CSC265 Homework 1

Question 1 (Mark: 8/8)

Part a. (Mark: 4/4)

The tight asymptotic upper bound on the worst-case running time of FUNC(x,M) is $O(n)$. This function traverses a square matrix starting from the top-right corner and either left or down depending on an comparison with x. The function stops when it either finds x or reaches the left or bottom edge of the matrix. Since we are looking for worst-case running time, we disregards cases with early return and look for when x is not in the matrix.

Hence, in the worst case, the algorithm will have to reach $x = n$ and $j = 1$. Since $i = 1$ and $j = n$ before the while loop starts, and for each iteration either i goes up by 1 or j goes down by 1, we have that before a maximum of $2n$ iterations is finished, one of the conditions must become false the function returns. Within each iteration, at most three comparisons and one variable assignment are executed, all of which take constant time of $O(1)$.

Hence the worst-case running time is $O(4 \times 2n)$ which can be simplified to $O(n)$ asymptotically.

Part b. (Mark: 4/4)

Let $n \geq 1$ be arbitrary integer. Let $x = 2$, and set all matrix entries (i, j) to 3 if $i = n$, and 1 otherwise. Note that such matrix is non-decreasing, satisfying the requirement from both left to right and top to bottom for each row and column. For instance, a 3×3 matrix would look like this:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 3 & 3 & 3 \end{bmatrix}$$

With this constructed matrix M, if we call FUNC(2,M), since no entries is equal to 2, the code will never enter the first if-block. It starts at the top right corner $(1, n)$, since this entry is 1, which is smaller than $x = 2$, the first else-if is true, and we have $i = i + 1$ and so it goes the entry $(2, n)$, or the entry directly below the last one. Continuing in this way, with how this matrix is defined, we see that this first else-if will be true for the first $n - 1$ comparisons, and only on the n-th comparison where we are at entry (n, n) that the while loop will go to the second else-if. Since 3 is greater than 2, we have that the while loop will run n more times where each time j is decreased by 1 until it reaches $j = 0$ and the while loop stops.

Hence, the code executes a total of $2n - 1$ iterations, with each iteration taking constant time, resulting in a time complexity of $O(2n - 1)$. Asymptotically, we can discard constants and replace nonzero coefficients with 1, simplifying it to $O(n)$ run-time. Since this holds true for any integer $n \geq 1$, our worst-case analysis is indeed a tight bound.

Question 2 (Mark: 12/12)

Part a. (Mark: 6/6)
In this algorithm, we will utilize an array-based min-heap, which provides constant-time access to elements by their indices. The min-heap will refer to the set 'S' described in the problem, and have size k. To search for elements in the min-heap by their *id*, we must maintain an array 'record' that tracks the position of each element within the min-heap.

To achieve this, we will introduce a new class **Item** which contains three attributes, id, count, and pos. The first two attribute refers to the same id and count as in the question, whereas the pos refers to the position of that *id* in the min-heap.

The min-heap stores **Item** and compare them based on their value of *count*.

The min-heap will consist of three methods: 1. **insert(item)** adds an item to the heap and then heapify it. 2. **get(i)** returns the element at the i-th position without modifying the heap. 3. **increaseKey(i, dcount)** increases the *count* of the element at the i-th position by *dcount* in the list and then heapify the heap. Note: even though heapify was not explicitly named in class, it was repeatedly used in various operations for a heap, furthermore, we can take reference from Chapter 6 section 2 of CLRS which explained Max-Heapify and proved its running to be $O(\log k)$.

In addition to the standard implementation of these methods, we must maintain the accuracy of the 'pos' attribute for each element in the heap. Given the implementations covered in lectures, all heap modifications in the mentioned methods consist of swapping elements and placing new items in empty positions. Consequently, updating the 'pos' attribute of the item during each swap or placement ensures that it accurately reflects its current position within the heap.

Now with enough preparation, we can continue on to the main function

```
Item[] record = new Item[1...n];
MinHeap minHeap = new MinHeap(size = k);
int currentSize = 0;
for i = 1 to m {
    if (record[A[i].id] != null) {
        minHeap.increaseKey(record[A[i].id].pos, A[i].count);
    } else if (currentSize <= k - 1) {
        Item item = new Item(A[i].id, A[i].count);
        minHeap.insert(item);
        record[A[i].id] = item;
        currentSize++;
    } else {
        Item item = minHeap.get(1);
        record[item.id] = null;
        item.id = A[i].id;
        record[item.id] = item;
        minHeap.increaseKey(1, A[i].count)
    }
}
```

Now, we can establish the correctness and prove that the worst-case complexity is $O(n + m \log k)$.

We first prove a few loop invariants:
1. The number of non-null items in 'minHeap' equals to 'currentSize'. Before the loop, minHeap is empty and currentSize = 0. Inside the loop, neither are changed in the 'If' or the 'else' block. In the 'else-if' block, minHeap inserts an new element, and currentSize is increased by 1. Therefore this invariant holds.
2. The i-th entry of 'record' points to an item if and only if item.$id = i$. Before entering the loop, this condition holds true since there are no non-null entries in 'record', and there are no items. Hence, our focus lies in monitoring new item creation or ID changes. In the 'else-if' block, an item with $id = A[i].id$ is created, and we set 'record[A[i].id] = item.' In the 'else' block, the *id* is changed from the original ID to a new one. We set 'record[original id]' to null

and 'record[new id]' to item. Therefore, this invariant holds.

Based on the invariants we have established and assuming the proper implementation of the methods in the min-heap, the correctness of the 'if' and 'else-if' blocks is evident, as they directly correspond to those in the original problem statement. Regarding the correctness of the 'else' block, it's essential to note that in a well-implemented non-empty array-based min-heap, the first entry always holds the minimum value. The non-empty condition is ensured by the 'else-if' block's condition and the fact that k is a positive integer.

Hence, we have established the correctness of the 'for-loop', with 'minHeap' containing the desired elements that should be in 'S'. Consequently, at the end of the function, we can simply convert 'minHeap' into a set and return it. Finally, we proved the correctness of the function.

For the rest of the part a), for simplicity sake, we will take all time complexity as the *worst-case time complexity*. Lets first look at the running times of the three methods which we've used in our implementation.

We first calculate the time complexity for the methods in min-heap:

1. The **Insert** operation has been proven in lectures to have a worst-case time complexity of $O(\log k)$, where k is the number of nodes in the heap. During the process, a maximum of $O(\log k)$ swaps are executed. In our modified version, we assign two values to the 'pos' variable of the item with each swap, resulting in an additional $O(2 \log k)$ complexity. When combined, the overall complexity remains $O(\log k)$ in asymptotic terms.

2. The **increaseKey** operation initially assigns a value to an entry in the heap, which takes constant time. Then, we may perform swaps with parent or children to maintain the heap. In this case, since each A[i].count is positive, entries can only swap with their children. Hence, a maximum of $O(\log k)$ swaps are executed. Similar to the analysis for **Insert**, the complexity of our modified version remains $O(\log k)$.

3. The **get** operation simply involves accessing and returning an entry of the array, which has a constant time complexity of $O(1)$.

Now, we calculate the time complexity for function:

Starting from line one to line three, the initialization phase consists of a variable assignment which takes constant time and the initialization of two arrays of length n and k, which takes $O(n + k)$ time. So the initialization phase takes $O(n + k)$

The for-loop executes exactly m times and consists of three conditionals, since they are all comparisons, all of them take constant time to complete. The body of the first conditional calls **increaseKey** thus it is $O(\log k)$. Furthermore, since instantiation, assignment, and basic arithmetic are all $O(1)$, the body of the second conditional is also $O(\log k)$, and similarly for the third conditional, line 13-16 takes constant time, and line 17 which calls **increaseKey** take $O(\log k)$. Since the three conditionals are a series of if and else ifs, only one of them can happen each iteration. Hence the total running time of the for loop is $m \cdot O(\log k) = O(m \log k)$.

Finally, converting 'minHeap' into a set involves calling minHeap.get() k times, and appending non-null value to the set, which takes constant time and so overall contributes to $O(k)$.

Summing up the complexities of each phase, we obtain a worst-case time complexity of $O(n + k + m \log k + k) = O(n + m \log k + 2k)$. Assuming $0 < k \leq n$ so $0 < 2k \leq 2n$, we have $n + m \log k \leq n + 2k + m \log k \leq 3n + m \log k$, which simplifies to $O(n + m \log k)$ asymptotically, as desired. (Note: Prof Nikolov commented on piazza explained we can assume $k \leq n$, see question @9)

Part b. (Mark: 6/6)

While my explanation and proof may be lengthy, I've included relevant pseudo-code for your reference, with the hope that it simplifies your work :)

We will be using a linked-list-style data structure in this part. Before describing the algorithm, let's first clarify how we've implemented the linked list. Each node in the list contains four attributes:

1 & 2: id & count: which refers to the same attribute in the question.
3: left: This attribute points to the node on the left in the list.
4: right: Similarly, this attribute points to the node on the right.

Unlike the typical linked list, we employ an empty node as the list's starting point. This design simplifies the process of removing nodes from arbitrary positions in the list (since we'll never remove the empty node).

Furthermore, we have implemented two methods. The first method is 'remove()', which removes the node from the list. It's important to note that this method is only called when the node is not the first node (i.e., the empty node), as we never remove the empty node.

The second method is 'add(node)'. This method is invoked only when the current node is the first node. It inserts the new node into the list as the first nontrivial node.

```
class Node {
    public int id;
    public int count;
    public Node left;
    public Node right;

    remove() {
        left.right = right;
        if(right != null){
            right.left = left;
        }
        left = null;
        right = null;
    }

    add(Node newNode){
        if (right != null) {
            newNode.left = this;
            newNode.right = right;
            right = newNode;
            newNode.right.left = newNode;
        } else {
            right = newNode;
            newNode.left = this;
        }
    }
}
```

Here is an example code for initializing the list and adding a node.

```
Node list = new Node(); //Initialize the list

Node firstNode = new Node(); firstNode.id = 1; firstNode.count = 1;
//Initialize the nontrivial node that we're adding to the list

firstNode.left = list;
list.right = firstNode; //Added to list
```

Now, let's begin describing the algorithm.

We first initialize two variables: the first is $size = 0$, which counts how many items we have recorded, and the second is $floor = 0$, indicating that every item x we've recorded satisfies $x.count \geq floor$. Next, we initialize two arrays, namely record[1...n] and set[1...m]. We then initialize set in such a way that each entry holds an empty node.

The list 'record' contains all the nodes we have recorded, where the k-th entry is either null, indicating that we haven't recorded an item with $id = k$, or it holds the node with $node.id = k$. The list 'set' holds a linked-list in each entry, where the k-th entry contains all nodes in 'record' such that $node.count = k$.

```
Node[] record = new Node[1...n];
Node[] set = new Node[1...m];
for i = 1 to m {
    set[i] = new Node();
}
```

Then, we leave the for-loop in the question unchanged, and work on the 'if', 'if-else' and 'else' block.

In the 'if' block, we employ the condition 'record[A[i].id] $\neq$ null'. If this condition is met, we can access that node and increment the *count* by 1. However, to ensure the proper functioning of 'set', we should remove the node from the 'set[*count*]' list and add it to the 'set[$count + 1$]' list.

In the 'if-else' block, we utilize the condition 'size $\leq$ k - 1'. If this condition holds true, we create a new node with id =A[i].id and $count = 1$. To ensure the correct operation of 'set,' 'record,' and 'size,' we append it to the list 'set[1],' set 'record[A[i].id] = node,' and increment 'size' by 1.

In the 'else' block, our objective is to locate a node with the minimum *count*. To achieve this, we increment the value of 'floor' until 'set[floor]' contains nontrivial nodes. Once we identify the first nontrivial node in 'set[floor],' we remove it. Since we will be modifying its id, we set 'record[node.id] = null,' then assign 'node.id = A[i].id' and set 'record[node.id] = node.' Subsequently, we add it to 'set[count+1]' and increment its *count* by 1.

After the for-loop, we convert 'record' into a set and return it.

```
for i = 1 to m {
    if (record[A[i].id] != null) {
        Node currentNode = record[A[i].id];
        currentNode.remove();
        currentNode.count++;
        set[currentNode.count].add(currentNode);
    } else if (size <= k - 1) {
        Node currentNode = new Node(A[i].id, A[i].count);
        record[A[i].id] = currentNode;
        set[currentNode.count].add(currentNode);
        size++;
    } else {
        while (set[floor].right == null) {
            floor++;
        }
        Node currentNode = set[floor].right;
        currentNode.remove();
        record[currentNode.id] = null;
        currentNode.id = A[i].id;
        currentNode.count++;
        record[currentNode.id] = currentNode;
        set[currentNode.count].add(currentNode);
    }
}
```

Finally, let's demonstrate the correctness and calculate the worst-case running time.

We begin by analyzing the methods in the Node class. These methods, devoid of loops and consisting solely of variable comparisons and assignments, exhibit a time complexity of $O(1)$.

For the 'remove()' method, correctness can be established through two cases. We assume 'this' is not the first node, if 'right' is null, we simply set 'left.right' to null, thereby removing 'this' from the list. If 'right' is not null, we link 'left.right' to 'right' and 'right.left' to 'left', effectively detaching 'this' from the list. Finally, we set 'left' and 'right' to null, completely disconnecting the node from the list. Therefore, 'remove()' is indeed correct.

Similarly, for the 'add(node)' method, we assume 'this' is the first node. If 'right' is null, we set 'right = node' and 'node.left = this' to insert it into the list. If 'right' is not null, we set 'node.left = this' and 'node.right = right' to attach it to the list. Then, we set 'right = node' and 'node.right.left = node' to ensure that the list elements are properly linked to 'node'. Hence, add(node) is indeed correct.

Let's establish the correctness of the algorithm. First, we prove a few loop invariants:
1. The number of nontrivial nodes in 'set' is equal to the number of nodes in 'record', which is also equal to 'size'. In the 'if' block, if entered, 'size' and 'record' remain unchanged, and 'set' has one node removed and one added, so the invariant holds. In the 'if-else' block, all three increase by one, maintaining the invariant. In the 'else' block, 'size' remains unchanged, and both 'record' and 'set' have one node removed and one added, again preserving the invariant. This invariant implies that the condition of the 'else-if' block is correct.
2. The list 'set' maintains a linked-list in each entry, with the r-th entry containing all nodes for which $node.count = r$. In the code, whenever 'count' increases by 1, the node is relocated from its current entry to the next. Thus, the invariant is upheld.
3. Every recorded item, denoted as x, always fulfills the condition $x.count \geq floor$. Initially, this holds when $floor = 0$. The value of 'floor' increases only when 'size = k' becomes true, with 'size' being non-decreasing, we have $x.count$ is non-decreasing for each x. Assume the invariant holds, to allow 'floor' to increase by 1, it's essential for 'set[floor]' to contain no nontrivial nodes. Consequently, the invariant holds.

Since each A[i].$count = 1$, $x.count$ is at most m for each x. Therefore , the list 'set' indexed $1 \dots m$ is well-defined.

Based on the previously established invariants, the correctness of both the 'if' and 'else-if' sections is evident. Because they perform essentially the same actions as in the 'FUNC' function while maintaining the loop invariants.

In the 'else' part, we have $size > 0$ since $k > 0$. According to the loop-invariant, there exists some $r > 0$ such that 'set[r]' is a nontrivial list, and $floor \leq r$. Therefore, the 'while' loop terminates, and based on the termination condition, 'set[floor]' is nontrivial. As per the loop-invariant, all nontrivial nodes in 'set[floor]' all have the minimal 'count' among all nodes in 'set'. Thus, by labeling the first one as 'node', we instantiate the 'x' as described in 'FUNC'. The remaining code in the 'else' part executes the same actions as in the 'FUNC' function while preserving the loop invariants. Consequently, the 'else' part is also correct.

Hence, the 'for-loop' is correct. Subsequently, since 'set' and 'record' contain identical nontrivial nodes, we can convert 'record' into a set and return it, yielding the correct result.

Now lets calculate the running time. The initialization phase creates arrays of length m and n, contributing to $O(m + n)$. Each entry in 'set' initializes a new node in constant time. With m entries, this adds another $O(m)$. Therefore, the overall complexity is $O(m + n)$.

The for-loop executes precisely m times. Excluding the 'while-loop', all actions within the for-loop, such as variable comparisons, assignments, method calls in the Node class, and array access, are completed in constant time. The total time for each iteration of the for-loop is therefore $O(1)$. Consequently, the for-loop itself takes $O(m)$ time, without considering the 'while-loop' within. Now, let's analyze the 'while-loop' separately. As $floor$ is non-decreasing, bounded by a maximum of m, and increases by 1 each time it enters the 'while-loop', we can conclude that the 'while-loop' executes at most m times. Since each iteration takes constant time, it contributes $O(m)$. Considering both, the entire 'for-loop' takes $O(m)$ for worst case.

Finally, converting 'record' into a set involves iterating through 'record' and appending each entry to the set, which takes constant time per entry and overall contributes to $O(n)$.

Summing everything up, we conclude that the worst-case running time of my algorithm is $O(m + n)$ as desired.