

561. Array Partition I

April 22, 2017 | 26.1K views

Previous

Next

★★★★★

Average Rating: 3.41 (22 votes)

Given an array of $2n$ integers, your task is to group these integers into n pairs of integer, say $(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n)$ which makes sum of $\min(a_i, b_i)$ for all i from 1 to n as large as possible.

Example 1:

Input: [1,4,3,2]

Output: 4

Explanation: n is 2, and the maximum sum of pairs is $4 = \min(1, 2) + \min(3, 4)$.

Note:

- 1. n is a positive integer, which is in the range of $[1, 10000]$.
- 2. All the integers in the array will be in the range of $[-10000, 10000]$.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

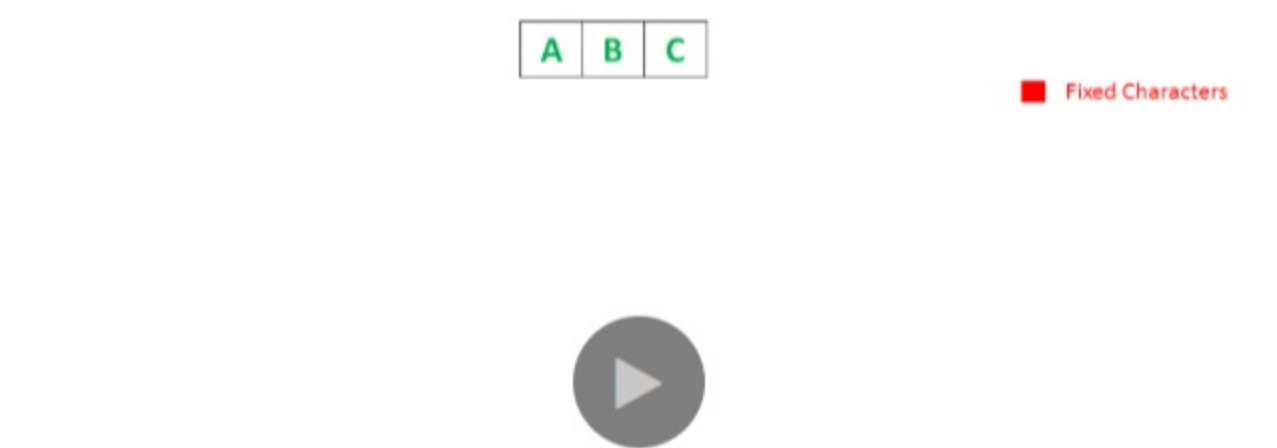
Algorithm

The simplest solution is to consider every possible set of pairings possible by using the elements of the `nums` array. For generating all the possible pairings, we make use of a function `permute(nums, current_index)`. This function creates all the possible permutations of the elements of the given array.

To do so, `permute` takes the index of the current element `current_index` as one of the arguments. Then, it swaps the current element with every other element in the array, lying towards its right, so as to generate a new ordering of the array elements. After the swapping has been done, it makes another call to `permute`, but this time with the index of the next element in the array. While returning back, we reverse the swapping done in the current function call.

Thus, when we reach the end of the array, a new ordering of the array's elements is generated. We consider the elements to be taken for the pairings such that the first element of every pair comes from the first half of the new array and the second element comes from the last half of the array. Thus, we sum up the minimum elements out of all these possible pairings and find out the maximum sum out of them.

The animation below depicts the ways the permutations are generated.



Java Copy

```
1 public class Solution {
2     int max_sum = Integer.MIN_VALUE;
3     public int arrayPairSum(int[] nums) {
4         permute(nums, 0);
5         return max_sum;
6     }
7     public void permute(int[] nums, int i) {
8         if (i == nums.length - 1) {
9             int sum = 0;
10            for (int j = 0; j < nums.length / 2; j++) {
11                sum += Math.min(nums[j], nums[nums.length / 2 + j]);
12            }
13            max_sum = Math.max(max_sum, sum);
14        }
15        for (int j = i + 1; j < nums.length; j++) {
16            swap(nums, i, j);
17            permute(nums, i + 1);
18            swap(nums, i, j);
19        }
20    }
21    public void swap(int[] nums, int x, int y) {
22        int temp = nums[x];
23        nums[x] = nums[y];
24        nums[y] = temp;
25    }
26 }
27 }
```

Complexity Analysis

- Time complexity: $O(n!)$. A total of $n!$ permutations are possible for n elements in the array.
- Space complexity: $O(1)$. Constant extra space is used.

Approach #2 Using Sorting [Accepted]

Algorithm

In order to understand this approach, let us look at the problem from a different perspective. We need to form the pairings of the array's elements such that the overall sum of the minimum out of such pairings is maximum. Thus, we can look at the operation of choosing the minimum out of the pairing, say (a, b) as incurring a loss of $a - b$ (if $a > b$), in the maximum sum possible.

The total sum will now be maximum if the overall loss incurred from such pairings is minimized. This minimization of loss in every pairing is possible only if the numbers chosen for the pairings lie closer to each other than to the other elements of the array.

Taking this into consideration, we can sort the elements of the given array and form the pairings of the elements directly in the sorted order. This will lead to the pairings of elements with minimum difference between them leading to the maximization of the required sum.

Java Copy

```
1 public class Solution {
2     public int arrayPairSum(int[] nums) {
3         Arrays.sort(nums);
4         int sum = 0;
5         for (int i = 0; i < nums.length; i += 2) {
6             sum += nums[i];
7         }
8         return sum;
9     }
10 }
```

Complexity Analysis

- Time complexity: $O(n \log n)$. Sorting takes $O(n \log n)$ time. We iterate over the array only once.
- Space complexity: $O(1)$. Constant extra space is used.

Approach #3 Using Extra Array [Accepted]

Algorithm

This approach is somewhat related to the sorting approach. Since the range of elements in the given array is limited, we can make use of a hashmap `arr`, such that `arr[i]` stores the frequency of occurrence of $(i - 10000)^{\text{th}}$ element. This subtraction is done so as to be able to map the numbers in the range $-10000 \leq i \leq 10000$ onto the hashmap.

Thus, now instead of sorting the array's elements, we can directly traverse the hashmap in an ascending order. But, any element could also occur multiple times in the given array. We need to take this factor into account.

For this, consider an example: `nums: [a, b, a, b, b, a]`. The sorted order of this array will be `nums_sorted: [a, a, a, b, b, b]`. (We aren't actually sorting the array in this approach, but the sorted array is taken just for demonstration). From the previous approach, we know that the required set of pairings is $(a, a), (a, b), (b, b)$. Now, we can see that while choosing the minimum elements, a will be chosen twice and b will be chosen once only. This happens because the number of a 's to be chosen has already been determined by the frequency of a , leaving the rest of the places to be filled by b . This is because, for the correct result we need to consider the elements in the ascending order. Thus, the lower number always gets priority to be added to the end result.

But, if the sorted elements take the form: `nums_sorted: [a, a, b, b, b, b]`, the correct pairing will be $(a, a), (b, b), (b, b)$. Again, in this case the number of a 's chosen is already predetermined, but since the number of a 's is odd, it doesn't impact the choice of b in the final sum.

Thus, based on the above discussion, we traverse the hashmap `arr`. If the current element is occurring `freq` number of times, and one of the elements is left to be paired with other elements in the right region (considering a virtual sorted array), we consider the current element $\lfloor \frac{freq}{2} \rfloor$ number of times and the next element occurring in the array $\lfloor \frac{freq}{2} \rfloor$ number of times for the final sum. To propagate the impact of this left over chosen number, we make use of a flag `d`. This flag is set to 1 if there is a leftover element from the current set which will be considered one more time. The same extra element already considered is taken into account while choosing an element from the next set.

While traversing the hashmap, we determine the correct number of times each element needs to be considered as discussed above. Note that the flag `d` and the `sum` remains unchanged if the current element of the hashmap doesn't exist in the array.

Below code is inspired by @fallcreek

Java Copy

```
1 public class Solution {
2     public int arrayPairSum(int[] nums) {
3         int[] arr = new int[20001];
4         int lim = 10000;
5         for (int num : nums) {
6             arr[num + lim]++;
7         }
8         int d = 0, sum = 0;
9         for (int i = -10000; i <= 10000; i++) {
10            sum += (arr[i + lim] + 1 - d) / 2 * i;
11            d = (2 + arr[i + lim] - d) % 2;
12        }
13        return sum;
14    }
15 }
```

Complexity Analysis

- Time complexity: $O(n)$. The whole hashmap `arr` of size n is traversed only once.
- Space complexity: $O(n)$. A hashmap `arr` of size n is used.


Rate this article: ★★★★★

Previous

Next


Comments: 31

Sort By ▾



Type comment here... (Markdown is supported)


PreviewPost



Karma_Shunya ★ 30 · June 14, 2019 11:07 AM

This article is poorly written, has typos and needs updation.

24 · Share · Reply




bomboneiro ★ 61 · September 15, 2018 2:51 PM

why are time and space complexity in the last solution $O(n)$? independent of the size of the input array, we create an array of size 20000. Are space and time complexities not equal to $O(m)$, where m is the size of the array we create?

10 · Share · Reply


SHOW 2 REPLIES



XiaojingHu ★ 8 · August 2, 2018 4:43 AM

It can be seen as greedy algorithm.

3 · Share · Reply




donggua_fu ★ 187 · April 23, 2017 1:49 PM

I think so--the sorting way is accepted.

And the second method is actually counting sort. Am I right?

2 · Share · Reply




treemantan ★ 27 · October 23, 2019 1:05 AM

With built-in sort, already quick enough

def arrayPairSum(self, nums):
 """
 :type nums: List[int]
 """

Read More

1 · Share · Reply




reyou ★ 133 · May 7, 2019 6:24 AM

How come "The simplest solution is to consider every possible set of pairings possible" can be a suggestion at all?

It is far more most complex and slowest one if you compare to others.

1 · Share · Reply

SHOW 1 REPLY




yukinoshita123 ★ -3 · September 8, 2017 12:23 PM

class Solution {
 public int arrayPairSum(int[] nums) {
 int len=nums.length;
 Arrays.sort(nums);
 int sum=0;
 }
}

Read More

1 · Share · Reply




kobe2417 ★ 0 · September 9, 2017 8:59 AM

class Solution {
 public:
 int arrayPairSum(vector& nums) {
 sort(nums.begin(),nums.end());
 int mins=0,output=0;
 }
}

Read More

0 · Share · Reply




prakhara_avasthi ★ 0 · September 3, 2017 3:19 AM

Solution in C

/*
 * main.c
 */

Read More

0 · Share · Reply



gsamba ★ 5 · September 1, 2017 8:03 AM

class Solution {
 public int arrayPairSum(int[] nums) {
 Arrays.sort(nums);
 int sum = 0;
 for(int i=0;i<nums.length;i+=2) {

Read More

0 · Share · Reply

< 1 2 3 4 >