Problem 1. Great Box Shuffling (Author: Sam Lee)

Problem Statement (Formally):

You are given two arrays a, b of length n. Then you will be given q queries, for each query, you will read four positive integers l_1, r_1, l_2, r_2, you will answer whether you can shuffle the subarray $a_{l_1}, a_{l_1+1}, \cdots, a_{r_1}$ into the subarray $b_{l_2}, b_{l_2+1}, \cdots, b_{r_2}$.

Solution:

1. **Subtask 1.** $n, q \leq 1000$

   You can easily do this deterministically with any kind of method. The easist way would be to sort the two subranges, and then check if they are the same. Some people also used methods like checking the frequencies to achieve this. This can easily be done in $O(n)$ or $O(n \log n)$ per query. Notice that no matter which method you use, you will need to think of some common things that would show up in both ranges in order to

2. **Subtask 2. No additional constraints**

   The intended solution for this problem is to apply **Hashing**. It is not an easy task to check whether two subarrays are equal since it would usually take at least linear time to do it.

   The idea of hashing is to create a hash function f such that:

   (a) For two inputs x, y, if $x = y$ then $f(x) = f(y)$.

   (b) For two inputs x, y, if $f(x) \neq f(y)$ then $x \neq y$ **with high probability**.

   (c) We can compute $f(x)$ in $O(1)$ or $O(\log n)$

   There are several ways to create this hash function. The simplest way is to simply map each integers into a different 32-bit or 64-bit integer, then compute the sum of the subarrays.

   You might ask, why would this work? We will check if this method satisfies the three conditions.

   (a) It is clear that for two subarrays, (a) is satisfied since we mapped the same integers to a same value, so if two subarrays are equal, the elements after mapping will also be equal, and therefore the sum will be equal.

   (b) We can compute the prefix sums $p_0 = 0$, $p_i = \sum_{k=1}^{i} a_k$ of the arrays. Then to compute the sum of the subarray $[l, r]$ of an array, you can simply take $p_r - p_{l-1}$. This takes $O(n)$ precomputation, and $O(1)$ per query, which is efficient enough for our purposes.

   (c) Now, for (2). We won't provide a rigorous proof here. Assume we map the integers to a 32-bit integer, which will fall into the range $[-2^{31}, 2^{31} - 1]$. Then, the sum of an subarray of length k will be hashed to an integer $i \in [-2^{31}k, (2^{31} - 1)k]$. The probability that two subarrays has the same hashing will be approximately $\frac{1}{2^{32}k} \approx \frac{1}{2^{32}} \approx 2.32 \times 10^{-10}$, which is really low. Therefore, we can assume that it wouldn't happen.

   This problem is a great demonstration on how randomization can help simplify a problem. It might seems like the problem is really hard to be solved fast, however, applying some randomization can easily speed up the algorithm.

Similar Problems for Practice (Competitive Programming Problems):

1. Codeforces 2014H: $O(1)$ per query if all elements in a subarray $[l, r]$ has a even frequency. Problem link: https://codeforces.com/contest/2014/problem/H
2. AtCoder Beginner Contest 238G: $O(\log n)$ per query if the product of a subarray $[l, r]$ is a cubic number. Problem link: https://atcoder.jp/contests/abc238/tasks/abc238_g

3. Codeforces 1514D: $O(\log n)$ per query the majority element (appears more than $\lceil \frac{n}{2} \rceil$ times in an array. Problem link: https://codeforces.com/problemset/problem/1514/D

Problem 2. Scroll of Secrets (Author: Sam Lee)

Problem Statement (Formally):

There is a hidden linked list where each node is indexed an integer in $[1, n]$, and the values are strictly-increasing. You will also be given the index of the starting node of the linked list. Each time you can ask a question about the value and the next index of a node with index i. You can only ask at most 5000 questions. Find the first number on the linked list that is greater or equal to x, if there is none, output -1. **Main Constraint:** $1 \leq n \leq 10^5$

Solution

1. **Subtask 1.** You can ask at most 10^5 questions.

   This is a subtask for the students to be familiar with interactive problems. There are two main solutions to solve this problem:

   (a) Start from the start index, and traverse through the whole linked list by asking at most n questions.

   (b) Ask all n indices, and find the minimum value within all those node values that is $\geq x$.

   Both solutions would fully solve this subtask. However, only (a) would be helpful for the full solution of this problem.

2. **Subtask 2.** You can ask at most 60000 questions.

   There is actually no intended solution of this subtask. However, if you apply any kind of randomization that doesn't work well, then your solution might be able to pass this subtask.

3. **Subtask 3.** You can ask at most 5000 questions.

   Similar to problem 1. Since we can't ask for all the $O(n)$ informations, we need to somehow find a way to make sure we can compute the answer **with a high probability**.

   Therefore, we will apply the following steps for this problem, and there are two steps:

   (a) Use half of the quota for a random sampling. Keep track of the index of the node with a **maximum value** $\leq x$.

   (b) Use the other half of the quota to apply the solution (a) of subtask 1 starting from the index found from previous step.

   We claim that this algorithm has a high probability of finding the correct answer, and we will provide a proof.

   Let the length of the linked list be n, and the index node of the answer be i. Let's choose some positive integer $k \leq n$. Now, suppose we random sample r integers within the interval $[1, n]$. The probability of an integer from the r integers chosen falling in the range $[i-k+1, i]$ is $1 - \left(\frac{n-k}{n}\right)^r$.

   If we plug in $n = 10^5,\ r = 2500,\ n = 2500$, then the probabilty of success would be $\approx 1 - 10^{-28}$. Hence, this method should always work.

   This problem provides a different perspective of randomization compared to the hashing used in Problem 1. However, both problems utilized the idea of randomization to speed up our algorithms.