# Two Pointers

## 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 $O(N^2)$ where 'N' is the number of elements in the input array.

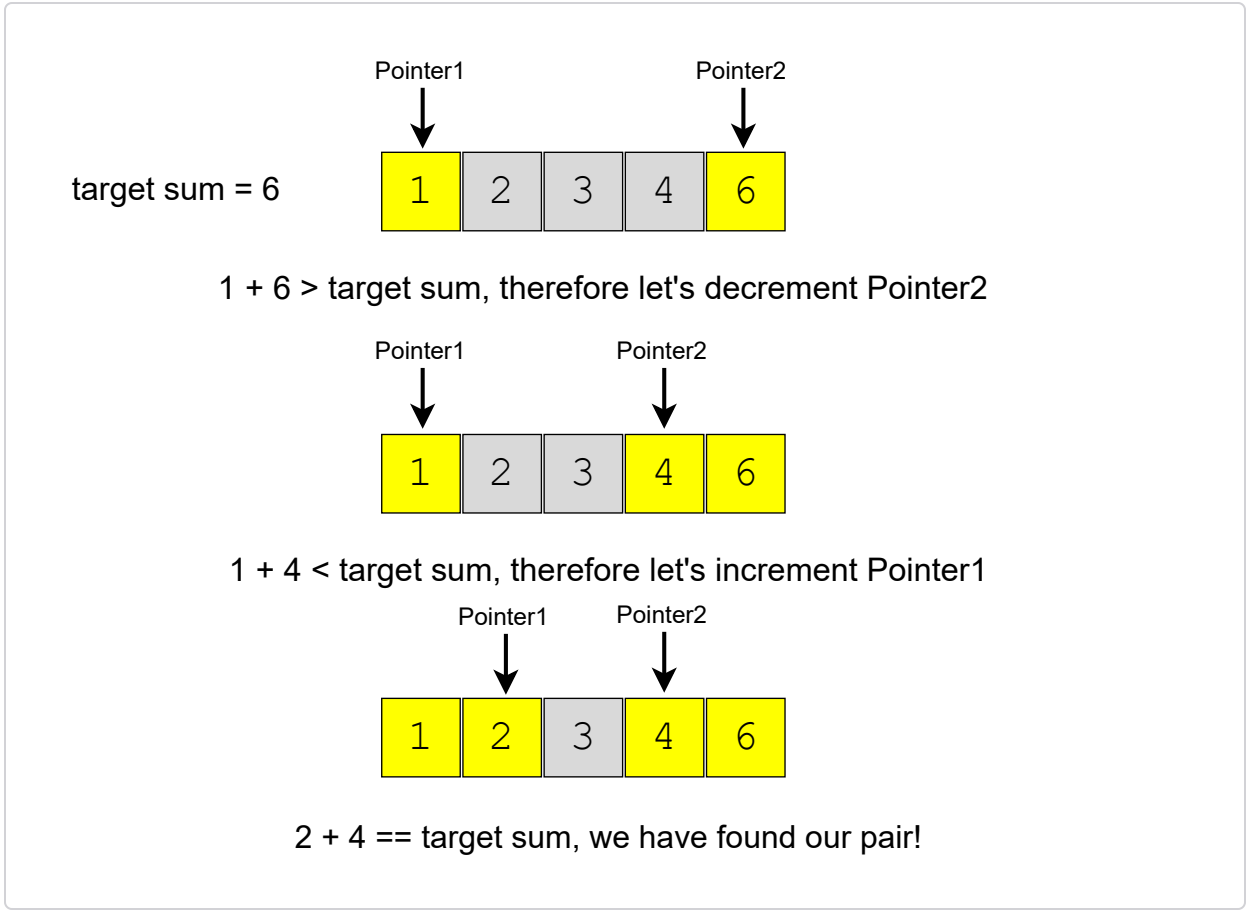
Given that the input array is sorted, an efficient way would be to start with one pointer in the beginning and another pointer at the end. At every step, we will see if the numbers pointed by the two pointers add up to the target sum. If they do not, we will do one of two things:

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 $O(N)$.

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

# Pair with Target Sum (easy)

## 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
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <iostream>
#include <vector>

class PairWithTargetSum {
 public:
  static pair<int, int> search(const vector<int>& arr, int targetSum) {
    // TODO: Write your code here
    return make_pair(-1, -1);
  }
};

```

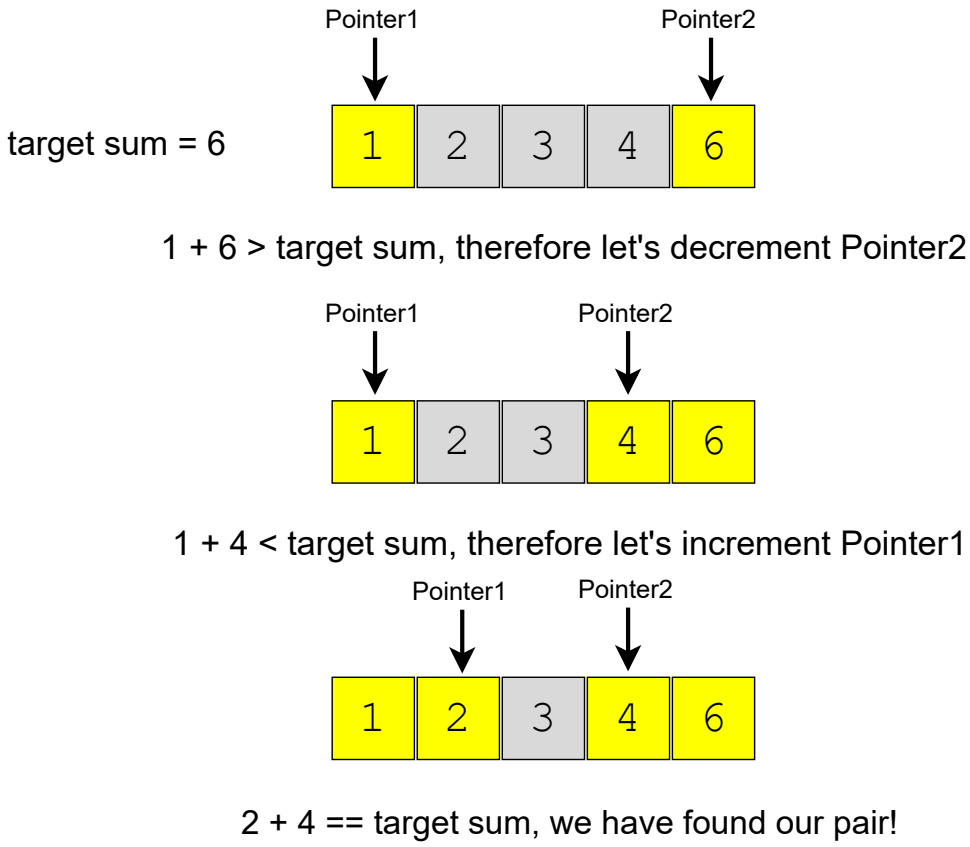
## Solution #

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 $O(N * logN)$. 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:




Code #

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <vector>

class PairWithTargetSum {
public:
  static pair<int, int> search(const vector<int> &arr, int targetSum) {
    int left = 0, right = arr.size() - 1;
    while (left < right) {
      int currentSum = arr[left] + arr[right];
      if (currentSum == targetSum) { // found the pair
        return make_pair(left, right);
      }

      if (targetSum > currentSum)
        left++; // we need a pair with a bigger sum
      else
        right--; // we need a pair with a smaller sum
    }
    return make_pair(-1, -1);
  }
};

int main(int argc, char *argv[]) {
  auto result = PairWithTargetSum::search(vector<int>{1, 2, 3, 4, 6}, 6);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
  result = PairWithTargetSum::search(vector<int>{2, 5, 9, 11}, 11);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
}
```

Output 0.892s

```
Pair with target sum: [1, 3]
Pair with target sum: [0, 2]
```

### Time Complexity #

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

### Space Complexity #

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

## 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 "$X + Y == Target$". We will do two things here:

1. Search for 'Y' (which is equivalent to "$Target - X$") 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:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <unordered_map>
#include <vector>

class PairWithTargetSum {
 public:
  static pair<int, int> search(const vector<int>& arr, int targetSum) {
    unordered_map<int, int> nums;  // to store number and its index
    for (int i = 0; i < arr.size(); i++) {
      if (nums.find(targetSum - arr[i]) != nums.end()) {
        return make_pair(nums[targetSum - arr[i]], i);
      } else {
        nums[arr[i]] = i;  // put the number and its index in the map
      }
    }
    return make_pair(-1, -1);  // pair not found
  }
};

int main(int argc, char* argv[]) {
  auto result = PairWithTargetSum::search(vector<int>{1, 2, 3, 4, 6}, 6);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
  result = PairWithTargetSum::search(vector<int>{2, 5, 9, 11}, 11);
  cout << "Pair with target sum: [" << result.first << ", " << result.second << "]" << endl;
}
```

Output 1.551s

```
Pair with target sum: [1, 3]
Pair with target sum: [0, 2]
```

### Time Complexity #

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

### Space Complexity #

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

# Remove Duplicates (easy)

## Problem Statement #

Given an array of sorted numbers, **remove all duplicates** from it. You should **not use any extra space**; after removing the duplicates in-place return the new length of the array.

**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].
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

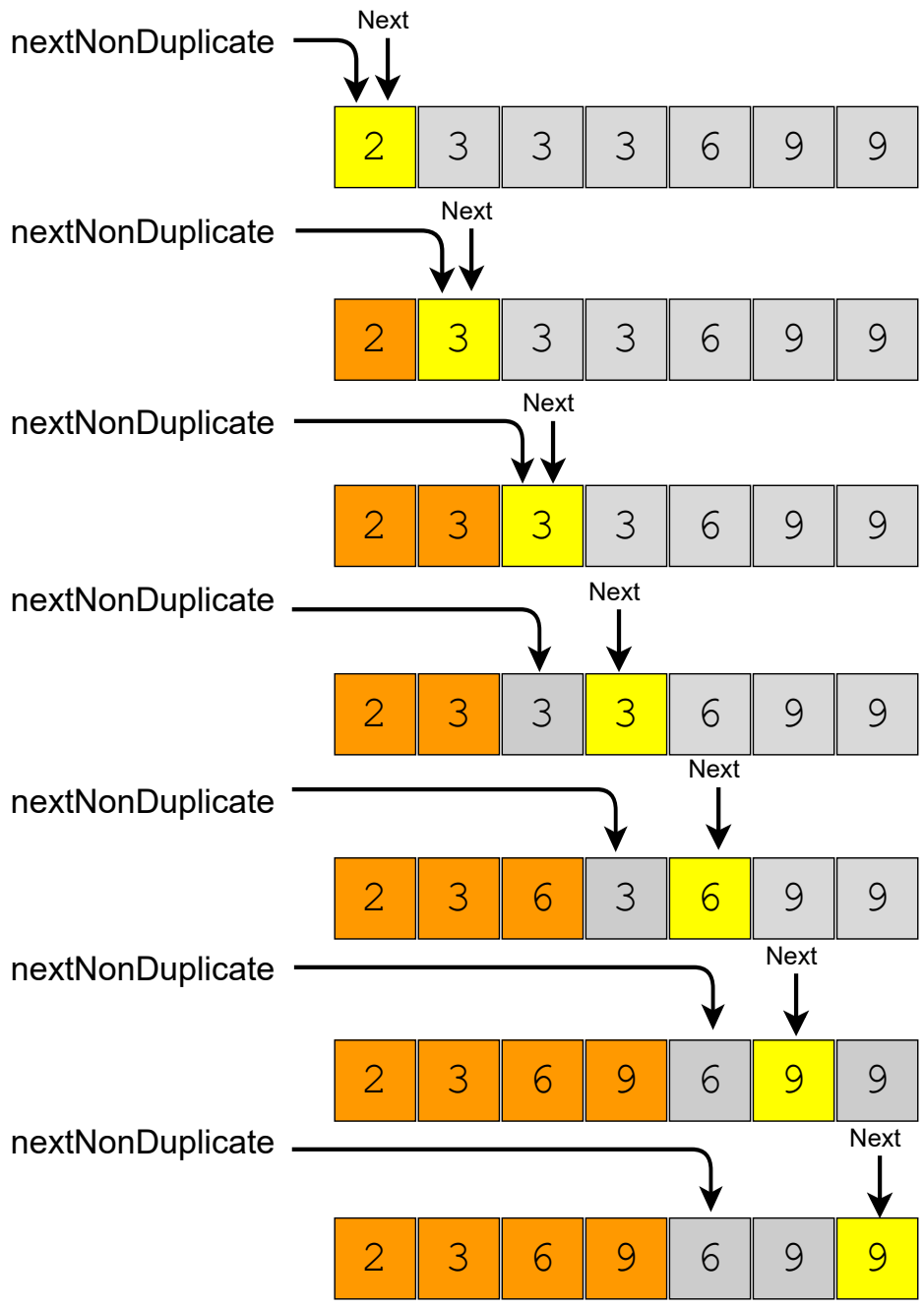
```
using namespace std;

#include <iostream>
#include <vector>

class RemoveDuplicates {
 public:
  static int remove(vector<int>& arr) {
    // TODO: Write your code here
    return -1;
  }
};

```

## Solution #

In this problem, we need to remove 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:




Code #

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```cpp
using namespace std;

#include <iostream>
#include <vector>

class RemoveDuplicates {
 public:
  static int remove(vector<int>& arr) {
    int nextNonDuplicate = 1;  // index of the next non-duplicate element
    for (int i = 1; i < arr.size(); i++) {
      if (arr[nextNonDuplicate - 1] != arr[i]) {
        arr[nextNonDuplicate] = arr[i];
        nextNonDuplicate++;
      }
    }

    return nextNonDuplicate;
  }
};

int main(int argc, char* argv[]) {
  vector<int> arr = {2, 3, 3, 3, 6, 9, 9};
  cout << "Array new length: " << RemoveDuplicates::remove(arr) << endl;

  arr = vector<int>{2, 2, 2, 11};
  cout << "Array new length: " << RemoveDuplicates::remove(arr) << endl;
}

```

Output 0.919s

```
Array new length: 4
Array new length: 2
```

### Time Complexity #

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

### Space Complexity #

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

---

## 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:

Java | Python3 | C++ | JS

```
using namespace std;

#include <iostream>
#include <vector>

class RemoveElement {
 public:
  static int remove(vector<int>& arr, int key) {
    int nextElement = 0;  // index of the next element which is not 'key'
    for (int i = 0; i < arr.size(); i++) {
      if (arr[i] != key) {
        arr[nextElement] = arr[i];
        nextElement++;
      }
    }

    return nextElement;
  }
};

int main(int argc, char* argv[]) {
  vector<int> arr = {3, 2, 3, 6, 3, 10, 9, 3};
  cout << "Array new length: " << RemoveElement::remove(arr, 3) << endl;

  arr = vector<int>{2, 11, 2, 2, 1};
  cout << "Array new length: " << RemoveElement::remove(arr, 2) << endl;
}

```

```
Output                                                                1.301s

  Array new length: 4
  Array new length: 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)

## Problem Statement #

Given a sorted array, create a new array containing **squares of all the number 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]
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

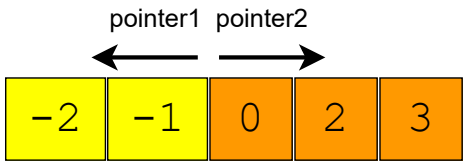
```
using namespace std;

#include <iostream>
#include <vector>

class SortedArraySquares {
 public:
  static vector<int> makeSquares(const vector<int>& arr) {
    int n = arr.size();
    vector<int> squares(n);
    // TODO: Write your code here
    return squares;
  }
};

```

| Result | Input | Expected Output | Actual Output | Reason |
|---|---|---|---|---|
| ✗ | makeSquares([-2, -1, 0, 2, 3]) | [0, 1, 4, 4, 9] | [0, 0, 0, 0, 0] | Incorrect Output |
| ✗ | makeSquares([-3, -1, 0, 1, 2]) | [0, 1, 1, 4, 9] | [0, 0, 0, 0, 0] | Incorrect Output |

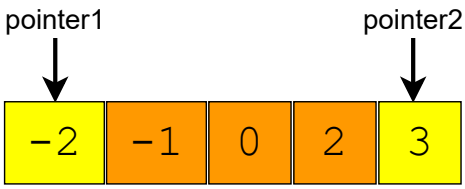
## Solution #

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 the ends can give us the largest square, an alternate approach could be to use two pointers starting at both the 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:




### Code #

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
using namespace std;

#include <iostream>
#include <vector>

class SortedArraySquares {
 public:
```

```cpp
  public:
    static vector<int> makeSquares(const vector<int>& arr) {
      int n = arr.size();
      vector<int> squares(n);
      int highestSquareIdx = n - 1;
      int left = 0, right = n - 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;
    }
};

int main(int argc, char* argv[]) {
  vector<int> result = SortedArraySquares::makeSquares(vector<int>{-2, -1, 0, 2, 3});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;

  result = SortedArraySquares::makeSquares(vector<int>{-3, -1, 0, 1, 2});
  for (auto num : result) {
    cout << num << " ";
  }
  cout << endl;
}
```

Output 1.326s

```
0 1 4 4 9
0 1 1 4 9
```

### Time complexity #

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

### Space complexity #

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

# Triplet Sum to Zero (medium)

## 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.
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class TripletSumToZero {
 public:
  static vector<vector<int>> searchTriplets(vector<int> &arr) {
    vector<vector<int>> triplets;
    // TODO: Write your code here
    return triplets;
  }
};

```

## Solution #

This problem follows the **Two Pointers** pattern and shares similarities with Pair with Target

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 $X + Y + Z == 0$. At this stage, our problem translates into finding a pair whose sum is equal to "$-X$" (as from the above equation $Y + Z == -X$).

Another difference from Pair with Target Sum 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.

Code #

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class TripletSumToZero {
public:
  static vector<vector<int>> searchTriplets(vector<int> &arr) {
    sort(arr.begin(), arr.end());
    vector<vector<int>> triplets;
    for (int i = 0; i < arr.size() - 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(const vector<int> &arr, int targetSum, int left,
                         vector<vector<int>> &triplets) {
    int right = arr.size() - 1;
    while (left < right) {
      int currentSum = arr[left] + arr[right];
      if (currentSum == targetSum) { // found the pair
        triplets.push_back({-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 ) {
```

```
39          left++; // we need a pair with a bigger sum
40        } else {
41          right--; // we need a pair with a smaller sum
42        }
43      }
44    }
45  };
46
47  int main(int argc, char *argv[]) {
48    vector<int> vec = {-3, 0, 1, 2, -1, 1, -2};
49    auto result = TripletSumToZero::searchTriplets(vec);
50    for (auto vec : result) {
51      cout << "[";
52      for (auto num : vec) {
53        cout << num << " ";
54      }
55      cout << "]";
56    }
57    cout << endl;
58
59    vec = {-5, 2, -1, -2, 3};
60    result = TripletSumToZero::searchTriplets(vec);
61    for (auto vec : result) {
62      cout << "[";
63      for (auto num : vec) {
64        cout << num << " ";
65      }
66      cout << "]";
67    }
68  }
69
```

Output 1.451s

```
 [-3 1 2 ][-2 0 2 ][-2 1 1 ][-1 0 1 ]
 [-5 2 3 ][-2 -1 3 ]
```

### Time complexity #

Sorting the array will take $O(N * logN)$. The `searchPair()` function will take $O(N)$. 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 #

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)

## 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.
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
1  using namespace std;
2
3  #include <algorithm>
4  #include <iostream>
5  #include <limits>
6  #include <vector>
7
8  class TripletSumCloseToTarget {
9   public:
10   static int searchTriplet(vector<int>& arr, int targetSum) {
11     // TODO: Write your code here
12     return -1;
```

0 of 3 Tests Passed

| Result | Input | Expected Output | Actual Output | Reason |
|---|---|---|---|---|
| ✗ | searchTriplet([-2, 0, 1, 2], 2) | 1 | -1 | Incorrect Output |
| ✗ | searchTriplet([-3, -1, 1, 2], 1) | 0 | -1 | Incorrect Output |
| ✗ | searchTriplet([1, 0, 1, 1], 100) | 3 | -1 | Incorrect Output |

4.705s

## Solution #

This problem follows the **Two Pointers** pattern and is quite similar to Triplet Sum to Zero.

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:

Java | Python3 | C++ | JS

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <limits>
#include <vector>

class TripletSumCloseToTarget {
 public:
  static int searchTriplet(vector<int>& arr, int targetSum) {
    sort(arr.begin(), arr.end());
    int smallestDifference = numeric_limits<int>::max();
    for (int i = 0; i < arr.size() - 2; i++) {
      int left = i + 1, right = arr.size() - 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 - targetDiff;  // return sum of all the numbers
        }

        if (abs(targetDiff) < abs(smallestDifference)) {
          smallestDifference = targetDiff;  // save the closest difference
          }

        if (targetDiff > 0) {
           left ++ l   // We need a triplet with a bigger sum
```

```
        } else {
          right--;  // we need a triplet with a smaller sum
        }
      }
    }
    return targetSum - smallestDifference;
  }
};

int main(int argc, char* argv[]) {
  vector<int> vec = {-2, 0, 1, 2};
  cout << TripletSumCloseToTarget::searchTriplet(vec, 2) << endl;
  vec = {-3, -1, 1, 2};
  cout << TripletSumCloseToTarget::searchTriplet(vec, 1) << endl;
  vec = {1, 0, 1, 1};
  cout << TripletSumCloseToTarget::searchTriplet(vec, 100) << endl;
}

```

Output 1.273s

```
1
0
3
```

### Time complexity #

Sorting the array will take $O(N * logN)$. Overall `searchTriplet()` will take $O(N * logN + N^2)$, which is asymptotically equivalent to $O(N^2)$.

### Space complexity #

The space complexity of the above algorithm will be $O(N)$ which is required for sorting.

# Triplets with Smaller Sum (medium)

## Problem Statement #

Given an array `arr` of unsorted numbers and a target sum, **count all triplets** in it such that `arr[i] + arr[j] + arr[k] < target` where `i`, `j`, and `k` are three different indices. Write a function to return the count of such triplets.

**Example 1:**

```
Input: [-1, 0, 2, 3], target=3
Output: 2
Explanation: There are two triplets whose sum is less than the target: [-1, 0, 3], [-
1, 0, 2]
```

**Example 2:**

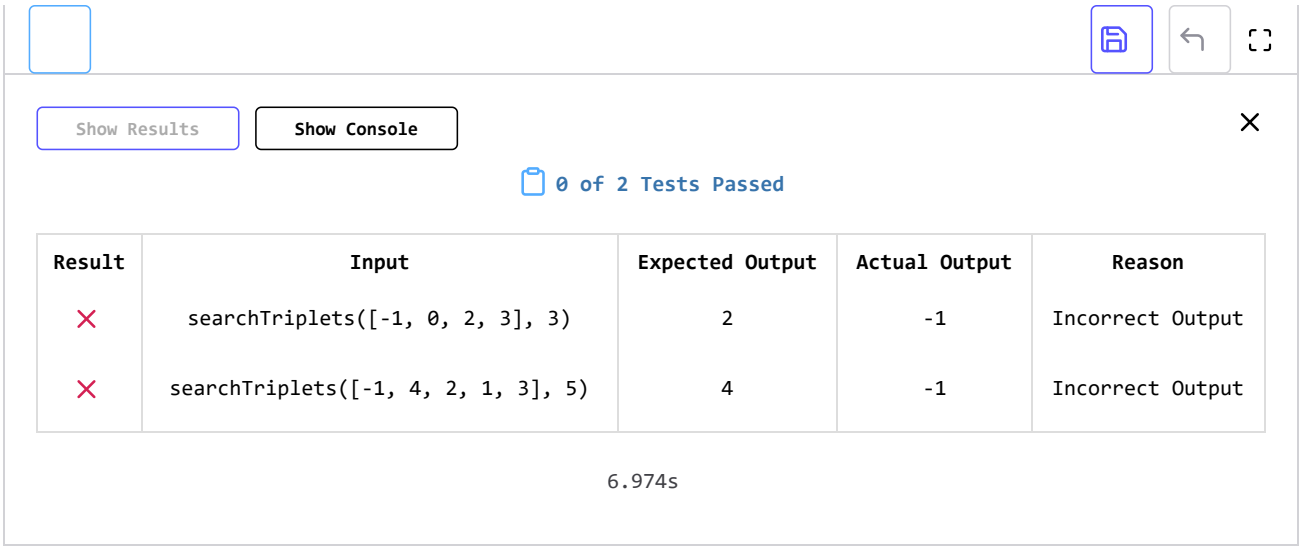
```
Input: [-1, 4, 2, 1, 3], target=5
Output: 4
Explanation: There are four triplets whose sum is less than the target:
   [-1, 1, 4], [-1, 1, 3], [-1, 1, 2], [-1, 2, 3]
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class TripletWithSmallerSum {
 public:
  static int searchTriplets(vector<int> &arr, int target) {
    int count = -1;
    // TODO: Write your code here
    return count;
  }
};

```




## Solution #

This problem follows the **Two Pointers** pattern and shares similarities with Triplet Sum to Zero. The only difference is that, in this problem, we need to find the triplets whose sum is less than the given target. To meet the condition `i != j != k` we need to make sure that each number is not used more than once.

Following a similar approach, first we can 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 $X + Y + Z < target$. At this stage, our problem translates into finding a pair whose sum is less than "$ target - X$" (as from the above equation $Y + Z == target - X$). We can use a similar approach as discussed in Triplet Sum to Zero.

### Code #

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class TripletWithSmallerSum {
 public:
  static int searchTriplets(vector<int> &arr, int target) {
    sort(arr.begin(), arr.end());
    int count = 0;
    for (int i = 0; i < arr.size() - 2; i++) {
      count += searchPair(arr, target - arr[i], i);
    }
    return count;
  }

private :
 static int searchPair ( const vector<int> &arr, int targetSum, int first) {
    int count = 0;
```

```
    int left = first + 1, right = arr.size() - 1;
    while (left < right) {
      if (arr[left] + arr[right] < targetSum) {  // found the triplet
        // since arr[right] >= arr[left], therefore, we can replace arr[right] by any number between
        // left and right to get a sum less than the target sum
        count += right - left;
        left++;
      } else {
        right--;  // we need a pair with a smaller sum
      }
    }
    return count;
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {-1, 0, 2, 3};
  cout << TripletWithSmallerSum::searchTriplets(vec, 3) << endl;
  vec = {-1, 4, 2, 1, 3};
  cout << TripletWithSmallerSum::searchTriplets(vec, 5) << endl;
}
```

Output 1.332s

```
2
4
```

### Time complexity #

Sorting the array will take $O(N * logN)$. The `searchPair()` will take $O(N)$. So, overall `searchTriplets()` will take $O(N * logN + N^2)$, which is asymptotically equivalent to $O(N^2)$.

### 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 if we are not using an in-place sorting algorithm.

## Similar Problems #

**Problem:** Write a function to return the list of all such triplets instead of the count. How will the time complexity change in this case?

**Solution:** Following a similar approach we can create a list containing all the triplets. Here is the code - only the highlighted lines have changed:

Java | Python3 | C++ | JS

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

class TripletWithSmallerSum {
 public:
  static vector<vector<int>> searchTriplets(vector<int> &arr, int target) {
    sort(arr.begin(), arr.end());
    vector<vector<int>> triplets;
    for (int i = 0; i < arr.size() - 2; i++) {
      searchPair(arr, target - arr[i], i, triplets);
    }
    return triplets;
  }

 private:
  static void searchPair(vector<int> &arr, int targetSum, int first,
                         vector<vector<int>> &triplets) {
    int left = first + 1, right = arr.size() - 1;
    while (left < right) {
      if (arr[left] + arr[right] < targetSum) {  // found the triplet
        // since arr[right] >= arr[left], therefore, we can replace arr[right] by any number between
        // left and right to get a sum less than the target sum
        for (int i = right; i > left; i--) {
          triplets.push_back({arr[first], arr[left], arr[i]});
        }
        left++;
      } else {
        right--;  // we need a pair with a smaller sum
      }
    }
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {-1, 0, 2, 3};
  auto result = TripletWithSmallerSum::searchTriplets(vec, 3);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
  cout << endl;

  vec = {-1, 4, 2, 1, 3};
  result = TripletWithSmallerSum::searchTriplets(vec, 5);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
}
```

```
Output                                                                    1.969s

 [-1 0 3 ][-1 0 2 ]
 [-1 1 4 ][-1 1 3 ][-1 1 2 ][-1 2 3 ]
```

Another simpler approach could be to check every triplet of the array with three nested loops and create a list of triplets that meet the required condition.

### Time complexity #

Sorting the array will take $O(N * logN)$. The `searchPair()`, in this case, will take $O(N^2)$; the main `while` loop will run in $O(N)$ but the nested `for` loop can also take $O(N)$ - this will happen when the target sum is bigger than every triplet in the array.

So, overall `searchTriplets()` will take $O(N * logN + N^3)$, which is asymptotically equivalent to $O(N^3)$.

### 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.

# Subarrays with Product Less than a Target (medium)

## Problem Statement #

Given an array with positive numbers and a target number, find all of its contiguous subarrays whose **product is less than the target number**.

**Example 1:**

```
Input: [2, 5, 3, 10], target=30
Output: [2], [5], [2, 5], [3], [5, 3], [10]
Explanation: There are six contiguous subarrays whose product is less than the target.
```

**Example 2:**

```
Input: [8, 2, 6, 5], target=50
Output: [8], [2], [8, 2], [6], [2, 6], [5], [6, 5]
Explanation: There are seven contiguous subarrays whose product is less than the target.
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
using namespace std;

#include <deque>
#include <iostream>
#include <vector>

class SubarrayProductLessThanK {
 public:
  static vector<vector<int>> findSubarrays(const vector<int>& arr, int target) {
    vector<vector<int>> result;
    // TODO: Write your code here
    return result;
  }
};
```

0 of 2 Tests Passed

| Result | Input | Expected Output | Actual Output | Reason |
|---|---|---|---|---|
| ✗ | findSubarrays([2, 5, 3, 10],30) | [[2], [3], [3, 5], [5], [10], [5, 2]] | [] | Incorrect Output |
| ✗ | findSubarrays([8, 2, 6, 5],50) | [[2], [5, 6], [5], [6], [2, 6], [8], [2, | [] | Incorrect Output |

14.297s

## Solution #

This problem follows the **Sliding Window** and the **Two Pointers** pattern and shares similarities with Triplets with Smaller Sum with two differences:

1. In this problem, the input array is not sorted.
2. Instead of finding triplets with sum less than a target, we need to find all subarrays having a product less than the target.

The implementation will be quite similar to Triplets with Smaller Sum.

### Code #

Here is what our algorithm will look like:

```cpp
using namespace std;

#include <deque>
#include <iostream>
#include <vector>

class SubarrayProductLessThanK {
 public:
  static vector<vector<int>> findSubarrays(const vector<int>& arr, int target) {
    vector<vector<int>> result;
    int product = 1, left = 0;
    for (int right = 0; right < arr.size(); right++) {
      product *= arr[right];
      while (product >= target && left < arr.size()) {
        product /= arr[left++];
      }
      // since the product of all numbers from left to right is less than the target therefore,
      // all subarrays from lef to right will have a product less than the target too; to avoid
      // duplicates, we will start with a subarray containing only arr[right] and then extend it
      deque<int> tempList;
      for (int i = right; i >= left; i--) {
        tempList.push_front(arr[i]);
        vector<int> resultVec;
        std::move(std::begin(tempList), std::end(tempList), std::back_inserter(resultVec));
        result.push_back(resultVec);
      }
    }
    return result;
  }
};

int main(int argc, char* argv[]) {
  auto result = SubarrayProductLessThanK::findSubarrays(vector<int>{2, 5, 3, 10}, 30);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
  cout << endl;

  result = SubarrayProductLessThanK::findSubarrays(vector<int>{8, 2, 6, 5}, 50);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
}

```

```
[2 ][5 ][2 5 ][3 ][5 3 ][10 ]
[8 ][2 ][8 2 ][6 ][2 6 ][5 ][6 5 ]
```

### Time complexity #

The main `for-loop` managing the sliding window takes $O(N)$ but creating subarrays can take up to $O(N^2)$ in the worst case. Therefore overall, our algorithm will take $O(N^3)$.

### Space complexity #

Ignoring the space required for the output list, the algorithm runs in $O(N)$ space which is used for the temp list.

Can you try estimating how much space will be required for the output list?

Hide Hint

The worst case will happen when every subarray has a product less than the target!

So the question will be, how many contiguous subarray an array can have?

It is definately not all Permutations of the given array, is it all Combinations of the given array?

It is not all the Combinations of all elements of the array!

For an array with distinct elements, finding all of its contiguous subarrays is like finding the number of ways to choose two indices i and j in the array such that `i <= j`.

If there are a total of `n` elements in the array, here is how we can count all the contiguous subarrays:

- When `i = 0`, `j` can have any value from '0' to 'n-1', giving a total of 'n' choices.
- When `i = 1`, `j` can have any value from '1' to 'n-1', giving a total of 'n-1' choices.
- Similarly, when `i = 2`, `j` can have 'n-2' choices.

  ...

  ...
- When `i = n-1`, `j` can only have '1' choice.

Let's combine all the choices:

```
n + (n-1) + (n-2) + ... 3 + 2 + 1
```

Which gives us a total of: $n * (n + 1)/2$

So, at the most, we need a space of $O(n^2)$ for all the output lists.

# Dutch National Flag Problem (medium)

## Problem Statement #

Given an array containing 0s, 1s and 2s, sort the array in-place. You should treat numbers of the array as objects, hence, we can't count 0s, 1s, and 2s to recreate the array.

The flag of the Netherlands consists of three colors: red, white and blue; and since our input array also consists of three different numbers that is why it is called Dutch National Flag problem.

**Example 1:**

```
Input: [1, 0, 2, 1, 0]
Output: [0 0 1 1 2]
```

**Example 2:**

```
Input: [2, 2, 0, 1, 2, 0]
Output: [0 0 1 2 2 2 ]
```

## Try it yourself #

Try solving this question here:

Java | Python3 | JS | C++

```
1  using namespace std;
2
3  #include <iostream>
4  #include <vector>
5
6  class DutchFlag {
7   public:
8    static void sort(vector<int> &arr) {
9      // TODO: Write your code here
10   }
11 };
12
```

0 of 2 Tests Passed

| Result | Input | Expected Output | Actual Output | Reason |
|---|---|---|---|---|
| ✗ | sort([1, 0, 2, 1, 0]) | [0, 0, 1, 1, 2] | [1, 0, 2, 1, 0] | Incorrect Output |
| ✗ | sort([2, 2, 0, 1, 2, 0]) | [0, 0, 1, 2, 2, 2] | [2, 2, 0, 1, 2, 0] | Incorrect Output |

7.666s

## Solution #

The brute force solution will be to use an in-place sorting algorithm like Heapsort which will take $O(N * logN)$. Can we do better than this? Is it possible to sort the array in one iteration?

We can use a **Two Pointers** approach while iterating through the array. Let's say the two pointers are called `low` and `high` which are pointing to the first and the last element of the array respectively. So while iterating, we will move all 0s before `low` and all 2s after `high` so that in the end, all 1s will be between `low` and `high`.

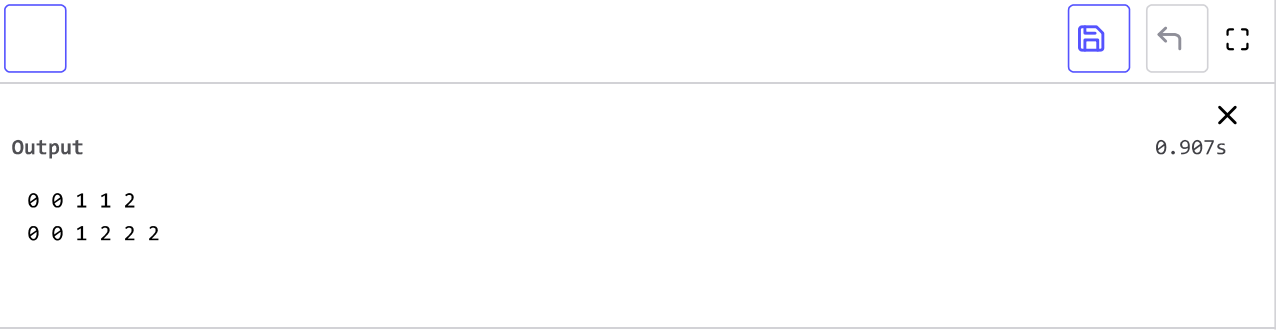
### Code #

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
using namespace std;

#include <iostream>
#include <vector>

class DutchFlag {
 public:
  static void sort(vector<int> &arr) {
    // all elements < low are 0 and all elements > high are 2
    // all elements from >= low < i are 1
    int low = 0, high = arr.size() - 1;
    for (int i = 0; i <= high;) {
      if (arr[i] == 0) {
        swap(arr, i, low);
        i++;
        low++;
      } else if (arr[i] == 1) {
        i++;
      } else {  // the case for arr[i] == 2
        swap(arr, i, high);
        // decrement 'high' only, after the swap the number at index 'i' could be 0, 1 or 2
        high--;
      }
    }
  }

 private:
  static void swap(vector<int> &arr, int i, int j ){
     int temp = arr[ i ] ;
     arr[ i ] = arr[ j ] ;
```

```
31      arr[j] = temp;
32    }
33  };
34
35  int main(int argc, char *argv[]) {
36    vector<int> arr = {1, 0, 2, 1, 0};
37    DutchFlag::sort(arr);
38    for (auto num : arr) {
39      cout << num << " ";
40    }
41    cout << endl;
42
43    arr = vector<int>{2, 2, 0, 1, 2, 0};
44    DutchFlag::sort(arr);
45    for (auto num : arr) {
46      cout << num << " ";
47    }
48  }
49
```



```
Output                                                    0.907s

 0 0 1 1 2
 0 0 1 2 2 2
```

### Time complexity #

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

### Space complexity #

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

# Solution Review: Problem Challenge 1

## Quadruple Sum to Target (medium) #

Given an array of unsorted numbers and a target number, find all **unique quadruplets** in it, whose **sum is equal to the target number**.

**Example 1:**

```
Input: [4, 1, 2, -1, 1, -3], target=1
Output: [-3, -1, 1, 4], [-3, 1, 1, 2]
Explanation: Both the quadruplets add up to the target.
```

**Example 2:**

```
Input: [2, 0, -1, 1, -2, 2], target=2
Output: [-2, 0, 2, 2], [-1, 0, 1, 2]
Explanation: Both the quadruplets add up to the target.
```

## Solution #

This problem follows the **Two Pointers** pattern and shares similarities with Triplet Sum to Zero.

We can follow a similar approach to iterate through the array, taking one number at a time. At every step during the iteration, we will search for the quadruplets similar to Triplet Sum to Zero whose sum is equal to the given target.

### Code #

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

```
using namespace std;

#include <algorithm>
#include <iostream>
#include <vector>

class QuadrupleSumToTarget {
 public:
  static vector<vector<int>> searchQuadruplets(vector<int> &arr, int target) {
    sort(arr.begin(), arr.end());
    vector<vector<int>> quadruplets;
     for(int i = 0; i < arr.size() - 3 ; i++ ) {
          if ( i > 0 && arr[ i ] == arr[ i-1 ] ) {   // skip same element to avoid duplicate element
               continue;
        }
```

```
      for (int j = i + 1; j < arr.size() - 2; j++) {
        if (j > i + 1 &&
            arr[j] == arr[j - 1]) {  // skip same element to avoid duplicate quadruplets
          continue;
        }
        searchPairs(arr, target, i, j, quadruplets);
      }
    }
    return quadruplets;
  }

 private:
  static void searchPairs(const vector<int> &arr, int targetSum, int first, int second,
                          vector<vector<int>> &quadruplets) {
    int left = second + 1;
    int right = arr.size() - 1;
    while (left < right) {
      int sum = arr[first] + arr[second] + arr[left] + arr[right];
      if (sum == targetSum) {  // found the quadruplet
        quadruplets.push_back({arr[first], arr[second], arr[left], arr[right]});
        left++;
        right--;
        while (left < right && arr[left] == arr[left - 1]) {
          left++;  // skip same element to avoid duplicate quadruplets
        }
        while (left < right && arr[right] == arr[right + 1]) {
          right--;  // skip same element to avoid duplicate quadruplets
        }
      } else if (sum < targetSum) {
        left++;  // we need a pair with a bigger sum
      } else {
        right--;  // we need a pair with a smaller sum
      }
    }
  }
};

int main(int argc, char *argv[]) {
  vector<int> vec = {4, 1, 2, -1, 1, -3};
  auto result = QuadrupleSumToTarget::searchQuadruplets(vec, 1);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
  cout << endl;

  vec = {2, 0, -1, 1, -2, 2};
  result = QuadrupleSumToTarget::searchQuadruplets(vec, 2);
  for (auto vec : result) {
    cout << "[";
    for (auto num : vec) {
      cout << num << " ";
    }
    cout << "]";
  }
}

```

```
[-3 -1 1 4 ][-3 1 1 2 ]
[-2 0 2 2 ][-1 0 1 2 ]
```

### Time complexity #

Sorting the array will take $O(N * logN)$. Overall `searchQuadruplets()` will take $O(N * logN + N^3)$, which is asymptotically equivalent to $O(N^3)$.

### Space complexity #

The space complexity of the above algorithm will be $O(N)$ which is required for sorting.

# Solution Review: Problem Challenge 2

## Comparing Strings containing Backspaces (medium) #

Given two strings containing backspaces (identified by the character '#'), check if the two strings are equal.

**Example 1:**

```
Input: str1="xy#z", str2="xzz#"
Output: true
Explanation: After applying backspaces the strings become "xz" and "xz" respectively.
```

**Example 2:**

```
Input: str1="xy#z", str2="xyz#"
Output: false
Explanation: After applying backspaces the strings become "xz" and "xy" respectively.
```

**Example 3:**

```
Input: str1="xp#", str2="xyz##"
Output: true
Explanation: After applying backspaces the strings become "x" and "x" respectively.
In "xyz##", the first '#' removes the character 'z' and the second '#' removes the characte
r 'y'.
```

**Example 4:**

```
Input: str1="xywrrmp", str2="xywrrmu#p"
Output: true
Explanation: After applying backspaces the strings become "xywrrmp" and "xywrrmp" respectiv
ely.
```

## Solution #

To compare the given strings, first, we need to apply the backspaces. An efficient way to do this would be from the end of both the strings. We can have separate pointers, pointing to the last element of the given strings. We can start comparing the characters pointed out by both the pointers to see if the strings are equal. If, at any stage, the character pointed out by any of the pointers is a backspace ('#'), we will skip and apply the backspace until we have a valid character available for comparison.

## Code #

Here is what our algorithm will look like:

```
using namespace std;

#include <iostream>
#include <string>

class BackspaceCompare {
 public:
  static bool compare(const string &str1, const string &str2) {
    // use two pointer approach to compare the strings
    int index1 = str1.length() - 1, index2 = str2.length() - 1;
    while (index1 >= 0 || index2 >= 0) {
      int i1 = getNextValidCharIndex(str1, index1);
      int i2 = getNextValidCharIndex(str2, index2);

      if (i1 < 0 && i2 < 0) {  // reached the end of both the strings
        return true;
      }

      if (i1 < 0 || i2 < 0) {  // reached the end of one of the strings
        return false;
      }

      if (str1[i1] != str2[i2]) {  // check if the characters are equal
        return false;
      }

      index1 = i1 - 1;
      index2 = i2 - 1;
    }

    return true;
  }

 private:
  static int getNextValidCharIndex(const string &str, int index) {
    int backspaceCount = 0;
    while (index >= 0) {
      if (str[index] == '#') {  // found a backspace character
        backspaceCount++;
      } else if (backspaceCount > 0) {  // a non-backspace character
        backspaceCount--;
      } else {
        break;
      }

      index--;  // skip a backspace or a valid character
    }
    return index;
  }
};

int main(int argc, char *argv[]) {
  cout << BackspaceCompare::compare("xy#z", "xzz#") << endl;
  cout << BackspaceCompare::compare("xy#z", "xyz#") << endl;
  cout << BackspaceCompare::compare("xp#", "xyz##") << endl;
  cout << BackspaceCompare::compare("xywrrmp", "xywrrmu#p") << endl;
}

```

```
Output                                                    0.859s

 1
 0
 1
 1
```

### Time complexity #

The time complexity of the above algorithm will be $O(M + N)$ where 'M' and 'N' are the lengths of the two input strings respectively.

### Space complexity #

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

# Solution Review: Problem Challenge 3

## Minimum Window Sort (medium) #

Given an array, find the length of the smallest subarray in it which when sorted will sort the whole array.

**Example 1:**

```
Input: [1, 2, 5, 3, 7, 10, 9, 12]
Output: 5
Explanation: We need to sort only the subarray [5, 3, 7, 10, 9] to make the whole array sor
ted
```

**Example 2:**

```
Input: [1, 3, 2, 0, -1, 7, 10]
Output: 5
Explanation: We need to sort only the subarray [1, 3, 2, 0, -1] to make the whole array sor
ted
```

**Example 3:**

```
Input: [1, 2, 3]
Output: 0
Explanation: The array is already sorted
```

**Example 4:**

```
Input: [3, 2, 1]
Output: 3
Explanation: The whole array needs to be sorted.
```

## Solution #

As we know, once an array is sorted (in ascending order), the smallest number is at the beginning and the largest number is at the end of the array. So if we start from the beginning of the array to find the first element which is out of sorting order i.e., which is smaller than its previous element, and similarly from the end of array to find the first element which is bigger than its previous element, will sorting the subarray between these two numbers result in the whole array being sorted?

Let's try to understand this with Example-2 mentioned above. In the following array, what are the first numbers out of sorting order from the beginning and the end of the array:

```
[1, 3, 2, 0, -1, 7, 10]
```

1. Starting from the beginning of the array the first number out of the sorting order is '2' as it is smaller than its previous element which is '3'.
2. Starting from the end of the array the first number out of the sorting order is '0' as it is bigger than its previous element which is '-1'

As you can see, sorting the numbers between '3' and '-1' will not sort the whole array. To see this, the following will be our original array after the sorted subarray:

```
[1, -1, 0, 2, 3, 7, 10]
```

The problem here is that the smallest number of our subarray is '-1' which dictates that we need to include more numbers from the beginning of the array to make the whole array sorted. We will have a similar problem if the maximum of the subarray is bigger than some elements at the end of the array. To sort the whole array we need to include all such elements that are smaller than the biggest element of the subarray. So our final algorithm will look like:

1. From the beginning and end of the array, find the first elements that are out of the sorting order. The two elements will be our candidate subarray.
2. Find the maximum and minimum of this subarray.
3. Extend the subarray from beginning to include any number which is bigger than the minimum of the subarray.
4. Similarly, extend the subarray from the end to include any number which is smaller than the maximum of the subarray.

Code #

Here is what our algorithm will look like:

```cpp
#include <limits>
#include <vector>

class ShortestWindowSort {
 public:
  static int sort(const vector<int>& arr) {
    int low = 0, high = arr.size() - 1;
    // find the first number out of sorting order from the beginning
    while (low < arr.size() - 1 && arr[low] <= arr[low + 1]) {
      low++;
    }

    if (low == arr.size() - 1) {  // if the array is sorted
      return 0;
    }

    // find the first number out of sorting order from the end
    while (high > 0 && arr[high] >= arr[high - 1]) {
      high--;
    }

    // find the maximum and minimum of the subarray
    int subarrayMax = numeric_limits<int>::min(), subarrayMin = numeric_limits<int>::max();
    for (int k = low; k <= high; k++) {
      subarrayMax = max(subarrayMax, arr[k]);
      subarrayMin = min(subarrayMin, arr[k]);
    }

    // extend the subarray to include any number which is bigger than the minimum of the subarray
    while (low > 0 && arr[low - 1] > subarrayMin) {
      low--;
    }
    // extend the subarray to include any number which is smaller than the maximum of the subarray
    while (high < arr.size() - 1 && arr[high + 1] < subarrayMax) {
      high++;
    }

    return high - low + 1;
  }
};

int main(int argc, char* argv[]) {
  cout << ShortestWindowSort::sort(vector<int>{1, 2, 5, 3, 7, 10, 9, 12}) << endl;
  cout << ShortestWindowSort::sort(vector<int>{1, 3, 2, 0, -1, 7, 10}) << endl;
  cout << ShortestWindowSort::sort(vector<int>{1, 2, 3}) << endl;
  cout << ShortestWindowSort::sort(vector<int>{3, 2, 1}) << endl;
}
```

### Time complexity #

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

### Space complexity #

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