493. Reverse pairs

July 3, 2017 | 46.9K views

Given an array `nums`, we call `(i, j)` an ***important reverse pair*** if `i < j` and `nums[i] > 2*nums[j]`.

You need to return the number of important reverse pairs in the given array.

Example1:

```
Input: [1,3,2,3,1]
Output: 2
```

Example2:

```
Input: [2,4,3,5,1]
Output: 3
```

Note:

1. The length of the given array will not exceed `50,000`.
2. All the numbers in the input array are in the range of 32-bit integer.

Solution

Approach 1: Brute Force

Intuition

Do as directed in the question. We can simply check all the pairs if they are important reverse pairs or not.

Algorithm

- Iterate over i from 0 to $size - 1$
  - Iterate over j from 0 to $i - 1$
    - If $\text{nums}[j] > 2 * \text{nums}[i]$, increment count

C++

```
int reversePairs(vector<int>& nums)
{
    int n = nums.size();
    int count = 0;
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (nums[i] > nums[j] * 2LL)
                count++;
        }
    }
    return count;
}
```

Complexity Analysis

- Time complexity: $O(n^2)$
  - We iterate over all the possible pairs wherein ($i<j$) in the array which is $O(n^2)$
- Space complexity: $O(1)$ only constant extra space for n, $count$ etc.

Trivia

The above code can be expressed as one-liner in Python:

Python

```
def reversePairs(self, nums):
    return sum([nums[i] > 2 * nums[j] for i in range(len(nums)) for j in range(0, i)])
```

Herein, we iterate over all the pairs and store the boolean values for $\text{nums}[i] > 2 * \text{nums}[j]$. Finally, we count the number of true in the array to get the result.

Approach 2: Binary Search Tree

Intuition

In Approach 1, for each element i, we searched the subarray $[0, i)$ for elements such that their value is greater than $2 * \text{nums}[i]$. In the previous approach, the search is linear. However, we need to make the process efficient. Maybe, memoization can help, but since, we need to compare the elements, we cannot find a linear DP solution.

Observe that the indices of the elements in subarray $[0, i)$ don't matter as we only require the count. So, we can sort the elements and perform binary search on the subarray. But, since the subarray keeps growing as we iterate to the next element, we need a data structure to store the previous result as well as to allow efficient searching(preferably $O(\log n)$) - Binary Search Tree(BST) could be a good bet.

Refreshing BST

BST is a rooted binary tree, wherein each node is associated with a value and has 2 distinguishable sub-trees namely *left* and *right* subtree. The left subtree contains only the nodes with lower values than the parent's value, while the right subtree conatins only the nodes with greater values than the parent's value.

Voila!

This is exactly what is required. So, if we store our elements in BST, then we can search the larger elements thus eliminating the search on smaller elements altogether.

Algorithm

Define the Node of BST that stores the val and pointers to the left and right. We also need a count of elements(say count_ge) in the subtree rooted at the current node that are greater than or equal to the current node's val. count_ge is initialized to 1 for each node and left, right pointers are set to NULL.

We define the insert routine that recursively adds the given val as an appropriate leaf node based on comparisons with the $Node.val$. Each time, the given val is smaller than $Node.val$, we increment the count_ge and move the val to the right subtree. While, if the val is equal to the current $Node$, we simply increment the count_ge and exit. While, we move to the left subtree in case $(\text{val} < Node.val)$.

We also require the $search$ routine that gives the count of number of elements greater than or equal to the target. In the $search$ routine, if the $head$ is NULL, return 0. Otherwise, if $\text{target} == \text{head.val}$, we know the count of values greater than or equal to the target, hence simply return head.count_ge. In case, $\text{target} < \text{head.val}$, the ans is calculated by adding Node.count_ge and recursively calling the search routine with head.left. And if $\text{target} > \text{head.val}$, ans is obtained by recursively calling the search routine with head.right.

Now, we can get to our main logic:

- Iterate over i from 0 to $(size-1)$ of nums :
  - Search the existing BST for $\text{nums}[i] * 2 + 1$ and add the result to count
  - Insert $\text{nums}[i]$ to the BST, hence updating the count_ge of the previous nodes

C++

```
class Node {
public:
    Node *left, *right;
    int val;
    int count_ge;
    Node(int val)
    {
        this->val = val;
        this->count_ge = 1;
        this->left = NULL;
        this->right = NULL;
    }
};

Node* insert(Node* head, int val)
{
    if (head == NULL)
        return new Node(val);
    else if (val == head->val)
        head->count_ge++;
    else if (val < head->val)
        head->left = insert(head->left, val);
    else {
        head->count_ge++;
        head->right = insert(head->right, val);
    }
    return head;
```

Complexity analysis

- Time complexity: $O(n^2)$
  - The best case complexity for BST is $O(\log n)$ for search as well as insertion, wherein, the tree formed is complete binary tree
  - Whereas, in case like [1,2,3,4,5,6,7,8,...], insertion as well as search for an element becomes $O(n)$ in time, since, the tree is skewed in only one direction, and hence, is no better than the array
  - So, in worst case, for searching and insertion over n items, the complexity is $O(n * n)$
- Space complexity: $O(n)$ extra space for storing the BST in Node class.

Approach 3: BIT

Intuition

The problem with BST is that the tree can be skewed hence, making it $O(n^2)$ in complexity. So, need a data structure that remains balanced. We could either use a Red-black or AVL tree to make a balanced BST, but the implementation would be an overkill for the solution. We can use BIT (Binary Indexed Tree, also called Fenwick Tree) to ensure that the complexity is $O(n \log n)$ with only 12-15 lines of code.

BIT Overview:

Fenwick Tree or BIT provides a way to represent an array of numbers in an array(can be visualized as tree), allowing prefix/suffix sums to be calculated efficiently($O(\log n)$). BIT allows to update an element in $O(\log n)$ time.

We recommend having a look at BIT from the following link before getting into details:

- https://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/

So, BIT is very useful to accumulate information from front/back and hence, we can use it in the same way we used BST to get the count of elements that are greater than or equal to $2 * \text{nums}[i] + 1$ in the existing tree and then adding the current element to the tree.

Algorithm

First, lets review the BIT query and update routines of BIT. According to the convention, query routine goes from index to 0, i.e., $\text{BIT}[i]$ gives the sum for the range $[0, index]$, and update updates the values from current index to the end of array. But, since, we require to find the numbers greater than the given index, as and when we update an index, we update all the ancestors of the node in the tree, and for search, we go from the node to the end.

The modified update algorithm is:

```
update(BIT, index, val):
  while(index > 0):
    BIT[index] += val
    index -= (index & (-index))
```

Herein, we find get the next index using: $\text{index -= index \& (-index)}$, which is essentially subtracting the rightmost 1 from the index binary representation. We update the previous indices since, if an element is greater than the index

And the modified query algorithm is:

```
query(BIT,index):
  sum=0
  while(index<BIT.size):
    sum+=BIT[index]
    index+=(index&(-index))
```

Herein, we find get the next index using: $\text{index += index \& (-index)}$. This gives the suffix sum from $index$ to the end.

So, the main idea is to count the number of elements greater than $2 * \text{nums[i]}$ in range $[0, i)$ as we iterate from 0 to size-1. The steps are as follows:

- Create a copy of nums, say nums_copy ans sort nums_copy. This array is actually used for creating the Binary indexed tree
- Initialize $\text{count} = 0$ and BIT array of size $\text{size(nums)} + 1$ to store the BIT
- Iterate over i from 0 to $\text{size(nums)} - 1$:
  - Search the index of element not less than $2 * \text{nums[i]} + 1$ in nums_copy array. query the obtained index+1 in the BIT, and add the result to count
  - Search for the index of element not less than $nums[i]$ in nums_copy. We need to update the BIT for this index by 1. This essentially means that 1 is added to this index(or number of elements greater than this index is incremented). The effect of adding 1 to the index is passed to the ancestors as shown in update algorithm

C++

```
void update(vector<int>& BIT, int index, int val)
{
    while (index > 0) {
        BIT[index] += val;
        index -= index & (-index);
    }
}

int query(vector<int>& BIT, int index)
{
    int sum = 0;
    while (index < BIT.size()) {
        sum += BIT[index];
        index += index & (-index);
    }
    return sum;
}
int reversePairs(vector<int>& nums)
{
    int n = nums.size();
    vector<int> nums_copy(nums);

    sort(nums_copy.begin(), nums_copy.end());

    vector<int> BITS(n + 1, 0);
    int count = 0;
    for (int i = 0; i < n; i++) {
```

Complexity analysis

- Time complexity: $O(n \log n)$
  - In query and update operations, we see that the loop iterates at most the number of bits in index which can be at most n. Hence, the complexity of both the operations is $O(\log n)$(Number of bits in n is $\log n$)
  - The in-built operation lower_bound is binary search hence $O(\log n)$
  - We perform the operations for n elements, hence the total complexity is $O(n \log n)$
- Space complexity: $O(n)$. Additional space for BITS array

Approach 4: Modified Merge Sort

Intuition

In BIT and BST, we iterate over the array, dividing the array into 3 sections: already visited and hence added

In BIT and BST, we iterate over the array, dividing the array into 3 sections: already visited and hence added to the tree, current node and section to be visited. Another approach could be divide the problem into smaller subproblems, solving them and combining these problems to get the final result - Divide and conquer. We see that the problem has a great resemblance to the merge sort routine. The question is to find the inversions such that $\text{nums[i]} > 2 * \text{nums[j]}$ and i. So, we can easily modify the merge sort to count the inversions as required.

Mergesort

Mergesort is a divide-and-conquer based sorting technique that operates in $O(n \log n)$ time. The basic idea to divide the array into several sub-arrays until each sub-array is single element long and merging these sublists recursively that results in the final sorted array.

Algorithm

We define $\text{mergesort\_and\_count}$ routine that takes parameters an array say A and start and end indices:

- If start>=end this implies that elements can no longer be broken further and hence we return 0
- Otherwise, set $\text{mid} = (\text{start} + \text{end})/2$
- Store $count$ by recursively calling $\text{mergesort\_and\_count}$ on range [start,mid] and [mid+1,end] and adding the results. This is the divide step on our routine, breaking it into the 2 ranges, and finding the results for each range separately
- Now, we that we have separately calculated the results for ranges [start,mid] and [mid+1,end], but we still have to count the elements in [start,mid] that are greater than 2 * elements in [mid+1,end]. Count all such elements and add the result to count
- Finally, merge the array from start to end
  - Make 2 array : L from elements in range [start,mid] and R from elements in range R[mid+1,end]
  - Keep pointers i and j to L and R respectively both initialized to start to the arrays
  - Iterate over k from start to end and set A[k] to the smaller of L[i] or R[j] and increment the respective index

C++

```
void merge(vector<int>& A, int start, int mid, int end)
{
    int n1 = (mid - start + 1);
    int n2 = (end - mid);
    int L[n1], R[n2];
    for (int i = 0; i < n1; i++)
        L[i] = A[start + i];
    for (int j = 0; j < n2; j++)
        R[j] = A[mid + 1 + j];
    int i = 0, j = 0;
    for (int k = start; k <= end; k++) {
        if (j >= n2 || (i < n1 && L[i] <= R[j]))
            A[k] = L[i++];
        else
            A[k] = R[j++];
    }
}

int mergesort_and_count(vector<int>& A, int start, int end)
{
    if (start < end) {
        int mid = (start + end) / 2;
        int count = mergesort_and_count(A, start, mid) + mergesort_and_count(A, mid + 1, end);
        int j = mid + 1;
        for (int i = start; i <= mid; i++) {
            while (j <= end && A[i] > A[j] * 2LL)
                j++;
```

Complexity analysis

- Time complexity: $O(n \log n)$
  - In each step we divide the array into 2 sub-arrays, and hence, the maximum times we need to divide is equal to $O(\log n)$
  - Additional $O(n)$ work needs to be done to count the inversions and to merge the 2 sub-arrays after sorting. Hence total time complexity is $O(n \log n)$
- Space complexity: $O(n)$. Additional space for storing L and R arrays

Shoutout to @FUN4LEETCODE for the brilliant post!

Rate this article:

Previous · Next

Comments: 24 · Sort By

Type comment here... (Markdown is supported)

Preview · Post

by6 · 181 · July 25, 2018 4:53 PM
most poorly written solution/analysis I've seen on here
38 · Share · Reply
SHOW 1 REPLY

sp4658 · 33 · October 13, 2018 10:14 PM
I tried BST solution and merge sort in C++ it gives TLE.
23 · Share · Reply

Elenana · 15 · February 18, 2018 5:13 PM
Approach #2 - Time limit exceed for me
15 · Share · Reply

cee · 13 · September 28, 2018 5:13 AM
WTF about the `while (index < 0)` in the `update` function
9 · Share · Reply

bwv988 · 349 · October 1, 2018 10:57 AM
The idea of solution 3 is brilliant, though the explanation could be improved. The solution uses a modified version of BIT, which makes it kind of obscure. Here is my version sorts the array in a descendant order, which might be a little more intuitive. The item in the BIT is the count of numbers in the original array so far >= the according number in sorted array: when update, we increase the item count of all numbers <= current number; when search, we accumulate item count of all numbers > 2 *
Read More
6 · Share · Reply
SHOW 1 REPLY

phasme · 6 · December 3, 2018 7:10 PM
Stopped reading after solution 1 but it does not respect the rules of the problem.
Too bad.
5 · Share · Reply

eddiezhang · 8 · November 12, 2018 9:22 AM
Can anyone explain why in merge sort approach， the time complexity for counting the inversions is O(n). There are two loops and the worse case is O(n^2) from my point of view. Can anyone tell me what I am missing?
4 · Share · Reply
SHOW 1 REPLY

shashankprof05 · 4 · October 3, 2018 6:49 AM
For python, similar algorithms don't work. I guess the judge is too hard on python APIs.
3 · Share · Reply
SHOW 1 REPLY

pooyax · 35 · August 23, 2019 10:35 PM
In Approach 4: Modified Merge Sort, what does `count += j - (mid +1)` come from??
2 · Share · Reply

denis631 · 495 · January 8, 2019 1:43 PM
How can the second approach be accepted if `37 test case` is literally just a sorted array, which makes binary search tree a list in case you didn't implement a balanced BST ...
2 · Share · Reply