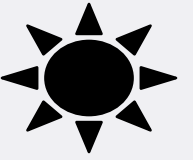


# MORE CODING PRACTICES



# PROBLEM 1 : DESCRIPTION

Given an array of integers named as num, and an integer target, return indices of two input that would add up to the target.

**Input:** nums = [2,11,7,15],  
target = 9

**Output:** [0,2]

You may assume that each input would have *exactly one solution*

You may not use the *same* element twice.

You can return the answer in any order.

Fun Fact

Amazon   87	Adobe   48	Apple   38	Google   28	Bloomberg   21	Microsoft   13
Facebook   12	Oracle   6	Accenture   6	Uber   5	Spotify   5	Expedia   5
Goldman Sachs   3	Yahoo   3	eBay   3	Cisco   3	tcs   3	Samsung   2
Yandex   2	Nagarro   2				



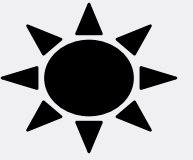
## Problem 1 : Example

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

**Output:** [1,2]

**Input:** nums = [3,3],  
target = 6

**Output:** [0,1]



# Problem 1 : Example

## Constraints:

- `2 <= nums.length <= 104`
- `-109 <= nums[i] <= 109`
- `-109 <= target <= 109`
- **Only one valid answer exists.**

You may assume that each input would have ***exactly one solution***

You may not use the *same* element twice.

You can return the answer in any order.

## HOW TO APPROACH

Brute force way would be to search for all possible pairs of numbers but that would be too slow.

$O(n^2)$

Solve it first , improve it later!

Run time with brute force will be  $O(n^2)$

Aux. space  $O(1)$

## MAKE A NOTE

- When we have a run time of  $O(n^2)$  Aux. space  $O(1)$
- You can usually fix this but we must use some space.
- Run time is always more important than space.
- **Trade space for speed!**
- We can bring our run time from quadratic to linear, but we will use space so our space will go to  $O(n)$ .

## HOW TO APPROACH

So, if we fix one of the numbers, say  $x$ ,  
we scan the entire array to find the next number  $y$   
which is:  $\text{value} - x$

where  $\text{value}$  is the input parameter.

Can we change our array somehow so that this  
search becomes faster?

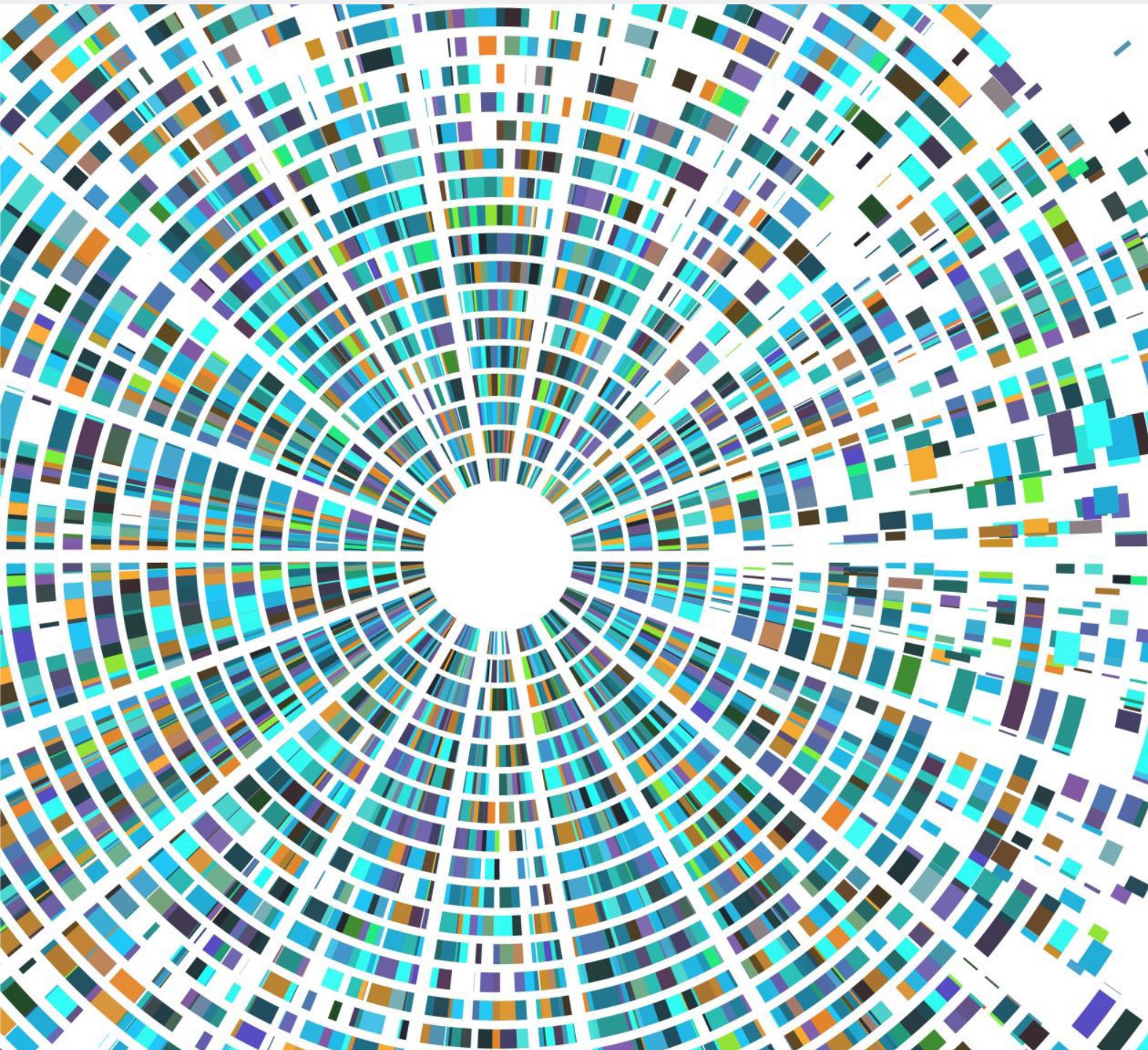
Or without changing the array, can we use additional  
space somehow? Like maybe a hash map to speed up  
the search?

# BRUTE FORCE $O(N^2)$



```
def two_sum_quadratic(self, nums: List[int], target: int) -> List[int]:  
    # Iterate over the `nums` list  
  
    # Iterate over the remaining elements in the `nums` list, starting from the next index  
  
    # If the current value and the next value add up to the target,  
        # return the current index and the next index  
        pass
```





# WE CAN IMPROVE OUR RUN TIME

From  $O(n^2)$  to  $O(n)$

## BETTER SOLUTION

To improve our runtime complexity, we need a more efficient way to check if the complement exists in the array.

**If the complement exists, we need to get its index.**

What is the best way to maintain a mapping of each element in the array to its index? Hash Map → Dictionary

We can reduce the lookup time from  $O(n)$  to  $O(1)$  by trading space for speed.

A hash table is well suited for this purpose because it supports fast lookup in *near* constant time.



## BETTER SOLUTION I USE A DICTIONARY

- First implementation uses two iterations.
  - The first iteration, we add each element's value as a key and its index as a value to the hash table.
  - The second iteration, we check if each element's complement ( $target - nums[i]$ ) exists in the hash table.
    - If it does exist, we return current element's index and its complement's index.
- Beware that the complement must not be  $nums[i]$  itself!



# BETTER SOLUTION I USE A DICTIONARY

```
def two_sum_v1(self, nums: List[int], target: int) -> List[int]:  
    # Create a dictionary to store the value and its index in the `nums` list  
  
    # Iterate over the `nums` list and store each value and its index in the dictionary  
  
    # Iterate over the `nums` list again and check if the complement (target - current value) is in the dictionary  
  
    # If the complement is in the dictionary and its index is not the current index,  
    # return the current index and the index of the complement  
    pass
```



WE CAN DO  
EVEN BETTER

## BETTER SOLUTION II

It turns out we can do it in one-pass!

While we are iterating and inserting elements into the hash table, we also look back to check if current element's complement already exists in the hash table.

If it exists, we have found a solution and return the indices immediately.

# BETTER SOLUTION II



```
def two_sum_v2(self, nums: List[int], target: int) -> List[int]:  
    # Create a dictionary to store the complement and its index in the `nums` list  
  
    # Iterate over the `nums` list  
  
        # Calculate the complement (target - current value)  
  
        # If the complement is in the dictionary, return the current index and the index of the complement  
  
        # If the complement is not in the dictionary, store the current value and its index in the dictionary  
    pass
```

## PROBLEM : DESCRIPTION

Given a **1-indexed** array of integers numbers that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific target number.

Let these two numbers be `numbers[index1]` and `numbers[index2]` where  $1 \leq \text{index}_1 < \text{index}_2 \leq \text{numbers.length}$ .



Sneaky way of saying indexing starts from 1

Return *the indices of the two numbers, index<sub>1</sub> and index<sub>2</sub>, **added by one** as an integer array [index<sub>1</sub>, index<sub>2</sub>] of length 2.*

The tests are generated such that there is **exactly one solution**.

You **may not** use the same element twice.

Your solution must use only constant extra space.



## PROBLEM : DESCRIPTION

Return *the indices of the two numbers*,  $\text{index}_1$  and  $\text{index}_2$ , ***added by one*** as an integer array  $[\text{index}_1, \text{index}_2]$  of length 2.

The tests are generated such that there is **exactly one solution**.

You **may not** use the same element twice.

Your solution must use only constant extra space.

## Problem 1 : Example

**Input:** nums = [1,3,4,5,7,10,11]

target = 9


**Output:** [3,4]

Remember the indexing starts from 1 as the specs.  
states

# REMEMBER THE ARRAY IS ALREADY SORTED

We can use this to our advantage

[1,3,4,5,7,10,11]



$$1 + 11 = 12 > 9$$

# REMEMBER THE ARRAY IS ALREADY SORTED

We can use this to our advantage

[1,3,4,5,7,10,11]

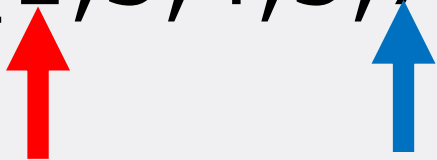


$$1 + 10 = 11 > 9$$

# REMEMBER THE ARRAY IS ALREADY SORTED

We can use this to our advantage

[1,3,4,5,7,10,11]



$$1 + 7 = 8 < 9$$

Remember we  
guaranteed a solution so  
keep looking

# REMEMBER THE ARRAY IS ALREADY SORTED

We can use this to our advantage

[1,3,4,5,7,10,11]



$$3 + 7 = 10 > 9$$

# REMEMBER THE ARRAY IS ALREADY SORTED

We can use this to our advantage

[1,3,4,5,7,10,11]

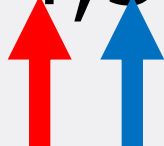


$$3 + 5 = 8 < 9$$

# REMEMBER THE ARRAY IS ALREADY SORTED

We can use this to our advantage

[1,3,4,5,7,10,11]



$4 + 5 = 9$  we found it !

Return [3,4]



# COMPLETE THE CODE



```
def two_sum_sorted(self, numbers: List[int], target: int) -> List[int]:  
    pass
```

# Time complexity analysis

If we used the Brute force like we did earlier :  $O(n^2)$  time  $O(1)$  space.

If we used the Hash Table approach as we did earlier:  $O(n)$  time  $O(n)$  space

**This approach of using left and right references:  $O(n)$  time,  $O(1)$  space**

