S3774430 Myeonghoon Sun

I certify that this is all my own original work. If I took any parts from elsewhere, then they were non-essential parts of the assignment, and they are clearly attributed in my submission. I will show that I agree to this honour code by typing “Yes": Yes

Problem 1.

a) The algorithm computes the number of nodes in the binary tree by traversing to the leftmost node, at which recursive calls return 0 (if it does not have a right child, in which case a number 1 or greater is returned depending on the number of child nodes to its right), which is added to 1 (denoting its own count), which is then returned to a recursive call made from the parent node, which then the same recursive call to its right child before repeating this whole process up to the root node, at which the total number of nodes is returned.

b) Divide-and-conquer: a problem is divided into subproblems, solutions to which are combined to get a solution to the original problem without discarding any of them as in decrease-and-conquer.

c) $C(h) = C(h-1) + 2^{h+1}$ for every h > 0, $C(0) = 2$

d) The method of backward substitutions is applied to find a pattern, with which an explicit formula in terms of h can be derived:

   1. $C(h) = C(h-1) + 2 \cdot 2^h$ substitute $C(h-1) = C(h-2) + 2 \cdot 2^{h-1}$
   2. $C(h) = C(h-2) + 2(2^{h-1} + 2^h)$ substitute $C(h-2) = C(h-3) + 2 \cdot 2^{h-2}$
   3. $C(h) = C(h-3) + 2(2^{h-2} + 2^{h-1} + 2^h)$
   4. Now, a general formula according to the pattern above is:
   $$C(h) = C(h-i) + 2\left(2^{h-(i-1)} + \cdots + 2^{h-(i-i)}\right)$$
   5. Plugging in h in i yields
   $$C(h) = C(h-h) + 2\left(2^{h-(h-1)} + \cdots + 2^{h-(h-h)}\right)$$
   $$C(h) = 2 + 2(2^1 + \cdots + 2^h)$$
   $$C(h) = 2 + 2(1 + 2^1 + \cdots + 2^h) - 2$$
   $$C(h) = 2(1 + 2^1 + \cdots + 2^h)$$
   $$C(h) = 2\sum_{k=0}^{h} 2^k = 2\left(2^{\mathrm{h}+1} - 1\right)$$

e) $\Theta(2^n)$

Problem 2.

In all the algorithms below, all the operations will be done in-place.

a) Description

   1. Iterate over *list_triple.*
   2. In the current iteration, linearly search the passport number of the current person in *list_double*. Once it is found, remove the person from *list_double.*
   3. Repeat the step 2 for the remainder of the iteration.
   4. Once the iteration is over, return *list_double.*

   The time complexity of the algorithm above is $O(mn)$.

b) Description

   1. Heapsort *list_triple* by the passport number [$O(m \log m)$]

2. Iterate over *list_double* to find the current passport in *list_triple.* If it is found, remove the associated person from *list_double*; else the person is kept.
3. Repeat the step 2 until the iteration is over $[O(n \log m)]$.
4. Return the remaining people in *list_double.*

The time complexity of the algorithm above is calculated as follows:

- Heapify *list_triple:* $O(m)$
- Sort the heap using siftDown() on m elements: $O(m \log m)$
- The dominating term of the two prevails: $O(m + m \log m) = O(m \log m)$
- Search in the heap, $O(\log m)$, n times and remove the person if found: $O(n \log m)$
- The dominating term of the two operations $O(m \log m)$ and $O(n \log m)$ is $O(n \log m)$ since $m \leq n$.

The requirement for the constant space complexity is satisfied as the heapsort's space complexity is $O(1)$.

c) Description

1. Build a hash table for *list_triple* and name it *hash_triple* (*HT)*
2. Iterate over *list_double* to find the current passport in *HT* and remove it from *list_double* if found or keep it otherwise
3. Once the iterations are over, return the modified version of *list_double*.

Pseudocode

```
ALGORITHM FindWhoNeedBoosterShots(LD,LT)
 // Finds who have been double vaccinated but have not received third shots.
    // Input: A list LD of $n$ people, represented as passport numbers, who have been
    double vaccinated and a list LT of m people, again represented as passport numbers,
    who have been double vaccinated and have received the third shot.
    // Output: A modified list of LD that contains all the passport number of the people
    who need the third shot

    HT <- new Hashtable(LT) // hash table of $m$ people from LT, with each of the
    passport numbers as a key and the Boolean value True as its value

 FOR each passport_number pn in LD:
     IF HT contains pn:
   REMOVE pn from LD
 RETURN LD
```

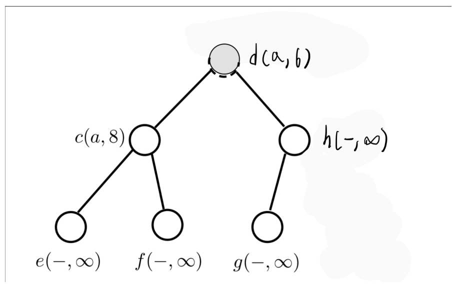
Time Complexity

- $O(m)$ since the time complexity for inserting an element into a hash table is $O(1)$ on average and m insertions are required to build *HT*.
- $O(n)$ since the time complexity for searching an element in a hash table is $O(1)$ on average and n searches (iterations) are required.
- $O(m + n)$ on average where $m \leq n$ equals $O(n)$.

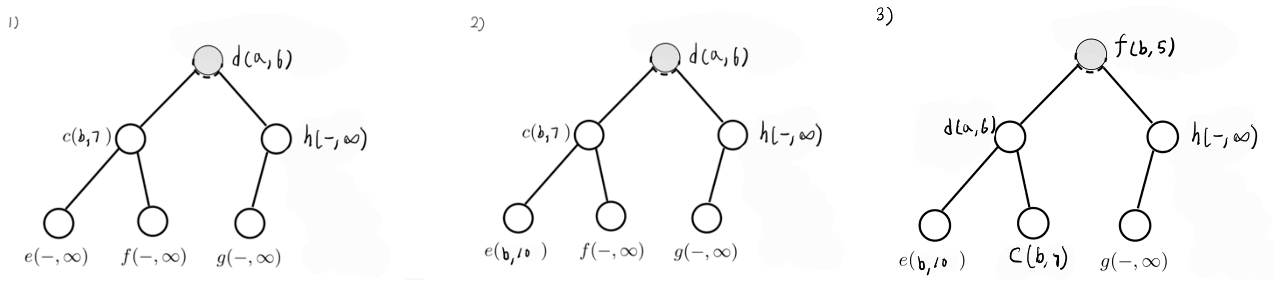
Problem 3.

a)

S3774430 Myeonghoon Sun



b)



c)

	S: vertices whose shortest paths have been known	Priority queue of remaining vertices
1	$a(a,0)$	$b(a,4), c(a,8), d(a,6), e(-,\infty), f(-,\infty), g(-,\infty), h(-,\infty)$
2	$a(a,0), b(a,4)$	f(b,5), d(a,6), h(-,∞), e(b,10), C(b,7), g(-,∞)
3	a(a,0), b(a,4), f(b,5)	d(a,6), C(b,7), h(f,12), e(b,10), g(-,∞)
4	a(a,0), b(a,4), f(b,5), d(a,6)	c(b,7), e(b,10), h(f,12), g(d,10)
5	Same as above (SAA); C(b,7)	g(c,8), e(b,10), h(f,12)
6	SAA, g(C,8)	e(g,9), h(g,10)
7	SAA, e(g,9)	h(g,10)
8	SAA, h(g,10)	

Table 1: Complete this table for Part c).

d)

	Shortest Paths	Distances
a	$a \to a$	0
b	$a \to b$	4
c	$a \to b \to c$	7
d	$a \to d$	6
e	$a \to b \to c \to g \to e$	9
f	$a \to b \to f$	5
g	$a \to b \to c \to g$	8
h	$a \to b \to c \to g \to h$	10

Problem 4.

a) The algorithm has two functions; MysteryWrapper(MW) function contains MysteryRecursive(MR) that is called on the first coin in R[0], which is put in the set named visitedCoins. In MR, the first coin traverses to the first transaction in its associated linked list in L. Like the first coin, the transaction is put in the set named vistedTrans. Once traversed to the transaction, MR is called on the next unvisited coin in its linked list within R. The steps above are repeated until the algorithm arrives at one of the 3 end states.

- a coin connected to a single transaction
- a transaction connected to a single visited coin (the one it traversed from)
- a transaction connected to multiple visited coins

In the first end state, the coin returns the control over to the transaction that called a recursive function on it. The coin is then mapped to the transaction in M. This process is repeated up to the very first coin. Once the control is passed over to the first coin, it finishes iterating its associated transactions and finds its pair, if any. Additionally, throughout the process, whether the graph contains any cycle is checked. If it turns out cyclic, MW clears the contents of the map and returns an empty map. This cyclic check is incorporated for all the states. In the second end state, the final transaction is mapped to the coin (within the same MR) before returning the control to the previous transaction that called MR on the coin. In the third end state, it is important to check whether each of the coins, aside from the one traversed from, connected to the final transaction is connected to a unique transaction that has only one coin in its associated linked list in R. If not, the graph is deemed cyclic. If the coin cannot sequentially travel to the last coin or the last transaction because of disconnection in the graph, MW first checks whether the disconnected portion of the graph just explored is cyclic. If it is acyclic, another MR is called on the next unvisited coin.

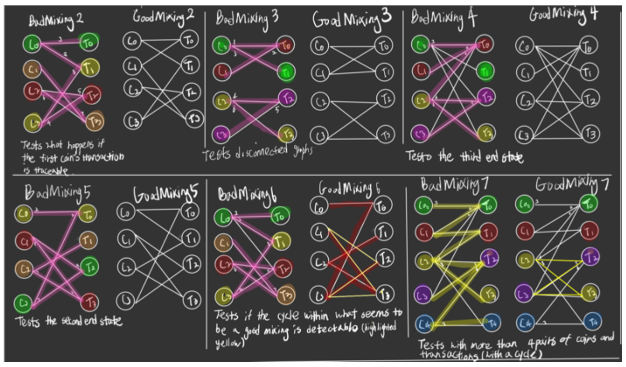
ALGORITHM *MysteryWrapper(n, m, L, R)*

```
// Input: identical to the required input from the specification
// Output: unique mapping M
DECLARE visitedCoins as set
DECLARE visitedTrans as set
SET isCyclic and endsWithTran to false
firstCoin <- the head of the first linkedlist in R
// 0 is a null value for lastTran denoting the last transaction that called MR
lastTran <- 0
INSERT firstCoin in visitedCoins
WHILE isCyclic is false and visitedCoins size is not the same as n
   // L, R, visitedCoins, visitedTrans, M, isCylic, and endsWithTran are passed as references
   MysteryRecursive(firstCoin,lastTran,L,R,visitedCoins,visitedTrans,M, isCyclic,
   endsWithTran)
IF isCylic is true,
   CLEAR the contents of M
RETURN M
```

ALGORITHM
MysteryRecursive(currCoin,lastTran,L,R,visitedCoins,visitedTrans,M,isCyclic,endsWithTran)

```
// Output: M with coins as keys and transactions as values
currTrans <- L[currCoin]'s linked list
FOR each currTran in currTrans
   IF currTran is not in visitedTrans
      INSERT currTran in visitedTrans
      INCREMENT numTransTravelled
      nextCoins <- R[currTran]'s linked list
      FOR each nextCoin in nextCoins
```

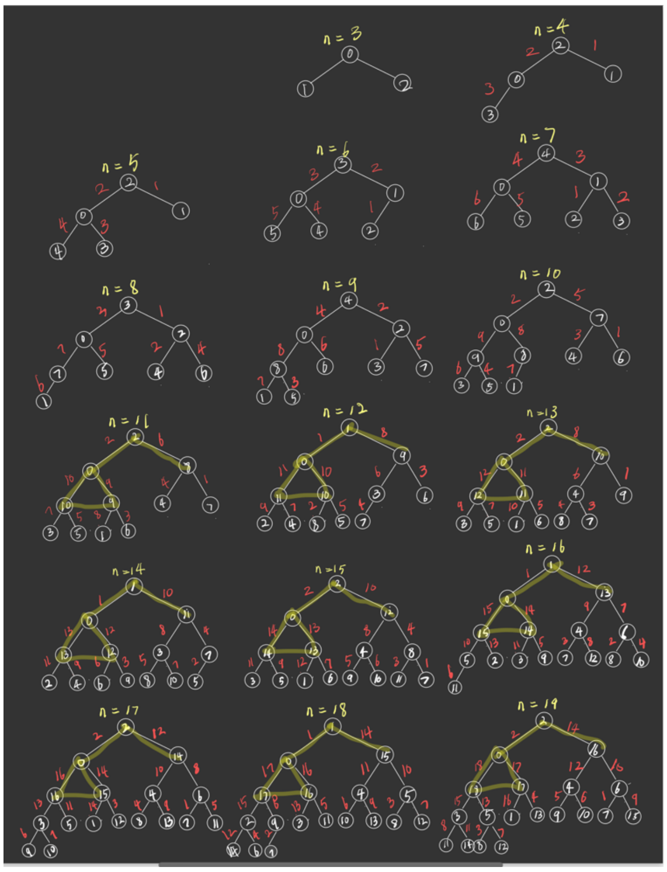
```
            IF nextCoin is not in visitedCoins
               INSERT nextCoin in visitedCoins
               MR(nextCoin,currTran,L,R,visitedCoins,visitedTrans,M,isCylic,endsWithTran)
               IF isCyclic is false
                  IF endsWithTran is true
                     INSERT currCoin and currTran in M
                  ELSE
                     INSERT nextCoin and currTran in M
            IF no nextCoin is travelled // the traversal ends at a transaction
               IF R[currTran] is connected to multiple coins
                  SET isAllUniquelyMapped to true
                  FOR each nextCoin in nextCoins
                     IF nextCoin is not currCoin
                        SET isUniquelyMapped to true
                        nextTrans <- L[nextCoin]
                        FOR nextTran in nextTrans
                           IF R[nextTran] contains only one coin
                              SET isUniquelyMapped to true
                        IF isUniquelyMapped is false
                           SET isAllUniquelyMapped false
                  IF IsAllUniquelyMapped
                     INSERT currCoin and currTran in M
                  ELSE
                     SET isCyclic to true
               ELSE // If R[currTran] is only connected to currCoin
                  SET endsWithTran to true
                  INSERT currCoin and currTran in M
      ELSE // When exiting the recursion, check whether the graph is cyclic
         IF currTran is not lastTran // if currTran is not same as the tran that called recusion
            FOR each coin in R[currTran]
               IF coin is not currCoin
                  FOR each tran in L[coin]
                     IF M already has currCoin and tran mapped
                        SET isCyclic to true
    IF there is no unvisited currTran for currCoin
      Set endsWithTran to False
```


BadMixing 2
GoodMixing 2
BadMixing 3
Good Mixing 3
BadMixing 4
GoodMixing 4
Tests what happens if the first coin's transaction is traceable.
Tests disconnected graphs
Tests the third end state
BadMixing 5
GoodMixing 5
BadMixing 6
GoodMixing 6
BadMixing 7
GoodMixing 7
Tests the second end state
Tests if the cycle within what seems to be a good mixing is detectable (highlighted yellow)
Tests with more than 4 pairs of coins and transactions (with a cycle)

c) The left diagram shows some of the scenarios considered for the algorithm to be as error-free as possible. Unit tests for the scenarios have been written in C++ and been attached as an extra reference to demonstrate the algorithm.

d) The algorithm is guaranteed to check at worst 2n+ m as it traverses to the end state. When it arrives at the end state, the additional check may incur additional traversal of at most (n + m). The more edges the graph has, the earlier it will be deemed cyclic, prompting the early termination of the algorithm. Therefore, the time complexity is at most 3n + 2m.

Problem 5.



a) The brute-force search works relatively efficiently compared to a human's manual calculation for a tree up to a certain size. However, the number of permutations factorially increases as n gets bigger, making it impossible to produce results within the required running time of 5 minutes. However, given the hard-earned results, a pattern to exploit emerges for a tree of size n > 10. First, the largest number n – 1 must be adjacent to 0 to produce the largest label, n – 1. Also, it seems intuitively easier to pair the second largest with 0 to produce the second largest label. As a result, the triangular positioning on the left side of the root node effectively reduces the size of the problem by 3. To compute a permutation for a tree of size 19 in time, the problem still needs to be reduced at least by 2. For bigger trees with the even number of nodes, the root's value oddly turned out to be 1 many times; with the odd number of nodes, it evenly turned out to be 2. Finally, the right child of the root persistently contained a value of n – 3. Given this pattern, the maximum n to solve becomes 14, making it likely to compute a permutation for each of the trees within the required time (the pattern is highlighted in yellow in the diagram above).

As for the brute-force algorithm using recursion that is executed on trees of all sizes, it first tries a number within available numbers to see whether the leftmost element (node) whose value is None can contain the number while preserving the magical labelling property. It does so by checking whether the resultant label(s) is a duplicate already claimed by other nodes. If not, it recursively repeats the same process for the next leftmost element whose value is None. If no magical label is available, the control of the flow is returned where the recursion was last called and another num in nums is tried to try the same index yet again. This process is repeated until there are no magical labels left to choose from, in which case, all the recursion calls come off the stack, stop searches and, finally, return the permutation.

ALGORITHM *build_magical_tree(n)*
// Input: n (number of nodes in a tree)
// Output: prints the nodes of a tree

```
    SET p to a list to hold the permutation
    SET nums to a set of available numbers to permutate
    SET mls to a set of available magical labels which is the same as nums except for the zero
    IF n is equal to or less than 10
        ret <- build_magical_tree_recursive(p, 0, nums, mls, false)
        OUTPUT ret
    ELSE // if n is greater than 10
        SET the elements of indices from 0 to 4 according to the pattern discussed above
        REMOVE the values used above from nums
        REMOVE the values used above from mls save for 0.
        ret <- build_magical_tree_recursive(p, 5, nums, mls, False) // start from the fifth index
        OUTPUT ret
```

ALGORITHM *build_magical_tree_recursive(p, idx, nums, mls, is_ml_found)*

// Input: p, a list to contain a valid permutation; idx, the index of a node whose value is being determined; nums, a list of available numbers; mls, a list of available magical labels; is_ml_found, a Boolean to help terminate the search once the valid permutation found

// Output: ret, the permutation that holds the magical labelling property.

```
    SET ret to nothing
    FOR each num in nums
        IF the final permutation is not found
            mls_copy <- updated_mls(p, idx, num, mls)
        IF mls_copy is a copy of mls:
            MAKE a copy of p, nums & SET p_copy[idx] to num & REMOVE num from nums_copy
            IF p_copy has an element (node) whose value needs to be assigned
                SET left_most_none_idx to the index of such an element
                ret <- build_magical_tree_recursive(p_copy, left_most_none_idx, nums_copy,
                mls_copy, is_ml_found)
                IF ret contains the required number of numbers
                    SET is_ml_found to true
        IF there is no more mls to choose from
            SET Is_ml_found to true
            ret <- p_copy
    RETURN ret
```

ALGORITHM *update_mls(p, idx, num, mls)*

// This auxiliary algorithm helps determine whether the selected number results in a duplicated magical label. If not, return the updated_mls. Otherwise return nothing. Please refer to the code for more details.