1331. Rank Transform of an Array

Hash Table Sorting **Easy**

Problem Description

The problem states that we have an array of integers called arr. We need to transform this array so that instead of the original

Rank is assigned starting from 1.

numbers, each element is replaced by its "rank". The rank of an element in the array is determined by the following rules:

- The bigger the number, the higher the rank it should have.
- If two numbers are the same, they should have the same rank.
- The rank should be as small as possible, meaning that the smallest number gets rank 1, the second smallest gets rank 2, and so on, without any gaps in the ranks.
- Therefore, the task is to rewrite the array in such a way that the numbers are replaced with their corresponding ranks.

ntuition

To come up with a solution to this problem, we can use a step-by-step approach. The first part of the problem suggests a sorting

duplicates (elements of equal value should have the same rank). To do this effectively, we can utilize the set data structure in Python, which stores only unique elements. Here is the intuition broken down: • First, sort the unique elements of the array. This can be done by converting the array to a set to remove duplicates, and then converting it back to a list to sort it. This gives us a sorted list of the unique elements in ascending order.

operation since we need to determine the order of the elements (to assign ranks properly). However, we must also handle

• Second, to find the rank of the original elements, we need to determine their position in this sorted list of unique elements. • Third, we use the bisect_right function from the bisect module in Python, which is designed to find the position in a sorted list where an

element should be inserted to maintain sorted order. However, in our case, it effectively gives us the number of elements that are less than or

- equal to our target number. Since our list is already sorted, this is equivalent to finding the rank.
- Finally, apply the bisect_right operation to each element in the original array to replace it with its rank. The elegance of this solution lies in its simplicity and efficiency. By using sorting and binary search (which bisect_right
- **Solution Approach**

employs), we arrive at a solution that is both clear and optimized for performance.

• sorted(...): Takes the set of unique elements and returns a sorted list of these elements, t.

The implementation of the solution can be understood by breaking down the steps and the algorithms or data structures used in each.

The code begins by defining a method arrayRankTransform within the Solution class which takes one argument, the array arr.

The first line within the method: t = sorted(set(arr))

• set(arr): Converts the list arr into a set, removing any duplicate elements.

return [bisect_right(t, x) for x in arr]

• Every unique element will have a unique rank in increasing order.

involves two operations:

Now that t holds a sorted list of unique elements from the original array, the next step is to find out the rank of each element in

the original array arr according to its position in t.

The following line:

uses a list comprehension to create a new list by going through each element x in the original array arr. • For each element x, bisect_right(t, x) is called, which uses a binary search to find the insertion point for x in t to the right of any existing

entries. Essentially, it finds the index of the first element in the sorted list t that is greater than x.

elements less than or equal to x. This is exactly the rank of x because: • The smallest element in arr will have a rank of 1 (because there will be 0 elements smaller than it in t). • If elements in arr are equal, they will have the same rank since bisect_right will return the same index for them.

Since t is sorted and contains every unique value from arr, the index returned by bisect_right corresponds to the number of

does so without the need for cumbersome loops or manually implementing a binary search, instead relying on Python's built-in high-performance algorithms.

By combining set, sorted, and bisect_right, the solution effectively creates a ranking system for the elements in the array arr. It

Finally, the method returns the array which consists of ranks corresponding to each element in arr. This list represents the transformed version of the input where each value is replaced by its rank, effectively solving the problem.

Let's illustrate the solution approach with a small example. Consider the following array:

Example Walkthrough

Following the solution approach step by step, we first convert the arr into a set to remove duplicates, and then sort it: Convert to set:

This set does not include duplicates and now contains only the unique values 10, 20, and 40.

t = [10, 20, 40]

rank is 3.

arr = [3, 2, 1, 1]

Example array

class Solution:

class Solution {

t = sorted(set(arr))

ranked_arr = arrayRankTransform(arr)

ranked_arr is now [3, 2, 1, 1]

Sort the unique elements:

provided by the bisect_right function:

{10, 20, 40}

arr = [40, 20, 10, 10]

The next step is to find the rank for each element in arr by determining its position in the sorted list t using the binary search

Now we have a sorted list t with the unique elements from the array in ascending order.

• For the first 10 in arr, bisect_right(t, 10) returns 1, since there are no elements in t that are less than 10. Hence, its rank is 1. • The second 10 in arr will have the same rank as the first 10 since they are equal. Therefore, bispect_right(t, 10) again returns 1.

When we replace each element in arr with its calculated rank, we get the final transformed array:

Each number in the original array has now been replaced with its respective rank, completing the process. The code executing the described steps for the provided example would look like this:

This demonstrates how the algorithm effectively translates the initial array into one representing the ranks of each element.

• For the first element 40 in arr, bisect_right(t, 40) returns 3 because there are two elements in t that are less than or equal to 40. Hence, its

• For the element 20 in arr, bisect_right(t, 20) returns 2, as there is only one element less than or equal to 20. Hence, its rank is 2.

- from bisect import bisect_right def arrayRankTransform(arr):
- arr = [40, 20, 10, 10]# Transforming the array

For every element 'x' in `arr`, find its rank. The rank is determined by the

position of 'x' in the `unique_sorted_arr` plus one because bisect_right will

Solution Implementation **Python** from typing import List from bisect import bisect_right

unique_sorted_arr = sorted(set(arr))

public int[] arrayRankTransform(int[] arr) {

for (int i = 0; i < arrayLength; ++i) {</pre>

// Create an array to hold the answers

function arrayRankTransform(arr: number[]): number[] {

// Count unique elements in the sorted array

// Sort the array while preserving the original via spreading

if (i === 0 || sortedUniqueElements[i] !== sortedUniqueElements[i - 1]) {

const binarySearch = (sortedArray: number[], arrayLength: number, target: number) => {

sortedUniqueElements[uniqueCounter++] = sortedUniqueElements[i];

const sortedUniqueElements = [...arr].sort((a, b) => a - b);

for (let i = 0; i < sortedUniqueElements.length; ++i) {</pre>

// Binary search function to find the rank of an element

if (sortedArray[midIndex] >= target) {

return leftIndex + 1; // The rank is index + 1

def arrayRankTransform(self, arr: List[int]) -> List[int]:

unique sorted arr = sorted(set(arr))

Create a sorted list of the unique elements in `arr`

Since ranks are 1-indexed, the position itself is the rank.

Return the list of ranks corresponding to each element in `arr`

ranks = [bisect_right(unique_sorted_arr, x) for x in arr]

equal to n. Therefore, this step takes 0(n + k * log(k)).

const midIndex = (leftIndex + rightIndex) >> 1;

int[] ranks = new int[arrayLength];

// Get the size of the input array

int[] sortedArray = arr.clone();

int arrayLength = arr.length;

Arrays.sort(sortedArray);

int uniqueSize = 0;

def arrayRankTransform(self, arr: List[int]) -> List[int]:

Create a sorted list of the unique elements in `arr`

Since ranks are 1-indexed, the position itself is the rank.

// Create a copy of the array to sort and find unique elements

// This will be the actual size of the array of unique elements

// Loop through the sorted array to filter out the duplicates

if (i == 0 || sortedArray[i] != sortedArray[i - 1]) {

sortedArray[uniqueSize++] = sortedArray[i];

// Assign the rank to each element in the original array

// If it's the first element or it's different from the previous, keep it

ranks = [bisect_right(unique_sorted_arr, x) for x in arr]

return [bisect_right(t, x) for x in arr]

Return the list of ranks corresponding to each element in `arr` return ranks Java

give us the insertion point which is the number of elements that are less than or equal to 'x'.

```
for (int i = 0; i < arrayLength; ++i) {</pre>
            // Binary search finds the index of the current element in the unique array
            // Since array indexing starts at 0, add 1 to get the correct rank
            ranks[i] = Arrays.binarySearch(sortedArray, 0, uniqueSize, arr[i]) + 1;
       // Return the ranks array containing the ranks of each element in arr
       return ranks;
C++
#include <vector>
#include <algorithm>
class Solution {
public:
   // Function to transform an array into ranks of elements
    vector<int> arrayRankTransform(vector<int>& arr) {
       // Create a copy of the original array to sort and find unique elements
       vector<int> sortedArr = arr;
       sort(sortedArr.begin(), sortedArr.end());
       // Remove duplicates from sortedArr to get only unique elements
       sortedArr.erase(unique(sortedArr.begin(), sortedArr.end()), sortedArr.end());
       // Prepare a vector to store the answer
       vector<int> ranks;
       // For each element in the original array, find its rank
        for (int element : arr) -
           // Find the position (rank) of the element in the sorted unique array
           // The rank is the index + 1 because ranks are 1-based
            int rank = upper_bound(sortedArr.begin(), sortedArr.end(), element) - sortedArr.begin();
            ranks.push_back(rank);
       // Return the vector of ranks corresponding to the original array's elements
       return ranks;
};
```

```
// Compute the rank for each element in the original array
for (const element of arr) {
    ranks.push(binarySearch(sortedUniqueElements, uniqueCounter, element));
```

return ranks;

from typing import List

class Solution:

from bisect import bisect_right

return ranks

Time and Space Complexity

};

TypeScript

let uniqueCounter = 0;

let leftIndex = 0;

} else {

let rightIndex = arrayLength;

// Create an array for the answer

const ranks: number[] = [];

// Return the array of ranks

while (leftIndex < rightIndex) {</pre>

rightIndex = midIndex;

leftIndex = midIndex + 1;

```
Time Complexity
  The time complexity of the given code can be broken down into the following parts:
```

For every element 'x' in `arr`, find its rank. The rank is determined by the

position of 'x' in the `unique_sorted_arr` plus one because bisect_right will

give us the insertion point which is the number of elements that are less than or equal to 'x'.

- [bisect_right(t, x) for x in arr]: For each element x in the original arr, we perform a binary search using bisect_right. Binary search takes 0(log(k)) time per search, and since we are performing it for each element n times, this step takes 0(n *
- log(k)) time. Thus, the overall time complexity is 0(n + k * log(k) + n * log(k)), which simplifies to 0(n * log(k)) because n will typically be larger than k and hence n * log(k) would be the dominant term.

sorted(set(arr)): Creating a set from the array removes duplicates and takes 0(n) time, where n is the number of elements

in arr. Sorting the set will take 0(k * log(k)) time, where k is the number of unique elements in arr which is less than or

Space Complexity The space complexity can be considered as follows:

t = sorted(set(arr)): Creating a set and then a sorted list from the array takes 0(k) space where k is the number of unique elements.

- The list comprehension does not use additional space that depends on the size of the input (other than space for the output list, which is always required for the problem). The output list itself takes O(n) space.
- In conclusion, the space complexity is O(n + k), which simplifies to O(n) if we do not count the output space as extra space (as is common in complexity analysis), or if we assume that k is at most n.