GA1 – Report

Group members: Siya Sonpatki, Jamie Liu, Shraddha Hegde

PART (A)

Description of Algorithm

Our first algorithm (time complexity: n log C) depends on the binary search algorithm and the greedy algorithm approach. The outer loop uses binary search by comparing and calculating the midpoints of the range between the largest group size (of n) and the total amount of participants.

The firstpart of the algorithm involves Binary Search by generating the midpoints to compare the club points to (number of members added up). The while loop works with two variables:

- Low – the largest group in the list of clubs
    - Works as the bottom bound for the binary search range
- High - sum of all the members registered (C)

The while loop is run on the condition that the low (largest group) is less than the high, and will constantly check as the numbers between low and high get slimmer as C (or high) is divided in 2. The mid point between low and high is calculated by adding the two then dividing by 2. The function from the first part is then used to checks if the days need another pass (seeing if the mid point is too high). If true, we set high to the midpoint –1, and if false, we set high to the midpoint + 1 to further check whether the true min number lies between (low, >midpoint) or the midpoint up to C. From there, once all the days are allocated we check the final value of low, or a modified midpoint from C, and return that day.

The second loop starts by keeping track of the following variables:

- Days_used
    - Keeps track of the # of times clubs can be split (via days) on the current pass
- Current_day_total
    - Current number of added members per day

We first loop through all the clubs within the specified array, and check if the number of added members is less than or equal to the max_capacity (which is the total amount of people registered or C). If so, we add the clubs to the current_day_total. Otherwise, we move on to the next day, check if the amount of passed days is less than D, and return

false/true otherwise. This is the greedy implementation, as we're immediately moving onto the next day and sequentially allocate the number of days to place each club.

Running Time Analysis

The runtime of our algorithm can be calculated by multiplying the runtimes of its two components: the outer loop, which implements the binary search, and the inner loop, which implements the greedy algorithm.

In the outer loop, we perform a binary search over the range from low the size of the largest club) to high (the total number of members across all clubs). In every iteration of the loop, we check if it is possible to schedule the clubs with mid as the capacity, and depending on the result, we either:

- Reduce the upper bound to mid – 1
  or
- Increase the lower bound to mid + 1.

This effectively halves the search space in every iteration, making it logarithmic. If the range starts at C, then it reduces to C/2, and then C/4, and continues until we are left with one element to search. Thus, the time complexity for the outer loop is O(logC).

For the inner loop, we check which clubs can be scheduled without exceeding the maximum capacity. For this, we need to go through every club in the list, and because there are n clubs, the time complexity for this will be O(n).

Since we need to run the greedy algorithm for every iteration of the binary search, we multiply their respective time complexities to calculate the overall time complexity of the algorithm. Thus, the time complexity for this algorithm is O(n logC).

Proof of Correctness

This algorithm uses Binary Search with a time complexity of log n, then multiplied by a constant of n as the algorithm searches through the list of groups. The first function is_schedule_possible first checks whether all the clubs can be helped in D or less days by adding the attendees to a current day total. It tracks how many days are used, and whether its less than or equal to D, (returning true or false), ensuring the condition that capacity can be done in D days.

This part ensures that order is maintained, no clubs are split, and the division can be done in D days, satisfying these conditions in the description.

The second part ensures the answers are correct with Binary search. The initial bounds ensure the largest club is included no matter what, and sets the bounds to decrease by the midpoint until the lowest minimum is achieved. Additionally, the inclusion of high (C, or sum of all the clubs) accounts for the edge case of 1 day. The true capacity will eventually be found because of its inclusion in the initial search range. Once the binary search completes, the value of low is the minimum capacity for all the clubs.

This ensures the final answer is correct, and accounts for the edge cases of minimum days, or clubs.

PART (B)

Description of Algorithms

Given the following, which is read from the first and second lines of a text file respectively:

- An integer *D*, representing the number of days that the program must sort the clubs into
- An array *clubs[]* of size *n*, where *n* is the total number of clubs, and each element of the array contains the specific number of members in each club

Our polynomial time algorithm for this problem consists of the following steps:

1. **Initialize an empty array *max_sum[]***: this array will record the highest sum during each iteration.
2. **Initialize *D* empty arrays, name them *Day0[],…,Day D[]*:** each array will contain a different combination of elements from clubs[] as determined by the different iterations.
3. **Set up the initial division (Iteration One):** In the first iteration of this program, assign all elements in *clubs[]* to *Day D[]*, representing the last day of the conference, while leaving *Day0[],…,Day D-1[]* arrays empty.
4. **Iterations**: For each iteration of this algorithm, pop the first element off the 'current' array and push it to the back of the 'previous' array. Start at the array representing the last day of the conference (*Day D[]* is the 'current' array in this iteration) - pop the front element off of *Day D[]* and push it to the back of *Day D- []*, where *Day D-1[]* is the 'previous' array.
    a. Set the current array to the previous array, and repeat this step for all arrays *Day D-1[] through Day0[]*
    b. Iterations repeat until *Day0[]* contains all the elements in *clubs[]*
        i. Total iterations = *n*

5. **Sum**: For each iteration, calculate the sum of the elements in each *Day* array. Record the maximum of these sums in the *max_sum[]* array.

6. **Empty arrays**: Arrays *Day0[]* through *Day D[]* must be emptied before next iteration.
7. **Return a final result:** Once all iterations have been completed (n total iterations), find the minimum value in the *max_sum[]* array, which represents the minimum number of attendees that the library must accommodate each day.

Running Time Analysis

Deconstructing the Algorithm

- **Initializing *max_sum[]* : O(1)**
    - Initializing an empty array requires a constant time
- **Initializing *D* empty arrays (*Day0[],...,DayD[]*): O(D)**
    - Initializing *D* empty arrays is a constant time because *D* is given as a constant before the algorithm begins
- **Initial Division: O(n)**
    - All elements from *club[]* are placed into *Day D[]* while the remaining Day arrays are empty
    - Assigning *n* elements to an array is a linear time
- **Iterations: O(n x D)**
    - Since there are *D* arrays, moving an element across each array takes O(D) per iteration, and since each array can have, at most, *n* elements, the total complexity for iterations is O(n x D)
- **Sum Calculations for Each Iteration: O(n x D)**
    - Since there are *D* arrays and each array can sum, at most, *n* elements, the total complexity for the sum calculations is O(n x D)
- **Storing Maximum Sums: O(n)**
    - During each iteration, the max of the *Day* array sums is stored in *max_sum[]*
    - Storing a single value in an array is a constant time and since there are *n* iterations, the running time is O(n)
- **Finding Minimum Value of max_sum[]: O(n)**
    - Since *max_sums[]* contains *n* elements, finding the minimum value requires searching through the array elements, thus the running time is O(n)

Total Running Time Complexity

- Initializing *max_sum[]* : O(1)
- Initializing *D* empty arrays (*Day0[],…,Day D[]*): O(D)
- Initial Division: O(n)
- Iterations: O(n x D)
- Sum Calculations for Each Iteration: O(n x D)
- Storing Maximum Sums: O(n)
- Finding Minimum Value of max_sum[]: O(n)

Final RT Complexity: **O(n x D)**, *n = elements in clubs[] & D = # of days*

- Combination of terms: O(n x D) + O(n) + O(D)
- Dominant term is O(n x D), thus it represents the final RT complexity

Proof of Correctness

To show that this algorithm works, we will show how each of the criteria is met.

1. **All clubs get to attend the event within the D days-** this holds because we place each of the clubs in one of the day arrays to be sorted and never remove an element without placing it elsewhere.
2. **Every club attends together, without being split across different days-** since we are working with the clubs and not the members in the club in this algorithm, there is no way to split a single club, hence this is requirement also holds.
3. **The order of registration is respected: any club that registered earlier must attend before or on the same day as any later club-** while transferring the elements into a new array, we make sure that we only move the first element in every array to the bottom of the previous array. This means that a club that registered first gets priority and will get moved up to an array representing an earlier or the same day than an array containing a club that registered after.
4. **Determine the minimum number of attendees they must accommodate each day- our algorithm satisfies this condition because we:**
    - Explore the possible combinations that clubs can go to while maintaining the order in which they register
    - Calculate the total number of members attending every day in all those combinations and find the largest number of people attending.

- Compare these totals and then pick the minimum number of people who could be attending during a single conference day, hence finding the minimum number of attendees the library must accommodate each day.