<https://leetcode.com/problems/minimum-size-subarray-in-infinite-array/description/>

You are given a **0-indexed** array nums and an integer target.

A **0-indexed** array infinite\_nums is generated by infinitely appending the elements of nums to itself.

Return *the length of the****shortest****subarray of the array*infinite\_nums*with a sum equal to*target*.* If there is no such subarray return -1.

**Example 1:**

**Input:** nums = [1,2,3], target = 5

**Output:** 2

**Explanation:** In this example infinite\_nums = [1,2,3,1,2,3,1,2,...].

The subarray in the range [1,2], has the sum equal to target = 5 and length = 2.

It can be proven that 2 is the shortest length of a subarray with sum equal to target = 5.

**Example 2:**

**Input:** nums = [1,1,1,2,3], target = 4

**Output:** 2

**Explanation:** In this example infinite\_nums = [1,1,1,2,3,1,1,1,2,3,1,1,...].

The subarray in the range [4,5], has the sum equal to target = 4 and length = 2.

It can be proven that 2 is the shortest length of a subarray with sum equal to target = 4.

**Example 3:**

**Input:** nums = [2,4,6,8], target = 3

**Output:** -1

**Explanation:** In this example infinite\_nums = [2,4,6,8,2,4,6,8,...].

It can be proven that there is no subarray with sum equal to target = 3.

**Constraints:**

1 <= nums.length <= 10^5

1 <= nums[i] <= 10^5

1 <= target <= 10^9

**Attempt 1: 2025-07-09**

**Solution 1: Not Fixed Length Sliding Window (60 min)**

class Solution {

    public int minSizeSubarray(int[] nums, int target) {

        long sum = 0;

        for(int num : nums) {

            sum += num;

        }

        int n = nums.length;

        // Calculates complete copies needed first

        int complete = n \* (int) (target / sum);

        // Reduces target to remainder

        int newTarget = (int) (target % sum);

        int partialMinLen = n;

        int partialSum = 0;

        int i = 0;

        // We need concatenated two original nums array to exhaust

        // all potential subarrays, hence expands window by moving

        // end pointer between 0 to 2n - 1

        // Uses modulo operation (% n) to virtually extend the array

        // Avoids O(n) space for creating extended array

        // Shrinks window from start when sum exceeds new target

        // Tracks minimum valid window size (partial)

        for(int j = 0; j < 2 \* n; j++) {

            partialSum += nums[j % n];

            while(partialSum > newTarget) {

                partialSum -= nums[i % n];

                i++;

            }

            if(partialSum == newTarget) {

                partialMinLen = Math.min(partialMinLen, j - i + 1);

            }

        }

        return partialMinLen != n ? partialMinLen + complete : -1;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Solution 2: Prefix + Hash Table (60 min)**

class Solution {

    public int minSizeSubarray(int[] nums, int target) {

        long sum = 0;

        for(int num : nums) {

            sum += num;

        }

        int n = nums.length;

        // Calculates complete copies needed first

        int complete = n \* (int) (target / sum);

        // Reduces target to remainder

        int newTarget = (int) (target % sum);

        int partialMinLen = n;

        // Hashmap store <k,v> as <presum,index>

        // and since we update Hashmap during iteration,

        // the presum always mapping to the most recent

        // index, and when we try to find difference

        // between current index and stored presum's index

        // and do minus calculation against the presum

        // index, will always guarantee the difference

        // between two indexes is minimum

        long preSum = 0;

        Map<Long, Integer> pos = new HashMap<>();

        pos.put(0L, -1);

        // We need concatenated two original nums array to exhaust

        // all potential subarrays, hence expands window by moving

        // end pointer between 0 to 2n - 1

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

            preSum += nums[i % n];

            // Check regular subarray

            // |------------------ presum ------------------|

            // |--- presum - newTarget ---|--- newTarget ---|

            if(pos.containsKey(preSum - newTarget)) {

                partialMinLen = Math.min(partialMinLen, i - pos.get(preSum - newTarget));

            }

            // Check wrapped subarray

            // |----------- presum -----------|--- newTarget ----|

            // |<------------- sum -----------|--1->|<-----2-----|-- sum ------------->|

            //                                  preSum + newTarget - sum

            //                                i     n

            //                                      0     pos.get(preSum + newTarget - sum)

            // Section 1 as suffix length = n - i

            // Section 2 as prefix length = pos.get(preSum + newTarget - sum) - 0

            // newTarget length = Section 1 + Section 2

            if(pos.containsKey(preSum + newTarget - sum)) {

                partialMinLen = Math.min(partialMinLen, n - i + pos.get(preSum + newTarget - sum));

            }

            pos.put(preSum, i);

        }

        return partialMinLen != n ? partialMinLen + complete : -1;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/minimum-size-subarray-in-infinite-array/solutions/4124360/c-java-python-sliding-window-example-image-explanation/>

**Intuition**

Since we have infinite copies of *nums*, we know *target* can be much larger than total sum of *nums* and still achievable. But we will have to take that many copies of complete array as required. And the remainder of target will be then made by some other part of array.

Let total = sum(nums)

Then, we will be taking ⌊target / total⌋ complete copies of nums. And remainder target % total will be achieved by some partial part of array.

In other words, this can be just written in division form as

target = div \* total + rem

where div = ⌊target / total⌋ and rem = target % total

Now,

complete = n \* div = n \* ⌊target / total⌋

newTarget = target % total

The final answer will be dependent on whether we are able to make newTarget from the array. If we can make newTarget from partial number of elements, final answer will be complete + partial, otherwise -1.

But, what are all possible "kind" of subarrays we can use to make newTarget?

To answer this, first we need to see what are the ways we can select div number of complete copies of array.

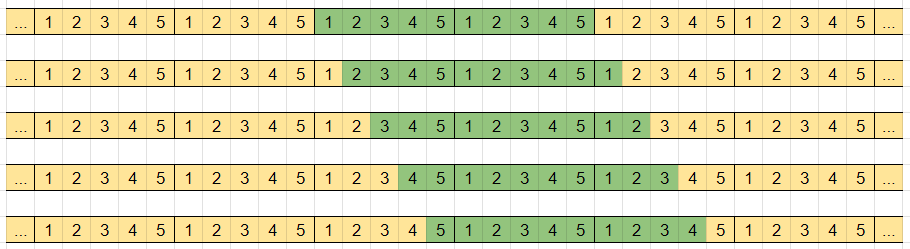
**Example**

nums = [1, 2, 3, 4, 5], target = 36

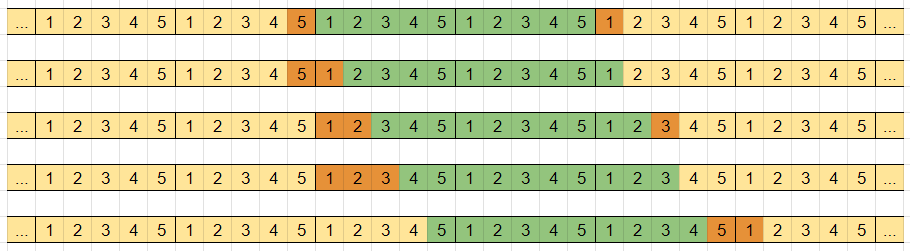
total = sum(nums) = 15

target = 2 \* total + 6

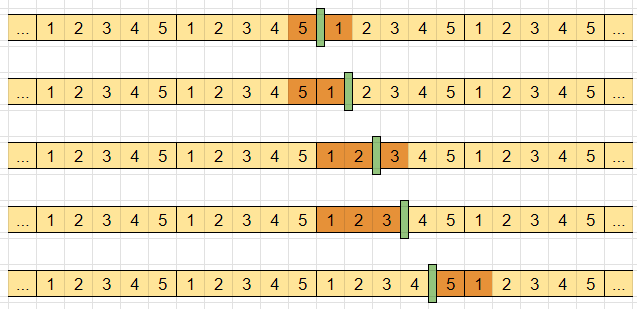
We need to take 2 complete copies of array. There are nums.length = 5 ways to do this (start from any one index and take 2 complete copies)



Now, we have taken window of size 2 \* 5 = 10 in our subarray, and are left to make newTarget = target % total = 6. This can be achieved for above different cases as:



But, how are we calculating the smallest size window to make newTarget? Let's *collapse* the 2 complete copies of array to see what is actually happening with the remaining newTarget.



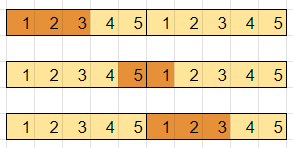
If you look carefully, collapsing any number of complete copies of array from infinite\_nums will not change it. It will still be infinite\_nums. But now we would be searching for smallest subarray making newTarget = target % total instead of whole big target.

newTarget = target % total

0 <= newTarget <= total - 1

Since, newTarget is atmost total - 1, it will be achieved from atmost n - 1 elements. So, we need to search for newTarget in only one copy of array.

This will cover the cases of subarray [1, 2, 3] in above example. But it will ignore the cases of subarray [5, 1]. These are the cases where newTarget can be made by taking some elements from beginning of array and some elements from end of array and ignoring some elements in between them. To calculate for these cases, we need 2 copies of array to search for newTarget.



Using this, we will be able to evaluate for these type of cases too and give minimum from them finally.

**Approach**

Calculate

total = sum(nums)

complete = n \* ⌊target / total⌋

newTarget = target % total (or just call it target only), target = target % total

To operate on 2 copies of nums to find target, either append one copy to given array (linear extra space), or use modular arithmetic.

For any index i where 0 <= i < 2n, it's corresponding index in nums will be i' = i % n.

For 0 <= i < n, i' = i

For n <= i < 2n, i' = i - n

Maintain variable partial that will store minimum size of subarray making target. It can be maximum n - 1 so initialize it with n. Then if it is still n until last, we will know that target can't be achieved and return -1.

Maintain start, end pointers of current window and its current sum.

Loop over end from 0 <= end < 2n, adding nums[end'] to sum. Remove elements from beginning of window nums[start'] if current window's sum is bigger target.

Update the answer for partial if sum = target.

class Solution {

public int minSizeSubarray(int[] nums, int target) {

long total = Arrays.stream(nums).sum(), sum = 0;

int n = nums.length, partial = n, complete = n \* (int)(target / total), start = 0;

target %= total;

for (int end = 0; end < 2 \* n; end++) {

sum += nums[end % n];

while (sum > target) {

sum -= nums[start++ % n];

}

if (sum == target) {

partial = Math.min(partial, end - start + 1);

}

}

return partial != n ? partial + complete : -1;

}

}

**Complexity**

Time complexity: *O*(*n*)

Space complexity: *O*(1)

**Refer to Deepseek**

**Problem Overview**

Given a 0-indexed array nums and an integer target, we need to find the length of the shortest contiguous subarray in an infinite array formed by repeating nums infinitely that sums to exactly target. If no such subarray exists, return -1 .

**Key Insights**

**Infinite Array Concept**: The infinite array is formed by concatenating nums infinitely (e.g., [1,2,3] becomes [1,2,3,1,2,3,...])

**Sum Reduction**: If target is larger than the sum of nums, we can reduce the problem by considering complete copies of nums first

**Prefix Sum + Hash Map**: For the remaining target after reduction, we use prefix sums and a hash map to efficiently find subarrays

**Solution Approach**

Calculate the total sum s of nums

Handle cases where target is larger than s by using complete copies

Use prefix sums and a hash map to find the shortest subarray for the remaining target

Consider both regular subarrays and "wrapped around" subarrays (combining end and beginning)

**Java Solution Code**

class Solution {

public int minSizeSubarray(int[] nums, int target) {

long s = Arrays.stream(nums).sum();

int n = nums.length;

int a = 0;

// Handle cases where target > s

if (target > s) {

a = n \* (target / (int) s);

target -= target / s \* s;

}

// If remaining target equals the sum, return n

if (target == s) {

return n;

}

// Use prefix sum and hash map to find the shortest subarray

Map<Long, Integer> pos = new HashMap<>();

pos.put(0L, -1);

long pre = 0;

int b = Integer.MAX\_VALUE;

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

pre += nums[i];

// Check for regular subarray

if (pos.containsKey(pre - target)) {

b = Math.min(b, i - pos.get(pre - target));

}

// Check for wrapped around subarray

if (pos.containsKey(pre - (s - target))) {

b = Math.min(b, n - (i - pos.get(pre - (s - target))));

}

pos.put(pre, i);

}

return b == Integer.MAX\_VALUE ? -1 : a + b;

}

}

**Explanation**

**Initial Setup**: Calculate the sum s of nums and initialize variables

**Target Reduction**: If target is larger than s, calculate how many full copies of nums we need and reduce target accordingly

**Prefix Sum Tracking**: Use a hash map to store prefix sums and their indices for O(1) lookups

**Subarray Search**: For each element:

Check if pre - target exists in the map (regular subarray)

Check if pre - (s - target) exists (wrapped around subarray)

Update the minimum length found

**Result**: Return the combined length of full copies and the found subarray, or -1 if none exists

**Complexity Analysis**

**Time Complexity**: O(n) - We make a single pass through the array

**Space Complexity**: O(n) - For storing prefix sums in the hash map

This solution efficiently handles the infinite array concept while maintaining optimal time and space complexity.

**How about Sliding Window solution ? is that better ?**

A sliding window approach can indeed be applied to this problem, but with some modifications to handle the infinite array aspect. Here's how it compares to the prefix sum + hash map solution:

**Sliding Window Approach**

class Solution {

public int minSizeSubarray(int[] nums, int target) {

long totalSum = Arrays.stream(nums).sum();

int n = nums.length;

int fullCopies = 0;

// Handle cases where target is larger than totalSum

if (target > totalSum) {

fullCopies = n \* (target / (int)totalSum);

target %= totalSum;

if (target == 0) return fullCopies;

}

// Create extended array (nums + nums) to handle circular cases

int[] extended = new int[2 \* n];

System.arraycopy(nums, 0, extended, 0, n);

System.arraycopy(nums, 0, extended, n, n);

int left = 0;

long currentSum = 0;

int minLength = Integer.MAX\_VALUE;

for (int right = 0; right < extended.length; right++) {

currentSum += extended[right];

while (currentSum > target && left <= right) {

currentSum -= extended[left];

left++;

}

if (currentSum == target) {

minLength = Math.min(minLength, right - left + 1);

}

}

return minLength == Integer.MAX\_VALUE ? -1 : fullCopies + minLength;

}

}

**Comparison of Approaches**

**Prefix Sum + Hash Map (Original Solution)**

**Pros**:

O(n) time complexity with single pass

Handles circular cases without extending array

More efficient for very large targets

**Cons**:

Requires O(n) space for hash map

Logic for wrapped-around subarrays is slightly complex

**Sliding Window (New Solution)**

**Pros**:

More intuitive implementation

Doesn't require hash map (though uses O(n) space for extended array)

Easier to understand the window movement

**Cons**:

Needs to create extended array (2n space)

In worst case does 2n iterations

Less efficient when target is much larger than totalSum

**When to Use Each Approach**

**Use Prefix Sum + Hash Map when**:

Memory is a concern (though both use O(n) space)

Dealing with very large targets (more efficient reduction)

Need optimal theoretical performance

**Use Sliding Window when**:

Prefer simpler, more readable code

The array size is reasonably small

Circular logic is easier to handle via array extension

**Final Recommendation**

The prefix sum solution is generally better for this specific problem because:

It handles the infinite array aspect more elegantly

Doesn't require creating an extended array

Has better performance characteristics for large inputs

However, the sliding window approach is perfectly valid and might be preferable in interview settings where code readability is prioritized. Both solutions have O(n) time complexity, but the prefix sum solution has slightly better constant factors.

**I find a Sliding Window solution not requires extra 'extended' array creation as above one**

You're absolutely right! This is an excellent optimization of the sliding window approach that avoids creating an extended array by using modulo arithmetic. Let me analyze this improved solution:

**Optimized Sliding Window Solution Analysis**

class Solution {

public int minSizeSubarray(int[] nums, int target) {

long total = Arrays.stream(nums).sum(), sum = 0;

int n = nums.length, partial = n, complete = n \* (int)(target / total), start = 0;

target %= total;

for (int end = 0; end < 2 \* n; end++) {

sum += nums[end % n];

while (sum > target) {

sum -= nums[start++ % n];

}

if (sum == target) {

partial = Math.min(partial, end - start + 1);

}

}

return partial != n ? partial + complete : -1;

}

}

**Key Improvements:**

**Circular Array Handling**:

Uses modulo operation (% n) to virtually extend the array

Avoids O(n) space for creating extended array

Still maintains the sliding window properties

**Efficient Target Reduction**:

Calculates complete copies needed first (complete = n \* (target / total))

Reduces target to remainder (target %= total)

**Sliding Window Mechanics**:

Expands window by moving end pointer (0 to 2n-1)

Shrinks window from start when sum exceeds target

Tracks minimum valid window size (partial)

**Complexity Analysis:**

**Time Complexity**: O(n)

The outer loop runs 2n times

Each element is processed at most twice (added and removed once)

**Space Complexity**: O(1)

Only uses constant extra space (no hash map or extended array)

**Comparison with Previous Approaches:**

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Time | Space | Key Advantage |
| Prefix Sum + Hash Map | O(n) | O(n) | Handles very large targets well |
| Extended Array Sliding Window | O(n) | O(n) | More intuitive |
| **This Optimized Sliding Window** | O(n) | O(1) | Best space efficiency |

**Why This is the Best Solution:**

**Space Efficiency**: Truly O(1) space by avoiding both hash map and extended array

**Readability**: Clean sliding window implementation

**Practical Performance**: Excellent cache locality due to sequential access

**Handles Edge Cases**: Properly deals with:

Target equal to sum of array

Need for multiple array copies

Wrapped-around subarrays

**Example Walkthrough:**

For nums = [1,2,3,4,5], target = 12:

total = 15, complete = 0 (12 < 15)

Window finds [3,4,5] (sum=12) at positions 2-4

Returns length 3

For nums = [1,1,1,2,3], target = 4:

total = 8, complete = 0 (4 < 8)

Window finds [1,1,2] (positions 0-2) and [1,3] (positions 3-4)

Returns minimum length 2

This solution is indeed superior to both the prefix sum and extended array approaches for this problem, providing optimal space efficiency while maintaining excellent time complexity.

**Refer to**

[L209.P2.2.Minimum Size Subarray Sum (Ref.L862)](note://195893F25BDC45F2B63535FC5D9A4B26)

[L345.Reverse Vowels of a String (Ref.L344,L1119)](note://CA879E2192A94099996642D7560B6FB9)

[L918.Maximum Sum Circular Subarray (Ref.L53,L1658)](note://WEB1495ae8db0086f5ab30de99c6528274c)