Justin Szabo 51768264 APS Assignment 1

Task 1.1

Algorithm 1

```
Require: A sorted list L, an interval [x_s, x_f]
function list_interval(L, x_s, x_f):
        answer = [ ]
        for each element in L do
                if the element is between or equal to x_s and x_f do
                        Append the element to answer
                end if
        end for
        print(answer)
end function
```

Algorithm 1.1

```
function input_string:
        Q = 0
        C = 0
        L, Q_1 = [ ]
        Q, L = STDIN
        for |Q| do
                L.append(STDIN)
        end for
        for |L| do
                create empty sublists in Q_1
        Q_1[c].append(STDIN)
                c increments by 1
        end for
        return L, Q_1
        end function
```

Algorithm 1 takes a list and a range of two points which it'll find numbers between. For each element in the list if the elements are in within the range they are then appended to another list. When each element is gone over then the final list of elements in the range is printed.

Algorithm 1.1 takes the size of the list and the number of queries as an *STDIN*. The for the size of the value in the variable *I,* it takes integers, via *STDIN*, and append them to the list *L*. Then for the size of the value in *Q* it creates empty sublists in Q_1 and for each sublist it takes the intervals as a *STDIN* and then increments to the next sublist.

Correctness

Algorithm 1

Input: Consider a list of n elements such that $L = (x_1, x_2, ..., x_n)$, and then 2 elements, x_s, x_f, which are the range.

Output: A list such that $x_s \leq x_n \leq x_f$ for example:

L = (1, 2, 3, 4), $x_s = 1$, $x_f = 3$ $\longrightarrow$ *L = (1, 2, 3)*

L = (1, 6, 4, 2, 8, 5, 3), $x_s = 4$, $x_f = 7$ $\longrightarrow$ *L = (6, 4, 5)*

Complexity

Algorithm 1 & 1.1

Algorithm 1.1 only uses one for loop, its complexity is only $\theta(n)$. Algorithm 1, however, goes through the first for loop $\theta(n)$ times, and we know that the inner-loop will be executed $n + \frac{n}{2} + \frac{n}{4} + \ldots + 1 \approx 2n$ times and appending to a list is done in $\theta(1)$ time.

$\theta(n) + \theta(n) + \theta(1) = \theta(2n) = \theta(n)$

Task 2.1

Algorithm 1.2

```
function input_string:
        Q = 0
        C = 0
        L, Q_1 = [ ]
        Q, L = STDIN
        for |Q| do
                L.append(STDIN)
        end for
        for |L| do
                create empty sublists in Q_1
        Q_1[c].append(STDIN)
                c increments by 1
        end for
        return L, Q_1
end function
```

Algorithm 1.3

Require: ordered list *L,* standard *Node* class
function balanced_tree:
  mid = |*L*| / / *2*
  root = Node(*L*[mid])
  root.left = balanced_tree(*L*[:mid])
  root.right = balanced_tree(*L*[mid + 1:])
  return root
end function

Algorithm 1.4

Require: Inorder traversal of binary tree (*T*), an interval [x_s ,x_f]
function find_range(*T*, x_s, x_f):
  T is bisected into two halves T_1 & T_2
  start = T_1 ,the leftmost, is searched for x_s .
    if x_s is in T_1
      return the point that appears
    end if
  end = T_2 ,the rightmost, is searched for x_f.
    if x_f is in T_2
      return the point that appears
    end if
  return *T[start:end]*

Algorithm 1.2 takes the size of the list and the number of queries as an *STDIN*. The for the size of the value in the variable *I,* it takes integers, via *STDIN*, and append them to the list *L*. Then for the size of the value in *Q* it creates empty sublists in Q_1 and for each sublist it takes the intervals as a *STDIN* and then increments to the next sublist.

Algorithm 1.3 takes a sorted list. It then takes the midpoint of this and turns it into a balanced binary tree. It then returns this tree.

Algorithm 1.4 takes an inorder traversal of a binary tree tree and two points. It then bisects the tree into two seperate lists. In the leftmost list, T_1, it searches for values the same as x_s. And the will return the index of the point that appears. In the rightmost list, T_2, it search for values the same as x_f . And then will return the index of the point that appears. Lastly it returns the list with all the values between the two indexes. This will only work for sorted lists.

Correctness

Input: Consider a sorted list of n elements such that $L = (x_1, x_2, ..., x_n)$ that are put into a balanced binary tree and the inorder traversal is given. Then 2 elements, x_s, x_f , which are the range.

Output: A list such that $x_s \leq x_n \leq x_f$ for example:
$L = (1, 2, 3, 4), x_s = 1, x_f = 3 \longrightarrow L = (1, 2, 3)$
$L = (1, 6, 4, 2, 8, 5, 3), x_s = 4, x_f = 7 \longrightarrow L = (4, 5, 6)$

Complexity

An insertion into a balanced binary tree takes $\theta(logn)$ time, however since algorithm 1.2 has a complexity of $\theta(n)$ then the total time taken for an insertion is $\theta(nlogn)$.
A range query takes $\theta(k + logn)$ time as it takes $\theta(logn)$ to obtain a inorder traversal. And the bisecting, as it may have to go through each item of a list takes $\theta(k)$ time, leading to it being $\theta(k + logn)$

Task 2

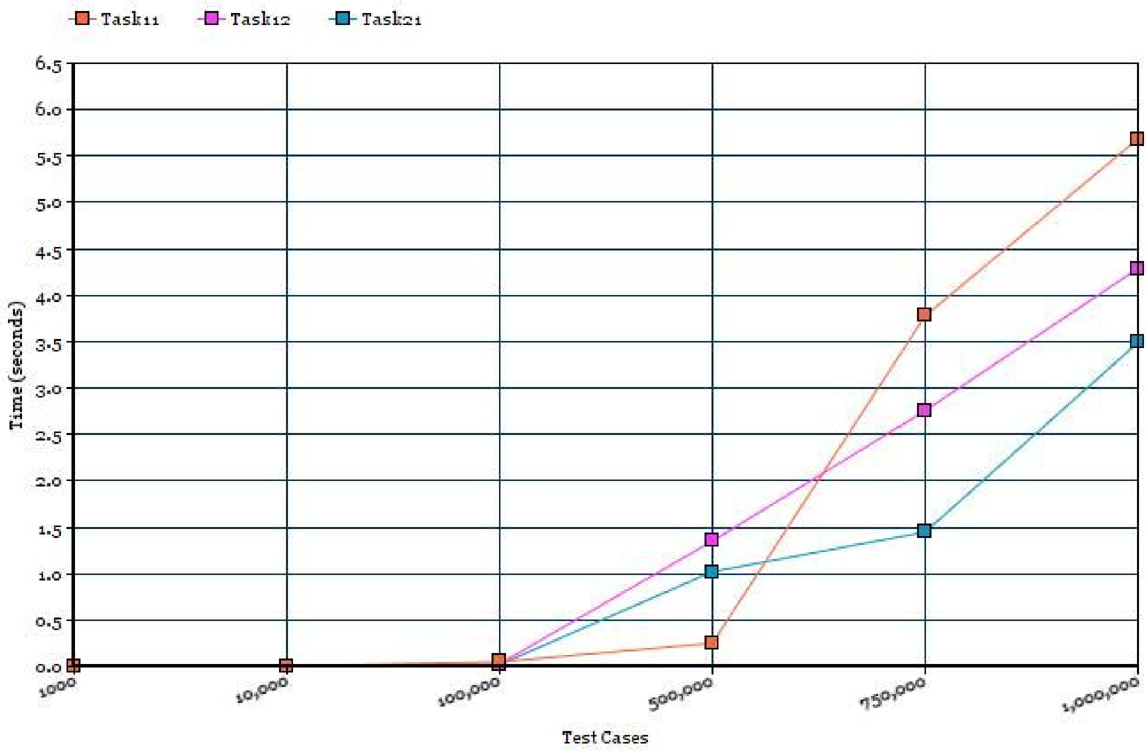
Task 2.2

It takes $O(nlogn)$ time to build a binary search tree from a list.
A binary search tree recurrence is $T(n) = T(n / 2) + O(1)$. If we use the master theorem where the complexity is $T(n) = aT(n / b) + f(n)$, where $a = 1$, $b = 2$, $log_b a = 1$.
$f(n) = n^c log^{k(n)} / k = 0$ & $c = log_b a$, then $T(n) = O(n^c log^{(k+1)} n) = O(logn)$ construction time.
As the tree needs to be built from a list, it takes $O(n)$ to traverse each value in a list.
Therefore the total time is $O(nlogn)$

Task 2.3

Checking if a node is to be reported takes $O(k)$ time. Since, a binary search tree is also balanced it takes $O(logn)$ time to go down each path. So the total time of the query is $O(k+logn)$

Task 2.4



To test the algorithms were run through varying test cases. These test cases contained randomly sized lists sizes, and were timed using the timeit library in python.
We can see in the graph that runtimes are very similar for all Tasks until they reach higher test cases.
Task11 managed until we see a massive spike at the 750,000 test case mark, this indicates that *Task11* may be useful for processing smaller amounts of data.
Task12 ends up being in the middle and runs with no obvious spikes in performance. This may be useful for processing medium sized datasets. An interesting thing to note, however, is the linear rising in time after the 100,000 test case mark.
Task21 isn't the fastest until the 750,000 mark. This may mean that it is better used for larger datasets.

Task 3

Task 3.1

Algorithm 3

Require: a matrix, intervals [x_1, x_2], no, of dimensions

```
function search_matrix(matrix, x1, x2, dimensions):
    n = [ ]
    for x in matrix do
        counter = 0
        for i in |dimensions| do
            if x1 ≤ x[i] ≤ x2 do
                counter += 1
            end if
        if counter = |dimensions\ do
            n.append(x)
        end if
        end for
    end for
    return n
end function
```

This algorithm takes a matrix(selection of points), two points, and the number of dimensions. It iterates over every value in the matrix and checks to see if it's within the two points given. A counter also runs to check the dimension. So if the points checked are then it appends it to a list, *n*, so they can be returned.

Task 3.2

Since the algorithm uses a nested for loop the complexity of it would be $O(n*m)$, as the number of the iterations in the loop behaving like: $n + n\text{-}1 + n\text{-}2, + \ldots + 3 + 2 + 1$ which leads to the complexity being $O(n^2)$ in the worst time.

Correctness

Input Consider a small matrix, such $m = [[1, 2], [10, 20]]$, and two points x_1, $x_2 = [[1, 0], [11, 21]]$
The matrix is checked to see which points (in the matrix) lay within a rectangle made by the two points in one dimension.

Output Points such that $x_1 \leq m \leq x_2$
So, $[1, 0] \leq [[1, 2], [10. 20]] \leq [11, 21] = [10, 20]$

Task 4.1

Algorithm 4

```
Require: A sorted list L
function median(L):
        if do
                return None
        end if
        if modulo by 2 is 1 do
                return L[|L| / 2]
        end if
        else do
                L_1 = L[(|L| / 2) - 1]
                L_2 = L[(|L| / 2) + 1]
                L_3 = ⌊(L_1 + L_2) / 2⌋
                return L_3
        end else
end function
```

The complexity of this algorithm is simply O(n) as $O(n) + O(n) + O(n) = O(3n) = O(n)$

Task 4.2

Since the function is recursive, we can say, $T(n) = 2T(n/2) + n$. So, this means – $2(2T(n/4) + n/2) + n = 4T(n/4) + n + n = 4T(n/4) + 2n$. This will continue further. And it will simplify down to $\theta(nlogn)$. Since the depth of the tree is $O(logn)$ and for each level we must calculate the median which is $O(n)$.

<u>Task 5</u>

<u>Task 5.2</u>

Algorithm 5

Require: region of a KDTree, two points [x_1 , x_2], dimension
function fully_contained(region, x_1, x_2, dimension):
    for i in *range(0, |dimension|)* **do**
        if not $x_1[i] \leq region[i] \leq x_2[i]$ **do**
            return False
        end if
    end for
    return True
end function

<u>Task 5.3</u>