Task 1: Pilot

Pang Wen Yuen

Abridged Problem Statement

Given an array of N integers, answer Q queries of the form: how many tuples (s, e) exist such that $s \leq e$ and $\max(H_s...H_e) \leq Y_i$?

Subtask 1

For this subtask, $N = 2$, $Q = 1$. Therefore, we only need to check if H_1 and H_2 is greater than or equal to Y_1. If neither are, then the answer is 0. If either one satisifies this condition, then the answer is 1. Otherwise, the answer is 3.

Subtask 2

In this subtask, $1 \leq N, Q \leq 30$. For each query, we can loop through the N^2 possible tuples, and for each tuple, we can use an $\mathrm{O}(N)$ loop to determine the maximum for that particular subarray, and add 1 to an overall counter if it is lower than Y_i. The overall complexity is $\mathrm{O}(QN^3)$.

Subtask 3

In this subtask, $1 \leq N, Q \leq 200$. Notice that instead of recomputing the maximum for each tuple, we can first fix a starting point s, then loop the ending point from s to $N - 1$, keeping a running maximum along the way. We can therefore compute the maximum of each tuple in $\mathrm{O}(1)$ time. The overall complexity of this solution is $\mathrm{O}(QN^2)$.

Subtask 4

In this subtask, $1 \leq N, Q \leq 10^3$. Notice how we can precompute the maximum for each subarray in $\mathrm{O}(N^2)$ time, total. Sort this list of maximums and for each query, we can use binary search to determine the number of subarrays where the maximum is less than Y_i. The overall complexity of this solution is $\mathrm{O}(N^2 \log N + Q \log N)$.

Subtask 5

In this subtask, $Y_i = 10^6$, implying that every flight is possible. The answer to the only query would therefore be $\frac{N(N+1)}{2}$.

Subtask 6

In this subtask, $H_i = i$. This means that for a particular query Y_i, all flights where $1 \leq s \leq e \leq Y_i$ would be possible. Therefore the answer to the query would be $\frac{Y_i(Y_i+1)}{2}$.

Subtask 7

In this subtask, H is strictly increasing. We can extend the solution from Subtask 6 to achieve this. We first do a binary search on H to find the maximum index x where $H_x \leq Y_i$. Then the answer would be $\frac{x(x+1)}{2}$.

Subtask 8

In this subtask, $Q = 1$. Consider an array A where $A_i = 0$ if $Q_1 \leq H_i$, and $A_i = 0$ otherwise. All positions where $A_i = 1$ are positions that are accessible to the plane. As such, the answer would be the sum of $\frac{L(L+1)}{2}$ where L is the length of each segment of 1s.

Subtask 9

The solution to Subtask 9 involves processing of queries in decreasing order of Y_i. We first consider a plane where $Y = 10^7$, meaning the total number of flights it can take is $\frac{N(N+1)}{2}$. We keep decreasing Y through each value of Y_i, maintaining the total number of flights that can be taken by a plane where $Y_i = Y$.

When Y passes through a value of A_i, a segment of 1s (as described in Subtask 8) is broken into two. Therefore, the current answer can be maintained by subtracting $\frac{L(L+1)}{2}$ where L is the length of the original segment and adding back the value for the two split segments. As this can be done with the aid of a STL set, the time complexity is $O(N \log N)$.

Subtask 10

The final solution extends from Subtask 8. We can process all queries simultaneously using a stack. Consider processing the array H from left to right.

We can maintain a descending hull using the stack, where the top element of the stack contains the value and position of the closest element to the left greater than the current element being processed, the next element is the closest element to the left of it greater than it, and so on.

When a new element is processed, we pop out all elements in the stack smaller than it, and modify the answers to all queries as follows after each pop, letting the popped element be of value x: add $\frac{D(D+1)}{2}$ (D is the distance between the new element and the popped element) to the answer of queries where $Y_i < h$ and Y_i is greater than the value of the previously popped element.

This reduces the problem to a set of range add updates. This can be solved in $O(N + H_i)$.

Task 2: Lasers

Boo Kai Hsien

Abridged Problem Statement

There is a grid of R rows and L columns. Each row has a positive number of walls that can be slid along the same row (each with a positive integer width), without crossing over one another. Find the number of columns cannot be empty in all arrangement of the walls.

Subtask 1

In this subtask, $R = 1$, $X = 1$. We consider the walls that are blocked in all arrangements instead. Notice that if the length of the wall segment is w, then the segment $(L - w,\ w)$ will always be blocked (provided the segment is non-empty). Therefore, the answer would be $\max(0, 2w - L)$.

Subtask 2

To solve the problem without the $R = 1$ limit, notice that only the longest wall matter, becasue every other wall can "hide" behind the longer wall. Hence, you can find the maximum length by using a running max algorithm, and this reduces to subtask 1.

Subtask 3

The next two subtasks utilise a different approach to the problem. To check if a column is free in at least one arrangement, we process each row, and flush as many walls toward the left edge of the row as possible, until it is not possible to do so without blocking the current column, at which point we flush the remaining walls to the right. If we do this for both rows, we are able to determine if a column is free in at least one arrangement.

Instead of manually flushing each wall to the left and right, we can create a separate array for each row containing the prefix sum of the lengths of the walls. A simple binary search would allow us to find the total length of walls flushed to the left and right. Therefore, the overall solution complexity is $O(LR \log X)$.

Subtask 4

The solution outlined in Subtask 3 can be trivially extended to solve Subtask 4 by looping through all rows instead of just two.

Subtask 5

In this subtask, $1 \leq L \leq 10^6$. We can extend the solution from Subtask 4 to solve this. We first consider the arrangement where all the walls are flushed toward the left, i.e. when the column to be cleared is the Lth column. As the column to be cleared moves to the left, we shift the walls ending at the "selected column" to the right, keeping track of the maximum length of the walls flushed to the right with a running maximum (since those lengths always increase as walls are flushed toward the right), in order to check if it is viable to clear each row. This runs in $O(L + \sum X)$.

Subtask 6

Notice that for each row, we can generalise the set of freeable columns to the union of $(l_1 + \cdots + l_{i-1}, L - l_{i+1} - \cdots - l_X) - (L - l_i - \cdots - l_X, l_1 + \cdots + l_i)$ where $1 \leq i \leq X$.

To extend this solution to multiple rows, we simply consider the inverse of those ranges in each row (the set of columns that are blocked), and take the union of that across all rows. This solution runs in $O(\sum X \log \sum X)$.

Task 3: Feast

Zhang Guangxuan

Abridged Problem Statement

Given an array of N integers, select K non-overlapping subarrays such that their sum is maximised.

Subtask 1

In this subtask, $A_i > 0$. Therefore, it is always optimal to pick the entire array. Computing this takes $O(N)$.

Subtask 2

In this subtask, there will at most be one position where $A_i < 0$. Let the sum to the left of that position be S_l, and the right be S_r. The answer will be the maximum of S_l, S_r, and the sum of the entire array. This solution runs in $O(N)$.

Subtask 3

In this subtask, $K = 1$. This reduces to the classic maxsum problem, which can be solved with Kadane's algorithm in $O(N)$.

Subtask 4

In this subtask, $1 \leq N, K \leq 80$. This subtask requires a dynamic programming solution. Let $f(n, k)$ be the maximum sum of k subarrays within the first n integers. To compute $f(n, k)$, we take the maximum of:

- $f(n-1, k)$, indicating the nth integer was not chosen,
- $f(i, k-1) + sum(A_{i+1}, A_n)$ for all $i < n$, indicating that the nth integer belongs to a selected subarray from the $(i+1)$th to nth position.

Naively, the state is $O(NK)$ and the transition is $O(N^2)$. The solution complexity is therefore $O(N^3K)$.

Subtask 5

Optimising the Subtask 4 solution using prefix sum would bring the transition down to $O(N)$, making the overall complexity $O(N^2K)$.

Subtask 6

To solve this subtask, we need to define an auxiliary DP function $g(n, k)$, which is the maximum sum of k subarrays within the first n integers such that the nth integer is chosen. To calculate $g(n, k)$, we take the maximum of:

- $g(n-1, k) + A_n$, which means extending the last subarray to the nth integer,
- $f(n-1, k-1) + A_n$, which means starting a new subarray at the nth position.

To calculate $f(n, k)$, we take the maximum of $f(n-1, k)$ and $g(n, k)$. The number of states in this DP is $O(NK)$ and the transition is $O(1)$, making the overall time complexity $O(NK)$.

Subtask 7

Obtaining the last subtask requires a greedy approach. Firstly, we combine adjacent values with the same sign, and remove negative values from the two ends of the array. This creates an odd-length array where the first and last element are positive, and the sign of each integer alternates between positive and negative.

If there are no more than K positive integers, then the solution would just be picking all of them. If there are more than K integers, we repeat the following process until there are only K positive integers:

- Pick the element with the lowest absolute value.
- Combine this element with the two elements adjacent to it.

Each of these operations can be done in $O(\log N)$ time, by using an STL set or priority queue. This makes the overall complexity of this solution $O(N \log N)$.

Task 4: Rigged Roads

Pang Wen Yuen

Abridged Problem Statement

Given a general graph G and a spanning tree S of edges, assign the lexicographically minimum permutation of weights to the edges of G such that S is the minimum spanning tree of G.

Subtask 1

In this subtask, $1 \leq N, Q \leq 9$. The approach to this subtask is to brute force all possible permutations of W, and check if the MST of G is S, then output the one with that is lexicographically minimal. The MST of the graph can be found with Kruskal's algorithm which runs in $O(E \log E)$. Therefore the overall time complexity is $O(E! \times E \log E)$.

Subtask 2

Subtask 2 can be solved with an unoptimised version of the final solution. Finding the lexicographically minimum permutation of W requires us to minimise W_1, then W_2, followed by W_3, etc.

We process the edges in increasing order of index, and do the following:

- If the edge is already labelled with a weight, ignore it.
- Otherwise, if the edge is in S, we label the edge with the lowest weight that has not been used on any other edge.
- Lastly, if the edge (a, b) is not in S, we consider the path from a to b along the edges in S. Since all of them must be labelled with a weight that is less than (a, b), we can consider the edges along the path that are unlabelled, and label them with the minimum available weights in increasing order of their indices. After which, we label the edge (a, b) with the next lowest weight.

The processing of each edge takes $O(N)$ naively, and thus the overall complexity of the solution is $O(NE)$.

Subtask 3

The solution outlined in Subtask 2 will work for Subtask 3, as there are only a maximum of two edges in the path (a, b), and so the processing of each edge is $O(1)$ and the overall running time is $O(E)$. The implementation of this subtask is significantly easier than the general case as described in Subtask 2.

Subtask 4

In this subtask, S is a line. This makes the optimmisation of the solution outlined in Subtask 2 easier. We can maintain a set of unlabelled edges in an STL set, and the cost of processing each edge will only be equal to the number of unlabelled edges along the path (a, b). Since each edge will only be labelled once, the solution will be amortised $O(E \log E)$.

Subtask 5

G is a cycle in this subtask. To solve it, simply label the only edge that is not in S with weight N, and the rest of the edges in ascending order of their indices.

Subtask 6

In this subtask, $E = N$. Let the only edge not in S be (a, b). In ascending order of weights, we first label all the edges with indices less than that of (a, b), followed by all the unlabelled edges along the path (a, b), followed by the edge (a, b), then the remaining unlabelled edges in G in ascending order of their indices. This runs in $O(E \log E)$ if implemented efficiently.

Subtask 7

The final solution involves optimising the general solution in Subtask 2. We can use a Union-Find Disjoint Set data structure to ensure that only unlabelled edges are enumerated when an entire path has to be labelled.

To do this, we join two nodes the edge between them is labelled, and set the parent to that of the node higher up in the tree. When an entire path has to be labelled, we can repeatedly raise the node of a greater depth and merging it to its parent, until their lowest common ancestor is found. The edges enumerated can be kept track of in order to label them in the order of their indices. The overall complexity of this solution is $O(E \log E)$.

Task 5: Shuffle

Zhang Guangxuan

Abridged Problem Statement

You are not given a permutation of size N, $(A_1, A_2, \cdots A_N)$.

You are allowed to query a permutation of size N, $(X_1, X_2, \cdots X_N)$. A black box splits X into B sets of K integers. The order of integers within each set is randomized, then the order of sets is randomized. The black box applies permutation A to each element in each set of X to obtain Y ($Y_i = A_{X_i}$), and the result Y is returned.

Recover A with the least number of queries.

Subtask 1

In this subtask, $B = 2$ and $K = 3$.

There are ${}^6C_3 \div 2! = 10$ possible unique queries that can be made, well under the limit of $Q = 100$. Additionally, N is small and there are $N! = 720$ possible arrangements of A. All 10 possible queries can be generated and queried to the grader. Then iterate through all possible A and simulate the 10 queries, comparing results between the grader and the simulation. Once all results match, A is obtained.

Subtask 2

In this subtask, $B = 3$ and $K = 2$.

There are ${}^6C_2 \times^4 C_2 \div 3! = 15$ possible unique queries that can be made. A can once again be determined using the brute force method presented in Subtask 1.

Subtask 3

In this subtask, $Q = 12$ and the grader is prevented from randomizing the order among sets.

For each query, the n^{th} group of K integers in a query will in some way map to the n^{th} group of K integers in its result. If some integer i appears in the $n_1^{\text{th}}, n_2^{\text{th}}, n_3^{\text{th}}, \cdots n_q^{\text{th}}$ sets across q queries, the integer A_i will also appear in the same $n_1^{\text{th}}, n_2^{\text{th}}, n_3^{\text{th}}, \cdots n_q^{\text{th}}$ sets across q results.

The goal of this subtask is to find some sequence of queries such that each integer from 1 to N has a unique 'fingerprint' sequence of $n_1^{\text{th}}, n_2^{\text{th}}, n_3^{\text{th}}, \cdots n_q^{\text{th}}$ of sets that it appears in. A_i is determined by finding the integer which across all queries has the same 'fingerprint'.

Considering the limitations imposed by the black box, this problem can be reduced to:

> Generate N unique base-B integer IDs, such that each digit from 0 to $B-1$ appears in each place value K times

With each query we recover one digit of all IDs. A optimal construction can be guaranteed to generate N IDs with a maximum ID length (and thus optimal number of queries) of $\lceil \log_B(N) \rceil$.

Subtask 4

In this subtask, $K = 2$ and $Q = 4$.

A key observation is that this problem can be bijected to a graph problem. Each input integer can be a node. Placing two integers in a box of size $K = 2$ is to create an undirected edge between them. By querying the black box the node IDs are reassigned but the overall structure of the graph is still preserved: if a group $[i, j]$ is in the query, the black box will return a group with (A_i, A_j) in some order, indicating an edge between them. Multiple queries can be likened to creating edges of a different 'color'.

The final product is two isomorphic graphs with colored, undirected edges: one collected from data in queries and the other from the results of the black box.

To recover A_i, node i in the query graph must have the same connectivity as another node A_i in the result graph. To recover all of A, the all nodes must be uniquely identifiable by their connectivity i.e. the graph must not be automorphic.

With few queries to work with, an intuitive step would be to try create a single connected component in as few queries as needed. A cyclic graph can be made in 2 queries:

- Pair: $[1, 2], [3, 4], [5, 6] \cdots [N-3, N-2], [N-1, N]$ for 1^{st} query, colored red
- Offset groups by one and pair: $[2, 3], [4, 5], [6, 7] \cdots [N-2, N-1], [N, 1]$ for 2^{nd} query, colored blue.

The result is a cyclic graph with edges alternating in red and blue color, but this graph is still automorphic. To remove the symmetry, two more queries can be used to uniquely identify a 'root' node. There are various possible methods to do so, but one possible method to obtain node 1 as the root is is:

- Mix first two groups of first query: $[1, 3], [2, 4], [5, 6] \cdots [N-3, N-2], [N-1, N]$ for 3^{rd} query, colored green
- Mix first two groups of second query: $[2, 4], [3, 5], [6, 7] \cdots [N-2, N-1], [N, 1]$ for 4^{th}, colored yellow

These third and fourth queries will generate the same edges for the first and second queries respectively, except for the first two groups. After eliminating duplicate (green-red) and (blue-yellow) edges, note

that node 1 in the query graph and thus A_1 in the result graph will have a green edge but no yellow edge connected. The graph is no longer automorphic. The remainder of A can then be recovered by iterating through red and blue edges in the result graph, starting from A_1.

This subtask can thus be solved in 4 queries.

Subtask 5

In this subtask, $B = 2$ and $Q = 12$.

Using three queries it is possible to identify a single value of A, such as A_1. For subsequent queries, always place this 'root' element in the first set. Thus, whenever a result is received the set containing A_1 is known to be the first set, and all other elements in the same result set as A_1 must originate from elements in the first query set. The randomized order among sets can be undone, reducing the problem to the case in Subtask 3.

This subtask can thus be solved in $3 + \lceil \log_B(N) \rceil$ queries. Further optimisations (reusing query data) will bring this value to 12 in the worst case.

Subtask 6

For a more generalized version of the bijection to graph problem, in a testcase with some K a set containing some K elements is as effectively as a densely connected $k - K$ component.

For the general case with any value of K, we can reuse the strategy presented in Subtask 4 by using a approach of 'merging nodes'. For example, a set of K integers can be represented as one 'root' with ID 1 and a collective supernode with IDs $[2, 3 \cdots K]$. As each set now only contains two 'effective' nodes, we can proceed to use Subtask 4's strategy to identify the value of A_1 and the rest of the B 'root' nodes. presented

Note: the strategy presented in Subtask 4 will only require 3 queries instead of 4, because the uneven sizes of root and supernodes destroy some symmetry in the graph.

Once knowing the value of B 'root' elements in A, we can reuse the strategy presented in Subtask 5. We will place the first 'root' element in the first set, second 'root' element in the second set and so on until the B^{th} element is in the B^{th} set. When a result is received from the black box, the order among sets can be recovered by finding which 'root' element a set contains. This then reduces to the case in Subtask 3.

This subtask can thus be solved in $3 + \lceil \log_B(N) \rceil$ queries. Further optimsations (reusing query data) will bring this value to 9 in the worst case.