

[🔍 Previous \(/articles/iterator-for-combination/\)](/articles/iterator-for-combination/) [🔍 Next \(/articles/vertical-order-traversal-of-a-binary-tree/\)](/articles/vertical-order-traversal-of-a-binary-tree/)

528. Random Pick with Weight (/problems/random-pick-with-weight/)

May 6, 2020 | 4.2K views

Average Rating: 5 (14 votes)

Given an array `w` of positive integers, where `w[i]` describes the weight of index `i`, write a function `pickIndex` which randomly picks an index in proportion to its weight.

Note:

1. `1 <= w.length <= 10000`
2. `1 <= w[i] <= 10^5`
3. `pickIndex` will be called at most `10000` times.

Example 1:

Input:

```
["Solution", "pickIndex"]
```

```
[[[1]], []]
```

Output: `[null, 0]`

Example 2:

Input:

```
["Solution", "pickIndex", "pickIndex", "pickIndex", "pickIndex", "pickIndex"]
```

```
[[[1, 3]], [], [], [], [], []]
```

Output: `[null, 0, 1, 1, 1, 0]`

Explanation of Input Syntax:

The input is two lists: the subroutines called and their arguments. `Solution`'s constructor has one argument, the array `w`. `pickIndex` has no arguments. Arguments are always wrapped with a list, even if there aren't any.

Solution

Overview

This is actually a very practical problem which appears often in the scenario where we need to do **sampling** over a set of data.

Nowadays, people talk a lot about machine learning algorithms. As many would reckon, one of the basic operations involved in training a machine learning algorithm (e.g. Decision Tree) is to sample a batch of data and feed them into the model, rather than taking the entire data set. There are several rationales behind doing sampling over data, which we will not cover in detail, since it is not the focus of this article.

If one is interested, one can refer to our Explore card of Machine Learning 101 (<https://leetcode.com/explore/learn/card/machine-learning-101/>) which gives an overview on the fundamental concepts of machine learning, as well as the Explore card of Decision Tree (<https://leetcode.com/explore/learn/card/decision-tree/>) which explains in detail on how to construct a decision tree algorithm.

Now, given the above background, hopefully one is convinced that this is an interesting problem, and it is definitely worth solving.

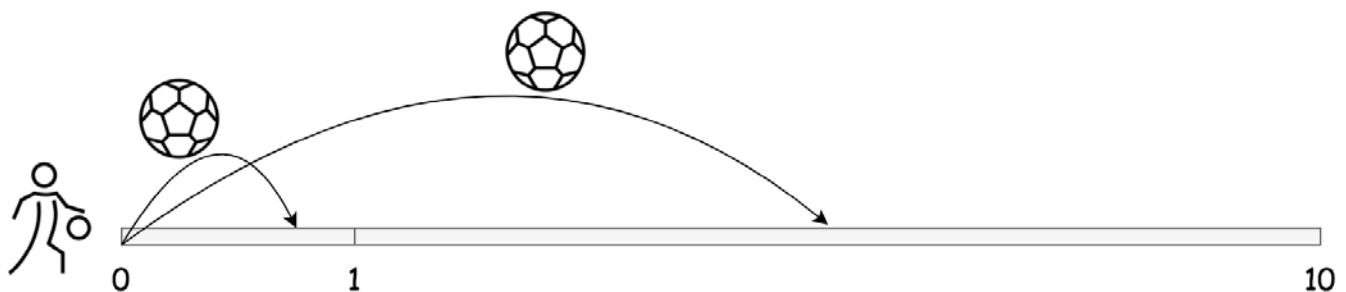
Intuition

Given a list of positive values, we are asked to *randomly* pick up a value based on the weight of each value. To put it simple, the task is to do **sampling with weight**.

Let us look at a simple example. Given an input list of values $[1, 9]$, when we pick up a number out of it, the chance is that 9 times out of 10 we should pick the number 9 as the answer.

In other words, the **probability** that a number got picked is proportional to the value of the number, with regards to the total sum of all numbers.

To understand the problem better, let us imagine that there is a line in the space, we then project each number into the line according to its value, i.e. a large number would occupy a broader range on the line compared to a small number. For example, the range for the number 9 should be exactly nine times as the range for the number 1.



Now, let us throw a ball **randomly** onto the line, then it is safe to say there is a good chance that the ball will fall into the range occupied by the number 9. In fact, if we repeat this experiment for a large number of times, then *statistically* speaking, 9 out of 10 times the ball will fall into the range for the

number 9 .

Articles > 528. Randor

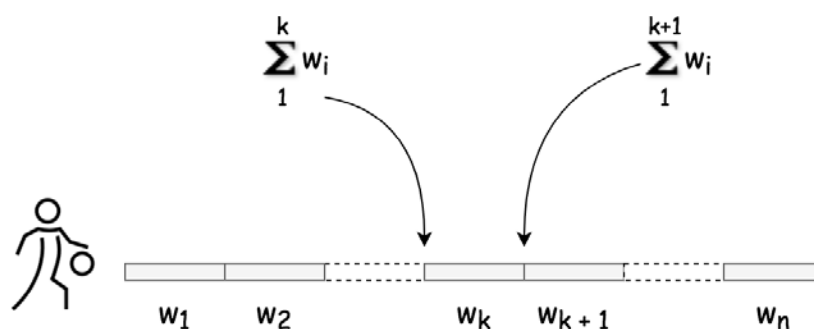
Voila. That is the intuition behind this problem.

Simulation

So to solve the problem, we can simply **simulate** the aforementioned experiment with a computer program.

First of all, let us construct the line in the experiment by **chaining up** all values together.

Let us denote a list of numbers as $[w_1, w_2, w_3, \dots, w_n]$. Starting from the beginning of the line, we then can represent the **offsets** for each range K as $(\sum_1^K w_i, \sum_1^{K+1} w_i)$, as shown in the following graph:



As many of you might recognize now, the offsets of the ranges are actually the prefix sums (https://en.wikipedia.org/wiki/Prefix_sum) from a sequence of numbers. For each number in a sequence, its corresponding prefix sum, also known as **cumulative sum**, is the sum of all previous numbers in the sequence plus the number itself.

As an observation from the definition of prefix sums, one can see that the list of prefix sums would be *strictly* monotonically increasing, if all numbers are positive.

To throw a ball on the line is to find an *offset* to place the ball. Let us call this offset **target**.

Once we randomly generate the target offset, the task is now boiled down to finding the range that this target falls into.

Let us rephrase the problem now, given a list of offsets (*i.e.* prefix sums) and a target offset, our task is to fit the target offset into the list so that the ascending order is maintained.

Approach 1: Prefix Sums with Linear Search

Intuition

If one comes across this problem during an interview, one can consider the problem almost resolved, once one reduces the original problem down to the problem of inserting an element into a sorted list.

Concerning the above problem, arguably the most intuitive solution would be **linear search**. Many of you might have already thought one step ahead, by noticing that the input list is *sorted*, which is a sign to apply a more advanced search algorithm called **binary search**.

Let us do one thing at one time. In this approach, we will first focus on the linear search algorithm so that we could work out other implementation details. In the next approach, we will then improve upon this approach with a binary search algorithm.

So far, there is one little detail that we haven't discussed, which is how to *randomly* generate a target offset for the ball. By "randomly", we should ensure that each point on the line has an equal opportunity to be the target offset for the ball.

In most of the programming languages, we have some `random()` function that generates a random value between 0 and 1. We can **scale up** this randomly-generated value to the entire range of the line, by multiplying it with the size of the range. At the end, we could use this *scaled* random value as our target offset.


As an alternative solution, sometimes one might find a `randomInteger(range)` function that could generate a random integer from a given range. One could then directly use the output of this function as our target offset.

Here, we adopt the `random()` function, since it could also work for the case where the weights are float values.

Algorithm

We now should have all the elements at hand for the implementation.

- First of all, before picking an index, we should first set up the playground, by generating a list of prefix sums from a given list of numbers. The best place to do so would be in the constructor of the class, so that we don't have to generate it again and again at the invocation of `pickIndex()` function.
 - In the constructor, we should also keep the total sum of the input numbers, so that later we could use this total sum to scale up the random number.

- For the `pickIndex()` function, here are the steps that we should perform.  Articles > 528. Random Pick with Weight
 - Firstly, we generate a random number between 0 and 1. We then scale up this number, which will serve as our `target` offset.
 - We then scan through the prefix sums that we generated before by *linear search*, to find the first prefix sum that is larger than our `target` offset.
 - And the index of this prefix sum would be exactly the *right* place that the target should fall into. We return the index as the result of `pickIndex()` function.

C++
Java
Python3

Copy

```

1 class Solution {
2     vector<int> prefixSums;
3 public:
4     Solution(vector<int> &w) {
5         for (auto n : w)
6             prefixSums.push_back(n + (prefixSums.empty() ?
7                 0 : prefixSums.back()));
8     }
9     int pickIndex() {
10         auto target = rand() % prefixSums.back();
11         // run a linear search to find the target zone
12         for (int i = 0; i < prefixSums.size(); ++i)
13             if (target < prefixSums[i])
14                 return i;
15         return prefixSums.size() - 1;
16     }
17 };


```

Complexity Analysis

Let N be the length of the input list.

- Time Complexity
 - For the constructor function, the time complexity would be $\mathcal{O}(N)$, which is due to the construction of the prefix sums.
 - For the `pickIndex()` function, its time complexity would be $\mathcal{O}(N)$ as well, since we did a linear search on the prefix sums.

- Space Complexity

 Articles > 528. Randor

- For the constructor function, the space complexity would be $\mathcal{O}(N)$, which is again due to the construction of the prefix sums.
- For the `pickIndex()` function, its time complexity would be $\mathcal{O}(1)$, since it uses constant memory. Note, here we consider the prefix sums that it operates on, as the input of the function.

Approach 2: Prefix Sums with Binary Search

Intuition

As we promised before, we could improve the above approach by replacing the linear search with the **binary search**, which then can reduce the time complexity of the `pickIndex()` function from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$.

As a reminder, the condition to apply binary search on a list is that the list should be *sorted*, either in ascending or descending order. For the list of prefix sums that we search on, this condition is guaranteed, as we discussed before.

Algorithm

We could base our implementation largely on the previous approach. In fact, the only place we need to modify is the `pickIndex()` function, where we replace the linear search with the binary search.

As a reminder, there exist built-in functions of binary search in almost all programming languages. If one comes across this problem during the interview, it might be acceptable to use any of the built-in functions.


On the other hand, the interviewers might insist on implementing a binary search by hand. It would be good to prepare for this request as well.

There are several code patterns to implement a binary search algorithm, which we cover in the Explore card of Binary Search algorithm (<https://leetcode.com/explore/learn/card/binary-search/>). One can refer to the card for more details.

C++

Java

Python3

 Articles > 528. Random Copy

```
1 class Solution {
2     vector<int> prefixSums;
3
4 public:
5     Solution(vector<int> &w) {
6         for (auto n : w)
7             prefixSums.push_back(n + (prefixSums.empty() ?
8                                     0 : prefixSums.back()));
9     }
10    int pickIndex() {
11        auto target = rand() % prefixSums.back();
12        return upper_bound(begin(prefixSums), end(prefixSums), target) - begin(prefixSums);
13    }
14};
```

Complexity Analysis

Let N be the length of the input list.

- Time Complexity

- For the constructor function, the time complexity would be $\mathcal{O}(N)$, which is due to the construction of the prefix sums.
- For the `pickIndex()` function, this time its time complexity would be $\mathcal{O}(\log N)$, since we did a binary search on the prefix sums.

- Space Complexity

- For the constructor function, the space complexity remains $\mathcal{O}(N)$, which is again due to the construction of the prefix sums.
- For the `pickIndex()` function, its time complexity would be $\mathcal{O}(1)$, since it uses constant memory. Note, here we consider the prefix sums that it operates on, as the input of the function.

Rate this article:

[Previous \(/articles/iterator-for-combination/\)](/articles/iterator-for-combination/)[Articles](#) > 528. Random[Next \(/articles/vertical-order-traversal-of-a-binary-tree/\)](/articles/vertical-order-traversal-of-a-binary-tree/)Comments: **5**

Sort By ▼



Type comment here... (Markdown is supported)

Preview

Post



(/aditya2)

aditya2 (/aditya2) ★ 4 🕒 May 25, 2020 9:48 AM

Although this problem seems intimidating at first, the basic idea and implementation is based on common sense.

3 ▲ ▼ | Share | Reply

SHOW 1 REPLY



(/jiangleetcode)

jiangleetcode (/jiangleetcode) ★ 16 🕒 May 31, 2020 8:12 PM

Thanks for the explanation. I like the C++ example code which many other articles do not have. I feel like a second class citizen when I use C++ here :)

1 ▲ ▼ | Share | Reply



(/zhdv)

zhdv (/zhdv) ★ 0 🕒 19 minutes ago

The method to get a random value in a range, the author use, is not uniform. For more information you can search or read this: <https://ericlippert.com/2013/12/16/how-much-bias-is-introduced-by-the-remainder-technique/> (<https://ericlippert.com/2013/12/16/how-much-bias-is-introduced-by-the-remainder-technique/>)

0 ▲ ▼ | Share | Reply



(/javaman775)

javaman775 (/javaman775) ★ 0 🕒 2 days ago

How the input given in the examples are applicable to this ?

0 ▲ ▼ | Share | Reply



(/ycl2112)

ycl2112 (/ycl2112) ★ 0 🕒 2 days ago

in the python3 binary search solution, shouldn't line 17 be `(self.total_sum - 1) * random.random()` because you want to generate a random number corresponding to the indices of `prefix_sum` which are from 0 to `total_sum-1`

0 ▲ ▼ | Share | Reply

SHOW 4 REPLIES