Previous Next

453. Minimum Moves to Equal Array Elements

Jan. 5, 2017 | 14.9K views

★★★★★
Average Rating: 4.40 (20 votes)

Given a **non-empty** integer array of size *n*, find the minimum number of moves required to make all array elements equal, where a move is incrementing *n* - 1 elements by 1.

Example:

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

Output:
3

Explanation:
Only three moves are needed (remember each move increments two elements):

[1,2,3]  =>  [2,3,3]  =>  [3,4,3]  =>  [4,4,4]
```

Solution

Approach #1 Brute Force [Time Limit Exceeded]

Firstly, we know that in order to make all the elements equal to each other with minimum moves, we need to do the increments in all but the maximum element of the array. Thus, in the brute force approach, we scan the complete array to find the maximum and the minimum element. After this, we add 1 to all the elements except the maximum element, and increment the count for the number of moves done. Again, we repeat the same process, and this continues until the maximum and the minimum element become equal to each other.

Java Copy

```java
public class Solution {
    public int minMoves(int[] nums) {
        int min = 0, max = nums.length - 1, count = 0;
        while (true) {
            for (int i = 0; i < nums.length; i++) {
                if (nums[max] < nums[i]) {
                    max = i;
                }
                if (nums[min] > nums[i]) {
                    min = i;
                }
            }
            if (nums[max] == nums[min]) {
                break;
            }
            for (int i = 0; i < nums.length; i++) {
                if (i != max) {
                    nums[i]++;
                }
            }
            count++;
        }
        return count;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2k)$, where n is the length of the array and k is the difference between maximum element and minimum element.
- Space complexity : $O(1)$. No extra space required.

Approach #2 Better Brute Force[Time Limit Exceeded]

Algorithm

In the previous approach, we added 1 to every element in a single step. But, we can modify this approach to some extent. In order to make the minimum element equal to the maximum element, we need to add 1 atleast k times, after which, the maximum element could change. Thus, instead of incrementing in steps of 1,we increment in bursts, where each burst will be of size $k = max - min$. Thus, we scan the complete array to find the maximum and minimum element. Then, we increment every element by k units and add k to the count of moves. Again we repeat the same process, until the maximum and minimum element become equal.

Java Copy

```java
public class Solution {
    public int minMoves(int[] nums) {
        int min = 0, max = nums.length - 1, count = 0;
        while (true) {
            for (int i = 0; i < nums.length; i++) {
                if (nums[max] < nums[i]) {
                    max = i;
                }
                if (nums[min] > nums[i]) {
                    min = i;
                }
            }
            int diff = nums[max] - nums[min];
            if (diff == 0) {
                break;
            }
            count += diff;
            for (int i = 0; i < nums.length; i++) {
                if (i != max) {
                    nums[i] = nums[i] + diff;
                }
            }
        }
        return count;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. In every iteration two elements are equalized.
- Space complexity : $O(1)$. No extra space required.

Approach #3 Using Sorting [Accepted]

Algorithm

The problem gets simplified if we sort the given array in order to obtain a sorted array a. Now, similar to Approach 2,we use the difference $diff = max - min$ to update the elements of the array, but we need not traverse the whole array to find the maximum and minimum element every time, since if the array is sorted, we can make use of this property to find the maximum and minimum element after updation in $O(1)$ time. Further, we need not actually update all the elements of the array. To understand how this works, we'll go in a stepwise manner.

Firstly, assume that we are updating the elements of the sorted array after every step of calculating the difference $diff$. We'll see how to find the maximum and minimum element without traversing the array. In the first step, the last element is the largest element. Therefore, $diff = a[n-1] - a[0]$. We add $diff$ to all the elements except the last one i.e. $a[n-1]$. Now, the updated element at index 0 ,$a'[0]$ will be $a[0] + diff = a[n-1]$. Thus, the smallest element $a'[0]$ is now equal to the previous largest element $a[n-1]$. Since, the elements of the array are sorted, the elements upto index $i-2$ satisfy the property $a[j] >= a[j-1]$. Thus, after updation, the element $a'[n-2]$ will become the largest element, which is obvious due to the sorted array property. Also, a[0] is still the smallest element.

Thus, for the second updation, we consider the difference $diff$ as $diff = a[n-2] - a[0]$. After updation, $a''[0]$ will become equal to $a'[n-2]$ similar to the first iteration. Further, since $a'[0]$ and $a'[n-1]$ were equal. After the second updation, we get $a''[0] = a''[n-1] = a'[n-2]$. Thus, now the largest element will be $a[n-3]$. Thus, we can continue in this fashion, and keep on incrementing the number of moves with the difference found at every step.

Now, let's come to step 2. In the first step, we assumed that we are updating the elements of the array a at every step, but we need not do this. This is because, even after updating the elements the difference which we consider to add to the number of moves required remains the same because both the elements max and min required to find the $diff$ get updated by the same amount everytime.

Thus, we can simply sort the given array once and use $moves = \sum_{i=1}^{n-1}(a[i] - a[0])$.

Java Copy

```java
public class Solution {
    public int minMoves(int[] nums) {
        Arrays.sort(nums);
        int count = 0;
        for (int i = nums.length - 1; i > 0; i--) {
            count += nums[i] - nums[0];
        }
        return count;
    }
}
```

Complexity Analysis

- Time complexity : $O\big(nlog(n)\big)$. Sorting will take $O\big(nlog(n)\big)$ time.
- Space complexity : $O(1)$. No extra space required.

Approach #4 Using DP [Accepted]

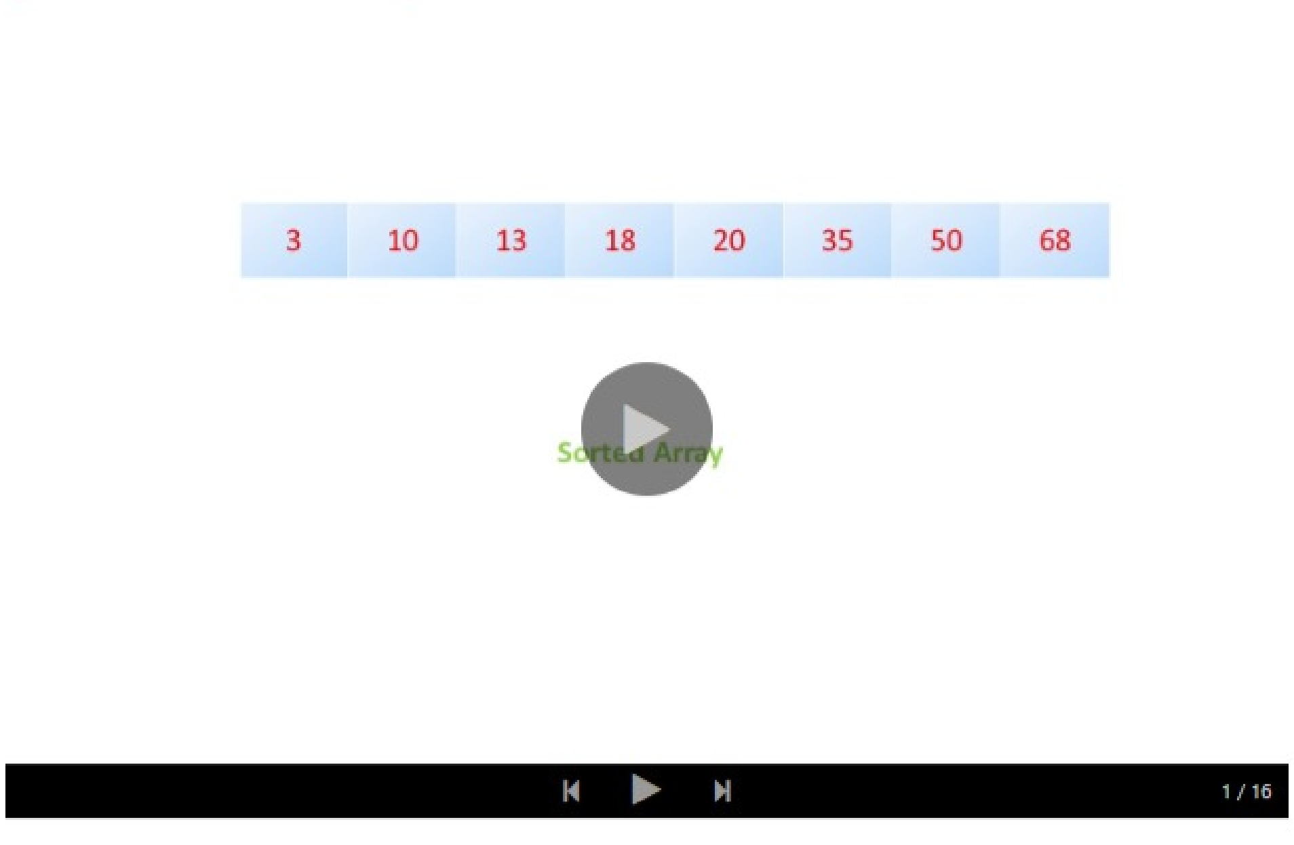
Algorithm

The given problem can be simplified if we sort the given array once. If we consider a sorted array a, instead of trying to work on the complete problem of equalizing every element of the array, we can break the problem for array of size n into problems of solving arrays of smaller sizes. Assuming, the elements upto index $i-1$ have been equalized, we can simply consider the element at index i and add the difference $diff = a[i] - a[i-1]$ to the total number of moves for the array upto index i to be equalized i.e. $moves = moves + diff$. But when we try to proceed with this step, as per a valid move, the elements following $a[i]$ will also be incremented by the amount $diff$ i.e. $a[j] = a[j] + diff$, for $j > i$. But while implementing this approach, we need not increment all such $a[j]$'s. Instead, we'll add the number of $moves$ done so far to the current element i.e. $a[i]$ and update it to $a'[i] = a[i] + moves$.

In short, we sort the given array, and keep on updating the $moves$ required so far in order to equalize the elements upto the current index without actually changing the elements of the array except the current element. After the complete array has been scanned $moves$ gives the required solution.

The following animation will make the process more clear for this example:

`[13 18 3 10 35 68 50 20 50]`



Java Copy

```java
public class Solution {
    public int minMoves(int[] nums) {
        Arrays.sort(nums);
        int moves = 0;
        for (int i = 1; i < nums.length; i++) {
            int diff = (moves + nums[i]) - nums[i - 1];
            nums[i] += moves;
            moves += diff;
        }
        return moves;
    }
}
```

Complexity Analysis

- Time complexity : $O\big(nlog(n)\big)$. Sorting will take $O\big(nlog(n)\big)$ time.
- Space complexity : $O(1)$. Only single extra variable is used.

Approach #5 Using Math[Accepted]

Algorithm

This approach is based on the idea that adding 1 to all the elements except one is equivalent to decrementing 1 from a single element, since we are interested in the relative levels of the elements which need to be equalized. Thus, the problem is simplified to find the number of decrement operations required to equalize all the elements of the given array. For finding this, it is obvious that we'll reduce all the elements of the array to the minimum element. But, in order to find the solution, we need not actually decrement the elements. We can find the number of moves required as $moves = \sum_{i=0}^{n-1} a[i] - min(a) * n$, where n is the length of the array.

Java Copy

```java
public class Solution {
    public int minMoves(int[] nums) {
        int moves = 0, min = Integer.MAX_VALUE;
        for (int i = 0; i < nums.length; i++) {
            moves += nums[i];
            min = Math.min(min, nums[i]);
        }
        return moves - min * nums.length;
    }
}

```

Complexity Analysis

- Time complexity : $O(n)$. We traverse the complete array once.
- Space complexity : $O(1)$. No extra space required.

Approach #6 Modified Approach Using Maths[Accepted]

Algorithm

There could be a problem with the above approach. The value $\sum_{i=0}^{n-1} a[i]$ could be very large and hence could lead to integer overflow if the $a[i]$'s are very large. To avoid this problem, we can calculate the required number of $moves$ on the fly. $\sum_{i=0}^{n-1}(a[i] - min(a))$.

Java Copy

```java
public class Solution {
    public int minMoves(int[] nums) {
        int moves = 0, min = Integer.MAX_VALUE;
        for (int i = 0; i < nums.length; i++) {
            min = Math.min(min, nums[i]);
        }
        for (int i = 0; i < nums.length; i++) {
            moves += nums[i] - min;
        }
        return moves;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. One pass for finding minimum and one pass for calculating moves.
- Space complexity : $O(1)$. No extra space required.

Rate this article: ★★★★★

Previous Next

Comments: 23 Sort By

Type comment here... (Markdown is supported)

Preview Post

anton4 ★ 671 January 27, 2017 1:39 AM

Are you guys sure this is an easy problem??

75 Share Reply

SHOW 1 REPLY

panahi ★ 300 July 14, 2018 6:58 AM

The brute force approaches are easy but not accepted so it should be considered medium at least!

41 Share Reply

zoran.lovric ★ 43 March 13, 2018 7:19 AM

lol at this being marked as easy

14 Share Reply

lishichengyan ★ 232 November 7, 2019 9:17 AM

Let's explain it in a clear and short manner:
suppose there're `k` elements, the sum of original array is `s`, the minimum move is `m`, eventually all the elements become `e`, we know each move contributes `(k-1)` to the sum, so we have:
`s + (k-1)*m = k*e`
for the minimum element `min`, it must be added `m` times, i.e.

Read More

8 Share Reply

SHOW 2 REPLIES

Mr-Bin ★ 129 June 8, 2017 2:50 PM

How to really understand this? "adding 1 to all the elements except one is equivalent to decrementing 1 from a single element,"

3 Share Reply

SHOW 2 REPLIES

s961206 ★ 751 September 8, 2019 10:06 PM

Can't understand how method 2 work? Maybe it needs more proof?

1 Share Reply

zcc0328 ★ 1 June 19, 2018 9:47 PM

i have a question for method #2. Though it works fine, but , in some cases,before min reachs max, the max could be changed to another element.
For example, [1,4,5] . the method could always get the right result but i really don't understand the logic behind it. could anyone help with this?

1 Share Reply

lstop1215 ★ 1 July 28, 2017 2:23 PM

Find the smallest element in the array and calculate the sum of the difference from the other

1 Share Reply

Ltdan ★ 2 March 1, 2017 5:53 AM

Why isn't the first solution O((2N)^K instead of O(n^2k)? Seems like you loop through the array twice, but its not nested, so that would be 2N, and you do that K times.

1 Share Reply

SHOW 1 REPLY

mdorgham ★ 0 April 2, 2017 3:10 AM

I think we still need to make the moves variable long to avoid possible overflow .. consider this test case: [1, 5 ,2147483647] .

0 Share Reply

‹ 1 2 3 ›