

# ARRAY

# Arrays

- **Memory layout:** hold values in a **contiguous** block of memory.
- **Fixed Size:** the size of an array is defined when it is created and cannot be changed.  
However, high-level languages have different implementations, making it dynamic.
- **Homogeneous elements:** all elements are of the same data type (int, float, char...)
- **Efficiency:** accessing elements by index is very efficient  $O(1)$ , since each index maps directly to a memory location. Also, range scans benefit from CPU cache lines since arrays are stored in contiguous blocks of memory.

# Problem – 1. Two Sum

Easy



LeetCode

[leetcode.com/problems/two-sum](https://leetcode.com/problems/two-sum)

## Problem

- Given an **array** of numbers and a **target**, example: **array** [2,7,11,15] and **target** 9
- Return indices of two numbers where they add up to **target**
- **Output:** [0,1]

$\text{array}[0] + \text{array}[1] = 2 + 7 = 9$

# Solution – 1. Two Sum

Easy



LeetCode

[leetcode.com/problems/two-sum](https://leetcode.com/problems/two-sum)

## Solution

- Iterative over each number in the array
- Calculate the difference between target and each number, example:  
 $\text{array}[0] = 2, \text{ target } 9, \text{ then } 9 - 2 = 7$
- Now we know we need the number **7** to sum up to **9**
- Check in a *hashmap* if we have 7 in some part of the array  
 $\text{hash}[7] \text{ exists?}$
- If yes, return the current index and the index of 7
- If not, store the index of the current number in the hashmap for future evaluation

$\text{hash}[2] = 0$

# Code – 1. Two Sum

Easy



LeetCode

[leetcode.com/problems/two-sum](https://leetcode.com/problems/two-sum)

## Code Time: $O(n)$ Space: $O(n)$

```
vector<int> twoSum(vector<int>& nums, int target) {
    std::unordered_map<int, int> numMap;
    // n being the size of nums
    for (int i = 0; i < nums.size(); i++) {
        // current number of the array
        int number = nums[i];
        int diff = target - number;

        // check if the difference is in some part of the array
        // by using a hashmap
        if (numMap.find(diff) != numMap.end()) {
            return { numMap[diff], i};
        }

        // register the current number index
        numMap[number] = i;
    }
    // no matches
    return {};
}
```

# Problem – 217. Contains Duplicate

Easy



LeetCode

[leetcode.com/problems/contains-duplicate](https://leetcode.com/problems/contains-duplicate)

## Problem

- You are given an array of numbers
- Return any value that appears at least twice

## Solution

- Loop through the array
- Check if the value is in a hash table
- Return **true** if the value exist
- The problem requires at least twice, but one modification may be having a specific count

# Code – 217. Contains Duplicate

Easy



LeetCode

[leetcode.com/problems/contains-duplicate](https://leetcode.com/problems/contains-duplicate)

**Code** Time:  $O(n)$  Space:  $O(n)$

```
bool containsDuplicate(vector<int>& nums) {
    unordered_map<int, int> seen;
    for (int i = 0; i < nums.size(); ++i) {
        if (seen[nums[i]] == 1) {
            return true;
        }
        seen[nums[i]]++;
    }
    return false;
}
```

**Another solution (less flexible)**

```
bool containsDuplicate(vector<int>& nums) {
    unordered_map<int, bool> seen;
    for (const auto& num : nums) {
        if (seen[num]) {
            return true;
        }
        seen[num] = true;
    }
    return false;
}
```

# Problem – 238. Product of Array Except Self

Medium



LeetCode

[leetcode.com/problems/product-of-array-except-self](https://leetcode.com/problems/product-of-array-except-self)

## Problem Statement

- You are given an integer array **nums**
- Return another array where each element is multiplied by all the elements except itself
- Example:

```
nums = [14,2,5,99]
```

```
nums[0] = 2 * 5 * 99 (all except 14)
```

```
nums[1] = 14 * 5 * 99 (all except 2)
```

```
nums[2] = 14 * 2 * 99
```

```
nums[3] = 14 * 2 * 5
```



# Solution – 238. Product of Array Except Self

Medium



LeetCode

[leetcode.com/problems/product-of-array-except-self](https://leetcode.com/problems/product-of-array-except-self)

## Solution

- Go over the array once and calculate the product of the left side. Example:

```
nums = [14,2,5]
```

```
left[0] = 1 (think of multiplying all elements before 14, so 1 because there is none)
```

```
left[1] = 14 (all elements from the left multiplied, except 2)
```

```
left[2] = 14 * 2 = 28 (all elements from the left multiplied, except 5)
```

```
left = [1, 14, 28]
```

- Using the same logic, do the same calculation but starting from the right

```
right[2] = 1 (no elements after 5)
```

```
right[1] = 5 (only 5 after 2)
```

```
right[0] = 2 * 5 = 10
```

```
right = [10, 5, 1]
```

- Multiply each element from left and right:

```
left ⊙ right = [1, 14, 28] ⊙ [10,5,1] = [10, 70, 28]
```

# Code – 238. Product of Array Except Self

Medium



LeetCode

[leetcode.com/problems/product-of-array-except-self](https://leetcode.com/problems/product-of-array-except-self)

**Code** Time:  $O(n)$  Space:  $O(n)$

```
vector<int> productExceptSelf(vector<int>& nums) {
    int n = nums.size();
    vector<int> output(n, 1);
    vector<int> right(n, 1);

    // calculate left first
    for (int i = 1; i < n; ++i) {
        output[i] = nums[i - 1] * output[i - 1];
    }
    // calculate right
    for (int i = n - 1; i >= 0; --i) {
        right[i] = nums[i + 1] * right[i + 1];
        output[i] = output[i] * right[i];
    }

    /* or you can save some space using this logic,
       although I don't find it as intuitive as the previous one
    int right = 1;
    for (int i = n - 1; i >= 0; --i) {
        output[i] *= right;
        right *= nums[i];
    }
    */

    return output;
}
```

# Code – 238. Product of Array Except Self

Medium



LeetCode

[leetcode.com/problems/product-of-array-except-self](https://leetcode.com/problems/product-of-array-except-self)

**Code (better)** Time:  $O(n)$  Space:  $O(n)$

```
vector<int> productExceptSelf(vector<int>& nums) {  
    vector<int> res;  
  
    int prefix = 1;  
    for (int i = 0; i < nums.size(); ++i) {  
        res.push_back(prefix);  
        prefix *= nums[i];  
    }  
  
    int suffix = 1;  
    for (int i = nums.size() - 1; i >= 0; --i) {  
        res[i] *= suffix;  
        suffix *= nums[i];  
    }  
  
    return res;  
}
```

# Problem – 53. Maximum Subarray

Medium



LeetCode

[leetcode.com/problems/maximum-subarray](https://leetcode.com/problems/maximum-subarray)

## Problem

- You are given an array `nums`
- Find the subarray with the largest sum

- **Example:**

`nums = [-2,1,-3,4,-1,2,1,-5,4]`

`output = 6`

The subarray `[4, -1, 2, 1]` has the largest sum 6.

# Problem – 53. Maximum Subarray

Medium



LeetCode

[leetcode.com/problems/maximum-subarray](https://leetcode.com/problems/maximum-subarray)

## Solution

- Use Kadane's algorithm to find the maximum sum of a contiguous subarray in linear time
- Core idea:  
at each index, either:
  1. start a new subarray at **nums[i]** or
  2. extend the current one by adding **nums[i]**

# Arrays – Kadane's algorithm

- Kadane's algorithm is a dynamic programming algorithm to solve **maximum subarray sum**
- At every **index i**:  
start a new subarray at **i**  
extend the previous subarray to include **array[i]**

- **Algorithm**

## 1. Initialize:

```
int maxSoFar = array[0];  
int maxEndingHere = array[0];
```

## 2. Loop through the array

```
for (int i = 1; i < array.size(); ++i) {  
    maxEndingHere = max(array[i], maxEndingHere + array[i]);  
    maxSoFar = max(maxSoFar, maxEndingHere);  
}
```

## 3. Return maxSoFar;

# Problem – 53. Maximum Subarray

Medium



LeetCode

[leetcode.com/problems/maximum-subarray](https://leetcode.com/problems/maximum-subarray)

**Code** Time:  $O(n)$  Space:  $O(1)$

```
int maxSubArray(vector<int>& nums) {  
    int maxSum = nums[0];  
    int currentSum = nums[0];  
    for (int i = 1; i < nums.size(); ++i) {  
        currentSum = max(nums[i], currentSum + nums[i]);  
        maxSum = max(maxSum, currentSum);  
    }  
    return maxSum;  
}
```

# Problem – 152. Maximum Product Subarray

Medium



LeetCode

[leetcode.com/problems/maximum-product-subarray](https://leetcode.com/problems/maximum-product-subarray)

## Problem

- You are given an array **nums**
- Find a subarray that has the largest product and return the product
- The array may contain negative numbers

- **Example:**

`nums = [2, 3, -2, 4]`

`output = 6`

`[2,3]` has the largest 6 ( $2 * 3$ )



# Problem – 152. Maximum Product Subarray

Medium



LeetCode

[leetcode.com/problems/maximum-product-subarray](https://leetcode.com/problems/maximum-product-subarray)

## Solution

- Use a modified version of Kadane's algorithm
- Keep track of the minimum and maximum product
- Once the current number is negative, swap minimum product with maximum product
- Check the largest product between maximum product and the final result

# Problem – 152. Maximum Product Subarray

Medium



LeetCode

[leetcode.com/problems/maximum-product-subarray](https://leetcode.com/problems/maximum-product-subarray)

**Code** Time:  $O(n)$  Space:  $O(1)$

```
int maxProduct(vector<int>& nums) {
    int result = nums[0];
    int maxProd = nums[0];
    int minProd = nums[0];
    for (int i = 1; i < nums.size(); ++i) {
        if (nums[i] < 0) {
            swap(minProd, maxProd);
        }

        minProd = min(nums[i], nums[i] * minProd); // -2
        maxProd = max(nums[i], nums[i] * maxProd); // -30

        result = max(result, maxProd);
    }
    return result;
}
```

# Problem – 153. Find Minimum in Rotated Sorted Array

Medium



LeetCode

[leetcode.com/problems/find-minimum-in-rotated-sorted-array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array)

## Problem

- You are given a sorted array but “rotated”
- Rotated means the elements are displaced in order
- Return the **minimum element**

- **Example:**

nums = [3,4,5,1,2]

output = 1 (minimum element)

# Solution – 153. Find Minimum in Rotated Sorted Array

Medium

 [leetcode.com/problems/find-minimum-in-rotated-sorted-array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array)

## Solution

- Perform an adapted binary search
- Example:

[3,4,5,1,2]

**left = 3, mid = 5, right = 2**

You find mid (5), but have to go right, so adjust left:

```
if (mid > right)
```

```
    left = mid + 1
```

```
else
```

```
    right = mid
```

# Code – 153. Find Minimum in Rotated Sorted Array

Medium



LeetCode

[leetcode.com/problems/find-minimum-in-rotated-sorted-array](https://leetcode.com/problems/find-minimum-in-rotated-sorted-array)

**Code** Time:  $O(\log n)$  Space:  $O(1)$

```
int findMin(vector<int>& nums) {
    int left = 0;
    int right = nums.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > nums[right]) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return nums[left];
}
```

# Problem - Best Time to Buy and Sell Stock

Easy



LeetCode

[leetcode.com/problems/best-time-to-buy-and-sell-stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock)

## Problem Statement

- You are given an integer **array** of stock prices
- Choose a **price[i]** to buy and **price[i]** to sell where you achieve maximum profits

- **Example:**

`prices = [9, 1, 3, 4]`

- **Output:** [1,3]

`array[3] - array[1] = 4 - 1 = 3`

# Solution - Best Time to Buy and Sell Stock

Easy



LeetCode

[leetcode.com/problems/best-time-to-buy-and-sell-stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock)

## Solution

- Initialize **profit = 0**
- Initialize **lowestBuyPrice = prices[0]**
- Loop through the prices
- Track the lowest buy price → **min(lowestBuyPrice, prices[i])**
- Check if selling “today” will make the maximum profit and update profit:  
**max(prices[i] - buy > profit, profit)**
- Update profit  
max(prices[i] - buy

# Code - Best Time to Buy and Sell Stock

Easy



LeetCode

[leetcode.com/problems/best-time-to-buy-and-sell-stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock)

## Code (simplified) Time: $O(n)$ Space: $O(1)$

```
int maxProfit(vector<int>& prices) {  
    int profit = 0;  
    int buy = prices[0];  
    for (auto i = 1; i < prices.size(); i++) {  
        buy = min(buy, prices[i]);  
        profit = max(profit, prices[i] - buy)  
    }  
    return profit;  
}
```



# Code - Best Time to Buy and Sell Stock

Easy



LeetCode

[leetcode.com/problems/best-time-to-buy-and-sell-stock](https://leetcode.com/problems/best-time-to-buy-and-sell-stock)

## Code (optimized) Time: $O(n)$ Space: $O(1)$

- Same logic, but with better branch prediction and less computation

```
int maxProfit(vector<int>& prices) {  
    int profit = 0;  
    int buy = prices[0];  
    for (auto i = 1; i < prices.size(); i++) {  
        if (prices[i] < buy) {  
            buy = prices[i];  
        } else if (prices[i] - buy > profit) {  
            profit = prices[i] - buy;  
        }  
    }  
    return profit;  
}
```

# Problem - Best Time to Buy and Sell Stock II

Medium

 LeetCode [leetcode.com/problems/best-time-to-buy-and-sell-stock-ii](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii)

## Problem

- You are given an integer **array** of stock prices
- Choose a **price[i]** to buy and **price[i]** to sell where you achieve maximum profits
- You can buy/sell **multiple times**, but only hold **at most one** transaction at a time
- Output is the **maximum profits**
- **Example:**

`prices = [9, 1, 3, 4]`

**Output:**  $2 + 1 = 3$

`buy (price = 1), sell (price = 3), profit = 2`

`buy (price = 3), sell (price = 4), profit = 1`

# Solution - Best Time to Buy and Sell Stock II

Medium

 LeetCode [leetcode.com/problems/best-time-to-buy-and-sell-stock-ii](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii)

## Solution

- Loop through the array starting from index 1
- If current **price[i]** is lower than previous **price[i - 1]**, buy and sell

- **Example:**

`prices = [1, 8, 4]`      `prices[0] = 1, prices[1] = 8, prices[2] = 4`

`prices[0] < prices[1] → true, profit = 8 - 1 = 7`

`prices[2] < prices[1] → false, do nothing`

# Code - Best Time to Buy and Sell Stock II

Medium



LeetCode

[leetcode.com/problems/best-time-to-buy-and-sell-stock-ii](https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii)

**Code** Time:  $O(n)$  Space:  $O(1)$

```
int maxProfit(vector<int>& prices) {  
    int profit = 0;  
    for (int i = 1; i < prices.size(); ++i) {  
        if (prices[i] > prices[i-1]) {  
            profit += prices[i] - prices[i - 1];  
        }  
    }  
    return profit;  
}
```

# Problem – 33. Search in Rotated Sorted Array

Medium



LeetCode

[leetcode.com/problems/search-in-rotated-sorted-array](https://leetcode.com/problems/search-in-rotated-sorted-array)

## Problem

- Variation of *Find Minimum in Sorted Rotated Array* problem
- You are given a sorted **array** but “rotated” and a target number **n**
- Rotated means the elements are displaced in order
- Search the number **n** and return its index

- **Example**

`nums = [4,5,6,7,0,1,2], target = 0`

**Output:** 4

**4** is the index where the target number **0** is located

# Problem – 33. Search in Rotated Sorted Array

Medium



LeetCode

[leetcode.com/problems/search-in-rotated-sorted-array](https://leetcode.com/problems/search-in-rotated-sorted-array)

## Solution

- Perform a binary search with some modification
- One side is always sorted, so find which side (left or right)
- Check if the target is in the range of the sorted side and adjust mid

Example:

[2,4,5,6,7,0,1] target = 0

1. Find mid (6)
2. Find the sorted side (left) = [2,4,5]
3. Check if your target is in this side. Is **target** between 2 and 5?
4. Adjust mid to search at the other side if not, otherwise continue searching at the same side

# Problem – 33. Search in Rotated Sorted Array

Medium



LeetCode

[leetcode.com/problems/search-in-rotated-sorted-array](https://leetcode.com/problems/search-in-rotated-sorted-array)

**Code** Time:  $O(\log n)$  Space:  $O(1)$

```
int search(vector<int>& nums, int target) {
    int left = 0;
    int right = nums.size() - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;

        // figure it out the sorted side
        if (nums[mid] >= nums[0]) {
            // left side is sorted
            // is target within this range?
            if (target >= nums[0] && target < nums[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        } else {
            // right side is sorted
            // is target within this range?
            if (target <= nums[right] && target > nums[mid]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }
    return -1;
}
```

# Problem – 15. 3Sum

Medium



LeetCode

[leetcode.com/problems/3sum](https://leetcode.com/problems/3sum)

## Problem

- You are given an array of integer **nums**
- Find distinct triples that the final sum is equal to zero

- **Example:**

`nums = [-1,0,1,2,-1,-4]`

**Output:**

`[[-1,-1,2],[-1,0,1]]`

**Explanation:**

$\text{nums}[0] + \text{nums}[1] + \text{nums}[2] = (-1) + 0 + 1 = 0.$

$\text{nums}[1] + \text{nums}[2] + \text{nums}[4] = 0 + 1 + (-1) = 0.$

$\text{nums}[0] + \text{nums}[3] + \text{nums}[4] = (-1) + 2 + (-1) = 0.$

The distinct triplets are `[-1,0,1]` and `[-1,-1,2]`.





LeetCode

[leetcode.com/problems/3sum](https://leetcode.com/problems/3sum)

## Solution

- Use three pointers: **i**, **j** and **k**
- Sort the array. This is necessary to move the pointers **j** and **k**
- Pointer **i** starts at the beginning the array
- Pointer **j** starts at  $i + 1$  (second position)
- Pointer **k** starts at the end of the array
- Pointer **i** always move forward until the end of the array
- For each value of **i**, **j** and **k** will move either forward or backward, depending on the results of the sum
- Once find a sum == 0, add to a set to guarantee no duplicates



LeetCode

[leetcode.com/problems/3sum](https://leetcode.com/problems/3sum)**Code** Time:  $O(n^2 \log n)$  Space:  $O(n^2)$ 

```
vector<vector<int>> threeSum(vector<int>& nums) {
    set<vector<int>> triplets;
    sort(nums.begin(), nums.end());
    for (int i = 0; i < nums.size() - 2; ++i) {
        int j = i + 1;
        int k = nums.size() - 1;
        // it is a solution
        while (j < k) {
            int sum = nums[i] + nums[j] + nums[k];
            if (sum == 0) {
                triplets.insert({nums[i], nums[j], nums[k]});
                j++;
                k--;
            }
            if (sum < 0) {
                j++;
            } else {
                k--;
            }
        }
    }
    vector<vector<int>> result;
    // convert the solutions to the expected return
    for (const auto& t : triplets) {
        result.push_back(t);
    }

    return result;
}
```

Note: why not a unordered\_set which is hash-based?

# Problem – 11. Container With Most Water

Medium



LeetCode

[leetcode.com/problems/container-with-most-water](https://leetcode.com/problems/container-with-most-water)

## Problem Statement

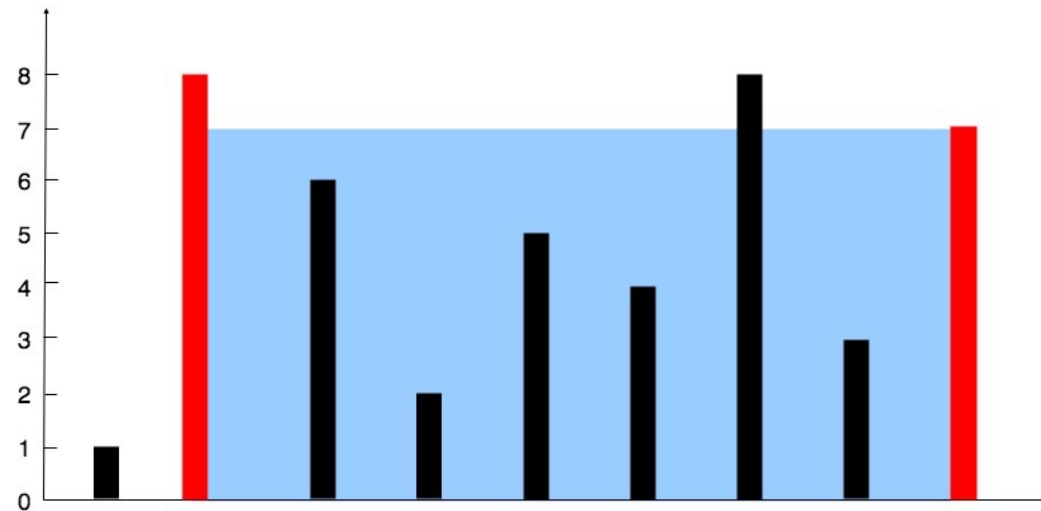
- You are given an integer array **height**
- Find two lines that together with x-axis form a container with most water
- Example:

### Input:

height = [1,8,6,2,5,4,8,3,7]

### Output:

49



# Solution – Container With Most Water

Medium



LeetCode

[leetcode.com/problems/container-with-most-water](https://leetcode.com/problems/container-with-most-water)

## Solution

- Initialize the maximum area **maxArea = 0**
- Initialize two pointers, **left = 0** and **right = height.size - 1**
- Loop while pointer **left < right**
- Calculate the area:  
**area = min(height[left], height[right]) \* (right - left)**
- Update the global maximum area:  
**maxArea = max(maxArea, area)**
- Move the smallest pointer (increment **left** or decrement **right**)
- Return **maxArea**

# Code – 11. Container With Most Water

Medium



LeetCode

[leetcode.com/problems/container-with-most-water](https://leetcode.com/problems/container-with-most-water)

**Code** Time:  $O(n)$  Space:  $O(1)$

```
int maxArea(vector<int>& height) {  
    // left and right = positions  
    int left = 0;  
    int right = height.size() - 1;  
    int maxArea = 0;  
    while (left < right) {  
        int area = min(height[left], height[right]) * (right - left);  
        maxArea = max(maxArea, area);  
        // adjust left and right based on 'greedy' algorithm  
        // move from the lowest height  
        if (height[left] < height[right]) {  
            left++;  
        } else {  
            right--;  
        }  
    }  
    return maxArea;  
}
```

# Problem – 268. Missing Number

Easy



LeetCode

<https://leetcode.com/problems/missing-number>

## Problem

- You are given an **array of nums** with **n** distinct numbers
- Return only the number in the range that is missing from the array

- **Example:**

nums = [3, 0, 1]

**Output:** 2

from the sequence 0,1,x,3 the missing number x is 2

# Solution – 268. Missing Number

Easy



LeetCode

<https://leetcode.com/problems/missing-number>

## Solution

- You have an array with **size n**. Example:

`nums = [3,0,1]`

`size = 3`

- It is expected the sum **0 + 1 + 2 + ... + n**:

`0 + 1 + 2 + 3 = 6`

- The **expected sum** can be calculated with **Gauss Formula**:  $\frac{n(n+1)}{2}$

`n = 3`

`n * (n + 1) / 2 = 3 * (3 + 1) / 2 = 6`

- Once you know the expected sum, calculate the sum from the elements of the array:

`sum = 3 + 0 + 1 = 4`

- Subtract from the expected value to find out the number:

`6 - 4 = 2 is the missing number`

# Code – 268. Missing Number

Easy



LeetCode

<https://leetcode.com/problems/missing-number>

**Code** Time:  **$O(n)$**  Space:  **$O(1)$**

```
int missingNumber2(vector<int>& nums) {
    int n = nums.size();
    // Gauss's formula: the sum of numbers 0 to n is n * (n + 1) / 2
    int expected = n * (n + 1) / 2;
    // calculate actual sum
    int actual = std::accumulate(nums.begin(), nums.end(), 0);
    // result is the expected - actual
    return expected - actual;
}
```