Task 1: Palindromic FizzBuzz

Pang Wen Yuen

Abridged Problem Statement

Print a list of integers from S to E, and replace the integer with the string "Palindrome!" if it is a palindrome.

Subtask 1

Subtask 1 has the limits of $S = E$ and $1 \leq S, E \leq 9$. Since all numbers from 1 to 9 are palindromic, and $S = E$, the solution is simply to print "Palindrome!" on a single line.

Subtask 2

Subtask 2 has the limits of $1 \leq S, E \leq 9$. This is an extension of Subtask 1, and requires contestants to know how to use a loop to print $E - S + 1$ lines with "Palindrome!" on each line.

Subtask 3

Subtask 3 has the limits of $1 \leq S, E \leq 100$. Since the number of palindromes less than 100 is quite limited, it is possible for contestants to check each number between S and E with a manually generated list of palindromic numbers, and print "Palindrome!" if the value is in the list of palindromic numbers.

Subtask 4

Subtask 4 has the limits of $1 \leq S, E \leq 10^5$. This subtask can be solved by one of two ways: a hardworking contestant can conceivably manually create a list of all the palindromic numbers below 10^5, and use the same method as stated in Subtask 3 to solve this subtask, or it can be solved by a unoptimised Subtask 6 solution which loops from 1 to E instead of S to E.

Subtask 5

Subtask 5 has the limits of $1 \leq S, E \leq 10^9$ and $S = E$. This requires contestants to check if a particular integer is a palindrome. This can be done by converting the integer into a string via the str() function in Python, or converting it manually with a while loop in C++, then checking if the ith character matches with the $(length - i - 1)$th character in the string for all $1 \leq i \leq length$.

Subtask 6

Subtask 6 has the limits of $1 \leq S, E \leq 10^9$. This subtask is an extension of Subtask 5, and requries that contestants use a loop to check if each integer between S and E is palindromic. C++ solutions that do not use the `long long` data type would pass this subtask but not Subtask 7.

Subtask 7

Subtask 7 has the limits of $1 \leq S, E \leq 10^{18}$. The solution outlined in Subtask 6 would pass this subtask as long as the `long long` data type is used.

Task 2: Lost Array

Lim Li

Abridged Problem Statement

You are not given an array of size N.

You are given 3 arrays, A_i, B_i and C_i, of sized M. You are given that for all $i = 0$ to $(M-1)$, $\min(X_{A_i}, X_{B_i}) = C_i$. Reconstruct a possible array X.

Subtask 1

Subtask 1 has the limits of $N = 2, M = 1$. Let $K = \max(A_1, B_1)$. We can simply set $X = [K, K]$.

Subtask 2

Subtask 2 has the limits of $M = 3$. Since M is small, this can be done with some case whacking. The indices that do not have any constraints on them can be assigned any value.

Subtask 3

Subtask 3 has the limits of $N, M \leq 1000$.

Note that if $\min(X_{A_i}, X_{B_i}) = C_i$, then we know that X_{A_i} and X_{B_i} cannot be less than C_i. Hence, for all $k = 1$ to n, we can set:

$$X_k = \max_{\forall p, A_p = k \text{ or } B_p = k} C_p$$

This can be proven with an exchange argument: suppose another solution, array Y, exists, then it is no worse to decrease Y_k to X_k.

This can be implemented using a for loop. For every index of the array, run a for loop and take the running max of all the C_i's that satisfies the constraint. This can be done in $O(N^2)$

Subtask 4

Subtask 4 has the limits of $C_i \leq 10, N \leq 5$.

Since $C_i \leq 10$, there exist a solution where each value of X_i is between 1 and 10 inclusive. Hence, we can brute all 10^N arrays for X and check which one is valid. To check if an array X is valid, we can for loop through all M constraints. However, note that there will be many duplicated constrains, as there are at most $N(N-1)/2$ unique pairs of indices. Hence, the checking of validity can be done in $O(N^2)$.

The total time complexity is $O({C_i}^N \times N^2)$.

Subtask 5

It uses the same idea as Subtask 3, but the time complexity can be improved to $O(N)$ by only running 1 for loop for the M constraints, and incrementing the corresponding X_i.

Task 3: Experimental Charges

Ranald Lam, William Gan

Abridged Problem Statement

N particles each have either a positive or negative charge but the charges are unknown. Particles of the same charge will repel each other, and particles of different charges will attract each other.

Q events happen where in each event, either an experiment between 2 particles is performed and the result is obtained, or a query is asked as to whether 2 particles will attract, repel or if either is possible based only on the experiments so far.

Subtask 1

In this subtask, $N = 2$ so there is only 1 possible experiment and 1 possible query. Thus in every query, we return the result of the experiment if it has been performed. Otherwise, we do not know the behaviour.

Subtask 2

In this subtask, all experiments and queries involve particle 1. Hence, there are only $N - 1$ possible experiments and queries so we can use a linear array to store the result of the experiment with each particle and particle 1. For each query, we return the result if the corresponding experiment has been performed, otherwise we do not know the behaviour.

Subtask 3

In this subtask, the result of every experiment is that the 2 particles repel. If 2 particles repel, then we know that they have the same charge. Thus, we can

create a graph where each particle is a node. This graph initially has no edges and for every experiment, we add an edge between the 2 particles involved.

In this way, any 2 particles in the same connected component must have the same charge and thus repel each other. However, if 2 particles are in different connected components, we do not know the behaviour.

Hence, by using a Union-Find Disjoint Set data structure, we are able to answer each query by checking if the 2 particles are in the same connected component. This gives us a time complexity of $O(N + Q\alpha(N))$.

Subtask 4

In this subtask, all the experiments occur before the queries. Similar to Subtask 3, we first create a graph where each particle is a node and for each experiment, we add an edge connecting the 2 nodes represented by the particles. However, in this case, we store the result of the experiment in the node.

After all the experiments, we run a graph traversal algorithm such as Depth First Search (DFS) or Breadth First Search (DFS) in each connected component. When doing so, we first assign one node in the component a positive charge without loss of generality. Then for each edge we traverse, since must have assigned a charge to at least one of the nodes, we can assign a charge to the other node if needed based on the result of the experiment attached to that edge.

For each query, we first check if they are in the same component using a Union-Find Disjoint Set data structure. If they are not in the same component, then we do not know their interaction. Otherwise, we can find the result of the query based on the charges assigned to them. This gives us a time complexity of $O(N + Q\alpha(N))$.

Subtask 5

In this subtask where $N, Q \leq 1000$, the solution idea is simililar to the previous one. However, in this case, we run the graph traversal algorithm before each query to answer the query. After each query, we reset all the charges. This gives us a time complexity of $O(NQ\alpha(N))$.

Subtask 6

In the final subtask where we have $N, Q \leq 100,000$, we need to modify the Union-Find Disjoint Set data structure such that for each connected component,

we store the list of vertices in the connected component.

We first assign a positive charge to each particle, without loss of generality. For each experiment, if the 2 particles involved are in the same connected component, we ignore it. Otherwise, we check the result of the experiment against the currently assigned charges. If the result is wrong, we iterate through all the particles in the smaller connected component and flip their charges from positive to negative and vice versa. Doing so will ensure that the experiment result matches the assigned charges. Then, we merge the 2 components as per the Union-Find Disjoint Set data structure.

The processing of the query is the same as that of the previous 2 subtasks. We leave it as an exercise to the reader as to why the final time complexity of the algorithm is $O((N + Q) \log N)$.

Task 4: Square or Rectangle?

Pang Wen Yuen

Abridged Problem Statement

You have a square or a rectangle of minimum area 400 on a 100 by 100 grid. Each query involves checking if a single grid square is inside or outside the shape. Use the minimum number of queries to determine if it is a square or a rectangle.

Subtask 1

Subtask 1 has the limits of $Q = 10^4$. That allows you to query every single square. The easiest method to do this would be to query every square, then find the minimum and maximum x and y value where it is inside the shape. Then, simply see if the difference between the minimum and maximum in the x and y axes are the same. If they are, then it is a square. Otherwise, it is a rectangle.

Subtask 2

Subtask 2 has the limits of $Q = 100$. There are multiple approaches that solve this subtask. One of the simpler approaches would be a randomised solution, where we query random squares on the grid until one of them is inside the grid. We then proceed to use binary search anchored from the point inside the grid to find the four edges of the shape, to determine if it is a square.

Subtask 3

Subtask 3 has the limits of $Q = 40$, and the shape would cover at least 25% of the grid area. A key observation in this subtask would be that out of the

squares (50, 50), (50, 51), (51, 50) and (51, 51), at least one of them must be inside the shape for it to be a square. As such, we can query these four squares. If none of them are inside the shape, we can determine that it is a rectangle. Otherwise, we can use $log_2(50) = 6$ queries with binary search to find each side of the shape. This will take a total of $4 + 6 * 4 = 28$ queries.

Subtask 4

A maximum of $Q = 33$ queries can be used to obtain full marks for this question. A key observation is that the minimum size of a square is length 20. So first, we will query a grid of 16 squares: (20, 20), (20, 40), ..., (20, 80), (40, 20), ... , (40, 80), ... (80, 80). After this point, we can consider two cases.

Case 1: At least one of the squares queried is inside the shape.

Using the queries, we would be able to know the location of each side to within 20 squares. (i.e. if (20, 20) is not inside the shape while (20, 40) is, the left side of the shape would be between 21 and 40.) We can use $log_2(20) = 5$ queries to find 3 of the 4 sides with binary search. We only need 2 queries on the last side to figure out if the shape is a square or not. As such we use exactly $16 + 3 * 5 + 2 = 33$ queries in this case.

Case 2: None of the squares queried are in the shape.

The only situation where it would be a square in this case would be that the square is of length 20 and lies completely on the right or bottom edge. We would query 9 more squares: (20, 100), (40, 100), ... (100, 100), (100, 80), ... (100, 20). If still no squares are found to be inside the shape, then it is definitely a rectangle. Otherwise, we can binary search out from the point found to be within the shape to find one side. This takes $log_2(20) = 5$ queries. Only 2 queries is needed to check the opposite side to see if it is a square (since a square would definitely be of length 20). This takes $25 + 5 + 2 = 32$ queries.