Mixtures

The crux of solving this problem is to think about it from a geometric perspective and discover a couple of properties, afterwards it's a matter of implementing efficient ways / data structures to process the queries accordingly.

There are a couple of possible approaches, but it seems easiest to translate it to a 2D geometry problem. We first translate all mixtures - the target mixture and the bottles - to 2D points in the following way: given a mixture with proportions (S, P, G) we transform it to a point $(x, y) = (S/(S + P + G), P/(S + P + G))$. Intuitively x and y are the relative amounts of salt and pepper, respectively, in the mixture. Now scaling the proportion values (i.e., multiplying S, P, G with the same coefficient, which describes an identical mixture) keeps (x, y) constant. Mixing two or more bottles ends up combining these 2D points (or vectors) with positive weights the sum of which is 1.

Having that, we consider these lemmas (proofs are left as an exercise):

Lemma 1. If a bottle point matches the target point, the target mixture can be obtained using only that one bottle.
Lemma 2. If the line segment defined by two different bottle points contains the target point, the target mixture can be obtained using those two bottles.
Lemma 3. If the triangle defined by three different bottle points contains the target point, the target mixture can be obtained using those three bottles.
Lemma 4. If the target point is not contained by any triangle defined by three different bottles, the target mixture cannot be obtained.

Based on these properties we can define the following general algorithm:

1. Maintain the set of points.
2. At each step, check the state to find the answer:
   2a. If there is a point matching the target point $\Rightarrow$ 1.
   2b. Otherwise, if there is a pair of points whose line segment contains the target point $\Rightarrow$ 2.
   2c. Otherwise, if there is a triplet of points whose triangle contains the target point $\Rightarrow$ 3.
   2d. Otherwise $\Rightarrow$ 0.

Subtask 1 ($N \leq 50$)

At each step we can simply check each point / pair of points / triplet of points to see if we have case (2a), (2b), or (2c). This takes $O(N)$ / $O(N^2)$ / $O(N^3)$ time for each query, so the total time complexity to process all queries is $O(N^4)$.

Total Complexity: $O(N^4)$

Subtask 2 ($N \leq 500$)

Full search on all points / pairs of points is still feasible here. We need to speed-up the case (2c): checking the triangles. Here the key observation is that instead of considering all triangles individually we can check whether the target point is inside the convex hull defined all bottle points; namely, if the target point is inside the convex hull there exists at least one triplet of bottle points the triangle of which contains the target point, and if it is outside the hull no such triplet exists. It's possible to build the convex hull and check if the target point is inside it in time $O(N \log N)$, but for this subtask a sub-optimal approach up to $O(N^2)$ is also good enough.

Total Complexity: $O(N^3)$

Subtask 3 ($N \leq 5000$)

Full search on all single points still feasible. To check triplets we do the same convex hull approach as in the previous subtask, but we have to use the optimal $O(N \log N)$ method this time. For pairs of points we need something better. If we fix a point as one end of the line segment, to have the target point on the segment we know that the other point has to be exactly opposite from the first point relative to the target point (direction-wise, the distance doesn't matter). In other words, after fixing the first point we know exactly at what angle the second point should be (with respect to the target point). So if we have a data structure that we can store points in and test for existence by angle (e.g., using tangent value), we can load all current points in it and then go through each bottle point and quickly check whether an opposite point currently exists. If there is never an opposite point, it means we don't have any line segment containing the target point, and vice versa. It can be done in $O(N \log N)$ for each query.

Total Complexity: $O(N^2 \log N)$

Subtask 4 ($N \leq 10^5$)

For the previous subtasks we were answering each query completely independently. For the full solution we aim for a $O(\log N)$ time for each query so we need to find a way to maintain the state of points through time so that we can update the state (add / remove bottles) and check the current answer quickly for each query. Let's look at all cases (single point, pair, triplet) separately:

a. For single point checks we can keep the points in a structure that let's us add/remove/find in $O(\log N)$ time. We can also just maintain a counter for matching bottles that we increase/decrease whenever we add/remove a point that matches the target point - then at any step the answer is 1 if the counter is greater than zero.

b. For solving the line segment case we go back to the previous idea. If we have all bottle points stored in an appropriate data structure, we need $O(N \log N)$ time per query to check whether there exist opposite points (N points, $O(\log N)$ check if an opposite point exists), which is too slow. However, we can maintain a counter for the total number of pairs of points that are opposite to each other (with respect to the target point) within the set of all current points, and update it after each query. Then, similarly to the single point case, the answer is 2 if this counter is greater than zero. To achieve this we need to maintain a data structure that stores all angles of points (e.g., tangent value) and checks for (and allows to update) the number of elements with a certain value currently stored. This can be done in $O(\log N)$ time for each operation.

c. For the triplet case we can use the same convex hull idea, but we need a dynamic version that we can update query-to-query (add/remove points) and test whether the target point is inside it. This can be done with an amortized time of $O(\log N)$ per query. However, we don't really need to construct the exact convex hull itself, we just need a way to tell whether the (single fixed) target point is inside of it. If we order all points around the target point by angle, then it's enough to check whether all angles between consecutive points are less than 180 degrees. If that's not the case (i.e., some two consecutive points are more than 180 degrees apart) then the target point is outside the hull (and the answer is 0), otherwise inside (answer 3). We need to maintain the points ordered in such a way by adding/removing points for each query, and be able to check for angles bigger than 180 degrees. There are various ways how to do this technically, and it can be done in $O(\log N)$ time per query.

Total Complexity: $O(N \log N)$.

Final notes

1. In the end for the final solution all cases can be handled using the same data structure that stores the points/angles, so the solution becomes relatively concise.

2. This problem requires divisions. You can simply use floating point numbers, but there is a risk of making the wrong discrete decisions due to floating point imprecision. The correct way here is to work with rational number, i.e., store and operate with values as numerators and denominators (p/q). However, you still have to be careful to not cause overflows. Note that subtasks have varying constraints on the proportion values, which allows more freedom in operations without causing overflows. The full problem has a constraint of 10^6, and it is possible to implement a solution that only multiplies these values once so we use 2nd order values. (3rd order is also tolerated, but more typical careless approaches that yield 4th or 8th degree values are penalized in later subtasks).