548. Split Array With Equal Sum

April 1, 2017 | 8.2K views
Average Rating: 4.80 (5 votes)

Given an array with n integers, you need to find if there are triplets (i, j, k) which satisfies following conditions:

1. 0 < i, i + 1 < j, j + 1 < k < n - 1
2. Sum of subarrays (0, i - 1), (i + 1, j - 1), (j + 1, k - 1) and (k + 1, n - 1) should be equal.

where we define that subarray (L, R) represents a slice of the original array starting from the element indexed L to the element indexed R.

Example:

```
Input: [1,2,1,2,1,2,1]
Output: True
Explanation:
i = 1, j = 3, k = 5.
sum(0, i - 1) = sum(0, 0) = 1
sum(i + 1, j - 1) = sum(2, 2) = 1
sum(j + 1, k - 1) = sum(4, 4) = 1
sum(k + 1, n - 1) = sum(6, 6) = 1
```

Note:

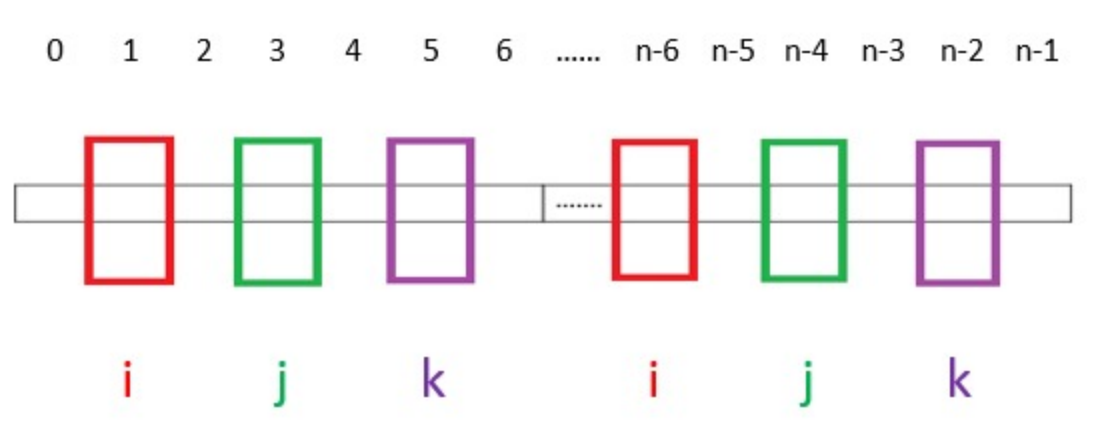
1. 1 <= n <= 2000.
2. Elements in the given array will be in range [-1,000,000, 1,000,000].

Solution

Approach #1 Brute Force [Time Limit Exceeded]

Algorithm

Before we start looking at any of the approaches for solving this problem, firstly we need to look at the limits imposed on i, j and k by the given set of constraints. The figure below shows the maximum and minimum values that i, j and k can assume.



Thus, the limits based on the length of the array n can now be rewritten as:

$1 \leq i \leq n-6$

$i+2 \leq j \leq n-4$

$j+2 \leq k \leq n-2$

Having discussed the limits imposed on the cuts i, j, k that we will apply on the given array $nums$, let's look at the first solution that comes to our mind.

We simply traverse over all the elements of the array. We consider all the possible subarrays taking care of the constraints imposed on the cuts, and check if any such cuts exist which satisfy the given equal sum quadruples criteria.

Java Copy

```java

public class Solution {

    public int sum(int[] nums, int l, int r) {
        int summ = 0;
        for (int i = l; i < r; i++)
            summ += nums[i];
        return summ;
    }

    public boolean splitArray(int[] nums) {
        if (nums.length < 7)
            return false;
        for (int i = 1; i < nums.length - 5; i++) {
            int sum1 = sum(nums, 0, i);
            for (int j = i + 2; j < nums.length - 3; j++) {
                int sum2 = sum(nums, i + 1, j);
                for (int k = j + 2; k < nums.length - 1; k++) {
                    int sum3 = sum(nums, j + 1, k);
                    int sum4 = sum(nums, k + 1, nums.length);
                    if (sum1 == sum2 && sum3 == sum4 && sum2 == sum4)
                        return true;
                }
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^4)$. Four for loops inside each other each with a worst case run of length n.
- Space complexity : $O(1)$. Constant Space required.

Approach #2 Cumulative Sum [Time Limit Exceeded]

Algorithm

In the brute force approach, we traversed over the subarrays for every triplet of cuts considered. Rather than doing this, we can save some calculation effort if we make use of a cumulative sum array sum, where $sum[i]$ stores the cumulative sum of the array $nums$ upto the i^{th} element. Thus, now in order to find the $sum(subarray(i:j))$, we can simply use $sum[j] - sum[i]$. Rest of the process remains the same.

Java Copy

```java
public class Solution {
    public boolean splitArray(int[] nums) {
        if (nums.length < 7)
            return false;
        int[] sum = new int[nums.length];
        sum[0] = nums[0];
        for (int i = 1; i < nums.length; i++) {
            sum[i] = sum[i - 1] + nums[i];
        }
        for (int i = 1; i < nums.length - 5; i++) {
            int sum1 = sum[i - 1];
            for (int j = i + 2; j < nums.length - 3; j++) {
                int sum2 = sum[j - 1] - sum[i];
                for (int k = j + 2; k < nums.length - 1; k++) {
                    int sum3 = sum[k - 1] - sum[j];
                    int sum4 = sum[nums.length - 1] - sum[k];
                    if (sum1 == sum2 && sum3 == sum4 && sum2 == sum4)
                        return true;
                }
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^3)$. Three for loops are there, one within the other.
- Space complexity : $O(n)$. sum array of size n is used for storing cumulative sum.

Approach #3 Slightly Better Approach [Time Limit Exceeded]

Algorithm

We can improve the previous implementation to some extent if we stop checking for further quadruples if the first and second parts formed till now don't have equal sums. This idea is used in the current implementation.

Java Copy

```java
public class Solution {
    public boolean splitArray(int[] nums) {
        if (nums.length < 7)
            return false;

        int[] sum = new int[nums.length];
        sum[0] = nums[0];
        for (int i = 1; i < nums.length; i++) {
            sum[i] = sum[i - 1] + nums[i];
        }
        for (int i = 1; i < nums.length - 5; i++) {
            int sum1 = sum[i - 1];
            for (int j = i + 2; j < nums.length - 3; j++) {
                int sum2 = sum[j - 1] - sum[i];
                if (sum1 != sum2)
                    continue;
                for (int k = j + 2; k < nums.length - 1; k++) {
                    int sum3 = sum[k - 1] - sum[j];
                    int sum4 = sum[nums.length - 1] - sum[k];
                    if (sum3 == sum4 && sum2 == sum4)
                        return true;
                }
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^3)$. Three loops are there.
- Space complexity : $O(n)$. sum array of size n is used for storing the cumulative sum.

Approach #4 Using HashMap [Time limit Exceeded]

Algorithm

In this approach, we create a data structure called map which is simply a HashMap, with data arranged in the format:

$\{csum(i) : [i_1, i_2, i_3, ...]\}$, here $csum(i)$ represents the cumulative sum in the given array $nums$ upto the i^{th} index and its corresponding value represents indices upto which cumulative sum=csum(i).

Once we create this map, the solutions gets simplified a lot. Consider only the first two cuts formed by i and j. Then, the cumulative sum upto the $(j-1)^{th}$ index will be given by: $csum(j-1) = sum(part1) + nums[i] + sum(part2)$. Now, if we want the first two parts to have the same sum, the same cumulative sum can be rewritten as:

$csum'(j-1) = csum(i-1) + nums[i] + csum(i-1) = 2csum(i-1) + nums[i]$.

Thus, we traverse over the given array, changing the value of the index i forming the first cut, and look if the map formed initially contains a cumulative sum equal to $csum'(j-1)$. If map contains such a cumulative sum, we consider every possible index j satisfying the given constraints and look for the equalities of the first cumulative sum with the third and the fourth parts.

Following the similar lines as the discussion above, the cumulative sum upto the third cut by k^{th} index is given by

$csum(k-1) = sum(part1) + nums[i] + sum(part2) + nums[j] + sum(part3)$.

For equality of sum, the condition becomes:

$csum'(k-1) = 3 * csum(i-1) + nums[i] + nums[j]$.

Similarly, the cumulative sum upto the last index becomes:

$csum(end) = sum(part1) + nums[i] + sum(part2) + nums[j] + sum(part3) + nums[k] + sum(part4)$.

Again, for equality, the condition becomes:

$csum'(end) = 4 * csum(i-1) + nums[i] + nums[j] + nums[k]$.

For every cut chosen, we look if the required cumulative sum exists in map. Thus, we need not calculate sums again and again or traverse over the array for all the triplets (i, j, k) possible. Rather, now, we directly know, what cumulative sum to look for in the map, which reduces a lot of computations.

Java Copy

```java
public class Solution {
    public boolean splitArray(int[] nums) {
        HashMap < Integer, ArrayList < Integer >> map = new HashMap < > ();
        int summ = 0, tot = 0;
        for (int i = 0; i < nums.length; i++) {
            summ += nums[i];
            if (map.containsKey(summ))
                map.get(summ).add(i);
            else {
                map.put(summ, new ArrayList < Integer > ());
                map.get(summ).add(i);
            }
            tot += nums[i];
        }
        summ = nums[0];
        for (int i = 1; i < nums.length - 5; i++) {
            if (map.containsKey(2 * summ + nums[i])) {
                for (int j: map.get(2 * summ + nums[i])) {
                    j++;
                    if (j > i + 1 && j < nums.length - 3 && map.containsKey(3 * summ + nums[i] + nums[j]))
{
                        for (int k: map.get(3 * summ + nums[j] + nums[i])) {
                            k++;
                            if (k < nums.length - 1 && k > j + 1 && 4 * summ + nums[i] + nums[j] + nums[k]
== tot)
                                return true;
                        }
```

Complexity Analysis

- Time complexity : $O(n^3)$. Three nested loops are there and every loop runs n times in the worst case. Consider the worstcase [0,0,0....,1,1,1,1,1,1,1].
- Space complexity : $O(n)$. HashMap size can go upto n.

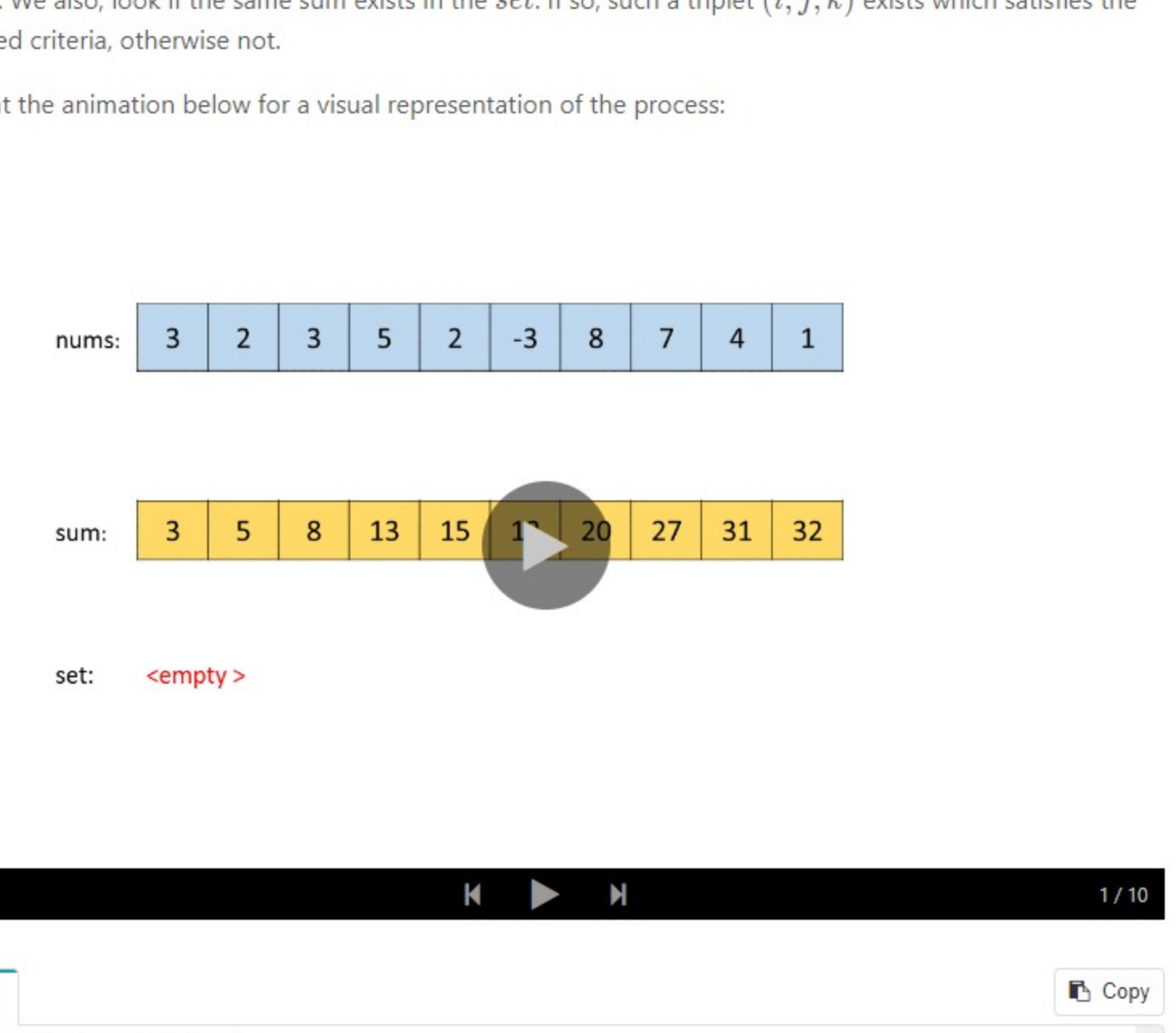
Approach #5 Using Cumulative Sum and HashSet [Accepted]

Algorithm

In this approach, firstly we form a cumulative sum array sum, where $sum[i]$ stores the cumulative sum of the array $nums$ upto the i^{th} index. Then, we start by traversing over the possible positions for the middle cut formed by j. For every j, firstly, we find all the left cut's positions, i, that lead to equalizing the sum of the first and the second part (i.e. $sum[i-1] = sum[j-1] - sum[i]$) and store such sums in the set (a new HashSet is formed for every j chosen). Thus, the presence of a sum in set implies that such a sum is possible for having equal sum of the first and second part for the current position of the middle cut(j).

Then, we go for the right cut and find the position of the right cut that leads to equal sum of the third and the fourth part ($sum[n-1] - sum[k] = sum[k-1] - sum[j]$), for the same middle cut as chosen earlier. We also, look if the same sum exists in the set. If so, such a triplet (i, j, k) exists which satisfies the required criteria, otherwise not.

Look at the animation below for a visual representation of the process:



Java Copy

```java
public class Solution {
    public boolean splitArray(int[] nums) {
        if (nums.length < 7)
            return false;
        int[] sum = new int[nums.length];
        sum[0] = nums[0];
        for (int i = 1; i < nums.length; i++) {
            sum[i] = sum[i - 1] + nums[i];
        }
        for (int j = 3; j < nums.length - 3; j++) {
            HashSet < Integer > set = new HashSet < > ();
            for (int i = 1; i < j - 1; i++) {
                if (sum[i - 1] == sum[j - 1] - sum[i])
                    set.add(sum[i - 1]);
            }
            for (int k = j + 2; k < nums.length - 1; k++) {
                if (sum[nums.length - 1] - sum[k] == sum[k - 1] - sum[j] && set.contains(sum[k - 1] -
sum[j]))
                    return true;
            }
        }
        return false;
    }
```

Complexity Analysis

- Time complexity : $O(n^2)$. One outer loop and two inner loops are used.
- Space complexity : $O(n)$. HashSet size can go upto n.

Analysis written by: @vinod23

Rate this article:

Previous Next

Comments: 13 Sort By

Type comment here... (Markdown is supported)

Preview Post

Three_Thousand_world ★ 1146 March 10, 2019 8:37 AM
WHY
13 Share Reply

Mr-Bin ★ 123 June 7, 2017 5:15 PM
You miss the ending ')' in this line: if (sum[nums.length - 1] - sum[k] == sum[k - 1] - sum[j] && set.contains(sum[k - 1] - sum[j]))
0 Share Reply

vinod23 ★ 425 April 25, 2017 1:59 AM
@aruballos Thanks for pointing it out. I have updated the animation. Please take a look.
0 Share Reply

aruballos ★ 3 April 24, 2017 8:31 AM
I believe the animation might be wrong? See the screenshot below:
Read More
0 Share Reply

xhd2015 ★ 9 April 15, 2017 8:54 PM
No, I think time complexity is O(n^2log(n)) , because set.add() will cost log(n)
0 Share Reply
SHOW 2 REPLIES

sean46 ★ 91 April 10, 2017 4:44 AM
what's the complexity of the DFS solution?
0 Share Reply

vinod23 ★ 425 April 9, 2017 2:07 AM
DFS approach is getting accepted because of some weak test cases.
0 Share Reply

tedhow ★ 0 April 7, 2017 12:15 PM
To find positions of i,j,k in the array, just need to select i first, record the sum of first i-1 elements as testSum, use the testSum to search for j in the subarray [i:], if any then k in the subarray [j:]. If j, k are found, done. Otherwise select another i until all possible positions for i have been tested.
0 Share Reply

lee215 ★ 47126 April 4, 2017 1:58 PM
Solution O(N^2)

```
def splitArray(self, nums):
        n = len(nums)
```

Read More
0 Share Reply

LJerRRy ★ 34 April 3, 2017 7:49 PM
using DFS can also be accepted

```
public class Solution {
    public boolean splitArray(int[] nums) {
```

Read More
0 Share Reply

‹ 1 2 ›