<https://leetcode.com/problems/minimum-moves-to-make-array-complementary/description/>

You are given an integer array nums of **even** length n and an integer limit. In one move, you can replace any integer from nums with another integer between 1 and limit, inclusive.

The array nums is **complementary** if for all indices i (**0-indexed**), nums[i] + nums[n - 1 - i] equals the same number. For example, the array [1,2,3,4] is complementary because for all indices i, nums[i] + nums[n - 1 - i] = 5.

Return the ***minimum*** *number of moves required to make* nums***complementary***.

**Example 1:**

Input: nums = [1,2,4,3], limit = 4

Output: 1

Explanation: In 1 move, you can change nums to [1,2,2,3] (underlined elements are changed).

nums[0] + nums[3] = 1 + 3 = 4.

nums[1] + nums[2] = 2 + 2 = 4.

nums[2] + nums[1] = 2 + 2 = 4.

nums[3] + nums[0] = 3 + 1 = 4.

Therefore, nums[i] + nums[n-1-i] = 4 for every i, so nums is complementary.

**Example 2:**

Input: nums = [1,2,2,1], limit = 2

Output: 2

Explanation: In 2 moves, you can change nums to [2,2,2,2]. You cannot change any number to 3 since 3 > limit.

**Example 3:**

Input: nums = [1,2,1,2], limit = 2

Output: 0

Explanation: nums is already complementary.

**Constraints:**

* n == nums.length
* 2 <= n <= 10^5
* 1 <= nums[i] <= limit <= 10^5
* n is even.

**Attempt 1: 2023-12-07**

**Solution 1: Sweep Line + Presum (720 min)**

class Solution {

public int minMoves(int[] nums, int limit) {

int n = nums.length;

int[] delta = new int[2 \* limit + 2];

for(int i = 0; i < n / 2; i++) {

// Initialize all slot in delta with 2 moves

// Range [2, 2 \* limit], we need to add 2 at number = 2

// and remove those 2 at number = 2 \* limit + 1

delta[2] += 2;

delta[2 \* limit + 1] -= 2;

// (1)[~, oneMoveMin - 1] -> 2 moves

// (2)[oneMoveMin, justGood-1] -> 1 move (if initial with all 2 moves,

// 1 less move needed, in Sweep Line tech mark as -1)

// (3)[justGood] -> 0 move (if initial with all 2 moves, 2 less move

// needed, in delta array means another 1 less move needed, in

// Sweep Line tech mark as -1)

// (4)[justGood + 1, oneMoveMax] -> 1 move (in delta array after

// 'justGood' which decrease to 0 move, now it means 1 more

// move needed, in Sweep Line tech mark as +1)

// (5)[oneMoveMax + 1, ~] -> 2 moves (in delta array after 'oneMoveMax'

// which require 1 move, now it means another 1 more move needed,

// in Sweep Line tech mark as +1)

// +2 -1 -1 +1 +1 -2

// |----2----------------------------------2\*limit----|

// ^ ^ ^ ^ ^ ^ ^ ^

// oneMoveMin justGood oneMoveMax

int pair\_min = Math.min(nums[i], nums[n - 1 - i]);

int pair\_max = Math.max(nums[i], nums[n - 1 - i]);

int oneMoveMin = pair\_min + 1;

int oneMoveMax = pair\_max + limit;

int justGood = pair\_min + pair\_max;

delta[oneMoveMin] -= 1;

delta[justGood] -= 1;

delta[justGood + 1] += 1;

delta[oneMoveMax + 1] += 1;

}

int result = n;

// Sweep Line with running count

int count = 0;

for(int i = 2; i <= 2 \* limit; i++) {

count += delta[i];

result = Math.min(result, count);

}

return result;

}

}

Time Complexity: O(n + limit)

Space Complexity: O(limit)

**Wrong Solution**

**这个方法直接把题目中关于limit的定义搞错了，当成了变动nums[i]的delta的范围，比如limit = 3，nums[i] = 1要变为5需要delta = 4 > limit = 3，这个active因为limit的限制不能成立，然而实际上limit的定义是在[1, limit]这个范围中选取一个数替换nums[i]让nums[i] + nums[n - 1 - i]的和为target sum，和之前错误的理解截然不同**

class Solution {

// The core is find max freq complementary sum up

// of nums[i] + nums[n - 1 - i]

public int minMoves(int[] nums, int limit) {

int n = nums.length;

int min = 100001;

int[] sum = new int[n];

for(int i = 0; i < n; i++) {

sum[i] = nums[i] + nums[n - 1 - i];

min = Math.min(min, nums[i]);

}

Map<Integer, Integer> freq = new HashMap<>();

// n is guaranteed as even, and sum up array looks

// like palindrome pairs, only need to check half

for(int i = 0; i < n / 2; i++) {

freq.put(sum[i], freq.getOrDefault(sum[i], 0) + 1);

}

// Max freq sum at maxPQ peek

PriorityQueue<int[]> maxPQ = new PriorityQueue<int[]>((a, b) -> b[1] - a[1]);

for(Map.Entry<Integer, Integer> e : freq.entrySet()) {

maxPQ.offer(new int[]{e.getKey(), e.getValue()});

}

int target\_sum = 0;

while(!maxPQ.isEmpty()) {

target\_sum = maxPQ.poll()[0];

// Find a 'target\_sum' able to convert all unequal sum to it

if(target\_sum - min <= limit) {

break;

}

}

int count = 0;

for(int i = 0; i < n / 2; i++) {

if(sum[i] != target\_sum) {

if(nums[i] != nums[n - 1 - i]) {

count += 1;

} else {

count += 2;

}

}

}

return count;

}

}

Input

nums =

[1,3,1,1,1,2,3,2,3,1,3,2,1,3]

limit =

3

Use Testcase

Output

5

Expected

4

Potential issue 1:

May because of different target sum may have same highest frequency, the

different pick up of target sum will create different set of remain sum(s)

which sum up via two different ways:

(1) same value nums[i] == nums[n - 1 - i]

(2) different value nums[i] != nums[n - 1 - i]

for (1) it take 2 actions to update to target sum

for (2) it take 1 action to update to target sum

and because of different set of remain sum(s) will derive different

required number of actions

============================================================================================

Then change to below but still failure

class Solution {

// The core is find max freq complementary sum up

// of nums[i] + nums[n - 1 - i]

public int minMoves(int[] nums, int limit) {

int n = nums.length;

int min = 100001;

int[] sum = new int[n];

for(int i = 0; i < n; i++) {

sum[i] = nums[i] + nums[n - 1 - i];

min = Math.min(min, nums[i]);

}

Map<Integer, Integer> freq = new HashMap<>();

// n is guaranteed as even, and sum up array looks

// like palindrome pairs, only need to check half

for(int i = 0; i < n / 2; i++) {

freq.put(sum[i], freq.getOrDefault(sum[i], 0) + 1);

}

// Max freq sum at maxPQ peek

PriorityQueue<int[]> maxPQ = new PriorityQueue<int[]>((a, b) -> b[1] - a[1]);

for(Map.Entry<Integer, Integer> e : freq.entrySet()) {

maxPQ.offer(new int[]{e.getKey(), e.getValue()});

}

int result = n;

int target\_sum = 0;

while(!maxPQ.isEmpty()) {

target\_sum = maxPQ.poll()[0];

// Find a 'target\_sum' able to convert all unequal sum to it

if(target\_sum - min <= limit) {

result = Math.min(result, action\_count(target\_sum, sum, nums, n));

}

}

return result;

}

private int action\_count(int target\_sum, int[] sum, int[] nums, int n) {

int count = 0;

for(int i = 0; i < n / 2; i++) {

if(sum[i] != target\_sum) {

if(nums[i] != nums[n - 1 - i]) {

count += 1;

} else {

count += 2;

}

}

}

return count;

}

}

Input

nums =

[1,3,1,1,1,2,3,2,3,1,3,2,1,3]

limit =

3

Use Testcase

Output

5

Expected

4

Potential issue 2:

It may because even when nums[i] == nums[n - 1 - i], we still need to two actions ?

Combined with below limit condition, it will go soar

// Find a 'target\_sum' able to convert all unequal sum to it

if(target\_sum - min <= limit) {

result = Math.min(result, action\_count(target\_sum, sum, nums, n));

}

For example, we have a nums pair = [1,3], limit = 3, target sum = 8, if only update 1 to 8 or update 3 to 8 will > limit = 3, but if we separate to two actions, 1 update to 4, 3 update to 4 also, which will make sum = 8 and not exceed limit = 3, anyway, the below condition not cover all scenarios:

if(sum[i] != target\_sum) {

if(nums[i] != nums[n - 1 - i]) {

count += 1;

} else {

count += 2;

}

}

**Step by Step**

e.g nums = [1,2,4,3], limit = 4

sum = [4,6,6,4] -> only need to consider half [4,6] since palindrome

sum 4 freq = 1, sum 6 freq = 1, freq same,

try both as both are max freq

all remain sum have unequal freq should attempt to change to max freq sum

if failed to convert to max freq sum, we have to try to convert all

sum to second max freq sum

(1) Try to change 4 to 6 -> sum = [6,6]

6 - 1 = 5 > limit = 4, not able to

(2) Try to change 6 to 4 -> sum = [4,4]

4 - 1 = 3 < limit = 4, able to, need to update sum[1] from 6

to 4, which create by nums[1] and nums[2] (2 = n - 1 - i),

and since nums[1] != nums[2], we only need 1 move on nums[2]

to decrease it from 4 to 2, so nums = [1,2,4,3] update to

nums = [1,2,2,3] require 1 move to make nums complementary

e.g nums = [1,2,2,1], limit = 2

sum = [2,4,4,2] -> only need to consider half [2,4] since palindrome

try both as both are max freq

all remain sum have unequal freq should attempt to change to max freq sum

if failed to convert to max freq sum, we have to try to convert all

sum to second max freq sum

(1) Try to change 2 to 4 -> sum = [4,4]

4 - 1 = 3 > limit = 2, not able to

(2) Try to change 4 to 2 -> sum = [2,2]

2 - 1 = 1 < limit = 2, able to need to update sum[1]

from 4 to 2, which create by nums[1] and nums[2] (2 = n - 1 - i),

and since nums[1] == nums[2], we need 2 moves on both nums[1] and

nums[2] to decrease it from 2 to 1, so nums = [1,2,2,1] update to

nums = [1,1,1,1] require 2 moves to make nums complementary

**Refer to**

<https://algo.monster/liteproblems/1674>

## Problem Description

You are presented with an integer array nums with an even length n and an integer limit. You're able to perform operations where you replace any integer in the array with another integer that is within the inclusive range from 1 to limit. The goal is to make the array complementary.

An array is complementary if for every index i (0-indexed), the sum nums[i] + nums[n - 1 - i] is the same across the entire array. For instance, the array [1,2,3,4] is complementary because every pair of numbers adds up to 5.

Your task is to determine the **minimum** number of moves required to make the array nums complementary.

## Intuition

To solve this problem efficiently, we need to think in terms of pairs and how changing an element impacts the sums across the array. We also need to record the minimum operations required to achieve a target sum for each pair at various intervals. The strategy is to use a difference array d to record the number of operations required to achieve each possible sum.

Here's the process of thinking that leads to the solution approach:

* Since nums has an even length, we only need to consider the first half of the array when paired with their corresponding element from the other end because we want to make nums[i] + nums[n - 1 - i] equal for all i.
* Each pair (nums[i], nums[n - 1 - i]) can contribute to the sum in three different ways:

1. No operation is needed if the sum of this pair already equals the target complementary sum.
2. One operation is needed if we can add a number to one of the elements in the pair to achieve the target sum.
3. Two operations when neither of the elements in the pair can directly contribute to the target sum and both need to be replaced.

* We use a trick called "difference array" or "prefix sum array" to efficiently update the range of sums with the respective counts of operations. We track the changes in the required operations as we move through different sum ranges.
* We increment moves for all the sums and later decrement as per the pair values and the allowed limits.
* Finally, we iterate over the sum range from 2 to 2 \* limit and accumulate the changes recorded in our difference array. This reveals the total moves required for each sum to be the target sum.
* Our goal is to find the minimum of these accumulated moves, which represents the least number of operations needed to make the array complementary.

Understanding the optimized approach is a bit tricky, but it cleverly manages the different cases to minimize the total number of moves.

## Solution Approach

The solution implements an optimization strategy called **difference array**, which is a form of prefix sum optimization. Here's the step by step explanation of how the algorithm works by walking through the code:

1. Initialize a difference array d with a length of 2 \* limit + 2. This array will store the changes in the number of moves required to make each sum complementary. The extra indices account for all possible sums which can occur from 2 (minimum sum) to 2 \* limit (maximum sum).

d = [0] \* (limit \* 2 + 2)

1. Loop through the first half of the nums array to check pairs of numbers. Here n >> 1 is a bitwise operation equivalent to dividing n by 2. For each pair (nums[i], nums[n - 1 - i]), determine their minimum (a) and maximum (b) values.

for i in range(n >> 1):

a, b = min(nums[i], nums[n - i - 1]), max(nums[i], nums[n - i - 1])

1. Update the difference array to reflect the default case where 2 moves are needed for all sums from 2 to 2 \* limit.

d[2] += 2

d[limit \* 2 + 1] -= 2

1. Update the difference array for scenarios where only 1 move is required. This happens when adding a number to the minimum of the pair, up to the maximum of the pair with limit.

d[a + 1] -= 1

d[b + limit + 1] += 1

1. For the exact sum a + b, decrease the number of moves by 1 as this is already part of the complementary sum. For a + b + 1, revert this operation since we consider the sum a + b only.

d[a + b] -= 1

d[a + b + 1] += 1

1. Initialize ans with n, representing the maximum possible number of moves (every element needing 2 moves), and a sum s which will be used to accumulate the total moves from the difference array.

ans, s = n, 0

1. Loop through the potential sums to calculate the prefix sum (which represents the cumulative number of moves) at each value. Update ans to hold the minimum number of moves required.

for v in d[2 : limit \* 2 + 1]:

s += v

if ans > s:

ans = s

1. Finally, we return ans, which now contains the minimum number of moves required to make nums complementary.

By leveraging the difference array technique, the algorithm efficiently calculates the minimum number of operations required by reducing the problem to range updates and point queries, which can be processed in a linear time complexity relative to the limit.

### Example Walkthrough

Let's walk through an example using the above solution approach. Suppose we have the following:

Integer array nums: [1, 2, 4, 3]

Integer limit: 4

The length of the array n is 4, which is even, and the limit is 4, so any replacement operations must yield a number between 1 to 4.

* We initialize our difference array d with a size of 2 \* limit + 2 to account for all possible sums.

d = [0] \* (4 \* 2 + 2) # [0, 0, 0, 0, 0, 0, 0, 0, 0]

* There are two pairs to consider: (1, 3) and (2, 4).
* The difference array is first updated to indicate that, by default, 2 moves are needed for each sum.

d[2] += 2

d[9] -= 2 # Since our limit is 4, `limit \* 2 + 1` is 9.

* Now, we loop through each pair. For the first pair (1, 3), the minimum is a = 1 and the maximum is b = 3. We then update the difference array based on the cases where 1 move is required.

d[a + 1] -= 1 # d[2] becomes 1

d[b + limit + 1] += 1 # d[8] becomes 1

* For the exact sum a + b, which is 4, we decrease the number of moves by 1 as this is already part of the complementary sum. For the value after a + b, which is 5, we revert this operation.

d[a + b] -= 1 # d[4] becomes -1

d[a + b + 1] += 1 # d[5] becomes 1

* We repeat steps 4 and 5 for the second pair (2, 4). The minimum is a = 2 and the maximum is b = 4.

d[a + 1] -= 1 # d[3] becomes 1

d[b + limit + 1] += 1 # d[9] stays at -1 because of the earlier decrement

d[a + b] -= 1 # d[6] becomes -1

d[a + b + 1] += 1 # d[7] becomes 1

* After updating the difference array with all pairs, we initialize ans with the maximum possible number of moves and then accumulate the changes to find the total moves required for each sum.

ans, s = n, 0 # ans = 4, s = 0

* Finally, we iterate through the possible sums using the accumulated changes and find the minimum number of moves.

for v in d[2:8 + 1]:

s += v

ans = min(ans, s)

The final accumulated changes in the difference array would be something like [0, 0, 1, 1, 0, 1, -1, 0, 2, -1]. We add these to our running total s and continuously update ans to find that the minimum number of moves required to make nums complementary is 1.

Therefore, the minimum number of moves required to make the array [1, 2, 4, 3] complementary with a limit of 4 is 1, which could be accomplished by changing the 4 to a 2.

class Solution {

public int minMoves(int[] nums, int limit) {

int pairsCount = nums.length / 2;// Store the number of pairs

int[] delta = new int[limit \* 2 + 2]; // Delta array to store the cost changes

// Iterate over each pair

for (int i = 0; i < pairsCount; ++i) {

// Taking the minimum and maximum of the pair for optimization

int minOfPair = Math.min(nums[i], nums[nums.length - i - 1]);

int maxOfPair = Math.max(nums[i], nums[nums.length - i - 1]);

// Always needs 2 moves if sum is larger than max possible sum 'limit \* 2'

delta[2] += 2;

delta[limit \* 2 + 1] -= 2;

// If we make minOfPair + 1 the new sum, we would need one less move

delta[minOfPair + 1] -= 1;

// If the sum is greater than maxOfPair + limit, then we'll need an additional move

delta[maxOfPair + limit + 1] += 1;

// We need one less move for making the sum equals minOfPair + maxOfPair (to make a straight sum)

delta[minOfPair + maxOfPair] -= 1;

delta[minOfPair + maxOfPair + 1] += 1;

}

int minMoves = pairsCount \* 2; // Initialize minMoves to the maximum possible value

int currentCost = 0; // Variable to accumulate current cost

// Iterate over possible sums

for (int sum = 2; sum <= limit \* 2; ++sum) {

currentCost += delta[sum]; // Update current cost

// If current cost is less than minMoves, update minMoves

if (currentCost < minMoves) {

minMoves = currentCost;

}

}

return minMoves; // Return the minimum moves required

}

}

**Prefix sum O(n + limit) with detailed examples and pseudocode**

**Refer to**

<https://leetcode.com/problems/minimum-moves-to-make-array-complementary/solutions/953078/prefix-sum-o-n-limit-with-detailed-examples-and-pseudocode/>

## **Intuition**

For each pair of numbers (at index i and N - 1 - i) l and r:

By 2 moves we can get any number between [2, limit\*2] (including move a number to the same number)

After only one move (change one of the numbers to a number between 1 and limit)

The minimum sum we can get is (min(l, r) + 1) (let this be oneMoveMin)

The maximum sum we can get is (max(l, r) + limit) (let this be oneMoveMax)

We need no move to get (l + r) (let this be justGood)

Therefore, to get:

* [~, oneMoveMin - 1] - 2 moves
* [oneMoveMin, justGood-1] - 1 move
* [justGood] - 0 move
* [justGood + 1, oneMoveMax] - 1 move
* [oneMoveMax + 1, ~] - 2 moves

For each pair of numbers

* We start with 2 moves
* From oneMoveMin we need 1 less move
* From justGood we need another 1 less move
* From justGood + 1 we need 1 more move
* From oneMoveMax + 1 we need another 1 more move

For all numbers

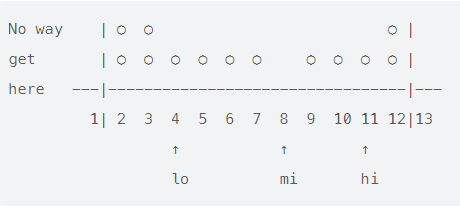
* We can at most move N times
* From 2 to limit\*2, we accumulate the number of moves all pairs of numbers subtract/add
* Find the smallest

## **Examples**

Suppose we have a pair of numbers (3, 5) and limit 6

* oneMoveMin (lo) = replace bigger number with 1 = 3 + (5->1) = 3 + (1) = 4
* oneMoveMax (hi) = replace smaller number with limit = (3->6) + 5 = (6) + 5 = 11
* justGood (mi) = 3 + 5 = 8

The diagram below shows minimum (optimal) number of moves (○) we need to get each target



No way | ○ ○ ○ |

get | ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ |

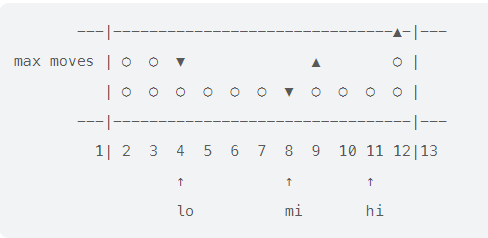
here ---|---------------------------------|---

1| 2 3 4 5 6 7 8 9 10 11 12|13

↑ ↑ ↑

lo mi hi

Now, if we want to scan this range from left (2) to right (limit\*2), and we know the maximum number of moves we can make is 2 (replace both numbers). We can transfer the diagram into below:



---|-------------------------------▲-|---

max moves | ○ ○ ▼ ▲ ○ |

| ○ ○ ○ ○ ○ ○ ▼ ○ ○ ○ ○ |

---|---------------------------------|---

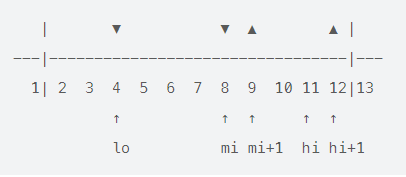
1| 2 3 4 5 6 7 8 9 10 11 12|13

↑ ↑ ↑

lo mi hi

**The black arrows ▼(-1) and ▲(+1) are what we need to take care.**

Now we can further simplify the diagram as:



| ▼ ▼ ▲ ▲ |

---|---------------------------------|---

1| 2 3 4 5 6 7 8 9 10 11 12|13

↑ ↑ ↑ ↑ ↑

lo mi mi+1 hi hi+1

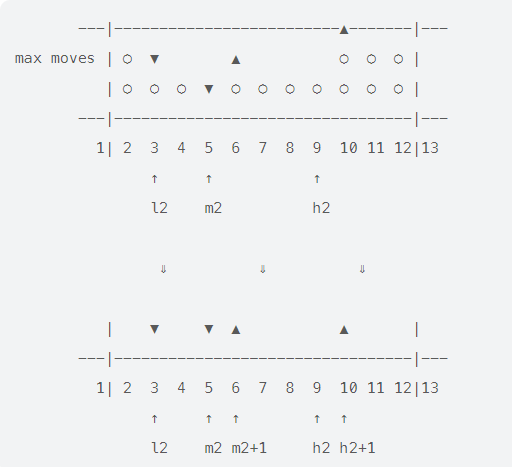
OK. Now let's look at another pair of numbers (2, 3) and limit is still 6

oneMoveMin (l2) = 2 + (1) = 3

oneMoveMax (h2) = (6) + 3 = 9

justGood (m2) = 2 + 3 = 5

We can make the diagram



---|-------------------------▲-------|---

max moves | ○ ▼ ▲ ○ ○ ○ |

| ○ ○ ○ ▼ ○ ○ ○ ○ ○ ○ ○ |

---|---------------------------------|---

1| 2 3 4 5 6 7 8 9 10 11 12|13

↑ ↑ ↑

l2 m2 h2

⇓ ⇓ ⇓

| ▼ ▼ ▲ ▲ |

---|---------------------------------|---

1| 2 3 4 5 6 7 8 9 10 11 12|13

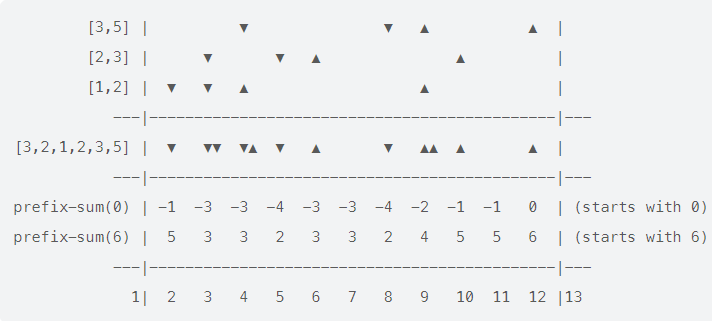
↑ ↑ ↑ ↑ ↑

l2 m2 m2+1 h2 h2+1

Finally, let's look at the array [3, 2, 1, 2, 3, 5] and limit 6.

This is essentially the combination of pairs [3, 5] [2, 3] [1, 2] (the first 2 pairs are just examples above)

We simply combine the diagrams we got from previous steps:



[3,5] | ▼ ▼ ▲ ▲ |

[2,3] | ▼ ▼ ▲ ▲ |

[1,2] | ▼ ▼ ▲ ▲ |

---|---------------------------------------------|---

[3,2,1,2,3,5] | ▼ ▼▼ ▼▲ ▼ ▲ ▼ ▲▲ ▲ ▲ |

---|---------------------------------------------|---

prefix-sum(0) | -1 -3 -3 -4 -3 -3 -4 -2 -1 -1 0 | (starts with 0)

prefix-sum(6) | 5 3 3 2 3 3 2 4 5 5 6 | (starts with 6)

---|---------------------------------------------|---

1| 2 3 4 5 6 7 8 9 10 11 12 |13

We know the maximum number of moves we can do is 6 (all numbers)**Result: the minimum moves required to get 5 or 8 is:** (you can do either)

6 + (-4) = 2 (starts with 0)

2 (starts with limit)

## **Implementation**

### **Step 1 - build the array**

array memo(size=(limit\*2 + 2), initialValue=(0))

for (i from 0 to size(nums)/2-1) do

l = ith number of nums

r = ith number from the right hand side of nums

memo[min(l, r) + 1] -= 1

memo[l + r] -= 1

memo[l + r + 1] += 1

memo[max(l, r) + limit + 1] += 1

end for

Note: size of the array memo is limit\*2 + 2. Slot 0 is placeholder, and 1 cannot be reached. We use slot 2~limit\*2, and we use an extra slot to keep the calculation smooth. Because it is possible that max(l, r) + limit + 1 and/or l + r + 1 goes beyond the limit\*2 boundary. It's not a must though since we don't care about the value.

### **Step 2 - calculate the result**

ans = MAX

curr = 0

for (i from 2 to limit\*2) do

curr += memo[i]

ans = min(ans, size(nums) + curr)

end for

Or, below works the same

ans = MAX

curr = size(nums)

for (i from 2 to limit\*2) do

curr += memo[i]

ans = min(ans, curr)

end for

### **Sample code in C++**

int minMoves(vector<int>& nums, int limit) {

int N = nums.size();

vector<int> memo(limit\*2 + 2, 0);

for (int i = 0; i < N/2; ++i) {

int l = nums[i], r = nums[N-1-i];

--memo[min(l, r) + 1];

--memo[l + r];

++memo[l + r + 1];

++memo[max(l, r) + limit + 1];

}

int ans = N, curr = N;

for (int i = 2; i <= limit\*2; ++i) {

curr += memo[i];

ans = min(ans, curr);

}

return ans;

}

## **Complexity**

Time: O(n + limit)

Space: O(limit)