<https://leetcode.ca/all/1570.html>

Given two sparse vectors, compute their dot product.

Implement class SparseVector:

**SparseVector(nums)** Initializes the object with the vector nums

**dotProduct(vec)** Compute the dot product between the instance of SparseVector and vec

A **sparse vector** is a vector that has mostly zero values, you should store the sparse vector **efficiently** and compute the dot product between two SparseVector.

**Follow up:** What if only one of the vectors is sparse?

**Example 1:**

**Input:** nums1 = [1,0,0,2,3], nums2 = [0,3,0,4,0]

**Output:** 8

**Explanation:** v1 = SparseVector(nums1) , v2 = SparseVector(nums2)

v1.dotProduct(v2) = 1\*0 + 0\*3 + 0\*0 + 2\*4 + 3\*0 = 8

**Example 2:**

**Input:** nums1 = [0,1,0,0,0], nums2 = [0,0,0,0,2]

**Output:** 0

**Explanation:** v1 = SparseVector(nums1) , v2 = SparseVector(nums2)

v1.dotProduct(v2) = 0\*0 + 1\*0 + 0\*0 + 0\*0 + 0\*2 = 0

**Example 3:**

**Input:** nums1 = [0,1,0,0,2,0,0], nums2 = [1,0,0,0,3,0,4]

**Output:** 6

**Constraints:**

n == nums1.length == nums2.length

1 <= n <= 10^5

0 <= nums1[i], nums2[i] <= 100

**Attempt 1: 2024-06-16**

**Solution 1: Hash Table (10 min)**

In this solution:

**SparseVector Class:**

The SparseVector class stores the non-zero elements of the vector using a HashMap where the key is the index and the value is the element at that index.

The constructor initializes the HashMap by iterating through the input array and adding only the non-zero elements.

**dotProduct Method:**

The dotProduct method calculates the dot product of the current SparseVector instance with another SparseVector instance.

It iterates through the entries of the first vector's indexValueMap and for each entry, checks if the corresponding index exists in the second vector's indexValueMap.

If the index exists in both vectors, it multiplies the values and adds the result to the result variable.

import java.util.HashMap;

import java.util.Map;

class SparseVector {

private Map<Integer, Integer> indexValueMap;

SparseVector(int[] nums) {

indexValueMap = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

if (nums[i] != 0) {

indexValueMap.put(i, nums[i]);

}

}

}

// Return the dotProduct of two sparse vectors

public int dotProduct(SparseVector vec) {

int result = 0;

for (Map.Entry<Integer, Integer> entry : indexValueMap.entrySet()) {

int index = entry.getKey();

int value = entry.getValue();

if (vec.indexValueMap.containsKey(index)) {

result += value \* vec.indexValueMap.get(index);

}

}

return result;

}

}

// Example usage

public class Main {

public static void main(String[] args) {

int[] nums1 = {1, 0, 0, 2, 3};

int[] nums2 = {0, 3, 0, 4, 0};

SparseVector v1 = new SparseVector(nums1);

SparseVector v2 = new SparseVector(nums2);

int result = v1.dotProduct(v2);

System.out.println("Dot Product: " + result); // Output: 8

}

}

**Solution with follow up concern:**

**Follow up:** What if only one of the vectors is sparse?

**When one of the vectors is sparse and the other is not, you can optimize the dot product calculation by iterating over the non-zero elements of the sparse vector and accessing the corresponding elements in the dense vector directly. Here is an implementation that addresses this follow-up:**

import java.util.HashMap;

import java.util.Map;

class SparseVector {

private Map<Integer, Integer> indexValueMap;

SparseVector(int[] nums) {

indexValueMap = new HashMap<>();

for (int i = 0; i < nums.length; i++) {

if (nums[i] != 0) {

indexValueMap.put(i, nums[i]);

}

}

}

// Return the dotProduct of this sparse vector and a dense vector

public int dotProductWithDense(int[] denseVec) {

int result = 0;

for (Map.Entry<Integer, Integer> entry : indexValueMap.entrySet()) {

int index = entry.getKey();

int value = entry.getValue();

result += value \* denseVec[index];

}

return result;

}

}

// Example usage

public class Main {

public static void main(String[] args) {

int[] sparseNums = {1, 0, 0, 2, 3};

int[] denseNums = {0, 3, 0, 4, 0};

SparseVector sparseVec = new SparseVector(sparseNums);

int result = sparseVec.dotProductWithDense(denseNums);

System.out.println("Dot Product: " + result); // Output: 8

}

}

**Solution with promotion on only start with smaller map:**

class SparseVector {

// Using a HashMap to efficiently store non-zero elements and their positions

private Map<Integer, Integer> nonZeroElements = new HashMap<>();

// Constructor to populate the map with non-zero elements from the input array

SparseVector(int[] nums) {

for (int i = 0; i < nums.length; ++i) {

if (nums[i] != 0) {

nonZeroElements.put(i, nums[i]);

}

}

}

// Return the dotProduct of two sparse vectors

public int dotProduct(SparseVector vec) {

// Reference to the smaller of the two maps to iterate over for efficiency

Map<Integer, Integer> smallerMap = nonZeroElements;

Map<Integer, Integer> largerMap = vec.nonZeroElements;

// Swap if 'vec's map has fewer elements to iterate over the smaller map

if (largerMap.size() < smallerMap.size()) {

Map<Integer, Integer> temp = smallerMap;

smallerMap = largerMap;

largerMap = temp;

}

int productSum = 0; // The result of the dot product operation

// Iterating through the smaller map and multiplying the matching values

for (var entry : smallerMap.entrySet()) {

int index = entry.getKey();

int value = entry.getValue();

productSum += value \* largerMap.getOrDefault(index, 0);

}

return productSum; // Return the computed dot product

}

}

**Refer to**

<https://algo.monster/liteproblems/1570>

**Problem Description**

The task is to create a class, SparseVector, which encapsulates the concept of a sparse vector and implements a method to compute the dot product of two sparse vectors. A sparse vector is defined as a vector that contains mostly zeros and therefore should not be stored in a typical dense format as that would waste space. The goal is to store such vectors efficiently and perform operations on them.

To efficiently store the sparse vector, we can use a dictionary to hold only the non-zero elements, where the keys are the indices of these elements, and the values are the elements themselves. This way we aren't storing all the zero values, which can dramatically reduce memory usage for very sparse vectors.

The SparseVector class has two methods:

**SparseVector(nums):** The constructor takes a list of integers nums and initializes the sparse vector using a dictionary comprehension that filters out the zero values.

**dotProduct(vec):** This method computes the dot product between the vector represented by the current instance of SparseVector and another sparse vector, vec.

The dot product of two vectors is computed by multiplying corresponding entries and summing those products. In the case of sparse vectors, most of these products will be zero, because they involve multiplications by zero, so we only need to consider the non-zero entries.

The follow-up question asks about efficiently computing the dot product if only one of the two vectors is sparse.

**Intuition**

For the solution, the idea is to leverage the sparsity of the vectors to optimize the dot product computation. Given that most of the elements in the vectors are zeros, we want to perform multiplications only for the non-zero elements. By converting the vectors into a dictionary structure with non-zero elements, we can quickly identify which elements actually need to be multiplied.

**In the dotProduct method, we iterate over the items in the smaller dictionary (to optimize the number of iterations) and multiply values by the corresponding values in the other dictionary, if they exist. If an index does not exist in the other dictionary, it means that the value for that index in the other vector is zero and thus does not contribute to the dot product. Therefore, we use the .get method to handle such cases, which allows us to specify a default value of 0 when an index is not found.**

When dealing with one sparse and one non-sparse vector, the current approach still works efficiently because the dot product will focus on iterating over the non-zero elements of the sparse vector and lookup the corresponding values in the non-sparse vector.

**Solution Approach**

**Algorithm and Data Structure**

**Dictionary for Sparse Representation:** A Python dictionary is an ideal data structure for representing a sparse vector. It allows storing key-value pairs where the key is the index of a non-zero element in the original vector, and the value is the non-zero element itself. This structure is memory efficient since we only store entries for non-zero elements.

**Constructor** **init**:

We use a dictionary comprehension in the constructor to iterate over nums using enumerate to get both the index i and the value v together.

The condition if v ensures that we're only storing the non-zero values (as zero is considered False in Python).

The resulting dictionary self.d holds only the elements of the input list that are non-zero, along with their indices.

**dotProduct Method:**

We receive another SparseVector object, vec, and we want to compute the dot product with the current sparse vector instance.

We access the internal dictionaries of the current instance (a) and the vec (b) — these dictionaries store the indices and values of the non-zero entries.

By comparison of the lengths of these two dictionaries, we choose to iterate over the smaller dictionary (here represented as a). This is an optimization step; since the dot product will be zero for all indices not present in both vectors, we can skip the zero values of the larger vector and only iterate over the potential non-zero counterparts.

We then use a generator expression to iterate over the items of a: for i, v in a.items().

For each element, we calculate the product v \* b.get(i, 0). The get method of the dictionary is very handy in this case, as it will return 0 if the index i doesn't exist in b—a common occurrence with sparse vectors, and also safe considering the default value for any index not in the dictionary would be zero.

Finally, the sum function accumulates all the products to give us the result of the dot product.

**Pattern Used**

**Efficient Computation with Sparse Representation:** By representing the vectors in a sparse form, computations are made more efficient since we ignore all zero-product cases which would contribute nothing to the final sum.

**Iterating Over a Smaller Set:** Choosing to iterate over the smaller set of elements to reduce the number of operations is a common optimization strategy in algorithms involving collection processing.

Combining these algorithms and patterns, the SparseVector class efficiently implements the computation of a dot product between two sparse vectors, considering only the meaningful, non-zero elements, and thus avoiding unnecessary computations.

By using a compressed representation for the vectors with dictionaries and strategically leveraging the sparsity, the implementation maximizes efficiency in terms of both time and space complexity.

**Example Walkthrough**

Let's walk through a small example to illustrate the solution approach.

Suppose we have two sparse vectors represented as follows:

Vector A: [1, 0, 0, 2, 0]

Vector B: [0, 3, 0, 4, 0]

Using the given solution approach, we first convert these lists into SparseVector objects. Let's call these objects sparseA and sparseB. When initializing these objects, our dictionary comprehension will filter out the zeroes and store only the non-zero elements and their indices:

sparseA's internal dictionary will have the elements {0: 1, 3: 2}, corresponding to indices and values (index 0 has value 1, and index 3 has value 2).

sparseB's internal dictionary will have the elements {1: 3, 3: 4}.

To find the dot product of sparseA and sparseB, we invoke the dotProduct method on one of them, let's say sparseA.dotProduct(sparseB).

Here's a step-by-step explanation:

Inside the dotProduct method, we compare the sizes of the internal dictionaries of sparseA and sparseB. Since both have two elements, we can choose either one to iterate over, but for the sake of this example, we will iterate over sparseA because we called the method on it.

We now loop over the items in sparseA's dictionary. For each item, we look up whether the corresponding index is in sparseB's dictionary. We have two iterations in our loop:

First iteration: For index 0 in sparseA, there is no corresponding index in sparseB. When we attempt to multiply 1 (from sparseA) with sparseB.get(0, 0) (using .get to specify a default value of 0), the result is 0 since index 0 is not present in sparseB.

Second iteration: For index 3 in sparseA, there is a matching index in sparseB which has the value 4. We perform the multiplication 2 (from sparseA) \* 4 (from sparseB) which equals 8.

We sum the results of the multiplications. In this example, the only non-zero result came from the second iteration (8), so the sum and thus the dot product of A and B is 8.

In conclusion, the SparseVector class successfully computes the dot product of sparseA and sparseB as 8 using an efficient approach, taking advantage of the sparse representation by only considering non-zero elements and their indices in the computations.

**Solution Implementation**

import java.util.HashMap;

import java.util.Map;

class SparseVector {

// Using a HashMap to efficiently store non-zero elements and their positions

private Map<Integer, Integer> nonZeroElements = new HashMap<>();

// Constructor to populate the map with non-zero elements from the input array

SparseVector(int[] nums) {

for (int i = 0; i < nums.length; ++i) {

if (nums[i] != 0) {

nonZeroElements.put(i, nums[i]);

}

}

}

// Return the dotProduct of two sparse vectors

public int dotProduct(SparseVector vec) {

// Reference to the smaller of the two maps to iterate over for efficiency

Map<Integer, Integer> smallerMap = nonZeroElements;

Map<Integer, Integer> largerMap = vec.nonZeroElements;

// Swap if 'vec's map has fewer elements to iterate over the smaller map

if (largerMap.size() < smallerMap.size()) {

Map<Integer, Integer> temp = smallerMap;

smallerMap = largerMap;

largerMap = temp;

}

int productSum = 0; // The result of the dot product operation

// Iterating through the smaller map and multiplying the matching values

for (var entry : smallerMap.entrySet()) {

int index = entry.getKey();

int value = entry.getValue();

productSum += value \* largerMap.getOrDefault(index, 0);

}

return productSum; // Return the computed dot product

}

}

// Example of usage:

// SparseVector v1 = new SparseVector(nums1);

// SparseVector v2 = new SparseVector(nums2);

// int ans = v1.dotProduct(v2);

**Time and Space Complexity**

**Time Complexity:**

The constructor **init** has a time complexity of O(n) where n is the number of elements in nums, as it needs to iterate through all elements to create the dictionary with non-zero values.

The dotProduct function has a time complexity of O(min(k, l)) where k and l are the number of non-zero elements in the two SparseVectors, respectively. This is because the function iterates over the smaller of the two dictionaries (after ensuring a has the smaller length, swapping if necessary) and attempts to find matching elements in the larger one. The get operation on a dictionary has an average case time complexity of O(1).

**Space Complexity:**

The space complexity of the **init** function is O(k), where k is the number of non-zero elements in nums, since the space required depends on the stored non-zero elements.

The dotProduct function operates in O(1) space complexity because it calculates the sum on the fly and does not store intermediate results or allocate additional space based on input size, other than a few variables for iteration and summing.