Solution: Random Pick with Weight

Let's solve the Random Pick with Weight problem using the Modified Binary Search pattern.

We'll cover the following

Statement

You're given an array of positive integers, `weights`, where `weights[i]` is the weight of the i^{th} index.

Write a function, **Pick Index()**, which performs weighted random selection to return an index from the `weights` array. The larger the value of `weights[i]`, the heavier the weight is, and the higher the chances of its index being picked.

Suppose that the array consists of the weights $[12, 84, 35]$. In this case, the probabilities of picking the indexes will be as follows:

- **Index 0:** $12/(12 + 84 + 35) = 9.2\%$
- **Index 1:** $84/(12 + 84 + 35) = 64.1\%$
- **Index 2:** $35/(12 + 84 + 35) = 26.7\%$

Constraints:

- $1 \leq$ `weights.length` $\leq 10^4$
- $1 \leq$ `weights[i]` $\leq 10^5$
- **Pick Index()** will be called at most 10^4 times.

> **Note:** Since we're randomly choosing from the options, there is no guarantee that in any *specific* run of the program, any of the elements will be selected with the exact expected frequency.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

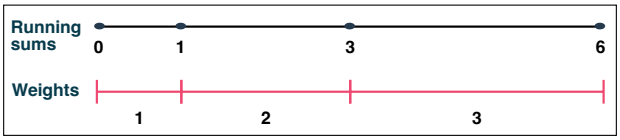
To correctly add bias to the randomized pick, we need to generate a probability line where the length of each segment is determined by the corresponding weight in the input array. Positions with greater weights have

longer segments on the probability line, and positions with smaller weights have shorter segments. We can represent this probability line as a list of running sums of weights.

Let's implement a linear search approach as our naive solution:

- Starting off with an example of an input list of weights containing $[1, 2, 3]$, the list of the running sums of weights is $[1, (1 + 2) = 3, (3 + 3) = 6]$.

The segment length can be used as the likelihood metric of each index. The likelihood metric (based on the running sums) for each weight is shown in the diagram to the right.



- Next, a random number between the range of 0 and the total sum of weights is generated to make a biased selection.
- We then locate the position of this number on the probability line, corresponding to an index in the input array.
- By scanning the running sums list, we find the first sum larger than the random number and return the corresponding index.

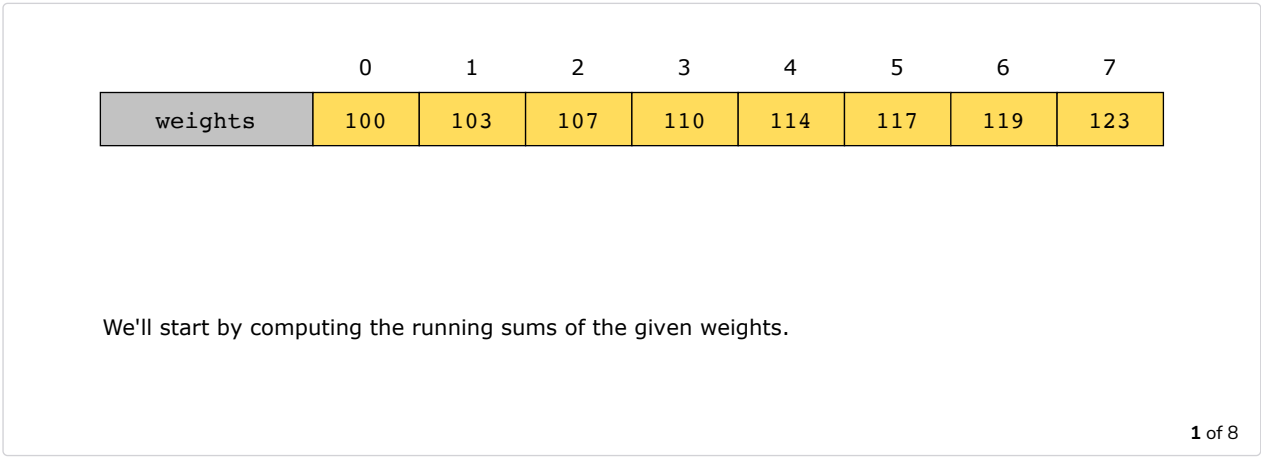
Both the time and the space complexity of this naive approach is $O(n)$.

Optimized approach using modified binary search

We can use binary search to speed up the search step in the naive approach. This will reduce the time complexity of our **Pick Index()** function. Since the running sums list is sorted in ascending order, the search will execute in $O(\log n)$ time, which is faster than the initial linear scanning method.

Here's how the algorithm proceeds:

1. In **Init()**, we generate the list of running sums from the given list of weights so that we don't have to compute it again every time we call **Pick Index()**.
2. The **Pick Index()** method returns an index at random, taking into account the weights provided:
    - Generate a random number, `target`, between 1 and k, where k is the largest value in the list of running sums of weights.
    - Use binary search to find the index of the first running sum that is greater than the random value. Initialize the `low` index to 0 and the `high` index to the length of the list of running sums of weights. While the `low` index is less than or equal to the `high` index:
        - Calculate the `mid` index as $\lfloor (\texttt{low} + (\texttt{high} - \texttt{low})/2 \rfloor$.
        - If the running sum at the `mid` index is less than or equal to `target`, update the `low` index to `mid + 1`.
        - Otherwise, update the `high` index to `mid`.
    - At the end of the binary search, the `low` pointer will point to the index of the first running sum greater than `target`. Return the index found as the chosen index.



Let’s look at the code for this solution below:

Java

```java
import java.util.*;

class RandomPickWithWeight {
    private List<Integer> runningSums;
    private int totalSum;

    public RandomPickWithWeight(int[] weights) {
        // List to store running sums of weights
        runningSums = new ArrayList<>();
        // Variable to calculate running sum
        int runningSum = 0;

        // Iterate through the given weights
        for (int w : weights) {
            // Add the current weight to the running sum
            runningSum += w;
            // Append the running sum to the running_sums list
            runningSums.add(runningSum);
        }
        // Store the total sum of weights
        totalSum = runningSum;
    }

    // Method to pick an index based on the weights
        Random random = new Random();
        int target = random.nextInt(totalSum) + 1;
```

Random Pick with Weight

Solution summary

To recap, the solution to this problem can be divided into the following three parts:

1. Generate a list of running sums from the given list of weights.
2. Generate a random number, the range for the random number begins from 1 and ends at the largest number in the running sums list.
3. Use binary search to find the index of the first running sum that is greater than the random value.

Time complexity

Constructor: Since the list of running sums list is calculated in the constructor, the time complexity for the constructor is $O(n)$, where n is the size of the array of weights.

Pick Index(): Since we're performing a binary search on a list of length n, the time complexity is $O(\log n)$.

Space complexity

Constructor: The list of running sums takes $O(n)$ space during its construction.

Pick Index(): This function takes $O(1)$ space, since constant space is utilized.

Back
Random Pick with We...

Next
Find K Closest Elemen...

Mark as