Random Pick with Weight – Full Documentation

1. Problem Statement

Given a list of positive integers w, where each element w[i] represents the weight of index i, implement a function pickIndex() such that:

- It randomly picks an index in the range [0, w.length - 1].
- The probability of picking index i is proportional to its weight:

Probability(i) = $\frac{\omega[i]}{\sum\omega}$

2. Intuition

We want to simulate a weighted random choice. A common strategy is to treat the weights like ranges on a number line. The greater the weight, the larger the corresponding range. If we generate a random number in that range, the region it falls into tells us which index to pick.

3. Key Observations

- The total range can be defined as [1, sum(w)].
- If we map each weight to a cumulative prefix sum, each index occupies a unique segment in this range.
- Using binary search, we can efficiently find which segment the random number falls into.

4. Approach

- Preprocessing:
  - Create a prefix sum array from the weights.
  - Store the total weight sum.

- pickIndex():
  - Generate a random integer r in range [1, total sum].
  - Use bisect_left() to find the first prefix sum >= r.
  - Return the corresponding index.

5. Edge Cases

- Only one element in w: always return index 0.
- All weights equal: returns uniform distribution.
- Large weights: handled via prefix sums, no overflow with Python's int.

6. Complexity Analysis

□ Time Complexity:

- Constructor (__init__): O(n) for creating the prefix sum array.
- pickIndex(): O(log n) using binary search.

📦 Space Complexity:

- O(n) for storing the prefix sum array.

7. Alternative Approaches

- Reservoir Sampling with weights: More complex and less efficient.
- Binary Tree approach: For dynamic updates, use segment trees or binary indexed trees. But for static weight lists, prefix sums + binary search is optimal.

8. Test Cases

Test Case 1:

```
w = [1]
obj = Solution(w)
print(obj.pickIndex())  # Always 0
```

Test Case 2:

```
w = [1, 3]
obj = Solution(w)
results = [obj.pickIndex() for _ in range(1000)]
print(results.count(0), results.count(1))  # Approx. 250 and 750
```

Test Case 3:

```
w = [10, 0, 0, 0]
obj = Solution(w)
assert all(obj.pickIndex() == 0 for _ in range(100))
```

9. Final Thoughts

This problem showcases a powerful technique of combining prefix sums and binary search to perform probabilistic decisions efficiently. It's a common pattern in scenarios like weighted shuffling, lottery simulation, or randomized algorithms where bias is needed.