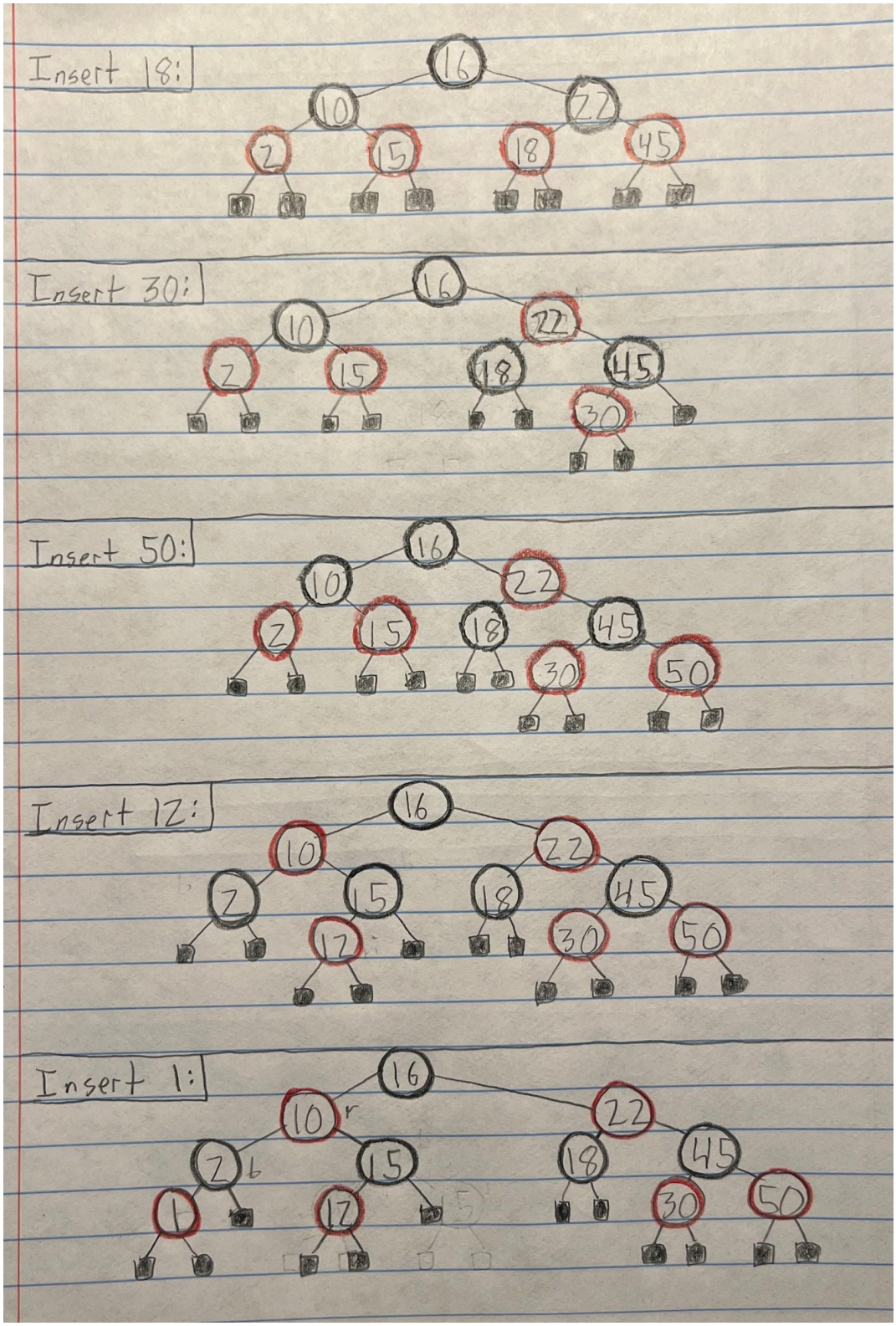
Problem #1 **Name:** Alexander Michelbrink



Problem #2

Pseudocode:

```
Algorithm inInterval(v,l,r)
    if ( isExternal(v) )
        return false
    if ( key(v)≥ l AND key(v)≤ r )
        return true
    if ( key(v) < l )
        return inInterval(leftChild(v),l,r)
    else
        return inInterval(rightChild(v),l,r)
```

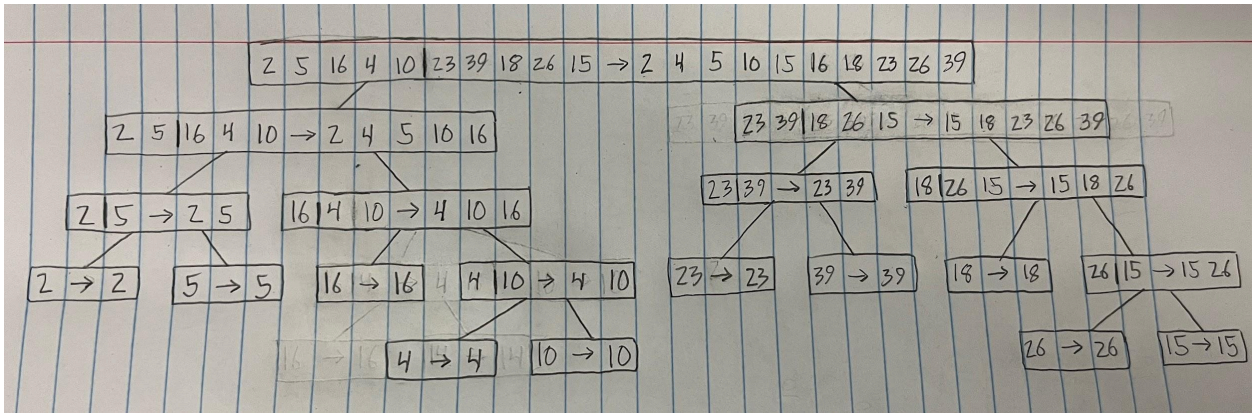
The above algorithm operates in O($log(n)$) time.

This can be seen by considering the worst-case of the first node lying within the interval [***l***,***r***] being at the bottom of the tree. Here, the algorithm would recursively call itself ***h*** times, where ***h*** is the height of the tree. This is because it moves down the tree one layer at a time until a key within the interval is found (returning true) or until it reaches a leaf (returning false). Therefore, the algorithm operates in O(***h***) time.

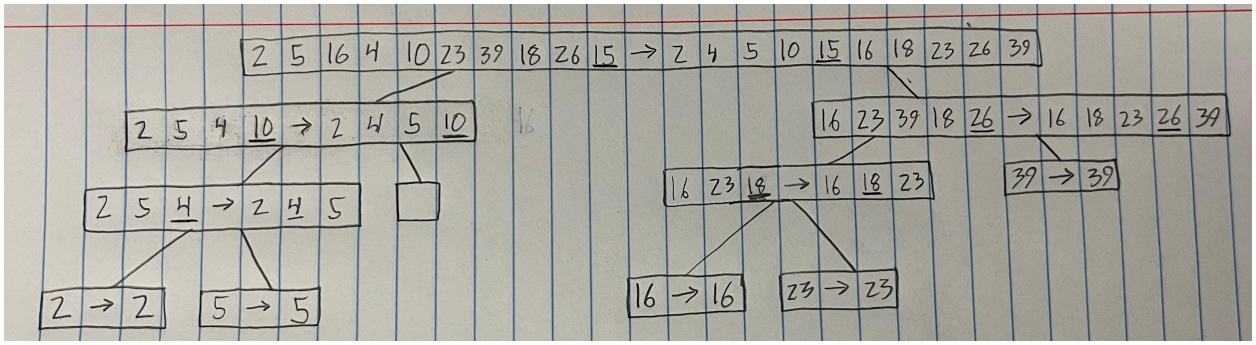
Since we are considering the case of a red-black tree storing ***n*** items, the tree ***T***'s height ***h*** is always equal to $log(n)$. Therefore, the algorithm operates in O($log(n)$) time.

Problem #3

A.) _____



B.) _____



C.) _____

The running time of the version of quick-sort that uses n/2 as the pivot and when the input sequence is already sorted is O($nlog(n)$).

Similar to merge sort, this would result in the height of the tree being equal to $log(n)$. This occurs since, at each new layer (each recursive call), we divide the previous layer in half. This halving occurs due to the pivot always being at the center of the already sorted sequence (about half of the values become the left child and the other half become the right child).

Since the work for nodes at each layer is done in O(n) time, we can multiply this by the height to determine the overall running time of O($nlog(n)$).

Problem #4

The following algorithm assumes the existence of a color() method (returns a string of the element's color) and a swapColor() method (swaps the colors of two elements) that operate in O(1) time.

Pseudocode:

```
Algorithm sortColors(S)
      first ← first(S)
      last ← last(S)
      while (rankOf(first) < rankOf(last))
            if (color(first) = "blue" AND color(last) = "red")
                  swapColor(first,last)
            if (color(first) = "red")
                  first = after(first)
            if (color(last) = "blue")
                  last = before(last)
```

The above algorithm operates in linear time.

The first and last elements of the sequence are tracked using the variables ***first*** and ***last***. The while-loop of the algorithm then loops until these ends become the same element or ***first***'s rank becomes greater than ***last***'s rank (they 'pass' each other). Inside the loop, ***first***, ***last***, or both variables move towards the center element in all cases and swap elements when necessary. This means, in the worst case (all red elements or all blue elements), the while-loop will execute ***n*** times where ***n*** is the amount of elements in ***S***. Since all methods used operate in O(1) time, we can conclude that the algorithm operates in O(n) time (linear time).

Problem #5

Pseudocode:

```
Algorithm findSumElements(A,B,m)

      for each p in A
            replaceElement(p,m - elemAtRank(rankOf(p)))

      mergeSort(A)
      mergeSort(B)

      while (!isEmpty(A) AND !isEmpty(B))
            AFirst ← A.elemAtRank(0)
            BFirst ← B.elemAtRank(0)
            if (AFirst = BFirst)
                  return true
            else if (AFirst < BFirst)
                  A.remove(first(A))
            else
                  B.remove(first(B))
      return false
```

The above algorithm works by first taking the sequence A and changing each of its integer elements to equal their value subtracted from m (element → m - element). We can now compare the two sequences directly to see if there were values in the original A and B that summed together to get m. We can do this since $a + b = m$ is equivalent to $m - a = b$. This step of the algorithm operates in O(n) time since there are n elements to redefine (causing the for-loop to execute n times).

Next, both A and B must be sorted. We can use the mergeSort() algorithm once on A and a second time on B to do so. Using this method, the two sorts are done in O($nlog(n)$) time.

Finally, we must see if there are any matching integers between the two sequences. The while-loop does this by removing the smallest element from between the lists until a match is found (returning true) or until the end of one of the sequences is reached (returning false). This can loop at most 2n-1 times in the worst case, meaning it operates in O(n) time.

Taking in all this information ($O(n) + O(nlog(n)) + O(n)$), we can see that the runtime of the entire algorithm is O($nlog(n)$).

Problem #6

A.) _____

We could use the radix-sort method to sort ***S*** in O(n) time. First, we must represent the integer of each item as the item's element and assign each item a 2-tuple key corresponding to…

$$(k1 = \frac{integer}{n}, k2 = integer \,\%\, n)$$

Here, the division in $k1$ is always rounded down. After doing this, the number of items will still be n, but the range of the values of the keys becomes [0,$n - 1$]. This is seen by dividing the original edge values of the range by n.

$$[\frac{0}{n} = 0\ , \frac{n^2-1}{n} = n - \frac{1}{n}$$ (which rounds down to $n - 1$)]

From here, we can carry out the radixSort() algorithm on ***S***, which runs the stable bucketSort() algorithm on each of the two keys in the tuple. Because bucketSort() is stable and since it is run on $k2$ (the remainder portion) before $k1$ (the whole portion), the resulting sorted ***S*** will have each item's integer element in increasing order.

Since radix-sort has a running time of O($d * (n + N)$), using this method would result in sorting ***S*** with a worst-case running time of…

O($2 * (n + n)$) = O($4n$) = O(n)

B.) _____

<u>The running time of radix-sort does NOT depend on the order of keys in the input.</u>

The running time of radix-sort is O(d*(n + N)), where n is the number of items in the input sequence S, N is the range of each key, and d is the number of keys per tuple. Since the order of the keys in the input sequence does not affect the number of keys, the maximum/minimum values of the keys (the range), nor the number of keys per tuple, this has no way of affecting the running time of the radix-sort algorithm.