Algorithm Design for Creating a National COVID-19 Advisory Committee

1. Algorithm Components

1.1 Components

A. initialInput() – Input Parsing

- Complexity: O(n^2)
- Purpose: Is to take in the user input strings using the Scanner class and converts them into integers which represent ids so that they can be added to their appropriate lists using nested loops. Specifically, the user input is read until the number of groups is reached meaning there is no more inputs from the user expected. One of the lists takes all the integers, whereas another list is created with each iteration of the outer loop which stores the values until it reaches 0 meaning the end of a line where the inner loop is broken out of so that the next set of ids can be stored. The list containing all the inputs is used for putting each id into a hash map where the key represents the id, and the value holds how many times that id occurs throughout the input. The main function of the hash map is to determine which ids have duplicates which is done by checking if the hash map contains a key that is in list of all the inputs. If it does then the value of that key is stored, incremented, and put back into the hash map updating the key. If it doesn't then the id is put into the hash map with a value of 1 to mark the 1st occurrence.

B. repeats() – ID picking

- Complexity: O(n)
- Purpose: The intention of this function is to loop through the hash map which contains the id keys and the value of how many times that id occurs to find the ids with the greatest number of repeats. Through each iteration the values will check to see if they're greater than 1 and greater than or equal to a variable which if so, will update that variable to that value and will be compared against the next value. These greater than or equal values will then be added to a list and the keys (ids) associated to these values will be added to separate list. Additionally, any values that are less than the compared variable, but greater than 1 (meaning they have a duplicate) will be stored in an alternative list as a precautionary action if the list containing lists still has any lists unaccounted for. Finally, the array list holding the values of repeats is sorted so that the list will be in descending order thus, it can be determined whether the input had ids with varying amounts of repeats or if all ids had the same number of repeats. If there was a variation, then the list containing ids from the hash map would also be sorted in descending order for a natural order of numbers.

C. mergeSort() – Sorting

- Complexity: O(n log n)
- Purpose: To sort ArrayLists in descending order.

D. remove() – Finding IDs and Setting Lists Associated to Empty

- Complexity: O(n^3)
- Purpose: Now containing the ids that need to be found the remove function will loop through the list of chosen ids with a nested loop that will iterate through array list of array lists which represents each set of ids. Using the contains method from the ArrayList library this allows each subset to be checked for the current id. If a match is found then a checker variable will increment, when it is 1 a variable will hold the current index for the subset. If it becomes 2 then the index of the initial list found when checker is 1 and the current list are set to empty lists and that chosen id is put into a final result list. Any id found in any other sub-list will result in the checker variable incrementing past 2 resulting in those lists being set to empty. This functionality was intended as through testing it indicated that some sets of data would contain ids with the same number of duplicates meaning that they were all valid ids to be removed. However, just removing the first list that contained the current id did not return the optimal selection of ids but rather a large majority of lists as ids that may have had 2 or more occurrences may now have 1 match but are still considered valid due to initially having 2 or more repeats. Hence, checking if that id has 2 or more repeats before setting them to empty lists is vital to having the optimal output. A precautionary worst-case check is carried out on the overarching lists sub-lists which checks if any of them are not empty, meaning the ids with the greatest number of duplicates did not find a match in some sub-lists. In this case the triggered Boolean will allow code which loops through an alternative list which contains ids that have less repeats than the greatest ids but more than one repeat and do the same checks prior to find the optimal id among the

lesser ids. Additionally in the case of there being only 1 remaining sub-list where the checks will not properly apply as there will only be 1 instance of the id being contained within the sub-lists the temporary variable holding the index of the single matched list will be stored. The variable will be used for a constant access point to the sub-list left where both lists of greater and lesser ids will be checked to see which id first matches to that list and then is added to the result list with the singular sub-list being set to empty.

E. result() – Printing out Results

- Complexity: O(n)
- Purpose: Takes an ArrayList input and uses that list to print required outputs, number of ids and which ids.

1.2 Total Time Complexity

Based on each component of the program the total complexity is:
O(n^2) + O(n) + O(n log n) + O(n^3) + O(n)
Total = O(n^3), Worst Case = O(n^3)

2. Justification

This program is effective in solving the problem of selecting a minimal number of ids which accounts for each group as it selects the ids which have the greatest amount of repeats, loops through the list of lists finding the lists which contain those ids, setting those lists to empty, adding the id to a result lists all until each sub-list is empty. This solution incorporates storage methods of hash maps to add ids and the frequency of times they occur concurrently and ArrayLists to easily store/manipulate sets of unique and essential data. Hence, why in some cases of the program the complexity is O(n^3), O(n^2) as the process of storing and accessing the crucial data requires nested loops and use of class functions like contains. The incorporation of mergeSort was essential to achieve the desired functionality of having the ids with the most occurrences picked. It was noticed that having the ids picked by the greatest amount of occurrences each iteration resulted in an ascending order of repeat values and ids meaning when being sent to the remove function the first great id would be picked but not the largest id as it is at the end of the list. Attempts at trying to reverse the list using the sort function from collection or looping from the end of the list to the beginning were tried, however only resulted list out of bound errors or the wrong output for certain tests. Implementing mergeSort based of [1] and modifying it slightly as to not sort in an ascending order but rather a descending order allowed the list containing values to be in the desired format of descending and a list containing ids to be organised in a natural fashion close to the idea of a reversed list. Once tested it provided results which were aligned with the scope of the program. Further testing of the program spotted issues which highlighted scenarios where this algorithm did not provide the most optimal selection of ids but still performed admirably with a result which had a small selection of ids that spanned across each subset and thus meeting the requirements specified. Ids which had the greatest amount of duplicates at the beginning of the input resulted in there not being enough ids to cover each sub-list. The results of these cases resulted in the precautionary worst-case checks and responses being developed where if there are still subsets that aren't empty then using a list of ids that don't contain most duplicates is used to find an id in those remaining lists so they can be set to empty and thus provide an output that satisfies the guidelines. Further testing of this worst-case outlined an edge case what if there was a singular remaining list? However, through the appropriate checks result in whichever id from either the greatest or lesser ids matches 1st. Conclusively, this algorithm implements a lot of tools to successfully achieve desired results, it finds the ids with the greatest amount of repeats and ensures it picks the appropriate ids from the subsets even after each set is made empty potentially voiding a valid id. Providing an optimal output, it covers for edge cases and worst-case scenarios which would normally result in a wrong output from the program if unaccounted for. With an overall complexity of O(n^3) each component has been determined necessary for providing an efficient and accurate program.

3. References

[1]N. Patel, "Merge sort using ArrayList in Java with example - withexample.com", withexample.com, 2017. [Online]. Available: https://www.withexample.com/merge-sort-using-arraylist-java-example/. [Accessed: 27-Aug- 2021].