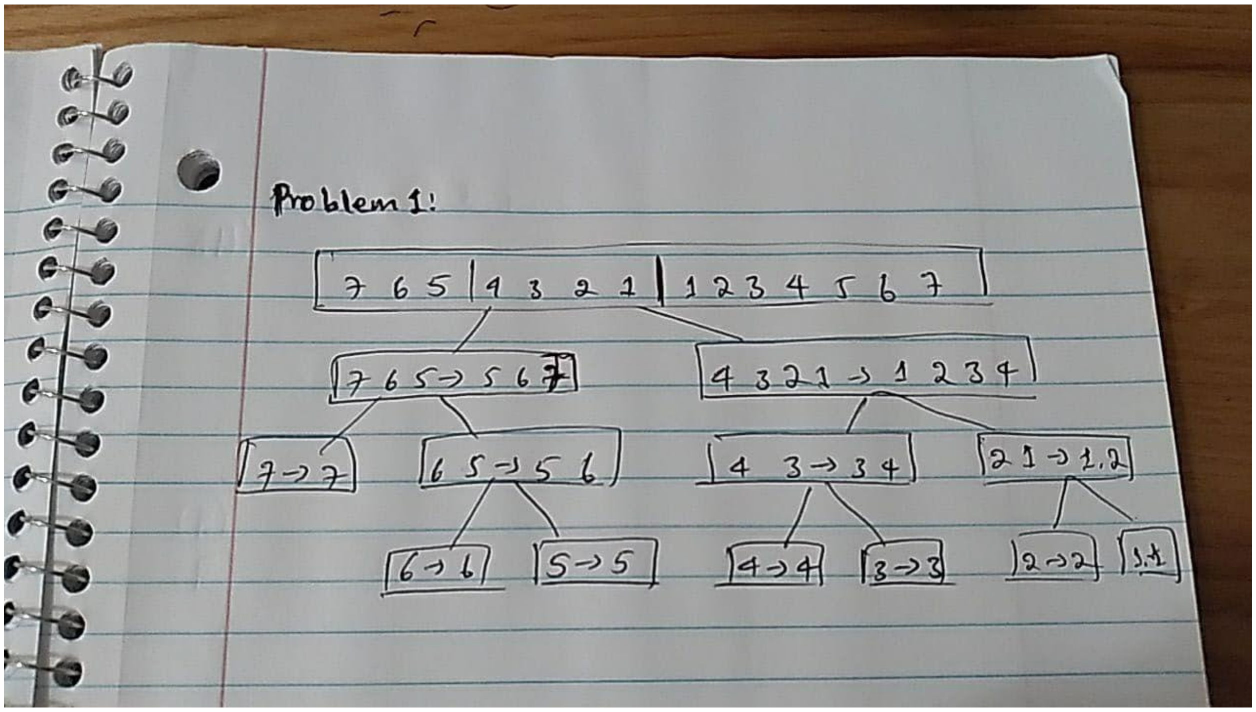
1.Perform the MergeSort algorithm by hand on the list **[7, 6, 5, 4, 3, 2, 1]** using a MergeSort recursion tree, as was done in the lecture.



4.

```
Algorithm fibIterative (n):
    Input: integer n
    Output: Fibonacci number of n or -1 (for invalid input)
    if n < 0 then
        return -1
    else if n = 0 then
        return 0
    fibNMinusTwo ← 0
    fibNMinusOne ← 1
    fibN ← fibNMinusTwo + fibNMinusOne
    for k ← 2 to n do
        fibN ← fibNMinusOne + fibNMinusTwo
        fibNMinusTwo ← fibNMinusOne
        fibNMinusOne ← fibN
    return fibN
```

The running time of the iterative Fibonacci is O(n) where n is the value of the input.

```
Algorithm fibRecursive (n):
    Input: Integer n
    Output: Fibonacci number of n if exists
    return fibRecursiveImpl(n, null)


Algorithm fibRecursiveImpl (n, fibs):
    Input: Integer n; HashMap fibs containing already computed Fibonacci
    Output: Fibonacci number of n or -1 (for invalid input)
    if n < 0 then return -1
    else if n = 0 then return 0
    else if n = 1 then return 1
    if fibs = null then fibs = new HashMap
    else if fibs.containsKey(n) then return fibs.get(n)
    fibN ← fibRecursiveImpl(n – 1, fibs) + fibRecursiveImpl(n – 2, fibs)
    fibs.put(n, fibN)
    return fibN
```

The above algorithm runs in O(n) but has a space complexity of O(n). The hash-map used provides a runtime efficiency but takes O(n) space.

5. Using the same technique as secondSmallest problem we can indeed solve thirdSmallest problem. We can store the first, second and third smallest numbers in three separate variables and iterate through the loop one time to determine the thirdSmallest element. This algorithm runs in O(n) time.

Sorting the given array and locating the k_{th} smallest element can be an easy and straight forward task but increases the running time of the algorithm to be (n log n). This is introduced thanks to our sorting procedure. Depending on the sorting algorithm, the the space complexity of our algorithm might get O(n) too.

One optimization for finding the k_{th} smallest element from the array can be achieved by the use of heap data structure. We can build a max-heap of size k (can be either in place (notice it will change the given array into arbitrary form) or new one) and heapify (a process of ensuring our max-heap properties are kept) the first k elements of the original (given array). Then we can show that a running time of O(n) is achievable. Let's dive into the algorithm

```
Algorithm findKthValue (arr, k):
    Input: An array of size n; k a positive integer between 2 and n-1
    Output: The k^th smallest value
    heap = buildHeap (arr, k)
    for j ← k to n – 1 do
        if arr[k] < heap[0] then
            heap[0] = arr[k]
            heapifyDown(heap, 0)      // maintain heap property starting from root
    return heap[0]       // max element of the heap (k^th smallest element)
```

The abstractions of building heap (buildHeap) and maintaining heap property from a given index down (heapifyDown) can be explained in detail.

HeapifyDown: This method iterates starting the given index down to the child nodes until a max-heap property is satisfied for the given number. Hence at most, this method takes O(log n) since log n is the height of a heap.

BuildHeap: This method builds a heap starting from the last index and applying the heapifyDown method (to maintain heap property) and makes its way to the root. It is proved using Taylor Series that this operation is going to take O(k) where k is the size of the heap.

Hence, the total running time can be calculated as (given n = arr.length and k)
O(k) + O((n – k) * log(k)) + c where c is some constant. Our assumption is that k >= 2. At worst, this algorithm runs in O(n) time.