**Introduction**

In problems where we deal with sorted arrays (or LinkedLists) and need to find a set of elements that fulfill certain constraints, the Two Pointers approach becomes quite useful. The set of elements could be a pair, a triplet or even a subarray. For example, take a look at the following problem:

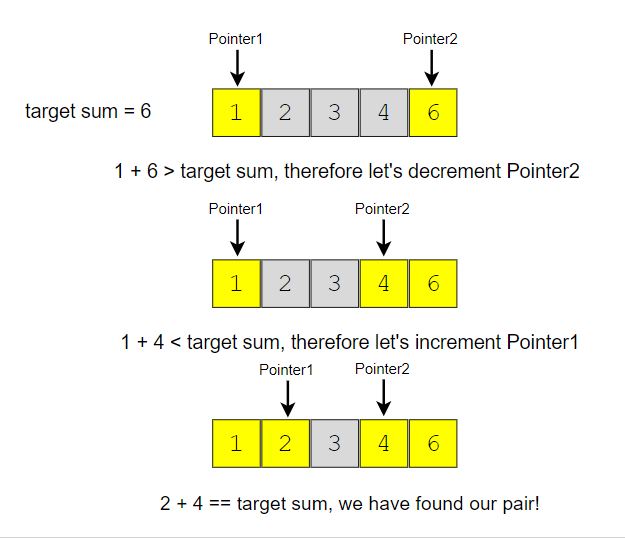
Given an array of sorted numbers and a target sum, find a pair in the array whose sum is equal to the given target.

To solve this problem, we can consider each element one by one (pointed out by the first pointer) and iterate through the remaining elements (pointed out by the second pointer) to find a pair with the given sum. The time complexity of this algorithm will be O(N^2)*O*(*N*2) where ‘N’ is the number of elements in the input array.

Given that the input array is sorted, an efficient way would be to start with one pointer in the beginning and another pointer at the end. At every step, we will see if the numbers pointed by the two pointers add up to the target sum. If they do not, we will do one of two things:

1. If the sum of the two numbers pointed by the two pointers is greater than the target sum, this means that we need a pair with a smaller sum. So, to try more pairs, we can decrement the end-pointer.
2. If the sum of the two numbers pointed by the two pointers is smaller than the target sum, this means that we need a pair with a larger sum. So, to try more pairs, we can increment the start-pointer.

Here is the visual representation of this algorithm:



The time complexity of the above algorithm will be O(N)*O*(*N*).

In the following chapters, we will apply the **Two Pointers** approach to solve a few problems.

# Pair with Target Sum (easy)

**We'll cover the following**

* [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Problem-Statement)
* [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Try-it-yourself)
* [Solution](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Solution)
  + - [Code](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Code)
  + [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Time-Complexity)
  + [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Space-Complexity)
* [An Alternate approach](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#An-Alternate-approach)
  + [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Time-Complexity)
  + [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Space-Complexity)

### Problem Statement[**#**](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Problem-Statement)

Given an array of sorted numbers and a target sum, find a **pair in the array whose sum is equal to the given target**.

Write a function to return the indices of the two numbers (i.e. the pair) such that they add up to the given target.

**Example 1:**

Input: [1, 2, 3, 4, 6], target=6  
Output: [1, 3]  
Explanation: The numbers at index 1 and 3 add up to 6: 2+4=6

**Example 2:**

Input: [2, 5, 9, 11], target=11  
Output: [0, 2]  
Explanation: The numbers at index 0 and 2 add up to 11: 2+9=11

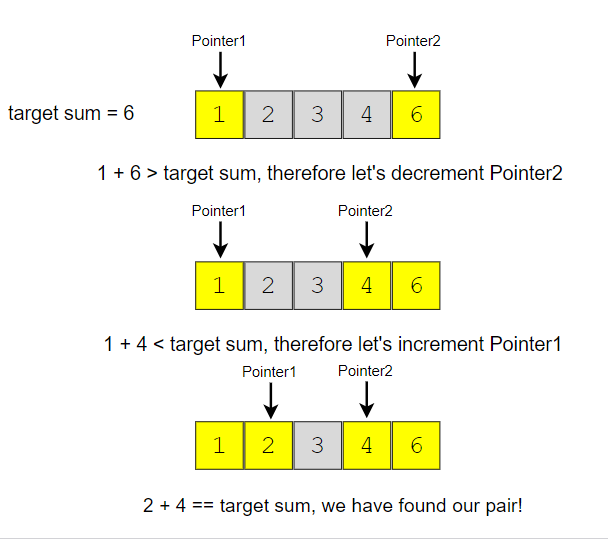
## Solution[**#**](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Solution)

Since the given array is sorted, a brute-force solution could be to iterate through the array, taking one number at a time and searching for the second number through **Binary Search**. The time complexity of this algorithm will be O(N\*logN)*O*(*N*∗*logN*). Can we do better than this?

We can follow the **Two Pointers** approach. We will start with one pointer pointing to the beginning of the array and another pointing at the end. At every step, we will see if the numbers pointed by the two pointers add up to the target sum. If they do, we have found our pair; otherwise, we will do one of two things:

1. If the sum of the two numbers pointed by the two pointers is greater than the target sum, this means that we need a pair with a smaller sum. So, to try more pairs, we can decrement the end-pointer.
2. If the sum of the two numbers pointed by the two pointers is smaller than the target sum, this means that we need a pair with a larger sum. So, to try more pairs, we can increment the start-pointer.

Here is the visual representation of this algorithm for Example-1:



Here is what our algorithm will look like:

class PairWithTargetSum {

public static int[] search(int[] arr, int targetSum) {

int left = 0, right = arr.length - 1;

while (left < right) {

int currentSum = arr[left] + arr[right];

if (currentSum == targetSum)

return new int[] { left, right }; // found the pair

if (targetSum > currentSum)

left++; // we need a pair with a bigger sum

else

right--; // we need a pair with a smaller sum

}

return new int[] { -1, -1 };

}

public static void main(String[] args) {

int[] result = PairWithTargetSum.search(new int[] { 1, 2, 3, 4, 6 }, 6);

System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");

result = PairWithTargetSum.search(new int[] { 2, 5, 9, 11 }, 11);

System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");

}

}

### Time Complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Time-Complexity)

The time complexity of the above algorithm will be O(N)*O*(*N*), where ‘N’ is the total number of elements in the given array.

### Space Complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Space-Complexity)

The algorithm runs in constant space O(1)*O*(1).

## An Alternate approach[**#**](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#An-Alternate-approach)

Instead of using a two-pointer or a binary search approach, we can utilize a **HashTable** to search for the required pair. We can iterate through the array one number at a time. Let’s say during our iteration we are at number ‘X’, so we need to find ‘Y’ such that “X + Y == Target*X*+*Y*==*Target*”. We will do two things here:

1. Search for ‘Y’ (which is equivalent to “Target - X*Target*−*X*”) in the **HashTable**. If it is there, we have found the required pair.
2. Otherwise, insert “X” in the **HashTable**, so that we can search it for the later numbers.

Here is what our algorithm will look like:

import java.util.HashMap;

class PairWithTargetSum {

public static int[] search(int[] arr, int targetSum) {

HashMap<Integer, Integer> nums = new HashMap<>(); // to store numbers and their indices

for (int i = 0; i < arr.length; i++) {

if (nums.containsKey(targetSum - arr[i]))

return new int[] { nums.get(targetSum - arr[i]), i };

else

nums.put(arr[i], i); // put the number and its index in the map

}

return new int[] { -1, -1 }; // pair not found

}

public static void main(String[] args) {

int[] result = PairWithTargetSum.search(new int[] { 1, 2, 3, 4, 6 }, 6);

System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");

result = PairWithTargetSum.search(new int[] { 2, 5, 9, 11 }, 11);

System.out.println("Pair with target sum: [" + result[0] + ", " + result[1] + "]");

}

}

### Time Complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Time-Complexity)

The time complexity of the above algorithm will be O(N)*O*(*N*), where ‘N’ is the total number of elements in the given array.

### Space Complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/xog6q15W9GP#Space-Complexity)

The space complexity will also be O(N)*O*(*N*), as, in the worst case, we will be pushing ‘N’ numbers in the **HashTable**.

# Remove Duplicates (easy)

**We'll cover the following**

* [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/mEEA22L5mNA#Problem-Statement)
* [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/mEEA22L5mNA#Try-it-yourself)
* [Solution](https://www.educative.io/courses/grokking-the-coding-interview/mEEA22L5mNA#Solution)
  + - [Code](https://www.educative.io/courses/grokking-the-coding-interview/mEEA22L5mNA#Code)
  + [Time Complexity](https://www.educative.io/courses/grokking-the-coding-interview/mEEA22L5mNA#Time-Complexity)
  + [Space Complexity](https://www.educative.io/courses/grokking-the-coding-interview/mEEA22L5mNA#Space-Complexity)
* [Similar Questions](https://www.educative.io/courses/grokking-the-coding-interview/mEEA22L5mNA#Similar-Questions)

## Problem Statement

Given an array of sorted numbers, remove all duplicate number instances from it in-place, such that each element appears only once. The relative order of the elements should be kept the same and you should **not use any extra space** so that that the solution have a space complexity of O(1).

Move all the unique elements at the beginning of the array and after moving return the length of the subarray that has no duplicate in it.

**Example 1:**

Input: [2, 3, 3, 3, 6, 9, 9]  
Output: 4  
Explanation: The first four elements after removing the duplicates will be [2, 3, 6, 9].

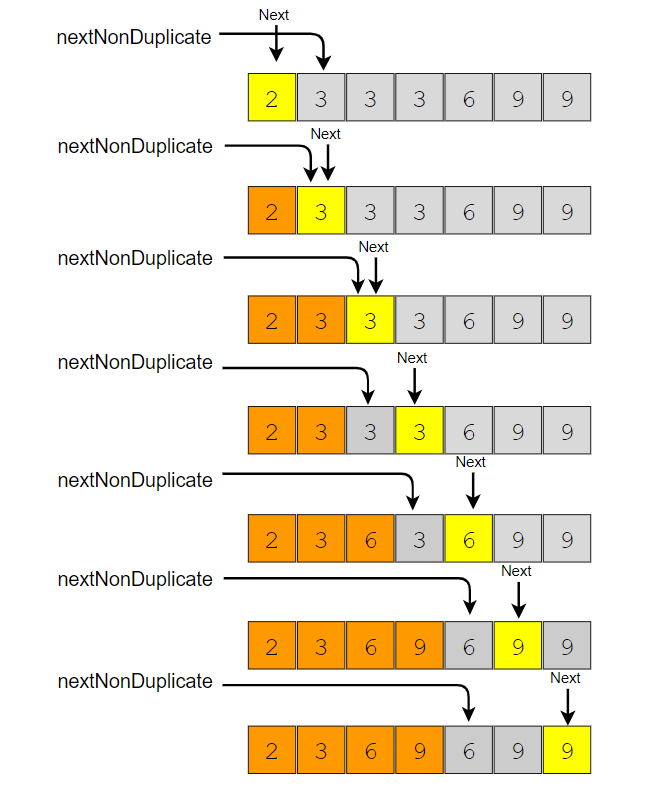
**Example 2:**

Input: [2, 2, 2, 11]  
Output: 2  
Explanation: The first two elements after removing the duplicates will be [2, 11].

## Solution

In this problem, we need to separate the duplicates in-place such that the resultant length of the array remains sorted. As the input array is sorted, therefore, one way to do this is to shift the elements left whenever we encounter duplicates. In other words, we will keep one pointer for iterating the array and one pointer for placing the next non-duplicate number. So our algorithm will be to iterate the array and whenever we see a non-duplicate number we move it next to the last non-duplicate number we’ve seen.

Here is the visual representation of this algorithm for Example-1:



class RemoveDuplicates {

public static int remove(int[] arr) {

int nextNonDuplicate = 1; // index of the next non-duplicate element

for (int i = 0; i < arr.length; i++) {

if (arr[nextNonDuplicate - 1] != arr[i]) {

arr[nextNonDuplicate] = arr[i];

nextNonDuplicate++;

}

}

return nextNonDuplicate;

}

public static void main(String[] args) {

int[] arr = new int[] { 2, 3, 3, 3, 6, 9, 9 };

System.out.println(RemoveDuplicates.remove(arr));

arr = new int[] { 2, 2, 2, 11 };

System.out.println(RemoveDuplicates.remove(arr));

}

}

### Time Complexity

The time complexity of the above algorithm will be O(N)*O*(*N*), where ‘N’ is the total number of elements in the given array.

### Space Complexity

The algorithm runs in constant space O(1)*O*(1).

## Similar Questions

**Problem 1:** Given an unsorted array of numbers and a target ‘key’, remove all instances of ‘key’ in-place and return the new length of the array.

**Example 1:**

Input: [3, 2, 3, 6, 3, 10, 9, 3], Key=3  
Output: 4  
Explanation: The first four elements after removing every 'Key' will be [2, 6, 10, 9].

**Example 2:**

Input: [2, 11, 2, 2, 1], Key=2  
Output: 2  
Explanation: The first two elements after removing every 'Key' will be [11, 1].

**Solution:** This problem is quite similar to our parent problem. We can follow a two-pointer approach and shift numbers left upon encountering the ‘key’. Here is what the code will look like:

class RemoveElement {

public static int remove(int[] arr, int key) {

int nextElement = 0; // index of the next element which is not 'key'

for (int i = 0; i < arr.length; i++) {

if (arr[i] != key) {

arr[nextElement] = arr[i];

nextElement++;

}

}

return nextElement;

}

public static void main(String[] args) {

int[] arr = new int[] { 3, 2, 3, 6, 3, 10, 9, 3 };

System.out.println(RemoveElement.remove(arr, 3));

arr = new int[] { 2, 11, 2, 2, 1 };

System.out.println(RemoveElement.remove(arr, 2));

}

}

**Time and Space Complexity:** The time complexity of the above algorithm will be O(N), where ‘N’ is the total number of elements in the given array.

The algorithm runs in constant space O(1).

# Squaring a Sorted Array (easy)

**We'll cover the following**

* [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Problem-Statement)
* [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Try-it-yourself)
* [Solution](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Solution)
  + - [Code](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Code)
  + [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Time-complexity)
  + [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Space-complexity)

## Problem Statement[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Problem-Statement)

Given a sorted array, create a new array containing **squares of all the numbers of the input array** in the sorted order.

**Example 1:**

Input: [-2, -1, 0, 2, 3]  
Output: [0, 1, 4, 4, 9]

**Example 2:**

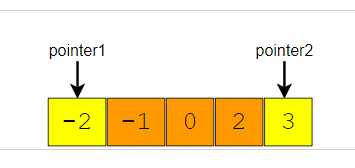
Input: [-3, -1, 0, 1, 2]  
Output: [0, 1, 1, 4, 9]

## Solution[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Solution)

This is a straightforward question. The only trick is that we can have negative numbers in the input array, which will make it a bit difficult to generate the output array with squares in sorted order.

An easier approach could be to first find the index of the first non-negative number in the array. After that, we can use **Two Pointers** to iterate the array. One pointer will move forward to iterate the non-negative numbers, and the other pointer will move backward to iterate the negative numbers. At any step, whichever number gives us a bigger square will be added to the output array. For the above-mentioned Example-1, we will do something like this:

Since the numbers at both ends can give us the largest square, an alternate approach could be to use two pointers starting at both ends of the input array. At any step, whichever pointer gives us the bigger square, we add it to the result array and move to the next/previous number according to the pointer. For the above-mentioned Example-1, we will do something like this:



class SortedArraySquares {

public static int[] makeSquares(int[] arr) {

int n = arr.length;

int[] squares = new int[n];

int highestSquareIdx = n - 1;

int left = 0, right = arr.length - 1;

while (left <= right) {

int leftSquare = arr[left] \* arr[left];

int rightSquare = arr[right] \* arr[right];

if (leftSquare > rightSquare) {

squares[highestSquareIdx--] = leftSquare;

left++;

} else {

squares[highestSquareIdx--] = rightSquare;

right--;

}

}

return squares;

}

public static void main(String[] args) {

int[] result = SortedArraySquares.makeSquares(new int[] { -2, -1, 0, 2, 3 });

for (int num : result)

System.out.print(num + " ");

System.out.println();

result = SortedArraySquares.makeSquares(new int[] { -3, -1, 0, 1, 2 });

for (int num : result)

System.out.print(num + " ");

System.out.println();

}

}

### Time complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Time-complexity)

The above algorithm’s time complexity will be O(N) as we are iterating the input array only once.

### Space complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/R1ppNG3nV9R#Space-complexity)

The above algorithm’s space complexity will also be O(N); this space will be used for the output array.

# Triplet Sum to Zero (medium)

**We'll cover the following**

* [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Problem-Statement)
* [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Try-it-yourself)
* [Solution](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Code)
  + [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Time-complexity)
  + [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Space-complexity)

### Problem Statement [**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Problem-Statement)

Given an array of unsorted numbers, find all **unique triplets in it that add up to zero**.

**Example 1:**

Input: [-3, 0, 1, 2, -1, 1, -2]  
Output: [-3, 1, 2], [-2, 0, 2], [-2, 1, 1], [-1, 0, 1]  
Explanation: There are four unique triplets whose sum is equal to zero.

**Example 2:**

Input: [-5, 2, -1, -2, 3]  
Output: [[-5, 2, 3], [-2, -1, 3]]  
Explanation: There are two unique triplets whose sum is equal to zero.

## Solution[**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Solution)

This problem follows the **Two Pointers** pattern and shares similarities with [Pair with Target Sum](https://www.educative.io/collection/page/5668639101419520/5671464854355968/6618310940557312/). A couple of differences are that the input array is not sorted and instead of a pair we need to find triplets with a target sum of zero.

To follow a similar approach, first, we will sort the array and then iterate through it taking one number at a time. Let’s say during our iteration we are at number ‘X’, so we need to find ‘Y’ and ‘Z’ such that X + Y + Z == 0*X*+*Y*+*Z*==0. At this stage, our problem translates into finding a pair whose sum is equal to “-X−*X*” (as from the above equation Y + Z == -X*Y*+*Z*==−*X*).

Another difference from [Pair with Target Sum](https://www.educative.io/collection/page/5668639101419520/5671464854355968/6618310940557312/) is that we need to find all the unique triplets. To handle this, we have to skip any duplicate number. Since we will be sorting the array, so all the duplicate numbers will be next to each other and are easier to skip.

import java.util.\*;

class TripletSumToZero {

public static List<List<Integer>> searchTriplets(int[] arr) {

Arrays.sort(arr);

List<List<Integer>> triplets = new ArrayList<>();

for (int i = 0; i < arr.length - 2; i++) {

if (i > 0 && arr[i] == arr[i - 1]) // skip same element to avoid duplicate triplets

continue;

searchPair(arr, -arr[i], i + 1, triplets);

}

return triplets;

}

private static void searchPair(int[] arr, int targetSum, int left, List<List<Integer>> triplets) {

int right = arr.length - 1;

while (left < right) {

int currentSum = arr[left] + arr[right];

if (currentSum == targetSum) { // found the triplet

triplets.add(Arrays.asList(-targetSum, arr[left], arr[right]));

left++;

right--;

while (left < right && arr[left] == arr[left - 1])

left++; // skip same element to avoid duplicate triplets

while (left < right && arr[right] == arr[right + 1])

right--; // skip same element to avoid duplicate triplets

} else if (targetSum > currentSum)

left++; // we need a pair with a bigger sum

else

right--; // we need a pair with a smaller sum

}

}

public static void main(String[] args) {

System.out.println(TripletSumToZero.searchTriplets(new int[] { -3, 0, 1, 2, -1, 1, -2 }));

System.out.println(TripletSumToZero.searchTriplets(new int[] { -5, 2, -1, -2, 3 }));

}

}

### Time complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Time-complexity)

Sorting the array will take O(N \* logN). The searchPair() function will take O(N)*O*(*N*). As we are calling searchPair() for every number in the input array, this means that overall searchTriplets() will take O(N \* logN + N^2) which is asymptotically equivalent to O(N^2).

### Space complexity[**#**](https://www.educative.io/courses/grokking-the-coding-interview/gxk639mrr5r#Space-complexity)

Ignoring the space required for the output array, the space complexity of the above algorithm will be O(N) which is required for sorting.

# Triplet Sum Close to Target (medium)

**We'll cover the following**

* [Problem Statement](https://www.educative.io/courses/grokking-the-coding-interview/3YlQz7PE7OA#Problem-Statement)
* [Try it yourself](https://www.educative.io/courses/grokking-the-coding-interview/3YlQz7PE7OA#Try-it-yourself)
* [Solution](https://www.educative.io/courses/grokking-the-coding-interview/3YlQz7PE7OA#Solution)
  + [Code](https://www.educative.io/courses/grokking-the-coding-interview/3YlQz7PE7OA#Code)
* [Time complexity](https://www.educative.io/courses/grokking-the-coding-interview/3YlQz7PE7OA#Time-complexity)
* [Space complexity](https://www.educative.io/courses/grokking-the-coding-interview/3YlQz7PE7OA#Space-complexity)

### Problem Statement

Given an array of unsorted numbers and a target number, find a **triplet in the array whose sum is as close to the target number as possible**, return the sum of the triplet. If there are more than one such triplet, return the sum of the triplet with the smallest sum.

**Example 1:**

Input: [-2, 0, 1, 2], target=2  
Output: 1  
Explanation: The triplet [-2, 1, 2] has the closest sum to the target.

**Example 2:**

Input: [-3, -1, 1, 2], target=1  
Output: 0  
Explanation: The triplet [-3, 1, 2] has the closest sum to the target.

**Example 3:**

Input: [1, 0, 1, 1], target=100  
Output: 3  
Explanation: The triplet [1, 1, 1] has the closest sum to the target.

### Solution

This problem follows the **Two Pointers** pattern and is quite similar to [Triplet Sum to Zero](https://www.educative.io/collection/page/5668639101419520/5671464854355968/5679549973004288/).

We can follow a similar approach to iterate through the array, taking one number at a time. At every step, we will save the difference between the triplet and the target number, so that in the end, we can return the triplet with the closest sum.

#### Code

Here is what our algorithm will look like:

import java.util.\*;

class TripletSumCloseToTarget {

public static int searchTriplet(int[] arr, int targetSum) {

if (arr == null || arr.length < 3)

throw new IllegalArgumentException();

Arrays.sort(arr);

int smallestDifference = Integer.MAX\_VALUE;

for (int i = 0; i < arr.length - 2; i++) {

int left = i + 1, right = arr.length - 1;

while (left < right) {

// comparing the sum of three numbers to the 'targetSum' can cause overflow

// so, we will try to find a target difference

int targetDiff = targetSum - arr[i] - arr[left] - arr[right];

if (targetDiff == 0) // we've found a triplet with an exact sum

return targetSum; // return sum of all the numbers

// the second part of the above 'if' is to handle the smallest sum when we have more than one solution

if (Math.abs(targetDiff) < Math.abs(smallestDifference)

|| (Math.abs(targetDiff) == Math.abs(smallestDifference) && targetDiff > smallestDifference))

smallestDifference = targetDiff; // save the closest and the biggest difference

if (targetDiff > 0)

left++; // we need a triplet with a bigger sum

else

right--; // we need a triplet with a smaller sum

}

}

return targetSum - smallestDifference;

}

public static void main(String[] args) {

System.out.println(TripletSumCloseToTarget.searchTriplet(new int[] { -2, 0, 1, 2 }, 2));

System.out.println(TripletSumCloseToTarget.searchTriplet(new int[] { -3, -1, 1, 2 }, 1));

System.out.println(TripletSumCloseToTarget.searchTriplet(new int[] { 1, 0, 1, 1 }, 100));

}

}

### Time complexity

Sorting the array will take O(N\* logN). Overall, the function will take O(N \* logN + N^2)*O*(*N*∗*logN*+*N*2), which is asymptotically equivalent to O(N^2)*O*(*N*2).

### Space complexity

The above algorithm’s space complexity will be O(N)*O*(*N*), which is required for sorting.