<https://leetcode.com/problems/random-pick-with-weight/description/>

You are given a **0-indexed** array of positive integers w where w[i] describes the **weight** of the ith index.

You need to implement the function pickIndex(), which **randomly** picks an index in the range [0, w.length - 1] (**inclusive**) and returns it. The **probability** of picking an index i is w[i] / sum(w).

For example, if w = [1, 3], the probability of picking index 0 is 1 / (1 + 3) = 0.25 (i.e., 25%), and the probability of picking index 1 is 3 / (1 + 3) = 0.75 (i.e., 75%).

**Example 1:**

Input

["Solution","pickIndex"]

[[[1]],[]]

Output

[null,0]

Explanation

Solution solution = new Solution([1]);

solution.pickIndex(); // return 0. The only option is to return 0 since there is only one element in w.

**Example 2:**

Input

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

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

Output

[null,1,1,1,1,0]

Explanation

Solution solution = new Solution([1, 3]);

solution.pickIndex(); // return 1. It is returning the second element (index = 1) that has a probability of 3/4.

solution.pickIndex(); // return 1

solution.pickIndex(); // return 1

solution.pickIndex(); // return 1

solution.pickIndex(); // return 0. It is returning the first element (index = 0) that has a probability of 1/4.

Since this is a randomization problem, multiple answers are allowed.

All of the following outputs can be considered correct:

[null,1,1,1,1,0]

[null,1,1,1,1,1]

[null,1,1,1,0,0]

[null,1,1,1,0,1]

[null,1,0,1,0,0]

......

and so on.

**Constraints:**

* 1 <= w.length <= 10^4
* 1 <= w[i] <= 10^5
* pickIndex will be called at most 10^4 times.

**Attempt 1: 2023-08-12**

**Solution 1: Binary Search (60min)**

class Solution {

// e.g w = {1,3}

// -> weighted mapping index arr = {0,1,1,1}

// -> consider arr[i] as target index used in binary search and return i as final result index(refer to L35/P12.2.Search Insert Position) instead of weighted mapping index arr, we can create an accumulated arr = {1,4}, similar to L35, when try to find random value in accumulate arr, either we find exactly matched arr[i] or not find, we can return an index(if not find just return a position where we can insert random value), and also in random value generation, the random.nextInt(n) parameter n will equal to max index range = 4, becuase arr index is 0-3, but for random.nextInt(n) method, to generate random value between 0-3, have to pass actual arr length for random.nextInt method, as n = 4

// Declaration :

// public int nextInt(int n)

// Parameters:n, this is the bound on the random number to be returned. Must be positive.

// Returns a random number, between 0 (inclusive) and n (exclusive).

// e.g w = {1,5,2}

// -> weighted mapping index arr = {0,1,1,1,1,1,2,2}

// -> accumulated arr = {1,6,8}, max index range = 8

// e.g w = {2,5,3,4}

// -> weighted mapping arr = {0,0,1,1,1,1,1,2,2,2,3,3,3,3}

// -> accumulated arr = {2,7,10,14}, max index range = 14

int max\_range = 0;

int[] arr;

Random random = new Random();

public Solution(int[] w) {

arr = new int[w.length];

arr[0] = w[0];

max\_range += w[0];

for(int i = 1; i < w.length; i++) {

arr[i] = arr[i - 1] + w[i];

max\_range += w[i];

}

}

// Refer L35/P12.2.Search Insert Position

// Convert problem into find the most recent value position smaller

// or equal to target

// 1. If find the value equal to target, return position

// 2. If not find, return most recent smaller value's position plus 1

// e.g for accumulated arr = {2,7,10,14}

// idx in [1,2] return 0

// idx in [3,7] return 1

// idx in [8,10] return 2

// idx in [11,14] return 3

// if target = random.nextInt(14) + 1 -> 9, expect return 2, because even

// 9 not a value in {2,7,10,14}, but we just need to find which index

// we can insert 9 into {2,7,10,14}

public int pickIndex() {

// Don't forget + 1 to get random value between [1,n] rather than

// original range of nextInt as [0,n)

int target = random.nextInt(max\_range) + 1;

int lo = 0;

int hi = arr.length - 1;

while(lo <= hi) {

int mid = lo + (hi - lo) / 2;

if(arr[mid] == target) {

return mid;

} else if(arr[mid] > target) {

hi = mid - 1;

} else {

lo = mid + 1;

}

}

// 'hi' is the upper boundary, add '1' is the insert position

return hi + 1;

}

}

/\*\*

\* Your Solution object will be instantiated and called as such:

\* Solution obj = new Solution(w);

\* int param\_1 = obj.pickIndex();

\*/

**Refer to**

<https://leetcode.com/problems/random-pick-with-weight/solutions/605331/clear-problem-explanation-thought-process-optimal-code/>

The idea here is that each every time we call random pick we want to get back an index that is picked randomly. Each index i has a weight w[i] that represents how likely it can appear in a random pick relative to other indices.

What does that mean? Let's take a look at the second example. The weights array is [1,3]This simply means that if we're randomly picking an index from this array, index 1 is three times more likely to appear in a random pick than index 0.

Well, how can we do this? Let's say we have a deck of 4 cards and I "own" 3 cards out of them, while my friend owns only one card, and we were to pick a card randomly. One can simply say that it's three times more likely for the picked card to be mine than it would be for my friend. In other words, the probability to get a card that is mine is 3/4 which is 3 times my friend's 1/4 probability.

Let's use this idea of owning numbers in our problem. In our [1,3] example, each index will "own" a group of numbers such that if we picked any of these numbers we would return that index. The weight of the index 0 is 1 so it will own a single number, let's say it's zero. We're then going to move to the next index and we find out that its weight is 3, so we're going to give it 3 numbers to own. Index 0 already owns 0 so we'll give index 1 the next 3 numbers [1,2,3].

We've only used numbers in the range [0,3]. Our numbers range is mapped to indices as follows:

Number Index0 01 12 13 1

So picking indices will now translate to picking a random int in our numbers range, in this example [0,3] and then mapping this random int to its corresponding index.

To do this efficiently we're going to build a cum array which contains the sum of all weights up to this index. Each element in this cum array represents the largest number "owned" by that index. When we need to pick a random index we'll pick a random number in our range [0,3] and then using binary search we'll locate the index in which it should be placed.

The time complexity of pickIndex is O(logN) time complexity where N is the sum of all weights in our array.

Here is the code:

import random

from bisect import bisect\_left

class Solution:

def \_\_init\_\_(self, w: List[int]):

total=-1

self.cum=[]

for weight in w:

total+=weight

self.cum.append(total)

def pickIndex(self) -> int:

randVal=random.randint(0,self.cum[-1])

return bisect\_left(self.cum,randVal)

**Refer to**

<https://leetcode.com/problems/random-pick-with-weight/solutions/154044/java-accumulated-freq-sum-binary-search/>

Use accumulated freq array to get idx.

w[] = {2,5,3,4} => wsum[] = {2,7,10,14}

then get random val random.nextInt(14)+1, idx is in range [1,14]

idx in [1,2] return 0

idx in [3,7] return 1

idx in [8,10] return 2

idx in [11,14] return 3

then become LeetCode 35. Search Insert Position

Time: O(n) to init, O(logn) for one pick

Space: O(n)

class Solution {

Random random;

int[] wSums;

public Solution(int[] w) {

this.random = new Random();

for(int i=1; i<w.length; ++i)

w[i] += w[i-1];

this.wSums = w;

}

public int pickIndex() {

int len = wSums.length;

int idx = random.nextInt(wSums[len-1]) + 1;

int left = 0, right = len - 1;

// search position

while(left < right){

int mid = left + (right-left)/2;

if(wSums[mid] == idx)

return mid;

else if(wSums[mid] < idx)

left = mid + 1;

else

right = mid;

}

return left;

}

}

**Refer to**

<https://leetcode.com/problems/random-pick-with-weight/solutions/154432/very-easy-solution-based-on-uniform-sampling-with-explanation/>

say we have the numbers 1, 5, 2 easiest solution is to construct the following arrayarr[] = {0,1,1,1,1,1,2,2}then generate a random number between 0 and 7 and return the arr[rnd]. This is solution is not really good though due to the space requirements it has (imagine a number beeing 2billion).

The solution here is similar but instead we construct the following array (accumulated sum){1, 6, 8} generate a number between 1-8 and say all numbers generated up to 1 is index 0. all numbers generated greater than 1 and up to 6 are index 1 and all numbers greater than 6 and up to 8 are index 2. After we generate a random number to find which index to return we use binary search.

class Solution {

int[] arr;

int max = 0;

Random random = new Random();

public Solution(int[] w) {

int[] arr = new int[w.length];

arr[0] = w[0];

max += w[0];

for(int i=1; i<w.length; i++){

arr[i] = arr[i-1] + w[i];

max+=w[i];

}

this.arr = arr;

}

public int pickIndex() {

int rnd = random.nextInt(max) + 1; // generate random number in [1,max]

//this returns the index of the random number,

//if the number does not exist in the array it returns -(the position it should have been) - 1

int ret = Arrays.binarySearch(arr, rnd);

if(ret < 0) ret = -ret - 1; //if not in the array

return ret;

}

}