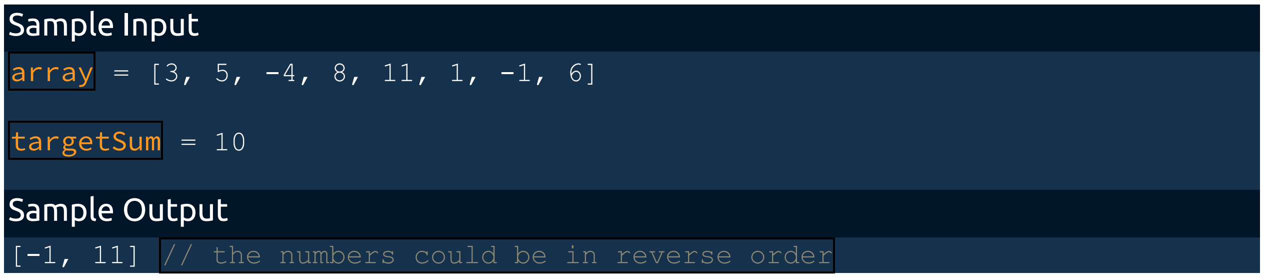
# Code Challenge #1 Two Number Sum (Easy)


Sample Input

```
array = [3, 5, -4, 8, 11, 1, -1, 6]

targetSum = 10
```

Sample Output

```
[-1, 11] // the numbers could be in reverse order
```

## Solution #1

```
function twoNumberSum(array, targetSum) {
  for (let i = 0; i < array.length - 1; i++) {
        const firstNum = array[i];
        for (let j = i + 1; j < array.length; j++) {
              const secondNum = array[j];
              if (firstNum + secondNum === targetSum) {
                    return [firstNum, secondNum];
              }
        }
        }
        return[];
 }

```

## Explanation

The first solution uses two variables as tracker variables that loop through the array. The first second tracker variable is always one item ahead of the second tracker variable. The loop is a dual loop where the second loop starts as the first loop iterates through each item in the array. If there is a match for the target sum then it will return the match. If there is no match then it will return an empty array once the dual loops finish. The code efficiency is 0(n^2) due to the dual loops.

## Code Pattern: Dual Loop

```
for (let i = 0; i < array.length - 1; i++) {
 const firstNum = array[i];
  for ( let j = i + 1; j < array.length; j++) {
   const secondNum = array[j];
  }
}

```

**Solution # 2**

```
function twoNumberSum(array, targetSum) {
  const nums = {};
 for (const num of array) {
        const potentialMatch = targetSum - num;
        if (potentialMatch in nums) {
               return [potentialMatch, num];
        } else {
               nums[num] = true;
        }
        }
        return []
 }


```

**Explanation**

The second solution used a hash/object called nums in order to keep track of numbers from the array. There is only one loop that uses "syntactic sugar" loop to iterate through the items of the array. We use a variable called potentialMatch to see if there is a potential match from the numbers in the nums object using simple math. The potential number match would be based on subtracting the targetSum derived from argument of the function minus a number from the array. If there is a match it will return the potential match and number as part of an array. If there isn't a match it will add the number from the array to the nums object with a value of true so that it can be checked later. If there aren't any matches at all the function will return an empty array. This solution is O(n) time complexity which is better than the first.

**Code Pattern: Hash/Object and Loop**

```
  const nums = {};
   for (const num of array) {

        } else {
               nums[num] = true;
        }
 }
 return []
}


```

**Math Pattern:**

**A + B = C**

**C – A = B or C – B = A**

**Example: 2 + 3 = 5**

**5 – 3 = 2**

**5 – 2 = 3**

**Solution # 3**

```
function twoNumberSum(array, targetSum) {
array.sort((a, b) => a - b)
let left = 0;
let right = array.length - 1;
while (left < right) {
 const currentSum = array[left] + array [right];
 if (currentSum === targetSum) {
        return [array[left], array[right]];
 } else if (currentSum < targetSum) {
               left++;
        } else if (currentSum > targetSum) {
               right--;
        }
  }
```

```
        return []
}

```

## Explanation

The third solution uses an array sort method specifically for numbers in order to put the numbers in ascending order. It utilizes two tracking variables that start at opposite ends. The first one called left starts at the first item in the array and the second one called right starts at the last item of the array. A while loop is used to create a loop that continues as long as the left value is less than the right value. We create a variable called currentSum which tracks the total values of the left and right item in the array. If it matches the targetSum from the argument of the function we will return an array of the two values. If the currentSum is less than the target sum we will increase the position of the left value by one. If the currentSum is greater than the  the targetSum we will lower the right value by one. We lower the right value if the currentSum is too big because the array is sorted. By lowering the right value we move on to a smaller number that may match the targetSum. If the currentSum is smaller than the targetSum we increase the left value because we will go to a bigger number which may help in matching the targetSum. This methodology depends on the numbers being sorting in order to work. If the targetSum isn't found we return an empty array. This solution is O(nlog(n))  time complexity because the while loop shrinks the array based on the comparison of the currentSum with the targetSum.

**Code Pattern: Array Sort and while loop with pointers.**

```
array.sort((a, b) => a - b)
let left = 0;
let right = array.length - 1;
while (left < right) {

  } else if () {
        left++;
  } else if () {
        right--;
        }
   }
        return []
}
```