Two Number Sum

array = [3, 5, -4, 8, 11, 1, -1, 6]
targetSum = 10
[-1, 11] or [11, -1]

input: non-empty array of distinct integers (no values are repeated)
an integer representing the target sum

output: an array of two numbers that sum to the target sum (in any order)
return an empty array if no two numbers sum to the target sum
Target sum to be obtained by summing two different integers (can't add to itself)

1) O(n²) time ; O(1) space

```
// Solution 1: O(n^2) time | O(1) space
function twoNumberSum(array, targetSum) {
  for(let i = 0; i < array.length - 1; i++) {
    let curr = array[i];
    for(let j = i + 1; j<array.length; j++) {
      let next = array[j];
      if (curr + next === targetSum) {
        return [curr, next];
      }
    }
  }
  return [];
}
```

Start: [3, 5, -4, 8, 11, 1, -1, 6] (i at 0, j at 1)
End: [3, 5, -4, 8, 11, 1, -1, 6] (i at 6, j at 7)

- We want i to stop at the second last index (6) so we go to array.lenghth - 1 (7)
- We want j to stop at the last index (7) so we go until array.length (8)
- We return an empty array [] if no match is found
- This is O(n²) time because we traverse the array twice. O(1) space because no additional space is needed as the input increases

2) O(n) time ; O(n) space

```
// Solution 2: O(n) time | O(n) space
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 [];
}
```

- We know targetSum = num1 + num2
- TargetSum is always known, so:
  num1 = targetSum - num2 (iterator)
  OR
  num2 = targetSum - num1 (iterator)
- We can iterate through the array and evaluate the above equation each time
- We can then store the iterator (num2) in an object.
- As we continue to iterate through the array we evaluate num1 and then check if num2 is already in our object
- If num2 doesn't already exist in our object, we add it
- If it does exist, we have found a match since targetSum = num1 + num2 (we know num2 and just found num1)
- We then return [num1, num2]
- If no match is found, return an empty array
- Note: Object is better since it has O(1) lookup

nums = {3: true, 5: true, 11: true}
[3, 5, -4, 8, 11, 1, -1, 6] (i at 6)
potentialMatch = targetSum - num

1st iteration:
- potentialMatch evaluates to 7.
- Is there a 7 in our object?
- No, so add num to our object.
- continue
- We get to index 6 (value -1)
- potentialMatch evaluates to 11.
- Is there a 11 in our object?
- Yes! so return [-1, 11]

- This method uses Hashing. Hash tables have constant time look up
- O(n) time since we are traversing the whole array
- O(n) space since we are adding to our object (Hash Table) are each index

3) O(nlogn) time ; O(1) space

```
// Solution 3: O(nlong(n)) time | O(1) space
function twoNumberSum(array, targetSum) {
  array.sort((a,b) => a - b)
  let left = 0;
  let right = array.length - 1;
  while (left < right) {
    let currentSum = array[left] + array[right];
    if (currentSum === targetSum) {
      return [array[left], array[right]];
    }
    if (currentSum < targetSum) {
      left++;
    }
    if (currentSum > targetSum) {
      right--;
    }
  }
  return [];
}
```

- If we sort the array, we will get values in increasing order
  [-4, -1, 1, 3, 5, 6, 8, 11]
- We can then point at the first and last values in the array
- We then sum their values and are left with three scenarios
  1. Sum < targetSum
  2. Sum > targetSum
  3. Sum = targetSum
- For 1, this means that we need to advance the left (smaller) pointer
- For 2, this means that we need to advance the right (larger) pointer
- If we do not reach 3, we return an empty array

- For large arrays, JavaScript uses QuickSort
- For arrays containing 10 or fewer elements, it uses Insertion Sort

Quick sort: O(nlogn) "avg" time ; O(logn) space
Insertion sort: O(n²) time ; O(1) space

- We assume we have a very large array so Quicksort is being used
- We also traverse the array fully so time complexity would be:
  T(n) = n + nlogn or O(nlogn) asymptotic notation
- Space complexity is O(1) since we are not using any new space as input size increases