Dhirubhai Ambani
Institute of Information and Communication Technology

Subject: DATA STRUCTURES

Subject code: IT 205

Capstone Project

Date of submission:**07/04/2024**

Due date of submission:**07/04/2024**

Group Members:

Yammanuru Sai Sreepadha	**202301042**
Siva Suhas Thatavarthy	**202301050**
Sri Sai Madhava Teja Yanduri	**202301138**
Viswa Vignan Baddula	**202301160**

Our Analysis of the Question:

The question allocated to us is P-9 (TV Channel Scheduler)

1. The task is to create a **TV channel scheduler** for a family with **N** members.

2. We consider that the TV series are **pre-recorded** rather than **live streaming**, each lasting **one hour**.

3. A conflict arises if the TV both records and live-streams content simultaneously.

4. Each family member has a set of favourite series (**S**) from a pool of available series (**M**).

5. Family members can have **overlapping interests** in series.

6. Each member's favourite series has **equal priority**.

7. Input includes the name of the **family member**, their **availability slots**, and their **favourite series** set.

8. Each member must be allocated at least **p** slots from their favorite series.

How we tried to implement the required parts of the question:

We divided the question into 3 parts

1. File handling of the data inputted:

We tried to take the input data from the **Excel sheet(xls)** directly and give it to the code, but we were not able to use the Excel libraries in the C++ library. We considered taking the input from Excel by converting the file into a **CSV (comma-separated value) file.**

2. Data structures to be used to store input data:

We tried to explore different data structures to be considered for the storing of the input data for easy access into the algo.

We tried out different data structures like **vector, list**.

3. Creating of the algo :

We tried to address how the flow of the data should be, how to access the data during the scheduling of the TV series for each time slot, and also different cases in the scheduling of the series.

The algorithms we tried to implement:

We tried 3 logics out of which we have the code for logic 2 and logic 3.

Logic 1:

Input Files:

1. Series.csv: which stores the series data in the form of an Excel sheet and CSV converted.

2. Slot.csv: which stores the fields like names and availability slots and interested series of the person.

Index:

1) p – Minimum number of times at least a person should be on time. It means a person should be at least notified by 'p' time.

2) T – Total number of timeslots a particular person is available in 1 day.

3) M – Total number of series.

4) S – Set of favourite series for a particular person.

5) N – Number of people in a family.

6) pr - Number of timeslots a particular person is already notified from T.

Data Structures: -

Hash Table (Hashtable): Stores time slots **(1st column)** and person availability for that time slot. **Size:- 24 rows and (n+1) columns.**

Linked list (List): Stores series (1st node), number of members watching it (L) (2nd node) and their names.

Linked list (Store): Stores timeslot (1st column), series name (2nd column) and members watching it.

Reason:

In the above case, the Hash table is used over the array due to mapping issues. Linked lists are used due to dynamic allocation.

Algorithm:

Read the data of time slots and store it in 'Hashtable' and data of series and stores it in 'List'.

Case 1: - If **p < pr** for at least 1 person

(A) If no person is present in the time slot 'Hashtable' àstore NULL in 'Store'

(B) If only 1 person is present in the time slot 'Hashtable'à Check 'List' for a person with L=1

(a) If foundà store in 'Store'

(b) Elseà check all the 'Lists' in which person is presentàstore the series with the least L in 'Store'

(C) If more than 1 person is present in the time slot of 'Hashtable'

1. Person priority: - Val=T + pr - p

(a) Give priority based on 'Val'

(b) If two persons have the same 'Val' à check 'Hashtable' for a particular person with individual timeslots(B)àGive priority to a person with less number of such timeslots

(c) If the above priority is sameà check the favourite series with L=1 of the person in 'Lists'àgive priority to one with more such 'Lists'

2. Series allocation: -

(a) Take favourite series of most priority person (Prior)àfind sum of priorities (Sum) of persons with that series as their favourite series.

(b) If Sum> Priorà Allocate that series and persons with it as favourite series in 'Store'

(c) Elseà check if his favourite series has L=1

(i) If found, store person and series in 'Store'

(ii) Else, store the favourite series with least length and person in 'Store'.

Case 2: - For all members pr>= p

(A) For remaining time slots à take a series ‘List’ and check if it is most common series for timeslot in ‘Hashtable’

(B) Store persons with it as favourite series and the series in ‘Store’

Display the timeslot, name of persons and name of series as a notification.

Cases where algorithm works:-

It works efficiently when at least a member is not notified p times. It covers the cases where timeslot is empty, only 1 person is available for timeslot and when multiple members are available for timeslot.

Cases when algorithm doesn’t work: -

Algorithm covers the case where all members are notified at least p times but it doesn’t work efficiently. Total misses for each person can’t be calculated.

Pseudocode

```
// Initialize data structures

Hashtable timeSlotAvailability // stores time slots and person availability

List seriesList // stores series and number of members watching it

Store allocationStore // stores time slot, series, and members allocated


// Function to read data from CSV files and populate data structures

function ReadDataFromCSV():

   ReadTimeSlotsFromCSV("Slot.csv", timeSlotAvailability)

   ReadSeriesFromCSV("Series.csv", seriesList)


// Function to allocate series to time slots

function AllocateSeriesToTimeSlots(p):
```

```
        for each timeSlot in timeSlotAvailability:
        pr = number of times person is already notified from T for each person in timeSlot
        if any person's pr < p:
        if no person present in timeSlotAvailability[timeSlot]:
          allocationStore.store(timeSlot, NULL)
        else if only one person present in timeSlotAvailability[timeSlot]:
        person = findPersonWithLeastWatchedSeries(seriesList)
          allocationStore.store(timeSlot, person)
        else:
          prioritizeAndAllocate(timeSlot, p)
        else:
       allocateRemainingTimeSlots()


// Function to prioritize and allocate series for time slots
function prioritizeAndAllocate(timeSlot, p):
        priorityMap = {} // Map to store person priority
        for each person in timeSlotAvailability[timeSlot]:
        priority = T + pr - p // Person priority calculation
      priorityMap[person] = priority
  sortPersonsByPriority(priorityMap)
        mostPriorityPerson = getMostPriorityPerson(priorityMap)
        favoriteSeries = mostPriorityPerson.favoriteSeries
        sumOfPriorities = sumOfPrioritiesForSeries(favoriteSeries)
        if sumOfPriorities > priority:
      allocationStore.store(timeSlot, favoriteSeries, personsWithFavoriteSeries)
```

```
        else:
    allocateSeriesForLeastWatchedPerson(timeSlot, favoriteSeries)

// Function to allocate series for the least watched person
function allocateSeriesForLeastWatchedPerson(timeSlot, favoriteSeries):
        leastWatchedPerson = findLeastWatchedPersonForSeries(seriesList, favoriteSeries)
  allocationStore.store(timeSlot, favoriteSeries, leastWatchedPerson)

// Function to allocate remaining time slots
function allocateRemainingTimeSlots():
        for each timeSlot in timeSlotAvailability:
    mostCommonSeries = findMostCommonSeriesForTimeSlot(timeSlot)
    personsWithMostCommonSeries = findPersonsForSeries(mostCommonSeries)
    allocationStore.store(timeSlot, mostCommonSeries, personsWithMostCommonSeries)

// Function to display notifications
function DisplayNotifications():
        for each entry in allocationStore:
    display(entry.timeSlot, entry.series, entry.persons)

// Main function
function main():
        ReadDataFromCSV()
        p = GetMinimumNotificationsRequired() // Get minimum number of notifications
required
```

```
        AllocateSeriesToTimeSlots(p)

    DisplayNotifications()


// Run the main function

main()
```

Time complexity:

1) **ReadDataFromCSV() :** Takes linear time as to store n elements. So time complexity is **O(n).**

2) **AllocateSeriesToTimeSlots(p):** It iterates over every timeslot and person. It either calls priortizeAndAllocate() or allocateRemainingTimeSlots() which have different values. So time complexity is **O(m*n*k)** where m= number of time slots, n= average number of people in each slot and k= time complexity of inner function.

3) **prioritzeAndAllocate(timeslot, p):** This person iterates over every person in the time slot and sort them using priority. So time complexity i**s O(nlogn)**.

4) **allocateRemainingTimeSlots():** The function iterates over each time slot and allots to persons in it. As both takes linear time, the time complexity is **O(m*n)** where m= number of time slots and n= numbers of persons on average in it.

5) DisplayNotifications(): The functions iterates over entire members stored in allocationStore. So time complexity is **O(n)** where n is number of members in it.

6) main(): AllocationSeriesToTimeSlots(p) dominates in the main() function. So time complexity is **O(m*n*k).**

Space complexity:

1) Hashtable timeSlotAvailability: Each time slot(m) stores average number of members(n) all of with same space. So space complexity is **O(m*n).**

2) **List seriesList:** It stores number of series present. So space complexity is **O(s)** where n is number of series.

3) **Store allocationStore:** Each row stores time slot, series and members which is constant for all members in it. So space complexity is **O(l)** where l is number of members.

4) Overall space complexity is determined by above 3 space complexities. So space complexity is **O(m*n+s+l).**

Here all recursive function calls and other data types are negligible and are considered as constants. Also all the members in each slots and series are taken on average. So they don't disturb the complexities.

Logic 2:

Input Files:

Series.csv: This file stores the series data in the form of an Excel sheet and CSV converted.

Slot.csv: This file stores the fields like names, availability slots, and interested series of each person.

Data Structures Used:

Vector: Stores records of people and available series.

Unordered Set: Stores unique series names for each person and available series.

Unordered Map: Stores the schedule where keys are time slots and values are assigned persons with series.

Custom Linked List Node: Used to store persons with clashes in each time slot.

Variables Used:

struct person_record: Contains string person_Name and an unordered_set of strings Series_Names.

struct node: Contains all the variables of struct person and node* for iteration.

vector<person>: Stores all data of all people.

vector<string>: Stores all the series.

unordered_map<string, string>: Stores the schedule data.

Functions Used:

1.read_series_file():

Reads the data from an **input CSV file** and stores it in **vector<string> available_series.**

Working: Each series in the file is separated by a comma. The getline function is used with the delimiter as "," each time.

2.read_person_records():

Reads the data from an input CSV file and stores it in **vector<person> records.**

Working: The line is read using the getline function and later divided based on the **delimiter** used. The values (of different data types) are then stored.

Availability slots is an **array of size 24, per person, indicating availability at each hour of the day.**

3.print_available_series() and print_person_records():

Iterate through the loop and print the series and person records respectively.

4.scheduler_slots():

This function uses **available_series and records.**

It **iterates** through each hour of the day and checks each **person's availability.**

Nodes are created for **available persons**, and a **map is created to count the frequency of series.**

Allocation logic:

Case 1: Highest frequency series is allocated.

Case 2: If **multiple series** have the same frequency, **total series preference** is considered.

Case 3: If **no series preference** is available, the slot is **marked as free.**

Case 4: Single person is allocated.

After allocation, time slots are printed.

total_misses():

Calculates the total misses of the series for all people.

It **sums the remaining series** (in the series_list of each person) and prints the total misses.

Algorithm Capabilities:

The algorithm effectively allocates series for most cases.

Failed Cases of the Algorithm:

If **two or more series have the same frequency** and **number of total series to be watched**, the code **randomly assigns** one of the series.

The code may not **minimize misses optimally in all scenarios.**

The logic does not include the p value (person getting into atleast p) which is a imp.

Hence we tried a logic 3 in which it includes the p value.

Pseudocode:

```
STRUCT Person_Record
        STRING Person_Name
        SET Series_Names
        ARRAY time_slots[25]
```

```
STRUCT Node
        STRING Person_Name
        SET Series_Names
        ARRAY time_slots[25]
        Node next

FUNCTION printRecords(records: VECTOR[Person_Record])
        FOR record IN records
        OUTPUT "Person Name: " + record.Person_Name
        OUTPUT "Total Available Slots: " + record.time_slots[0]
        OUTPUT "Available Slot Timings:"
        FOR i FROM 1 TO 24
        IF record.time_slots[i] == 1
        OUTPUT "Hour " + i + ": Available"
        ELSE
        OUTPUT "Hour " + i + ": Not Available"
        ENDIF
        ENDFOR
        OUTPUT ""
        ENDFOR
ENDFUNCTION

FUNCTION printAvailableSeries(available_series: VECTOR[STRING])
        OUTPUT "Available Series:"
        FOR series IN available_series
        OUTPUT series
        ENDFOR
ENDFUNCTION

FUNCTION slot_file_read(records: VECTOR[Person_Record])
        OPEN file "final_input.csv"
        IF file NOT OPEN
        OUTPUT "Error opening file."
        RETURN
        ENDIF

        READ line FROM file

        WHILE READ line FROM file
        READ name, series_names, time_slots FROM line

        CREATE record OF Person_Record
        record.Person_Name = name

        WHILE READ series_name FROM series_names
        INSERT series_name INTO record.Series_Names
        ENDWHILE
```

```
        hour = 0
        WHILE READ slot FROM time_slots USING ','
        value = CONVERT slot TO INTEGER
        record.time_slots[hour + 1] = value
        hour = hour + 1
        ENDWHILE

        record.time_slots[0] = 0
        FOR i FROM 1 TO 24
        IF record.time_slots[i] == 1
        record.time_slots[0] = record.time_slots[0] + 1
        ENDIF
        ENDFOR

        INSERT record INTO records
        ENDWHILE

        CALL printRecords(records)
ENDFUNCTION

FUNCTION series_file_read(available_series: VECTOR[STRING])
        OPEN file "final_series_input.csv"
        IF file NOT OPEN
        OUTPUT "Error opening file."
        RETURN
        ENDIF

        WHILE READ line FROM file
        WHILE READ series_name FROM line USING ','
        INSERT series_name INTO available_series
        ENDWHILE
        ENDWHILE
ENDFUNCTION

FUNCTION find_and_update_clashes(records: VECTOR[Person_Record], schedule_map:
MAP[STRING, STRING], available_series: VECTOR[STRING])
        FOR hour FROM 0 TO 23
        slot_key = CONVERT hour TO STRING + ":00-" + CONVERT ((hour + 1) MOD 24) TO STRING +
":00"

        CREATE clashes OF LIST[Node]

        FOR record IN records
        IF record.time_slots[hour + 1] == 1
        CREATE newNode OF Node
        newNode.Person_Name = record.Person_Name
        newNode.Series_Names = record.Series_Names
```

```
FOR i FROM 0 TO 24
        newNode.time_slots[i] = record.time_slots[i]
ENDFOR
IF NOT newNode.Series_Names.EMPTY
        APPEND newNode TO clashes
ENDIF
IF newNode.Series_Names.EMPTY
        record.time_slots[hour + 1] = 0
        record.time_slots[0] = record.time_slots[0] - 1
ENDIF
ENDIF
ENDFOR

IF clashes.EMPTY
OUTPUT slot_key + ": Empty slot"
CONTINUE
ENDIF

IF SIZE OF clashes == 1
node = FIRST OF clashes
most_common_series = FIRST OF node.Series_Names
schedule_entry = node.Person_Name + " (" + most_common_series + ") "

node.time_slots[hour + 1] = 0
REMOVE most_common_series FROM node.Series_Names

FOR record IN records
IF record.Person_Name == node.Person_Name
        REMOVE most_common_series FROM record.Series_Names
        record.time_slots[hour + 1] = 0
        BREAK
ENDIF
ENDFOR
schedule_map[slot_key] = schedule_entry
OUTPUT "Time Slot: " + slot_key + " - Persons: " + schedule_map[slot_key]
ELSE IF NOT clashes.EMPTY
CREATE series_count OF MAP[STRING, INTEGER]
FOR series IN available_series
series_count[series] = 0
ENDFOR

FOR node IN clashes
FOR series IN node.Series_Names
        IF series_count[series] EXISTS
        series_count[series] = series_count[series] + 1
        ENDIF
ENDFOR
ENDFOR
```

```
        CREATE max_series OF VECTOR[STRING]
        max_count = 0
        FOR pair IN series_count
        IF pair.second > max_count
                max_series = [pair.first]
                max_count = pair.second
        ELSE IF pair.second == max_count
                APPEND pair.first TO max_series
        ENDIF
        ENDFOR

        IF SIZE OF max_series > 1
        CREATE series_slots OF MAP[STRING, INTEGER]
        FOR series IN max_series
                series_slots[series] = 0
        ENDFOR

        FOR node IN clashes
                FOR series IN node.Series_Names
                IF series_slots[series] EXISTS
                series_slots[series] = series_slots[series] + node.Series_Names.COUNT(series)
                ENDIF
                ENDFOR
        ENDFOR

        max_net_count = INT_MIN
        FOR pair IN series_slots
                net_count = pair.second - max_count
                IF net_count > max_net_count
                most_common_series = pair.first
                max_net_count = net_count
                ENDIF
        ENDFOR

        schedule_entry = ""
        FOR node IN clashes
                IF node.Series_Names.CONTAINS(most_common_series)
                schedule_entry = schedule_entry + node.Person_Name + " (" +
most_common_series + ") "
                node.time_slots[hour + 1] = 0
                REMOVE most_common_series FROM node.Series_Names
                FOR record IN records
                IF record.Person_Name == node.Person_Name
                REMOVE most_common_series FROM record.Series_Names
                  record.time_slots[hour + 1] = 0
                  record.time_slots[0] = record.time_slots[0] - 1
                BREAK
```

```
                ENDIF
                ENDFOR
                ENDIF
        ENDFOR

        schedule_map[slot_key] = schedule_entry
        OUTPUT "Time Slot: " + slot_key + " - Persons: " + schedule_map[slot_key]
        ELSE
        most_common_series = FIRST OF max_series
        schedule_entry = ""
        FOR node IN clashes
                IF node.Series_Names.CONTAINS(most_common_series)
                schedule_entry = schedule_entry + node.Person_Name + " (" +
most_common_series + ") "
                node.time_slots[hour + 1] = 0
                REMOVE most_common_series FROM node.Series_Names
                FOR record IN records
                IF record.Person_Name == node.Person_Name
                REMOVE most_common_series FROM record.Series_Names
                  record.time_slots[hour + 1] = 0
                  record.time_slots[0] = record.time_slots[0] - 1
                BREAK
                ENDIF
                ENDFOR
                ENDIF
        ENDFOR

        schedule_map[slot_key] = schedule_entry
        OUTPUT "Time Slot: " + slot_key + " - Persons: " + schedule_map[slot_key]
        ENDIF
        ELSE
        schedule_map[slot_key] = "No person free here"
        ENDIF

        FOR node IN clashes
        DELETE node
        ENDFOR
        ENDFOR
ENDFUNCTION

FUNCTION total_misses(records: VECTOR[Person_Record])
        total_misses = 0
        FOR record IN records
        total_misses = total_misses + SIZE OF record.Series_Names
        ENDFOR
        OUTPUT "Total misses: " + total_misses
ENDFUNCTION
```

```
FUNCTION main()
        DECLARE available_series AS VECTOR[STRING]
        DECLARE records AS VECTOR[Person_Record]

        CALL series_file_read(available_series)
        CALL slot_file_read(records)

        CALL printAvailableSeries(available_series)

        DECLARE schedule_map AS MAP[STRING, STRING]

        CALL find_and_update_clashes(records, schedule_map, available_series)

        CALL total_misses(records)

        RETURN 0
ENDFUNCTION
```

Time Complexity:

slot_file_read() function:

Time Complexity: The time complexity for this function is **O(N * M)**, where N is the number of lines in the input file and M is the maximum number of slots (here, 25). This is because the function iterates through each line of the file and each slot in the time_slots array.

Space Complexity: The space complexity for this function is **O(N)**, where N is the number of Person_Record structs created and stored in the records vector. This is because the function reads data from the input file and creates Person_Record structs to store the data, and the space required for these increases with the number of records.

series_file_read() function:

Time Complexity: The time complexity for this function is **O(N)**, where N is the number of lines in the input file. This is because the function iterates through each line of the file.

Space Complexity: The space complexity for this function is **O(N)**, where N is the number of series read from the file and stored in the available_series vector. This is because the function reads data from the input file and stores it in a vector, and the space required increases as the records

find_and_update_clashes() function:

Time Complexity: The time complexity for this function is **O(N * M)**, where N is the number of records and M is the number of hours in a day **(here, 24)**. This is because the function

iterates through each record and each hour, resulting in a linear time complexity with respect to the number of records and the number of hours.

Space Complexity: The space complexity for this function is **O(N)**, where N is the number of clashes stored temporarily in the list of clashes. This is because the function creates a list to store clashes, and the space required increases with a number.

total_misses() function:

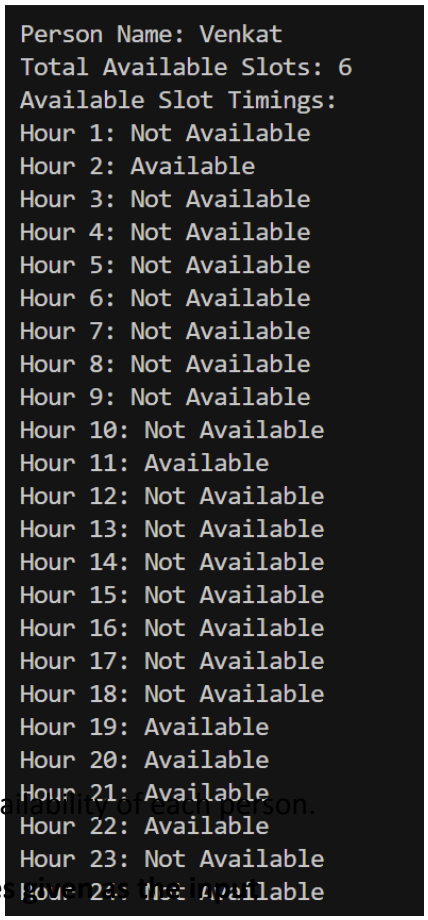
Time Complexity: The time complexity for this function is **O(N)**, where N is the number of records. This is because the function iterates through each record, resulting in a linear time complexity with respect to the number of records.

Space Complexity: The space complexity for this function is **O(1)**, as it only uses a constant amount of space.

<u>Output images:</u>

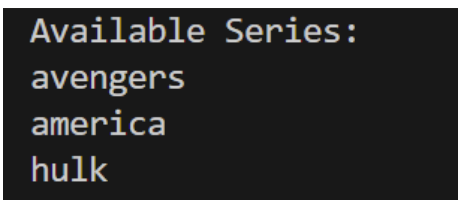
we will get mainly 3 outputs

1. **Output printing the availability slot of each person**



In this way it checks the availability of each person.

2. **checking the total series given as the input**



this checks the series which was given using the series.csv file.

3. The final allocated schedule of all the persons and the misses number

```
Time Slot: 0:00-1:00 - Persons: Suhas (avengers) SriSai (avengers) Viswa (avengers) Sreepadha (avengers)
Time Slot: 1:00-2:00 - Persons: Venkat (car)
Time Slot: 2:00-3:00 - Persons: Sreepadha (farzi)
Time Slot: 3:00-4:00 - Persons: Rishik (depression)
Time Slot: 4:00-5:00 - Persons: Suhas (spiderman)
Time Slot: 5:00-6:00 - Persons: SriSai (fission)
Time Slot: 6:00-7:00 - Persons: SriSai (fusion) Viswa (fusion)
Time Slot: 7:00-8:00 - Persons: Rishik (breakingbad)
Time Slot: 8:00-9:00 - Persons: Viswa (fission)
Time Slot: 9:00-10:00 - Persons: Rishik (cherry)
Time Slot: 10:00-11:00 - Persons: Venkat (breakingbad)
Time Slot: 11:00-12:00 - Persons: Viswa (cascade)
Time Slot: 12:00-13:00 - Persons: Rishik (moneyheist)
Time Slot: 13:00-14:00 - Persons: Sreepadha (cascade)
Time Slot: 14:00-15:00 - Persons: Sreepadha (familyman)
15:00-16:00: Empty slot
16:00-17:00: Empty slot
17:00-18:00: Empty slot
Time Slot: 18:00-19:00 - Persons: Venkat (moneyheist)
Time Slot: 19:00-20:00 - Persons: Venkat (bike)
20:00-21:00: Empty slot
21:00-22:00: Empty slot
22:00-23:00: Empty slot
23:00-0:00: Empty slot
Total misses: 2
```

LOGIC 3:

Input files:

Input.csv: All the data of time slots is converted from excel sheet to csv file and input is taken.

Series_input.csv: All the data of family members is converted from excel sheet to csv file and input is taken.

Data structure:

Unordered map: It is used to store the members available for the particular time slot.

Unordered set: It is used to store series.

Working:

Iterate through each member and if member is available for time slot then chain him to that particular time slot. Repeat the process for all time slots.

To find common series, take series of a member in unordered map and check the unordered set to find number of common people watching it.

Iterate through all the series and all the members and find the common series with maximum value.

For more than 1 person, find and allocate the common series watched by members for that time slot.

Case 1: All people have a common series

Allot the series and all members to that slot.

Case 2: Some people have a common series

We allot the common series randomly to members.

Case 3: No common series for the members

We prioritize all the members and the person with least number of slots is allotted who isn't notified for at least p slots.

Each time slot has only one type of series and the members watching it are removed after watching the series. If there is no series for a member in the unordered set then don't allocate him to remaining time slots.

To calculate the number of misses, calculate the number of series left for each member and are printed

Cases where the algorithm works:

The algorithm works efficiently for multiple-membered time slots reducing the number of misses, and increasing the number of allocations for the members.

Cases where the algorithm doesn't work:

The algorithm works well for scheduling series up to a day. To make it more efficient we need to prioritize the people who have missed the series on the previous day so as to ensure that every member is notified to the maximum extent at the end of the week to increase the efficiency of the working of the algorithm.

Pseudocode:

```
// Initialize data structures

unordered_map<TimeSlot, unordered_set<Member>> timeSlotMembersMap

unordered_map<Member, unordered_set<Series>> memberSeriesMap

unordered_set<Series> allSeries


// Load data from CSV files

LoadTimeSlotsFromCSV("Input.csv", timeSlotMembersMap)

LoadSeriesFromCSV("Series_input.csv", memberSeriesMap, allSeries)


// Function to find common series among members

function FindCommonSeries():
```

```
    maxCommonSeriesCount = 0
        maxCommonSeries = empty
        for each series in allSeries:
      commonSeriesCount = 0
        for each member in memberSeriesMap:
        if series in memberSeriesMap[member]:
          commonSeriesCount++
        if commonSeriesCount > maxCommonSeriesCount:
        maxCommonSeriesCount = commonSeriesCount
        maxCommonSeries = series
        return maxCommonSeries

// Function to allocate series to time slots
function AllocateSeriesToTimeSlots():
        for each timeSlot in timeSlotMembersMap:
        commonSeries = FindCommonSeries()
        if commonSeries is empty:
        // Case 3: No common series for the members
        prioritizedMembers = prioritizeMembers(timeSlotMembersMap[timeSlot])
        for each member in prioritizedMembers:
        if member not notified for at least p slots:
            allocateSeriesToMember(member, timeSlot, randomSeries)
        else:
        // Case 1: All people have a common series
        if all members in timeSlotMembersMap[timeSlot] have commonSeries:
```

```
            allocateSeriesToAllMembers(commonSeries, timeSlot)
        // Case 2: Some people have a common series
        else:
        for each member in timeSlotMembersMap[timeSlot]:
                if member has commonSeries:
                allocateSeriesToMember(member, timeSlot, commonSeries)

// Function to prioritize members
function prioritizeMembers(members):
        // Implement prioritization logic here
        return prioritizedMembers

// Function to allocate series to a member for a time slot
function allocateSeriesToMember(member, timeSlot, series):
        // Allocate series to the member for the time slot
        remove series from memberSeriesMap[member]
        remove member from timeSlotMembersMap[timeSlot]

// Function to allocate series to all members for a time slot
function allocateSeriesToAllMembers(series, timeSlot):
        for each member in timeSlotMembersMap[timeSlot]:
    allocateSeriesToMember(member, timeSlot, series)

// Function to calculate number of misses
function CalculateMisses():
```

```
        for each member in memberSeriesMap:
        print number of series left for member


// Main function
function main():
   AllocateSeriesToTimeSlots()
        CalculateMisses()


// Run the main function
main()
```

Space complexity:

1) Data structures:

(A) unordered map<TimeSlot, unordered set<Member>> timeSlotMembersMap: The map stores time slots(n) and members(m) in it. So space complexity is **O(m*n).**

(B) unordered map<Member, unordered set<Series>> memberSeriesMap: The members stores members(m) and set of series(s) on average for each member. So space complexity is **O(m*s).**

(C) unordered set<Series> allSeries: The set stores all unique series(s). So space complexity is **O(s).**

(D)Overall space complexity is the sum of space complexitiethe s of above complexities. So space complexity is **O(m*n+m*s+s).**

Time Complexity:

1) LoadTimeSlotsFromCSV(“Input.csv”, timeSlotMembersMap): It iterates over members(k) of the file. So time complexity is **O(k).**

2) LoadSeriesFromCSV(“Series_input.csv”, memberSeriesMap, allSeries): It just iterates over series(k) of the file. So time complexity is **O(k)**.

3) AllocateSeriesToTimeSlots(): It iterates over n time slots which have m members in each time slot. So time complexity is **O(m*n).**

4) CalculateMisses(): The function iterates over each member(m) to find count of each member. So time complexity is **O(m).**

5) Overall time complexity is sum of complexities of LoadTimeSlotsFromCSV("Input.csv", timeSlotMembersMap), LoadSeriesFromCSV("Series_input.csv", memberSeriesMap, allSeries) and AllocateSeriesToTimeSlots(). So time complexity is O(m*n+k+k)= **O(m*n+k).**

Here all recursive function calls and other data types are negligible and are considered as constants. Also, all the members in each slots and series are taken on average. So they don't disturb the complexities.

Output images:

```
 ENTER THE VALUE OF P: 2
FINAL SCHEDULE:
   SLOT 1  0:00-1:00 - Suhas (avengers) SriSai (avengers) Viswa (avengers) Sreepadha (avengers)
   SLOT 2  1:00-2:00 - Suhas (hulk)
   SLOT 3  2:00-3:00 - NO ONE HAS BEEN ALLOCATED TO THIS SLOT
   SLOT 4  3:00-4:00 - Rishik (depression)
   SLOT 5  4:00-5:00 - Suhas (spiderman)
   SLOT 6  5:00-6:00 - NO ONE HAS BEEN ALLOCATED TO THIS SLOT
   SLOT 7  6:00-7:00 - SriSai (fission) Viswa (fission)
   SLOT 8  7:00-8:00 - Rishik (breakingbad)
   SLOT 9  8:00-9:00 - Suhas (america)
   SLOT 10  9:00-10:00 - Rishik (cherry)
   SLOT 11 10:00-11:00 - NO ONE HAS BEEN ALLOCATED TO THIS SLOT
   SLOT 12 11:00-12:00 - SriSai (fusion) Viswa (fusion)
   SLOT 13 12:00-13:00 - Rishik (moneyheist)
   SLOT 14 13:00-14:00 - Viswa (cascade) Sreepadha (cascade)
   SLOT 15 14:00-15:00 - Sreepadha (farzi)
   SLOT 16 15:00-16:00 - Sreepadha (familyman)
   SLOT 17 16:00-17:00 - NO ONE HAS BEEN ALLOCATED TO THIS SLOT
   SLOT 18 17:00-18:00 - NO ONE HAS BEEN ALLOCATED TO THIS SLOT
   SLOT 19 18:00-19:00 - Venkat (moneyheist)
   SLOT 20 19:00-20:00 - Venkat (breakingbad)
   SLOT 21 20:00-21:00 - Venkat (bike)
   SLOT 22 21:00-22:00 - Venkat (car)
   SLOT 23 22:00-23:00 - NO ONE HAS BEEN ALLOCATED TO THIS SLOT
   SLOT 24 23:00-24:00 - NO ONE HAS BEEN ALLOCATED TO THIS SLOT

MISSED SERIES:

 THE TOTAL NUMBER OF SERIES MISSED: 0
```

This is the output image of the logic 3.

The outputs of logic 2 and logic 3 are varied because logic 3 uses the “p” value and the allocation of the series for logic 2 and 3 is different when there is only a single person for the slot. The major difference between them is that in logic 3 we are prioritising the schedule and using the P value to reduce the clashes and also to reduce the misses. In algorithm 3 we tried to reduce the number of clashes and the number of misses.

github link: https://github.com/ViswaVignan/Code_Blasters/tree/main