

Problem 1: Two Sum

Problem Information

Difficulty: [Easy](#)

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

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 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 <= 10`

4

-10

9

`<= nums[i] <= 10`

9

-10

9

`<= target <= 10`

9

Only one valid answer exists.

Follow-up:

Can you come up with an algorithm that is less than

$O(n$

2

)

time complexity?

Code Snippets

C++:

```

class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {

    }

};

```

Java:

```

class Solution {
    public int[] twoSum(int[] nums, int target) {

    }

}

```

Python3:

```

class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:

```

Python:

```

class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """

```

JavaScript:

```

/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function(nums, target) {

};

```

TypeScript:

```
function twoSum(nums: number[], target: number): number[] {  
  
};
```

C#:

```
public class Solution {  
    public int[] TwoSum(int[] nums, int target) {  
  
    }  
}
```

C:

```
/**  
 * Note: The returned array must be malloced, assume caller calls free().  
 */  
int* twoSum(int* nums, int numsSize, int target, int* returnSize) {  
  
}
```

Go:

```
func twoSum(nums []int, target int) []int {  
  
}
```

Kotlin:

```
class Solution {  
    fun twoSum(nums: IntArray, target: Int): IntArray {  
  
    }  
}
```

Swift:

```
class Solution {  
    func twoSum(_ nums: [Int], _ target: Int) -> [Int] {  
  
    }  
}
```

Rust:

```
impl Solution {  
    pub fn two_sum(nums: Vec<i32>, target: i32) -> Vec<i32> {  
  
    }  
}
```

Ruby:

```
# @param {Integer[]} nums  
# @param {Integer} target  
# @return {Integer[]}  
def two_sum(nums, target)  
  
end
```

PHP:

```
class Solution {  
  
    /**  
     * @param Integer[] $nums  
     * @param Integer $target  
     * @return Integer[]  
     */  
    function twoSum($nums, $target) {  
  
    }  
}
```

Dart:

```
class Solution {  
    List<int> twoSum(List<int> nums, int target) {  
  
    }  
}
```

Scala:

```
object Solution {  
    def twoSum(nums: Array[Int], target: Int): Array[Int] = {  
  
    }  
}
```

```
}  
}
```

Elixir:

```
defmodule Solution do  
  @spec two_sum(nums :: [integer], target :: integer) :: [integer]  
  def two_sum(nums, target) do  
  
  end  
end
```

Erlang:

```
-spec two_sum(Nums :: [integer()], Target :: integer()) -> [integer()].  
two_sum(Nums, Target) ->  
.
```

Racket:

```
(define/contract (two-sum nums target)  
  (-> (listof exact-integer?) exact-integer? (listof exact-integer?))  
  )
```

Solutions

C++ Solution:

```
/*  
 * Problem: Two Sum  
 * Difficulty: Easy  
 * Tags: array, hash  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(n) for hash map  
 */  
  
class Solution {
```

```

public:
vector<int> twoSum(vector<int>& nums, int target) {

}

};

```

Java Solution:

```

/**
 * Problem: Two Sum
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

class Solution {
public int[] twoSum(int[] nums, int target) {

}

}

```

Python3 Solution:

```

"""
Problem: Two Sum
Difficulty: Easy
Tags: array, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def twoSum(self, nums: List[int], target: int) -> List[int]:
# TODO: Implement optimized solution
pass

```

Python Solution:

```
class Solution(object):
    def twoSum(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: List[int]
        """
```

JavaScript Solution:

```
/**
 * Problem: Two Sum
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */

/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number[]}
 */
var twoSum = function(nums, target) {

};
```

TypeScript Solution:

```
/**
 * Problem: Two Sum
 * Difficulty: Easy
 * Tags: array, hash
 *
 * Approach: Use two pointers or sliding window technique
 * Time Complexity: O(n) or O(n log n)
 * Space Complexity: O(n) for hash map
 */
```

```
function twoSum(nums: number[], target: number): number[] {  
  
};
```

C# Solution:

```
/*  
 * Problem: Two Sum  
 * Difficulty: Easy  
 * Tags: array, hash  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(n) for hash map  
 */  
  
public class Solution {  
    public int[] TwoSum(int[] nums, int target) {  
  
    }  
}
```

C Solution:

```
/*  
 * Problem: Two Sum  
 * Difficulty: Easy  
 * Tags: array, hash  
 *  
 * Approach: Use two pointers or sliding window technique  
 * Time Complexity: O(n) or O(n log n)  
 * Space Complexity: O(n) for hash map  
 */  
  
/**  
 * Note: The returned array must be malloced, assume caller calls free().  
 */  
int* twoSum(int* nums, int numsSize, int target, int* returnSize) {  
  
}
```

Go Solution:

```
// Problem: Two Sum
// Difficulty: Easy
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

func twoSum(nums []int, target int) []int {

}
```

Kotlin Solution:

```
class Solution {
    fun twoSum(nums: IntArray, target: Int): IntArray {

    }
}
```

Swift Solution:

```
class Solution {
    func twoSum(_ nums: [Int], _ target: Int) -> [Int] {

    }
}
```

Rust Solution:

```
// Problem: Two Sum
// Difficulty: Easy
// Tags: array, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
    pub fn two_sum(nums: Vec<i32>, target: i32) -> Vec<i32> {
```

```
}  
}
```

Ruby Solution:

```
# @param {Integer[]} nums  
# @param {Integer} target  
# @return {Integer[]}  
def two_sum(nums, target)  
  
end
```

PHP Solution:

```
class Solution {  
  
    /**  
     * @param Integer[] $nums  
     * @param Integer $target  
     * @return Integer[]  
     */  
    function twoSum($nums, $target) {  
  
    }  
}
```

Dart Solution:

```
class Solution {  
    List<int> twoSum(List<int> nums, int target) {  
  
    }  
}
```

Scala Solution:

```
object Solution {  
    def twoSum(nums: Array[Int], target: Int): Array[Int] = {  
  
    }  
}
```

```
}
```

Elixir Solution:

```
defmodule Solution do
  @spec two_sum(nums :: [integer], target :: integer) :: [integer]
  def two_sum(nums, target) do

  end
end
```

Erlang Solution:

```
-spec two_sum(Nums :: [integer()], Target :: integer()) -> [integer()].
two_sum(Nums, Target) ->
.
```

Racket Solution:

```
(define/contract (two-sum nums target)
  (-> (listof exact-integer?) exact-integer? (listof exact-integer?))
)
```