the second portion of the algorithm never exceeds $\mathcal{O} = \log m$

The average time complexity of this equation is $\mathcal{O} = n \log m$, where in the worst case you have a severely unbalanced binary tree as a result of the order of unique ID's, where the time complexity would be $\mathcal{O} = nm$ as the tree would essentially be a linked list. The space complexity is constant, as you are not using any loops or recursion, and no temporary objects are being used to store data outside of the binary tree created from the input.

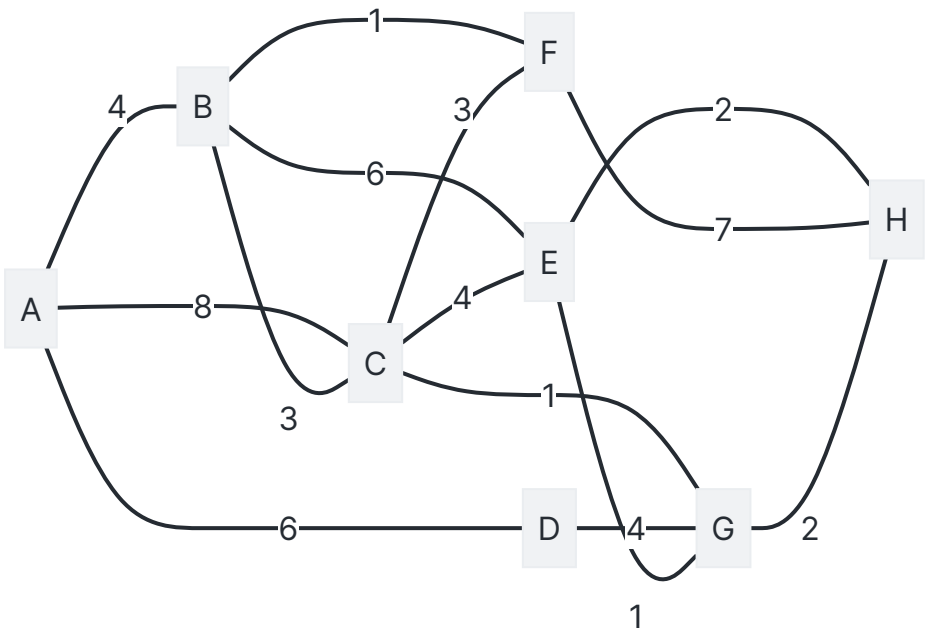
c.

1. Create a dictionary/ hashmap to add values
2. For every item in `list_double` add the value to the hashmap
3. For every item in `list_triple` search and remove the value from the list
4. The returned hashmap are the people who require a third dose

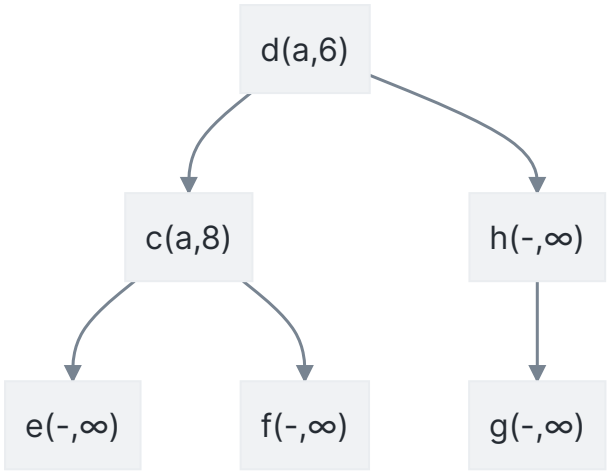
```
FUNCTION third_dose():
        INPUT list_double a list of people with two dose, list_triple a list of people
with three dose

        INIT a dictionary to store IDs

        FOR all values in list_double
                ADD value to dictionary
        ENDFOR

        FOR all values in list_triple
                REMOVE value from dictionary
        ENDFOR

        RETURN dictionary
END
```

The time complexity for this algorithm is determined by the two for loops, which are `n` and `m` complexity respectively. As dictionary inserts and deletes are constant time due to the hashing functions, $\mathcal{O}(1)$, the two for loops are sequential and not nested the total time complexity is $\mathcal{O}(m+n)$ which is bounded by the larger list, `n`. Therefore the time complexity at worst would be $\mathcal{O}(2n)$ or more simply, $\mathcal{O}(n)$.
In terms of space complexity, as we are looping over two input values of size `m` and `n`, as well as creating a dictionary to store values of `n`, the space complexity is $\mathcal{O}(m+n+n)$ which simplifies to $\mathcal{O}(n)$.

Problem 3

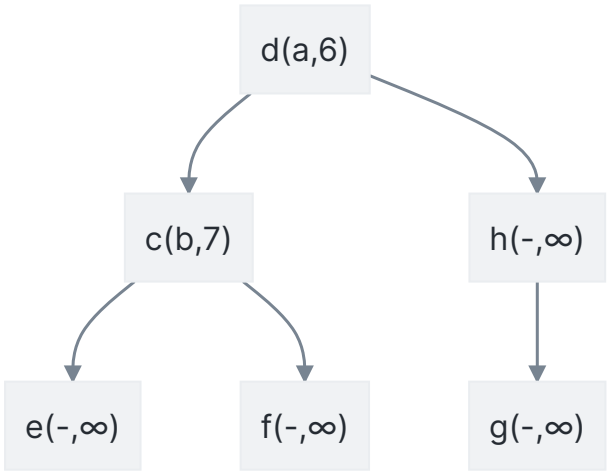


a.

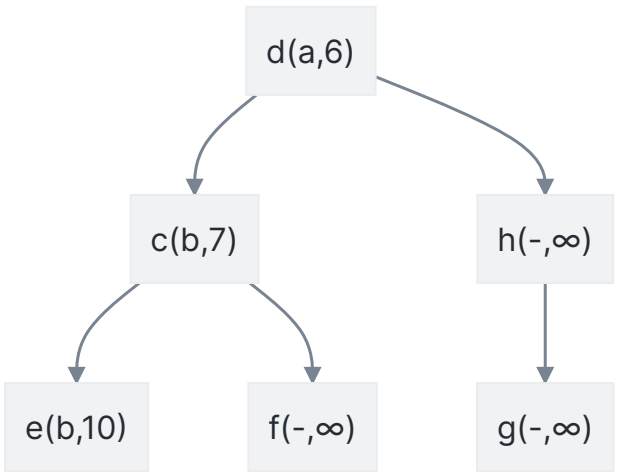


b.

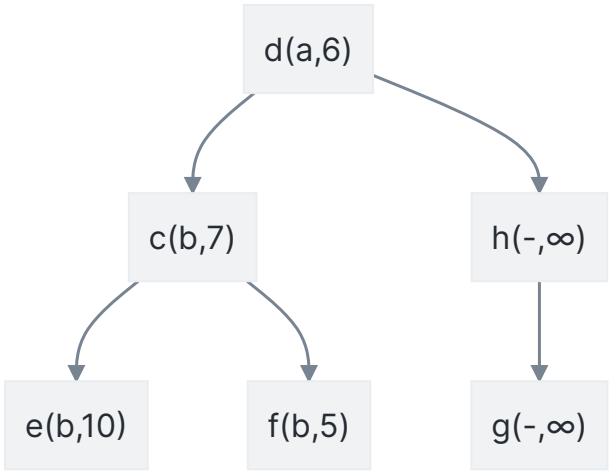
Step 1



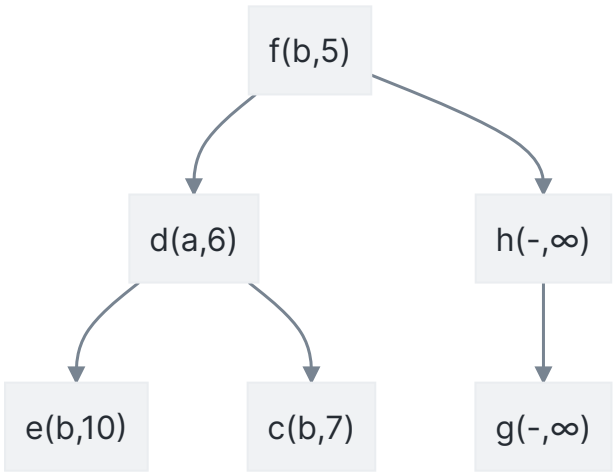
Step 2



Step 3



Step 4



c.

	S: vertices whose shortest paths have been known	Priority queue of remaining vertices
1	a(a,0)	b(a,4), c(a,8), d(a,6), e(-,∞), f(-,∞), g(-,∞), h(-,∞)
2	a(a,0), b(a,4)	c(b,7), d(a,6), e(b,10), f(b,5), g(-,∞), h(-,∞)
3	a(a,0), b(a,4), f(b,5)	c(b,7), d(a,6), e(b,10), g(d,10), h(f,12)
4	a(a,0), b(a,4), f(b,5), d(a,6)	c(b,7), e(b,10), g(d,10), h(f,12)
5	a(a,0), b(a,4), f(b,5), d(a,6), c(b,7)	e(b,10), g(c,9), h(f,12)
6	a(a,0), b(a,4), f(b,5), d(a,6), c(b,7), g(c,9)	e(b,10), h(g,11)
7	a(a,0), b(a,4), f(b,5), d(a,6), c(b,7), g(c,9), e(b,10)	h(g,11)
8	a(a,0), b(a,4), f(b,5), d(a,6), c(b,7), g(c,9), e(b,10), h(g,11)	

d.

	Shortest paths	Distances
a	a -> a	0
b	a -> b	4
c	a -> b -> c	7
d	a -> d	6
e	a -> b -> e	10
f	a -> b -> f	5

| g | a -> b -> c -> g | 9 |
| h | a -> b -> c -> g -> h | 11 |

Problem 4

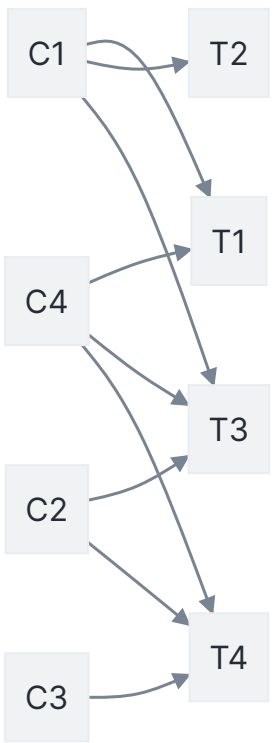
a.
We will test for unique solutions in this algorithm by first testing the proposer optimal solution for coins, and then comparing that with the result of the proposer optimal solution for transactions. We know that there exists a different unique stable solution for each of these cases if the algorithm has mapped values correctly. If not and both these solutions result in the same mapping, we can determine that this is a badly mixed structure.

We will use a greedy approach for this problem, whereby the coins and transactions will pick the highest available coin or transaction from their list. Let C_1 select its highest available pick T_n (the lowest index transaction) and continue to C_n. If there is no available pick remaining, choose the highest pick and remove it from the previous coin.

```
FUNCTION mapping(n, m, L, R):
        INPUT: n is number of coins/ transactions, m is number of edges, L is the
adjacency list for coins, R is adjacency list for transactions

        INIT dictionary to hold coins with transactions as keys
        INIT set of coins from R

        WHILE set of coins:
                FOR coin in set of coins:
                        FOR transaction in coin:
                                IF transaction is not in the dictionary THEN:
                                        ADD the transaction as a key, coin as value
                                        REMOVE the coin from the set
                                ENDIF

                                IF current coin does not have next transaction THEN:
                                        POP the coins first transaction from dictionary
                                        ADD the old coin back to the coins set
                                        ADD the current coins transaction as key and
coin as value
                                        REMOVE current coin from set of coins
                                ENDIF
                        ENDFOR
                ENDFOR
        ENDWHILE

        INIT dictionary of transactions
        INIT set of transactions from L

        WHILE set of transactions:
```

```
                FOR transaction in set of transactions:
                        FOR coin in transaction:
                                IF coin is not in the dictionary THEN:
                                        ADD the coin as a key, transaction as value
                                        REMOVE the transaction from the set
                                ENDIF

                                IF current transaction does not have next coin THEN:
                                        POP the transactions first coin from dictionary
                                        ADD the old transaction back to the transaction
set
                                        ADD the current transactions coin as key and
transaction as value
                                        REMOVE current transaction from set of
transactions
                                ENDIF
                        ENDFOR
                ENDFOR
        ENDWHILE

        COMPARE dictionary of coins and dictionary of transactions

        IF dictionaries are equivalent THEN:
                RETURN dictionary as matchings
        ELSE:
                RETURN None
```

b.



Starting from C_1, we assign it to it's first transaction that appears. As C_4 also selects T_1, we go to the next transaction T_3. This transaction is also taken so we move to T_4. This transaction is also taken, and there are no next values. Therefore, we take C_4s first transaction and deallocate the original coin C_1, and add the old coin back to the set. As there are still coins in the set at the end of the loop. We continue to a second round. C_1 is successfully allocated to it's second transaction T_2 and the process is complete as there are no coins left in the set.

	After round 1	**After round 2**
C1		T2
C2	T3	T3
C3	T4	T4
C4	T1	T1

We then do the same for transactions. T_1 is allocated C_1, but then is added back to the transaction set after T_2 is allocated C_1 as it has no available coins. T_2, T_3 and T_4 are all given first preferences and in the second round T_1 is given it's second preference.

	After round 1	**After round 2**
T1		C4
T2	C1	C1
T3	C2	C2
T4	C3	C3

We know that both of the pairings are stable as each coin and transaction in each example could not be swapped with another. Because the two results are the same, we know that this is a bad mixing.

c.
The time complexity for this algorithm is $\mathcal{O}(n+m)$ as we are iterating over every coin and transaction element `n`. This iteration may occur `a` number of times, where there might be repetitions of second and third rounds as the example above. Within that, every edge `m` might be tested several times `b` throughout the loops until a solution is converged on. Therefore, the time complexity could be understood as $\mathcal{O}(an+bm)$.

Problem 5

a.
b.
c.