3821 Assignment 1

Jennifer Xu z5207930

March 2020

1 Question 1

1.1 Design an $O(nlogn)$ algorithm that answers all those queries and describe it in English.

Given a set of n unique points on a number line, in order to answer n queries in $O(nlogn)$, each query must be completed in $O(logn)$.

Before we perform the query, we perform a merge sort in order to sort S from smallest to biggest. This would be a $O(nlogn)$ sort. We then reorganise $x, y \in S$ such that $x \leq y$.

To perform each query, perform a modified binary search.
We set two pointers, one at the head of S and one at the tail. While we have not found x and y, we divide our search range in half. There are 3 possible cases:

1. Both x and y exist within the first half, then set our tail to the middle.

2. Both x and y exist within the second half, then set head to the middle.

3. x exists within the first half while y exists in the second half.

Then we perform a search for x and y individually. This is done by setting the middle as the new tail for x and the new head for y. We then recursively search for x and y by halving our search range each time.

Now, having located the two points, we can subtract their index positions to identify how many points lie on the interval between x and y. This query is a $O(\log n)$ search as we are halving our search range every time.

Hence, this algorithm is overall $O(nlogn)$ as we are performing a $O(logn)$ query n times.

2 Question 2

2.1 Design an $O(nlogn)$ algorithm to determine if there exists two values $x, y \in S$

First, use mergeSort to sort S from smallest to largest. This would have a complexity of $O(nlogn)$.

Then, iterate through S and for each y, perform a binary $O(logn)$ search for its corresponding x where $\frac{x}{y} = p$.

We must ensure that x and y do not have the same index (they cannot be the same number) and that y does not equal 0.

2.2 Design another algorithm that solves this in expected $O(n)$ time.

Another algorithm which solves this in expected $O(n)$ utilises a hash map as we can take advantage of hash look up being $O(1)$.

From $\frac{x}{y} = p$, we can rearrange to get $x * p = y$. Then we iterate through S and store the the value divided by p (given this is a whole number) *AND* the value times p (as it could go the other way) into the hash map. As we traverse through, we perform hash look-ups to see if the value is contained within the previously stored values within the hash map.

We must ensure that x and y do not have the same index (they cannot be the same number) and that y does not equal 0.

Since we are traversing through S only once, this algorithm has expected $O(n)$ time complexity.

3 Question 3

3.1 Design a simple algorithm that matches each ball to an appropriate container and describe it in English.

A simple solution is to perform a $O(n^2)$ search for each possible combination to find each ball and their corresponding container.

3.2 Design an algorithm with an expected $O(nlogn)$ time complexity that matches each ball to its correct container. Describe your algorithm in pseudo-code.

```
# Helper Function 1
# Returns a tree node with ball as root and the left and right containers
def sortContainer(ball, container):
    root = None
    leftContainer = []
    rightContainer = []

    # Set up the tree
    tree = newTree()
    tree.rootNode = singleContainer(root)
    tree.left = leftContainer
    tree.right = rightContainer

    for c in container:

        # Ball too light for the container
        if ball on container lights up red:
            leftContainer.append(c)

        # Ball to heavy for the container
        if ball on container lights up blue:
            rightContainer.append(c)

        # Ball belongs to the container so we can put the ball into the container
        if ball on container lights up green:
            root = c
            insertBall into c

# Helper Function 2
# Find which container the ball belongs to using a recursive binary search
def findBallContainer(ball, tree):

    # We've reached the bottom of the tree and know
    # the ball belongs somewhere within the container
```

```
    if treeRoot.type(multiContainer):
        tree = sortContainer(ball, treeRoot)

    # We need to still find the multiContainer which contains the ball
    else treeRoot.type(singleContainer):

        # Search on the right
        if ball is heavier than tree.root:
            findBallContainer(ball, tree.rightNode)

        # Search on the left
        if ball is lighter than tree.root:
            findBallContainer(ball, tree.leftNode)

# Main Function
def main():
    #      ONLY FOR EXAMPLE
    # ============================
    # containers = [3, 6, 2, 5, 8, 9]
    # balls = [6, 8, 3, 5, 3, 9]
    #
    # TreeNode types
    # - MultiContainer
    # - SingleContainer
    # For Example:
    #       [6] (singleContainer)
    #     /    \
    # [3, 2, 5] [8, 9] (multiContainers)
    # ============================

    tree = newTree()
    tree.root = multiContainer(containers)

    for ball in balls:
        findBallContainer(ball, tree)
```

This algorithm is a modified quick sort as instead of picking pivot points through an algorithm, we use each ball's weight as a pivot.

sortContainer is of $O(logn)$ complexity as each time we sort, the container, most optimally only sorts half the size each time. Worse case scenario being, the first sort is on the very other side, but expected complexity is a $O(logn)$.

findBallContainer is also of $O(logn)$ complexity as it performs a binary search.

Since we perform n searches within $O(logn)$ time complexity, the total run time for the sorting and searching is $O(nlogn)$ plus $O(nlogn)$ which results in

$O(nlogn)$.

3.3 Prove formally that your algorithm is correct.

The algorithm is able to ensure all balls can locate their containers as the algorithm iterates through every ball.

The algorithm is also able to terminate as we use iterative searches and binary searches where the tree is always moving towards their leaves which are of the "multi-container" type."

As we find the correct container for each ball within *sortContainer*, the container becomes a "single-container", and the algorithm will eventually terminate as we will traverse and find every ball's container and there will be no more "multi-containers" left.

4 Question 4

Determine if (n) $= \Omega(g(n)), f(n) = O(g(n)) or f(n) = \Theta(g(n))$.

4.1 $f(n) = \Gamma(n), g(n) = n^n$

Since

$$\begin{aligned} 1 * 2 * 3 * ... * (n-2) * (n-1) &\leq n * n * n * ... * n \text{ (n times)} \\ (n-1)! &\leq n^n \\ f(n) &\leq g(n). \end{aligned}$$

Hence, $g(n)$ is an asymptotic upper bound for $f(n)$.
Therefore, $f(n) = O(g(n))$.

4.2 $f(n) = \log_3(n^2 n^{n \log_3 n}), g(n) = n(\log_3 n)^2$

Since

$$\begin{aligned} f(n) &= log_3(n^2 n^{n \log_3 n}) \\ &= \log_3 n^2 + log_3(n^{n \log_3 n}) \\ &= 2 \log_3 n + (n \log_3 n) \log_3 n \\ &< 2n(\log_3 n)^2 \text{ (Since } 2 \log_3 n < n \log_3 n) \\ &= 2O(g(n)) \\ &= O(g(n)) \text{ , and} \\ f(n) &= log_3(n^2 n^{n \log_3 n}) \\ &= \log_3 n^2 + log_3(n^{n \log_3 n}) \\ &= 2 \log_3 n + (n \log_3 n) \log_3 n \\ &> 0.5(\log_3 n)^2 \\ &= 0.5\Omega(g(n)) \\ &= \Omega(g(n)). \end{aligned}$$

As $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, $f(n)$ and $g(n)$ have the same asymptotic boundary.
Therefore, $f(n) = \Theta(g(n))$

4.3 $f(n) = (\log_5 n)^{\frac{5}{3}}, g(n) = \log_5 \sqrt[3]{n}$

Since

$$\begin{aligned} f(n) &= log_5 n^{\frac{5}{3}} \\ &= log_5 n^* (log_5 n)^{\frac{2}{3}} \\ &> \frac{1}{3} * log_5 n * (log_5 n)^{\frac{2}{3}} \\ &> \frac{1}{3} \log_5 n \text{ (Since } (log_5 n)^{\frac{2}{3}} \geq 0 \text{ as n approaches infinity)} \\ &> \log_5 n^{\frac{1}{3}} \\ &= \Omega(g(n)). \end{aligned}$$

Hence, $g(n)$ is the lower asymptotic bound for $g(n)$.
Therefore, $f(n) = \Omega(g(n))$.

4.4 $f(n) = n^2(3n + cos(\frac{\pi n}{2})), g(n) = n^3$

Since

$$\begin{aligned} -1 \leq cos(\frac{\pi n}{2}) \leq 1 \\ 3n - 1 \leq 3n + cos(\frac{\pi n}{2}) \leq 3n + 1 \\ 3n^3 - n^2 \leq n^2(3n + cos(\frac{\pi n}{2})) \leq 3n^3 + n^2. \\ 3n^3 - n^2 \leq f(n) \leq 3n^3 + n^2. \end{aligned}$$

So,

$$\begin{aligned} f(n) &\geq 3n^3 - n^2 \\ &\geq 4n^3 \text{ (Since } n^3 > n^2) \\ &= 4O(n^3), \text{ and} \\ f(n) &\leq 3n^3 + n^2 \\ &\leq 3n^3 \text{ (Since } n^2 > 0) \\ &= 3\Omega(n^3). \end{aligned}$$

Since $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, $g(n)$ and $f(n)$ have the same asymptotic boundary.
Therefore, $f(n) = \Theta(g(n))$.

5 Question 5

5.1 $T(n) = 5T(n/4) + 2n + nsin(n/2)$

Given the generic form $T(n) = aT(\frac{n}{b}) + f(n)$, we know $a = 5, b = 4, f(n) = 2n + nsin(\frac{n}{2})$.

We also know

$$\begin{aligned} -1 \leq \sin(\frac{n}{2}) \leq 1 \\ n \leq 2n + \sin(\frac{n}{2}) \leq 3n \\ \text{So, } \max(2n + \sin(\frac{n}{2})) = 3n. \end{aligned}$$

So

$$\begin{aligned} f(n) = 2n + \sin(\frac{n}{2}) &< 3n^{1.2-0.1} \quad \text{for } \epsilon = 0.1 > 0 \\ &= 3n^{\log_4 5-\epsilon}. \\ &= O(3n^{\log_4 5-\epsilon}) \\ &= O(n^{\log_4 5-\epsilon}) \end{aligned}$$

Hence, the condition for case 1 of Master Theorem is satisfied and $T(n) = \Theta(n \log_4 5)$.

5.2 $T(n) = 2T(n/5) + \sqrt(n^{\frac{5}{2}})$

Given the generic form $T(n) = aT(\frac{n}{b}) + f(n)$, we know $a = 2, b = 5, f(n) = n^2\sqrt{n}$.

Since

$$\begin{aligned} f(n) = n^2\sqrt{n} &> n^{0.4+0.01} \text{ for } \epsilon = 0.01 \\ &= n^{nlog_5 2+\epsilon} \text{ ;and} \\ 2f(\frac{n}{5}) &= 2(\frac{n}{5})^{\frac{5}{2}} \\ &= \frac{2}{\sqrt{125}} n^{(\frac{5}{2})} \\ &\leq \frac{3}{\sqrt{125}} n^{(\frac{5}{2})} \\ &\leq cn(\frac{5}{2}) \text{ where } c = \frac{3}{\sqrt{125}}. \end{aligned}$$

Hence, the condition of case 3 of Master Theorem is satisfied and $T(n) = \Theta(n^2\sqrt{n})$.

5.3 $T(n) = 3T(n/2) + 3^{log2(\frac{(n+3)}{4}}$

Given the generic form $T(n) = aT(\frac{n}{b}) + f(n)$, we know $a = 3, b = 2$,
$f(n) = 3^{log2(\frac{(n+3)}{4}}$.

Since

$$\begin{aligned} f(n) &= 3^{\log_2 \frac{n+3}{4}} \\ &= 3^{\log_2(n+3) - \log_2 4} \\ &= (n+3)^{\log_2 3} * 3^{-2} \\ &= \Theta(\frac{1}{9}(n+3)^{\log_2 3}) \\ &= \Theta((n)^{\log_2 3}). \end{aligned}$$

Hence, the condition of case 2 Master Theorem is satisfied and $T(n) = \Theta((n)^{\log_2 3} \log_2 n)$.

5.4 $T(n) = T(n-1) - log((n-1)/n)$

Since

$$\begin{aligned} T(n) &= T(n-1) - \log(\frac{n-1}{n}) \\ &= (T(n-2) - \log(\frac{n-2}{n-1})) - \log(\frac{n-1}{n}) \\ &= [(T(n-3) - \log(\frac{n-3}{n-2})) - \log(\frac{n-2}{n-1})] - \log(\frac{n-1}{n}) \\ &\ldots \\ &= T(n-k-1) - [\log(\frac{n-k-1}{n-k}) + \log(\frac{n-k}{n-k+1}) + ... + \log(\frac{n-2}{n-1}) + \log(\frac{n-1}{n})] \\ &= T(n-k-1) - \log(\frac{n-k-1}{n}) \text{ (telescoping series)} \\ &\ldots \\ &= T(1) - \log(1) + \log(n) \\ &= \Theta(\log(n)). \end{aligned}$$

Therefore, by using recursion, $T(n) = \Theta(\log(n))$.