Documentation on Random Pick Index (Leetcode 398)

1. Problem Statement

The problem requires designing a class that efficiently picks a random index of a target number from a given list, ensuring that all valid indices have an equal probability of being selected. This can be achieved using a dictionary to store each number's indices during initialization, allowing for quick lookups and uniform random selection.

2. Intuition

The intuition behind this solution is that iterating over the list every time a pick(target) is called would be inefficient, especially with large input sizes. Instead, by precomputing and storing the indices of each number in a dictionary, the selection process can be reduced to a simple lookup followed by random choice. This eliminates unnecessary computations and improves performance.

3. Key Observations

One key observation is that since the problem guarantees that the target exists in the list, we do not need to handle cases where the target is missing. This simplifies the implementation, as we can directly retrieve the indices from our precomputed dictionary and randomly select one using Python’s random.choice() function.

4. Approach

The approach consists of two main steps. First, we traverse the list once and store the indices of each number in a dictionary, where the key is the number and the value is a list of its occurrences. Second, when pick(target) is called, we retrieve the list of indices corresponding to the target and return a randomly chosen index. This ensures that each valid index has an equal probability of being selected.

5. Edge Cases

Handling edge cases is crucial for robustness. The solution must work efficiently even when the list contains a single element, when all elements are the same, when there are negative numbers, or when the list is at its maximum length of $2 * 10^4$. Since the target is always present, we do not need to handle cases where no index exists.

6. Complexity Analysis

- **Time Complexity:** The preprocessing step runs in O(n), as we traverse the list once to construct the dictionary. The pick(target) function runs in O(1), since dictionary lookups and random selection are constant-time operations.

- **Space Complexity:** The space complexity is O(n), as we store all indices in a dictionary, which can be large for lists with many unique elements.

7. Alternative Approaches

There are alternative approaches to solve this problem. A brute force method would involve iterating through nums every time pick(target) is called, leading to O(n) time complexity per query. Another approach is **Reservoir Sampling**, which selects one index randomly while iterating through nums, requiring O(n) time but using O(1) space. However, our dictionary-based approach is the most optimal for frequent queries.

8. Code Implementation

The implementation follows a clean and structured design, ensuring easy maintenance and understanding. The constructor initializes the dictionary with index mappings, and pick(target) retrieves the list of indices before returning a randomly chosen one. This ensures that our solution runs efficiently even for large inputs.

9. Test Cases

To validate correctness, we can test cases such as lists with single elements, multiple occurrences of a number, and large lists. The expected results should show uniform randomness when selecting indices. Edge cases such as negative numbers and large input sizes should also be tested to ensure the solution remains efficient.

10. Final Thoughts

In conclusion, this approach balances time efficiency and space usage, making it suitable for real-world applications requiring frequent random selections from a large dataset. It provides a simple yet powerful way to ensure uniform probability distribution while maintaining optimal performance.