398. Random Pick Index

Monday, March 13, 2017 3:05 PM

Given an array of integers with possible duplicates, randomly output the index of a given target number. You can assume that the given target number must exist in the array.

Note:

The array size can be very large. Solution that uses too much extra space will not pass the judge.

Example:

```
int[] nums = new int[] {1,2,3,3,3};
Solution solution = new Solution(nums);
// pick(3) should return either index 2, 3, or 4 randomly. Each index should have equal probability of
returning.
solution.pick(3);

// pick(1) should return 0. Since in the array only nums[0] is equal to 1.
solution.pick(1);
```

From <https://leetcode.com/problems/random-pick-index/#/description>

(Note: The problem statement is erronious, as it does not define any ordering for the given elements and then presents an example of a sorted list. The actual test programs present the algorithm with unsorted data, which in my opinion violates the questions contract. I made the decision to assume that the array would contain unsorted values because in the absence of clarification, it was more important to design an algorithm that worked than designing one with optimal memory use.)

Thoughts:

1. What kind of problem is this?
    a. Searching. We're searching for the indices at which specific values appear, and choosing one of them randomly.

2. How do I solve this problem?
    a. A map between values and recorded indices would make the algorithm very fast and not require very much time to set up. Using this technique, the setup requires O(n) space and O(n) time, and the search requires O(1) time and O(1) space.
        i. If the arrays have large numbers of runs of duplicates, we can save space by recording the runs rather than then every index.

    b. If the array is in sorted order, we can mark the start and length of each run of duplicate numbers. Using this technique, setup requires O(n) time and O(k) space, where k is the number of unique values in the array. Searching is again O(1) for both time and space.

    c. Since space is at a premium, we could do no processing at all on the list initially and simply pick a random occurrence to return. We can iterate through the list, keeping track of the occurrences , and when they match. we can return that one. If we hit the end of the list, we randomly return one of the indices we found.

       Using this technique, the setup requires O(1) time and space complexity. The search operates in O(n) time and O(k) space, where k is the number of occurrences of a given number.

        i. We can make sure we never hit the end of the array if we record the counts of each number first. Then we can pick a number between 0 and c. Setting up has a time complexity of O(n) and a space complexity O(k) where k is the number of unique values in the array. Doing the actual search is O(n) time, with O(1) space complexity.

ii. NOTE: This is very similar to the complexities for strategy b, except without the restriction that the data be ordered. The only real difference is the O(n) search time, which is our trade-off for not being able to "lump" values together into runs.

1. Implemented Strategy C:
   a. Results:
      i. Time Exceeded.
         1) Requiring Integer instead of int is causing slowdown for large inputs.
      ii. Resubmitted and was accepted.
      iii. Resubmitted several more times, all accepted.