Task 1: `Tourist`

Authored by: Ling Yan Hao

Prepared by: Ling Yan Hao

Editorial written by: Ling Yan Hao

For each day, we either spend y dollars to buy a day pass, or buy $a[i]$ single trip tickets and spend $a[i] \cdot x$ dollars. We clearly take the cheaper option and sum them up across all N days.

Task 2: `Party`

Authored by: Yu Zheyuan

Prepared by: Yu Zheyuan

Editorial written by: Yu Zheyuan

Subtask 1

Additional constraints: $n \leq 3$

We will solve this task for each value of n

When $n = 1$, we can always invite the friend if he gains positive happiness

When $n = 2$, we realize we can only seat one friend so out of the two friends we choose the one that gains more happiness. If both friends gain negative happiness then it is better not to invite anyone.

When $n = 3$, we now can seat two friends so we choose the two with the maximum happiness gained. Take note if either of them gain negative happiness we can again not invite that friend.

Subtask 2

Additional constraints: $n \leq 1000$

We realize we can't just case-work each n, so we try to come up with a general solution.

We try to make some observations about how many friends we can invite.

When n is odd, we can seat the friends at all the odd seats, meaning seats $1, 3, 5 \cdots n$ from the left. This is optimal as there is no way to fit more friends. Thus we can invite $(n + 1)/2$ friends when n is odd.

When n is even, we can seat the friends again at all the odd seats, meaning seats $1, 3, 5 \cdots n - 1$. Thus we can seat $n/2$ friends when n is even.

Now we realize it is always to optimal as many friends as we can **as long as their happiness is positive**. Hence, we sort the friends by their happiness and start inviting the people with the highest happiness until either we run out of seats or the person's happiness becomes negative.

This subtask is given to allow for any sorting algorithm.

Subtask 3

We utilize the same algorithm as Subtask 2 except with $O(n \log n)$ sorting, allowing us to pass the full problem.

Task 3: `School Photo`

Authored by: Yu Zheyuan

Prepared by: Sng James

Editorial written by: Sng James and Yu Zheyuan

Subtask 1

Additional constraints: $n = 2$

Let $h[i][j]$ denote the height of the j-th student in the i-th class.

We have to select exactly 1 student from each of the 2 classes. We can do so by trying to pair each student from class 0 with each student from class 1, and finding the minimum possible height difference.

Time complexity: O(s^2).

Subtask 2

Additional constraints: $n, s \leq 100$

Let H be the set of distinct heights among all students. This set has at most $n \times s$ elements.

We can try each $x \in H$ as the minimum height. Then, we iterate through each class i to find the shortest student with height at least x. We take the maximum height difference among all such students to obtain the minimum possible height difference where x is the minimum height we can select. This takes O$(n \times s)$ for each x.

Taking the minimum such value across all $x \in H$, we will obtain the minimum possible answer.

Time complexity: O$(n^2 s^2)$

Subtask 3

Additional constraints: $n, s \leq 250$

Instead of iterating through the entire class to find the shortest student with height $h[i][j] \geq x$, we can sort the heights of students for each class first, taking $O(n \times s \times \log(s))$ time in total.

Then, for each x, we can binary search each class to find the desired student, taking $O(n \times \log(s))$ time for each x.

Time complexity: $O(n^2 s \log(s))$

Subtask 4

Additional constraints: $n, s \leq 500$

Suppose we process the $x \in H$ in increasing order, and the heights of students in each class are also in increasing order. Then, notice that the minimum $h_{i,j} \geq x$ for each class will have monotonically increasing positions.

Thus, instead of performing a binary search to find the minimum $h[i][j] \geq x$ in each class for each x, we can store one pointer for each class to that student instead. Then, when processing a new x, we can simply iterate through each class and increment the pointers as necessary.

Time complexity: $O(n^2 s)$

Subtask 5

Additional constraints: None

Instead of iterating through each class to find the pointers to increment, we can store a set of pairs $(h[i][j], i)$, representing the elements the pointers point to. Then, when processing a new x, we remove all elements with $h[i][j] < x$, and add $(h[i][j+1], i)$ to the set.

We can then compute the answer for the current x as (maximum h in the set - x).

Time complexity: $O(ns(\log(s) + \log(n)))$

Alternative Solution 1

Notice that the elements we remove when processing x are those that have $h[i][j]$ equal to the previous x we processed. Thus, we can maintain an array of vectors for each x value instead of the set. If we do that and perform radix sort instead of library sort, we can remove all the extra log factors.

Time complexity: O(ns)

Alternative Solution 2

We binary search for the answer directly. Suppose we want to determine whether there is a choice for each class such that the tallest student is at most x taller than the shortest student.

We maintain a sorted set of pairs $(h[i][j], i)$. We do a sliding window on this set where the max height and minimum height differ by at most x. We observe that we only need to maintain which classes have appeared in this range. If all classes appear then this x is possible.

To maintain this info, we keep a counter for each class denoting how many elements are in the range cnt_i, along with how many distinct classes have appeared at least once $classes$.

When we add an element (h, i) to the range, we first need to check if $cnt_i = 0$, if so we add 1 to $classes$. Then, we increment cnt_i by 1.

When we delete an element (h, i) from the range, we first need to check if $cnt_i = 1$, if so we subtract 1 from $classes$. Then, we decrement cnt_i by 1.

If $classes = n$ at any point in time, this x is possible.

Time complexity: O($ns \log ns$)

Task 4: Amusement Park

Authored by: Stuart Lim

Prepared by: Stuart Lim

Editorial written by: Stuart Lim

Subtask 1

Additional constraints: $q \leq 1000$

This subtask is intended to check contestants' understanding of the boarding process. Since at most q groups can join the queue, the boarding algorithm in the problem statement runs in $O(q)$ time. Using it each time gives an overall time complexity of $O(q^2)$ which is fast enough.

Store the information in arrays indexed by group ID. Groups with smaller ID are always in front, so we can traverse the arrays according to increasing ID. A group that has completely left the queue can be represented with a size of 0 and we can ignore them during a *board* operation.

Alternatively, we can maintain the queue more directly using a data structure that supports insertions at the back and deletions anywhere. Suitable data structures with different time complexities per operation include the linked list, STL set and STL vector. Despite its name, the STL queue cannot be used as it does not support deletions and access in the middle.

Time complexity: At most $O(q)$ per operation, $O(q^2)$ overall

Subtask 2

Additional constraints: $s = 1$, $w = 0$, there are no *leave* operations

Each group has one person and people will only leave from the front of the queue when boarding the attraction. Hence, an STL queue can be used.

During a *board* operation, the groups that board are exactly the first b groups in the queue (or everyone, if there are fewer than b people in the queue). It is not necessary to consider more groups beyond the first b in the queue. Since there are at most q groups and each group is processed by at most one *board* operation, all *board* operations have a combined time complexity of $O(q)$.

Additionally, we only need to output the groups that board, so the time complexity of the output

is also $O(q)$.

Time complexity: $O(1)$ per *join* operation, $O(q)$ for all *board* operations, $O(q)$ overall

Subtask 3

Additional constraints: $s \leq 10$, $w = 0$, there are no *leave* operations

In this subtask, group sizes can vary. A group in front is not guaranteed to board first as it may be too big to board. However, it will still board earlier than other groups with the same size.

We can maintain a sub-queue using an STL queue for each possible group size. To decide which group should board, consider the groups in the front of each of the first $\min(b, s_{max})$ sub-queues. Among these groups, pick the one with the smallest ID as it is closest to the front. Reduce the number of available seats and repeat until there are no more available seats or until the first $\min(b, s_{max})$ sub-queues are empty. This loop runs $k + 1$ times, where k is the number of groups that board. The sum of all k is at most q, so the loop runs at most $2q$ times over all *board* operations.

Time complexity: $O(1)$ per *join* operation, $O(qs_{max})$ for all *board* operations, $O(qs_{max})$ overall

Subtask 4

Additional constraints: $s \leq 10$, there are no *leave* operations

Groups can split in this subtask. We cannot put groups that allow splitting and groups that do not allow splitting in the same sub-queue. On the other hand, notice that for any two groups that allow splitting, the group in front is always processed earlier. This means that we can have a separate sub-queue for all groups that allow splitting. We amend the solution to Subtask 3 by also checking this new sub-queue in the loop.

A group will only split when its size is larger than the number of seats remaining. However, after splitting, there will be no seats remaining. Hence, out of k groups where at least one person boards during a *board* operation, only the k-th group may split, whereas the previous $k - 1$ groups do not split and leave the queue completely. As such, the sum of k is at most $2q$ and the loop runs at most $3q$ times over all *board* operations.

Time complexity: $O(1)$ per *join* operation, $O(qs_{max})$ for all *board* operations, $O(qs_{max})$ overall

Subtask 5

Additional constraints: $s \leq 10$

To allow deletions anywhere in a sub-queue, we can implement sub-queues as linked lists or STL sets.

Alternatively, we can keep using the STL queues with a lazy deletion strategy: instead of deleting groups immediately when they leave, just do it later when we absolutely have to. To remember which groups to delete, we will maintain a Boolean array del indexed by group ID, with all values initialised to *false*. During a *leave* operation, set $del[i]$ to *true*, but do not edit the sub-queues yet. Inside the loop in our *board* operation, when a group i' is chosen, check $del[i']$. If it is *true*, delete group i' from the sub-queue but do not allow anyone to board.

Time complexity: $O(1)$ or $O(\log q)$ per *join* or *leave* operation, $O(qs_{max})$ for all *board* operations, $O(qs_{max})$ or $O(q(\log q + s_{max}))$ overall

Subtask 6

Additional constraints: None

Since the limit on s is increased, we need to speed up *board* operations. Recall that in the loop, we take the minimum group ID among the front of the sub-queue allowing splits and the fronts of the first $\min(b, s_{max})$ sub-queues forbidding splits.

We build a segment tree on the fronts of all sub-queues. Finding the minimum group ID becomes a range minimum query. Whenever the front of a sub-queue changes, we perform a point set update. Both segment tree operations run in $O(\log s_{max})$ time each.

Time complexity: $O(1)$ or $O(\log q)$ per *join* or *leave* operation, $O(q \log s_{max})$ for all *board* operations, $O(q \log s_{max})$ or $O(q(\log q + \log s_{max}))$ overall

Task 5: `Explosives`

Authored by: Ling Yan Hao

Prepared by: Ling Yan Hao

Editorial written by: Ling Yan Hao

Subtask 1

Additional constraints: $c = 1$

Observe that when $c = 1$, we are forced to alternate between pickup and offload operations. Whenever we transport explosives from $a[i]$ to $b[j]$, we pay a cost of $|a[i] - b[j]|$. Our objective simply reduces to the following:

- Pair up the factories and mines in a way such that the sum of distances among all pairs are minimized.

If we pair up the leftmost factory with the leftmost mine, second leftmost factory with the second leftmost mine, etc., this minimizes the total sum of distances. This can be proven using a standard exchange argument.

Subtask 2

Additional constraints: $a[i] \leq 5000$**,** $b[j]] > 5000$

We can execute the following algorithm:

- Let $c' = \min(c, n)$
- Pickup explosives from the leftmost c' factories
- Offload them to the rightmost c' mines
- Decrement n by c'

We can do this until n becomes 0, in which we are done.

As for why this greedy strategy is optimal, refer to next subtask.

Subtask 3

Additional constraints: None

We will change the notation to make this solution easier to describe. Let the $2n$ locations ($a[1..n]$ and $b[1..n]$) be $x[1..2n]$ instead, in increasing order. For each of these locations, let $s[i] = 1$ for factories and $s[i] = -1$ for mines. Let $p[i] = s[1] + \ldots + s[i]$ be the prefix sum of $s[i]$. We break down the straight road into segments, where the i-th segment is the portion of the road joining $x[i]$ and $x[i+1]$. We say that the truck is heading east if it moves from a position with a smaller number to a higher number, and west otherwise.

We can make the following observation:

- For all i, there must be a net movement of $p[i]$ explosives across the i-th road segment. Therefore, the truck must drive at least $\lceil |p[i]|/c \rceil$ times across this segment.

From here, we obtain that the minimum cost should be at least

$$\sum_{i=1}^{2n-1} \lceil |p[i]|/c \rceil (x[i+1] - x[i])$$

It turns out that this cost is indeed attainable. It remains to show how to construct a sequence of operations attaining this cost.

The formula suggests that:

- If $p[i] > 0$, we should try to carry $\min(c, p[i])$ units of explosives across when we move east across the i-th segment
- If $p[i] = 0$, we should not carry any explosives across the i-th segment
- If $p[i] < 0$, we should try to carry $\min(c, -p[i])$ units of explosives across when we move west across the i-th segment.

Let us suppose for now that $p[i] \geq 0$ for all i. A possible approach is to do the following:

- Drive the truck from $x[1]$ to $x[2n]$. Ensure that we carry exactly $\min(c, p[i])$ units of explosive across the i-th segment all the time.
- Every time we execute such a run, $p[i]$ will be either reduced to 0, or reduced by c.
- Repeat the following until all $p[i] = 0$, in which we will be done since all $s[i]$ must also be 0.

We now return to the original problem where we are no longer guaranteed that $p[i] < 0$. We run a modified version of the above algorithm, driving the truck from $x[1]$ to $x[2n]$, except that we carry $\max(0, \min(c, p[i]))$ units of explosives across the i-th segment instead.

If $p[i] > c$, $p[i]$ will reduce by c. If $0 < p[i] \leq c$, $p[i]$ will be reduced to 0. If $p[i] \leq 0$, then $p[i]$ remains unchanged. By doing so, we can reduce all positive values of $p[i]$ to 0, so that we have $p[i] \leq 0$ for all i.

Once this is done, the algorithm can be repeated in reverse to ensure that $p[i] = 0$ for all i.