Two Sum

Problem definition

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have **exactly one solution**, and you may not use the same element twice.

You can return the answer in any order.

Example cases

Example 1:

Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].

Example 2:

Input: nums = [3,2,4], target = 6
Output: [1,2]

Example 3:

Input: nums = [3,3], target = 6
Output: [0,1]

Constraints:

2 <= nums.length <= 104

-109 <= nums[i] <= 109

-109 <= target <= 109

Only one valid answer exists.

Example test cases

In [1]:
```
def test_cases():
    assert two_sum([2, 7, 11, 15], 9) == [0, 1]
    assert two_sum([3, 2, 4], 6) == [1, 2]
    assert two_sum([3, 3], 6) == [0, 1]
    print("All test cases passed!")
```

Solutions

$O(n^2)$

Brute-force solution by checking all the possible combinations.

In [2]:
```
def two_sum(nums, target):
    element_indices = range(len(nums))
    for i in element_indices:
        for j in element_indices:
            if nums[i] + nums[j] == target and (i != j):
                return [i, j]
```

In [3]:
```
test_cases()
```

```
All test cases passed!
```

$O(nlogn)$

If the array is sorted, there are some interesting solutions. Sorting is $O(nlogn)$. Then we can use two pointer approach to find the sum. The problem here arises with the fact that the indices are mixed after sorting. So, we need to remember them before sorting. Finally, get the original indices instead of the sorted ones.

In [4]:
```
def two_sum(nums, target):
    nums = [(num, i) for i, num in enumerate(nums)]
    nums.sort(key = lambda x: x[0])
    pointer1 = 0
    pointer2 = len(nums) - 1
    while not (pointer1 > pointer2):
        numbers_sum = nums[pointer1][0] + nums[pointer2][0]
        if numbers_sum == target:
            return [nums[pointer1][1], nums[pointer2][1]]
        elif numbers_sum > target:
            pointer2 = pointer2 - 1
        elif numbers_sum < target:
            pointer1 = pointer1 + 1
```

In [5]:
```
test_cases()
```

```
All test cases passed!
```

$O(n)$

If we keep the remainder (and it's index) after subtracting the number from the target, we will later search in the container if we have that remainder. If the remainder is found, we can just take the index out and use it with the index of the number corresponding to the remainder.

In [6]:
```
def two_sum(nums, target):
    remainders = {}
    for i, number in enumerate(nums):
        remainder = target-number
        if number in remainders:
            return [remainders[number], i]
        remainders[remainder] = i
```

In [7]:
```
two_sum([2, 7, 11, 15], 9)
```

Out[7]:
```
[0, 1]
```

In [8]:
```
test_cases()
```

```
All test cases passed!
```